Chevalier

Russell James Lowke

A Thesis in the Field of Information Technology

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

June 2006

Abstract

Chevalier is an online version of the historical tabletop wargame *De Bellis Magistrorum Militum* (*DBMM*), whose rules have been formed to accurately simulate ancient and medieval battles covering the pre-gunpowder period from 3000 BCE to 1500 CE. As turning points in history are often decided by combat and battles, simulations offered by *Chevalier* are ideal as an online teaching aid for history education. The battle simulations allow players to experience and experiment first hand the various tactics used by each side and thereby gain greater insight into problems faced across a varied historical range. *Chevalier* constitutes a new and innovative tool for the teaching of history.

Chevalier is available online at,

http://www.lowkemedia.com/games/as2/chevalier/Chevalier.html

Dedication

This thesis is dedicated to Philip Barker and Richard Bodly Scott, the creators of the *De Bellis* genre of historical wargames rules. It is also dedicated to the many professional and amateur historians that play *DBMM* who have helped, and continue to help, with formulating and refining the wargame rules in an attempt to create an accurate simulation of ancient and medieval historical battles.

Acknowledgments

Scott Traylor, Henry H. Leitner, William B. Robinson, Peter Gifford, Bruce Molay, Hanspeter Pfister, Tamara Bonn, Kenneth J. Basye, Yair Leviel, Jordan Bach, Billy Belfield, Christopher S. LaRoche, John Sharples, Andrew Jinks, Alister Lowke, and my parents John and Karil Lowke. In particular, I would like to acknowledge the support of Harvard University Extension School and the many resources it makes available to its students.

Table of Contents

List of Figures

Chapter 1 Introduction

In the technology driven era of today, with computers, mobile phones, TV and Internet browsing all competing for a student's time, traditional academic history, with its emphasis on names and dates, is proving to be unpalatable for the majority of high school students. Enrollments in traditional history subjects are declining and general knowledge of history is waning, to the detriment of our society.

In one of the first attempts to assess history education among American seventeen-year-olds, a study conducted by the National Assessment of Educational Progress (NAEP) and funded by the National Endowment for the Humanities (NEH) found that 11th grade high school "students could correctly answer only fifty-four percent of the questions," and that "if a national report card were given based on the achievement of 11th grade students in American history and literature, the nation would have earned failing grades" (Ravitch, 1987). One year later, the Bradley Commission for History in Schools similarly reported that "fifteen percent of students do not take American history in high school and that more than fifty percent of them study neither Western civilization nor world history" (Bradley Commission, 1988).

The traditional history curriculum, which stresses learning from a textbook and use of a chronological narrative in the classroom, is a limited approach. It results all too frequently in students rote learning lists of factual detail whose relevance is oftentimes poorly understood. Rodney M. White, in his *An Alternative Approach to Teaching History,* describes the history student as "a passive receiver of more information than one could ever hope to comprehend, analyze, and encode" (White, 1994). The situation is exasperated by history textbooks that endorse review of material and constantly ask

students to recall information. They seem to encourage the belief that history is comprised of facts to be learned and memorized, while seldom requiring students to write creatively (Sellers, 1993).

Due to the high demand for comprehensive history coverage in American schools, the textbooks employed tend to be deficient in themes and depth, often including generalized conclusions rather than debating historical issues (Simmonds, 1989). Dull textbooks result from pressures on educational publishers to avoid offending powerful groups, leading to textbooks omitting any real analysis of American society (Gaddis, 1990). Moreover, studies conducted by Marinka Bliss Hervey using the Degrees of Reading Power test, PA-form, and Likert-type scale test found that only thirty percent of students are able to easily read their textbooks (Yarema, 2002). Frequently those textbooks are relied upon to provide detail in order to save class time (Tate, 1986).

The challenge, according to Ronald W. Evans, is not more history, but rather to make history "real, vital, and meaningful to our students" (Evans, 1989). There is emerging a strong debate that history education requires less content and more depth. Too often, students view history as "just one-damn-thing-after-another" (Yarema, 2002). By covering less substance, it is maintained that students can more closely examine historical issues and learn that the study of history transcends simple memory learning. It has been found that, paradoxically, teachers who try to cover an excessive amount achieve limited results (Hampel, 1985). New technologies are raising new debates about how to best interest students and to teach history. Allan E. Yarema in his paper "A Decade of Debate: Improving Content and Interest in History education" concludes that "it is the student, finally, who educates himself or herself, and many students must be lured into even trying" (Yarema, 2002).

What is evident is that new teaching methodologies are essential to bring history as a subject back to acceptable levels. With the increasing availability of computers in most schools today, previously unimagined methods of learning are becoming possible by fusing the interactive aspects of games with the knowledge content of books.

Although research has indicated that there is little difference in student performance when game simulations are used over conventional instruction (Dekkers, 1981), students do report more interest in game simulation activities (Randel, 1992) and game simulation as an instructional strategy seems to be more effective than the lecture method for attitude formation and yielding higher levels of continuing motivation (Malouf, 1988). Furthermore, some studies indicate that students using game simulations show greater retention of material over time than those using conventional instruction (Randel, 1992), and reduced training time and instructor load (Allen, 1982). As subject interest and enthusiasm has proven such an issue with history education, as evidenced by Yarema's paper, it is evident that the motivational benefits of game simulation is a highly appealing solution. Games are starting to prove themselves a useful device for the teaching of historical concepts (Mork, 1979), and it is now becoming increasingly clear that computer game technologies which engage students as active learners can greatly assist teachers to convey specific complex intellectual concepts (Dempsey, 2002).

My thesis project, *Chevalier*, is an online implementation of the historical tabletop wargame *De Bellis Magistrorum Militum* (Barker, 2006), whose rules have been formulated to accurately simulate ancient and medieval battles covering the pre-gunpowder period from 3000 BCE to 1500 CE. As turning points in history are often decided by combat and battles, simulations offered by *Chevalier* are ideal as an online teaching aid for history education, allowing players to experience and experiment with

first hand the various battle tactics used by each side. *Chevalier* is intended to be used as a module to augment an educational Website specific to a historical period, allowing the site to simulate and replay key historical battles it discusses. It is proposed through further developments of *Chevalier* in conjunction with a historical Website, it could become a new and innovative teaching aid for use in schools in the teaching of History.

By reading variables and graphics defined in external XML and .swf files on a server, *Chevalier* simulates three important historical battles. These battles demonstrate *Chevalier's* potentially wide and varied historical range; from the conquests of Alexander against the Persians, specifically, at Gaugamela on October 1st, 331 BCE; to the downfall of the French knights against English longbow men at Agincourt, on October 25th, 1415 CE, during the 100 Years' War; to Richard Coeur de Lion's successful march along the Mediterranean coast to Arsuf while under constant attack by Saladin, on September 7th, 1191 CE, during the 3rd Crusade. *Chevalier* introduces a new form of history involving the student in real history, placing in the hands of the student key battles of the ancient and medieval world. The result of these battles, in many cases, still affect us and have relevance today.

No attempt has been made to develop an artificial computer player, due to the very high game complexity and strategies implicit in *DBMM*. The joy of the game exists in playing other people. The objective is instead to introduce added game value and playability by way of the computer, as it calculates the complex game rules, and facilitates games by being an easily available and playable system online. *Chevalier* runs from a variety of Internet browsers, utilizing the Adobe *Flash 8* plug-in, which can be found at, www.macromedia.com/go/getflashplayer.

Chevalier is available online at www.mocaz.com/games/Chevalier.html.

Chapter 2 Role of Technology in History Education

With the increasing availability of computers in almost all schools today, previously unimagined methods of instruction and learning that fuse the interactive aspects of games with the knowledge content of books and music are now possible. *Power On: New Tools for Teaching and Learning* reports that "individuals recall 30% of what they hear, 50% of what they see, almost 80% of what they see and hear, and over 90% of what they see, hear, and interact with" (Office of Technology Assessment, 1988). It is increasingly clear that technologies which engage students as active learners can greatly assist teachers to convey complex intellectual concepts.

Computers have become increasingly common in schools, as has access to the World Wide Web. Now, however, quality software as an online interactive service that complements traditional instruction is the limiting factor in fully exploiting the presence of computers. For example, a Presidential panel recommends that schools implement the following (Panel on Educational Technology, 1997):

> 1) *Focus on learning with technology, not about technology...* it is important to distinguish between technology as a subject area and the use of technology to facilitate learning about any subject area... it is important that technology be integrated throughout the K-12 curriculum, and not simply used to impart technology-related knowledge and skills.
> 2) *Emphasize content and pedagogy, not just hardware...* the development and utilization of useful educational software and information resources, and the adaptation of curricula to make effective use of technology, are likely to represent more formidable challenges [than acquiring modern computing and networking hardware.]
> 3) *Give special attention to professional development.* The substantial investment in hardware, infrastructure, software and content that is recommended in this report will largely be wasted if K-12 teachers are not provided with the preparation and support they will need to effectively integrate information technologies into their teaching.

Similarly, the National Endowment for the Humanities also reports similar goals for promoting humanities education in the digital age (NEH, 1997): "(1) Preserve and create high quality educational content; (2) Identify and disseminate high quality educational content; (3) Empower teachers to take full advantage of new technologies."

Chevalier is in fact a subset of a larger online project proposed to teach the history of the era of the Crusades 1095 - 1295 CE. This larger project, titled *The Crescent & the Cross,* had been envisioned to fully embrace the goals of the NEH by using interactive technology to facilitate both teaching and learning. Unlike traditional passive media like TV or print, interactive media allows decision-making by the participant, allowing self-paced learning through hands on experimentation and role-playing.

The Crescent & the Cross focused on the period of the Crusades which was a pivotal period of history that formed ethnic antagonisms which still influence modern Middle East politics. It is largely during this period that strong ethnic antagonisms were developed between Muslims, Christians, and Jews, who previously had lived largely in mutual toleration (Bulliet, 1979).

Surprisingly, however, the era of the Crusades is largely overlooked in world history classrooms and, thus, in the public consciousness. Were an instructor, recognizing the importance of the Crusades, to attempt to expand instruction on this subject in his or her classroom, he would find that support materials accessible to secondary education are very limited in scope, depth and overall appeal. *The Crescent & the Cross* proposed to bridge this gap, uniting the Higgins Armory Museum, with the largest collection of armor and weapons in the Western hemisphere, together with Medieval scholars at Harvard University, to create a widely available historical resource designed to explain the Crusades in an engaging, accurate, and culturally diverse manner.

The importance of history education covering the period of the Crusades cannot be overestimated. To this day, fundamentalist Muslims do not consider the Crusades as an episode in history; the attitudes of the Muslim world towards the West are still influenced by events that occurred seven centuries ago. Arab political and religious

dialogue frequently refers to Saladin, the fall of Jerusalem, and Jerusalem's recapture. Israel is regarded as a reborn Crusader state and the struggle between "Jerusalem and Damascus" continues. Former president Nasser of Egypt has been regularly compared to Saladin, who, like him, had united Syria and Egypt, and Saddam Hussein is known to have frequently placed himself beside Saladin in political propaganda. The Turkish gunman who fired on the Pope on May 13, 1981, expressed himself in these terms: "I have decided to kill John Paul II, supreme commander of the Crusades" (Lewis, 2003).

Osama bin Laden, on the October 7th videotape aired a month after the 9/11 attack, spoke dramatically of the "humiliation and disgrace" Islam had suffered for "more than eighty years." Bin Laden's Islamic audience would immediately have recognized the reference to the extinction of the Ottoman empire and the abolishment of the caliphate in 1922, a caliphate which represented for Muslims a potent symbol of Islamic unity and piety connected to the Ayyubid sultanate of Saladin, and back further still to the death of the Prophet Muhammad, the founder of Islam (Lewis, 2001).

In contrast, the Crusades have had a more uneven impact on the Western world. On the one hand, the concepts of "pilgrimage", "knighthood", "chivalry", "righteousness", "holy war" and indeed the word "Crusade" itself are a reflection of this time. President George W. Bush discovered this on September 16, 2001 after flippantly remarking, "This Crusade, this war on terrorism is going to take a while," referring to the attacks on the World Trade Center only five days earlier. This comment caused an uproar, and Bush later apologized for the remark (Lyons, 2001).

In the "No Child Left Behind" program signed by George W. Bush on January 8th, 2002, and directed by the U. S. Department of Education, the American states are compulsorily required only to test for math and english (U.S. Department of Education,

2006). Subjects such as finance and history, particularly world history, are not compulsory and are on the decline, although the National Assessment of Educational Progress (NAEP) has finally scheduled their first ever assessment of world history to be given to students in the twelfth grade in 2012 (NAEP, 2006). The assessment framework, specifications, and background variables for the curriculum are currently under development and, hopefully, will cover the period of the Crusades and the expansion of Islam, they being highly pertinent to the future of the United States and its involvement with the Middle East, as can be seen with the escalating situation between Iran and the United States as speculated by Seymour Hersh in the recent April 17th edition of the *New Yorker* article titled "The Iran Plans" (Hersh, 2006). Islam considers the U.S. the flag bearer and product of successful Western European expansionism.

The Crusades are the first instance of Western expansion and colonialism, and the first attempt to take a military initiative far from home and carry culture and religion abroad, a formula that would later prove itself a hallmark of European culture. On the other hand, the actual events of the Crusades—one of the most crucial events of the Medieval period—remain ambiguous and obscured. At best, the Crusades are dismissed as a regrettable lapse, the product of a somewhat unfortunate outburst of religious enthusiasm. At worst, they are entirely forgotten (Asbridge, 2004).

Likewise, most Americans associate Richard the Lion Heart with the tale of Robin Hood, not with his pivotal role in the Third Crusade, where Saladin's defeat of King Richard heralded the end of the Latin kingdoms established by the Europeans in the Holy Land. Nevertheless, the Crusades were launched to support a cause that has been portrayed with equal force as the most noble and the most ignoble, and over the years men have turned to them alternately for inspiration or as an object lesson in human

corruptibility (Riley-Smith, 1995).

As turning points in history are often decided by combat and battles, a turn based strategic game acting out conflicts of the Crusades was proposed in *The Crescent & the Cross*. The game utilized changing variables to simulate the battles of Hattin, Asuf and the Siege of Antioch, allowing the student to play either side, Saracen or Crusader. *Chevalier* is the battle simulator proposed for *The Crescent & the Cross,* only made more flexible and able to encompass a greater time span and thereby be more marketable and cross-purposed to other historical projects. Instead of the battles of Hattin and the Siege of Antioch, the battles of Gaugamela and of Agincourt have been substituted, as they are two of the most crucial battles in history and they provide a much wider historical scope, displaying *Chevalier's* potential to be used across a variety of different periods.

Fifty percent of all Americans play computer games (ESA, 2006),[1] and computer games can be a very effective educational tool. The *Chevalier* simulations allow users to experience and experiment first hand with the various tactics each side used. Once these simulations are incorporated into an educational environment targeting a specific period, links can be added allowing close-up inspection of terrain locations and personnel, and features added such as a comparison and contrast summary of the differences between what actually happened and the choices made by the student. A playback and step through sequence of historical events versus the players' chosen simulated events is highly feasible. The development of the *Chevalier* battle simulator is the first step, perhaps the most complicated step, in making pertinent educational online history resources such as *The Crescent & the Cross* a reality.

[1] According to the Entertainment Software Association, in 2004, strategy games were the largest selling genre of computer games, comprising 26.9% of sales. For more information see www.theesa.com.

Chapter 3 Why *DBMM*?

De Bellis Magistrorum Militum (*DBMM*) is a table top miniatures wargame under development by Philip Barker of Wargames Research Group (*WRG*) in the United Kingdom (Barker, 2006). WRG was founded in 1969 and, in contrast to most other gaming companies, concentrates specifically on creating rules based on and justified by historical research to give believable results. As a result, *WRG* have been the dominant publisher of miniatures rules for ancient era wargames since the 1970s (Allen, 1999). *WRG* is associated with the *British Historical Games Society* (see www.bhgs.co.uk), and the *Society of Ancients* (see www.soa.org.uk). The second is an international group dedicated to the study of military history of the period 3000 BCE to 1500 CE. Many of the members of both of these groups have contributed to WRG rules. Mr. Barker is a founding member of WRG and has been involved with the development of all seven editions of the original WRG wargames rules for ancient battle, and was pivotal in the revolutionary jump to the newer *De Bellis Multitudinis* (*DBM*) system released in the early early nineties, including its smaller predecessor *De Bellis Antiquitatis* (*DBA*).

DBMM is intended to be a radical new development of the *DBM* system, although retaining the main structures and procedures and much of the basic data. It continues the ongoing pursuit of creating a wargaming simulation that is an ever closer approximation of the actual dynamics of pre-gunpowder military battle. This latest edition has a focus of simulating command and control more realistically, and in particular emphasizing the Commander in Chief's (C-in-C's) plan. It is from this last aspect that the rules gets their title, which translates as "For the Wars of the Masters of Soldiers." *DBMM* is not officially released yet but the most current version of the rules may be found at,

www.phil-barker.pwp.blueyonder.co.uk/DBMM.doc. There is also a Yahoo Groups list dedicated to play testing called "DBMMlist" at http://games.groups.yahoo.com/group/DBMMlist and a Yahoo Groups list to help facilitate the production of well-researched army lists called "Tabulae Novae Exercituum" at http://games.groups.yahoo.com/group/Tabulae_Novae_Exercituum.

The "*DB*" genre rules have been very popular since their release in the early 1990s with each revision producing a more satisfying result. There are people playing *DBM* in most English speaking countries and in France, Spain, Italy, Sweden, Finland, and Germany. There are people playing both informally in their homes and in tournaments organized by local, national, and international groups. In particular, the rule set attracts strong tournament play, much like the game of *Chess*. Amateur historians and wargamers playing the *DBM* system enjoy pitting their favorite armies of ancient and medieval history against each other using a points scheme outlined in the rules. But it is not just army selection that wins victories on the tabletop, and experienced players enjoy showing off their skill by winning victories with unusual armies that were initially branded as ridiculous or hopeless and unlikely to triumph.

The simpler *DBA* system struck a positive note with wargame hobbyists as soon as it was released. The tournament aspect of the game, coupled with the simplicity and brevity of the *DBA* rules, coupled with the fact that *DBA* armies are smaller and therefore much less labor intensive to paint, generated a strong following, even though *DBA* does not carry nearly the realism of its *DBM* or *DBMM* counterparts. Mr. Barker revised the *DBA* system relatively recently, producing a version 2.2 of *DBA* that was released in January of 2004. This edition has proven popular, so popular that it has caught the attention of Games Workshop. Games Workshop's Website boasts them to be "the

largest and the most successful tabletop fantasy and futuristic battle-games company in the world" (Games Workshop, 2006). And that they spearhead "the millions of gamers aged 12 upwards, who spend many of their waking hours collecting, creating, painting, and building up the armies that they will go on to command on a carefully prepared tabletop battlefield" (Games Workshop, 2006).

Games Workshop are at the heart of the fantasy gaming miniatures hobby, as proven by their success. They are a large publicly quoted company with direct sales operations in the UK, the United States, Canada, France, Germany, Spain, and Australia. They have a chain of more than 250 Hobby Centers and sell into more than 3,500 independent toy and hobby shops around the world. There is even a Games Workshop store on Dunster St, across the road from Harvard University.[2] Games Workshop, somewhat unexpectedly, announced the acquisition of and rights to *DBA,* signaling their intention to finally branch into historical wargames. Games Workshop is the same company that makes and distributes the very popular *Lord of the Rings* and *Warhammer* miniatures rule set and figures. Their announcement of *DBA* is as follows (Games Workshop News, 2006),

> (1 April 2006). Games Workshop is pleased to announce the acquisition of the popular *De Bellis Antiquitatis* or *DBA* miniature wargaming rules from the Wargames Research Group. The acquisition marks a strategic foray by Games Workshop into the historical miniature tabletop demographic. Company spokesman Paul Ruddernick noted: "the generation of gamers who cut their teeth on our *Warhammer* and *Lord of the Rings* fantasy and Sci-Fi games is growing up. As their gaming interests mature, our market research shows a natural progression from fantasy to historical gaming. We found the famous *DBA* system has strong player support around the world and provides a natural entry point for Game Workshop to stake out its place in the historical battle gaming market. The rules are easy to master and the variety of armies that can be fielded will appeal."

[2] Games Workshop in Harvard Square, 11 Dunster St. Cambridge, Massachusetts 02138.

How the Games Workshop acquisition of *DBA* affects the potential of *Chevalier* is uncertain. It will at certainly generate more interest in the area of historical wargaming and simulation stimulating the industry. As Games Workshop pour money into and promote *DBA,* and their new range of historical figures become available, some of their audience will no doubt yearn for more ambitious and more accurate systems and hopefully turn to *DBM* or *DBMM*. The problem is that *DBA* is a restricted system that utilizes strictly a dozen elements each side. Other than generating interest in a historical period, *DBAs* usefulness and scope as a tool for teaching history is very limited.

Worth noting is that there is also a *De Bellis Renationis* (*DBR*) rule set by Philip Barker which covers the Renaissance period from 1492 to 1700. This is the period immediately after the period covered by *DBMM*. The *DBR* rule set has strong similarities to *DBMM* and is likewise a good candidate for implementation within the *Chevalier* environment. Indeed, *DBR* might be better suited to the *Chevalier* system as troops during the period tended to move in smaller groups, each of about four Elements wide—which is perfect for *Chevalier*'s movement.

The *DBMM* rule set represents the fourth revised version (4.0) and most complicated edition of the *DB* genera of rules. Of course, *DBMM* being a complicated set of rules lends them perfectly to computerization rather than the tabletop, where the players memories are relied upon for accurate results. As the main structures and procedures and much of the basic data is similar across all of the *DB* rule sets it is feasible that *Chevalier*, by abstracting away the game rules from the main body of the program, could create any of the *DB* systems (such as *DBA*, *DBM*, *DBR*, *DBMM*) while operating under the same *Chevalier* foundation. For more information on this abstraction see "Chapter 6 Application Design—Rules Objects" page 51.

Anomalies of *DBMM*

One of the frequent complaints about the playability of the *DBM* game is the "fiddlyness" of the movement system and the fact that, technically, precise differences in movement as measured to the millimeter on the table top can make a significant difference to player position and winning and losing. This problem is of course exasperated by occasional and inevitable bumps and jolts to the table and the clumsiness of fingers moving intricate bases of 15 mm figures in groups. During games I have found I would sometimes accidentally and most embarrassingly snag my sweater on the troops delicate spears, dragging the models from the table in the most highly upsetting fashion. I have learnt not to wear sweaters while playing.

Such movement issues are not usually problematic in friendly games or for the recreation of historical scenarios by amateur historians, but *DBM* is often played in highly competitive tournament environments with an umpire presiding over the game. Such umpires are often committing their time to the tournament for free to ensure its smooth running and even they are unwilling to make rulings over movement. This can be seen in the "Tournament Procedures" document for the Canberra Convention ("CanCon") DBM 2005 tournament, the largest and most recognized game convention in Australia. The document states explicitly that (SAAW, 2005),

> Players must not expect Umpires to rule on matters of measurement, or to make rulings if elements have been moved, without marking. It is the responsibility of the moving player to resolve such issues before moving elements. The benefit of any doubt will be given to the non-bounding player.

The "non-bounding player" referring to the player who is not having their turn, otherwise referred to as a "bound." So, whoever is not doing the moving gets the final word on movement issues. The newer rule set *DBMM* allows the moving player

additional movement when bringing elements into contact with the enemy, thus avoiding frequent squabbles over who can and who cannot make it into contact that turn. This extra movement is justified by *DBMM* by claiming that it accounts for additional impetus of troops as they charge into battle.

One of my primary motives for programming *Chevalier* was to overcome the movement issues by having the computer "snap" troop elements to a predefined grid. This allows for a much cleaner interaction between elements rather than the player being caught up in millimeter differences in distances and angles. For example, a common occurrence during table top games is for each player to set their army elements up directly opposite each other with bases aligned. Throughout the course of the game the table inevitably gets bumped and pieces are inadvertently shifted by players as they move them. The result is that elements that started aligned opposite each other, and should still be aligned so, are no longer. Crafty and pedantic players in tournament settings will often to use such accidental shifting to gain an advantage in competitive play. Barker has himself stated that there is always a player who is "the menace who insists on measuring everything repeatedly until he has succeeded in prodding your element to where he wants it to deal with" (Barker, 2006). Generalizing movement to a grid effectively removes such problems, freeing up game play and allowing players to concentrate on issues of strategy, rather than getting embroiled in technicalities of movement. For details on how this grid works see "Chapter 5 Game Design—The Grid" page 27.

Another issue with *DBMM* is that the rules are targeted at wargamers and amateur military historians rather than the broad and general student audience. As such, there are various terms that are likely to confuse the average person. In *Chevalier,* I have changed those terms in an attempt to reach the broader audience. The terminology changes are:

"Psiloi" of *DBMM* has become "Skirmisher" in *Chevalier.* Psiloi ("cy - loi") is a rather obscure term used by the ancient Greeks to refer to light skirmishing infantry. Although a term familiar to ancient wargamers it completely befuddles the non-indoctrinated player and is inappropriate across periods. The *DBMM* definition of Psiloi states them to be troops "including all dispersed skirmishers on foot shooting individually with javelin, sling, staff sling, bow, crossbow or hand gun" (Barker, 2006). So, the *Chevalier* use of the term "Skirmisher" is an appropriate one. The term "Psiloi "is used in the *DBMM* rules rather than "Skirmisher" as "Skirmisher" is taken, it being used (infrequently) in a broader sense to include Light Horse troops.

"Auxilia" of *DBMM* has become "Light Infantry" in *Chevalier.* This was done as "Light Infantry" is a more digestible term then "Auxilia," which is from the Latin term "Auxiliarius" meaning "assistants" and referring to the non-legionary parts of the Roman army (Bédoyère, 1999).

"Irregular" of *DBMM* has become "Clumsy" in *Chevalier.* Irregular as used in the military context specifically as a term denoting troops "not belonging to regular or established army units" (Oxford American Dictionaries, 2006). Their counterpart, "Regulars," being typically enlisted troops under officers appointed by the government and highly practiced in maneuver and combat techniques. Conversely, irregular troops typically join the army with acquaintances under local or tribal leaders, and are less accustomed to obeying formal orders. As such, irregulars are more unwieldy and noticeably more "Clumsy" on the battlefield. Since play testing established that "Irregular" is not an immediately recognized term by its military definition to the broader audience, I have used instead the term "Clumsy," as it conveys meaning immediately understandable to users and directly applicable to game play.

"War Band" of *DBMM* has become "Warriors" in *Chevalier* and "Blades" of *DBMM* has become "Swords" in *Chevalier.* This was done to keep consistency with other troop type names, such as Spears, Pikes, Skirmishers, Knights and Hordes.

Also, in *Chevalier,* the troop types of Archers and Crossbows have been differentiated from each other rather than being lumped together in a single class called "bows" as in *DBMM.* During testing it was found that having crossbows generalized and looking like bows appeared as an error to most players. Visually identifying them as Crossbows and changing the name became useful, especially as the most common type of bowman in *Chevalier* are longbows, which are graded as superior over crossbows, while crossbows are graded as ordinary. Differences of grade aside, *Chevalier*, like *DBMM*, treats both Archers and Crossbows as the same type.

One other significant change to the *DB* system is that in *Chevalier* all infantry (except Horde) are depicted on the same base size, that being the thinnest base with a depth of three blocks. In the *DB* system heavy infantry are based on a thinner base than other infantry, an equivalent heavy infantry base in *Chevalier* would be two blocks deep. There are three good reasons why such a base is not used. Firstly, on a thinner base the heavy infantry do not appear to the player as a more compact and more solid a formation as they do on the tabletop, instead, they appear on-screen as thinner and less significant than their lighter infantry counterparts; Secondly, a heavy infantry base two blocks deep is so thin that it does not accommodate enough space for the visual icon denoting the unit type; Thirdly, the grid system used by *Chevalier* allowing Element playing pieces to snap to the playing grid regardless of orientation only works for three base depth sizes. For more information on the grid design and base sizes see "Chapter 5 Game Design—The Grid" page 27.

Comparisons with other systems, *DBA* online

In comparison to other approaches taken to simulate the *DB* system online it should be noted that there is an online version of the simplest version *DBA*. This incarnation may be found at www.dbaol.com. It is still played frequently today by many wargame enthusiasts, although it currently implements the outdated version 1.1 of the *DBA* rules. It stands as evidence of the strength of the existing market for the *DB* game system, and its appropriateness as an online wargame. *DBA Online* cannot be played straight out of a browser, instead it works as an application that needs to be downloaded locally to the players machine. The *DBA Online* system is Windows only, and boasts a very clumsy game-play interface.

Chapter 4 Why the Flash Platform?

To develop a satisfying and attention catching game experience the *Chevalier* design demands a rich combination of interactivity, text control, vector graphics, raster graphics, animation, and sound, which all need to be brought together seamlessly under one easily accessible environment. Adobe Systems' *Flash 8* is such an environment.

Flash is the leading multimedia authoring platform used to create rich media content which can be viewed using *Flash Player*, a client application "runtime" available for most Web browsers. Flash content features support for vector and raster graphics, a scripting language called *ActionScript* and bidirectional streaming of audio and video. Since its introduction in 1996, Flash technology has become a popular method for adding animation and interactivity to Web pages and is commonly used to create animations and advertisements, to design Web-page elements, and to add video to Websites. More recently, with the implementation of *ActionScript 2,* the Flash environment has matured enough to be considered an object-oriented development platform and can be used to build rich Internet applications. *Chevalier* was written in *Flash 8* as Flash is now the world's most pervasive online software platform.

In September 2005, NPD Research,[3] conducted a study to determine the penetration of *Flash Player* on Web browsers. Their study concluded that 97.7% of Internet-enabled desktops in the US had a version of *Flash Player* installed, 93.5% of which were running version 7 of *Flash Player*, while a stunning 45.2% were already using *Flash Player* version 8. *Flash 8* had only been released the month previous to the survey. The sampling error for the NPD study is believed to be +/- 2% at the 95% confidence level (NPD Research, 2005). According to the International Data Corporation,

[3] For information on NPD Research see www.npd.com/proprietary.html

the forecasted number of personal computers using the Internet in September of 2005 was 663 million, making Flash penetration 645 million computers (IDC, 2005). What's more, there is also claimed an accelerated adoption curve for *Flash 8*, which Adobe anticipates to see on 80% of all desktops sometime in June 2006 (Mack, 2006).

Furthermore, not only does the *Flash Player* facilitate consistent playback over an impressive range of machines, it reaches the largest audience possible by utilizing a wide variety of browsers. On both the Windows and Macintosh platforms, *Firefox*, *Internet Explorer*, *Netscape*, *Mozilla*, *Opera*, and *AOL* browsers are all supported, and in the languages of English, French, German, Japanese, Italian, Korean, Spanish, Simplified Chinese and Traditional Chinese. The Macintosh platform also additionally supports Apple's *Safari* browser. On the Linux platform both the *Netscape* and *Mozilla* browsers support *Flash Player 7*, and the Solaris platform supports *Flash Player 7* under the *Mozilla* browser. Adobe is aware that a *Flash 8* solution is needed for the Linux and Solaris platforms, although a player for these platforms is not anticipated until the release of *Flash 8.5* or beyond.

Advantages of ActionScript

With the inclusion of *Flash ActonScript 2.0* into *Flash 7 (MX 2004)*, Flash is finally equipped to handle complex object-oriented programming (OOP). Due to the wide distribution and availability of the Adobe Flash plug-in, and the advancement in product stability and the *Flashcom* server, the Flash platform is quite simply the most flexible and widely accessible environment today for consistent playback and development of online applications. Also, as Flash was originally an animation package, it inherently gives the developer access to an enormous amount of graphic support which might otherwise be

extremely tedious to implement. Flash is essentially an enormous cross platform pre-compiled graphics library, and more. *Flash ActionScript 2.0* syntax mimics *JavaScript*, and is sufficiently close to regular *C* syntax to allow easy translation of code from *C* style environments. For instance, the generic Animatem, Matrix, Grid, Point2D and Rect objects used in *Chevalier* were all ported from *C++* code that was originally written for other *C++* projects. For more information on this see "Chapter 6 Application Design—Presentation Objects" page 54.

Furthermore, the target audience of *Chevalier* is not the audience of today but rather the audience of a year or more's time. As the Flash platform is continuously being enhanced by Adobe, it being one of the darlings of their product range, *Chevalier* will directly benefit from such development, placing it on the "bleeding edge" of Internet technology. As Flash is constantly upgraded and maintained, *Chevalier* will also be able to keep functionality and longevity because Adobe can be relied upon to do the development work and to maintain compatibility with new operating systems and computer environments. For example, the same was true with the *Director* environment (owned by Adobe, formally Macromedia), where projects I've written under *Director 4.1* in 1994 can be converted to the current version of *Director 10.1*, while only suffering very minor bugs. Such converted projects are breaching a period of over ten years of significant technological computer advancements and are even granted access to whole new operating systems that did not exist at the time of their writing, such as Macintosh OS X. Such compatibility and longevity of ten or more years is extremely good in terms of a project's life span and better even than the life span of some traditional programming languages.

In particular, Adobe is soon anticipating the release of *ActionScript 3*, due with the next release of Flash, version 8.5. *ActionScript 3* is a complete rewrite of ActionScript, with a new and highly-optimized *ActionScript Virtual Machine* (*AVM2*) which dramatically exceeds the performance of the original virtual machine. Reports are of *ActionScript 3* code executing at up to ten times faster than legacy *ActionScript 2* code (Grossman, 2006). *ActionScript 3* is a dialect of *ECMAScript* which formalizes the features of *ActionScript 2*, adding the capabilities of *ECMAScript* for XML (E4X) which transforms XML into a native data type, dramatically simplifying XML processing (Grossman, 2006). *Chevalier* is ideally placed to benefit from all of Adobe's development enhancements of *ActionScript 3*. It will be very interesting to see how *Chevalier* performs under the next 8.5 release of Adobe *Flash*.

Anomalies of Flash

There are various problematic issues and beneficial conveniences with *ActionScript 2* as it is implemented in *Flash 8.* In particular, the platform suffers heavily from the problem that many runtime errors fail in a "graceful but silent fashion" (Grossman, 2006). Although this means that *ActionScript* executes through bugs without some inexplicable dialog box appearing, much like *JavaScript* used to do under early Web browsers, the lack of error reporting results in it being somewhat challenging to debug *ActionScript* programs. This problem is augmented by the fact that *ActionScript 2* is not strongly typed. In fact, type annotations are used primarily as a developer aid and all values are actually dynamically typed (Grossman, 2006). Such issues are being resolved under *ActionScript 3*, but they still presented a problem with *Chevalier*'s development.

Fortunately my programming style is such that I build in very small incremental steps and test thoroughly as I develop. As such, I can usually catch problems as soon as they occur, and debug those problems as they can only be resident in the small portion of code that I have most recently changed. Regardless, the current solution for *ActionScript 2* development is to use an open source free third party ActionScript compiler called the *Motion-Twin ActionScript 2 Compiler* (MTASC, 2006) that is both faster and gives more detailed error reports. With this much stricter compiler it is far easier to identify problems resident in ActionScript programs. Such a compiler is vital to *ActionScript 2* development, as illustrated by the fact that when I attempted to reformat the *Chevalier* code for print by adding fairly innocent formatting returns and tabs to the entire code base in one sitting I found that I had introduced a plethora of hidden bugs that did not cause the compiler to crash. These bugs were so numerous that I returned to the previous unformatted version of the code, being unsure where and when a bug would surface.

What is apparent to any traditional programmer is the lack of any `int` and `float` type, nor is there any `const` indicator for variables. Strictly speaking there are no constants. ActionScript uses `Number` instead of `int` and `float` to cover both instances. This, of course, is slower, an issue that has been rectified as of *ActionScript 3*, which does utilize both `int` and `float` types for faster execution.

As all values in *ActionScript 2* are dynamically typed, *Flash 8* can sometimes get confused when a `trace` call is used to observe the contents of a variable. Unfortunately, when using `trace` to observe a `Number` member variable Flash will occasionally return the `Number` in a rounded form as it automatically converts it to a string for display, making, for example, 2.333 appear as "2." This has caused numerous problems for development of *Chevalier*, particularly when rotating a point location (Point2D) which

will often result in a very small trailing decimal value on the end, i.e. n = 2.00000003. When observed using `trace` such a variable n might appear as "2", but when tested using `if(n == 2)` the result is `false`, as n is not actually equal to 2 but rather 2.00000003. It is for this reason that there are numerous instances in *Chevalier* where numbers have been rounded to ensure that whole numbers are used when they are to be tested against other values.

Another problem with values in *ActionScript 2* being dynamically typed is that there is no allowance for operator or method overloading. It is possible to detect the type of a variable using `instanceof`, but this often makes for messy code and a heavier method. Similarly, when calling methods, the end parameters may be left empty causing them to be passed as `undefined`. It is not uncommon for a method to test a parameter to see if it is `undefined`, much like a void pointer in *C*. As *ActionScript 2* has no method overloading, testing for `undefined` is very useful, thought it appears a very unusual methodology at first.

There are two *ActionScript* programming metaphors which have been utilized to great effect in *Chevalier* and which are probably unfamiliar to some traditional programmers. They certainly were to me when I first encountered them. These are anonymous classes, and the use of square parentheses [] in dot syntax to allow a string to be used to express a path. Anonymous classes are a very useful tool for passing more than one parameter out of a method and utilizes the dynamically typed qualities of Flash that otherwise causes such problems during debugging. If, say, there are four values that need to be returned from a method, brightness(`_b`), hue(`_h`), saturation(`_s`) and a boolean flag(`true`), these can simply be returned bundled in an anonymous class. The anonymous class is syntactically very easy to create as the type of each parameter does not need to be

declared, as illustrated here, `return { brightness:_b, hue:_h, saturation:_s, boolFlag:true };` This feature is very useful for parsing XML, and is further discussed in “Chapter 6 Application Design—GeneralUtilities and XML Reader” page 73.

A key feature of ActionScript is the ability to use dot syntax dynamically with strings by use of square parentheses []. For instance, if an object has two methods, `car()` and `truck()`, a string may be used to specify which of the two methods is to be sent an argument. Normally, the methods might be called and sent the argument “88” as such, `objectInstance.car(“88”);` or `objectInstance.truck(“88”);` But in Flash the name of the method to be called may be held in a string, i.e. `myString = “truck”` and that string used to declare which method will receive the argument by wrapping the string in square brackets to invoke the call. This is done like so, `objectInstance[myString](“88”);` This feature is also useful for parsing XML, as is discussed further in “Chapter 6 Application Design—GeneralUtilities and XML Reader” page 73.

Finally, another important anomaly of Flash is the fact that any methods called from within a method are not actually executed until after the method has concluded. This makes it very difficult to trigger an ordered and timed sequence from within a linear sequence or loop. The code structure to “repeat while mouse down,” or rather, `while(mouseDown){},` a construct so common, if not central, to environments such as Hypercard or Director, is impossible to use in Flash. This is perhaps one of the reasons why programmers used to older multimedia environments often find Flash so difficult to grasp. This particular Flash anomaly is discussed in “Chapter 6 Application Design—Presentation Objects, PlaySnd Object” page 63.

Chapter 5 Game Design

The *Chevalier* game has been built using a cinematic 16 x 9 widescreen display aspect ratio, much like a movie, and has been designed with a black and white style chess metaphor, as the *DBM* rule set is frequently compared to a glorified game of *Chess*. The tournament aspects of *Chess* are also inherent in *DBM* making the *Chess* metaphor appropriate.

Similarly, the *Chevalier* pieces have been designed to reflect *Chess* pieces, particularly the knights and infantry themes. The opening page includes *Chess* like knights facing each other. Many of prototype versions of the type icons used for each piece in *Chevalier* were originally adapted from the playing pieces of old Simulations Publications, Inc. (SPI) games. These board games date from the 1970s, and include such titles as *Terrible Swift Sword* (SPI, 1976), *War of the Ring* (SPI, 1977), *The Crusades* (SPI, 1978) and *Empires of the Middle Ages* (SPI, 1980). After the prototype icons had been tested they were given to the graphic artist Peter Gifford, who completely rebuilt the whole icon set (see Figure 8.6) to be consistent in style, and created entirely new icons for the more tricky types, such as Skirmishers and Expendables, which had no effective prototype example.

All screens in *Chevalier* were initially prototype screens that were implemented and user tested for functionality before being passed to Peter to be "skinned;" he then replaced the functional prototype graphics with his end product graphic design. Such rapid prototyping and usability testing helped refine the interface before extensive labor was invested on graphic detail, avoiding wasted effort on designing controls which might later be proved redundant or inappropriate. Much of this prototyping and testing

contributes to the “feel” of the user interface, while the designer adds the “look;” for example, the Shooting and Battle window went through many iterations in prototype form. The final prototype version can be seen in Figure 5.0. Having reached this stage, the screen was then skinned, resulting in the look it has in Figure 8.17.

Figure 5.0 Prototype Shooting Window

The Grid

One of the most important design aspects of the game *Chevalier* was the conceptualization of the grid that the game pieces snap to. In particular, the pieces are required to move horizontally, vertically, and diagonally while still snapping to the same

grid. This poses a problem as interlocking rectangular elements on a square grid will rarely interlock cleanly when placed diagonally on the grid. As all playing pieces have the same width, that being 4 cm on the table top, or 60 meters in real life (see "Appendix B Units of Scale" page 121), it was decided to standardize using an on screen width size that could be transposed both horizontally, vertically and diagonally, onto the same grid.

There are a few instances where such a transposition is more or less possible. In one instance in particular, the conversion is not exact, but the transformation is close enough to be imperceptible to the player. On a square grid, a playing piece that is 7 squares wide (see Figure 5.1) will span almost exactly five squares when placed diagonally (see Figure 5.2). This phenomenon can be seen regardless of square size, and is evident mathematically with the Pythagorean Theorem, which states that "for any right angle triangle, the square of the hypotenuse is equal to the sum of the squares of the other two sides ($h^2 = a^2 + b^2$)" (Harris, 1998), where we see that a height and width of five, will yield a hypotenuse of 7.071, which is close enough to seven. Calculated in reverse, we see that a hypotenuse of seven has a height and width of $\sqrt{24.5}$, which is close enough to $\sqrt{25}$, which is equal to five.

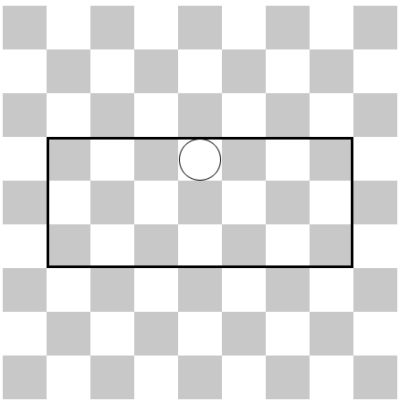

Figure 5.1 Seven wide piece

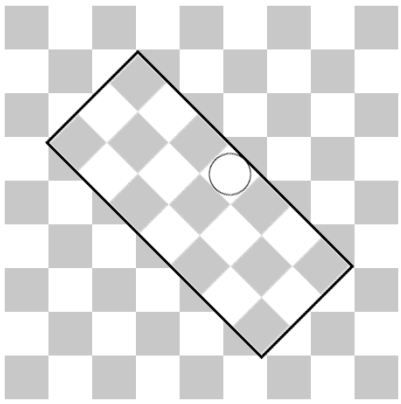

Figure 5.2 Spans five diagonally

This still leaves a problem with the depth of the piece. Although it was possible to create a situation where the width will always conform it is not so accommodating with the depth. A minimum of three depth sizes are needed for *Chevalier* to be an effective game. A shallow depth is needed to depict infantry, a medium depth for cavalry, and a deep depth for other types such as elephants and baggage. These depths were found by using pieces of depth three for infantry, four for cavalry, and seven for others. These depths conform close enough to the grid for a playable game. For instance, a depth of three, as seen in Figure 5.2, when placed diagonally will have a height and width of $\sqrt{4.5}$, which is equal to 2.1213, which is close enough to two. A depth of four, when placed diagonally, will have a height and width of $\sqrt{8}$, which is equal to 2.8284, which is close enough to three. A depth of seven will create a square piece, which will of course fit as the width of seven was chosen specifically for the purposes of aligning to the grid. Unfortunately, depths of one, two, five, and six do not fit acceptably and therefore *Chevalier* is strictly constrained to three possible depth sizes for its playing pieces.

The Figures

A further consideration was the very great number of troop types involved in the *Chevalier* game system. This variety of troops is what gives the game much of its appeal, but each and every troop needs a distinctive graphic and that graphic is unique from army to army. Historically a Crusader Light Horse "Turcopole" dressed and looked very different from a Saracen Light Horse "Bedouin." The graphics used must reflect this and they need to be accurate in their depiction.

To obtain this great and varied range of graphic representation I commissioned a Thai company called Siam Painting to purchase and paint the appropriate metal figures

for each and every troop type. The figures were then photographed, scanned and stylized for the game, resulting in what appears more like a detailed illustration than a painted metal figure. The equivalent cost to commission an illustrator to draw detailed images would be very expensive. Figure 5.3 shows a photograph of a figure of a Macedonian War Elephant as supplied by Siam Painting Service, Figure 5.4 shows the same photograph once processed for use in *Chevalier,* it has been converted to a 140 pixel high bitmap for use on screen. As almost all of Siam's clients are historical wargamers, one of the partners, John Sharples, oversees the process and ensures the figures are painted correctly and historically accurately. As such, the figures depicted in *Chevalier* are a well researched historical approximations of the troops they represent. The Siam Painting Service Website is at www.siampaintingservices.com.

Figure 5.3 Macedonian War Elephant Before processing

Figure 5.4 After processing

Chapter 6 Application Design

The *Chevalier* application has been organized into four distinct groups of objects, Game Objects, Presentation Objects, Game State Objets, and Rules Objects. For a System Diagram expressed using *Unified Modeling Language*[4] of objects in *Chevalier* and how they interact within the application see Figure 6.0.

The Game Objects, labeled in gray on the System Diagram, are controller objects, the most important of which is the Chevalier Object, which may be thought of as the main controller that is the first created object and from which all other objects for the application are generated. In particular, two Player Objects are created, one for each player. Each Player Object has four array lists with references to the created Elements pertaining to that player's army. There is a `_left` list, containing all the Elements in the player's left command; a `_right` list, for the right command; a `_center` list, for the center command; and a `_dead` list, where all Element references from any command are moved to once they are designated as removed from the game. Every Element Object has references to 9 Footprint Objects. The first eight Footprints are pre-generated templates representing the Element in every facing, one for each spoke of the compass, these eight pre-generated templates facilitate faster game response. The ninth Footprint, represents the current element position. There are also two sets of ten MoveType Objects that are used to describe types of moves Element Objects may make. For more information on the Game Objects see "Chapter 6 Application Design—Game Objects" on page 35.

[4] For information on *Unified Modeling Language* see www.uml.org.

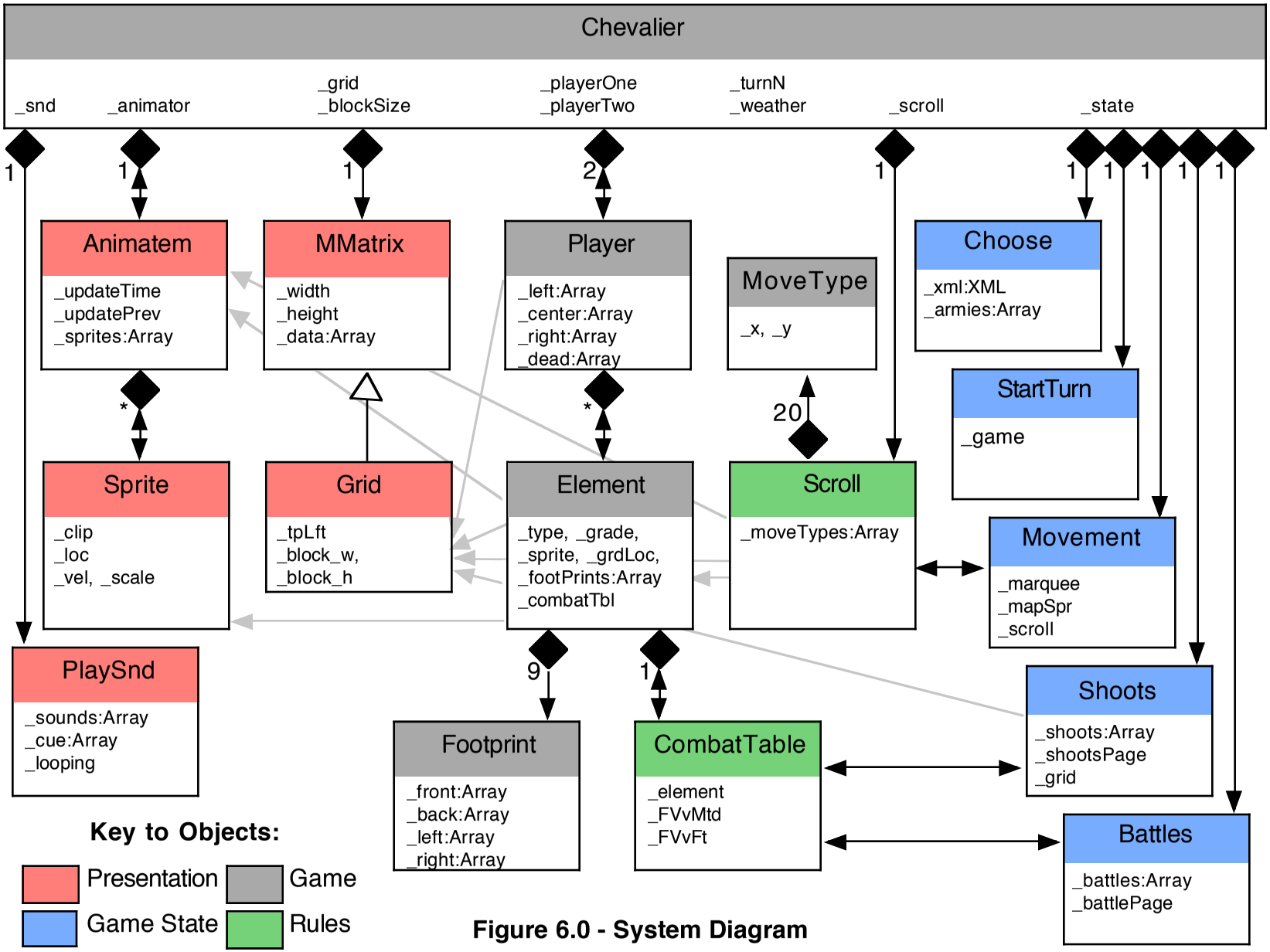

Figure 6.0 - System Diagram

The Game State Objects, labeled in blue on the System Diagram, are those objects that represent a specific state the *Chevalier* game is in. All Game State objects, implement the interface `IGameState`, which ensures that each State Object is prepared to accept various update calls, start state and end state calls, and standardized input from the keyboard and mouse.

There are five game states; "Choose," for choosing an army or battle scenario at the beginning of the game; "StartTurn," for displaying and initializing a new turn; "Movement," for moving elements around on the map; "Shoots," for conducting the resolution of distant shooting; and "Battles," for conducting close combat. The last four states each represent a game phase and are called cyclically each turn for each player. For more information on the Presentation Objects see "Chapter 6 Application Design—Presentation Objects" on page 54. For more information on the game phases that correspond to each state see "User Guide—What Happens each Turn" on page 95.

The Rules Objects, labeled in green on the System Diagram, have been intentionally encapsulated away from the Game Objects to allow for future interchangeability of game systems created by Wargames Research Group, namely *DBA*, *DBM*, *DBR* and *DBMM*. It is theoretically possible to cater for each *DB* rule set by creating each its own customized instance of Rules Objects. At the beginning of *Chevalier* the player could potentially choose a rule set by which to play the game and the appropriate Rules Objects be loaded. There are two Rules Objects, the CombatTable Object, which contains all the rules and tables for conducting combat, and the Scroll Object, which is partly a controller object, but has all the specifics of game movement. For more information on the Rules Objects see "Chapter 6 Application Design—Rules Objects" page 51.

The Presentation Objects, labeled in red on the System Diagram, are those that handle screen display and screen management. They are all generic objects that are encapsulated away from the main body of code and are in no way specific to *Chevalier*. The most important Presentation Objects are the Animatem and Sprite Objects, which together comprise the *Animatem* engine. This engine is essentially a velocity engine that handles the timed animation of multiple Sprites. When Sprites collide or reach a destination they send message back to the controlling *Chevalier* Object which deals with the situation appropriately. Every Element Object is assigned a Sprite that is used to display the state and position of the corresponding Element. There is also a Sprite assigned to the game map, allowing it to be moved, scaled, and rotated easily via the *Animatem* engine. A generic PlaySnd Presentation Object is used specifically to handle game audio, and there are MMatrix and Grid Objects that deal with map locations, terrain, and locations of Elements on the gaming map. For more information on the Presentation Objects see "Chapter 6 Application Design—Presentation Objects" page 54.

Game Objects

Class: Chevalier (for a full listing see “Appendix E—Chevalier.as” page 129)

Description:

The Chevalier Object is the root controller object and the first created, from which all other objects are made. The main task of the object is to initialize and regulate other game objects, handle message passing, and to establish the Flash Movie path to the game map and controls. These paths are subsequently passed to the other objects and are the key two paths for the whole game.

This object also regulates the game state, keeps track of the mouse location, listens for keystrokes, keeps track of the cursor state, invokes the sound object and uses it to trigger sounds, invokes the Animatem Engine and uses it to regulate animation, initializes the playing map with associated Grid Object, creates the two Player objects, keeps track of the turn and the weather, and initializes the Scroll object used to move Elements around on the map.

The `collision()` and `deactivated()` methods triggered by the Animatem Engine are resident here in this object. In particular `deactivate()` is called whenever an Element has finished moving, when the map has finished animating, and when the Information Scroll has finished opening or closing.

Methods:

Chevalier()	Constructor.
initialWeather()	Randomly generates weather at the beginning of the game.
weatherDice()	Randomly calculates a change in the weather, this is called at the start of every turn.
aboutToEngage()	Check the move lists of the player's elements to see if any engagements are still yet to occur. This is needed at the end of a turn to fix a bug where battles that are still yet to be triggered by moving Elements are otherwise skipped over and not fought.
ptGrdLoc()	Convert a screen location to a grid location.
collision()	When a sprite collides with another sprite this method is automatically called by the animator. Chevalier does not need to use collision detection of sprites so this method is empty.
deactivated()	When a sprite deactivates this method is automatically called by the animator. There are generally three cases of sprites deactivating. The scroll deactivates when it has finished opening or closing. The map deactivates when it finishes animating to a new location. And an Element deactivates once it reaches a destination location it was moving to.
spinMapTo()	Tells the map to rotate to a new angle.
scaleMapTo()	Tells the map to scale to a new size. Growth constant e (2.718) is used to change the map location with the scaling, so scale looks like map has perspective.
changeMap()	Animates the map to a new position according to four parameters, scale, angle, location and duration of animation.
testForElement()	Count instances of elements along a list of up to seven points and return the one with the most hits. Due to the width of Elements, when testing for seven adjacent points there can never be more than three possible elements.
testForElements()	Returns all instances of elements at various grid locations specified by an array of points. Any duplicate Elements are removed from the list.
cnvPtToMap()	Convert a global screen location to a local location on the map.
playSnd()	Tell the PlaySnd object to play a sound using, if necessary, a delay before playing. Many sounds such as walking and fighting sounds trigger one of a number of randomized variations such sounds usually also utilize a randomized stagger, allowing layering of sound to give the effect of a multitude. Some sounds, such as wind and rain, are set to loop perpetually.
useSmall()	Elements can have a "large" and a "small" version of their type the icon for improved clarity at distant v close maps. Currently this is only used for the generals star icon, and the effect is only subtle.
useLarge()	Elements can have a "large" and a "small" version of their type the icon for improved clarity at distant v close maps.
removeFilters()	Remove all filter effects from elements, this allows for faster map animation.
setFilters()	Reinstate filter effects, putting back glow filters that were removed while the map was animating.
normalizeElements()	Set the state (not status) of all the players elements to normal. Removes all functional state glows, roll highlights, and green shooting highlight. This is called at the end and beginning of a turn.
switchActivePlayer()	Switches the player turn and increments the turn counter if moving from the second players turn to the first.
freezeCursor()	Freezes the cursor. Useful for the watch cursor which should take priority over other cursor states.
setCursor()	Changes the cursor. Only works if cursor is not "frozen" using freezeCursor() Possible cursors are: watch, google, zoom_in, zoom_out, hand, grab, crosshair, battle, lft, tplft, tp, tpRht, rht, btmRht, btm, btmLft, wht_lft, wht_tplft, wht_tp, wht_tpRht, wht_rht, wht_btmRht, wht_btm, wht_btmLft.
unfreezeCursor()	Unfreeze the cursor and set it to a new state.

update()	Called constantly by onEnterFrame of the program, behaving much like a traditional main loop. Tells Animatem, PlaySnd and the currently active state object to update. If glow and bevel filters on the Elements need to be updated, for instance after the map has just animated, then those filters are told to update.
updateScrollText()	Occasionally the information scroll needs to be updated due to sudden changes in the selected element(s) state.
isFriendly()	return true if Element e is friendly to the current player.
addElement()	Called from the XML reader in the Choose Object. Builds and element for a player.
state()	Trigger a new game state.

Class: Player (for a full listing see “Appendix E—Player.as” page 152)

Description:

The Chevalier Object will create two instances of Player Object, each maintaining all game information pertaining to the player, such as player color, player army (i.e. “Crusader”), Elements in the left command, right command, center command, eliminated elements, which Elements are the commanding Elements for each command, morale values for each command, and default map and scroll positions for that player.

Player Object also has a static initialization that creates all the template data used for each of the three Element base depths when an Element is created. This static Footprint data is constantly referenced as a starting point by all Elements as they move, this way they don’t have to reconstruct Footprint data from scratch. Moreover, whenever an Element is created for a player that creation is done through the player object using the `add()` method. The template Footprints are stored here in a static form as it grants the `add()` method easy access to them.

Methods:

Player()	Constructor.
initialize()	Assign objects static variables, these are mostly base footprint definitions.
add()	Add a new Element to this player's army.
rollPIPs()	Roll player initiative dice for this player.
setAsGeneral()	Assign an element as a general of a command.
elementDead()	Remove an element from command lists and add to dead pile.
getCmdStatus()	Return a string describing the morale status of a command. Sometimes it's only important to know if the command is shattered, broken, or dispirited, [flag = false] such as during battle.
getMoraleValue()	Get the morale of one of this players commands.
getMoralePercent()	Get the morale % of one of this players commands.

Class: Element (for a full listing see “Appendix E—Element.as” page 179)

Description:

Element instances are always created by the Player Object. The Player instance passes player information and initialization parameters originating from XML to the `new Element()` constructor, so that the Element can be created under the command of that player.

The Element Object specifies a great many parameters giving the element its individuality, but it also contains the many methods needed to be self aware on the map grid. An Element is able to detect for other Elements around it, and also how to respond under certain circumstances, such as fleeing, being killed, pursuing, recoiling, finding the front rank of a group, finding the rear rank of a group, turning to face an enemy, and, perhaps most importantly, detecting when moving into combat with an enemy Element.

This Object also has the all important `testLocation()` calls used by the Scroll Object that determine if the Element is capable of moving to a certain location at a certain orientation. There are also calls for dealing with the movement shadow that appears under the Element when the Element is about to move to a location. Every Element Object maintains a corresponding *Animatem* reference to a Sprite Object (`_sprite`) used to display where it is on the map.

Methods:

Element()	Constructor.
initialize()	Initialize class with static variables.
stateNormal()	Set state of element to normal. Called by reset() unless Element engaged.
stateRollHL()	Set state of element to Roll (white glow if friendly, black glow is enemy).
stateSelectHL()	Set state of element to Highlighted/Selected (yellow glow).
statusEngaged()	Set status of element to Engaged (burgundy/red glow). This element is now in combat.
statusShoots()	Set status of element to Shoots (green glow). This element has become the target of bowmen, or is a bowmen shooting. This is a temporary status only occurring during the Shoots phase/mode, it cannot occur if the element is Engaged.
statusMoving()	Set status of element to Moving (white glow).
statusNormal()	Set status of element to Normal (no glow).
reset()	Called at the beginning of a players turn. Sets movement points back to full, clears its last move made, clears nudges made, clears flag indicating that Element withdrew from battle, and if not engaged in battle, set its status and state to normal.
disengage()	Disengage Element from battle state.
small()	Use the small/simplified version of icon for the Element that is more readable when the map is small.
large()	Use the large/detailed version of icon for the Element that is clearer when the map is large.
alpha()	Make the Element semi-transparent to denote it is in another command.
setFilters()	Set the filter effects (glows) for this Element according to state and status.
removeFilters()	Remove all filter effects for this Element. This is done before animating the map, so to allow very rapid animation.
getFootprint()	return the corresponding footprint to use when placing this elements data on the grid at a specific angle.
shadowAt()	Draw angled placement shadow at a location.
resetShadow()	Place shadow at location element grid data is at.
testLocation()	Test if this Element is able to move to a new location and angle on the grid. It is crucial that this call be made on a potential move before actually performing such a move with moveMeTo() or setLoc(). The Scroll object performs most calls to testLocation().
squareClear()	Called by testLocation(). Checks if a grid square can be moved into by this Element. Diagona Elements have "half points" which can contain "half an edge square" which can make this operation messy. Similarly, corner points in some instances are considered clear.
remove()	Remove element from the game. This is called (not surprisingly) when an element is "Killed" or "Spent".
moveMeTo()	Move this element to a new grid location. To do this cleanly the element's footprint must be removed from the grid using removeData() and reinstated at a new location using setLoc().
removeData()	Remove this Elements footprint of information from the map/grid. setLoc should be called soon after, this call to reestablish the Element on the grid, unless of course this element is killed/being removed.
setLoc()	Set the location and angle of this element to a new location and angle. All tests to see if this placement is legal will have already been done by other functions in the Scroll object, so all of the work here is in changing the Elements data footprint and testing for battles being triggered by moving to the new location.

aboutToEngage()	returns true if this element has any engagements about to happen on its _movelist. This test is needed by the Battles state to check if the player has ended his move but there are still resultant battles/engagements that have not yet been triggered.
advance()	Move this Element to the next location on the _movelist.
speedLimit()	return the speed of any element in front if it's slower.
globalLoc()	return the global location of this element, or a specific footprint corner/part.
globalRect()	Make Rect of Element global according to the _sprite path.
makeGlobal()	Make a Point2D global according to the _sprite path.
atDestination()	Triggered by Animatem when elements sprite reaches a destination it was moving to.
adjacent()	Return a list of any Elements directly adjacent to this one. This is useful for establishing Elements in a group.
elementInFront()	return - Element aligned directly in front.
elementBehind()	return - Element aligned directly to rear.
elementToLeft()	return - Element aligned directly to left.
elementToRight()	return - Element aligned directly to right.
strictTestForElement()	Test for an Element at a specific footprint point.
elementMostInFront()	return Element most directly in front edge.
elementMostBehind()	return Element most directly to rear edge.
elementMostToLeft()	return Element most directly to left edge.
elementMostToRight()	return Element most directly to right edge.
elementsInFront()	return list of Elements along front edge.
elementsBehind()	return list of Elements along rear edge.
elementsToLeft()	return list of Elements along left edge.
elementsToRight()	return list of Elements along right edge.
leftFlanked()	return any enemy element attacking the left flank.
rightFlanked()	return any enemy element attacking the right flank.
rearAttacked()	return any element attacking the rear of this element.
leftOverlap()	return element causing an overlap on left corner.
rightOverlap()	return element causing an overlap on right corner.
recoilPt()	return a point to recoil to, if available.
recoil()	This Element (and any behind it) recoil a base depth.
repulsed()	This Element (and any behind it) are repulsed, which is similar to flee but does not retreat as far.
flee()	This Element (and any behind it) flee from battle.
spent()	This Element is spent (similar to killed).
killed()	This Element is dead.
pursue()	Pursue a retreating enemy Element.
checkForUnexpectedContact()	Check if Element has moved/stumbled into an enemy flank overlap or a diagonal contact, triggering an engagement.
engagingWith()	Return whatever enemy Element this Element is fighting.
turnToFaceEnemy()	Turn this Element to face an attacking enemy Element. Elements already engaged will not turn to face.

firstTurnToFace()	Turn this Element to a facing, such as "Rear", "Left", or "Right." The first Element in a column that needs to turn to face is a special case, subsequent Elements behind use the regular (recursive) turnToFace() function. If the first Element is Engaged then the turnToFace() is passed through to the element directly to its rear.
turnToFace()	Recursive function called by firstTurnToFace. Turn this Element to a facing, such as "Rear", "Left", or "Right." Elements already engaged will not turn to face. Elements turning to face a flank will often have to push elements behind them back. All Elements in column will turn to face the same direction.
isFriendly()	Check if an element is friendly with this Element.
getFrontRankElement()	Recursive call to find the element at the front of a column.
getRearRankElement()	Recursive call to find the element at the rear of a column.

Class: Footprint (for a full listing see “Appendix E—Footprint.as” page 218)

Description:

Footprint is a data structure object that maintains lists of Point2Ds describing an Element’s key locations represented on the Grid. There are three sets of template Footprint instances created by the Player Object initializer. One Foorprint for each compass direction in each set. These templates are used to quickly generate a unique key Footprint kept by each Element whenever it moves. When looking for information on an Element and what Grid points it covers on the map the key Footprint for the Element is referred to.

There are accessors to reference into the following Footprint data:

`_wholeSquares`	Array of whole square points for this Footprint
`_halfSquares`	Array of 1/2 square points for this Footprint
`_front`	Locations at front of Footprint
`_back`	Locations at back of Footprint
`_left`	Locations to left of Footprint
`_right`	Locations to right of Footprint
`_tpLeft`	Outside top left point used to check for overlap
`_tpRht`	Outside top right point used to check for overlap
`_sftLeft`	Shifted back left point used for friendly element shifts
`_sftRht`	Shifted back right point used for friendly element shifts
`_flankLeft`	Flank contact point for elements contacting on left
`_flankRht`	Flank contact point for elements contacting on right
`_modifier`	Modifier for drawing the element

Class MoveType (for a full listing see “Appendix E—MoveType.as” page 222)

Description:

The MoveType Object is a wrapper class for defining attributes pertaining to a specific move operation to be performed an a group of Elements. All MoveType objects are created by the Scroll Object whose function is primarily to move Elements around the Grid. MoveType Objects are almost always created in pairs, one defining the move as performed by Elements with a diagonal orientation, the other for Elements that are horizontal or vertical.

Sometimes a temporary MoveType is created by the Scroll Object to perform dynamically changing moves such as Wheels, or when the Scroll needs to “shift” a group of Elements so they conform to another group or to make contact with the enemy. In such instances the MoveType is often mutated using `plusOne()`, or `minusOne()` methods. The `nonZero()` method is used after calculating dynamic moves to ensure the resultant modified move isn’t a “zero” move that results in no movement.

Methods:

MoveType()	Constructor.
fixMove()	Generate a new MoveType from this one, this is used for wheeling elements and shifting elements into contact.
plusOne()	Generate a new move that is one step further.
minusOne()	Generate a new move that is one step shorter.
nudgeRight()	If a diagonal move then nudge to the right (+1 in x). Non-diagonal moves cannot be nudged.
nonZero()	A move type should never be passed with both x and y as zero as the controller will show the move as enabled but selecting it will have no effect. If a "zero" move is found then plusOne() is called.

Game State Objects

All Game State objects, implement the interface `IGameState`, which ensures that each state object contains the eight standard methods that are regularly sent by the controlling Chevalier Object. In particular, each state is responsible for setting up the screen to commence the state, and cleaning up the screen on conclusion of the state.

Specifically, the methods required in each State Object are; `update()`; tells the state to update the screen; `mouseDown()`, state must handle the user pressing the mouse at a coordinate; `mouseUp()`, state must handle user releasing the mouse; `mouseMove()`, state must handle user moving mouse to a new coordinate; `keyDown()`, user has pressed a certain key; `keyUp()`, user releases that key; `start()`, handle setting up and stating the Game State; `end()`, finish and clean up the game state. The Chevalier Object simply redirects any mouse and keyboard input it receives to whichever State Object corresponds to the current state that *Chevalier* is in.

For a full listing of the interface IGameState see “Appendix E—IGameState.as” page 226.

Class: Choose (for a full listing see “Appendix E—Choose.as” page 228)

Description:

The Choose state is the initial state that *Chevalier* begins in. Under the Choose state the game scenario is chosen or a fantasy scenario is chosen with each player choosing different armies to play. The fantasy option is much like *DBM* tournament play, where any army from any historical period is matched with any other.

For examples of the screens involved with the Choose State see Figures 8.0 and 8.1.

Methods:

Choose()	Constructor.
start()	When the choose section is started Chevalier must go to the Chevalier "choose" frame.
rollChoose()	Triggered by mouse rollovers, gives a brief description of the selection for the button being rolled. Used by both the choose battle and choose fiction screens. The choose fiction screen has an additional feature of a silhouette that is fade/superimposed over the selection showing that armies insignia.
btnViewBattle()	Loads the battle description.
btnOK()	Confirms a battle selection.
btnBackToBattle()	goes from the choose fiction screen back to the select battle screen.
createBattle()	This is used by the battle game sequence. Once a battle has chosen that battle is loaded from the corresponding XML file and the players armies are registered with the main Chevalier game object.
btnChoose()	This is used by the "fictional" game sequence. Once a player has chosen an army, all buttons are disabled so they they can't be accidentally clicked and their army choice is registered with the main Chevalier game object.
createPlayer()	This is used by the "fictional" game sequence to trigger the XML loading of player one’s or player two's army. A randomized terrain is also chosen here, and buttons disabled for armies that have been selected by the other player.
startGame()	Exits the choose state and commences the actual game. This is called from the Flash timeline after both armies have been created from the XML files at the end of the "loadBattle" sequence.

Class: StartTurn (for a full listing see “Appendix E—StartTurn.as” page 234)

Description:

The is a very simple State where the game is put on hold until the next player is ready to commence their turn. In this state a banner pronounces the beginning of the next player’s turn.

See Figures 8.2 and 8.19. for examples of the Start Turn banners.

Methods:

StartTurn()	Constructor.
start()	Displays the "Start Turn" window for the player. The weather roll is made and all the players elements are reset. The map is flipped for the player.
btnBegin()	Triggered by "begin" button in start turn window. This begins the movement phase for the player.

Class: Movement (for a full listing see “Appendix E—Movement.as” page 237)

Description:

The Movement State is the state in which the player may “google” the various troops and move them around the map. This state is very sensitive to all mouse input and key strokes. The state also handles marquee selection and movement of the map. Hit tests are performed to check if mouse downs are to be sent to the Information Scroll Object or for the game map.

Methods:

Movement()	Constructor.
update()	Tell the scroll and the control palette to update also.
mouseMove()	Deal with mouse moving, which is usually just passing the mouseDown message through to the Scroll object.
mouseDown()	Deal with mouse down, which is usually just passing the mouseDown message through to the Scroll object, but can intercept the message for dragging or zooming the map.
mouseUp()	Deal with mouse release, specifically, if the map is being dragged then cease dragging and record the new map location.
keyDown()	Assign functions to key presses, specifically, SPACE lets the user drag the map, SHIFT allows zoom in, CONTROL allows zoom out, LEFT and RIGHT arrows spin the map, UP and DOWN arrows zoom in and out of the map. Keypad numbers scroll the map.
keyUp()	Release _keyHeld value when a key is released.
clearKey()	Empty _keyheld value and return the cursor to an arrow.
paletteHit()	Hit test for the controls palette, used to check if the mouse is selecting the controls or clicking on the map.
start()	Establish "Movement" state, namely, display information for the new turn on the control palette, and update the weather.
end()	Finalize "Movement" state, namely, record the position the player's map and ensure the scroll is closed.
updatePalette()	Update dynamic portions of palette, namely the compass and zoom btns.
btnZoomIn()	Triggered by the “zoom in” button, manages the amount zoom out by.
btnZoomOut()	Triggered by the “zoom out” button, manages the amount zoom out by.
btnSpinMap()	Turn the map by an amount.
btnSpinMapTo()	Turn the map to a specific angle.
stopMap()	Stop the map from animating, but only if it's not animating to a specific destination point.
mapVelocity()	Animate the map in direction of velocity vector.
marqueeStart()	Set the first (origin) marquee point and ready the cursor as a cross hair.
marqueeDraw()	Set the second marquee point and display the marquee.
marqueeSelect()	Test each element inside the marquee and try to select it.
marqueeClear()	Move the marquee off screen and clear its Rect definition.

Class: Shoots (for a full listing see “Appendix E—Shoots.as” page 240)

Description:

The Shoots state checks the map Grid Object to see if any of the player’s Elements are in range of the enemy and cues them up in the array `_shoots`. Once this list is established the Shooting combat interface is invoked and the player is allowed to page through the possible battles and conduct them via the “Fight!” button. The actual battles are resolved using the CombatTable Objects associated with each of the Elements in the combat.

Methods:

Shoots()	Constructor.
start()	Establish "Shoots" state, namely, generate a list of shoots based on trace lines of shooting and targets in range, and sort that list.
drawShootsWindow()	display the shooting window showing battle _battlePage.
btnNext()	flip forward a shoots page.
btnPrev()	flip back a shoots page.
btnDone()	Handle the "Done" button on the battle window.
btnOK()	Handle the "OK" button on the battle window.
btnFight()	Handle the "Fight!" button on the battle window.
isDiagonal()	Test if angle is a diagonal, needed when tracing lines of shooting.
thetaOf()	Needed when tracing lines of shooting.

Class: Battles (for a full listing see "Appendix E—Battles.as" page 248)

Description:

The Battles state checks all the player's Elements to see if any are engaged in close combat and cues them up in the array `_battles`. There are some cases where an Element that was moved by the player during the Movement phase is still moving and has not completed its movement into combat. In such instances a `_wait` boolean flag is set and the Battles state patiently waits for those Elements to complete their move before checking for engaged Elements. Once the `_battles` list is established the Battle combat interface is invoked and the player is allowed to page through the possible battles and conduct them via the "Fight!" button. The actual battles are resolved using the CombatTable Objects associated with each of the Elements in the combat.

Methods:

Battles()	Constructor.
update()	If _wait flag true then wait for all battles to be triggered before allowing this Battle state to proceed normally.
start()	Establish "Battles" state, namely, generate and sort a list of battles to be displayed. If there are still battles yet to be triggered due to movement then wait for them to trigger by use of a _wait flag.
drawBattleWindow()	Display the battle window showing battle _battlePage.
btnNext()	flip forward a battle page.
btnPrev()	flip back a battle page.
btnDone()	Handle the "Done" button on the battle window.
btnOK()	Handle the "OK" button on the battle window.
btnFight()	Handle the "Fight!" button on the battle window.

Rules Objects

There are two Rules Objects; the CombatTable Object, which contains all the rules and tables for conducting combat; and the Scroll Object, which is partly a controller object, but contains all the specifics for game movement. These objects have been intentionally encapsulated away from the Game Objects to allow for future interchangeability of game systems created by Wargames Research Group, such as *DBA*, *DBM*, *DBR*, *DBMM,* and whatever other versions of the *DB* rules they might release. It is possible to cater for each rule set by simply tweaking instances of the Rules Objects and customizing them for each version. At the beginning of *Chevalier* the player could potentially choose a set of rules under which they want to play their game and the appropriate Rules Objects be loaded.

Class: Scroll (for a full listing see "Appendix E—Scroll.as" page 254)

Description:

The Scroll Object is a control object that is specifically used to "google" and move single Elements or Element groups according to specific game rules. In particular, it keeps track of the selected units, displaying information on them in an "Information Scroll" Sprite that it also presides over. When units are selected a Movement Control will appear attached to the Information Scroll and tests are performed for each possible movement to see what moves are legal. There are many issues when checking moves to see if the group will need to "shift" and to snap into contact with other groups or enemy Elements, these tests are performed in the `testMoveType()`, `shiftPt_EnemyContact()`, and `shiftPt_FriendlyContact()` methods.

Methods:

Scroll()	Constructor.
handleMouseDown()	Handle mouse being pressed, usually to select an Element on the map, or to drag the scroll.
handleMouseUp()	Handle mouse release, namely, drop anything being dragged, including the scroll itself.
alphaOthers()	Darken all "other" elements, those not of the selected element's command.
handleMouseMove()	Handle the mouse moving over to a new grid location.
tryToSelect()	Try to add an element to the selection, this is called by handleMouseMove() and also by marqueeDraw() in the movement object.
deselect()	Deselect all selected elements and close the scroll.
update()	Update the scroll, in particular update the movement tool to reflect the angle of the selected elements.
scrollConcluded()	Scroll has either finished opening, or finished closing. This is triggered by the animator objec via the Chevalier object's collision() method once the scroll animation has finished.
openScroll()	Scroll is starting to animate open to display statistics for a rolled element.
scrollOpen()	Calculate PIP (initiative points) cost to perform a move cmd.
updateScrollText()	Update the text on the scroll to reflect the rolled element or selected group.
getStatus()	Return the collective influence for the rolled element or selected group.
closeScroll()	The scroll is starting to animate closed.
scrollClosed()	The scroll animation has finished closing.
calcPIPcost()	Calculate PIP (initiative points) cost to perform a move operation.
selectElement()	The user has selected the "rolled" element that the mouse is currently over.
buildAdjacentList()	Build a list of all the elements adjacent to the currently selected Elements. This is needed as these Elements are eligible to be added to the selected group of Elements.
addAdjacentList()	When an Element is added to the list of selected Elements any elements adjacent to it must also be added to the adjacentList.
drawMoveControl()	Draw the movement control from scratch. This requires testing each possible move type button with the selected Elements and either 1) store the result if a legal move or 2) disable the control if an illegal move. As there are many potential moves, and every selected element has to be tested with each move type, drawing the control is a lengthy process. As such the move control is only updated once the user has stopped moving/clicking the mouse.
dynamic_ffwd()	Modify the forward cmd according to maximum movement and location of other Elements.
dynamic_rert()	Modify the retreat cmd according to maximum movement and location of other Elements.
smplTestMoveType()	Greatly simplified version of testMoveType used by dynamic_ffwd() and dynamic_rert().
mvControlAngle()	Update the angle of the move control to reflect the angle of the selected elements being moved. This is called by both Scroll.update and drawMoveControl.
deducePivots()	Deduce pivot points from scratch, these are the edge points around which a group of selected Elements will wheel.
testMoveType()	Test if an element can perform a move type. return [false] if not, and [true, shift] if can, with 'shift' being a shift amount needed if there is contact with an enemy or friendly troops.
shiftPt_EnemyContact()	Calculate shift point for shifting to other enemy Elements.
shiftPt_FriendlyContact()	Calculate shift pt for shifting to other friendly Elements.

rotateShiftBack()	The "shift" point must be rotated back to "N" (270) with normal and "NE" (315) with diagonal moves for the shift to work with the MoveType definition.
isDiagonal()	Tests if angle is a diagonal.
rollShadow()	Called by rollOver of the mv buttons. The shadows of the selected Elements are shown where the Elements will be if this command is selected, also, the PIP cost of the move is displayed.
resetShadow()	Called by rollOut of the mv buttons. The shadows of the selected Elements are reset and the PIP cost of the move is cleared.
dynamic_wheel()	Change the wheel (pivot) movement command for a specific Element according to to the size and direction of all (selected) elements being wheeled.
moveThem()	Move all selected Elements according to a specific movement command.

Class: CombatTable (for a full listing see "Appendix E—CombatTable.as" page 292)

Description:

The Player Object creates a CombatTable Object for every Element that it creates and passes this reference to the Element on creation. CombatTable is a collection of methods specifically to conduct various rule intricacies for combat, such as combat factors, shooting factors, grading factors, battle result strings, and other intricacies such as what Element Types will pursue after battle and which Types can move through which when moving and recoiling.

Methods:

CombatTable()	Constructor.
lessThan()	return result when score less then enemy of type.
doubledBy()	return result when score doubled by enemy of a type.
convResult()	Convert battle symbols to String results.
factors()	Tally Support & Tactical Factors for this Element.
shootingFactors()	Tally Shooting factors for this Element.
shootingTally()	Rough tally of this elements shooting effectiveness against a specific enemy. Use by the Shoots object to determine who should be the primary shooter at a target.
grading()	Determine factors dictated by the grading of both Elements in combat. These Grading Factors are add AFTER the dice for the battle have been rolled.
displayOddsV()	display odds for battle against another element.
conductBattleV()	conduct battle (roll dice) against another element.
pursue()	return true if Element e will pursue this Element.
moveThrough()	return true if Element e can pass through this element.

Presentation Objects

The Presentation Objects all deal with generalized display capabilities such as as animation and sound. Although Flash itself is essentially a library of Presentation Objects, there are some important methods and objects that needed to be created for *Chevalier* to facilitate easy handling of various specific needs. These Presentation Objects all extend Flash's capabilities and, more importantly, are very generic. Meaning that, although they add features to *Chevalier*, they may also be used to add features to a wide range of game applications and have been intentionally written with an eye towards reusability, making these objects more interesting than the other groups.

In a *C++* application, the presentation layer objects are usually a very distinct in that they are platform specific, such as a Macintosh Window manager, or a Windows Window Manager, or a Unix X Window manager. Part of the beauty of Flash is that *Flash Player* runtimes are available for all three platforms without such specific presentation objects needing to be created. Flash is essentially a group of pre-compiled runtime presentation libraries. In the traditional sense of application programming, it would not be false to say that Flash is my presentation layer, as it deals with all the things a presentation layer traditionally would.

However, Flash's presentation capabilities are not all-encompassing. Some important objects needed to be built for *Chevalier* to extend Flash's capability. Flash is very good at prepackaged and predefined animation, but in order to deal with highly-interactive and freeform animation such as that controlled by frequently changing variables, an animation object extending Flash's capabilities needed to be written. This generic object I have called *Animatem*.

Animatem Animation Engine

The *Animatem* animation engine is the keystone around which the *Chevalier* game was built. It would be wrong to say it is the heart of the game, but it is certainly the heart of *Chevalier's* presentation layer. *Animatem* was the first object set made for *Chevalier* and was thoroughly built and tested before any other *Chevalier* code was written. In fact, *Animatem* was first built as part of a different and much smaller preliminary game designed to thoroughly stress test *Animatem's* capabilities. This was especially important as I needed to know for sure the engine to be a solid encapsulation of independent code, and not the possible source of frequent and unusual bugs as the *Chevalier* project was developed.

The preliminary game used to test and build *Animatem* in Flash was a port of an *Asteroids* game that I had previously written using *C++* and *OpenGL*[5] for the class "CSCI E-234 Introduction to Computer Graphics" taught by Hanspeter Pfister. This *C++* version, in turn, was based off an *Asteroids* game that I had written as a contractor some years earlier using the Macromedia Director environment. Director is a similar animation and programming environment as Flash, and is really the precursor to Flash, particularly as Director and Flash are made and sold by the same company.

The Director implementation of *Asteroids* is actually where the *Animatem* class was first conceived. As such, the *Asteroids* project was a good starting place for me to begin development in Flash as I'd already written the project twice before, once in *C++* and once in *Director*. I knew the constructs very well, and those constructs and needs are comparatively simple, at least when compared to the larger constructs of *Chevalier*.

[5] For information on *OpenGL* see www.opengl.org.

Asteroids also made a good bench testing project, as, having three versions of the same game, I could directly compare the differences between the three development environments. Not surprisingly, the *C++* version is the smoothest and fastest, followed by the Director version, which is also surprisingly quick, and lastly the Flash edition, which technically performs the slowest—but perfectly acceptably, particularly on today's machines. Not surprisingly, the Flash version is the most impressive since it has the advantage of inheriting the development experience of the two prior projects, and also because I could leverage the more modern presentation capabilities of *Flash 8*, which allowed me to easily implement advanced features such as glows, that otherwise would be far more difficult to implement in any of the preceding editions. For a full listing of the Asteroids code see "Appendix E—Asteroids.as" page 380. The completed Flash *Asteroids* game is available online at www.mocaz.com/games/Asteroids.html.

The *Animatem* engine originated from a small piece of sample code featured in the back of the manual for *Director 4.1.* At the time of *Director 4.1's* release Intel had just released their new Pentium processor which was then competing with much slower 386 and 486 machines. *Director*, along with many other applications, where experiencing an unusual problem in that the Pentiums were performing too fast for the software running on them. Presentation operations that had previously been optimized for performance, or rather, to simply display as fast as the processor would possibly allow, were suddenly performing too quickly.

What was suddenly greatly needed in *Director* was a timed solution with a high granularity (1/60ths of a second) to pace animations so to play consistently, correctly, and smoothly on all platforms regardless of processor clock speed. The granularity has much improved since then and the *Flash 8* timer operates not at 1/60ths of a second but at

millionths of a second (1/10000). Here is a version of the *Director* sample code called "smoothslide," as implemented by a colleague, Nigel Doyal.

```
--  smoothSlide - animate a sprite (ie. whichSprite) from its current
--                position to another position specified by newH and
--                newV. Make sure that the animation takes exactly the
--                number of ticks specified by numberOfTicks.
--                (nb. speed is constant whatever machine you run on)
--
on smoothSlide whichSprite, numberOfTicks, newH, newV

  set startTicks = the ticks
  set endTicks = startTicks + numberOfTicks
  set initialH = the locH of sprite whichSprite
  set initialV = the locV of sprite whichSprite

  --  calculate the distances to travel
  set distH = newH - initialH
  set distV = newV - initialV

  --  calculate the pixels per tick to get there (ie. velocity)
  set velocityH = float (distH) / float (numberOfTicks)
  set velocityV = float (distV) / float (numberOfTicks)

  repeat while the ticks < endTicks
    --  How much time has passed (in ticks)
    set timePassed = the ticks - startTicks
    --  calculate new locH
    set thisH = initialH + integer (velocityH * timePassed)
    set thisV = initialV + integer (velocityV * timePassed)

    set the locH of sprite whichSprite = thisH
    set the locV of sprite whichSprite = thisV
    updateStage
  end repeat

  --  Just to make sure it got there properly, set the position
  set the locH of sprite whichSprite = newH
  set the locV of sprite whichSprite = newV
  updateStage

end smoothSlide
```

As you can see this Smoothslide function[6] accepts four parameters,

[6] Director uses a scripting language called Lingo which was based off Hypercard's scripting language created by Bill Atkinson, one of the geniuses behind the Macintosh's original QuickDraw 2D graphics library, a core part of the classic Apple Macintosh operating system. With Hypercard, Atkinson tried to create an easily readable programming language "for everybody else." Unfortunately the experiment failed, largely, I believe, as English constructs, particularly words such as "the," tend to be ambiguous as to when exactly to use them in a programming context. Such constructs generally caused many frustrating and unnecessary syntax errors. Fortunately, the concise *C* style syntax has now taken over and been standardized on, as evidenced with Java, Javascript and Flash Actionscript.

`whichSprite`, which specifies the graphic to be animated, `numberOfTicks`, that specifies the duration of the animation in 1/60s of a second (called "ticks") and `newH`, and `newV`, which specify the target horizontal and vertical location the graphic is being moved to. The first line of code featured in `smoothslide()` sets the variable startTicks to the variable `the ticks`, which is a system variable that returns the number of ticks passed since the program was run. Smoothslide then calculates a precise horizontal and vertical velocity according to the specified duration of `numberOfTicks`. The animation is then conducted in a repeat loop[7] using the amount of time passed since the beginning of the animation and the velocity of the sprite to deduce the animated objects location. In this way, consistent animations can be made regardless of computer speed.

Animatem takes this idea of regulated animation using precise velocities and develops it considerably, expanding it to include multiple graphics (sprite objects) animating simultaneously, rather than just one graphic. The animation is no longer conducted in a repeat loop, but rather drawing of sprite positions is handled by an `update()` method that is called constantly by Flash's `onEnterFrame` call, effectively simulating a traditional program main loop.

Many other features have been introduced, such as; 1) changing the "cell" or frame displayed, as in a traditional animation cell, at a rate, as specified as Frames Per Second (FPS); 2) dealing with physics, such as friction; 3) and barriers such as walls; 4) and checking for collision detection between animating sprites; 5) and angular velocity and a scaling velocity. The end result is that *Animatem* is essentially a 2D velocity engine dealing with multiple animating sprite objects. When Flash becomes more friendly to three dimensional space (3D) I intend to upgrade *Animatem* to perform like a regular

[7] As mentioned previously in "Chapter 4 Why Flash Platform?—Anomalies of Flash" page 22, Flash is not designed to perform this style of looped and timed animation within a single function. *Animatem* uses on an update() call to get around this issue.

velocity engine, with 3D objects in 3D space.

The crucial step in translating *Animatem* over to Flash was finding a comparable call that would create a graphic on the fly and reference it to a location on the screen. Sample syntax for this operation came to me by way of the book *Flash MX 2004 Game Design Demystified* by Jobe Maker, on page 188, where map objects are attached to a Grid Object using the `attachMovie()` *ActionScript* command. With this command it was entirely possible to convert the whole *Animatem* engine from *OpenGL* and *C++* over to *Flash ActionScript*, much of it verbatim in syntax, as *Actionscript* uses a *C* style syntax much like *Javascript*.

Class: Animatem (for a full listing see "Appendix E—Animatem.as" page 320)

Description:

The Animatem object oversees multiple animating sprite objects to yield solid, smooth, fractional of a pixel animation. Each sprite has various properties such as velocity, friction, fps for cell animation, and collision detection. All of these properties need to be frequently updated for smooth animation. To do this Animatem maintains an `update()` method that is called as frequently as possible by the programs main loop.

The `update()` method simply records the amount of time passed since the last `update()`, and instructs each sprite being controlled to update according to such time passed. Time is measured in ticks, which are 1/60ths of a second, which was deemed a manageable yet fine enough granularly. For instance, a sprite with a velocity of 0.5 pixels per tick along the x plane receives an update from Animatem informing it that it has been three ticks since the last update. The Sprite then knows to draw itself (0.5 * 3 =) 1.5 pixels further along the x axis. Sprites can flexibly animate simultaneously and interactively, and each respond to each other and the changing environment. Velocities and other parameters can be instantly changed at the will of the environment, often according to highly randomized factors.

Flash 8 demands that all movies to play at the same constant speed (FPS) as the first root movie. It is for this reason that Animatem works best with the root movie set with a high FPS. Individual animating Sprite objects may then each then be assigned their own slower FPS. This gives *Animatem* much more animation flexibility and power than inherently present Flash, as animations can be played at different and precise rates, and even backwards.

Methods:

Animatem()	Constructor.
update()	Called by the main loop of the program. Updates the current _updateTime and determines the amount of time passed since the last update. This position accordingly.
addSprite()	Adds a new sprite to the animator, as a specific channel is not specified the next available channel is used.
addSpriteN()	As for addSprite() but a specific channel n is specified.
setSprite()	As for addSprite(), but an actual MovieClip is given instead of a link name.
setSpriteN()	As for setSprite(), but a specific channel n is specified.
clearAllSprites()	Empties the animator of all sprites. This is often useful when an environment resets.
releaseUpdate()	When an excessive amount of time passes between updates it is necessary to ignore the break, otherwise a very visible jump is seen in the animation. By setting the _updatePrev to 0 the animator knows to do the next update with a minimal time_passed value.
clearSprite()	Clears a specific sprite from the animator, removing it from both the _sprites list and the _spriteList and deleting its attached movie from the main movieclip.
reserve()	Specifies a sprite channel as reserved and not to be recycled.
notReserved()	Checks if a channel is marked as reserved.
collision()	Passes collision messages to controlling object.
deactivated()	Passes deactivated messages to controlling object.
goToLocAtSpd()	Animate sprite to a point at set speed.
goToLocInTme()	Animate sprite to point in set time.
rotateInTime()	Rotate sprite to an angle in set time.
scaleInTime()	Scale sprite to a size in a specific time.
addDropShadow()	Adds a drop shadow filter to an animating sprite.
addBevel()	Adds a bevel filter to an animating sprite.
addGlow()	Adds a glow filter to an animating sprite.
removeFilter()	Removes the last applied filter.

Notes: Although Animatem has many accessors and mutators to manipulate a Sprite's parameters, referencing into the Sprite by its channel number, generally it's better to simply get an object reference to the Sprite and access it directly. The only important sprite mutator that must be called via the Animatem Object is `setClip()`.

Class: Sprite (for a full listing see “Appendix E—Sprite.as” page 334)

Description:

The Sprite object contains numerous member properties that dictate how a sprite should behave when animating. The `update()` method called by *Animatem* is divided into five procedures for readability, 1) physics, 2) cycles, 3) walls, 4) collisions, and, 5) drawing. The whole object is really one long update method, utilizing a large set of member variables that determine behavior. Accessors and mutators are provided for all variables to give easy direct access and allow highly interactive manipulation.

Method:

Sprite()	Constructor.
update()	Do/calculate everything needed to update this sprite.
doPhysics()	Deal with physics on the sprite, specifically, friction, maximum velocity, velocity, scaling velocity, angular velocity, destination points, and termination time (time to live).
doCycles()	Deal with cycling frames. Sprites can animate through fame "cells" forward and backwards and at a rate specified by _tPerFrame which is really a specified Frames Per Second (FPS). This is a powerful feature as traditionally Flash restricts playing of movie frames to strictly forward and only at the FPS defined by the whole movie.
doWalls()	Deal with sprite boundaries ("walls"). The sprite has a display Rect that it should animate within, if it exceeds this Rect then something should happen, as defined by _wallType.
doCollisions()	Deal with collision detection buy searching through the _collisionList for sprites flagged as a collision hazard.
doDraw()	Once all factors have been taken into account, tell Flash to actually draw the sprite at a location with a rotation, using a scale, at at a specific frame.
clearCollisions()	Clear all collisions from this sprite. Useful after a sprite has fulfilled its purpose.
deactivate()	Sprite has triggered a deactivation, but cannot deactivate unless all triggers are satisfied.

PlaySnd Object

As opposed to the Animatem engine, PlaySnd was the last object to be built, as sound features were added last to the project. Although Flash has good inbuilt sound capability there is one feature in particular that needed to be supplied for *Chevalier* that was not featured. This is the ability to easily stagger and delay sound rather than having it play immediately when called.

When moving large groups of Elements simultaneously I needed it to sound like a multitude moving all at once. To achieve this purpose each Element Type has assigned to it three slightly different versions of its normal walking sound. Whenever an element moves one of the three walk sounds is chosen randomly, and the trigger time of the walk sound is delayed, or rather staggered, by a small random amount. When one element moves by itself the effect is not noticed, but when twenty elements move together the desired sound of a multitude is produced.

Because Flash triggers all events called within a method as soon as it exits that method, attempting to use some form of looped "wait" statement between triggering sounds, in a traditional manner, is highly ineffective. The looped wait causes a long accumulated pause, and then, once the method has finished, all the sounds trigger precisely at the same time. The end result is generally a slightly louder overall sound but there is no staggering effect.

PlaySnd overcomes this by having a list `_cue` that stores all sounds designated to be played with a time delay. *Chevalier's* main loop contains a call to PlaySnd's `update()` method, much like with *Animatem*, but in this case `update()` simply checks each sound in the `_cue` for a specified time to play, and, if that time is due, the associated sound is triggered and the sound removed from the `_cue`.

Granted, this timed staggering is the primary function of PlaySnd, but there are some other small benefits that PlaySnd delivers. When sounds are played using the timed `play()` method they can also be given a volume percentage to play at. A very useful property for tweaking sound effects and especially useful in the instance where elements are eliminated by shooting, as, by lowering the volume if the death cry, the sound is muffled so it seems occurring at a distance.

The other advantage of PlaySnd is that sounds may easily be specified to loop. Flash does have a looping capability inbuilt, but the inbuilt capability does not work for streaming audio, such as in the case of background music. As PlaySnd's `update()` is constantly called by *Chevalier's* main loop this update can also be used to check if a sound has finished playing. If a sound designated as looping has finished, i.e. the sounds play position exceeds its duration, then the sound is simply triggered again. This works regardless if the sound is a streaming sound or otherwise. All audio used in *Chevalier* was acquired from Sounddogs.com who have very reasonable rates for usage of their audio files, see www.sounddogs.com.

Class: PlaySnd (for a full listing see “Appendix E—PlaySnd.as” page 347)

Description:

PlaySnd uses an `update()` method called by the program’s main loop. PlaySnd maintains a list of loaded Sound objects ready to be triggered at any time. The list of sounds to be loaded are specified as a parameter in PlaySnd’s constructor. PlaySnd’s main function is to allow for delayed and staggered sounds, a feature not supplied by Flash. Looping sounds are also catered for as well as specific sound volume.

Methods:

PlaySnd()	Constructor.
play()	Sets a sound to play using a specific volume, delay and stagger.
loop()	Sets a sound to loop.
update()	Called by the main loop of the program, monitors sound progress.
triggerSound()	Triggers sound at a volume.

Matrices and Grids

Each player's army in *Chevalier* ranges in size from between 80 to 100 Elements. Each of these elements must be able to detect for each other, detect the mouse location, detect being clicked pressed, and detect being rolled by the mouse. Flash does have inbuilt methods for press and roll that can be attached to each of the Elements, but if these methods are applied to all 200 or so graphic elements the result is slow computer response as it must constantly listen to all 200 Elements for various responses.

The solution was to build a Grid object that divides the playing area into uniform square cells, each of which can contain a reference to an Element Object donating that the cell is occupied by that Element. By a process of simple division the mouse x and y position can be translated into a cell position. By checking the corresponding cell reference one can instantly detect what Element is under the mouse being pressed or rolled. Similarly, if it is needed to be known what Element is adjacent to a specific element one can simply check the nearby grid cells for Element references. The Grid greatly simplifies detection of Elements and keeps manageable information on which Element is where without having to constantly listen to all instances.

The two most crucial member variables associated with the Grid object are `_block_w` and `_block_h` which define the height and width in pixels of each individual cell. In the case of *Chevalier* the height and width is the same (square cells). Other member variables are the width and height of the grid (`_w` and `_h`), the top left point of the grid (`_tpLft`) and the overall resultant rectangle of the grid (`_rect`).

The *Chevalier* map was initially designed with a cell resolution of 4 pixels x 4 pixels, but, through play testing it soon became evident that Elements were going to have to be larger as a distinct visual icon was needed to distinguish exact Element type.

Adding the many icons, and getting those icons to work visually was no small feat, but getting the code to adopt to the larger cell size was easy, the map instance was simply defined as having cell sizes doubled to 8 x 8 pixels.

The Grid object extends an object called MMatrix used for storing 2D data in matrices and allowing them to be easily manipulated by other matrices or numbers with the use of add, subtract, divide, multiply, and matrix multiply operations. MMatrix was originally written in C++ for the class “CSCI E-124 Algorithms and Data Structures” taught by Michael Mitzenmacher. Unfortunately Flash does not support operator or method overloading, so the effectiveness of this class in Flash is greatly weakened.

Class: MMatrix (for a full listing see “Appendix E—MMatrix.as” page 351)

Description:

Base class used to store 2D information as a matrix. For the convenience of matrix multiplication, data is usually broken into columns first then rows.

Methods:

MMatrix()	Constructs MMatrix to s et width and height using Array data. Data is broken into columns first, and then rows.
congruent()	Tests to see if another matrix is congruent with this one. Matrices are congruent if their height and width are the same.
equal()	Tests to see if data in other matrix is the same as the data in this one. Matrices must be be congruent to be equal.
add()	Adds two matrices together or, if val is a Number, adds that, Number to each cell in this MMatrix.
subtract()	Subtracts a MMatrix from this MMatrix or, if val is a Number, subtracts that Number from each cell in this MMatrix.
multiply()	Multiplies two matrices together or, if val is a Number, multiplies that Number from each cell in this MMatrix. If the width of this matrix equals the height of the other matrix, then true matrix multiplication is performed.
mMultiply()	Performs true matrix multiplication, to do this the number of columns in this Matrix must equal the number of rows in other.
divide()	Divides this matrix by another or, if val is a Number, divides that Number from each cell in this MMatrix.
print()	Prints the rows and columns of this MMatrix.

Notes: Flash does not have method or operator overloading which is why the matrix operations accept an undefined value, that value is expected to be either another MMatrix or a Number.

Class: Grid (for a full listing see "Appendix E—Grid.as" page 357)

Description:

Grid object extends MMatrix adding functionality to define cell blocks with a width and height that can be referenced to screen coordinates and vice versa. This is very useful for implementing game boards and terrain.

Methods:

Grid()	Constructor.
ptToGridLoc()	Converts a point location, usually a screen location, to a grid cell location.
gridToPtLoc()	Convert a grid cell location to a point coordinate.

Point2D and Rect

Much of *Chevalier's* data is stored as clusters of points (x, y) and rectangles (x1, y1, x2, y2) and very frequently those points are required to be rotated by an angle or operations done such as checking if a point is within a specific rectangle. Flash does have its own Point and Rectangle class, but no methods are supplied in these for operations such as adding, subtracting, multiplying, dividing or rotating points. The Flash Rectangle class measures the rectangle from the center and defines it using the width, height and an angle. In my case it was more convenient to define the rectangle by a top left point and a bottom right point. As such, I wrote my own point class, called Point2D, and my own rectangle class, called Rect.

Initially the Point2D, and Rect classes, like the Grid class, were built extending the MMatrix class. This was convenient as these classes then inherited all the operations already available to MMatrix, in particular calculation functions and the matrix multiplication function, which greatly facilitated rotation of Point2Ds. However, the majority data type in *Chevalier* is a Point2D, and these many instances of Point2D are frequently re-created and cloned, resulting in many calls to its constructor. When Point2D extends MMatrix a `super` call is required in the constructor to create the parent MMatrix object with all its additional code, doubling the time it takes to construct a Point2D. It was found that construction of a lightweight version of Point2D, without it extending MMatrix resulted in a considerable performance boost.

Class: Point2D (for a full listing see “Appendix E—Point2D.as” 361)

Description:

Used to describe a point location in two dimensional space. Has useful Point2D to Point2D operations and includes distance, rounding and rotation methods.

Methods:

Point2D()	Constructs Point2D, defining x and y.
equal()	Test if another point has the same x and y as this one.
add()	Adds two Point2Ds together or, if val is a Number, adds that Number to both x and y in this Point2D.
subtract()	Subtracts a Point2D from this Point2D or, if val is a Number, subtracts that Number from botl x and y in this Point2D.
multiply()	Multiplies two Point2Ds together or, if val is a Number, multiplies that Number to both x and y in this Point2D.
divide()	Divides this Point2D by another or, if val is a Number, divides both x and y by that Number.
round()	Rounds both x and y values. This is often needed as Flash sometimes keeps decimal places but without displaying them when a trace() call is used.
distance()	Calculates the distance between this point and another.
rotate()	Rotates this point around (0, 0) by an angle.
limit()	Ensures neither x nor y are ever greater than limiting value.
clone()	Clones this Point2D, making a safe copy than can be manipulated.
print()	Prints this Point2D.

Class: Rect (for a full listing see “Appendix E—Rect.as” page 366)

Description:

Used to describe a rectangle in two dimensional space as described by a top left point and a bottom right. Has useful operations to detect intersections with other Rects and if a Point2D is inside the area bounded by the Rect.

Methods:

Rect()	Constructor.
equal()	Test if another Rect has the same x1, y1, x2, y2 as this one.
add()	Adds two Rects together or, if val is a Number, adds that Number to all four points in this Rect.
subtract()	Subtracts a Rect from this Rect or, if val is a Number, subtracts that Number from all four points in this Rect.
multiply()	Multiplies two Rects together or, if val is a Number, multiplies that Number to all four points in this Rect.
divide()	Divides this Rect by another or, if val is a Number, divides all four points by that Number.
round()	Rounds all four values. This is often needed as a Flash bug keeps decimal places but without always displaying them when a trace() call is used.
clone()	Clones this Rect, making a safe copy than can be manipulated.
inside()	Tests if a point is inside this rect.
intersect()	Tests if val Rect intersects with this Rect.
between()	Tests if val is between a and b.
print()	Prints the four points for this Rect.

General Utilities and XML Reader

The Utils object contains a number of static general purpose functions used throughout *Chevalier,* the most important of which is a very generic XML parser for easily disseminating information from a XML file, called `setXMLreader()`. The most commonly used utility functions are `randomInt()` and `cleanAngle()`, the first simply returns a randomized integer while the second ensures that numbers that are used to specify degrees of an angle maintain a value between 0 and 360.

The `setXMLreader()` method for reading XML accepts a Flash XML object, and a target object name and method name where each clump of parsed XML data is to be sent for interpretation. Specifically, `setXMLreader()` attaches a parsing function "reader" to the `onLoad()` method of the XML object so that, when an XML file is read in, it is automatically parsed using the parsing function. This function has been kept very generic. It iterates through all the items in the first child's XML node list looking for attributes. When an attribute is found it is bundled as a name value pair and added to an anonymous object. Once all attributes for the node have been added, the resultant anonymous object is sent to the global object method specified as parameters of the `setXMLreader()`. For example,

If the XML data looks like so,

```
<?xml version="1.0" encoding="UTF-8"?>
<army xmlns="http://mocaz.com/_Armies.xsd">
      <element>
            <command>Left</command>
            <type>Kn</type>
            <grade>Ordinary</grade>
            <regular>true</regular>
            <name>Thomas Camoys</name>
            <icon>Chevalier</icon>
            <locX>104</locX>
            <locY>173</locY>
            <angle>270</angle>
```

```
      <general>Sub-gen</general>
</element>
<element>
      <command>Left</command>
      <type>Pk</type>
      <grade>Fast</grade>
      <regular>false</regular>
      <name>Brigans</name>
      <icon>Swords</icon>
      <locX>48</locX>
      <locY>159</locY>
      <angle>270</angle>
</element>
```

Each `<element>` node will get individually wrapped in an anonymous object and sent to the specified global object method, in this case the `addElement()` method of Chevalier, where dot syntax is used to simply extract the instance data from the passed object.

```
public function addElement(obj:Object) {
      var _cmd:String         = obj.command;
      var _type:String        = obj.type;
      var _grade:String       = obj.grade;
      var _regular:Boolean    = obj.regular;
      var _name:String        = obj.name;
      var _icon:String        = obj.icon;
      var _loc:Point2D        = new Point2D(obj.locX, obj.locY);
      var _angle:Number       = obj.angle;
```

This style of variance programming, although perhaps slightly heavy, is extremely flexible and powerful. In this way the `setXMLreader()` can be used for any XML data that is one level deep and has values passed as attributes on its nodes. Information on defining, reading and parsing XML data in Flash was gleaned from *Flash and XML, A Developer's Guide* by Dov Jacobson, and *XML 101*, at www.actionscript.org/tutorials/intermediate/XML/index.shtml.

Class: Utils (for a full listing see “Appendix E—Utils.as” page 372)

Description: General purpose static utility functions.

Methods:

setXMLreader()	Attach generic XML reader to an XML object, triggered by onLoad.
randomInt()	Generate a random integer from "low" value to "high" value.
cleanAngle()	Clean angle variable so it ranges between 0 and 359.
makeRect()	Given an array of points give general rectangle.
compareAngle()	Give the difference between two angles.
smallerOfTwoPts()	Return the smaller of two points.
isACloser()	Return true if ptA is closer to origin than ptB.
parity()	Determine even/odd parity.
FPS_to_Ticks()	Convert frames per second to ticks (1/60ths of a second) per frame.
ptToAngle()	Return an angle in degrees given a velocity vector.
angleToPt()	Return a velocity vector as a Point2D when given an angle.
angleToCell()	Convert an angle (in degrees) to an animation cell number where a facing of 0 degrees (which is an East facing, or rather facing the right screen edge) will yield the first cell number, while a 359 degree facing will yield the last cell.
incOrDec()	Given a looping sequence in both directions with a range of max, if at n which direction (+ or) is quickest to get to destination.
radiansToDegrees()	Convert radians to degrees.
degreesToRadians()	Convert degrees to radians.

Known Bugs

- It should always cost to wheel or retreat Elements unless moving a single Element.
- On occasion fleeing elements get stuck on elements behind.
- After an army has two commands break a “VICTORY!” screen should be displayed for the other player.
- When wheeling a group of elements where the front-most Element is not flush with the others the Elements will end in disarray.
- The “expand left” and “expand right” movement controls are disabled as they are not fully implemented.
- All “shattered” troops should automatically flee from the battlefield.
- Terrain has not been fully implemented. All areas are currently considered as “clear” for purposes of game play.
- Long element names should display in a smaller font size, for instance names of commanders such as “Guy de Lusigan” don’t fit and so should be a point size smaller.
- In some movement instances the program falls into an eternal loop when calculating legal moves.

Chapter 7 Usability Testing

An important part of Game Design is usability testing. Some games evolve dramatically and unexpectedly during testing and development, literally "growing" during prototyping. Many highly successful and classic games such as *Pac Man*, *Space Invaders*, *Centipede*, and *Tetris* were all the product of testing, refining, and programmer tinkering. It is for this reason that rapid prototyping is so essential and why attempting to design a game in its entirety from scratch on paper is often found to be frustrating and unsuccessful. A balance between prototyping, testing, and conceptual design needs to be used (Apple Computer, Inc. 1992).

Aside from the Presentation Objects such as the *Animatem* Engine, *Chevalier* was developed over a reasonably short period of time, from mid November 2005 to the end of March 2006. All major development was frozen on April 1st, ironically the same day that Games Workshop announced its acquisition of rights to *DBA*. Over the period of four and a half months that *Chevalier* was programmed, limited though essential, usability testing was conducted, resulting in many unexpected and vital issues being faced and resolved.

Originally I designed *Chevalier* using smaller game pieces which were to be identified by their base size, "googling" over the Element pieces (see "Movement Phase—Googling Elements" page 96), and a rudimentary military symbol. The military symbols, such as a diagonal line across the unit to denote cavalry, or an "X" to denote infantry, are familiar to the audience of wargamers and were frequently used in game simulations published in the 1970s by SPI and Avalon Hill. Testing quickly revealed these basic symbols were not enough.

When an early version of *Chevalier* was shown to members of a wargames group in Rhode Island, the immediate response was that they could not decipher troop types and clearer identification was needed on the game pieces. The solution was to double the size of the pieces and the map board and to add a distinctive and clear icon for every Element Type. This proved a daunting task as there are 17 element Types (see Figure 8.6). A frantic rush ensued searching for appropriate icons to prototype and test, and dramatic changes where implemented in the code to cater for larger pieces. The cell size for game squares, as defined in the grid object (see "Chapter 6 Application Design—Matrices and Grids" page 66), became 8 pixels instead of 4 pixels, after much testing of various other possible alternatives such as 10, 6, and 12 pixel cells. The timing of all the movement had to be corrected as suddenly everything took twice as long to reach its destination as it had to move twice as far, and the workload for the graphic artist Peter Gifford was substantially increased as he was asked to additionally create a substantial icon set of 17 images that must all be clear and functional at various map resolutions. Introducing the new icons took a week and a half of development time and a substantial portion of Peter's limited time, but the end result was a marked improvement in game play.

The *Chevalier* pieces were originally blue and red, like in *Stratego,* rather than black and white, as in *Chess.* The blue and red colors were found to be difficult to view, lacking color contrast, and experienced *DBM* players were discouraged by the obvious association with *Stratego,* a mainstream and comparatively simple game, released by Milton Bradley, a gaming company that some of the testers did not associate with serious simulation. I preferred the use of blue and red as there was no negative connotation labeling one side as "black" or another as "white"— particularly as there are very sensitive historical issues involved with some of the scenarios. Peter resolved the issue by

creating icons that are “white on black,” and “black on white,” so neither side can be labeled as “black” or “white,” rather an “either or” situation arises.

Testing also revealed that players wanted to differentiate between Crossbows and Bows, even though functionally the *DBM* game does not discriminate between the two in terms of Type. As such, a new icon and Type was made for Crossbows. See “Chapter 3 Why *DBMM*?—Anomalies of *DBMM*” page 14 for more information on this.

The Playing pieces were found to start too far away from each other. This is also a problem in the actual *DBM* game, with many players desiring a fast start and Elements coming into contact almost immediately, though this reduces much of the game strategy. Players completely unfamiliar with the game system where reporting that, after playing a turn of *Chevalier,* “nothing happened” or that they did not see the other player and had no idea what they were meant to do, usually getting immediately discouraged after the first turn. As such, I moved all the units considerably closer to each other, in some instances being in bow shot on the first turn.

It was immediately apparent to players of *DBM* that the “fiddlyness” of the *DBM* system had been successfully taken out of the game (see “Chapter 3 Why *DBMM*?—Anomalies of *DBMM*” page 14), though more responsiveness was requested in selecting groups. Later it became evident that the majority of players kept their map viewed at 70%, and as such the default setting was changed to 70%, although the other map scales are useful, and 100% is still the most appropriate map scale for close combat.

Although there was very limited play testing done on *Chevalier,* what was done had a significant impact on the development of the game. In truth, there is considerable testing still to perform, particularly now that there is a User Guide available and final graphics instead of prototypes implemented.

Chapter 8 User Guide

Credit and Copyright

Design and Programming: Russell Lowke

Graphics: Peter Gifford

Terrain Maps: Ralf Schemmann

Title Art: Igor Dzis

Figure Painting: Siam Painting Services and Alister Lowke

Figure Photography: Tamara Bonn

Usability Testing: Tamara Bonn, Alister Lowke, William Belford

Special Thanks to: Scott Traylor, Henry H. Leitner, William B. Robinson, Bruce Molay, Hanspeter Pfister, Kenneth J. Basye, Yair Leviel, Jordan Bach, Billy Belfield, Christopher S. LaRoche, Andrew Jinks, and John Sharples.

Introduction

Chevalier is a two-player online game that is an adaptation of the tactical tabletop miniatures wargame *De Bellis Magistrorum Militum* (*DBMM*). *Chevalier* is intended to simulate ancient and medieval battles covering the pre-gunpowder period from 3000 BCE to 1500 CE. As turning points in history are often decided by combat and battles, simulations offered by *Chevalier* are ideal as an online teaching aid for history education, allowing players to experience and experiment first hand the various battle tactics used by each side. *Chevalier* is intended to be used as a module to augment an educational Website specific to a historical period, allowing that site to simulate and replay key historical battles it discusses. *Chevalier* introduces a new form of history involving the student in real history, placing in the hands of the student key battles of the ancient and medieval world.

The full *DBMM* tabletop miniatures rules can be found online at www.phil-barker.pwp.blueyonder.co.uk/DBMM.doc. At the time of this writing, the rules are still under development and, once published, they will no longer be available online.

Getting Started

Chevalier is an online game that can be found at www.mocaz.com/games/Chevalier.html.

Since *Chevalier* is written in *Flash 8,* it requires the *Flash 8 Player* to be installed in your browser. If you do not have *Flash 8 Player* installed when you arrive at the the *Chevalier* Website there will be an empty page with the link, "Get Flash 8 Player." Clicking on this link will take you to www.macromedia.com/go/getflash/ where the player can be downloaded and installed. Once you have *Flash 8 Player* installed, revisit the *Chevalier* site where the game will start loading and you will be presented with a splash screen featuring credits and a title image of a Knights Hospitaller of the Crusades at the battle of Arsuf.

Selecting a Battle Scenario

When the game has finished loading you will be presented with a window titled "Choose a Battle" (Figure 8.0) with a selection of four buttons. The first three buttons refer to the three possible historic battle scenarios that can be played, "Arsuf", "Gaugamela", and "Agincourt", while the last button allows selection of a fictional combat between any two of the six armies involved in the three battle scenarios.

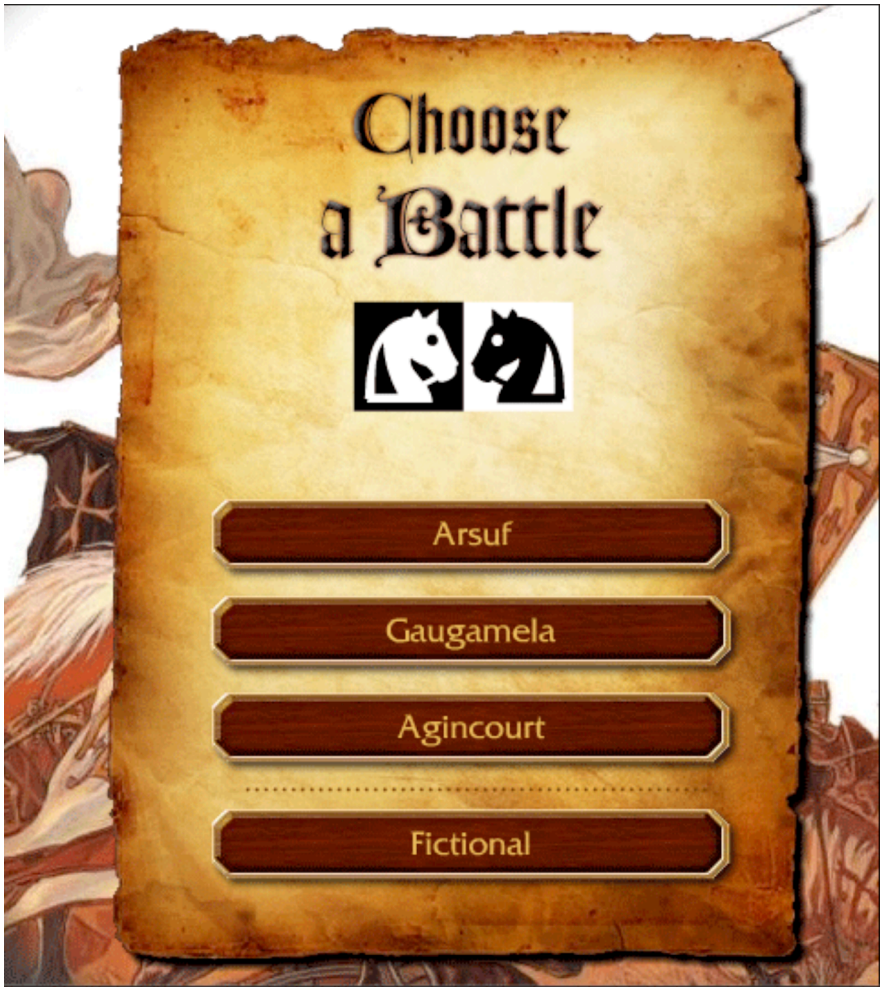

Figure 8.0 - Choose Battle Window

Rolling over each of the buttons will give a short description of the battle scenario pertaining to the button. For the purposes of this tutorial select the first button, "Arsuf."

Battle Introduction

You will then be presented with a short description of the chosen battle (Figure 8.1). If, after reading the battle description you are not interested in that particular battle scenario, you may return to the previous screen by selecting the backwards button in the top left corner of the window. This button can be identified by the left pointing arrow inside the button.

Figure 8.1 - Battle Introduction

In this case we see the introduction for the battle of Arsuf, where Richard I was marching along the coast to the city of Joppa when his progress was impeded by his adversary Saladin, whose troops began harassing the Crusader column. For the battle descriptions for each scenario see, “Appendix C Battle Scenarios” page 122.

To accept this scenario press the **OK** button in the lower right corner.

How to Play Chevalier

Chevalier is a turn based strategy game where play is in alternate player turns, the alternating "your turn, my turn" reflecting action and response on the battlefield. These turns are not representative of fixed and arbitrary divisions of time but rather initiatives and responses by the two sides. However, dividing known battle durations by the number of phases produces a rough estimate that a pair of turns is equivalent to about 20 minutes in real life (Barker, 2006). Each player turn has three phases, 1) Movement, 2) Shooting, and 3) Combat, with the beginning of a turn identified by a "Start Turn" window.

Start Turn

In the Arsuf scenario, the Crusader player goes first, followed by the Saracen player. The "Start Turn" window will appear immediately after accepting the battle scenario, showing that it is the first turn ("#1") and that it is the Crusader player's turn, as indicated by the word **Crusader** in bold and a silhouette of the Crusader insignia shadowed across the window (Figure 8.2). For a key to insignias see Figure 8.3.

Figure 8.2 - Start Turn Window for Crusader Player

Select the **Begin** button of this window to start the Crusader player turn.

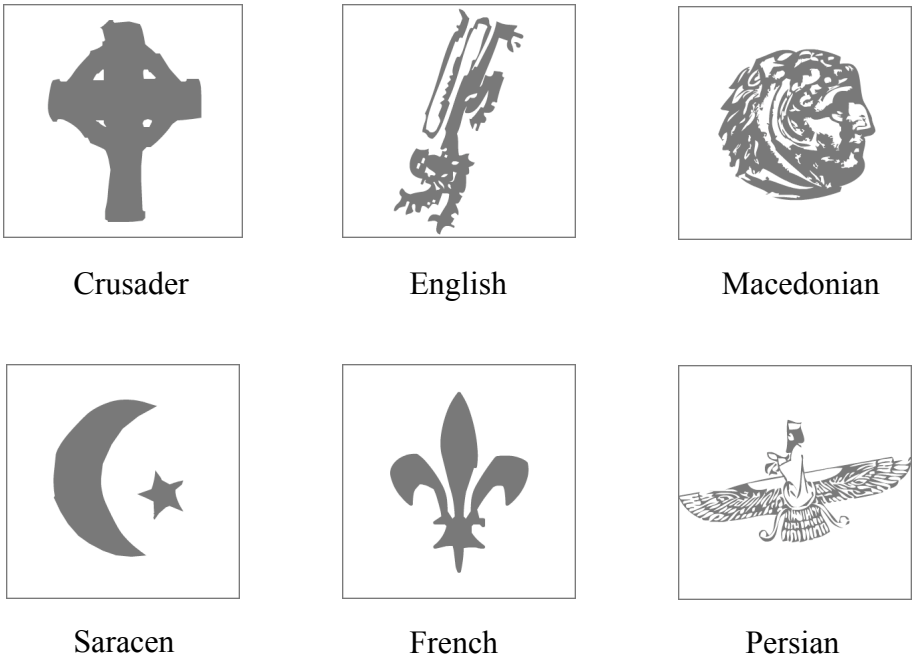

Figure 8.3 - Army Insignia

How to Interpret the Screen

At the beginning of a player's movement phase the map will zoom and rotate to wherever the player last left the map. At the beginning of the game each player's map is set to be viewed at 70% scaling from directly above and over the middle of battlefield. You can tell you are viewing the map at 70% as the Control Palette (Figure 8.4) says "70%" under the "Zoom" category. The control palette also displays the current turn number and the insignia for the player who is having their turn on the left. The Control Palette will be discussed in greater depth in the next and subsequent chapters.

Figure 8.4 - Control Palette

For an image of what the map looks like at the beginning of the Crusader's first movement phase see Figure 8.5.

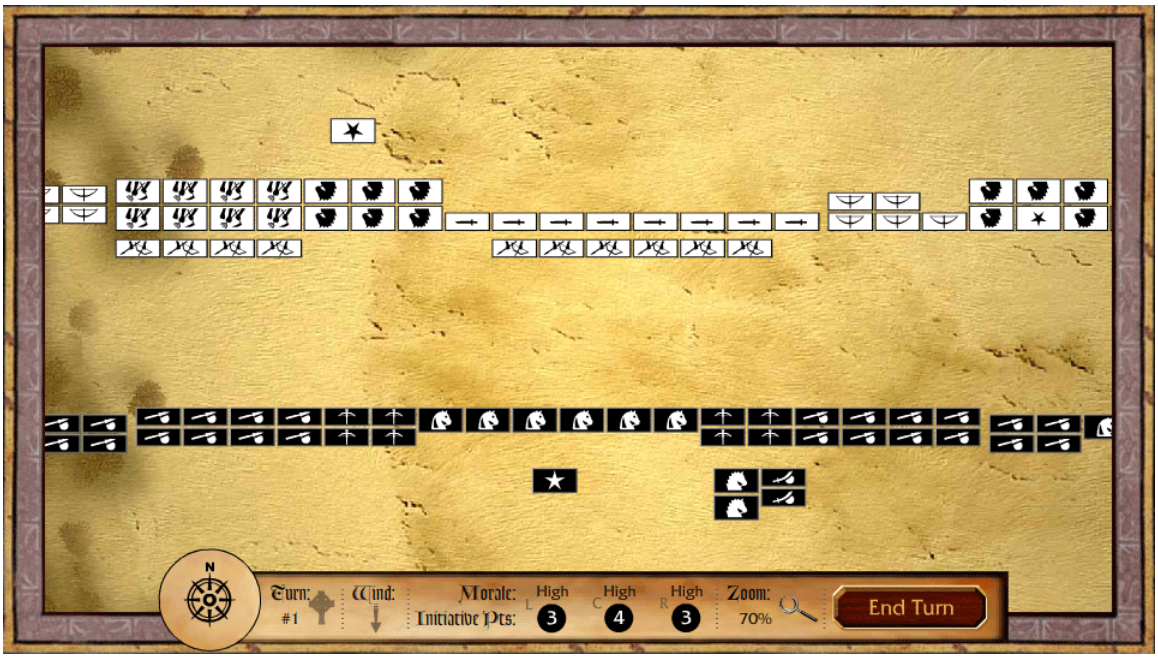

Figure 8.5 - Movement Phase

This opening map shows the Crusaders player's pieces in the lower portion of the map facing the enemy, much like in a game of chess. In *Chevalier* the individual playing pieces for a player's army are called elements. An element represents the smallest sub-unit of the army capable of operating independently. In the Arsuf scenario the Crusader elements are recognizable as white icons on black rectangular bases, while the Saracen elements are black icons on white rectangular bases. Each Element base depth is one of three different depth sizes. Infantry are represented by the thinnest base depth, followed by cavalry which use the medium base depth. Other special units, such as elephants and baggage, are represented by the square base which is the third and deepest base depth.

Each element has a type, which is represented on the map by a specific and immediately identifiable icon displayed on the Element base. For a key to all icon types, and a description of what those types represent, see Figure 8.6.

Scale

Each element represents approximately two-hundred to two-hundred and sixty infantry, or one-hundred and thirty to two hundred riders, or about sixteen elephants, the exact numbers vary depending on the army. An element of Hordes is an exception, representing up to 1,000 men in a deep mass. Similarly, Hordes are represented on a medium base size rather than a thin base even though they are infantry, as hordes tend to swarm in a large group rather than be in an organized block.

All elements have the same width, which is considered to be 80 paces. A pace in *DBMM* is 0.75 meters or 2.5 feet, which is the length of a mans stride. 2000 paces is 1 Roman mile, so the width of an element is approximately 60 meters. For more information on scales, see "Appendix B Units of Scale" page 121.

Navigating the Map

The whole map when viewed at 70% is much larger than what visually fits into the *Chevalier* game window. To scroll around the map and view the rest of the battlefield roll the cursor over the framed edges of the *Chevalier* game screen, thereby moving the map in the direction of the frame edge. If you find the scrolling too slow you can press and hold the mouse button "leaning" on the frame edge and causing the map to move at twice the speed. You can also use the number keys on the keypad to scroll the map in the corresponding direction. Holding down the spacebar will cause the cursor to change to a grab hand. If you click and drag the mouse while the space bar is held down you can drag the map.[8] Initially, the Arsuf battlefield appears to be an open plain as you can only see the center of the map. If you scroll to the right you will discover the Mediterranean coast

[8] Much like in most paint packages, such as Adobe *Photoshop.*

and water, accompanied by a road. If you scroll to the left you will find hills and dense trees and other rough terrain.

To zoom the map out click on the bottom half of the magnifying glass icon on the Control Palette (Figure 8.4). When you roll over this lower portion if the magnifying icon a "-" symbol will appear over the icon to indicate you are about to zoom out. Try it, click on the lower potion of the icon, you will see that the map zooms out to 50%. Now you can see more of the battlefield in one glance and most of your army. Zoom out again and you will be at 35%, giving you a extensive view of the battlefield. At this scale you can fit the whole width of the battlefield on the screen. Zoom out once more to 28%. This is the furthest you can zoom out, fitting the entire battlefield on the screen. Click on the upper portion of the magnifying glass icon to zoom into the map. You will see that when you roll over the upper potion of the zoom icon a "+" appears to indicate that you are about to zoom in. The zoom controls will allow you to zoom the map to the scales of 100% and 125%. As the keyboard can be used to scroll the map it can also be used to zoom it. Use the down arrow key to zoom the map out and the up arrow key to zoom in.

The map can also be rotated, which can be especially helpful for selecting groups of elements that are on an angle. Clicking on the inside portion of the compass graphic in the Control Palette (Figure 8.4) will rotate the map. The right inside half will rotate the map counter clockwise 45 degrees, while the left half will rotate the map clockwise 45 degrees. Like with the zoom controls, the keyboard may also be used for rotation. The left arrow rotates clockwise and the right arrow rotates counter clockwise, much like controls for a plane in a flight simulator. To rotate the map to a specific angle click on the outside edge of the compass graphic just beyond the compass points. The Map facing will spin immediately to the edge you select.

Element Troop Types

KNIGHTS (Kn), representing all those noble or heavy horsemen of high morale that charge at first instance without shooting, with the intention of breaking through and destroying the enemy by sheer weight and impetus. The impetuous charge that enables knights to sweep away lesser cavalry and all but the stoutest of foot is also their Achilles' heel, leading to dangerously rash pursuit.

Movement: 200 paces, *Combat v Mounted:* +3, *v Others:* +4
Quick Kill: Hd, Sk, LI, Cb, Ar, Wb*, Sw*, Pk*, Sp, Cv*

CAVALRY (Cv)**,** representing the majority of ancient horsemen, usually at least partially armored, combining or following close combat shooting with controlled charges. Being less impetuous, cavalry can retire out of danger or to breathe their horses when knights would charge on to disaster.

Movement: 240 paces, *Combat v Mounted:* +3, *v Others:* +4
Quick Kill: Sk, Cb, Ar

LIGHT HORSE (LH)**,** especially swift riders who scout or usually fight in a loose swarm with missiles rather than in formation and often gaining extra mobility from multiple remounts. A nuisance in small numbers, they become a menace in dense swarms, especially to foot that must endure without effective reply.

Movement: 320 paces, *Combat v Mounted:* +2, *v Others: +3*
Quick Kill: Sk, Cb, Bw, El

SWORDS (Sw)**,** including all close fighting infantry primarily skilled in fencing individually with swords or heavier cutting or cut-and-thrust weapons, sometimes supplemented by hand-hurled missiles or bows.

Movement: 160 paces, *Combat v Mounted:* +4, *v Others:* +4
Quick Kill: Wa*, Pk*, Sp*

SPEARS (Sp)**,** representing all close formation infantry fighting with thrusting spears and heavy shields in a rigid shield wall formation.

Movement: 160 paces, *Combat v Mounted:* +4, *v Others:* +4
Quick Kill: (none)

WARRIORS (Wa)**,** including all infantry that rely on an impetuous and ferocious collective charge to sweep away whole enemy formations, rather than on individual skill.

Movement: 160 paces, *Combat v Mounted:* +3, *v Others:* +3
Quick Kill: Hd, Cb, Bw, Sw*, Pk*, Sp*

PIKES (Pk)**,** including all close formation infantry fighting collectively with pikes or long spears wielded in both hands.

Movement: 160 paces, *Combat v Mounted:* +4, *v Others:* +3
Quick Kill: (none)

LIGHT INFANTRY (LI)**,** representing foot willing to fight hand-to-hand, but emphasizing mobility or fighting in difficult terrain, or against Elephants or Expendables rather than cohesion or aggression.

Movement: 200 paces, *Combat v Mounted:* +3, *v Others:* +3
Quick Kill: (none)

SKIRMISHERS (Sk)**,** including all dispersed infantry shooting individually with javelin, sling, staff sling, bow, crossbow or hand gun, who fight in a loose swarm hanging around enemy foot, running away when charged. They are useful to delay or even damage unsupported heavy infantry.

Movement: 200 paces, *Combat v Mounted:* +2, *v Others:* +2
Quick Kill: El

ARCHERS (Ar)**,** representing foot who fight in formed bodies by shooting collectively with missiles at longer range than Skirmishers, often in volleys at command, and who rely on dense shooting.

Movement: 160 paces, *Range:* 240 paces*, Combat v Mounted:* +4, *v Others: +3*
Quick Kill: (none)

CROSSBOWS (Cb)**,** representing foot who fight in formed bodies by shooting collectively with crossbows at longer range than Skirmishers, often in volleys at command, and who rely on dense shooting.

Movement: 160 paces, *Range:* 240 paces*, Combat v Mounted:* +4, *v Others: +3*
Quick Kill: (none)

SHOT (Sh)**,** including all hand-gunners that fight in ranks. Inaccurate and unreliable, their bullets could penetrate even heavy armor, and the novelty of unprecedented noise and smoke could frighten men as well as animals.

Movement: 160 paces, *Range:* 80 paces*, Combat v Mounted:* +5, *v Others:* +4
are always Quick Killed by others

HORDES (Hd)**,** including all unwilling or incompetent foot, brought to swell numbers and/or to perform menial services, or attracted by desperation, religious or political fanaticism or greed, and too tightly huddled, scared, stupid or indoctrinated to run away.

Movement: 160 paces, *Combat v Mounted:* +2, *v Others: +2*
Quick Kill: (none)

ELEPHANTS (El)**,** of either anciently-domesticated species and various crew complements. They are used to charge solid foot; to break through gateways, and to block mounted troops. They can most easily be killed by artillery or by the continued showers of missiles of Skirmishers or Light Infantry.

Movement: 200 paces, *Combat v Mounted:* +5, *v Others:* +4
Quick Kill: Hd, LI, Ar, Cb, Wa, Sw, Pk, Sp, Kn

EXPENDABLES (Exp)**,** scythed chariots fitted with scythe blades and spear points, usually pulled by 4 horses with a single crewman, intended to be driven into enemy formations in a single suicidal charge early in the battle to break up or destroy them. They are most dangerous to troops offering a solid target that cannot dodge easily, so are often countered by Skirmishers.

Movement: 200 paces, *Combat v Mounted:* +5, *v Others:* +4
Quick Kill: Hd, Wb, Sw, Pk, Sp, Kn

ARTILLERY (Art)**,** whether gunpowder, torsion, tension, counterweight or powered by men pulling ropes.

Movement: 160 paces, *Range:* 480 paces*, Combat if Shooting:* +4, *Otherwise:* +2
Quick Kill: El

BAGGAGE (Bg)**,** representing the army's logistic support, including all personnel, supplies and equipment that increase the physical or mental welfare of troops or generals.

Movement: 0 paces, *Combat v Mounted:* +2, *v Others:* +2
always Quick Killed by others

Figure 8.6 - Element Types

* Quick Killed only during opponents turn

Army Commands

Each army is broken into three commands, “left” command, “right” command, and “center” command. Each and every element in the army is designated to one of the three commands[9] and only elements of the same command may move together in a group (see “Moving Groups of Elements” page 100). Every command has a Commander whose element is represented on the map by a star instead of an element type icon. The Commander-in-Chief (C-in-C) can be identified by the largest star while the Sub-Commanders are the two smaller stars.

Initiative Points

At the beginning of every turn each command is randomly allocated a number of Player Initiative Points (PIPs) between 1 and 6. Initiative Points determine the number of groups of elements that may be moved by a command each turn. **Regular** armies, being those armies that train regularly and are a standing army, benefit by averaging their Initiative Points between all three commands, this greatly reduces the chance of having only 1 Initiative Point to move with, which can be very hazardous. **Clumsy** armies do not average their points, resulting in a wider range of scores. Medieval French are the only **Clumsy** army implemented in *Chevalier*, the other armies are all **Regular**.

Initiative Points awarded to each command can be seen in the three circles in the middle of the Control Palette along the bottom edge of the map (Figure 8.4). The left command is signified with an **L**, the center command with a **C**, and the right command with an **R**. In the example given in Figure 8.4, the Crusader player has been awarded moderate Initiative Points. The center **C** command, governed by the C-in-C, is always

[9] Historically some armies used only two commands, while other armies, particularly cavalry armies, such as the Mongols, used four.

awarded an additional Initiative Point, which is why the center **C** command has 4 Initiative Points while the edge commands each have 3.

As only elements from the same command can move together when selecting an element all the other elements from other commands gray out. To see this effect clearly it is helpful to zoom out so you are looking at most of the army. Selecting elements is a good way to see which elements are in what command. Also, when you select elements from one command the other Initiative Point circles on the control Palette also gray out. For more information on Initiative Points and moving units see, "Moving Groups of Elements" page 100.

Morale

In the Control Palette, above each Initiative Points circle, there is a word describing the morale of each command. Morale is determined by the collective influence of all elements in the command at game start compared with the influence of all remaining elements currently in the command. For information on finding an elements influence, the levels of influence, and the weighting those levels have on morale has see, "Googling Elements—Influence" page 99.

Once one-quarter of the command's total influence is lost to battle the command becomes **Dispirited** which means that its **Normal** and **Low** influence troops are unwilling to engage in battle and will receive a combat penalty when fighting. Once one-third of the commands influence is lost the command becomes **Broken** and all troops become reluctant to engage and receive a combat penalty. Once half of the commands influence is lost the command is considered **Shattered** and all troops in the command will automatically try to flee the battle.

Wind & Rain

On the left side of the Controls Palette to the right of the turn indicator is an indicator labeled **Wind**, which shows the direction and strength the wind is blowing. A strong wind is indicated by a large arrow, as shown in Figure 8.8 and Figure 8.4, which both depict a strong wind blowing south, directly into the player's army. When there is a strong wind you will also hear a faint looping wind sound throughout the turn. Strong winds will give a combat disadvantage to Archers and Crossbowmen when shooting into or across the wind.

A light wind (Figure 8.7) has no effect on combat, although light winds can develop into strong winds. Sometimes there will be overcast weather leading to rain. If the weather is overcast the wind indicator will be replaced by a cloud icon see Figure 8.9, which acts as a warning that it could commence raining in the next or subsequent turn. If it rains the icon becomes a raining thunder cloud icon (Figure 8.10) and you will hear a looping sound of thunder and rain in all subsequent turns until the rain stops. Once it has stopped raining it will not start again. All Archer, Crossbow, Shot and Artillery Element types are disadvantaged by rain and will receive a combat penalty both when shooting and when in combat during rain.

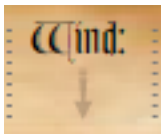

Figure 8.7
Light Wind

Figure 8.8
Strong Wind

Figure 8.9
Overcast

Figure 8.10
Rain & Thunder

How to Win

A player is considered victorious once he succeeds in "Breaking" two of the opponents commands. A command is considered broken once one-third or more of the total influence of the command has been lost to combat. See the "Morale" section above for more information on this. Note however, some historical scenarios have their own specific victory conditions, such as taking a particular objective or keeping certain troops alive.

What Happens each Turn

During each player turn a player should:

- Commence their the Movement phase by selecting **OK** from the Start Turn Window. See "Start Turn" section above.
- Look at Initiative Points awarded for the turn, moral of their troops and current wind direction. See "Player Initiative Points (Initiative Points)", "Morale", and "Wind & Rain" sections above.
- Survey the battlefield, googling at their troops and the enemy troops looking for strengths and weaknesses to exploit. See "Navigating the Map" above and "Googling Elements" page 96.
- The player should then move Elements into strategic positions, shooting range and into battle. When moving Elements the player should always try to move them in groups as expenditure of Initiative Points is more cost effective. See "Moving Groups of Elements" page 100.
- Once all Initiative Points have been expended, or the player is satisfied with their movement decisions, they should end the movement phase by pressing the "End Turn"

button on the right hand side of the Control Palette (Figure 8.4). Doing this concludes the Movement Phase.

- The player will then be presented with a "Shooting" window to conduct distant combat for any Elements able to shoot and that are in range of the enemy. These Elements will be marked by a green glow. If there are no eligible shoots then this phase is skipped. See "Shooting Phase" page 107.
- The player will then be presented with a "Battles" window to conduct close combat for any Elements that are engaged in battle. These Elements will be marked by a burgundy glow. If there are no battles to conduct then this phase is skipped. See "Battle Phase" page 111.

Movement Phase

The Movement Phase comprises the bulk of the player's turn, where the Player can survey the battlefield, "googling" at their troops and the enemy troops looking for strengths and weaknesses to exploit. Elements are then moved into appropriate strategic positions, shooting range and into battle.

Googling Elements

To look at an element and get information on what it is, otherwise referred to as googling, simply roll the cursor over the element. You may google any element of either side. When you do so the cursor will change to a spyglass and an information scroll will open displaying statistics for the element under the cursor, as shown in Figure 8.11.

Name - In the top left hand corner of the scroll you will see the name of the Element displayed in large old style Lombardic text. The element shown in Figure 8.11 is

monks known as “Hospitallers,” an order founded in Jerusalem in 1113. The knight featured as the title art for *Chevalier* is a Hospitaller knight. Usually the Element’s name is a distinct identifier, such as tribal group or class of warriors, such as “Mamluk” or “Turkomans.” In the case of a general, as identified by the star icon on the Element instead of a type icon, the Element name will be the actual name of the general, such as “Guy de Lusigan”, “Parmenio”, or “Saladin.” When there is no distinct identifier the name will be the same as the Element type, such as “Crossbows.”

Insignia - Directly to the right of the Element name is an army insignia silhouette identifying which army this element belongs to and thereby also signifying which player controls it. The element shown in 8.11 belongs to the Crusader army as evidenced by the Catholic Cross insignia. For a key to army insignia see Figure 8.3.

Figure 8.11 “Google” Information Scroll

Regular or *Clumsy* - The first word of second line of text on the Information Scroll, displayed in red, denotes if the Element is **Regular** or if it is **Clumsy**. **Regulars** are typically enlisted troops under officers appointed by the government and are highly practiced in maneuver and combat techniques, while **Clumsy** troops, as evidenced by the word **Clumsy**, are troops unaccustomed to waiting for and obeying formal orders. They often join the army with acquaintances under local or tribal leaders and are unpracticed and less drilled than **Regulars**. The Element shown in Figure 8.11 is **Regular** as they are full time members of a highly rigorous and practiced military order.

Grade - The second word of second line of text on the Information Scroll, displayed in red, denotes the Grade of the Element. Grade takes into account differences in morale, degree of training, and equipment or mobility, and can be one of four values; **Superior**, those troops recognized by contemporaries as significantly superior in morale or efficiency; **Ordinary**, representing the most common or most typical troops of that type; **Inferior**, brittle troops historically identifiable as of significantly inferior morale or efficiency; and **Fast**, those troops who move faster and further than average but are usually not as well protected. **Fast** troops move 40 paces more than other Elements of the same type. The Element shown in Figure 8.11 is **Superior**, the Hospitallers being of the best armed and trained Knights in Richard I's army.

Type - The word on the third line of text on the Information Scroll, displayed in red and embraced by "—" dashes, denotes the **Type** of the Element. For a table outlining specifics of Element types see Figure 8.6.

In - The "In:" line of Figure 8.11 designates the type of terrain this Element is in. For the purposes of this thesis prototype the terrain is always considered to be **Clear**. In future versions terrain will have a marked effect on movement and combat, with lighter

troop types such as Skirmishers and Light Infantry performing better in **Rough** and **Difficult** terrain.

Status - The "Status:" line of the Information Scroll (Figure 8.11) designates the current status of this Element. Status can be one of four values; **Normal**; **Engaged**, in which case the Element is displayed with a red burgundy glow to show that it is in combat; **Shooting**, in which case the Element is displayed with a green glow to show that it is shooting or in range of being shot at; or **Moving**, in which case the Element is displayed with a white glow to show that it is in the process of moving and Initiative Points have been invested by the player to permit it move around the map. **Engaged** elements may only withdraw from battle if their movement is greater than the movement of the Element they are fighting. No unit can withdraw from battle the same turn they entered into contact.

Influence - The "Influence:" line of the Information Scroll (Figure 8.11) designates the Influence of this Element on the morale of the command it is in. An Element with a **Very High** influence will have a disproportionate effect on morale of the command if it dies than an Element with **Low** influence. Influence can be one of five values, each value has a numbered weighting effect on moral; **Very High** influence is reserved for commanders, be they C-in-C or sub-commanders, and has a weighting of 4; **High** influence is awarded to most (but not all) Elements that are graded as **Superior** and has a weighting of 2; **Normal** influence is usually seen amongst Elements graded as **Ordinary** and has a weighting of 1; **Low** influence is attributed to Elements such as Hordes or Skirmishers, who have a lesser effect on morale if eliminated, they have a weighting of only 1/2; Elements with an influence of **None** are Elements such as Expendables, which are expected to be eliminated during the battle and have a 0

weighting. For information on how Influence effects Moral on a command see section "How to Interpret the Screen—Morale" page 93.

Move - The "Move:" line of the Information Scroll (Figure 8.11) designates how far the Element can move and is measured in paces (for what a pace is see "How to Interpret the Screen—Scale," page 88). At the beginning of a player's turn the player's elements are all refreshed to maximum movement. See Figure 8.6 for movement of each Element Type.

There is always an illustration depicting the Element googled at the bottom of the Information Scroll. This graphic will be facing to the right if a friendly Element and to the left if an enemy.

Moving Groups of Elements

How many groups of Elements you may move in a turn is determined by the number of Initiative Points awarded to each command at the beginning of the turn. In the case of the Crusader player in our example (Figure 8.4) there are 3 Initiative Points for the "Left" and "Right" wing commands and 4 Initiative Points for the "Center" command.

The Crusader player sees that he would like to move the Spears and Crossbows on the left side of the the center command forward so that they can get closer to the Saracen Horde and the Crossbows will be in range of the Saracen Cavalry to their front. To do this, start the mouse slightly to the left of the leftmost Spear of the Crusader center command and drag. As you do so you will see a marquee appear while you have the mouse held down. Drag the marquee over the Spears and Crossbows until all of them are within the marquee (Figure 8.12). You will notice that as you drag over the Elements

they highlight (glow) yellow and the Elements to the left of the spears become grayed out. This indicates that those to the left are in a different command from the Elements being selected, and cannot be selected with these elements that are in the center command. You will also notice that you may only select Elements adjacent to Elements already selected. This ensures a group selection is made.

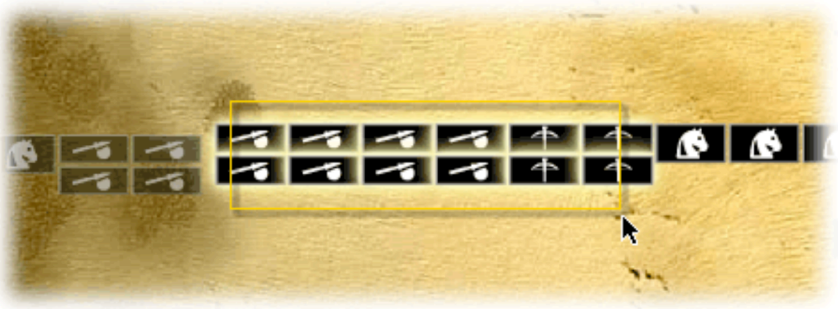

Figure 8.12 Selecting Elements

As you make the selection the "google" Information Scroll will open, and as you add the Crossbows to the group the scroll will change to indicate that a group of more than one type has been selected. Then, when you release the mouse, a circular movement wheel will appear at the base of the Information Scroll containing triangular movement buttons. These buttons may be used to move the selected group. When you roll over each button a shadow will appear for each selected Element to indicate what its new position will be if you select that move, also, a number in a circle will appear indicating how many Initiative Points it will cost the command to make that move. Furthermore, the circle on the Control Panel indicating Initiative Points for that command will highlight white, showing that that is the command from which points will be deducted (Figure 8.13). If the command has no Initiative Points to spend then the circle will flash red and no movement buttons will be active on the movement control.

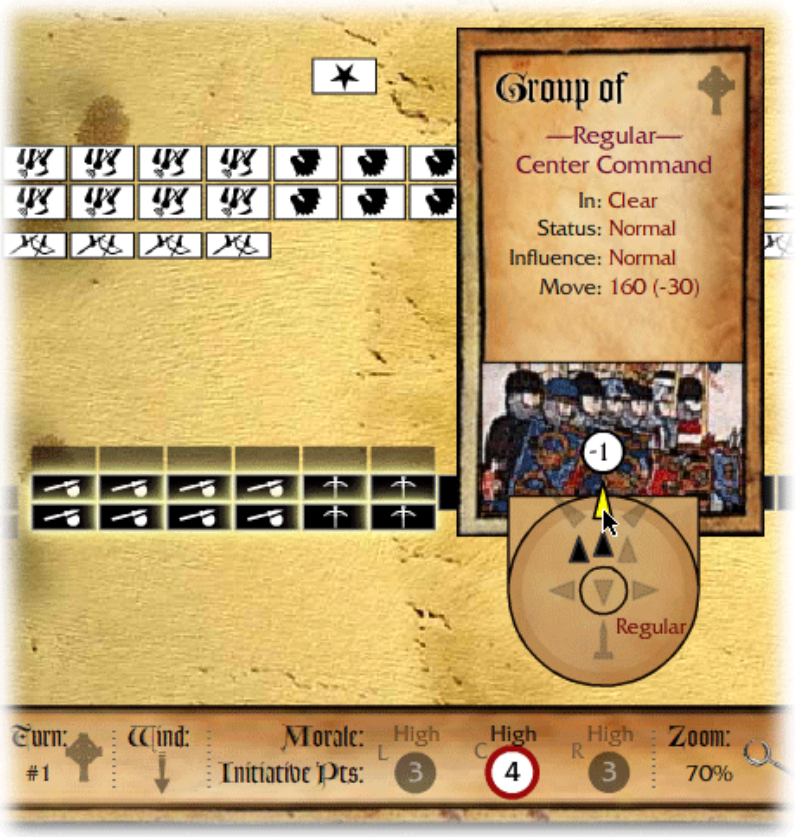

Figure 8.13 Movement Control when moving a group

In Figure 8.13 the mouse is over the Forward movement button and it can be seen that it will cost one point, as indicated by the "-1" in the circle that appears next to the button. Press the Forward button to move the selected Elements forward. Keep pressing forward until the Elements have exhausted all their 160 paces movement, this should be six clicks, the first five clicks moving 30 paces and the last click the final 10 paces. See Figure 8.14. for details on the Movement Wheel.

Once the Elements have moved their full movement click on an empty part of the map to deselect the group.

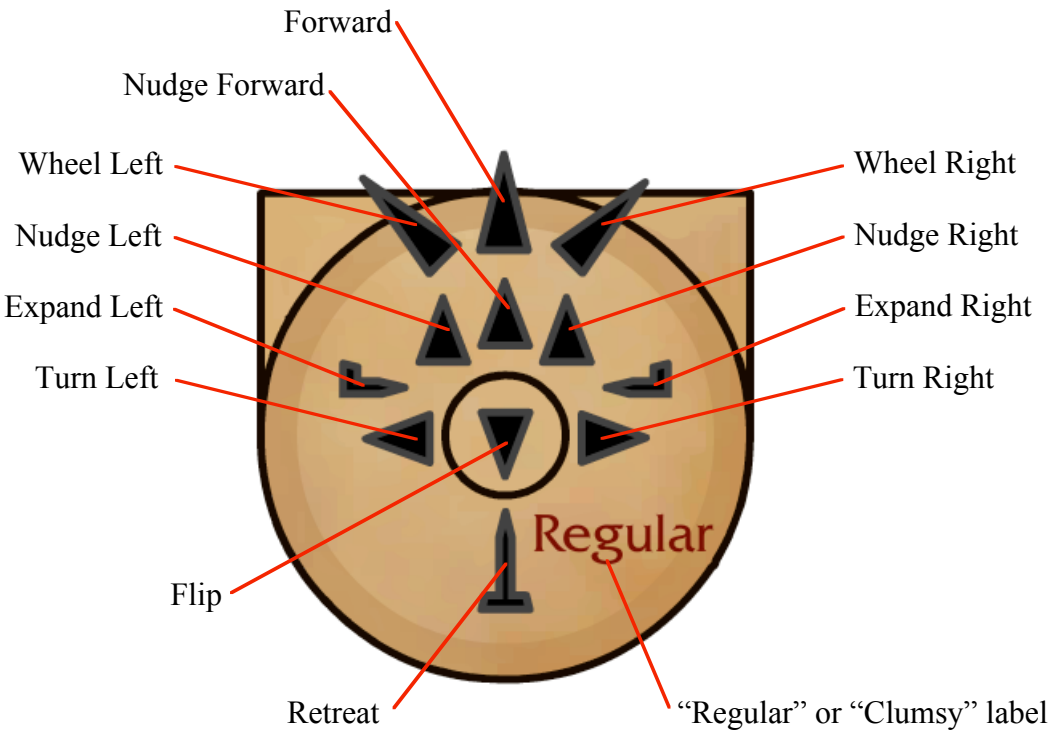

Figure 8.14 Movement Control

Here are some things to remember about moving Elements with the Movement Control:

- "Regular" troops cost 1 Initiative point to move, while "Clumsy" troops cost 1 to move forward but 2 to do any other move.
- Elements that have started moving forward may continue to do so at no cost until their full movement is reached. This is indicated by their Status changing to "Moving" and the Element glowing white.
- All costs for Light troops such as Skirmishers, Light Infantry and Light Horse are halved.
- It always costs Initiative Points for a group to Wheel or Retreat.
- "Regular" Elements may nudge four times in a turn, while "Clumsy" Elements may nudge two times in a turn.
- The cost to nudge is paid only for the first nudge.
- Some troops may move through each other if their bases are lined up. For instance Skirmishers, can move through or be moved through.
- Elements will automatically snap into alignment with other Elements to so form a group, or to fight a combat.
- When moving a single Element of "Regular" troops it may make any moves after paying 1 Initiative Point.
- When moving a single Element of "Clumsy" troops it may make any moves after paying 2 Initiative Points.

Now that the Crossbows are quite close enough to shoot the Saracen Cavalry, the Crusader player would like to wheel five of the knights in the center of the map 45 degrees clockwise, bringing the left edge knight into battle with the Saracen Skirmishers. To do this, click drag across the five knights starting from the left edge of the group as seen in Figure 8.15.

Figure 8.15 Select 5 Knights

When you release the mouse the movement wheel will appear as in the previous example. Select the Wheel Right button on the movement control (Figure 8.16).[10] It will cost two points to wheel these knights as they are **Clumsy** troops and **Clumsy** troops cost an additional point to do anything other than move straight ahead. Select the Wheel Right button. You will see that, as the knights wheel, the Movement Wheel turns also, reflecting the angle of the group. Notice also that as the knight on the left edge comes into contact with the Saracen Skirmishers the Skirmishers turn to face the Knights and both Elements glow burgundy indicating that they are in combat. The combat will not be resolved until after the Crusader player has finished all moves and completed the Movement Phase. For more information on Battles and how they are resolved see "Battle Phase" page 111.

[10] Note: if you include the sixth knight in the selection, the Wheel Right option on the movement wheel will not be available since there is not enough room for all six knights to wheel.

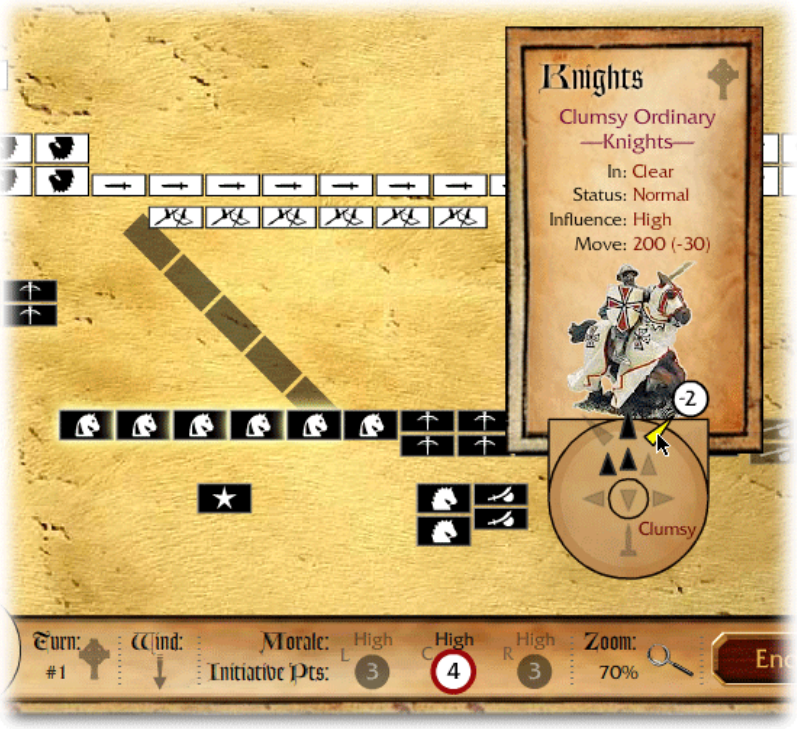

Figure 8.16 Wheel 5 Knights

Sometimes a single element will be in the way of movement. When this happens the group needs to be selected without the offending unit to move. This oftentimes happens with a group trying to move into battle when there is not enough room for the whole group to shift into contact with the enemy.

Moving Single Elements

If a player selects and moves only a single element then that element can make any subsequent maneuvers (such as Wheels) for free, once having paid 1 Initiative Point, in the case of "Regulars," or 2 Initiate Points, in the case of a "Clumsy" troops. This allows players to get single elements to where they want them without having to pay extra for multiple turns.

Moving Light Troops

All Initiative Point costs for moving groups light troops such as Skirmishers, Light Infantry and Light Horse are halved. Moving a group of **Regular** Skirmishers straight forward will only cost 1/2 an Initiative Point. Similarly, wheeling a group of **Clumsy** Light Horse will only cost 1 Initiative Point. This rule also applies when moving single elements.

The Crusader player potentially has many more moves he could make before finishing his turn, but, for the purposes of this tutorial, the Crossbows are in range of the Cavalry, and the Knights on the left are in combat with the Skirmishers, so select "End Turn" from the Controls Palette (Figure 8.4) to commence combat.

Combat

There are two forms of combat, distant combat, which occurs in a phase called "Shooting," and close combat, which occurs in a phase called "Battle." The "Shooting" phase is always conducted first. Both phases are very similar, with the player being allowed to elect which combats are conducted in what order. As adjacent Elements can support combating Elements in the "Battle" phase, the order that combats are conducted in that phase is very important. Players would do well to choose the order of their close combat "Battles" wisely for best results. See "Battle Phase" page 111 for more information on this.

Shooting Phase

Shooting is limited to those troop types that historically shot collectively at long range, and includes all collective shooting on command by formed bodies of archers or hand gunners, or artillery, at bodies of troops beyond the range at which shooting at individuals is possible. The ranges allowed are based on effective military ranges found in contemporary sources or established by experiment with reproduction weapons to allow a significant effect. Shooting at longer range with only a minor morale effect or a sprinkling of casualties is disregarded for the sake of speed and simplicity (Barker, 2006).

The shooting range for Archers and Crossbows is 240 paces. Shot have a range of only 80 paces, while artillery have a range of 480 paces. All elements who are shooting while being shot at suffer a -1, making each side easier to double, and thus reflecting the horrendous losses that would occur during shooting matches, which were often a very bloody affairs.

The Crusader player tutorial example has two groups of two Crossbows in range of the Saracens. The first group on the left were moved into range of the Saracen Cavalry by the player, as directed by the tutorial above, and the second group of Crossbows started in range of the Saracen Skirmishers. Due to these eligible shooting targets you will see a “Shooting” window appear when you end the Crusader player’s turn (Figure 8.17).

Figure 8.17 Shooting Window

There are four active combats, as indicated by the "1 of 4" between the arrow buttons along the bottom of the window. Click on the arrow buttons to step backwards and forward through the four battles, looking at the statistics for each one. You will notice in the information box on the left for the Crossbows that each receives "+1." This is awarded for the additional Element of Crossbows directly behind that are supporting the shooting. When there are additional shooters to the left and right all shooting at the same target you will see that the target gets a "-1" for each additional shooter to a maximum of -2.

You will also see a list of other combat factors depending upon the situation and also a base factor for each Element Type involved in combat. For a table of what each Type's base factor is see Figure 8.6. These factors are important as when the combat is conducted by pressing the "Fight!" button (an action akin to rolling dice on the tabletop of a *DBMM* game) a randomized dice factor between 1 and 6 is added to the combat factors for each side.

Immediately after combat dice have been rolled, an additional modifier is taken into account depending on the Grade of the Elements involved. **Superior** Elements are awarded "+1" if their total factors including dice are greater in their own turn, or, if their total is less in the opponent's turn. **Inferior** Elements are penalized "-1" if their total is equal or less. **Fast** Elements are penalized "-1" if their total is less in the opponent's turn.

The objective is to have a total combat factor that is higher or, better still, have a factor that is double that of the opponent. The result of combat is indicated for each side underneath the information box pertaining to their Element. The player's Element whose turn it is is displayed on the left, while the non-bounding (player whose turn it is not) player's Element is displayed on the right.

In these tutorial combat instances the Crossbows are unharmed if they score lower or are doubled, as evident by the words "Stand if less" and "Stand if doubled," under their side of the window. This is because the Saracen Cavalry and Skirmishers cannot shoot back since they are not shooting troops. The Saracen Elements on the other hand will "Recoil if less" or "Killed if doubled." Other possible combat results are "Repulsed", "Flee", and "Spent." In the case of ties there is no result. For a table of Combat Results and what they mean see Figure 8.17.

Different troop types will be given different Combat Results depending on the combat matches. Some troops do not need to double an opponent to kill them, they only need to score higher. This is referred to as a "Quick Kill." For instance, Cavalry are awarded a "Quick Kill" when fighting Skirmishers, Crossbows, and Archers. For a full listing of "Quick Kills" see Figure 8.6. Furthermore, some troops pursue after combat against certain troops while others do not. Elephants, Knights and Pike generally do pursue, while Cavalry, Light Horse, Light Infantry and Skirmishers do not. For details on who does and does not pursue after combat see "Pursuing Elements" on page 42 of the *DBMM* rules.

Stand	No effect.
Recoil	Represents troops responding gradually to enemy pressure. A recoiling element moves back its base depth to its rear. Recoiling Elements will push back friends directly to the rear, otherwise, if there is not enough room to recoil, Recoiling Elements are Killed.
Flee	Represents a disorganized panic move by individuals. Fleeing Elements first recoil its base depth if it can, then turns 180° and continues moving its full movement distance.
Spent	Represents Light Horse and Skirmishers that have exhausted their missiles, courage or patience. Spent Elements are removed, but do not count as lost and so do not effect the moral of the command.
Killed	Represents men being killed, disabled or made prisoner and survivors dispersing and quitting the field individually. Killed Elements are reoved.

Figure 8.17 Combat Results

To conduct a combat, press the **Fight!** button. The dice are rolled and you can see what score each player received, along with totals and Grading modifiers in the bottom of each Element's information box. A flag will appear in the color of the victorious player and the Combat Result for the outcome of the battle becomes bolded and displayed in red. If the loosing player's element is "Killed" or "Spent" then a skull icon will appear over that player's half of the window. Once the combat has been fought the **Fight!** button

changes to an **OK** button. Press this **OK** button or one of the arrow buttons to move to another combat. Repeat these steps until all four shooting combats have been done. Once the last combat is finished the **Fight!** button will change to a **Done** button instead of an **OK** button and the arrow buttons will gray out. Press the **Done** button to proceed to the Battle Phase.

Battle Phase

Battle includes not only hand-to-hand fighting using edged or pointed weapons, but also all shooting by mounted archers, javelin men and others that shot at close range, or at charging enemy (Barker, 2006).

The Battle Phase is conducted in exactly the same fashion as Shooting, with the same general interface. In the Crusader tutorial example there is one Battle to conduct between the Crusader Knight and the Saracen Skirmisher. For an example of this screen see Figure 8.18. You will see it is almost identical to the Shooting screen in Figure 8.16. The main difference between Shooting and Battle is that many of the combat modifiers are slightly different, the most important one being overlaps. Notice how in Figure 8.18 the Saracen Skirmisher suffers "-1" penalty for "left flank overlapped." This is because the Element of Knights adjacent to the right of the Knight in combat is aiding in the battle. As Battles cause Elements to recoil and move during each Battle's execution, which Element is overlapped and when can change according to the sequence Battles are executed. As such, it can make a significant difference to the outcome what order a player chooses to carry out battles. In general, it is best to conduct the battles you have the best chance in first, as by winning these battles you will create more overlaps where your troops can aid in other battles yet to be calculated.

In the Battle example shown in Figure 8.18 we see the single Battle initiated by the Crusader player's Knight in the tutorial. What's important to notice here is that the Saracen Skirmishers were forced to turn to face the Crusader Knights, putting them at a angle to their own troops and causing them to be in a position where they cannot Recoil or push back other friendly troops. This is indicated by the "-1 retreat blocked—cannot recoil or flee!" modifier displayed in the Skirmisher side of the Battle window. It is very likely here that the Crusader player will win this Battle. Press the **Fight!** button to conduct this battle and then the **Done** battle to finish the turn.

Figure 8.18 Battle Window

Once all Battles have been conducted the Start Turn Window will open for the other player's turn, in this case the Saracen player.

Figure 8.19 - Start Turn Window for Saracen Player

For more detailed information on general rules, in particular the intricacies of Shooting and Battle modifiers, please consult the *DBMM* rules by Philip Barker at www.phil-barker.pwp.blueyonder.co.uk/DBMM.doc.

Chapter 9 Summary and Conclusions

Chevalier is the first step in producing a flexible online tool for simulating historical battle over a wide range of periods. Its potential as a teaching aid for history education is great, as players are given the chance to experience and experiment first hand the various battle tactics used by each side. Chevalier is a contribution to the methodology of teaching history. Historical concepts can be taught using modern technology in a new and innovative way. When used in conjunction with a historical Website it promises to introduce a new form of history involving the student in real history, and placing in the hands of the student key battles of the ancient and medieval world.

By being developed in the Flash platform *Chevalier* successfully delivers its rich combination of interactivity, text, vector graphics, raster graphics, animation, and sound to provide a battle simulation that can reach the maximum possible audience delivered across a wide range of machines and browsers. Furthermore, *Chevalier* is ideally placed to benefit from Adobe's many development enhancements to Flash and the soon to be released update *Flash 8.5* which features *ActionScript 3*, which promises more control and faster execution.

The *Chevalier* thesis project is a successful implementation of the *DBMM* rules system, showing that the core mechanics of *DB* style simulation games are possible in a accessible online format. The work of this thesis constitutes the major basis of what is potentially a significant contribution for the teaching of history. Of course further developments would be needed to produce a commercially viable package.

References

Allen. E. (1999) What is DBM? <http://tetrad.stanford.edu/info/WhatIsDBM.html> (cited 8 April 2006).

Allen, J.P. & Chatelier, P. & Clark, H.J. & Sorenson, R. (1982). Behavioral science in the military: Research trends for the eighties. *Professional Psychology,* 13*,* 918-929.

Apple Computer, Inc. (1992). *Macintosh Human Interface Guidelines,* (New York: Addison-Wesley, May 1992).

Asbridge, T. *The First Crusade,* (Oxford University Press, 2004): ix.

Barker, P. (2006). "Competition/Results/Speed" email sent by Philip Barker to the DBMMList <http://games.groups.yahoo.com/group/DBMMlist/message/32658.> (cited 11 Apr 2006).

Barker, P. (2006). *De Bellis Magistrorum Militum Wargames Rules for Ancient and Medieval Battle 3000 BC to 1500 AD* <www.phil-barker.pwp.blueyonder.co.uk/DBMM.doc> (cited 21 April 2006).

Barker, P. & Scott, R. B. & LaflinBarker, S. (2004). *De Bellis Antiquitatis Simple Fast Play Ancient Wargame* (London: Wargames Research Group, 2004).

Barker, P. & Scott, R. B. (2000). *De Bellis Multitudinis Wargames Rules for Ancient and Medieval Battle 3000 BC to 1500 AD* (London: Wargames Research Group, July 2000).

Barker, P. & Scott, R. B. (2004). *De Bellis Renationis Wargames Rules for Renaissance Battle 1494 AD to 1700 AD* (London: Wargames Research Group, January 2004).

Bédoyère, G. (1999) The Roman Army in Britian <www.romanbritain.freeserve.co.uk/Auxilia.htm> (cited 20 April 2006).

Bradley Commission in Schools. (1988). *Building a History Curriculum: Guidelines for Teaching History in Schools* (Washington, DC: Educational Excellence Network, 1988).

Bulliet, R. (1979). *Conversion to Islam in the Medieval Period*, (Cambridge: Harvard University Press, 1979): 23.

DBA Online Wargame (2000). <www.dbaol.com> (cited 1 April 2006).

Dekkers, J. & Donate, S. (1981). The Integration of Research Studies on the Use of

Simulation as an Instructional Strategy, *Journal of educational Research,* 74, 424-427.

Dempsey, J. V. & Haynes, L. L. & Lucassen, B. A. & Casey, M. S. (2002), Forty simple computer games and what they could mean to educators, *Simulation & Gaming,* Vol. 33 No. 2, June 2002, 157-168.

Entertainment Software Association, (2006). Facts & Research—Game Player Data <www.theesa.com/facts/gamer_data.php> (cited 6 May 2006).

Evans, R. (1990). Social Studies Under Fire: Diane Ravitch and the Revival of History, *Georgia Social Science Journal* 20, no. 1. (1990): 17-18.

Gaddis, J. L. (1990). The Nature of Contemporary History, Occasional Paper, *National Council for History education* (Westlake, OH: National Council for History Education, 1990), 4.

Grossman, G. & Huang, E. (2006). ActionScript 3:Overview <http://labs.macromedia.com/wiki/index.php/ActionScript_3:overview> (cited 20 April 2006).

Games Workshop. (2006). About Us. <www.games-workshop.com/aboutus.htm> (cited 1 April 2006).

Games Workshop News. (2006). Games Workshop Acquisition of *DBA* Marks Entry to Historical Market <www.workshop-news.com/dbapressrelease.htm> (cited 1 April 2006).

Hampel, R. L. (1985). "Too Much is Too Little," *Social education* May (1985):364.

Harris, J. W. & Stocker, H. (1998), *Handbook of Mathematics and Computational Science,* (Springer-Verlag New York, 1998): 81.

Hersh, S M. (2006). The Iran Plans, *New Yorker,* April 17, 2006.

International Data Corporation (2005). <www.macromedia.com/software/player_census/flashplayer/penetration.html> (cited 20 April 2006).

Jacobson, D. & Jacobson, J. (2002). *Flash and XML, A Developer's Guide*, (Addison-Wesley, 2002).

Lewis, B. (2001). "The Revolt of Islam" *The New Yorker,* Issue of November 11th 2001.

Lewis, B. (2003). *The Crisis of Islam: Holy War and Unholy Terror*, Modern Library; Modern Lib edition (March 2003).

Lowke, R. J. (2001) 'The Crescent & the Cross," A proposal written for Harvard course program "CREA S-165 Writing Grant Proposals" taught by Frank White (August 2001).

Lyons, J. (2001). Bush enters Mideast's rhetorical minefield <www.positiveatheism.org/hist/quotes/bush.htm> (Reuters: September 21, 2001).

Mack, S. (2006). Adobe Looks to Future at Flashforward Conference <www.streamingmedia.com/r/printerfriendly.asp?id=9242> (cited 20 April 2006).

Makar, J. & Winiarczyk, B. (2004) *Flash MX 2004 Game Design Demystified,* (Macromedia Press, USA, 2004): 188.

Malouf, D. B. (1988). The effect of instructional computer games on continuing student motivation. *Journal of Special Education,* 21(4), 27-38.

Mork, G. R. (1979). Teaching History with Games, *American History Association Newsletter,* Vol:17 iss:6: 4-6. 1979.

Motion-Twin ActionScript 2 Compiler (2006). <www.mtasc.org/> (cited 20 April 2006).

National Assessment of Educational Progress. (2006). World History Assessment, the Nation's Report Card <http://nces.ed.gov/nationsreportcard/worldhistory/>. (cited 1 April 2006).

National Endowment for the Humanities (1997). National Endowment for the Humanities in the Digital Age: A Report to Congress and the Country, 1997.

NPD Research, (2005). Flash Player Penetration Survey <www.macromedia.com/software/player_census/flashplayer/version_penetration.html> (cited 20 April 2006).

Object Management Group. (2006). Unified Modeling Language <www.uml.org> (cited 30 April 2006).

Office of Technology Assessment. (1988).

Oxford American Dictionaries (2006). "Irregular" from Mac Dictionary Widget.
Panel on Educational Technology. (1997). Report to the President on the Use of Technology to Strengthen K-12 Education in the United States.

Randel, J. M. & Morris, B. & Wetzel, C. D. & Whitehill, B. V. (1992). The Effectiveness of Games for Educational Purposes: A Review of Recent Research, *Simulation and Gaming, An International Journal of Theory, Design, and Research,* 23, 261-275.

Ravitch, D. & Finn, C. E. (1987). *What Do 17-Year-Olds Know?: A Report of the First National Assessment of History and Literature* (New York:Harper & Row, 1987).

Riley-Smith, J. (1995). *Oxford Illustrated History of the Crusades* (Oxford, 1995).
Rothero, C. (1981). *The Armies of Agincourt,* (Osprey, 1981).

Stratford, J. (2005) XML 101
<www.actionscript.org/tutorials/intermediate/XML/index.shtml> (cited 1 April 2006).

Sellers, C. P. (1993). An Analysis of Writing Assignments in Selected History Textbooks for Grades Seven and Eleven (Ed. D. diss. Virginia Institute and State University, 1993).

Sekunda, N. (1984). *The Army of Alexander the Great,* (Osprey, 1984).

Simmonds, J. C. (1989). History Curriculum and Curriculum Change in Colleges and Universities of the United States: Astudy of Twenty-Three History Departments in 1988, *The History Teacher* 22, no. 3 (1989): 291-315.

Society of Australian Ancients Wargamers. (2005) Tournament Procedures for use at Cancon DBM 2005 <www.nwa.org.au/dbx/SAAW/TP_CanconDBM2005.doc> (cited 1 April 2006).

Tate, B. D. & Durand, R. C. (1986). Five American History Books for Survey Courses: A Review Essay, *Teaching History* 4 (1986): 221-26.

U.S. Department of Education (2006). No Child Left Behind.
<www.ed.gov/nclb/landing.jhtml> (cited 1 April 2006).

Wise, E. (1978) *Armies of the Crusades,* (Osprey, 1978).

White, R. M. (1994). An Alternative Approach to Teaching History, *OAH Magazine of History,* 8 no. 2 (1994): 58.

Yarema, A. E. (2002). A Decade of Debate: Improving Content and Interest in History Education, *The History Teacher,* 35, no. 3, (May 2002): 395-396.

Appendix A Glossary of Terms

- Commander in Chief (C-in-C)
- *De Bellis Magistrorum Militum* (*DBMM*) - translates to "For the Wars of the Masters of Soldiers." The game is so named due to the emphasis placed on the commanders to have a battle plan.
- Hot Seat game - When a game is played running from a single machine. The seat becomes "hot" as the players must constantly switch chairs to have access to the game.
- Impetuous Troops - Troops liable to advance without waiting for orders.
- Moral Equivalents (ME) - Elements of different types have different effects on morale, as measured in ME.
- Paces (p) - This is the relationship between distances on the table and those on a real battlefield. Distances in the text are in paces (p), each of 0.75 meters or 2.5 feet. This is because the length of a man's stride has remained constant throughout history, while such units as cubits, yards and meters come and go.
- Tabletop Wargame - Tabletop wargaming typically involves the use of miniature metal or plastic models for the game play units and model scenery placed on a tabletop as a playing surface.
- Knights (Kn) - representing all those noble or heavy horsemen of high morale that charge at first instance without shooting, with the intention of breaking through and destroying enemy by sheer weight and impetus.
- Cavalry (Cv) - representing the majority of ancient horsemen, usually partially armored, combining or following close range javelin or bow shooting with controlled charges.
- Light Horse (LH) - including all especially swift riders who scout or fight dispersed, evading enemy charges.
- Expendables (Ex) - scythed chariots fitted with scythe blades and spear points, usually with 4 horses and a single crewman, intended to be driven into enemy formations in a single suicidal charge early in the battle to break up or destroy them.
- Spears (Sp) - representing all close formation infantry fighting with thrusting spears

and heavy shields in a rigid shield wall.

- Pike (Pk) - including all close formation infantry fighting collectively with pikes or long spears wielded in both hands.
- Blades (Bd) - including all close fighting infantry primarily skilled in fencing individually with swords or heavier cutting or cut-and-thrust weapons, sometimes supplemented by hand-hurled missiles or bows.
- Light Infantry (LI) - representing foot willing to fight hand-to-hand, but emphasizing mobility or fighting in difficult terrain.
- Archers (Ar) - representing foot who fight in formed bodies by shooting collectively with missiles shot at longer range than psiloi, often in volleys at command, and who rely on dense shooting.
- Skirmishers (Sk) - including all dispersed skirmishers on foot shooting individually with javelin, sling, staff sling, bow, crossbow or hand gun, who fight in a loose swarm hanging around enemy foot, running away when charged.
- Hordes (Hd) - all unwilling or incompetent foot, brought to swell numbers and/or perform menial services, or attracted by desperation, religious or political fanaticism or greed, and too tightly huddled, scared, stupid or indoctrinated to run away.
- Baggage (Bg) - representing the army's logistic support, including all personnel, supplies and equipment that increase the physical or mental welfare of troops or generals.
- Superior (S) - Troops recognized as significantly superior in morale or efficiency.
- Ordinary (O) - Representing the most common or most typical troops of that type.
- Inferior (I) - Brittle troops of significantly inferior morale or efficiency.
- Fast (F) - Troops who move faster and further than average but are usually worse protected.
- Regulars (Reg) - troops typically enlisted into units under officers appointed by the government and practiced in maneuver and combat techniques.
- Irregulars (Irr) - troops who join with acquaintances under local or tribal leaders, and are less accustomed to waiting for, listening to, or precisely and instantly obeying formal orders.

Appendix B Units of Scale

A pace in *DBMM* is considered to be 0.75 meters or 2.5 feet, which is the length of a mans stride. 2000 paces is 1 Roman mile.

Each cavalry element represents 128-200 riders, varying with the army.
Each foot element, except Hordes, represents 200-256 infantry, varying with the army.
Each element of type Hordes represents up to 1,000 men in a deep mass.
Each element of type Elephants represents about 16 elephants.
Each element of type Expendables represents about 25 scythed chariots.

The width of an element in *DBMM* is equal to 80 paces, which is 60 meters.
In tabletop gaming terms, for 15 mm scale figures, an element width is 40 mm (4 cm).

The ideal playing area for a 15 mm game is 1.8m (72') x 1.2m (48'). Considering each element width on the tabletop is 4 cm then the ideal playing area is 45 widths x 30 widths.

Considering that a *Chevalier* element width is 7 grid blocks then the ideal playing area transposes to (45 widths x 7 grid blocks =) 315 blocks by (30 widths x 7 grid blocks =) 210 blocks.

When the *Chevalier* map is viewed at 100% a block is 8 pixels, making the map size (315 blocks x 8 pixels =) 2520 pixels by (210 blocks x 8 pixels =) 1680 pixels.

A player turn is equivalent to approximately 20 minutes in real life.

Appendix C Battle Scenarios

Battle of Arsuf

Date: September 7th, 1191 CE

Short Description:
Richard Coeur de Lion's march along the Mediterranean coast from Arsuf to Joppa during the Third Crusade.

Long Description:
During the Third Crusade, after the capture of Acre, Richard Coeur de Lion began marching towards Joppa, an important city port needed by the crusaders before they could assault Jerusalem. But the Muslim army under Saladin were intent on stopping their progress.

On September 7, 1191, soon after the crusaders had left Arsuf, Muslim attacks became more frequent and intense. Saladin tried to lure the Crusaders out with a light cavalry charge, but Richard delayed as long as possible while his crossbowmen held the Muslims back. When the crusaders finally did charge it overwhelmed Saladin's army and inflicted heavy losses on the forces attacking the rear. Seven hundred crusaders and several thousand Muslims were killed. Saladin was forced to retreat, and the legend of his invincibility was destroyed.

The victory at Arsuf enabled the crusaders to occupy Joppa but it was not a crushing blow to the Muslims. Saladin was able to regroup his forces, which the crusaders had not pursued for fear of ambushes. The Muslims soon renewed their harassing tactics, and Richard did not dare to push on to Jerusalem.

Battle of Gaugamela

Date: October 1st, 331 BCE

Short Description:
Clash between the forces of Alexander the Great and Darius III of Persia that brought the fall of the Persian empire.

Long Description:
Attempting to stop Alexander the Great's incursions into the Persian empire, Darius III, king of Persia prepared a battleground on the Plain of Gaugamela, near Arbela, in preparation for Alexander's advance. Darius had the terrain of the prospective battlefield smoothed level so that his larger army could operate with effectiveness against the Macedonians.

On October 1st, 331 BCE, soon after a total eclipse of the moon, Darius gathered all his military strength to fight Alexander—chariots with scythes on the wheels, elephants, and a great number of cavalry and foot soldiers. Alexander's well-trained army faced Darius' larger battle line.

During the combat, so much of Darius' cavalry were drawn into the battle that they left Darius and his Persian infantry exposed in the center of the battle. Anticipating this, Alexander lead his cavalry straight toward Darius. At this Darius fled, and panic spread through the entire Persian army which began a headlong retreat while being cut down by the pursuing Greeks.

This Macedonian victory spelled the end of the Persian empire and left Alexander the master of southwest Asia.

Battle of Agincourt

Date: October 25th, 1415 CE

Short Description:
The third great English victory over the French in the Hundred Years' War. This decisive battle proved the superiority of the longbow over the crossbow.

Long Description:
In pursuit of his claim to the French throne, Henry V invaded Normandy with an army of 11,000 men in August 1415. The English took Harfleur in September, but, with their forces cut in half by battle and disease, they resolved to return home to England.

On October 25th, 1415, near the village of Agincourt in northern France they were cornered by a French army of 20,000–30,000 men, including many mounted knights in heavy armor. On a cramped battlefield where the superior French numbers offered little advantage, Henry made skillful use of his lightly equipped, mobile archers. The French were disastrously defeated, losing over 6,000 men, while the English lost fewer than 450.

This decisive battle was the third great English victory over the French in the Hundred Years' War, proving the superiority of the longbow over the crossbow and hastening the end of the heavily armored knight, the military basis of feudalism.

Appendix D Future Enhancements

The estimated time quoted in the original thesis proposal for this project was 44 weeks (11 months), with 6 of those weeks attributed to documentation. Actual time available was only 24 weeks (6 months), that being the period from the beginning of November 2005 through to the end of April 2006. The month of April (4 weeks), was reserved for and used almost exclusively for documentation.

As there was approximately half the time available for development than was quoted in the original project proposal it may be seen as significant that the great majority of proposed features were implemented in the final project. Regardless, the proposal had to be reduced to create manageable thesis project for the time available.

The most important feature of the original proposal that was not implemented in the final project was a methodology for dealing with players playing across a network using different machines. It was deemed that there was no point implementing a networked game when the actual mechanics of the game were yet to be tested and decided. As such, it was resolved to complete the game before endeavoring on the network capabilities, but to do so keeping in mind network implementation issues and restrictions so that they may be added easily to future versions of *Chevalier*.

Proxy Server Strategy for Networked Games

The *Chevalier* networked game would be achieved by using a “Proxy Server” strategy, where a thin Server would be implemented as a central gateway between the two client Flash applications conducting the *Chevalier* game. This system is essentially a “peer to peer” strategy, but using the server purely as “client request, server response”

gateway. This is why the system is called a "proxy" strategy. The Server would be implemented under Microsoft .net using C#, or by utilizing the Open Source Flash Server *Red 5,* written in Java. *Red5* may be found at www.osflash.org/red5.

When launching *Chevalier* each player would be presented with a general forum displaying a list of other players the Server detects who are looking for a game. Players can create and enter a game room and "sit" to commence a game. When creating a game room the Server gives the requesting Client a Room Key identifier for the game room and the room then appears in the forum. Room Keys for game rooms time out after 20 minutes of inactivity.

During a game the Server would store a version of the whole playing map and pass XML payloads between the two clients move by move and battle by battle. If a Client drops out of the game then the Client would request the whole map data be refreshed from the Server when returning. Messages sent to the Server by the Client conducting the active players turn would be stashed until collected by the other Client. All Clients would keep an n counter of turns so far displayed. That way if one Client misses a message or a message is detected to be incorrect then there is an incremented order of messages on the Server and it is possible to backtrack, the Client requesting to be given the numbered move next on an incremented list. If there isn't one then the move hasn't been made yet.

Players who enter a game room that already has two participants "sitting" at a game are considered Spectators. Spectators can drop into a room and request entire game data, and then wait for game moves move by move and battle by battle, viewing the game but not participating in it, they simply observe counters as they progress.

During a game each Client needs to confirm with the Server that it has accepted the other Clients move. This is done using a tokenized challenge and response system. Tokenizing would be used where the Server passes to the Client a token that must be returned for verification of a move. In order to make sure that a Client never losses their confirmation token, that token would be stored on their machine in cookie.

Passive Clients, who are waiting for the other player to perform moves and battles, must request updates from the Server. This would be done with a ping request confirming its presence and asking for a new update to see if the other Client has made any new moves. A ping would be made every 5 seconds. In this way the active player Client generates moves and battles that are collected by the passive player Client. Once the active player concludes their turn the Client roles are switched, the active player becoming the passive and the passive player the active. The Server always needs to know which player is the active player so it won't acknowledge any false calls from the passive player Client or Spectators.

As far as game mechanics are concerned there are three crucial locations in the existing *Chevalier* code where transfer of game information between Server and Client are made.

1) In the Element Object there is a method called `setLoc()` which is used every time an element is moved on the map. At the end of this method a call would be added instructing the active player Client to send all information on the move to the Server to be collected by the passive player Client, who would receive the move and pass the data to its own `setLoc()` call, causing the move to appear on the passive computer.

2) In the CombatTable Object there is a method called `conductBattleV()` which is used to conduct every combat, be it a close "Battle" combat or distant "Shooting"

combat. At the end of this method a call would be added instructing the active player Client to send all information on the combat to the Server to be collected by the passive player Client, who would receive the move and pass the data to its own `conductBattleV()` call, causing the battle to appear on the passive computer.

3) The `update()` call in the main Chevalier Object controller would have to be substantially modified to send and retrieve whole map data with the Server when the game needs to be entirely refreshed. `update()` would also ping the Server every 5 seconds looking for updates from the active player Client that need to be refreshed to keep the Clint up to date with the game.

Of course, such a Proxy Server strategy for networked games would also require work constructing a general interface for game rooms and forums where players could meet and allocate games and send messages to each other, though there are standard formats for such interfaces.

Appendix E Application Code

All files listed are ActionScript 2 (.as) files. They are listed in the same order as discussed in "Chapter 6 Application Design." The last file listed is the *Asteroids* game object, which was used to create and test the *Animatem* engine that is part of the Presentation objects. One file, "IGameState.as" is an interface file specifying calls that must be resident in Game State objects.

Game Objects

Chevalier is available online at www.mocaz.com/games/Chevalier.html.

Chevalier.as

```
////////////////////////////////////////////////////////////////////////
//
//  Chevalier.as
//
//      AUTHOR: Russell Lowke
//              © Copyright 2006 Russell Lowke
//              All rights reserved.
//
//        Date: May 1st 2006
//
//  Description: The Chevalier Object is the root controller object and
//               the first created, from which all other objects are
//               made. The main task of the object is to initialize
//               and regulate other game objects, handle message
//               passing, and to establish the Flash Movie path to
//               the game map and controls.  These paths are
//               subsequently passed to the other objects and are the
//               key two paths for the whole game.
//
//               This object also regulates the game state, keeps track
//               of the mouse location, listens for keystrokes, keeps
//               track of the cursor state, invokes the sound object
//               and uses it to trigger sounds, invokes the Animatem
//               Engine and uses it to regulate animation,
//               initializes the playing map with associated Grid
//               Object, creates the two Player objects, keeps track
//               of the turn and the weather, and initializes the
//               Scroll object used to move Elements around on the
//               map.
//
```

```
//               The collision() and deactivated() methods triggered by
//               the Animatem Engine are resident here in this
//               object. In particular deactivate() is called
//               whenever an Element has finished moving, when the
//               map has finished animating, and when the Information
//               Scroll has finished opening or closing.
//
//       Method:
//
//       Chevalier() - Constructor
//  initialWeather() - Randomly generates weather at the beginning of
//                     the game
//     weatherDice() - Randomly calculates a change in the weather,
//                     this is called at the start of every turn.
//   aboutToEngage() - Check the move lists of the player's elements
//                     to see if any engagements are still yet to
//                     occur. This is needed at the end of a turn to
//                     fix a bug where battles that are still yet to
//                     be triggered by moving Elements are otherwise
//                     skipped over and not fought.
//      cnvPtToMap() - Convert a global screen location to a local
//                     location on the map.
//        ptGrdLoc() - Convert a screen location to a grid location.
//       collision() - When a sprite collides with another sprite this
//                     method is automatically called by the animator.
//                     Chevalier does not need to use collision
//                     detection of sprites so this method is empty.
//     deactivated() - When a sprite deactivates this method is
//                     automatically called by the animator.
//                     There are generally three cases of sprites
//                     deactivating.  The scroll deactivates when it
//                     has finished opening or closing.  The map
//                     deactivates when it finishes animating to a new
//                     location.  And an Element deactivates once it
//                     reaches a destination location it was moving to.
//       spinMapTo() - Tells the map to rotate to a new angle.
//      scaleMapTo() - Tells the map to scale to a new size. Growth
//                     constant e (2.718) is used to change the map
//                     location with the scaling, so scale looks like
//                     map has perspective.
//       changeMap() - Animates the map to a new position according to
//                     four parameters, scale, angle, location and
//                     duration of animation.
//  testForElement() - Count instances of elements along a list of up
//                     to seven points and return the one with the most
//                     hits.  Due to the width of Elements, with seven
//                     adjacent points there can never be more than
//                     three possible elements.
// testForElements() - Returns all instances of elements at various
//                     grid locations specified by an array of points.
//                     Any duplicates Elemets are removed from the
//                     list.
//         playSnd() - Tell the PlaySnd object to play a sound using,
//                     if necessary, a delay before playing. Many
//                     sounds, such as walking and fighting sounds,
//                     trigger one of a number of randomizedvariations,
//                     such sounds usually also utilize a randomized
```

```
//                      stagger,  allowing layering of sound to give the
//                      effect of a multitude. Some sounds, such as wind
//                      and rain, are set to loop perpetually.
//        useSmall() - Elements can have a "large" and a "small"
//                      version of their type the icon for improved
//                      clarity at  distant v close maps. Currently this
//                      is only used for the generals star icon, and the
//                      effect is only subtle.
//        useLarge() - Elements can have a "large" and a "small"
//                      version of their type the icon for improved
//                      clarity at distant v close maps.
//   removeFilters() - Remove all filter effects from elements, this
//                      allows for faster map animation.
//      setFilters() - Reinstate filter effects, putting back glow
//                      filters that were removed while the map was
//                      animating.
//normalizeElements() - Set the state (not status) of all the players
//                      elements to normal. Removes all functional
//                      state glows, roll highlights, and green
//                      shooting highlight. This is called at the end
//                      and beginning of a turn.
//switchActivePlayer() - Switches the player turn and increments the
//                      turn counter if moving from the second players
//                      turn to the first.
//       setCursor() - Changes the cursor.  Only works works if cursor
//                      not "frozen" using freezeCursor()
//                      Possible cursors are:
//                         watch, google, zoom_in, zoom_out, hand,
//                         grab, crosshair, battle, lft, tplft, tp,
//                         tpRht, rht, btmRht, btm, btmLft, wht_lft,
//                         wht_tplft, wht_tp, wht_tpRht, wht_rht,
//                         wht_btmRht, wht_btm, wht_btmLft
//    freezeCursor() - Freezes the cursor.  Useful for the watch
//                      cursor which should take priority over other
//                      cursor states.
//  unfreezeCursor() - Unfreeze the cursor and set it to a new state.
//          update() - Called constantly by onEnterFrame of the
//                      program, behaving much like a traditional main
//                      loop. Tells Animatem, PlaySnd and the active
//                      state object to update. If glow and bevel
//                      filters on the Elements need to be updated,
//                      for instance after the map has just animated,
//                      then those filters are told to update.
//updateScrollText() - Occasionally the information scroll needs to
//                      be updated due to sudden changes in the selected
//                      element(s) stats.
//      isFriendly() - return true if Element e is friendly to the
//                      current player
//      addElement() - Called from XML reader in Choose.as builds and
//                      element for a player
//           state() - Trigger a new game state
//
//        Notes:
//
```

```
class Chevalier {

    // instance members

    // window width
    private var _wWidth:Number;

    // window height
    private var _wHeight:Number;

    // animator object for game
    private var _a:Animatem;

    // sound object for game
    private var _snd:PlaySnd;

    // path of the root clip for the game map
    private var _pathMap:MovieClip;

    // path of the root clip for the game controls
    private var _pathCtrl:MovieClip;

    // sprite used for the battle map
    private var _mapSpr:Sprite;

    // 7 blocks = width of one 4cm element. 1 block is 8px x 8px.
    private var _blockSize:Number = 8;

    // location of the mouse
    private var _msePt:Point2D;

    // top left point pf the map
    private var _tpLft:Point2D;

    // Grid object containing locations of pieces
    private var _grid:Grid;

    // Animated display scroll
    private var _scroll:Scroll;

    // the turn number it is
    private var _turnN:Number = 0;

    // if true then cursor state frozen
    private var _cursorFrozen:Boolean = false;

    // finite state of the game, "None", "Choose", "StartTurn",
    // "Movement", "Shoots", "Battles"
    private var _state:String = "None";

    // obeject for the currently active state
    private var _stateObj:Object;

    // the "Choose" state object
    private var _choose:Choose;

    // the "StartTurn" state object
```

```
private var _startTurn:StartTurn;

// the "Movement" state object
private var _movement:Movement;

// the "Shoots" state object
private var _shoots:Shoots;

// the "Battles" state object
private var _battles:Battles;

// "Clear", "Overcast", "Rain"
private var _weather:String;

// "None", "Light", "Strong"
private var _windStregth:String;

// 0, 45, 90, 135, 180, 225, 270, 315
private var _windDirection:Number;

// true if weather has ever == "Rain"
private var _hadRain:Boolean = false;

// true for black, false for white
private var _playerTurn:Boolean = false;

// object for red player
private var _playerOne:Player;

// object for blue player
private var _playerTwo:Player;

// true if ready to load player two's army
private var _loadPlayerTwo:Boolean = false;

// if ! 0 then filters need to be restored
private var _restoreFilters:Number = 0;

//
//  Constructor
//
//   args:
//      wWidth     -- width of the game window
//      wHeight    -- height of the game window
//      pathMap    -- path to the animating game map
//      pathCtrl   -- path to general controls (usually _root)
//
public function Chevalier(wWidth:Number, wHeight:Number,
         pathMap:MovieClip, pathCtrl:MovieClip) {

    // window parameters
    _wWidth   = wWidth;
    _wHeight  = wHeight;
    // _quality = "MEDIUM";

    _pathMap  = pathMap;
    _pathCtrl = pathCtrl;
```

```
// set update call on enter frame
_pathMap.onEnterFrame = function() {
    _global.gChevalier.update();
};

// assign mouse listeners
var mouseListener:Object  = new Object();
mouseListener.onMouseDown = function() {
    _global.gChevalier.handleMouseDown();
}
mouseListener.onMouseUp   = function() {
    _global.gChevalier.handleMouseUp();
}
mouseListener.onMouseMove = function() {
    _global.gChevalier.handleMouseMove();
}
Mouse.addListener(mouseListener);

// assign key listeners
var myKeyListener:Object = new Object();
myKeyListener.onKeyDown = function() {
    _global.gChevalier.handleKeyDown();
}
myKeyListener.onKeyUp   = function() {
    _global.gChevalier.handleKeyUp();
}
Key.addListener(myKeyListener);

// set the cursor
setCursor("arrow");

// load sounds into PlaySnd object
_snd = new PlaySnd(["clk",
                    "denied",
                    "zoom_change",
                    "charge",
                    "playerOne",
                    "playerTwo",
                    "short_walk_1",
                    "short_walk_2",
                    "long_walk_1",
                    "long_walk_2",
                    "sword1",
                    "sword2",
                    "sword3",
                    "sword4",
                    "Arrow_1",
                    "Arrow_2",
                    "Arrow_3",
                    "Arrow_4",
                    "Arrow_Hit",
                    "drums",
                    "death1",
                    "death2",
                    "death3",
```

```
                    "death4",
                    "death5",
                    "death6",
                    "death7",
                    "runaway",
                    "Rain+Thunder_1",
                    "Rain+Thunder_2",
                    "Thunder_1",
                    "Thunder_2",
                    "wind"])

// point to store mouse position
_msePt = new Point2D(_xmouse, _ymouse);

// create animator
_a = new Animatem(_pathMap, 300, this);

// create a grid [4cm = element width = 7 blocks]
//  playing area is 1.8 meters x 1.2 meters
//  = 180cm / 4cm x 120cm / 4cm
//  = 45 element widths x 30 element widths
//  = 316 blocks x 210 blocks
var gmGrdWidth:Number  = 316;
var gmGrdHeight:Number = 210;

//  [1 block = 8 pixels]
// 2528 pixels (+2 blocks for edges = 2544)
var gmPxWidth:Number  = gmGrdWidth*_blockSize;

// 1680 pixels (+2 blocks for edges = 1696)
var gmPxHeight:Number = gmGrdHeight*_blockSize;

// set the top left pt the map
_tpLft = new Point2D(0 - gmPxWidth/2, 0 - gmPxHeight/2);
_grid = new Grid(_blockSize, _blockSize, _tpLft, gmGrdWidth,
         gmGrdHeight);

// assign the map to the animator
_mapSpr = _a.setSpriteN(1, _pathMap, _pathCtrl, -1);

// create player objects
Player.initialize(this, _grid);
_playerOne  = new Player("blk");
_playerTwo  = new Player("wht");

// assign to Elements static animator and grid objects
Element.initialize(this, _a, _grid);

// create the display scroll
_a.path = _pathCtrl;
_scroll = new Scroll(this, _a, _grid, _mapSpr, _pathCtrl);

// create the state objects
_choose    = new Choose(this, _pathCtrl, _mapSpr);
_startTurn = new StartTurn(this, _pathCtrl);
_shoots    = new Shoots(this, _grid, _pathCtrl);
_battles   = new Battles(this, _pathCtrl);
```

```
    _movement  = new Movement(this, _pathCtrl, _scroll, _mapSpr);

    // ensure animator path set to the map
    _a.path = _pathMap;

    // create the weather
    initialWeather();

    // start the map at 35%
    _mapSpr.scale = 30;
    _mapSpr.angle = 180;

    this.state = "Choose";
}

//
// initialWeather()
//
//  Randomly generates weather at the beginning of the game
//
private function initialWeather():Void {
    switch (Utils.randomInt(1, 10)) {
        case 2:
            _weather = "Overcast";
            break;
        default:
            _weather = "Clear";
            break;
    }
    switch (Utils.randomInt(1, 4)) {
        case 1:
            _windStregth = "Strong";
            break;
        case 2: case 3:
            _windStregth = "Light";
            break;
        case 4:
            _windStregth = "None";
            break;
    }
    _windDirection = Utils.randomInt(0, 7)*45;
}

//
// weatherDice()
//
//  Randomly calculates a change in the weather,
//  this is called at the start of every turn.
//
public function weatherDice():Void {
    //
    // wind changes direction
    if (Utils.randomInt(1, 5) == 1) {
        if (Utils.randomInt(0, 1) == 0) {
            _windDirection -= 45;
        } else {
            _windDirection += 45;
```

```
    }
}
_windDirection = Utils.cleanAngle(_windDirection);

// windy day?
if (Utils.randomInt(1, 5) == 1) {
    switch (_windStregth) {
        case "None":
            _windStregth = "Light";
            break;
        case "Light":
            if (Utils.randomInt(0, 1) == 0) {
                _windStregth = "Light";
            } else {
                _windStregth = "Strong";
                playSnd("wind");
            }
            break;
        case "Strong":
            _windStregth = "Light";
            break;
    }
}

// looks like rain...
var chanceOfChange:Number;
if (_weather == "Overcast") {
    chanceOfChange = 2;
} else if (_weather == "Rain") {
    chanceOfChange = 5;
} else {
    chanceOfChange = 10;
}
if (Utils.randomInt(1, chanceOfChange) == 2 ||
   (_weather == "Overcast" && _hadRain)) {
    switch (_weather) {
        case "Clear":
            if (! _hadRain) { _weather = "Overcast"; }
            break;
        case "Overcast":
            if (_hadRain) {
                _weather = "Clear";
            } else {
                _weather = "Rain";  _hadRain = true;
                playSnd("rainAndThunder");
            }
            break;
        case "Rain":
            _weather = "Overcast";
            break;
    }
}

// if not raining or windy then cease any looping audio
if ( ! (_weather == "Rain" || _windStregth == "Strong") ) {
    // sending an undefined loop stops any looping audio
    _snd.loop();
```

```
    }
}

//
// aboutToEngage()
//
//  Check the move lists of the player's elements to
//  to see if any engagements are still yet to occur.
//  This is needed at the end of a turn to fix a bug
//  where battles that are still yet to be triggered by
//  moving Elements are otherwise skipped over and not fought.
//
//   return     -- true if there are battles yet to be triggered
//
public function aboutToEngage():Boolean {

    var elements:Array = Player.active.all;
    for (var i:Number = 0; i < elements.length; ++i) {
        if (elements[i].aboutToEngage()) { return true; }
    }
    return false;
}

//
// cnvPtToMap()
//
//  Convert a global screen location to a local
//  location on the map.
//
//   args:
//      loc -- location being converted
//
//   return -- equivilent map location
//
public function cnvPtToMap(pt:Point2D):Point2D {
    var myPoint:Object = { x:pt.x, y:pt.y };
    _pathMap.globalToLocal(myPoint);
    return new Point2D(myPoint.x, myPoint.y);
}

//
// ptGrdLoc()
//
//  Convert a screen location to a grid location
//
//   args:
//      loc -- location being converted
//
//   return -- equivilent map _grid location
//
public function ptGrdLoc(loc:Point2D):Point2D {
    return _grid.ptToGridLoc(cnvPtToMap(loc));
}

//
// collision()
//
```

```
//  When a sprite collides with another sprite this method
//  is automatically called by the animator.
//  Chevalier does not need to use collision detection
//  of sprites so this method is empty.
//
public function collision(src:Number, trg:Number, str:String) {
}

//
// deactivated()
//
//  When a sprite deactivates this method is automatically
//  called by the animator. There are generally three cases
//  of sprites deactivating.  The scroll deactivates when
//  it has finished opening or closing.  The map deactivates when
//  it finishes animating to a new location.  And an Element
//  deactivates once it reaches a destination location it was
//  moving to.
//
//   args:
//      n   -- channel number of the deactivated sprite
//
public function deactivated(n:Number):Void {
    if (n == _scroll.sprite.number) {
        _scroll.scrollConcluded();
    } else if (n == _mapSpr.number) {
        _mapSpr.active = -1;    // keep map active after rotating

        // switch to detailed icons unless zoomed out far
        if (_mapSpr.scale <= 35) {
            useSmall();
        } else {
            useLarge();
        }
        _restoreFilters = getTimer();
        unfreezeCursor("arrow");

    } else {

        // element has arrived
        _a.getTag(n).atDestination();
    }
}

//
// spinMapTo()
//
//  Tells the map to rotate to a new angle.
//
//   args:
//       duration   -- duration of animation (in ticks)
//          angle   -- new angle to animate map to
//
public function spinMapTo(duration:Number, angle:Number)  {
    changeMap(duration, _mapSpr.scale, angle);
}
```

```
//
// scaleMapTo()
//
//  Tells the map to scale to a new size.
//  Growth constant e (2.718) is used to change the map location
//  with the scaling, so scale looks like map has perspective.
//
//   args:
//       duration   -- duration of animation (in ticks)
//          scale   -- new scale to animate map to
//
public function scaleMapTo(duration:Number, scale:Number) {


    var mpLoc:Point2D = _mapSpr.loc;
    var loc:Point2D   = new Point2D(Stage.width/2, Stage.height/2);

    loc.subtract(mpLoc);

    // divide by growth constant e,  2.718
    loc.divide(2.718);

    if (_mapSpr.scale > scale) {
        mpLoc.add(loc);
    } else {
        mpLoc.subtract(loc);
    }

    changeMap(duration, scale, undefined, mpLoc);
}

//
// changeMap()
//
//  Animates the map to a new position according to four
// parameters,
//  scale, angle, location and duration of animation
//
//   args:
//       duration   -- duration of animation (in ticks)
//          scale   -- scale to animate map to
//          angle   -- angle to animate map to
//            loc   -- new location of the map
//
public function changeMap(duration:Number, scale:Number,
         angle:Number, loc:Point2D):Void {

    if (angle != undefined) {
        angle = Utils.cleanAngle(angle);
    }

    // if scale is 50% or less then switch to less detailed icons
    if (scale <= 50 ) { useSmall(); }

    if (scale == _mapSpr.scale) { scale = undefined; }
    if (angle == _mapSpr.angle) { angle = undefined; }
```

```
    // remove all filters (i.e. bevels) for animation speed
    if (scale != undefined || angle != undefined) {
        removeFilters();
    }

    if (scale != undefined) {
        _a.scaleInTime(_mapSpr.number, scale, duration);
    }
    if (angle != undefined) {
        _a.rotateInTime(_mapSpr.number, angle, duration);
    }
    if (loc   != undefined) {
        _a.goToLocInTme(_mapSpr.number, loc, duration);
    }

    _movement.updatePalette();
}

//
// testForElement()
//
//  Count instances of elements along a list of up to seven points
//  and return the one with the most hits.
//  Due to the width of Elements, with seven adjacent points
//  there can never be more than three possible elements.
//
//   args:
//      ptList  -- list of up to seven adjacent points
//
public function testForElement(ptList:Array):Element {

    if (ptList.length > 7) {
        throw new Error("too many test points");
    }

    var e:Element;   var eCount:Number   = 0;
    var ee:Element;  var eeCount:Number  = 0;
    var eee:Element; var eeeCount:Number = 0;

    for (var i:Number = 0; i < ptList.length; ++i) {
        var val = _grid.getAt(ptList[i]);
        if (val instanceof Element) {
            if (e == undefined) {
                e = val; ++eCount;
            } else if (e.spriteN == val.spriteN) {
                ++eCount;
            } else if ( ee == undefined ) {
                ee = val; ++eeCount;
            } else if (ee.spriteN == val.spriteN) {
                ++eeCount;
            } else if ( eee == undefined ) {
                eee = val; ++eeeCount;
            } else if (eee.spriteN == val.spriteN) {
                ++eeeCount;
            }
        }
    }
```

```
    // return the largest of the three
    if (eeCount > eCount) {
        e = ee; eCount = eeCount;
    }
    if (eeeCount > eCount) { e = eee; }

    return e;
}

//
// testForElements()
//
//  Returns all instances of elements at various grid
//  locations specified by an array of points.
//  Any duplicates Elemets are removed from the list.
//
//   args:
//      ptList  -- list of points to check on _grid
//
//   return     -- list of Elements found on _grid
//
public function testForElements(ptList:Array):Array {

    var r:Array = new Array();
    var haveIt:Boolean;

    for (var i:Number = 0; i < ptList.length; ++i) {
        var val = _grid.getAt(ptList[i]);

        if (val instanceof Element) {

            // check list for element
            haveIt = false;
            for (var j:Number = 0; j < r.length; ++j) {
                if (val.spriteN == r[j].spriteN) {
                    haveIt = true; break;
                }
            }
            if (! haveIt) { r.push(val); }
        }
    }

    return r;
}

//
// playSnd()
//
//  Tell the PlaySnd object to play a sound
//  using, if necessary, a delay before playing.
//  Many sounds, such as walking and fighting sounds,
//  trigger one of a number of randomized variations,
//  such sounds usually also utilize a randomized
//  stagger, allowing layering of sound to give
//  the effect of a multitude.
//  Some sounds, such as wind and rain, are set to
```

```
//  loop perpetually.
//
//   args:
//      snd -- name of sound to trigger
//    delay -- time (in seconds) to delay before playing
//
public function playSnd(snd:String, delay:Number):Void {

    // deffine stagger time and volume
    var stagger:Number = 0.5;
    var vol:Number = 100;

    switch (snd) {

        case "wind":
            _snd.loop("wind", vol);
            break;

        case "rainAndThunder":
            _snd.loop("Rain+Thunder_"+ Utils.randomInt(1, 2), vol);
            _snd.play("Thunder_" + Utils.randomInt(1, 2), vol,
                      delay);
            break;

        case "arrow":
            _snd.play("Arrow_" + Utils.randomInt(1, 4), vol,
                      delay);
            break;

        case "sword":
            var rnd:Number = Utils.randomInt(1, 6);
            switch (rnd) {
                case 2: case 3:  case 4:
                    _snd.play("sword" + rnd, vol, delay);
                    break;
                default:
                    _snd.play("sword1", vol, delay);
                    break;
            }
            break;

        case "death":
            if(delay) { vol *= 0.7; }
            _snd.play("death" + Utils.randomInt(1, 7), vol, delay);
            break;

        case "shortWalk":
            _snd.play("short_walk_" + Utils.randomInt(1, 2), vol,
                       delay, stagger);
            break;

        case "longWalk":
            switch (Utils.randomInt(1, 4)) {
                case 1:
                    _snd.play("long_walk_1", vol, delay, stagger);
                    break;
                case 2:
```

```
                    _snd.play("long_walk_2", vol, delay, stagger);
                    break;
                case 3:
                    _snd.play("short_walk_1", vol, delay,
                            stagger);
                    break;
                case 4:
                    _snd.play("short_walk_2", vol, delay,
                            stagger);
                    break;
            }
            break;

        default:
            _snd.play(snd, vol, delay);
            break;
    }
}

//
// useSmall()
//
//  Elements can have a "large" and a "small" version of their type
//  the icon for improved clarity at distant v close maps
//  Currently this is only used for the generals star icon, and the
//
//  effect is only subtle
//
private function useSmall():Void {
    var playerOnesStuff:Array = _playerOne.all;
    var playerTwosStuff:Array = _playerTwo.all;

    for (var i:Number = 0; i < playerOnesStuff.length; ++i) {
        playerOnesStuff[i].small();
    }
    for (var i:Number = 0; i < playerTwosStuff.length; ++i) {
        playerTwosStuff[i].small();
    }
}

//
// useLarge()
//
//  Elements can have a "large" and a "small" version of their type
//  the icon for improved clarity at distant v close maps
//
private function useLarge():Void {
    var playerOnesStuff:Array = _playerOne.all;
    var playerTwosStuff:Array = _playerTwo.all;

    for (var i:Number = 0; i < playerOnesStuff.length; ++i) {
        playerOnesStuff[i].large();
    }
    for (var i:Number = 0; i < playerTwosStuff.length; ++i) {
        playerTwosStuff[i].large();
    }
}
```

```
//
// removeFilters()
//
//  Remove all filter effects from elements, this allows
//  for faster map animation
//
private function removeFilters():Void {
    var playerOnesStuff:Array = _playerOne.all;
    var playerTwosStuff:Array = _playerTwo.all;

    for (var i:Number = 0; i < playerOnesStuff.length; ++i) {
        playerOnesStuff[i].removeFilters();
    }
    for (var i:Number = 0; i < playerTwosStuff.length; ++i) {
        playerTwosStuff[i].removeFilters();
    }
}

//
// setFilters()
//
//  Reinstate filter effects, putting back glow filters that
//  were removed while the map was animating
//
private function setFilters():Void {
    var playerOnesStuff:Array = _playerOne.all;
    var playerTwosStuff:Array = _playerTwo.all;

    for (var i:Number = 0; i < playerOnesStuff.length; ++i) {
        playerOnesStuff[i].setFilters();
    }
    for (var i:Number = 0; i < playerTwosStuff.length; ++i) {
        playerTwosStuff[i].setFilters();
    }

    _restoreFilters = 0;
}

//
// normalizeElements()
//
//  Set the state (not status) of all the players elements to
// normal
//  Removes all functional state glows, roll highlights, and
//  green shooting highlight.
//  This is called at the end and beginning of a turn
//
public function normalizeElements():Void {
    for (var i:Number = 0; i < Player.active.all.length; ++i) {
        Player.active.all[i].stateNormal();
    }
}

//
// switchActivePlayer()
//
```

```
//  Switches the player turn and increments
//  the turn counter if moving from the
//  second players turn to the first
//
public function switchActivePlayer():Void {
    _playerTurn = ! _playerTurn;

    if (_playerTurn) { ++_turnN; }

    if (_playerTurn) {
        Player.active = _playerOne;
    } else {
        Player.active = _playerTwo;
    }
}

//
// setCursor()
//
//  Changes the cursor.  Only works works if cursor
//  not "frozen" using freezeCursor()
//  Possible cursors are:
//      watch, google, zoom_in, zoom_out, hand,
//      grab, crosshair, battle, lft, tplft, tp,
//      tpRht, rht, btmRht, btm, btmLft, wht_lft,
//      wht_tplft, wht_tp, wht_tpRht, wht_rht,
//      wht_btmRht, wht_btm, wht_btmLft
//
//   args:
//      val     -- name of cursor to use
//
public function setCursor(val:String):Void {

    if (! _cursorFrozen) {

        if (val == "arrow") {
            Mouse.show();
            _pathCtrl._cursor.gotoAndStop("none");
        } else {
            Mouse.hide();
            _pathCtrl._cursor.gotoAndStop(val);
        }

        updateAfterEvent();
    }
}

//
// freezeCursor()
//
//  Freezes the cursor.  Useful for the watch cursor which
//  should take priority over other cursor states
//
//   args:
//      val     -- name of cursor to freeeze with
//
public function freezeCursor(val:String):Void {
```

```
    unfreezeCursor(val);
    _cursorFrozen = true;
}

//
// unfreezeCursor()
//
//  Unfreeze the cursor and set it to a new state
//
//   args:
//      val     -- name of cursor to use
//
public function unfreezeCursor(val:String):Void {
    _cursorFrozen = false;
    setCursor(val);
}

//
// update()
//
//  Called constantly by onEnterFrame of the program, behaving much
//  like a traditional main loop. Tells Animatem, PlaySnd and the
//  active state object to update. If glow and bevel filters on the
//  Elements need to be updated, for instance after the map has
//  just animated, then those filters are told to update.
//
public function update():Void {

    // update animation engine sprites
    _a.update();

    // update sounds
    _snd.update();

    // update filters if necessary.
    if (_restoreFilters && getTimer() > _restoreFilters + 60) {
        setFilters();
    }

    _stateObj.update();
}

//
// updateScrollText()
//
//   Occasionally the information scroll needs to be updated
//   due to sudden changes in the selected element(s) stats.
//
public function updateScrollText():Void {
    _scroll.updateScrollText();
}

//
// isFriendly()
//
//   return true if Element e is friendly to the current player
//
```

```
//   args:
//            e     -- Element being tested
//
//       return     -- true if it's a friendly Element;
//
public function isFriendly(e:Element):Boolean {
    return (e.player.thePlayer == Player.active.thePlayer);
}

//
// addElement()
//
//   Called from XML reader in Choose.as
//   builds and element for a player
//
//   args:
//    obj -- object contains all parameters being passed from XML
//
public function addElement(obj:Object):Void {

    var player:Player;

    if (obj.player == "One") {
        player = _playerOne;
    } else if (obj.player == "Two") {
        player = _playerTwo;
    } else {
        player = _loadPlayerTwo ? _playerTwo : _playerOne;
    }

    var gen:String = obj.general;
    if (gen == "undefined") { gen = undefined; }

    var ply:String = obj.player;
    if (ply == "undefined") { ply = undefined; }

    var regular:Boolean;
    if (obj.regular == "true") {    // FLASH BUG interpereting bool
        regular = true;
    } else {
        regular = false;
    }

    player.add(obj.command,
               obj.type,
               obj.grade,
               regular,
               obj.name,
               obj.icon,
               new Point2D(obj.locX, obj.locY),
               obj.angle,
               gen,
               ply);
}

//
// state()
```

```
//
//   Trigger a new game state
//
//   args:
//          val     -- new state to be triggered
//
public function set state(val:String):Void {

    if (_state != val) {
        // elements should be normalized between all states
        normalizeElements();

        // clear settings of the old state
        _stateObj.end();

        // establish the new state
        _state = val;
        switch (_state) {
            case "Choose":      _stateObj = _choose;    break;
            case "StartTurn":   _stateObj = _startTurn; break;
            case "Movement":    _stateObj = _movement;  break;
            case "Shoots":      _stateObj = _shoots;    break;
            case "Battles":     _stateObj = _battles;   break;
        }

        _stateObj.start();
    }
}

public function releaseUpdate():Void {
    _a.releaseUpdate();
}

//
// message passing to the active state object
public function rollBtn(btn:String, val):Void {
    _stateObj["roll" + btn](val);
}
public function pressBtn(btn:String, val):Void {
    _stateObj["press" + btn](val);
}
public function pushBtn(btn:String, val):Void {
    _stateObj["btn" + btn](val);
}
public function doCmd(cmd:String, val):Void {
    _stateObj[cmd](val);
}
public function handleKeyDown():Void {
    _stateObj.keyDown();
}
public function handleKeyUp():Void {
    _stateObj.keyUp();
}
public function handleMouseDown():Void {
    _stateObj.mouseDown();
}
public function handleMouseUp():Void {
```

```
        _stateObj.mouseUp();
    }
    public function handleMouseMove():Void {
        _msePt = new Point2D(_xmouse, _ymouse);
        _pathCtrl._cursor._x = _msePt.x;
        _pathCtrl._cursor._y = _msePt.y;
        _stateObj.mouseMove();
    }

    //
    // accessors
    public function get playerTurn():Boolean {
        return _playerTurn;
    }
    public function get playerActive():Player {
        return Player.active;
    }
    public function get playerOne():Player {
        return _playerOne;
    }
    public function get playerTwo():Player {
        return _playerTwo;
    }
    public function get turnN():Number {
        return _turnN;
    }
    public function get restoreFiltersTime():Number {
        return _restoreFilters;
    }
    public function get state():String {
        return _state;
    }
    public function get mapSpr():Sprite {
        return _mapSpr;
    }
    public function get mseGrdLoc():Point2D {
        return ptGrdLoc(_msePt);
    }
    public function get blockSize():Number {
        return _blockSize;
    }
    public function get msePt():Point2D {
        return _msePt;
    }
    public function get movementObj():Movement {
        return _movement;
    }
    public function get pathCtrl():MovieClip {
        return _pathCtrl;
    }
    public function get pathMap():MovieClip {
        return _pathMap;
    }
    public function get weather():String {
        return _weather;
    }
    public function get windStregth():String {
```

```
        return _windStregth;
    }
    public function get windDirection():Number {
        return _windDirection;
    }

    //
    // mutators
    public function set loadPlayerTwo(val:Boolean):Void {
        _loadPlayerTwo = val;
    }
}
```

Player.as

```
////////////////////////////////////////////////////////////////////////
//
//  Player.as
//
//       AUTHOR: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: The Chevalier Object will create two instances of
//               Player Object, each maintaining all game information
//               pertaining to the player, such as player color,
//               player army (i.e. "Crusader"), Elements in the
//               left command, right command, center command,
//               eliminated elements, which Elements are the
//               commanding Elements for each command, morale values
//               for each command, and default map and scroll
//               positions for that player.
//
//               Player Object also has a static initialization that
//               creates all the template data used for each of the
//               three Element base depths when an Element is
//               created. This static Footprint data is constantly
//               referenced as a starting point by all Elements as
//               they move, this way they don't have to reconstruct
//               Footprint data from scratch. Moreover, whenever an
//               Element is created for a player that creation is
//               done through the player object using the add()
//               method. The template Footprints are stored here in a
//               static form as it grants the add() method easy
//               access to them.
//
//       Method:
//      initialize() - Assign objects static variables, these are
//                     mostly base footprint definitions.
//          Player() - Constructor
//             add() - Add a new Element to this player's army.
//        rollPIPs() - Roll player initiative dice for this player
//    setAsGeneral() - Aassign an element as a general of a command
//     elementDead() - Remove an element from command lists and add to
//                     dead pile
//    getCmdStatus() - Return a string describing the morale status of
//                     a command. Sometimes it's only important to know
//                     if the command is shattered, broken, or
//                     dispirited, [flag = false] such as during
//                     battle.
//  getMoraleValue() - Get the morale of one of this players commands
//getMoralePercent() - Get the morale % of one of this players commands
//
//        Notes:

class Player {

    // static members
```

```
// game object
private static var _game:Chevalier;

// "game board" grid object with location of game pieces
private static var _grid:Grid;

// the active player who is having their turn
private static var _active:Player;

private static var _footPrint3:Array;
private static var _footPrint4:Array;
private static var _footPrint7:Array;

// all living elements in game
private static var _everybody:Array;

// TEMP point subtracted to bring everybody closer
private static var _mvCloserPt:Point2D;

// instance members
// donates player, "Crusader", "Saracen", "Macedonian", "Persian",
// ... etc
private var _player:String;

// color of pieces, "blk" for black, "wht" for white
private var _color:String;


// general commanding the left
private var _leftGen:Element;

// general commanding the center
private var _centerGen:Element;

// general commanding the right
private var _rightGen:Element;


// list of elements in left command
private var _left:Array;

// list of elements in center command
private var _center:Array;

// list of elements in right command
private var _right:Array;

// Moral equivilents in left command at game start
private var _startLeftMoral:Number   = 0;

// Moral equivilents remianing in left command
private var _leftMoral:Number;

// Moral equivilents in center command at game start
private var _startCenterMoral:Number = 0;
```

```
// Moral equivilents remianing in center command
private var _centerMoral:Number;

// Moral equivilents in right command at game start
private var _startRightMoral:Number  = 0;

// Moral equivilents remianing in right command
private var _rightMoral:Number;

// Player Initiative points for left command
private var _PIPsLeft:Number;

// Player Initiative points for center command
private var _PIPsCenter:Number;

// Player Initiative points for right command
private var _PIPsRight:Number;


// list of all living elements in army
private var _all:Array;

// list of dead elements
private var _dead:Array;


// last map position used by player
private var _mapPos:Point2D;

// last map scale usewd by player
private var _mapScale:Number;

// last map angle used by player
private var _mapAngle:Number;

// position player keeps the scroll
private var _scrollPos:Point2D;

//
// initialize()
//
//  Assign objects static variables, these are mostly base
// footprint definitions.
//
//   args:
//     game  -- base horizontal footprint ("North" facing 270
// degrees)
//     grid  -- base diagonal footprint ("North East" facing 315
// degrees)
//
public static function initialize(game:Chevalier, grid:Grid) {

    _game       = game;
    _grid       = grid;
    _everybody  = new Array();
    _mvCloserPt = new Point2D(0, 23);
    // _mvCloserPt = new Point2D(0, 0);
```

```
// pt coverage for a horizontal (270 degrees, facing "North")
// 3 deep element
var h_footPrint3:Footprint = new Footprint(
   [                          // full sqaures
    new Point2D( 0,   0),  // front key (registration)
    new Point2D( 0,  +2),  // back key
    new Point2D(-3,   0),  // left key
    new Point2D(+3,   0),  // right key
    new Point2D(-3,  +2),  // back left corner
    new Point2D(+3,  +2),  // back right corner
    new Point2D(-2,   0),
    new Point2D(-1,   0),
    new Point2D(+1,   0),
    new Point2D(+2,   0),
    new Point2D(-3,  +1),
    new Point2D(-2,  +1),
    new Point2D(-1,  +1),
    new Point2D( 0,  +1),
    new Point2D(+1,  +1),
    new Point2D(+2,  +1),
    new Point2D(+3,  +1),
    new Point2D(-2,  +2),
    new Point2D(-1,  +2),
    new Point2D(+1,  +2),
    new Point2D(+2,  +2)],
   [],                         // 1/2 sqaures
   [new Point2D( 0,  -1),   // front
    new Point2D(-3,  -1),
    new Point2D(-2,  -1),
    new Point2D(-1,  -1),
    new Point2D(+1,  -1),
    new Point2D(+2,  -1),
    new Point2D(+3,  -1)],
   [new Point2D( 0,  +3),   // back
    new Point2D(-3,  +3),
    new Point2D(-2,  +3),
    new Point2D(-1,  +3),
    new Point2D(+1,  +3),
    new Point2D(+2,  +3),
    new Point2D(+3,  +3)],
   [new Point2D(-4,   0),   // left
    new Point2D(-4,  +1),
    new Point2D(-4,  +2)],
   [new Point2D(+4,   0),   // right
    new Point2D(+4,  +1),
    new Point2D(+4,  +2)],
    new Point2D(-4,  -1),   // top left outside
    new Point2D(+4,  -1),   // top right outside
    new Point2D(-7,  +2),   // shift left pt
    new Point2D(+7,  +2),   // shift right pt
    new Point2D(-3,  +3),   // flank left contact pt
    new Point2D(+3,  +3),   // flank right contact pt
    new Point2D( 0,  -2));  // modifier
```

```
// pt coverage for a diagonal (315 degrees, facing "North
// East") 3 deep element
var d_footPrint3:Footprint = new Footprint(
   [                          // full sqaure
    new Point2D( 0,   0),    // front key (registration)
    new Point2D(-1,  +1),    // back key
    new Point2D(-2,  -2),    // left key
    new Point2D(+2,  +2),    // right key
    new Point2D(-3,  -1),    // back left corner
    new Point2D(+1,  +3),    // back right corner
    new Point2D(-1,  -1),
    new Point2D(+1,  +1),
    new Point2D(-2,  -1),
    new Point2D(-1,   0),
    new Point2D( 0,  +1),
    new Point2D(+1,  +2),
    new Point2D(-2,   0),
    new Point2D( 0,  +2)],
   [new Point2D(-2,  -3),    // top left corner // 1/2 sqaures
    new Point2D(+3,  +2),    // top right corner
    new Point2D(-4,  -1),    // bottom left corner
    new Point2D(+1,  +4),    // bottom right corner
    new Point2D(-1,  -2),
    new Point2D( 0,  -1),
    new Point2D(+1,   0),
    new Point2D(+2,  +1),
    new Point2D(+2,  +3),
    new Point2D( 0,  +3),
    new Point2D(-1,  +2),
    new Point2D(-2,  +1),
    new Point2D(-3,   0),
    new Point2D(-3,  -2)],
   [new Point2D(+1,  -1),    // front
    new Point2D(-1,  -3),
    new Point2D( 0,  -2),
    new Point2D(+2,   0),
    new Point2D(+3,  +1)],
   [new Point2D(-2,  +2),    // back
    new Point2D(-4,   0),
    new Point2D(-3,  +1),
    new Point2D(-1,  +3),
    new Point2D( 0,  +4)],
   [new Point2D(-3,  -3),    // left
    new Point2D(-4,  -2)],
   [new Point2D(+3,  +3),    // right
    new Point2D(+2,  +4)],
    new Point2D(-2,  -4),    // top left outside
    new Point2D(+4,  +2),    // top right outside
    new Point2D(-6,  -4),    // shift left pt
    new Point2D(+4,  +6),    // shift right pt
    new Point2D(-4,  0),     // flank left contact pt
    new Point2D( 0,  +4),    // flank right contact pt
    new Point2D(+2,  -2));   // modifier

_footPrint3 = createBaseFootPrint(h_footPrint3, d_footPrint3);

// pt coverage for a horizontal (270 degrees, facing "North")
```

```
// 4 deep element
var h_footPrint4:Footprint = new Footprint(
   [                          // full sqaure
    new Point2D( 0,   0),    // front key (registration)
    new Point2D( 0,  +3),    // back key
    new Point2D(-3,   0),    // left key
    new Point2D(+3,   0),    // right key
    new Point2D(-3,  +3),    // back left  corner
    new Point2D(+3,  +3),    // back right corner
    new Point2D(-2,   0),
    new Point2D(-1,   0),
    new Point2D(+1,   0),
    new Point2D(+2,   0),
    new Point2D(-3,  +1),
    new Point2D(-2,  +1),
    new Point2D(-1,  +1),
    new Point2D( 0,  +1),
    new Point2D(+1,  +1),
    new Point2D(+2,  +1),
    new Point2D(+3,  +1),
    new Point2D(-3,  +2),
    new Point2D(-2,  +2),
    new Point2D(-1,  +2),
    new Point2D( 0,  +2),
    new Point2D(+1,  +2),
    new Point2D(+2,  +2),
    new Point2D(+3,  +2),
    new Point2D(-2,  +3),
    new Point2D(-1,  +3),
    new Point2D(+1,  +3),
    new Point2D(+2,  +3)],
   [],                        // 1/2 sqaures
   [new Point2D( 0,  -1),    // front
    new Point2D(-3,  -1),
    new Point2D(-2,  -1),
    new Point2D(-1,  -1),
    new Point2D(+1,  -1),
    new Point2D(+2,  -1),
    new Point2D(+3,  -1)],
   [new Point2D( 0,  +4),    // back
    new Point2D(-3,  +4),
    new Point2D(-2,  +4),
    new Point2D(-1,  +4),
    new Point2D(+1,  +4),
    new Point2D(+2,  +4),
    new Point2D(+3,  +4)],
   [new Point2D(-4,   0),    // left
    new Point2D(-4,  +1),
    new Point2D(-4,  +2),
    new Point2D(-4,  +3)],
   [new Point2D(+4,   0),    // right
    new Point2D(+4,  +1),
    new Point2D(+4,  +2),
    new Point2D(+4,  +3)],
    new Point2D(-4,  -1),    // top left outside
    new Point2D(+4,  -1),    // top right outside
    new Point2D(-7,  +3),    // shift left pt
```

```
    new Point2D(+7,  +3),    // shift right pt
    new Point2D(-3,  +3),    // flank left contact pt
    new Point2D(+3,  +3),    // flank right contact pt
    new Point2D( 0,  -2));   // modifier

// pt coverage for a diagonal (315 degrees, facing "North
// East") 4 deep element
var d_footPrint4:Footprint = new Footprint(
   [                         // full sqaure
    new Point2D( 0,   0),    // front key (registration)
    new Point2D(-2,  +2),    // back key
    new Point2D(-2,  -2),    // left key
    new Point2D(+2,  +2),    // right key
    new Point2D(-4,   0),    // back left corner
    new Point2D( 0,  +4),    // back right corner
    new Point2D(-1,  -1),
    new Point2D(+1,  +1),
    new Point2D(-2,  -1),
    new Point2D(-1,   0),
    new Point2D( 0,  +1),
    new Point2D(+1,  +2),
    new Point2D(-3,  -1),
    new Point2D(-2,   0),
    new Point2D(-1,  +1),
    new Point2D( 0,  +2),
    new Point2D(+1,  +3),
    new Point2D(-3,   0),
    new Point2D(-2,  +1),
    new Point2D(-1,  +2),
    new Point2D( 0,  +3),
    new Point2D(-3,  +1),
    new Point2D(-1,  +3)],
   [new Point2D(-2,  -3),    // top left corner // 1/2 sqaures
    new Point2D(+3,  +2),    // top right corner
    new Point2D(-5,   0),    // bottom left corner
    new Point2D( 0,  +5),    // bottom right corner
    new Point2D(-1,  -2),
    new Point2D( 0,  -1),
    new Point2D(+1,   0),
    new Point2D(+2,  +1),
    new Point2D(+2,  +3),
    new Point2D(+1,  +4),
    new Point2D(-1,  +4),
    new Point2D(-2,  +3),
    new Point2D(-3,  +2),
    new Point2D(-4,  +1),
    new Point2D(-4,  -1),
    new Point2D(-3,  -2)],
   [new Point2D(+1,  -1),    // front
    new Point2D(-1,  -3),
    new Point2D( 0,  -2),
    new Point2D(+2,   0),
    new Point2D(+3,  +1)],
   [new Point2D(-3,  +3),    // back
    new Point2D(-5,  +1),
    new Point2D(-4,  +2),
    new Point2D(-2,  +4),
```

```
    new Point2D(-1,  +5)],
   [new Point2D(-3,  -3),    // left
    new Point2D(-4,  -2),
    new Point2D(-5,  -1)],
   [new Point2D(+3,  +3),    // right
    new Point2D(+2,  +4),
    new Point2D(+1,  +5)],
    new Point2D(-2,  -4),    // top left outside
    new Point2D(+4,  +2),    // top right outside
    new Point2D(-7,  -3),    // shift left pt
    new Point2D(+3,  +7),    // shift right pt
    new Point2D(-4,  0),     // flank left contact pt
    new Point2D( 0,  +4),    // flank right contact pt
    new Point2D(+2,  -2));   // modifier

_footPrint4 = createBaseFootPrint(h_footPrint4, d_footPrint4);


// pt coverage for a horizontal (270 degrees, facing "North")
// 7 deep element
var h_footPrint7:Footprint = new Footprint(
   [                         // full sqaure
    new Point2D( 0,   0),    // front key (registration)
    new Point2D( 0,  +6),    // back key
    new Point2D(-3,   0),    // left key
    new Point2D(+3,   0),    // right key
    new Point2D(-3,  +6),    // back left corner
    new Point2D(+3,  +6),    // back right corner
    new Point2D(-3,  +3),    // left middle
    new Point2D(+3,  +3),    // right middle
    new Point2D(-2,   0),
    new Point2D(-1,   0),
    new Point2D(+1,   0),
    new Point2D(+2,   0),
    new Point2D(-3,  +1),
    new Point2D(-2,  +1),
    new Point2D(-1,  +1),
    new Point2D( 0,  +1),
    new Point2D(+1,  +1),
    new Point2D(+2,  +1),
    new Point2D(+3,  +1),
    new Point2D(-3,  +2),
    new Point2D(-2,  +2),
    new Point2D(-1,  +2),
    new Point2D( 0,  +2),
    new Point2D(+1,  +2),
    new Point2D(+2,  +2),
    new Point2D(+3,  +2),
    new Point2D(-2,  +3),
    new Point2D(-1,  +3),
    new Point2D( 0,  +3),
    new Point2D(+1,  +3),
    new Point2D(+2,  +3),
    new Point2D(-3,  +4),
    new Point2D(-2,  +4),
    new Point2D(-1,  +4),
    new Point2D( 0,  +4),
```

```
    new Point2D(+1,  +4),
    new Point2D(+2,  +4),
    new Point2D(+3,  +4),
    new Point2D(-3,  +5),
    new Point2D(-2,  +5),
    new Point2D(-1,  +5),
    new Point2D( 0,  +5),
    new Point2D(+1,  +5),
    new Point2D(+2,  +5),
    new Point2D(+3,  +5),
    new Point2D(-2,  +6),
    new Point2D(-1,  +6),
    new Point2D(+1,  +6),
    new Point2D(+2,  +6)],
   [],                          // 1/2 sqaure
   [new Point2D( 0,  -1),       // front
    new Point2D(-3,  -1),
    new Point2D(-2,  -1),
    new Point2D(-1,  -1),
    new Point2D(+1,  -1),
    new Point2D(+2,  -1),
    new Point2D(+3,  -1)],
   [new Point2D( 0,  +7),       // back
    new Point2D(-3,  +7),
    new Point2D(-2,  +7),
    new Point2D(-1,  +7),
    new Point2D(+1,  +7),
    new Point2D(+2,  +7),
    new Point2D(+3,  +7)],
   [new Point2D(-4,   0),       // left
    new Point2D(-4,  +1),
    new Point2D(-4,  +2),
    new Point2D(-4,  +3),
    new Point2D(-4,  +4),
    new Point2D(-4,  +5),
    new Point2D(-4,  +6)],
   [new Point2D(+4,   0),       // right
    new Point2D(+4,  +1),
    new Point2D(+4,  +2),
    new Point2D(+4,  +3),
    new Point2D(+4,  +4),
    new Point2D(+4,  +5),
    new Point2D(+4,  +6)],
    new Point2D(-4,  -1),       // top left outside
    new Point2D(+4,  -1),       // top right outside
    new Point2D(-7,  +6),       // shift left pt
    new Point2D(+7,  +6),       // shift right pt
    new Point2D(-3,  +3),       // flank left contact pt
    new Point2D(+3,  +3),       // flank right contact pt
    new Point2D( 0,  -2));      // modifier


// pt coverage for a diagonal (315 degrees, facing "North
// East") 7 deep element
var d_footPrint7:Footprint = new Footprint(
   [                            // full sqaure
    new Point2D( 0,   0),       // front key (registration)
```

```
 new Point2D(-4,  +4),    // back key
 new Point2D(-2,  -2),    // left key
 new Point2D(+2,  +2),    // right key
 new Point2D(-6,  +2),    // back left corner
 new Point2D(-2,  +6),    // back right corner
 new Point2D(-4,   0),    // middle left
 new Point2D( 0,  +4),    // middle right
 new Point2D(-1,  -1),
 new Point2D(+1,  +1),
 new Point2D(-2,  -1),
 new Point2D(-1,   0),
 new Point2D( 0,  +1),
 new Point2D(+1,  +2),
 new Point2D(-3,  -1),
 new Point2D(-2,   0),
 new Point2D(-1,  +1),
 new Point2D( 0,  +2),
 new Point2D(+1,  +3),
 new Point2D(-3,   0),
 new Point2D(-2,  +1),
 new Point2D(-1,  +2),
 new Point2D( 0,  +3),
 new Point2D(-3,  +1),
 new Point2D(-2,  +2),
 new Point2D(-1,  +3),
 new Point2D(-4,  +1),
 new Point2D(-3,  +2),
 new Point2D(-2,  +3),
 new Point2D(-1,  +4),
 new Point2D(-5,  +1),
 new Point2D(-4,  +2),
 new Point2D(-3,  +3),
 new Point2D(-2,  +4),
 new Point2D(-1,  +5),
 new Point2D(-5,  +2),
 new Point2D(-4,  +3),
 new Point2D(-3,  +4),
 new Point2D(-2,  +5),
 new Point2D(-5,  +3),
 new Point2D(-3,  +5)],
[new Point2D(-2,  -3),    // top left corner // 1/2 sqaures
 new Point2D(+3,  +2),    // top right corner
 new Point2D(-7,  +2),    // bottom left corner
 new Point2D(-2,  +7),    // bottom right corner
 new Point2D(-1,  -2),
 new Point2D( 0,  -1),
 new Point2D(+1,   0),
 new Point2D(+2,  +1),
 new Point2D(+2,  +3),
 new Point2D(+1,  +4),
 new Point2D(+0,  +5),
 new Point2D(-1,  +6),
 new Point2D(-3,  +6),
 new Point2D(-4,  +5),
 new Point2D(-5,  +4),
 new Point2D(-6,  +3),
 new Point2D(-6,  +1),
```

```
            new Point2D(-5,   0),
            new Point2D(-4,  -1),
            new Point2D(-3,  -2)],
           [new Point2D(+1,  -1),    // front
            new Point2D(-1,  -3),
            new Point2D( 0,  -2),
            new Point2D(+2,   0),
            new Point2D(+3,  +1)],
           [new Point2D(-5,  +5),    // back
            new Point2D(-7,  +3),
            new Point2D(-6,  +4),
            new Point2D(-4,  +6),
            new Point2D(-3,  +7)],
           [new Point2D(-3,  -3),    // left
            new Point2D(-4,  -2),
            new Point2D(-5,  -1),
            new Point2D(-6,   0),
            new Point2D(-7,  +1)],
           [new Point2D(+3,  +3),    // right
            new Point2D(+2,  +4),
            new Point2D(+1,  +5),
            new Point2D( 0,  +6),
            new Point2D(-1,  +7)],
            new Point2D(-2,  -4),    // top left outside
            new Point2D(+4,  +2),    // top right outside
            new Point2D(-9,  -1),    // shift left pt
            new Point2D(+1,  +9),    // shift right pt
            new Point2D(-4,  0),     // flank left contact pt
            new Point2D( 0,  +4),    // flank right contact pt
            new Point2D(+2,  -2));   // modifier

    _footPrint7 = createBaseFootPrint(h_footPrint7, d_footPrint7);

}

//
// createBaseFootPrint()
//
//  From a horizontal and diagonal footprint create a base
//  footprint list of 8 Footprints [0, 45, 90, 135, 180, 225,
//  270, 215].
//
//   args:
//     h  -- base horizontal footprint ("North" facing 270
//           degrees)
//     d  -- base diagonal footprint ("North East" facing 315
//           degrees)
//
// return -- Array of eight footprints derived from the original
//           two
//
private static function createBaseFootPrint(h:Footprint,
         d:Footprint):Array {

    var r:Array = new Array();

    for (var angle:Number = 0; angle < 405; angle += 45) {
```

```
// footprint to use with this angle
var shape:Footprint;

// "theta" angle to rotate footprint points
var theta:Number;


if (angle%90 == 0) {        // horizontal or vertical case
    shape = h;
    theta = angle + 90;

} else {                    // diagonal case
    shape = d;
    theta = angle + 45;
}

// ensure theta within bounds
if (theta >= 360) { theta -= 360; } else if (theta <  0) {
    theta += 360;
}

// apply rotation
var wholeSquares:Array = shape.wholeSquares;
for (var i:Number = 0; i < wholeSquares.length; ++i) {
    wholeSquares[i].rotate(theta);
    wholeSquares[i].round();
}
var halfSquares:Array = shape.halfSquares;
for (var i:Number = 0; i < halfSquares.length; ++i) {
    halfSquares[i].rotate(theta);
    halfSquares[i].round();
}
var front:Array = shape.front;
for (var i:Number = 0; i < front.length; ++i) {
    front[i].rotate(theta);
    front[i].round();
}
var back:Array = shape.back;
for (var i:Number = 0; i < back.length; ++i) {
    back[i].rotate(theta);
    back[i].round();
}
var left:Array = shape.left;
for (var i:Number = 0; i < left.length; ++i) {
    left[i].rotate(theta);
    left[i].round();
}
var right:Array = shape.right;
for (var i:Number = 0; i < right.length; ++i) {
    right[i].rotate(theta);
    right[i].round();
}
var tpLeft:Point2D = shape.tpLeftOutside;
tpLeft.rotate(theta);
tpLeft.round();
var tpRht:Point2D = shape.tpRhtOutside;
```

```
        tpRht.rotate(theta);
        tpRht.round();
        var shiftLeftPt:Point2D = shape.shiftLeftPt;
        shiftLeftPt.rotate(theta);
        shiftLeftPt.round();
        var shiftRightPt:Point2D = shape.shiftRightPt;
        shiftRightPt.rotate(theta);
        shiftRightPt.round();
        var flankLeft:Point2D = shape.flankLeft;
        flankLeft.rotate(theta);
        flankLeft.round();
        var flankRht:Point2D = shape.flankRht;
        flankRht.rotate(theta);
        flankRht.round();
        var modifier:Point2D = shape.modifier;
        modifier.rotate(theta);
        modifier.round();

        r.push( new Footprint(wholeSquares,
                              halfSquares,
                              front,
                              back,
                              left,
                              right,
                              tpLeft,
                              tpRht,
                              shiftLeftPt,
                              shiftRightPt,
                              flankLeft,
                              flankRht,
                              modifier) );

    }

    return r;
}

//
//  Constructor
//
//   args:
//     color  -- Color of player being create, "blk" or "wht"
//               The black ("blk") player is created first, moves
// first
//               and is usually on the bottom portion of the map.
//
public function Player(color:String) {

    _color    = color;
    _left     = new Array();
    _center   = new Array();
    _right    = new Array();
    _all      = new Array();
    _dead     = new Array();
    _mapPos   = new Point2D(480, 270);      // GET SCRN SIZE
    _mapScale = 70;
```

```
    if (_color == "blk") {
        _mapAngle = 0;
    } else {
        _mapAngle = 180;
    }

    // initial scrol position is top right
    _scrollPos = new Point2D(794.0, 159.0);
}

//
// add()
//
//  Add a new Element to this player's army.
//
//   args:
//      command -- name of command being added to, "Left", "Right",
//                 or "Center"
//         type -- type of Element being added,"El","Kn","Cv", etc
//        grade -- grade of Element, "Ordinary", "Inferior",
//                 "Fast", "Superior"
//          reg -- true if Element counts as Regular (not Clumsy)
//         name -- name of the Element being added
//          pic -- picture to be use for this Element
//          loc -- starting location on map
//        angle -- starting angle on map
//    isGeneral -- type of general = "Sub-gen", "C-in-C", or
//                 undefined
//       player -- undefined if loading a fictional scenario and
//                 second players Elements need to be rotated to
//                 top portion of the map
//
public function add(command:String,
                    type:String,
                    grade:String,
                    reg:Boolean,
                    name:String,
                    pic:String,
                    loc:Point2D,
                    angle:Number,
                    isGeneral:String,
                    player:String):Void {

    // move everybody forward so they fight sooner
    // loc.subtract(_mvCloserPt);

    // if player is white then switch location to opposite side of
    // map
    if ( _color == "wht" && command != "TEMP" &&
            player == undefined) {

        // twist 180.
        var modifer:Point2D = new Point2D(_grid.width/2,
                  _grid.height/2);
        // transform point so "0,0" is center of map
        loc.subtract(modifer);
```

```
    loc.rotate(180);                                // spin 180
    // transform point so "0,0" is back in corner of map
    loc.add(modifer);

    loc.round();
    // convert the angle
    angle = Utils.cleanAngle(angle - 180);

}

// generate XML
/*
trace("<element>");
trace("    <player>Two</player>");
trace("    <command>" + command   + "</command>");
trace("    <type>"    + type      + "</type>");
trace("    <grade>"   + grade     + "</grade>");
trace("    <regular>" + reg       + "</regular>");
trace("    <name>"    + name      + "</name>");
trace("    <icon>"    + pic       + "</icon>");
trace("    <locX>"    + loc.x     + "</locX>");
trace("    <locY>"    + loc.y     + "</locY>");
trace("    <angle>"   + angle     + "</angle>");
if (isGeneral != undefined) {
    trace("    <general>" + isGeneral + "</general>");
}
trace("</element>");
*/

// influence value of the element
var influence:Number = 1;
if (isGeneral != undefined) {
    influence = 4;
} else if (type == "Exp" ||
    (type == "Hd" && grade == "Inferior")) {
    influence = 0;
} else if (type == "Bg" ||
           type == "El" ||
           type == "Kn" ||
           (type == "Cv" && grade == "Superior") ||
           (type == "Sp" && grade == "Superior") ||
           (type == "Sw" && grade == "Superior")) {
    influence = 2;
} else if ( type == "Sk" ||
            (type == "Hd" && grade != "Inferior") ||
            (type == "Wb" && grade != "Superior") ||
            (type == "Ax" && grade != "Superior") ||
            (type == "Sp" && grade == "Inferior" && ! reg) ||
            (type == "Pk" && grade == "Inferior" && ! reg) ||
            (type == "Bw" && grade == "Inferior" && ! reg)) {
    influence = 0.5;
}

// attach to command group
var cmd:Array;
switch (command) {
    case "Left":
```

```
            cmd = _left;
            _leftMoral = _startLeftMoral += influence;
            break;
        case "Right":
            cmd = _right;
            _rightMoral = _startRightMoral += influence;
            break;
        default:
            cmd = _center;
            _centerMoral = _startCenterMoral += influence;
            break;
    }

    var icon:String;
    var depth:Number;
    var move:Number;
    var footPrints:Array;
    var cmbTbl:CombatTable;


    // some troop types are considered impetuous, these are
    //  Expendables, Warriors and irregular elements of - Knights
    // (S), (O) or (F), Light Horse (S),
    //   Swords (S) or (F), Spears (O), Hordes (S) or (F)
    var impetuous:Boolean = false;
    if (type == "Wb" || type == "Exp") {
        impetuous = true;
    } else if (! reg) {

        if (type == "Kn" && (grade == "Superior" ||
                             grade == "Ordinary" ||
                             grade == "Fast")) {
            impetuous = true;
        }
        if (type == "LH" && grade == "Superior") {
            impetuous = true;
        }
        if (type == "Sw" && (grade == "Superior" ||
                             grade == "Fast")) {
            impetuous = true;
        }
        if (type == "Sp" && grade == "Ordinary") {
            impetuous = true;
        }
        if (type == "Hd" && (grade == "Superior" ||
                             grade == "Fast")) {
            impetuous = true;
        }
    }


    // Combat table results sysmbols
    //
    // K = Killed
    // K*= Killed if in close combat
    // k = Killed if in clear terrain
    // E = Killed if enemy's turn
```

```
        // e = Killed if enemy's turn and in clear terrain
        // S = Spent
        // s = Spent if in difficult terrain
        // F = Flee
        // F*= Flee if in close combat
        // f = Flee if in difficult terrain
        // R = Repulsed
        // R*= Repulsed if in close combat
        // r = Repulsed if in clear terrain
        // O = Repulsed if own turn
        // o = Repulsed if own turn and in clear terrain
        switch (type) {

            case "El":

                icon = "El";
                depth = 7;
                move  = 20;     // 200 paces

                cmbTbl = new CombatTable(_game, type, grade, false,
                                         5,   4,   "-", "-", "-", "-", "K", "-
", "-", "-", "-", "-", "K", "K", "K", "K", "-", "-", "-",     // <- if
less
                                                 "K", "K", "K", "K", "K",
"K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K");    // <-
if doubled
                //              vMtd vFt    El  Exp   Kn   Cv   LH   Sp
Pk   Sw   Bw   Cb   Ax  AxS   Sk  Art   Hd   Bg   Wb

                type = "Elephants";
                break;

            case "Exp":

                icon = "Exp";
                depth = 7;
                move  = 20;     // 200 paces

                cmbTbl = new CombatTable(_game, type, grade, false,
                                          5,   4,   "K", "K", "K", "K", "K",
"K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K",     // <-
if less
                                                 "K", "K", "K", "K", "K",
"K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K");    // <-
if doubled
                //              vMtd vFt    El  Exp   Kn   Cv   LH   Sp
Pk   Sw   Bw   Cb   Ax  AxS   Sk  Art   Hd   Bg   Wb

                type = "Expendables";
                break;

            case "Kn":

                icon = "Kn";
                depth = 4;
                move  = 20;     // 200 paces
```

```
                cmbTbl = new CombatTable(_game, type, grade, false,
                                         3,   4,   "K", "K", "-", "-", "-",
"R", "R", "-", "F*","F*","R", "R", "R", "-", "R", "-", "R",     // <-
if less
                                                   "K", "K", "K", "K", "K",
"K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K");    // <-
if doubled
                //              vMtd vFt    El  Exp   Kn   Cv   LH   Sp
Pk   Sw   Bw   Cb   Ax  AxS   Sk  Art   Hd   Bg   Wb

                type = "Knights";
                break;

            case "Cv":

                icon = "Cv";
                depth = 4;
                move  = 24;     // 240 paces

                // CAN CHOOSE REPULSED INSTEAD OF RECOIL
                // FLEE, NOT RECOIL, IF IN DIFFICULT
                cmbTbl = new CombatTable(_game, type, grade, false,
                                         3,   4,   "-", "F", "e", "-", "-", "-
", "-", "-", "-", "-", "-", "-", "-", "-", "-", "-", "-", "-",     // <- if
less
                                                   "K", "K", "K", "K", "K",
"O", "O", "O", "K", "K", "K", "K", "K", "R", "K", "R", "K");    // <-
if doubled
                //              vMtd vFt    El  Exp   Kn   Cv   LH   Sp
Pk   Sw   Bw   Cb   Ax  AxS   Sk  Art   Hd   Bg   Wb

                type = "Cavalry";
                break;

            case "LH":

                icon = "LH";
                depth = 4;
                move  = 32;     // 320 paces [was 280]

                // FLEE, NOT RECOIL, IF IN DIFFICULT
                cmbTbl = new CombatTable(_game, type, grade, false,
                                         2,   3,   "-", "F", "-", "-", "-",
"r", "r", "r", "R*","R*","r", "r", "r", "r", "r", "r", "r",     // <-
if less
                                                   "K", "K", "K", "K", "K",
"s", "s", "s", "K", "K", "s", "s", "s", "S", "s", "S", "S");    // <-
if doubled
                //              vMtd vFt    El  Exp   Kn   Cv   LH   Sp
Pk   Sw   Bw   Cb   Ax  AxS   Sk  Art   Hd   Bg   Wb

                type = "Lt. Horse";
                break;

            case "Sp":

                icon = "Sp";
```

```
                depth = 3;
                move  = 16;     // 160 paces

                cmbTbl = new CombatTable(_game, type, grade, false,
                                         4,   4,   "K", "e", "e", "-", "-", "-
", "-", "E", "-", "-", "-", "E", "-", "-", "-", "-", "E",     // <- if
less
                                               "K", "K", "K", "K", "K",
"K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K");    // <-
if doubled
                //              vMtd vFt    El  Exp   Kn   Cv   LH   Sp
Pk   Sw   Bw   Cb   Ax  AxS   Sk  Art   Hd   Bg   Wb

                type = "Spears";
                break;

            case "Pk":

                icon = "Pk";
                depth = 3;
                move  = 16;     // 160 paces

                cmbTbl = new CombatTable(_game, type, grade, false,
                                         4,   3,   "K", "e", "e", "-", "-", "-
", "-", "E", "-", "-", "-", "E", "-", "-", "-", "-", "E",     // <- if
less
                                               "K", "K", "K", "K", "K",
"K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K");    // <-
if doubled
                //              vMtd vFt    El  Exp   Kn   Cv   LH   Sp
Pk   Sw   Bw   Cb   Ax  AxS   Sk  Art   Hd   Bg   Wb

                type = "Pikes";
                break;

            case "Sw":

                icon = "Sw";
                depth = 3;
                move  = 16;     // 160 paces

                cmbTbl = new CombatTable(_game, type, grade, false,
                                         4,   4,   "K", "e", "e", "-", "-", "-
", "-", "-", "-", "-", "-", "-", "-", "-", "-", "-", "E",     // <- if
less
                                               "K", "K", "K", "K", "K",
"K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K");    // <-
if doubled
                //              vMtd vFt    El  Exp   Kn   Cv   LH   Sp
Pk   Sw   Bw   Cb   Ax  AxS   Sk  Art   Hd   Bg   Wb

                type = "Swords";
                break;

            case "Wb":

                icon = "Wa";
```

```
                depth = 3;
                move  = 16;     // 160 paces

                cmbTbl = new CombatTable(_game, type, grade, false,
                                         3,   3,   "K", "e", "e", "-", "-", "-
", "-", "E", "-", "-", "-", "-", "-", "-", "-", "-", "-",     // <- if
less
                                                "K", "K", "K", "K", "K",
"K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K");    // <-
if doubled
                //              vMtd vFt    El  Exp   Kn   Cv   LH   Sp
Pk   Sw   Bw   Cb   Ax  AxS   Sk  Art   Hd   Bg   Wb

                type = "Warriors";
                break;

            case "Bw":

                icon = "Ar";
                depth = 3;
                move  = 16;     // 160 paces

                cmbTbl = new CombatTable(_game, type, grade, false,
                                         4,   2,   "K", "-", "K", "K", "K", "-
", "-", "-", "-", "-", "-", "-", "-", "-", "-", "-", "K",     // <- if
less
                                                "K", "K", "K", "K", "K",
"K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K");    // <-
if doubled
                //              vMtd vFt    El  Exp   Kn   Cv   LH   Sp
Pk   Sw   Bw   Cb   Ax  AxS   Sk  Art   Hd   Bg   Wb

                type = "Archers";
                break;

            case "Cb":

                icon = "Cb";
                depth = 3;
                move  = 16;     // 160 paces

                cmbTbl = new CombatTable(_game, type, grade, false,
                                         4,   2,   "K", "-", "K", "K", "K", "-
", "-", "-", "-", "-", "-", "-", "-", "-", "-", "-", "K",     // <- if
less
                                                "K", "K", "K", "K", "K",
"K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K");    // <-
if doubled
                //              vMtd vFt    El  Exp   Kn   Cv   LH   Sp
Pk   Sw   Bw   Cb   Ax  AxS   Sk  Art   Hd   Bg   Wb

                type = "Crossbows";
                break;


            case "Ax":
```

```
                icon = "LI";
                depth = 3;
                move  = 20;     // 200 paces

                cmbTbl = new CombatTable(_game, type, grade, false,
                                         3,   3,   "k", "-", "k", "-", "-", "-
", "-", "-", "-", "-", "-", "-", "-", "-", "-", "-", "-", "-",     // <- if
less
                                                "K", "K", "K", "K", "K",
"K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K");    // <-
if doubled
                //              vMtd vFt    El  Exp   Kn   Cv   LH   Sp
Pk   Sw   Bw   Cb   Ax  AxS   Sk  Art   Hd   Bg   Wb

                type = "Lt. Infantry";
                break;

            case "Sk":

                icon = "Sk";
                depth = 3;
                move  = 20;     // 200 paces

                cmbTbl = new CombatTable(_game, type, grade, false,
                                          2,   2,   "-", "-", "k", "k", "k",
"r", "r", "r", "R*","R*","r", "r", "-", "-", "r", "-", "r",     // <-
if less
                                                "f", "f", "f", "f", "f",
"S", "S", "S", "K", "K", "K", "K", "K", "F", "S", "F", "S");    // <-
if doubled
                //              vMtd vFt    El  Exp   Kn   Cv   LH   Sp
Pk   Sw   Bw   Cb   Ax  AxS   Sk  Art   Hd   Bg   Wb

                type = "Skirmishers";
                break;

            case "Sk*":

                icon = "Sk";
                depth = 3;
                move  = 20;     // 200 paces

                cmbTbl = new CombatTable(_game, "Sk", grade, true,
                                          2,   2,   "-", "-", "k", "k", "k",
"r", "r", "r", "r", "r", "r", "r", "-", "-", "r", "-", "r",     // <-
if less
                                                "f", "f", "f", "f", "f",
"S", "S", "S", "K", "K", "K", "K", "K", "F", "S", "F", "S");    // <-
if doubled
                //              vMtd vFt    El  Exp   Kn   Cv   LH   Sp
Pk   Sw   Bw   Cb   Ax  AxS   Sk  Art   Hd   Bg   Wb

                type = "Skirmishers";
                break;

            case "Art":
```

```
                icon = "Art";
                depth = 7;
                move  = 16;     // 160 paces [was 120]

                if (grade == "Superior") { move = 8; }

                cmbTbl = new CombatTable(_game, type, grade, false,
                                         2,   2,   "K", "K", "K", "K", "K",
"K", "K", "K", "K*","K*","K", "K", "K", "K*","K", "K", "K",     // <-
if less
                                               "K", "K", "K", "K", "K",
"K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K");    // <-
if doubled
                //              vMtd vFt    El  Exp   Kn   Cv   LH   Sp
Pk   Sw   Bw   Cb   Ax  AxS   Sk  Art   Hd   Bg   Wb

                type = "Artillery";
                break;

            case "Hd":

                icon = "Hd";
                depth = 4;
                move  = 16;     // 160 paces

                cmbTbl = new CombatTable(_game, type, grade, false,
                                         2,   2,   "K", "k", "k", "-", "-", "-
", "-", "-", "-", "-", "-", "-", "-", "-", "-", "-", "K",     // <- if
less
                                               "K", "K", "K", "K", "K",
"K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K");    // <-
if doubled
                //              vMtd vFt    El  Exp   Kn   Cv   LH   Sp
Pk   Sw   Bw   Cb   Ax  AxS   Sk  Art   Hd   Bg   Wb

                type = "Hordes";
                break;

            case "Bg":

                icon = "Bg";
                depth = 7;
                move  = 0;

                cmbTbl = new CombatTable(_game, type, grade, false,
                                         2,   2,   "K", "K", "K", "K", "K",
"K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K",     // <-
if less
                                               "K", "K", "K", "K", "K",
"K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K", "K");    // <-
if doubled
                //              vMtd vFt    El  Exp   Kn   Cv   LH   Sp
Pk   Sw   Bw   Cb   Ax  AxS   Sk  Art   Hd   Bg   Wb

                type = "Baggage";
                break;
        }
```

```
    // assign appropriate footprints according to depth
    footPrints = Player["_footPrint" + depth];

    // deal with generals (pre creating element)
    if (isGeneral != undefined) { icon = "_" + isGeneral +
        "_" + depth; }

    var e:Element = new Element( this,
                          command,
                          grade,
                          reg,
                          type,
                          influence,
                          name,
                          icon,
                          depth,
                          impetuous,
                          pic,
                          move,
                          cmbTbl,
                          _color,
                          loc,
                          angle,
                          footPrints,
                          isGeneral);

    // assign element to its combat table
    cmbTbl.element = e;

    cmd.push(e);
    _all.push(e);
    _everybody.push(e);

    // deal with generals (post creating element)
    if (isGeneral != undefined) { setAsGeneral(e, isGeneral,
             command); }
}

//
// rollPIPs()
//
//  Roll player initiative dice for this player
//
public function rollPIPs() {

    if (_centerGen.regular) {
        var average:Number = (Utils.randomInt(1, 6) +
                              Utils.randomInt(1, 6) +
                              Utils.randomInt(1, 6))/3;
        average = Math.floor(average);

        _PIPsLeft    = average;
        _PIPsCenter  = average + 1;
        _PIPsRight   = average;

    } else {
```

```
        _PIPsLeft    = Utils.randomInt(1, 6);
        _PIPsCenter  = Utils.randomInt(1, 6) + 1;
        _PIPsRight   = Utils.randomInt(1, 6);
    }
}

//
// setAsGeneral()
//
//  Aassign an element as a general of a command
//
//   args:
//      e       -- reference to Element
// isGeneral    -- type of general "Sub-gen" or "C-in-C"
//  command     -- name of command "Left", "Right", "Center"
//
private function setAsGeneral(e:Element, isGeneral:String,
         command:String) {

    if (command == "Left") {
        _leftGen = e;
    } else if (command == "Center") {
        _centerGen = e;
    } else if (command == "Right") {
        _rightGen = e;
    }
}

//
// elementDead()
//
//  Remove an element from command lists and add to dead pile
//
//   args:
//      e   -- Element being removed
//
public function elementDead(e:Element):Void {

    // remove e from "all" list
    for (var i:Number = 0; i < _all.length; ++i) {
        if (_all[i].spriteN == e.spriteN) {
            _all.splice(i, 1); break;
        }
    }

    // remove e from "everybody" list
    for (var i:Number = 0; i < _all.length; ++i) {
        if (_everybody[i].spriteN == e.spriteN) {
            _everybody.splice(i, 1); break;
        }
    }

    // remove e from its command group
    var cmd:Array;
    switch (e.command) {
        case "Left":
            cmd = _left;
```

```
            _leftMoral -= e.influence;
            break;
        case "Center":
            cmd = _center;
            _centerMoral -= e.influence;
            break;
        case "Right":
            cmd = _right;
            _rightMoral -= e.influence;
            break;
    }
    for (var i:Number = 0; i < cmd.length; ++i) {
        if (cmd[i].spriteN == e.spriteN) {
            cmd.splice(i, 1);
            break;
        }
    }

    // add e to pile of dead
    _dead.push(e);

}

//
// getCmdStatus()
//
//  Return a string describing the morale status of a command.
//  Sometimes it's only important to know if the command is
//  shattered, broken, or dispirited, [flag = false] such as
//  during battle.
//
//   args:
//      cmd     -- name of command "Left", "Right", "Center"
//      flag    -- true id full description needed
//
//   return     -- String describing commands morale
//
public function getCmdStatus(cmd:String, flag:Boolean):String {

    var percent:Number = getMoraleValue(cmd);

    if (percent < 0.5)   { return "Shattered"; }
    if (percent < 0.666) { return "Broken"; }
    if (percent < 0.75)  { return "Dispirited"; }

    if (flag) {
        if (percent < 0.80) { return "Very Low"; }
        if (percent < 0.85) { return "Low"; }
        if (percent < 0.90) { return "Fair"; }
        if (percent < 0.95) { return "Good"; }
        return "High";
    } else {
        return "Normal";
    }
}

//
```

```
// getMoraleValue()
//
//  Get the morale of one of this players commands
//
//   args:
//      cmd     -- name of command "Left", "Right", "Center"
//
//   return     -- value between 0 and 1
//
private function getMoraleValue(cmd:String):Number {
    switch (cmd) {
        case "Left":   return _leftMoral/_startLeftMoral;
        case "Center": return _centerMoral/_startCenterMoral;
        case "Right":  return _rightMoral/_startRightMoral;
    }
}

//
// getMoralePercent()
//
//  Get the morale % of one of this players commands
//
//   args:
//      cmd     -- name of command "Left", "Right", "Center"
//
//   return     -- value (%) between 0 and 100
//
public function getMoralePercent(cmd:String):Number {
    return Math.round(getMoraleValue(cmd)*100);
}

//
// accessors
public static function get active():Player {
    return _active;
}
public static function getEverybody():Array {
    return _everybody;
}
public function get thePlayer():String {
    return _player;
}
public function get left():Array {
    return _left;
}
public function get center():Array {
    return _center;
}
public function get right():Array {
    return _right;
}
public function get all():Array {
    return _all;
}
public function get centerGen():Element {
    return _centerGen;
}
```

```
    public function get mapPos():Point2D {
        return _mapPos;
    }
    public function get mapScale():Number {
        return _mapScale;
    }
    public function get mapAngle():Number {
        return _mapAngle;
    }
    public function get scrollPos():Point2D {
        return _scrollPos;
    }
    public function get PIPsLeft():Number {
        return _PIPsLeft;
    }
    public function get PIPsCenter():Number {
        return _PIPsCenter;
    }
    public function get PIPsRight():Number {
        return _PIPsRight;
    }

    //
    // mutators
    public static function set active(val:Player):Void {
        _active = val;
    }
    public function set thePlayer(val:String):Void {
        _player = val;
    }
    public function set mapPos(val:Point2D):Void {
        _mapPos = val;
    }
    public function set mapScale(val:Number):Void {
        _mapScale = val;
    }
    public function set mapAngle(val:Number):Void {
        _mapAngle = val;
    }
    public function set scrollPos(val:Point2D):Void {
        _scrollPos = val;
    }
    public function set PIPsLeft(val:Number):Number {
        _PIPsLeft = val;
    }
    public function set PIPsCenter(val:Number):Number {
        _PIPsCenter = val;
    }
    public function set PIPsRight(val:Number):Number {
        _PIPsRight = val;
    }
}
```

Element.as

```
////////////////////////////////////////////////////////////////////////
//
//  Element.as
//
//       AUTHOR: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: Element instances are always created by the Player
//               Object.  The Player instance passes player
//               information and initialization parameters
//               originating from XML to the new Element()
//               constructor, so that the Element can be created
//               under the command of that player.
//
//               The Element Object specifies a great many parameters
//               giving the element its individuality, but it also
//               contains the many methods needed to be self aware on
//               the map grid. An Element is able to detect for other
//               Elements around it, and also how to respond under
//               certain circumstances, such as fleeing, being
//               killed, pursuing, recoiling, finding the front rank
//               of a group, finding the rear rank of a group,
//               turning to face an enemy, and, perhaps most
//               importantly, detecting when moving into combat with
//               an enemy Element.
//
//               This Object also has the all important testLocation()
//               calls used by the Scroll Object that determine if
//               the Element is capable of moving to a certain
//               location at a certain orientation. There are also
//               calls for dealing with the movement shadow that
//               appears under the Element when the Element is about
//               to move to a location. Every Element Object
//               maintains a corresponding Animatem reference to a
//               Sprite Object (_sprite) used to display where it is
//               on the map.
//
//       Method:
//
//         Element() - Constructor
//      initialize() - Initialize class with static variables
//     stateNormal() - Set state of element normal. Called by reset()
//                           unless Element engaged.
//     stateRollHL() - Set state of element to Roll (white glow if
//                     friendly, black glow is enemy)
//   stateSelectHL() - Set state of element to Highlighted(yellow glow)
//   statusEngaged() - Set status of element to Shoots (burgundy glow).
//                     This element is now in combat
//    statusShoots() - Set status of element to Shoots (green glow).
//                     This element has become the target of bowmen,
//                     or is a bowmen shooting. This is a temporary
//                     status only occuring during the Shoots phase,
```

```
//                      it cannot occur if the element is Engaged.
//    statusMoving() - Set status of element to Moving (white glow)
//    statusNormal() - Set status of element to Normal (no glow)
//           reset() - Called at the beginning of a players turn.
//                     Reset Element, setting it movement point bac to
//                     full, clear it's last move made, clear nudges
//                     made, clear flag indicating that Element
//                     withdrew from battle, and, if not engaged in
//                     battle, set  it's  status and state to normal.
//       disengage() - Disengage Element from battle state
//           small() - Use the small/simplified version of icon for the
//                     Element that is more readable when the map
//                     is small
//           large() - Use the large/detailed version of icon for the
//                     Element that is clearer when the map is large
//           alpha() - Make the Element semi transparent to denote it
//                     is in another command
//      setFilters() - Set the filter effects (glows) for this Element
//                     according to state and status
//   removeFilters() - Remove all filter effects for this Element.
//                     This is done before animating the map, so to
//                     allow very rapid animation
//    getFootprint() - return the corresponding footprint to use when
//                     placing this elements data on the grid at
//                     an angle
//        shadowAt() - Draw angled placement shadaw at a location
//     resetShadow() - Place shadow at location element grid data is at
//    testLocation() - Test if this Element is able to move to a new
//                     location and angle on the grid. It is crucial
//                     that this call be  made on a potential move
//                     before actually performing  such a move with
//                     moveMeTo()  or setLoc().  The Scroll object
//                     performs most  calls to testLocation().
//     squareClear() - Called by testLocation(). Checks if a grid
//                     square can be moved into by this Element.
//                     Diagonal Elements have "half points" which can
//                     contain "half an edge square" which can make
//                     this operation messy. Similarly, corner points
//                     in some instances are considered clear.
//          remove() - Remove element from the game. This is called
//                     (not surpisingly) when an element is "Killed"
//                     or "Spent".
//        moveMeTo() - Move this element to a new grid location.  To do
//                     this cleanly the elements footprint must be
//                     removed from the grid using removeData() and
//                     reinstated at a new location using setLoc()
//      removeData() - Remove this Elements footprint of information
//                     from the map/grid.  setLoc should be called soon
//                     after, this call to reestablish the Element on
//                     the grid, unless of course this element is
//                     killed/being removed.
//          setLoc() - Set the location and angle of this element to a
//                     new location and angle. All tests to see if this
//                     placement is legal will have already been done
//                     by other functions in the Scroll object, so all
//                     of the work here is in changing the Elements
//                     data footprint and testing for battles being
```

```
//                      triggered by moving to the new location.
//   aboutToEngage() - returns true if this element has any engagements
//                      about to happen on its _movelist.  This test is
//                      needed by the Battles state to check if the
//                      player has ended his move but thereare still
//                      resultant battles/engagements thathave not yet
//                      been triggered.
//         advance() - Move this Element to the next location on the
//                      _movelist
//      speedLimit() - return the speed of any element in front if
//                      it's slower
//       globalLoc() - return the global location of this element, or
//                      a specific footprint corner/part
//      globalRect() - Make Rect of Element global according to the
//                      _sprite path
//      makeGlobal() - Make a Point2D global according to the
//                      _sprite path.
//   atDestination() - Triggered by Animatem when this elements sprite
//                      reaches a destination it was moving to.
//        adjacent() - Return a list of any Elements directly adjacent
//                      to this one. This is useful for establishing
//                      Elements in a group.
//  elementInFront() - return - Element aligned directly in front
//   elementBehind() - return - Element aligned directly to rear
//   elementToLeft() - return - Element aligned directly to left
//  elementToRight() - return - Element aligned directly to right
//strictTestForElement() - Test for Element at a specific footprint pt
//elementMostInFront() - rtn Element most directly in front edge
//elementMostBehind() - rtn Element most directly to rear edge
//elementMostToLeft() - rtn Element most directly to left edge
//elementMostToRight() - rtn Element most directly to right edge
// elementsInFront() - rtn list of Elements along front edge
//  elementsBehind() - rtn list of Elements along rear edge
//  elementsToLeft() - rtn list of Elements along left edge
// elementsToRight() - rtn list of Elements along right edge
//     leftFlanked() - rtn any enemy element attacking the left flank
//    rightFlanked() - rtn any enemy element attacking the right flank
//    rearAttacked() - rtn any element attacking the rear of
//                      this element
//     leftOverlap() - rtn element causing an overlap on left corner
//    rightOverlap() - rtn element causing an overlap on right corner
//        recoilPt() - rtn a point to recoil to, if available
//          recoil() - This Element (and any behind it) recoil a
//                      base depth
//        repulsed() - This Element (and any behind it) are repulsed,
//                      which is similar to flee but does not retreat
//                      not as far.
//           flee() - This Element (and any behind it) flee.
//          spent() - Hark! This Element is spent (similar to killed).
//         killed() - Hark! This Element is dead.
//         pursue() - Pursue a retreating enemy Element
//chckForUnxpectedContct() - Check if Element has moved/stumbled into
//                      an enemy flank overlap or  a diagonal contact,
//                      triggering an engagement.
//    engagingWith() - Return whatever enemy Element this Element
//                      is fighting
// turnToFaceEnemy() - Turn this Element to face an attacking enemy
```

```
//                      Element. Elements already engaged will not
//                      turn to face.
// firstTurnToFace() - Turn this Element to a facing, such as "Rear",
//                      "Left", "Right". The first Element in a column
//                      that needs to turn to face is a special case,
//                      subsequent Elements behind use the regular
//                      (recursive) turnToFace function. If the first
//                      Element is Engaged then the turnToFace is passed
//                      through to the element directly to its rear.
//      turnToFace() - Recursive function called by firstTurnToFace.
//                      Turn this Element to a facing, such as "Rear",
//                      "Left", "Right". Elements already engaged will
//                      not turn to face. Elements turning to face a
//                      flank will often have to push elements behind
//                      them back. All Elements in a column will turn to
//                      face the same direction.
//      isFriendly() - Check if an element isfriendly with this Element
//getFrontRankElement() - Recursive call to find front element of
//                                  column
//getRearRankElement() - Recursive call to find rear element of column
//
//        Notes:
//

class Element {

    // static members

    // chavalier game object
    private static var _game:Chevalier

    // generic animator used
    private static var _a:Animatem

    // "game board" grid object with location of game pieces
    private static var _grid:Grid;

    // instance members
    // player who owns this element
    private var _player:Player;

    // Command group this element is in, "left", "right", "center"
    private var _cmdGroup:String;

    // sprite used by animator for this object
    private var _sprite:Sprite;

    // location of this element on the game board grid
    private var _grdLoc:Point2D;

    // location of this element on the screen
    private var _srnLoc:Point2D;

    // name of military icon/symbol used for this element
    private var _icon:String;

    // name of image picture used for this element
```

```
private var _picture:String;

// name of color who controls this element ("rd" or "bl")
private var _color:String;

// placement angle of the element NOT NEEDED? TAKE FROM SPRITE?
private var _angle:Number;

// unit type, "Swords", "Spears", "Bow", "Auxilia", "Knights",
// "Cavalry", "Light Horse", etc
private var _type:String;

// grading of the element "Superior", "Ordinary", "Fast",
// "Inferior", or "Exception"
private var _grade:String;

// true if Regular, otherwise Irregular
private var _regular:Boolean;

private var _desc:String;               // element description
// "Normal", "Moving", "Engaged", "Flanked", "Surrounded", ...etc
private var _status:String;

// base (start) movement points for this elemet
private var _baseMve:Number;

// remaining movement points
private var _mvePts:Number;

// table with information needed to conduct combat for this element
private var _combatTbl:CombatTable;

// terrain this element is in
private var _terrain:String = "Clear";

// "Sub-gen" or "C-in-C" if a general, otherwise undefined
private var _isGeneral:String;

// some troops are "impetuous" and will charge into battle without
// orders
private var _impetuous:Boolean;

// the units moral effect (ME) / influence on moral
private var _influence:Number;

// stored move results for each possible move
private var _StoredMoveResults:Array;

// list of moves this element have been told to make.
private var _movelist:Array;

// true if element animating
private var _advancing:Boolean = false;

// name of the last move made, this is reset at the beginning of
// each new turn.
private var _lastMvMade:String;
```

```
// true if element withdrew from combat this turn
private var _withdrawn:Boolean;

// number of nudge moves element has made in a turn
private var _nudges:Number;

// the depth of the base of this element
private var _baseDepth:Number;

// clip used for the movement shadow
private var _shadowClip:MovieClip;

// alpha value of shadow when stationary
private var _dark:Number = 56;

// alpha value of shadow when moving
private var _light:Number = 28;

// alpha value of units when "in other command"
private var _otherCmdAlpha:Number = 50;

// current footprint of element on grid
private var _footPrint:Footprint;

// array of base footprints, one for each angle [0, 45, 90, 135,
// 180, ... etc]
private var _footPrints:Array;

// the element state, "Normal", "Highlight", "Roll"
private var _state:String = "Normal";

// the number of PIPs paid for movement this turn
private var _PIPsPaid:Number;

//
// initialize()
//
//   Initialize class with static variables
//
//   args:
//  chevalier  -- Master game object
//          a  -- animator being used
//       grid  -- grid object for the map
//
public static function initialize(chevalier:Chevalier, a:Animatem,
         grid:Grid) {
    _game = chevalier;
    _a    = a;
    _grid = grid;
}

//
//  Constructor
//
//   args:
//          player -- reference to player object who owns this
```

```
// Element
//        cmdGroup -- command this Element is is, "Left", "Right",
// "Center"
//              gd -- the grade of this Element, "Ordinary",
// "Superior", etc
//         regular -- true if considered Regular (not Clumsy)
//            type -- The type of this Element, "Knights",
// "Spears", etc
//       influence -- Effect on morale, 0, 1/2, 1, 2, or 4
//            desc -- description (name)
//           mIcon -- icon used by
//       baseDepth -- basic base depth, 2 (infantry), 3 (cavalry +
// Hd), or 7 (other)
//       impetuous -- true if considered impetuous troops
//         picture -- picture to be used for the Element
//        movement -- max movement
//       combatTbl -- combat table object
//           color -- designates player "blk" or "wht"
//             loc -- starting location for
//           angle -- starting angle of
//      footPrints -- Array of base footprints used
//       isGeneral -- undefined, "Sub-gen", or "C-in-C"
//
public function Element(player:Player,
                        cmdGroup:String,
                        gd:String,
                        regular:Boolean,
                        type:String,
                        influence:Number,
                        desc:String,
                        mIcon:String,
                        baseDepth:Number,
                        impetuous:Boolean,
                        picture:String,
                        movement:Number,
                        combatTbl:CombatTable,
                        color:String,
                        loc:Point2D,
                        angle:Number,
                        footPrints:Array,
                        isGeneral:String) {

    _player      = player;
    _cmdGroup    = cmdGroup;
    _type        = type;
    _grade       = gd;
    _regular     = regular;
    _desc        = desc;
    _color       = color;
    _icon        = _color + mIcon;
    _picture     = picture;
    _baseMve     = movement;
    _baseDepth   = baseDepth;
    _impetuous   = impetuous;
    _combatTbl   = combatTbl;
    _status      = "Normal";
    _isGeneral   = isGeneral;
```

```
    _influence   = influence;

    _movelist  = new Array();
    _StoredMoveResults = new Array();

    // fast troops may move an additional 4
    if (_grade == "Fast") { _baseMve += 4; }

    _mvePts     = _baseMve;
    _footPrints = footPrints;

    // get sprites from the animator
    _sprite    = _a.addSprite(_icon, -1000, -1000, 0);
    _sprite.tag = this;
    _a.addBevel(_sprite.number, 4, 3, 0.42, 3 );

    // sprites start depth at 300, so have shadows start at 50
    var shadowDepth:Number = _sprite.number + 50;
    _shadowClip = _sprite.path.attachMovie("shadow_" + _baseDepth,
             "shadow" + _sprite.number, shadowDepth);
    _shadowClip._x = -1000; _shadowClip._y = -1000;
    _shadowClip._alpha = _dark;

    setLoc(loc, angle, true);

    setFilters();
}

//
// change of Elelemnt states and status.
//

//
// stateNormal()
//
//  Set state of element normal.
//  Called by reset() unless Element engaged.
//
public function stateNormal():Void {
    _shadowClip._visible = true;
    _sprite.movieClip._alpha = 100;
    resetShadow();
    _state = "Normal"; setFilters();
}

//
// stateRollHL()
//
//  Set state of element to Roll (white glow if friendly, black
// glow is enemy)
//
public function stateRollHL():Void {
    _state = "Roll";
    setFilters();
}

//
```

```
// stateSelectHL()
//
//  Set state of element to Highlighted/Selected (yellow glow)
//
public function stateSelectHL():Void {
    _state = "Highlight";
    setFilters();
}

//
// statusEngaged()
//
//  Set status of element to Shoots (burgundy/red glow)
//  This element is now in combat
//
public function statusEngaged():Void    {
    _status = "Engaged";
    setFilters();
    _mvePts = 0;
    _game.updateScrollText();
}

//
// statusShoots()
//
//  Set status of element to Shoots (green glow)
//  This element has become the target of bowmen, or is a bowmen
//  shooting. This is a temporary status only occuring during the
//  Shoots phase/mode, it cannot occur if the element is Engaged.
//
public function statusShoots():Void {

    if (_status == "Engaged") {
     throw new Error("Can't shoot at an element that's in combat");
    }
    _status = "Shoots";
    setFilters();
}

//
// statusMoving()
//
//  Set status of element to Moving (white glow)
//
public function statusMoving():Void     {
    if (_status != "Engaged") {
        if (_mvePts > 0.5) {
            _status = "Moving";
        } else {
            _status = "Moved";
        }
        setFilters();
    }
}

//
// statusNormal()
```

```
//
//  Set status of element to Normal (no glow)
//
public function statusNormal():Void     {
    _status = "Normal";
    setFilters();
}

//
// reset()
//
//  Called at the beginning of a players turn.
//  Reset Element, setting it movement point back to full,
//  clear it's last move made, clear nudges made, clear
//  flag indicating that Element withdrew from battle, and,
//  if not engaged in battle, set it's status and state to normal
//
public function reset():Void            {
    _mvePts     = _baseMve;
    _lastMvMade = undefined;
    _withdrawn  = false;
    _PIPsPaid   = 0;
    _nudges     = 0;
    if (_status != "Engaged") {
        statusNormal();
        stateNormal();
    }
}

//
// disengage()
//
//  Disengage Element from battle state
//
public function disengage():Void {
    if (_status == "Engaged") { statusNormal(); stateNormal();
    } else {
    throw new Error("Can't disengage element that's not engaged.");
    }
}

//
// small()
//
//  Use the small/simplified version of icon for the Element
//  that is more readable when the map is small
//
public function small():Void            { _sprite.frame = 1; }

//
// large()
//
//  Use the large/detailed version of icon for the Element
//  that is clearer when the map is large
//
public function large():Void            { _sprite.frame = 2; }
```

```
//
// alpha()
//
//  Make the Element semi transparent to denote it is in another
// command
//
public function alpha():Void {
    // lighten the element and remove the shadow
    _sprite.movieClip._alpha = _otherCmdAlpha;
    _shadowClip._visible = false;
}

//
// setFilters()
//
//  Set the filter effects (glows) for this Element according
//  to state and status
//
private function setFilters():Void {

    var filters:Array = new Array();

    // if engaged then add a deep burgandy glow
    if (_status == "Engaged") {

        // add a red glow
        filters = _a.addGlow(_sprite.number, 0x660000, 4, 3.5, 3,
                filters);

    } else if (_status == "Shoots") {

        // add a green glow
        filters = _a.addGlow(_sprite.number, 0x006600, 4, 3.5, 3,
                filters);

    } else if (_status == "Moving" && _state != "Highlight") {

        // add a white glow
        filters = _a.addGlow(_sprite.number, 0xFFFFFF, 7, 1.4, 3,
                filters);
    }

    // if rolled then
    if (_state == "Roll") {

        if (this.player.thePlayer == Player.active.thePlayer) {

            // add a soft white glow (player owned)
            filters = _a.addGlow(_sprite.number, 0xFFFFFF, 5,1.2,3,
                    filters);
        } else {

            // add a soft black glow (enemy owned)
            filters = _a.addGlow(_sprite.number, 0x000000, 5,1.2,3,
                    filters);
        }
    } else if (_state == "Highlight") {
```

```
        // add a soft white and yellow glow
        if (_status != "Moved") {
            filters = _a.addGlow(_sprite.number, 0xFFFFFF, 5,1.2,3,
                     filters);
        }
        filters = _a.addGlow(_sprite.number, 0xFFFF88, 10, 1.4, 3,
                 filters);
    }

    _sprite.movieClip.filters = filters;

}

//
// removeFilters()
//
//  Remove all filter effects fpr this Element
//  This is done before animating the map, so to
//  allow very rapid animation
//
private function removeFilters():Void {
    var filters:Array = _sprite.movieClip.filters;
    while (filters.length > 0) { filters.pop(); }
    _sprite.movieClip.filters = filters;
}

//
// getFootprint()
//
//  return the corresponding footprint to use when placing
//  this elements data on the grid at a specific angle
//
//       angle  -- angle of footprint requested
//
private function getFootprint(angle:Number):Footprint {
    angle = Utils.cleanAngle(angle);
    return _footPrints[angle/45];
}

//
// shadowAt()
//
//  Draw angled placement shadaw at a location
//
//         loc  -- grid location shadow is to be seen at
//       angle  -- angle of shadow
//
public function shadowAt(loc:Point2D, angle:Number):Void {

    var f:Footprint = getFootprint(angle);

    var scnLoc:Point2D = _grid.gridToPtLoc(loc);
    scnLoc.add(f.modifier);
    _shadowClip._visible = true;
    _shadowClip._rotation = angle;
    _shadowClip._x = scnLoc.x;
```

```
    _shadowClip._y = scnLoc.y;

}

//
// resetShadow()
//
//  Place shadow at location element grid data is at
//
public function resetShadow():Void {
    shadowAt(_grdLoc, _angle);
}

//
// testLocation()
//
//  Test if this Element is able to move to a new location and
//  angle on the grid. It is crucial that this call be made on
//  a potential move before actually performing such a move with
//  moveMeTo() or setLoc().  The Scroll object performs
//  most calls to testLocation()
//
//   args:
//         loc  -- location being tested
//    diagonal  -- true if the movement being tested results in a
//                 diagonal element
//    halfTest  -- true if the test if a "half point"
// isACrnrTest  -- true if the test is for the corner of the
//                 Element move
//      return  -- true if the location and angle is considered
//                 clear
//
public function testLocation(loc:Point2D, angle:Number):Boolean {

    // squareClear will need to know if a diagonal
    var diagonal:Boolean = isDiagonal(angle);


    // get a temp copy of the footprint corresponding to location
    // and angle
    var footPrint:Footprint = getFootprint(angle);

    // check the whole points of footprint--must be entirely clear
    var wholePts:Array = footPrint.wholeSquares;
    //   ...only need check the first 6 (or 8)
    var length:Number = (_baseDepth == 7) ? 8 : 6;

    for (var i:Number = 0; i < length; ++i) {
        wholePts[i].add(loc);
        if (! squareClear(wholePts[i], diagonal, false,
                 false)) { return false; }
    }

    // check the 1/2 points of footprint--must be at least 1/2
    // clear
    //  ...non-diagonal elements will have no half points
    if (diagonal) {
```

```
        var halfPts:Array = footPrint.halfSquares;
        for (var i:Number = 0; i < halfPts.length; ++i) {

            // check reference on the grid
            halfPts[i].add(loc);
            if (! squareClear(halfPts[i], diagonal, true,
                     (i < 4))) { return false; }
        }
    }

    return true;        // location clear
}

//
// squareClear()
//
//  Called by testLocation(). Checks if a grid square can be moved
//  into by this Element.  Diagonal Elements have "half points"
//  which can contain "half an edge square" which can make this
//  operation messy. Similarly, corner points in some instances
//  are considered clear.
//
//   args:
//         loc  -- location being tested
//    diagonal  -- true if the movement being tested results in a
//                 diagonal element
//    halfTest  -- true if the test if a "half point"
// isACrnrTest  -- true if the test is for the corner of the
//                 Element move
//
//      return  -- true if the square is considered clear
//
private function squareClear(loc:Point2D, diagonal:Boolean,
         halfTest:Boolean, isACrnrTest:Boolean):Boolean {

    // target will be either (A) undefined, (B) an element object,
    //  or (C) an array of elemet objects, donating half point(s)
    var target = _grid.getAt(loc);
    var e:Element; var crnrPt:Boolean;

    if (target == undefined) {
        // (A) square is empty
        return true;


    // (B) square has an element
    } else if (target instanceof Element) {


        // if square is part of self
        if (target.spriteN == this.spriteN ||

            //  or part of the selected group
            target.state   == "Highlight"  ||

            //  or testing a 1/2 pt and target isn't diagonal
            halfTest && ! target.diagonal) {
```

```
            //  then square is considered clear
            return true;

        } else {
            //  otherwise square is blocked
            return false;

        }

    // (C) square has "part(s)" of a diagonal element (1/2 points)
    } else if (target instanceof Array) {


        // half points need not check for other half points
        if (halfTest) { return true; }


        // test all half points at location (can be up to 4, if a
        // corner)
        for (var i:Number = 0; i < target.length; ++i) {

            e = target[i].element; crnrPt = target[i].corner;
            // if this 1/2 pt is not part of self
            if (e.spriteN != this.spriteN &&

                //  and it isn't part of the selected group
                e.state   != "Highlight") {

                if (! diagonal && crnrPt) {
                    // non-diagonal pieces ignore corner 1/2
                    // points,
                    //  allowing corners to be clipped when moving
                // then the square is blocked
                } else { return false; }

            }
        }
        // otherwise the square is a part of self/selected group
        return true;

    }
}

//
// remove()
//
//  Remove element from the game. This is called (not surpisingly)
//  when an element is "Killed" or "Spent"
//
private function remove():Void {
    // tell player object element is dead and to tally as such
    _player.elementDead(this);

    // remove element "footprint" data from map
    removeData();
```

```
    // remove the shadow movieClip
    _shadowClip.removeMovieClip();

    // remove element sprite from animator
    _a.clearSprite(_sprite.number);

}

//
// moveMeTo()
//
//  Move this element to a new grid location.  To do this
//  cleanly the elements footprint must be removed from
//  the grid using removeData() and reinstated at a new location
//  using setLoc()
//
//   args:
//        loc   -- new location for this Element
//      angle   -- new angle for this element
//
public function moveMeTo(loc:Point2D, angle:Number):Void {
    removeData();
    setLoc(loc, angle);
}

//
// removeData()
//
//  Remove this Elements footprint of information from
//  the map/grid.  setLoc should be called soon after,
//  this call to reestablish the Element on the grid, unless
//  of course this element is killed/being removed.
//
private function removeData():Void {

    // remove element's whole point references from grid
    var wholePts:Array = _footPrint.wholeSquares;
    for (var i:Number = 0; i < wholePts.length; ++i) {
        _grid.setAt(wholePts[i], undefined);
    }

    // remove element's 1/2 point references from grid
    var e:Element;
    var halfPts:Array = _footPrint.halfSquares;
    for (var i:Number = 0; i < halfPts.length; ++i) {
        var val = _grid.getAt(halfPts[i]);
        if (val instanceof Array) {
            for (var j:Number = 0; j < val.length; ++j) {
                e = val[j].element;
                if (e.spriteN == this.spriteN) {
                    val.splice(j, 1);
                    break;
                }
            }
            if (val.length == 0) {
                _grid.setAt(halfPts[i], undefined);
```

```
            } else {
                _grid.setAt(halfPts[i], val);
            }
        }
    }
}

//
// setLoc()
//
//  Set the location and angle of this element to a new location
//  and angle. All tests to see if this placement is legal will
//  have already been done by other functions in the Scroll object,
//  so all of the work here is in changing the Elements data
//  footprint and testing for battles being triggered by moving to
//  the new location.
//
//   args:
//        loc   -- new location for this Element
//      angle   -- new angle for this element
//  firstDraw   -- true if this Element is being placed on the map
//                 for the first time.
//
public function setLoc(loc:Point2D, angle:Number,
         firstDraw:Boolean):Void {

    // ensure loc is safe and without decimal places .000
    loc.round();

    if (firstDraw == undefined) { firstDraw = false; }

    // ensure angle within bounds
    angle = Utils.cleanAngle(angle);
    var turning:Boolean = (angle != _angle);

    if ( ! testLocation(loc, angle)) {
        // throw new Error(_type + " " + _grade + " can't move
        // like that.");
        trace(_type + " " + _grade + " can't move like that.");
        return;
    }

    //
    // place element on grid
    _grdLoc    = loc;
    _grdLoc.round();
    _angle     = angle;
    var footPrint:Footprint = getFootprint(_angle);

    // add element's whole point references to grid
    var wholePts:Array = footPrint.wholeSquares;
    for (var i:Number = 0; i < wholePts.length; ++i) {

        wholePts[i].add(_grdLoc);

        // place element reference on the grid
        _grid.setAt(wholePts[i], this);
```

```
}

// add element's 1/2 point references to grid
var halfPts:Array = footPrint.halfSquares;
var corner:Boolean;
for (var i:Number = 0; i < halfPts.length; ++i) {

    halfPts[i].add(_grdLoc);
    corner = (i < 4);

    // place 1/2 element reference on the grid
    var val:Object = _grid.getAt(halfPts[i]);
    // empty square
    if (val == undefined) {

        _grid.setAt(halfPts[i], [{ element:this,
                  corner:corner }]);
    // partially filled square
    } else if (val instanceof Array) {

        val.push({ element:this, corner:corner });
        _grid.setAt(halfPts[i], val);
    }
}

// add _gridLoc to footprint front, back, left and right
var front:Array = footPrint.front;
for (var i:Number = 0; i < front.length; ++i) {
    front[i].add(_grdLoc);
}
var back:Array = footPrint.back;
for (var i:Number = 0; i < back.length; ++i) {
    back[i].add(_grdLoc);
}
var left:Array = footPrint.left;
for (var i:Number = 0; i < left.length; ++i) {
    left[i].add(_grdLoc);
}
var right:Array = footPrint.right;
for (var i:Number = 0; i < right.length; ++i) {
    right[i].add(_grdLoc);
}
var tpLeft:Point2D = footPrint.tpLeftOutside;
tpLeft.add(_grdLoc);
var tpRht:Point2D = footPrint.tpRhtOutside;
tpRht.add(_grdLoc);
var shiftLeftPt:Point2D  = footPrint.shiftLeftPt;
shiftLeftPt.add(_grdLoc);
var shiftRightPt:Point2D = footPrint.shiftRightPt;
shiftRightPt.add(_grdLoc);
var flankLeft:Point2D = footPrint.flankLeft;
flankLeft.add(_grdLoc);
var flankRht:Point2D = footPrint.flankRht;
flankRht.add(_grdLoc);

// save the new footprint
_footPrint = new Footprint( wholePts,
```

```
                                halfPts,
                                front,
                                back,
                                left,
                                right,
                                tpLeft,
                                tpRht,
                                shiftLeftPt,
                                shiftRightPt,
                                flankLeft,
                                flankRht,
                                footPrint.modifier);


        if (firstDraw) {

            // drawing unit for the first time
            _srnLoc = _grid.gridToPtLoc(_grdLoc);
            _srnLoc.add(_footPrint.modifier);

            _sprite.loc    = _srnLoc;
            _sprite.angle  = _angle;
            _sprite.active = 0;

        } else {


            //
            // contact with enemy

            var e:Element = elementMostInFront();

            if (e == undefined) {
                // special case test tpRhtOutside and tpLeftOutside pts
                // for diagonal contact
                e = _grid.getAt(_footPrint.tpRhtOutside);
                if (e == undefined) {
                    e = _grid.getAt(_footPrint.tpLeftOutside);
                }
                if (e instanceof Array ||
                    ! isDiagonal(Utils.compareAngle(this.angle,
                              e.angle)) )  { e = undefined; }
            }
            // don't engage own element
            if ( isFriendly(e) ) { e = undefined; }

            if (e != undefined) {

                // determine contact facing
                var contactAngle = Utils.cleanAngle(this.angle -
                        e.angle);
                switch (contactAngle) {
                    // frontal contact
                    case 180:

                        break;
                    // rear contact
```

```
                case 0:

                    e.turnToFace("Rear")
                    e = undefined;
                    break;
                // left flank contact to turn to face
                case 90:

                    e = e.getFrontRankElement();
                    e.firstTurnToFace("Left")
                    e = undefined;
                    break;
                // right flank contact to turn to face
                case 270:

                    e = e.getFrontRankElement();
                    e.firstTurnToFace("Right")
                    e = undefined;
                    break;
                // diagonal contacts to turn to face
                case 135: case 225: case 315: case 45:

                    e.turnToFaceEnemy(this)
                    e = undefined;
                    break;
                default:
                    throw new Error("Illegal contact angle of " +
                        contactAngle);
            }
        }

        // store move on the _movelist
        _srnLoc = _grid.gridToPtLoc(_grdLoc);
        _srnLoc.add(_footPrint.modifier);
        _movelist.push([_srnLoc, _angle, turning, e]);
        statusMoving();

        // if movement exhausted blacken the shadow
        if (_mvePts <= 0.5) { _shadowClip._alpha = _dark; }

    }

    resetShadow();

    checkForUnexpectedContact();
}

//
// aboutToEngage()
//
//  returns true if this element has any engagements
//  about to happen on its _movelist.  This test is
//  needed by the Battles state to check if the player has
//  ended his move but there are still resultant
//  battles/engagements that have not yet been triggered
//
//   return -- true if there is a engagement about to happen
```

```
//
public function aboutToEngage():Boolean {

    // note, first item [0] in move list can be
    // ignored as it will already have declared it's charge
    for (var i:Number = 1; i < _movelist.length; ++i) {
        if (_movelist[i][3] != undefined) {
            return true;
        }
    }
    return false;
}

//
// advance()
//
//  Move this Element to the next location on the _movelist
//
public function advance():Void {

    if (! _advancing && _movelist.length > 0) {

        var moveTo:Object = _movelist[0];

        var speed = speedLimit();

        // randomize speed slightly if clumsy
        if (! _regular) {
            speed += (2 - Utils.randomInt(1, 4));
        }

        // if turning
        if (moveTo[2]) {

            // 1600 = distance * 4 * 100
            _a.goToLocInTme(_sprite.number, moveTo[0], 1600/speed);

            _a.rotateInTime(_sprite.number, moveTo[1], 1600/speed);
        } else {
            // advance normal
            _a.goToLocAtSpd(_sprite.number, moveTo[0], speed/56);

        }

        // if moving into contact...
        // moveTo SHOULD USE ANONYMOUS CLASS
        if (moveTo[3] != undefined) {

            moveTo[3].statusEngaged();
            statusEngaged();

            /*
            switch(_type) {

                case "Elephants":
                    _game.playSnd("Elephants_Charge"); break;
```

```
            case "Expendables":
                _game.playSnd("Chariots_Charge"); break;

            case "Knights": case "Cavalry":
                _game.playSnd("HorseHeavy_Charge"); break;

            case "Lt. Horse":
                _game.playSnd("HorseLight_Charge"); break;

            case "Spears": case "Pikes": case "Swords":
                _game.playSnd("FootHeavy_Charge"); break;

            case "Archers": case "Crossbows":
            case "Lt. Infantry": case "Skirmishers":
                _game.playSnd("FootLight_Charge"); break;

            case "Warriors": case "Hordes":
                _game.playSnd("FootHorde_Charge"); break;

            case "Artillery":
                _game.playSnd("Artillery_Charge"); break;

            case "Baggage":
            default:
                _game.playSnd("charge"); break;
        }
        */
        _game.playSnd("charge"); break;


    } else {

        /*
        switch(_type) {

            case "Spears":
                _game.playSnd("Try_to_walk"); break;

            case "Lt. Infantry":
                _game.playSnd("FootLight_Walk"); break;

            case "Elephants":
                _game.playSnd("Elephants_Walk"); break;

            case "Expendables":
                _game.playSnd("Chariots_Walk"); break;

            case "Artillery":
                _game.playSnd("Artillery_Walk"); break;

            case "Knights": case "Cavalry":
                _game.playSnd("HorseHeavy_Walk"); break;

            case "Lt. Horse":
                _game.playSnd("HorseLight_Walk"); break;

            default:
```

```
                    _game.playSnd("longWalk"); break;
            }
            */

            _game.playSnd("longWalk"); break;

        }

        _sprite.active = 1;
        _advancing = true;

        // if movement exhausted or not in Movement phase
        //  then darken the shadow
        if (_mvePts > 0.5 && _game.state == "Movement") {
            _shadowClip._alpha = _light;
        } else {
            _shadowClip._alpha = _dark;
        }

    }
}

//
// speedLimit()
//
//  return -- the speed of any element in front if it's slower
//
private function speedLimit():Number {

    var speed:Number = _baseMve;
    var e:Element = elementInFront();
    var eSpeed = e.speedLimit();
    if (e != undefined && eSpeed < speed) {
        speed = eSpeed;
    }

    return speed;
}

//
// globalLoc()
//
//  return the global location of this element,
//  or a specific footprint corner/part
//
//   return     -- a rect defining the corners of this Element
//
public function globalLoc(crnr:String):Point2D {

    var pt:Point2D;
    if (crnr == undefined) {
        pt = _srnLoc;
    } else {
        pt = _grid.gridToPtLoc(_footPrint[crnr]);
    }

    return makeGlobal(pt);
```

```
}

//
// globalRect()
//
//  Make Rect of Element global according to the _sprite path
//
//   return     -- a rect defining the corners of this Element
//
public function globalRect():Rect {

    // get the four courners of the element
    var rect:Rect = Utils.makeRect([_footPrint.leftInside,
                                    _footPrint.rightInside,
                                    _footPrint.bkLeftInside,
                                    _footPrint.bkRightInside]);

    var ptA:Point2D = _grid.gridToPtLoc(rect.firstPt);
    var ptB:Point2D = _grid.gridToPtLoc(rect.secondPt);
    ptA = makeGlobal(ptA);
    ptB = makeGlobal(ptB);

    return new Rect(ptA.x, ptA.y, ptB.x, ptB.y);
}

//
// makeGlobal()
//
//  Make a Point2D global according to the _sprite path
//
//   args:
//      pt      -- point to convert
//
//   return     -- a refernce to the new sprite
//
private function makeGlobal(pt:Point2D):Point2D {

    // find global position of pt
    var myPoint:Object = { x:pt.x, y:pt.y };
    _sprite.path.localToGlobal(myPoint);

    return new Point2D(myPoint.x, myPoint.y);
}

//
// atDestination()
//
//  Triggered by Animatem when this elements sprite
//  reaches a destination it was moving to.
//
//   return -- list of adjacent elements
//
public function atDestination():Void {

    if (_advancing) {

        // take move command off the front of list
```

```
        var moveTo:Object = _movelist.shift();

        // stop sprite from animating
        _sprite.angularVel = 0;
        _sprite.angle  = moveTo[1];
        _advancing     = false;

        // darken the shadow
        _shadowClip._alpha = _dark;

        // get the next move data point, if any
        advance();
    }
}

//
// adjacent()
//
//  Return a list of any Elements directly adjacent to this one.
//  This is useful for establishing Elements in a group
//
//   return -- list of adjacent elements
//
public function adjacent():Array {

    var e:Element;
    var r:Array = new Array();

    e = elementInFront();
    if (e != undefined && e.state != "Highlight" &&
        e.sprite.angle == this.sprite.angle) {
        r.push(e);
    }
    e = elementBehind();
    if (e != undefined && e.state != "Highlight" &&
        e.sprite.angle == this.sprite.angle) {
        r.push(e);
    }
    e = elementToLeft();
    if (e != undefined && e.state != "Highlight" &&
        e.sprite.angle == this.sprite.angle) {
        r.push(e);
    }
    e = elementToRight();
    if (e != undefined && e.state != "Highlight" &&
        e.sprite.angle == this.sprite.angle) {
        r.push(e);
    }

    return r;
}

//
// elementInFront()
//
//  return - Element aligned directly in front
//
```

```
public function elementInFront():Element {
    return strictTestForElement(_footPrint.frontOutside,
             "backInside");
}

//
// elementBehind()
//
//  return - Element aligned directly to rear
//
public function elementBehind():Element {
    return strictTestForElement(_footPrint.backOutside,
             "frontInside");
}

//
// elementToLeft()
//
//  return - Element aligned directly to left
//
public function elementToLeft():Element {
    return strictTestForElement(_footPrint.leftOutside,
             "rightInside");
}

//
// elementToRight()
//
//  return - Element aligned directly to right
//
public function elementToRight():Element {
    return strictTestForElement(_footPrint.rightOutside,
             "leftInside");
}

//
// strictTestForElement()
//
//  Test for an Element at a specific footprint point
//
//   args:
//      pt  -- point being tested
//      str -- footprint type (as a string) needed to be matched
//
//   return     -- a refernce to the new sprite
//
private function strictTestForElement(pt:Point2D,
         str:String):Element {

    var val = _grid.getAt(pt);
    if (val instanceof Element && isFriendly(val) &&
        val.footPrint[str].equal(pt)) {
        return val;
    }

    return undefined;
}
```

```
//
// elementMostInFront()
//
//  return - Element most directly in front edge
//
public function elementMostInFront():Element {
    return _game.testForElement(_footPrint.front);
}

//
// elementMostBehind()
//
//  return - Element most directly to rear edge
//
public function elementMostBehind():Element {
    return _game.testForElement(_footPrint.back);
}

//
// elementMostToLeft()
//
//  return - Element most directly to left edge
//
public function elementMostToLeft():Element {
    return _game.testForElement(_footPrint.left);
}

//
// elementMostToRight()
//
//  return - Element most directly to right edge
//
public function elementMostToRight():Element {
    return _game.testForElement(_footPrint.right);
}

//
// elementsInFront()
//
//  return - list of Elements along front edge
//
public function elementsInFront():Array {
    return _game.testForElements(_footPrint.front);
}

//
// elementsBehind()
//
//  return - list of Elements along rear edge
//
public function elementsBehind():Array {
    return _game.testForElements(_footPrint.back);
}

//
// elementsToLeft()
```

```
//
//  return - list of Elements along left edge
//
public function elementsToLeft():Array {
    return _game.testForElements(_footPrint.left);
}

//
// elementsToRight()
//
//  return - list of Elements along right edge
//
public function elementsToRight():Array {
    return _game.testForElements(_footPrint.right);
}

//
// leftFlanked()
//
//  return - any enemy element attacking the left flank
//
public function leftFlanked():Element {
    var e:Element = _grid.getAt(_footPrint.leftOutside);
    if ( ! isFriendly(e) && Utils.compareAngle(e.angle,
              _angle + 90) == 0) {
        return e;
    } else {
        return undefined;
    }
}

//
// rightFlanked()
//
//  return - any enemy element attacking the right flank
//
public function rightFlanked():Element {
    var e:Element = _grid.getAt(_footPrint.rightOutside);
    if ( ! isFriendly(e) && Utils.compareAngle(e.angle,
              _angle - 90) == 0) {
        return e;
    } else {
        return undefined;
    }
}

//
// rearAttacked()
//
//  return - any element attacking the rear of this element
//
public function rearAttacked():Element {
    var e:Element = _grid.getAt(_footPrint.backOutside);
    if ( ! isFriendly(e) && Utils.compareAngle(e.angle,
              _angle - 180) == 0) {
        return e;
    } else {
```

```
            return undefined;
        }
    }

    //
    // leftOverlap()
    //
    //  return - element causing an overlap on left corner
    //
    public function leftOverlap():Element {

        // first check for enemy to immediate left
        var e:Element = elementMostToLeft();
        var angleDiff:Number = Utils.compareAngle(e.angle, _angle);
        if (isFriendly(e) || ! (angleDiff == 180 || angleDiff == 0 ||
            angleDiff == 135)) { e = undefined; }

        // check for top left overlapping element
        if (e == undefined) {
            e = _grid.getAt(_footPrint.tpLeftOutside);
            var angleDiff = Utils.compareAngle(e.angle, _angle);
            if (isFriendly(e) || !
                (angleDiff == 180 || angleDiff == 135) ||
                 e.status == "Engaged") {
                 e = undefined;
            }
        }
        return e;
    }

    //
    // rightOverlap()
    //
    //  return - element causing an overlap on right corner
    //
    public function rightOverlap():Element {

        // first check for enemy to immediate right
        var e:Element = elementMostToRight();
        var angleDiff:Number = Utils.compareAngle(e.angle, _angle);
        if (isFriendly(e) || ! (angleDiff == 180 || angleDiff == 0 ||
            angleDiff == 225)) { e = undefined; }

        // check for top right overlapping element
        if (e == undefined) {
            e = _grid.getAt(_footPrint.tpRhtOutside);
            angleDiff = Utils.compareAngle(e.angle, _angle);
            if (isFriendly(e) || !
                (angleDiff == 180 || angleDiff == 225) ||
                e.status == "Engaged") { e = undefined; }
        }
        return e;
    }

    //
    // recoilPt()
    //
```

```
//  return a point to recoil to, if available
//
private function recoilPt(squares:Number):Point2D {

    // test if element can recoil?
    var nGridLoc:Point2D = _grdLoc;
    var mvMod:Point2D = (this.diagonal ? new Point2D(-squares,
             squares) : new Point2D(0, squares));
    mvMod.rotate(this.theta);
    nGridLoc.add(mvMod);

    if (testLocation(nGridLoc, _angle)) {
        return nGridLoc;
    } else {
        return undefined;
    }
}

//
// Battle results
//

//
// recoil()
//
//  This Element (and any behind it) recoil a
//  base depth
//
public function recoil(useDepth:Number, delay:Number):Boolean {

    var r:Boolean = true;

    // PATCH: ENSURE USEDEPTH AT LEAST BASE DEPTH
    if (useDepth < this.depth || useDepth == undefined) {
        useDepth = this.depth;
    }

    var list:Array = elementsBehind();
    for(var i:Number = 0; i < list.length; ++i) {
        if (isFriendly(list[i]) && list[i].angle == _angle) {
            list[i].recoil(useDepth);
        }
    }


    var loc:Point2D;
    if (useDepth == undefined) {
        loc = recoilPt(this.depth);
    } else {
        loc = recoilPt(useDepth);
    }

    if (loc != undefined) {
        moveMeTo(loc, _angle);
    } else {
        killed(true, delay); r = false;
    }
```

```
    advance();

    return r;
}

//
// repulsed()
//
//  This Element (and any behind it) are repulsed, which is
//  similar to flee but does not retreat not as far.
//
public function repulsed():Void {

    var e:Element = elementBehind();
    if (e instanceof Element) { e.repulsed(); }

    if (recoil()) {

        // retreat an additional number of times
        var n:Number = Math.round(_baseMve/_baseDepth) - 2;
        for (var i:Number = 0; i < n; ++i) {
            var loc:Point2D = recoilPt(this.depth);
            if (loc != undefined) {
                moveMeTo(loc, _angle);
            } else {
                break;
            }
        }
        advance();
    }
}

//
// flee()
//
//  This Element (and any behind it) flee from battle.
//
public function flee():Void {

    var e:Element = elementBehind();
    if (e instanceof Element) { e.flee(); }

    if (recoil()) {

        // turn 180
        var angle:Number = Utils.cleanAngle(_angle - 180);
        var mod:Number = _baseDepth - 1;
        var loc:Point2D = recoilPt(this.depth + this.depth - 1);
        if (loc != undefined) {
            moveMeTo(loc, angle);

            // flee forward an additional number of times
            var n:Number = Math.round(_baseMve/_baseDepth) - 1;
            for (var i:Number = 0; i < n; ++i) {
                loc = recoilPt(- this.depth);
                if (loc != undefined) {
                    moveMeTo(loc, angle);
```

```
                } else {
                    break;
                }
            }
            advance();
        } else {
            killed(true);
        }
    }
}

//
// spent()
//
//  Hark! This Element is spent (similar to killed).
//
public function spent():Void {
    _game.playSnd("death");
    remove();
}

//
// killed()
//
//  Hark! This Element is dead.
//
//   args:
//      fromRecoil -- true if death due to another Element
// recoiling.
//           delay -- time delay if death due to shooting.
//
public function killed(fromRecoil:Boolean, delay:Number):Void {

    if (fromRecoil == undefined) { fromRecoil = false; }

    // Elephants and Expendables recoil before dying wrecking havok
    if (! fromRecoil &&
        (_type == "Elephants" || _type == "Expendables")) {
        recoil();
    }

    // sometimes elements behind are also killed
    var list:Array = elementsBehind();

    for (var i:Number = 0; i < list.length; ++i) {
        if (isFriendly(list[i])) {

            // Elephants and Expendables wreck havok
            if (_type == "Elephants" || _type == "Expendables") {
                list[i].killed(false, delay);
            }

            // Skirmishers, Archers, Crossbows, Hordes
            //  or Artillery die if in the way
            if (_type != "Skirmishers" &&
               (list[i].type == "Skirmishers" ||
                list[i].type == "Archers" ||
```

```
                list[i].type == "Crossbows" ||
                list[i].type == "Hordes" ||
                list[i].type == "Artillery")) {

                list[i].killed(false, delay);
            }

            // if they are in combat they will be killed
            if (list[i].status == "Engaged") {
                list[i].killed(false, delay);
            }

        }
    }

    // if killled by shooting then the sound
    //  needs to be delayed by delay seconds
    _game.playSnd("death", delay);
    remove();
}

//
// pursue()
//
//  Pursue a retreating enemy Element
//
//   args:
//          squares -- number of grid squares to pursue.
//
public function pursue(squares:Number):Void {

    // find element behind before moving
    var e:Element = elementBehind();

    // pursue forward
    var nGridLoc:Point2D = _grdLoc;
    var mvMod:Point2D = (this.diagonal ? new Point2D(squares,
             -squares) : new Point2D(0, -squares));
    mvMod.rotate(this.theta);
    nGridLoc.add(mvMod);
    moveMeTo(nGridLoc, _angle);

    // elements behind also pursue
    if (e instanceof Element) { e.pursue(squares); }

    advance();
}

//
// checkForUnexpectedContact()
//
//  Check if Element has moved/stumbled into an enemy flank
//  overlap or  a diagonal contact, triggering an engagement.
//
public function checkForUnexpectedContact():Void {
    if (_status != "Engaged") {
        var e:Element = leftFlanked();
```

```
        if (e == undefined) { e = rightFlanked(); }
        if (e == undefined) {
            e = elementMostInFront();
            if (isFriendly(e)) {
                e = undefined;
            } else {
                var contactAngle =
                    Utils.cleanAngle(this.angle - e.angle);
                if (!(contactAngle == 135 ||
                      contactAngle == 225)) {
                    e = undefined;
                }
            }
        }
        if (e instanceof Element) { turnToFaceEnemy(e); }
    }
}

//
// engagingWith()
//
//  Return whatever enemy Element this Element is fighting
//
//      return  -- engaged enemy Element
//
public function engagingWith():Element {
    if (_status == "Engaged") {
        return _grid.getAt(_footPrint.frontOutside);
    } else {
        return undefined;
    }
}

//
// turnToFaceEnemy()
//
//  Turn this Element to face an attacking enemy Element.
//  Elements already engaged will not turn to face.
//
//   args:
//          e   -- enemy Element to turn to face to
//
//      return  -- return true if able to turn to face
//
public function turnToFaceEnemy(e:Element):Boolean {
    if (_status != "Engaged") {
        var nLoc   = e.footPrint.frontOutside;
        var nAngle = Utils.cleanAngle(e.angle - 180);
        if (testLocation(nLoc, nAngle)) {
            moveMeTo(nLoc, nAngle);
            advance();
            return true;
        }
    }

    return false;
}
```

```
//
// firstTurnToFace()
//
//  Turn this Element to a facing, such as "Rear", "Left", "Right"
//  The first Element in a column that needs to turn to face is
//  a special case, subsequent Elements behind use the regular
//  (recursive) turnToFace function.
//  If the first Element is Engaged then the turnToFace is passed
//  through to the element directly to its rear.
//
//   args:
//      facing  -- direction to turn, "Rear", "Left", "Right"
//
//      return  -- return true if able to turn to face
//
public function firstTurnToFace(facing:String):Boolean {
    // if the front rank element is engaged
    //  then only turn those behind
    if (_status == "Engaged") {
        var e:Element = elementBehind();
        var recoilBy:Number = this.width - this.depth;
        if (e.baseDepth + this.baseDepth <= 7) {
            recoilBy = this.width - (this.depth + e.depth);
            e = e.elementBehind();
        }
        if (e != undefined) {
            e.recoil(recoilBy);
            return e.turnToFace(facing);
        }
    } else {
        return turnToFace(facing);
    }

    return false;
}

//
// turnToFace()
//
//  Recursive function called by firstTurnToFace.
//  Turn this Element to a facing, such as "Rear", "Left", "Right"
//  Elements already engaged will not turn to face. Elements
// turning
//  to face a flank will often have to push elements behind them
// back.
//  All Elements in a column will turn to face the same direction.
//
//   args:
//      facing  -- direction to turn, "Rear", "Left", "Right"
//
//      return  -- return true if able to turn to face
//
public function turnToFace(facing:String):Boolean {

    var e:Element; var ee:Element;
    var loc:Point2D; var angle:Number;
```

```
    var recoilBy:Number;

    if (_status != "Engaged") {

        if (facing == "Rear") {
            loc   = _footPrint.backInside;
            angle = Utils.cleanAngle(_angle - 180);
        } else {

            //recoil any elements behind
            e = elementBehind();
            recoilBy = this.width - this.depth;
            if (e != undefined &&
                e.baseDepth + this.baseDepth <= 7) {
                // turn element behind (ee) with this element
                ee = e;

                ee.removeData();
                e = e.elementBehind();
                recoilBy =  this.width - (this.depth + ee.depth);
            }
            if (recoilBy > 0) { e.recoil(recoilBy); }

            // get new turning element positions
            if (facing == "Right") {
                loc   = _footPrint.flankRht;
                angle = Utils.cleanAngle(_angle + 90);
            } else {
                loc = _footPrint.flankLeft;
                angle = Utils.cleanAngle(_angle - 90);
            }
        }

        removeData();
        setLoc(loc, angle); advance();
        if (ee != undefined) { ee.setLoc(_footPrint.backOutside,
                 angle); ee.advance(); }

        // recoiled elements also turn to face!
        if (e != undefined)  { e.turnToFace(facing); }

        return true;
    }

    return false;
}

//
// isFriendly()
//
//  Check if an element is friendly with this Element
//
//   args:
//      e       -- Element being tested with this Element
//
//   return     -- return true if Element e is a friendly element
//
```

```
public function isFriendly(e:Element):Boolean {
    return (e.player.thePlayer == this.player.thePlayer);
}

//
// getFrontRankElement()
//
//  Recursive call to find the element at the front of
//  a column of Elements.
//
//   return     -- the first element in a column
//
public function getFrontRankElement():Element {
    var e:Element = elementInFront();
    if (e == undefined) {
        return this;
    } else {
        return e.getFrontRankElement(e);
    }
}

//
// getRearRankElement()
//
//  Recursive call to find the element at the rear of
//  a column of Elements.
//
//   return     -- the last element in a column
//
public function getRearRankElement():Element {
    var e:Element = elementBehind();
    if (e == undefined) {
        return this;
    } else {
        return e.getRearRankElement();
    }
}

//
// accessors
public function get grdLoc():Point2D     { return_grdLoc.clone();}
public function get spriteN():Number     { return_sprite.number; }
public function get sprite():Sprite         { return _sprite; }
public function get type():String           { return _type; }
public function get grade():String          { return _grade; }
public function get desc():String           { return _desc; }
public function get picture():String        { return _picture; }
public function get regular():Boolean       { return _regular; }
public function get movePts():Number        { return _mvePts; }
public function get baseMve():Number        { return _baseMve; }
public function get diagonal():Boolean      {return(_angle%90!=0);}
public function get state():String          { return _state; }
public function get player():Player         { return _player; }
public function get command():String        { return _cmdGroup; }
public function get status():String         { return _status; }
public function get angle():Number          { return _angle; }
public function get footPrint():Footprint   { return _footPrint; }
```

```
public function get scrnLoc():Point2D       { return _srnLoc; }
public function get combatTbl():CombatTable { return _combatTbl; }
public function get terrain():String        { return _terrain; }
public function get isGeneral():String      { return _isGeneral; }
public function get game():Chevalier        { return _game; }
public function get baseDepth():Number      { return _baseDepth; }
public function get lastMvMade():String     { return _lastMvMade; }
public function get influence():Number      { return _influence; }
public function get PIPsPaid():Number       { return _PIPsPaid; }
public function get withdrawn():Boolean     { return _withdrawn; }
public function get nudges():Number         { return _nudges; }
private function isDiagonal(angle:Number):Boolean {
    return (angle%90 != 0);
}

// return diagonal base depth equiv.
public function get diagonalDepth():Number {

    switch (_baseDepth) {
        case 3: return 2;
        case 4: return 3;
        case 7: return 5;
    }

    throw new Error("Illegal base depth.");
}

// return depth of unit depending on its angle
public function get depth():Number {
    if (this.diagonal) { return this.diagonalDepth; }
    return _baseDepth;
}
// return width of unit depending on its angle
public function get width():Number {
    if (this.diagonal) { return 5; } else { return 7; }
}
public function get theta():Number {
    // "theta" is the angle by which data points need to be rotated
    var theta:Number = _angle + (this.diagonal ? 45 : 90);
    theta = Utils.cleanAngle(theta);
    return theta;
}
public function getMoveResult(cmd:String):Array {
    return _StoredMoveResults[cmd];
}

// mutators
public function set movePts(val:Number):Void {
    _mvePts = val;
}
public function set lastMvMade(val:String):Void {
    _lastMvMade = val;
}
public function set PIPsPaid(val:Number):Void {
    _PIPsPaid = val;
}
public function set withdrawn(val:Boolean):Void {
```

```
        _withdrawn = val;
    }
    public function set nudges(val:Number):Void {
        _nudges = val;
    }
    public function storeMoveResult(cmd:String, gridLoc:Point2D,
             angle:Number, cost:Number):Void {
        _StoredMoveResults[cmd] = [gridLoc, angle, cost];
    }
}
```

Footprint.as

```
////////////////////////////////////////////////////////////////////////
//
//  Footprint.as
//
//       Author: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: Footprint is a data structure object that maintains
//               lists of Point2Ds describing an Element's key
//               locations represented on the Grid.  There are three
//               sets of template Footprint instances created by the
//               Player Object initializer. One Foorprint for each
//               compass direction in each set.  These templates are
//               used to quickly generate a unique key Footprint kept
//               by each Element whenever it moves.  When looking for
//               information on an Element and what Grid points it
//               covers on the map the key Footprint for the Element
//               is referred to.
//
//        Notes:
//

class Footprint {

    //
    // instance members
    // array of whole square points for this footprint
    private var _wholeSquares:Array;

    // array of 1/2 square points for this footprint
    private var _halfSquares:Array;

    // locations at front of footprint
    private var _front:Array;

    // locations at back of footprint
    private var _back:Array;

    // locations to left of footprint
    private var _left:Array;

    // locations to right of footprint
    private var _right:Array;

    // outside top left point used to check for overlap
    private var _tpLeft:Point2D;

    // outside top right point used to check for overlap
    private var _tpRht:Point2D;

    // shifted back left point used for friendly element shifts
    private var _sftLeft:Point2D;
```

```
// shifted back right point used for friendly element shifts
private var _sftRht:Point2D;

// flank contact point for elements contacting on left
private var _flankLeft:Point2D;

// flank contact point for elements contacting on right
private var _flankRht:Point2D;

// modifier for drawing element
private var _modifier:Point2D;

//
// constructor
public function Footprint(wholeSquares:Array,
                          halfSquares:Array,
                          front:Array,
                          back:Array,
                          left:Array,
                          right:Array,
                          tpLft:Point2D,
                          tpRht:Point2D,
                          sftLeft:Point2D,
                          sftRht:Point2D,
                          flankLeft:Point2D,
                          flankRht:Point2D,
                          modifier:Point2D) {

    _wholeSquares = wholeSquares;
    _halfSquares  = halfSquares;
    _front        = front;
    _back         = back;
    _left         = left;
    _right        = right;
    _tpLeft       = tpLft;
    _tpRht        = tpRht;
    _sftLeft      = sftLeft;
    _sftRht       = sftRht;
    _flankLeft    = flankLeft;
    _flankRht     = flankRht;
    _modifier     = modifier;
}

//
// accessors
//
// make sure a clone is given to avoid permanent alteration
// to permanent footprint instances
public function get wholeSquares():Array {
    return cloneArray(_wholeSquares);
}
public function get halfSquares():Array {
    return cloneArray(_halfSquares);
}
public function get front():Array {
    return cloneArray(_front);
```

```
}
public function get back():Array {
    return cloneArray(_back);
}
public function get left():Array {
    return cloneArray(_left);
}
public function get right():Array {
    return cloneArray(_right);
}
public function get frontInside():Point2D {
    return _wholeSquares[0];
}
public function get backInside():Point2D {
    return _wholeSquares[1].clone();
}
public function get leftInside():Point2D {
    return _wholeSquares[2];
}
public function get rightInside():Point2D {
    return _wholeSquares[3];
}
public function get bkLeftInside():Point2D {
    return _wholeSquares[4];
}
public function get bkRightInside():Point2D {
    return _wholeSquares[5];
}
public function get frontOutside():Point2D {
    return _front[0].clone();
}
public function get backOutside():Point2D {
    return _back[0].clone();
}
public function get leftOutside():Point2D {
    return _left[0];
}
public function get rightOutside():Point2D {
    return _right[0];
}
public function get tpLeftOutside():Point2D {
    return _tpLeft.clone();
}
public function get tpRhtOutside():Point2D {
    return _tpRht.clone();
}
public function get shiftLeftPt():Point2D {
    return _sftLeft.clone();
}
public function get shiftRightPt():Point2D {
    return _sftRht.clone();
}
public function get flankLeft():Point2D {
    return _flankLeft.clone();
}
public function get flankRht():Point2D {
    return _flankRht.clone();
```

```
    }
    public function get modifier():Point2D {
        return _modifier.clone();
    }
    private function cloneArray(array:Array):Array {
        var r:Array = new Array();
        for (var i:Number = 0; i < array.length; ++i) {
            r.push(array[i].clone());
        }
        return r;
    }

    //
    // mutators
    public function set wholeSquares(val:Array):Void {
        _wholeSquares = val;
    }
    public function set halfSquares(val:Array):Void {
        _halfSquares = val;
    }
    public function set modifier(val:Point2D):Void {
        _modifier = val;
    }
}
```

MoveType.as

```
////////////////////////////////////////////////////////////////////////
//
//  MoveType.as
//
//       AUTHOR: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: The MoveType Object is a wrapper class for defining
//               attributes pertaining to a specific move operation
//               to be performed an a group of Elements. All MoveType
//               objects are created by the Scroll Object whose
//               function is primarily to move Elements around the
//               Grid. MoveType Objects are almost always created in
//               pairs, one defining the move as performed by
//               Elements with a diagonal orientation, the other for
//               Elements that are horizontal or vertical.
//
//               Sometimes a temporary MoveType is created by the
//               Scroll Object to perform dynamically changing moves
//               such as Wheels, or when the Scroll needs to
//               "shift" a group of Elements so they conform to
//               another group or to make contact with the enemy. In
//               such instances the MoveType is often mutated using
//               plusOne(), or minusOne() methods.  The nonZero()
//               method is used after calculating dynamic moves to
//               ensure the resultant modified move isn't a
//               "zero" move that results in no movement.
//
//      Methods:
//
//        MoveType() - Constructor
//         fixMove() - Generate a new MoveType from this one, this is
//                     used for wheeling elements and shifting elements
//                     into contact.
//         plusOne() - Generate a new move that is one step further.
//        minusOne() - Generate a new move that is one step shorter.
//      nudgeRight() - If a diagonal move then nudge to the right (+1
//                     in x). Non-diagonal moves cannot be nudged.
//         nonZero() - A move type should never be passed with both x
//                     and y as zero as the controller will show the
//                     move as enabled but selecting it will have no
//                     effect. If a "zero" move then plusOne.
//        Notes:
//

class MoveType {

    //
    // instance members

    // name of moveType
    private var _name:String;
```

```
// location modifier, results in new position when added to
// element's grid loc
private var _loc:Point2D;

// the movement cost to move this way
private var _cost:Number;

// the change in the angle caused by this movement
private var _angle:Number;

// true if this is a diagonal move type
private var _diagonal:Boolean;

//
// constructor
public function MoveType(name:String, loc:Point2D, cost:Number,
          angle:Number, diagonal:Boolean) {
    _name     = name;
    _loc      = loc;
    _cost     = cost;
    _angle    = angle;
    _diagonal = diagonal;
}

//
// fixMove()
//
//   Generate a new MoveType from this one,
//   this is used for wheeling elements and shifting
//   elements into contact
//
//   return     -- reference to new MoveType
//
public function fixMove(locMod:Point2D, costMod:Number):MoveType {

    var nLoc:Point2D = _loc.clone(); nLoc.add(locMod);
    var nCost:Number = _cost + costMod;

    return new MoveType(_name, nLoc, nCost, _angle, _diagonal);
}

//
// plusOne()
//
//   Generate a new move that is one step further
//
//   return     -- reference to new MoveType
//
public function plusOne():MoveType {
    if (_diagonal) {
        return fixMove(new Point2D(+1, -1), 1.5);
    } else {
        return fixMove(new Point2D( 0, -1), 1);
    }
}
```

```
//
// minusOne()
//
//   Generate a new move that is one step shorter
//
//   return     -- reference to new MoveType
//
public function minusOne():MoveType {
    if (_diagonal) {
        return fixMove(new Point2D(-1, +1), 1.5);
    } else {
        return fixMove(new Point2D( 0, +1), 1);
    }
}

//
// nudgeRight()
//
//  If a diagonal move then nudge to the right (+1 in x)
//  Non-diagonal moves cannot be nudged
//
//   return     -- reference to this MoveType
//
public function nudgeRight():MoveType {
    if (_diagonal) {
        return fixMove(new Point2D(+1, 0), 1)
    } else {
        return this;    // no such nudge
    }
}

//
// nonZero()
//
//  A move type should never be passed with both x and y as zero
//  as the controller will show the move as enabled but selecting
//  it will have no effect.  If a "zero" move then plusOne.
//
//   return     -- reference to this MoveType
//
public function nonZero():MoveType {
    if (_loc.x == 0 && _loc.y == 0) {
        return this.plusOne();
    } else {
        return this;
    }
}

//
// accessors
public function get name():String  { return _name; }
public function get loc():Point2D  { return _loc.clone(); }
public function get cost():Number  { return _cost; }
public function get angle():Number { return _angle; }

//
// mutators
```

```
        public function set loc(val:Point2D):Void  { _loc   = val; }
        public function set cost(val:Number):Void  { _cost  = val; }
        public function set angle(val:Number):Void { _angle = val; }

    }
```

IGameState.as

```
////////////////////////////////////////////////////////////////////////
//
//  IGameState.as
//
//       Author: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: All Game State objects, implement the interface
//               IGameState, which ensures that each state object
//               contains the eight standard methods that are regularly
//               sent by the controlling Chevalier Object. In
//               particular, each state is responsible for setting up
//               the screen to commence the state, and cleaning up the
//               screen on conclusion of the state. Specifically, the
//               methods required in each State Object are; update();
//               tells the state to update the screen; mouseDown(),
//               state must handle the user pressing the mouse at a
//               coordinate; mouseUp(), state must handle user
//               releasing the mouse; mouseMove(), state must handle
//               user moving mouse to a new coordinate; keyDown(),
//               user has pressed a certain key; keyUp(), user releases
//               that key; start(), handle setting up and stating the
//               Game State; end(), finish and clean up the game state.
//               The Chevalier Object simply redirects any mouse and
//               keyboard input it receives to whichever State Object
//               corresponds to the current state that Chevalier is in.
//
//             Specifically, the methods required in each State are:
//
//          update() - tells the state to update the screen.
//       mouseDown() - state must handle the user pressing the mouse
//                     at a coordinate; mouseUp(), state must handle
//                     user releasing the mouse; mouseMove(), state
//                     must handle user moving mouse to a new
//                     coordinate.
//         keyDown() - user has pressed a certain key.
//           keyUp() - user releases that key.
//           start() - handle setting up and stating the Game State.
//             end() - finish and clean up the game state.
//
//    The Chevalier Object simply redirects any mouse and keyboard
//    input it receives to whichever State Object corresponds to
//    the current state that Chevalier is in.
//

interface IGameState {

    public function update():Void;
```

```
    public function mouseDown():Void;
    public function mouseUp():Void;
    public function mouseMove():Void;
    public function keyDown():Void;
    public function keyUp():Void;
    public function start():Void;
    public function end():Void;
}
```

Choose.as

```
////////////////////////////////////////////////////////////////////////
//
//  Choose.as
//
//       Author: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: The Choose state is the initial state that Chevalier
//               begins in.  Under the Choose state the game scenario
//               is chosen or a fantasy scenario is chosen with each
//               player choosing different armies to play.  The
//               fantasy option is much like DBM tournament play,
//               where any army from any historical period is matched
//               with any other.
//
//       Method:
//
//          Choose() - Constructor
//           start() - When the choose section is started Chevalier
//                     must go to the Chevalier "choose" frame
//      rollChoose() - Rollover triggers a brief description of the
//                     selection for the button being rolled.  Used by
//                     both the choose battle and choose fiction
//                     screens. The choose fiction screen has an
//                     additional feature of a silhouette that is
//                     fade+superimposed over the selection showing
//                     that armies insignia.
//   btnViewBattle() - Loads the battle description
//           btnOK() - Confirms a battle selection
// btnBackToBattle() - goes from the choose fiction screen back to the
//                     select battle screen
//    createBattle() - This is used by the battle game sequence. Once a
//                     battle has chosen that battle is loaded from the
//                     corresponding XML file and the players armies
//                     are registered with the main Chevalier game
//                     object.
//       btnChoose() - This is used by the "fictional" game sequence.
//                     Once a player has chosen an army, all buttons
//                     are disabled so they they can't be accidentally
//                     clicked and their army choice is registered with
//                     the main Chevalier game object.
//    createPlayer() - This is used by the "fictional" game sequence to
//                     trigger the XML loading of player one or player
//                     two's army. A randomized terrain is also chosen
//                     here, and buttons disabled for armies that have
//                     been selected by the other player.
//       startGame() - Exits the choose state and commences the actual
//                     game. This is called from the Flash score after
//                     both armies have been created from the XML files
//                     at the end of the "loadBattle" sequence.
//        Notes:
//
```

```
class Choose implements IGameState {

    // instance members

    // game object
    private var _game:Chevalier;

    // path to the control movie (usually _root)
    private var _pathCtrl:MovieClip;

    // sprite used for the battle map
    private var _mapSpr:Sprite;

    // true if player choosing fictional battle
    private var _chooseFiction:Boolean = false;

    // xml object for reading army or battle data
    private var _xml:XML;

    // list of possible armies
    private var _armies:Array;

    //
    //  Constructor
    //
    //   args:
    //      game -- The Chevalier game object is needed
    //  pathCtrl -- path to the general control so display frame can
    //              be changed
    //    mapSpr -- Map sprite is needed as selctions can determine
    //              which map graphic is going to be used during
    //              the game
    //
    public function Choose(game:Chevalier, pathCtrl:MovieClip,
             mapSpr:Sprite) {
        _game     = game;
        _pathCtrl = pathCtrl;
        _mapSpr   = mapSpr;

        _armies   = ["Crusader", "Saracen", "English", "French",
                 "Macedonian", "Persian"];

        // Create new XML Object and set ignoreWhite true
        _xml = new XML();
        _xml.ignoreWhite = true;
        Utils.setXMLreader(_xml, "gChevalier", "addElement");
    }

    //
    // interfaces
    //

    //
    // start()
    //
    //  When the choose section is started Chevalier must
    //  go to the Chevalier "choose" frame
```

```
//
public function start():Void {
    _pathCtrl.gotoAndStop("choose");
}
public function end():Void          {}
public function update():Void       {}
public function mouseDown():Void    {}
public function mouseUp():Void      {}
public function mouseMove():Void    {}
public function keyDown():Void      {}
public function keyUp():Void        {}

//
// rollChoose()
//
//  Rollover triggers a brief description of the selection
//  for the button being rolled.  Used by both the choose battle
//  and choose fiction screens.  The choose fiction screen has
//  an additional feature of a silhouette that is fade+superimposed
//  over the selection showing that armies insignia.
//
public function rollChoose(val:String):Void {

    _pathCtrl._chooseAnim._choose._chooseAn.gotoAndPlay(val);

    if (_chooseFiction) {
        _pathCtrl._chooseAnim._choose._insignia.gotoAndPlay(val);
    }
}

//
// btnViewBattle()
//
//  Loads the battle description
//
//   args:
//      choice  -- selected battle
//
public function btnViewBattle(choice:String):Void {
    if (choice == "Fictional") {
        _pathCtrl._chooseAnim.gotoAndPlay("go_fiction");
        _chooseFiction = true;
    } else {
        _pathCtrl._chooseAnim.gotoAndPlay("in_" + choice);
    }
}

//
// btnOK()
//
//  Confirms a battle selection
//
public function btnOK(battle:String):Void {
    _pathCtrl._chooseAnim.gotoAndPlay("out_" + battle);
}

//
```

```
// btnBackToBattle()
//
//  goes from the choose fiction screen back to the
//  select battle screen
//
public function btnBackToBattle():Void {
    _chooseFiction = false;
    _pathCtrl._chooseAnim.gotoAndPlay("in_battle");
}

//
// createBattle()
//
//  This is used by the battle game sequence. Once a battle
//  has chosen that battle is loaded from the corresponding
//  XML file and the players armies are registered with the
//  the main Chevalier game object
//
// PLAYER'S ARMIES NAMES SHOULD BE IN XML
public function createBattle(battle:String):Void {
    _xml.load("_" + battle + ".xml");

    switch (battle) {
        case "Arsuf":
            _game.playerOne.thePlayer = "Crusader";
            _game.playerTwo.thePlayer = "Saracen";
            _mapSpr.frame = 3;
            // _mapSpr.movieClip.gotoAndStop("_valley");
            break;
        case "Gaugamela":
            _game.playerOne.thePlayer = "Macedonian";
            _game.playerTwo.thePlayer = "Persian";
            _mapSpr.frame = 1;
            // _mapSpr.movieClip.gotoAndStop("_desert");
            break;
        case "Agincourt":
            _game.playerOne.thePlayer = "English";
            _game.playerTwo.thePlayer = "French";
            _mapSpr.frame = 2;
            // _mapSpr.movieClip.gotoAndStop("_coastal");
            break;
    }

    _pathCtrl._chooseAnim.gotoAndPlay("loadBattle");
    _game.freezeCursor("watch");
}

//
// btnChoose()
//
//  This is used by the "fictional" game sequence. Once a player
//  has chosen an army, all buttons are disabled so they they
//  can't be accidentally clicked and their army choice is
//  registered with the main Chevalier game object
//
public function btnChoose(choice:String):Void {
```

```
    // in chooing "fictional" mode
    _pathCtrl._chooseAnim._choose._CrusaderBtn.enabled   = false;
    _pathCtrl._chooseAnim._choose._SaracenBtn.enabled    = false;
    _pathCtrl._chooseAnim._choose._EnglishBtn.enabled    = false;
    _pathCtrl._chooseAnim._choose._FrenchBtn.enabled     = false;
    _pathCtrl._chooseAnim._choose._MacedonianBtn.enabled = false;
    _pathCtrl._chooseAnim._choose._PersianBtn.enabled    = false;
    _pathCtrl._chooseAnim.gotoAndPlay("out_fiction");

    if (_game.playerOne.thePlayer == undefined) {
        _game.playerOne.thePlayer = choice;
    } else {
        _game.playerTwo.thePlayer = choice;
    }

    _game.freezeCursor("watch");
}

//
// createPlayer()
//
//  This is used by the "fictional" game sequence to trigger the
//  XML loading of player one or player two's army. A randomized
//  terrain is also chosen here, and buttons disabled for armies
//  that have been selected by the other player.
//
public function createPlayer():Void {

    if (_game.playerTwo.thePlayer != undefined) {

        // create player two's army
        // this["create" +
        // _game.playerTwo.thePlayer](_game.playerTwo);
        _game.loadPlayerTwo = true;
        _xml.load("_" + _game.playerTwo.thePlayer + ".xml");

        switch (Utils.randomInt(1, 3)) {
            case 1:
                _mapSpr.frame = 0;
                _mapSpr.movieClip.gotoAndStop("_desert");
                break;
            case 2:
                _mapSpr.frame = 1;
                _mapSpr.movieClip.gotoAndStop("_valley");
                break;
            case 3:
                _mapSpr.frame = 2;
                _mapSpr.movieClip.gotoAndStop("_coastal");
                break;
        }

        _pathCtrl._chooseAnim.gotoAndPlay("loadBattle");
        _game.freezeCursor("watch");

    } else {

        _pathCtrl._chooseAnim._choose._CrusaderBtn.enabled = true;
```

```
            _pathCtrl._chooseAnim._choose._SaracenBtn.enabled = true;
            _pathCtrl._chooseAnim._choose._EnglishBtn.enabled = true;
            _pathCtrl._chooseAnim._choose._FrenchBtn.enabled = true;
            _pathCtrl._chooseAnim._choose._MacedonianBtn.enabled= true;
            _pathCtrl._chooseAnim._choose._PersianBtn.enabled = true;

            // create player one's army
            _xml.load("_" + _game.playerOne.thePlayer + ".xml");

            // disable the selected armies button
            _pathCtrl._chooseAnim._choose["_" +
                _game.playerOne.thePlayer + "Btn"].enabled = false;
            _pathCtrl._chooseAnim._choose["_" +
                _game.playerOne.thePlayer + "Btn"]._alpha  = 50;

        _pathCtrl._chooseAnim._choose._insignia.gotoAndPlay("None");
        _pathCtrl._chooseAnim._choose._chooseAnArmy.gotoAndPlay("None");
        _pathCtrl._chooseAnim._choose.gotoAndStop("PlayerTwo");
        _pathCtrl._chooseAnim.gotoAndPlay("in_fiction");

            // get game ready to create player two's army
         }

         _game.unfreezeCursor("arrow");
    }

    //
    // startGame()
    //
    //  Exits the choose state and commences the actual game
    //  This is called from the Flash score after both armies have been
    //  created from the XML files at the end of the "loadBattle"
    //  sequence.
    //
    public function startGame() {
        _game.state = "StartTurn";
        _game.unfreezeCursor("arrow");
    }
}
```

StartTurn.as

```
////////////////////////////////////////////////////////////////////////
//
//  StartTurn.as
//
//       Author: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: The is a very simple State where the game is put on
//               hold until the next player is ready to commence
//               their turn.  In this state a banner pronounces the
//               beginning of the next player's turn.
//
//       Method:
//
//       StartTurn() - Constructor
//           start() - Displays the "Start Turn" window for the player.
//
//                     The weather roll is made and all the players
//                     elements are reset. The map is flipped for the
//                     player.
//        btnBegin() - Triggered by "begin" button in start turn window
//                     This begins the movement phase for the player.
//
//  Notes:
//

class StartTurn implements IGameState {

    // instance members
    private var _game:Chevalier;                          // game object
    // path to the control movie (usually _root)
    private var _pathCtrl:MovieClip;

    //
    //  Constructor
    //
    //   args:
    //      game -- The Chevalier game object is needed
    //  pathCtrl -- path to the general control so display frame can be
    // changed
    //
    public function StartTurn(game:Chevalier, pathCtrl:MovieClip) {
        _game = game;
        _pathCtrl = pathCtrl;
    }
    public function update():Void       {}
    public function mouseDown():Void    {
        switch (Key.getCode()) {

            // enter key presses the "begin" button
            case Key.ENTER: btnBegin(); break;
        }
```

```
}
public function mouseUp():Void      {}
public function mouseMove():Void    {}
public function keyDown():Void      {}
public function keyUp():Void        {}
public function end():Void          {}

//
// start()
//
//  Displays the "Start Turn" window for the player
//  The weather roll is made and all the players elements are reset
//  The map is flipped for the player.
//
public function start():Void {

    _pathCtrl.gotoAndPlay("start");

    // roll the weather dice
    _game.weatherDice();

    // reset all ending players elements
    var allElements:Array = _game.playerActive.all;
    for (var i:Number = 0; i < allElements.length; ++i) {
        allElements[i].reset();
    }

    _game.switchActivePlayer();
    var player:Player = _game.playerActive;

    // reset all starting players elements
    var allElements:Array = _game.playerActive.all;
    for (var i:Number = 0; i < allElements.length; ++i) {
        allElements[i].reset();
    }

    // roll PIPs
    _game.playerActive.rollPIPs();

    // retieve map position and zoom to.
    if (! _global.gTesting) { _game.changeMap(90, player.mapScale,
             player.mapAngle, player.mapPos); }

    //
    // display start turn banner
    var distance:Number = 8;
    if (_game.playerTurn) {
        _pathCtrl.turn.gotoAndStop("playerOne");
        _game.playSnd("playerOne");
    } else {
        _pathCtrl.turn.gotoAndStop("playerTwo");
        _game.playSnd("playerTwo");
    }
    _pathCtrl.turn._fld_player.text = player.thePlayer;
    _pathCtrl.turn._fld_turn.text = "turn #" + _game.turnN;
    _pathCtrl.turn._insignia.gotoAndPlay(player.thePlayer);
}
```

```
    //
    // btnBegin()
    //
    //  Triggered by "begin" button in start turn window
    //  This begins the movement phase for the player
    //
    public function btnBegin():Void {
        _game.state = "Movement";
    }
}
```

Movement.as

```
////////////////////////////////////////////////////////////////////////
//
//  StartTurn.as
//
//       Author: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: The is a very simple State where the game is put on
//               hold until the next player is ready to commence
//               their turn.  In this state a banner pronounces the
//               beginning of the next player's turn.
//
//       Method:
//
//       StartTurn() - Constructor
//           start() - Displays the "Start Turn" window for the player.
//
//                     The weather roll is made and all the players
//                     elements are reset. The map is flipped for the
//                     player.
//        btnBegin() - Triggered by "begin" button in start turn window
//                     This begins the movement phase for the player.
//
//  Notes:
//

class StartTurn implements IGameState {

    // instance members
    private var _game:Chevalier;                          // game object
    // path to the control movie (usually _root)
    private var _pathCtrl:MovieClip;


    //
    //  Constructor
    //
    //   args:
    //      game -- The Chevalier game object is needed
    //  pathCtrl -- path to the general control so display frame can be
    // changed
    //
    public function StartTurn(game:Chevalier, pathCtrl:MovieClip) {
        _game = game;
        _pathCtrl = pathCtrl;
    }
    public function update():Void       {}
    public function mouseDown():Void    {
        switch (Key.getCode()) {

            // enter key presses the "begin" button
            case Key.ENTER: btnBegin(); break;
```

```
    }
}
public function mouseUp():Void      {}
public function mouseMove():Void    {}
public function keyDown():Void      {}
public function keyUp():Void        {}
public function end():Void          {}

//
// start()
//
//  Displays the "Start Turn" window for the player
//  The weather roll is made and all the players elements are reset
//  The map is flipped for the player.
//
public function start():Void {

    _pathCtrl.gotoAndPlay("start");

    // roll the weather dice
    _game.weatherDice();

    // reset all ending players elements
    var allElements:Array = _game.playerActive.all;
    for (var i:Number = 0; i < allElements.length; ++i) {
        allElements[i].reset();
    }

    _game.switchActivePlayer();
    var player:Player = _game.playerActive;

    // reset all starting players elements
    var allElements:Array = _game.playerActive.all;
    for (var i:Number = 0; i < allElements.length; ++i) {
        allElements[i].reset();
    }

    // roll PIPs
    _game.playerActive.rollPIPs();

    // retieve map position and zoom to.
    if (! _global.gTesting) { _game.changeMap(90, player.mapScale,
             player.mapAngle, player.mapPos); }

    //
    // display start turn banner
    var distance:Number = 8;
    if (_game.playerTurn) {
        _pathCtrl.turn.gotoAndStop("playerOne");
        _game.playSnd("playerOne");
    } else {
        _pathCtrl.turn.gotoAndStop("playerTwo");
        _game.playSnd("playerTwo");
    }
    _pathCtrl.turn._fld_player.text = player.thePlayer;
    _pathCtrl.turn._fld_turn.text = "turn #" + _game.turnN;
    _pathCtrl.turn._insignia.gotoAndPlay(player.thePlayer);
```

```
        }

        //
        // btnBegin()
        //
        //  Triggered by "begin" button in start turn window
        //  This begins the movement phase for the player
        //
        public function btnBegin():Void {
            _game.state = "Movement";
        }
    }
```

Shoots.as

```
////////////////////////////////////////////////////////////////////////
//
//  Shoots.as
//
//       Author: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: The Shoots state checks the map Grid Object to see if
//               any of the player's Elements are in range of the
//               enemy and cues them up in the array _shoots.  Once
//               this list is established the Shooting combat
//               interface is invoked and the player is allowed to
//               page through the possible battles and conduct them
//               via the "Fight!" button.  The actual battles are
//               resolved using the CombatTable Objects associated
//               with each of the Elements in the combat.
//
//       Method:
//
//          Shoots() - Constructor
//           start() - Establish "Shoots" state, namely, Generate a
//                     list of shoots based on trace lines of shooting
//                     and targets in range, and sort that list.
//drawShootsWindow() - display the shooting window showing
//                     battle _battlePage
//         btnNext() - flip forward a shoots page
//         btnPrev() - flip back a shoots page
//         btnDone() - Handle the "Done" button on the battle window
//           btnOK() - Handle the "OK" button on the battle window
//        btnFight() - Handle the "Fight!" button on the battle window
//      isDiagonal() - Tests if angle is a diagonal, needed when
//                     tracing lines of shooting
//         thetaOf() - Needed when tracing lines of shooting
//
//   Notes:
//

class Shoots implements IGameState {

    // instance members

    // game object
    private var _game:Chevalier;

    // path to the control movie (usually _root)
    private var _pathCtrl:MovieClip;

    // Grid object containing locations of pieces
    private var _grid:Grid;

    // list of battles for this turn
```

```
private var _shoots:Array;

// shooting # currently shown, 1 = _shoots[0], 2= _shoots[1], etc
private var _shootsPage:Number = 0;

//
//  Constructor
//
//   args:
//      game -- The Chevalier game object is needed
//      grid -- Grid object is needed to trace lines of shooting
//  pathCtrl -- path to the general control so display frame can be
//              changed
//
public function Shoots(game:Chevalier, grid:Grid,
         pathCtrl:MovieClip) {
    _game     = game;
    _grid     = grid;
    _pathCtrl = pathCtrl;
}

public function update():Void       {}
public function mouseDown():Void    {}
public function mouseUp():Void      {}
public function mouseMove():Void    {}
public function keyDown():Void      {}
public function keyUp():Void        {}
public function end():Void          {}

//
// start()
//
//  Establish "Shoots" state, namely,
//  Generate a list of shoots based on trace lines of shooting
//  and targets in range, and sort that list.
//
public function start():Void {

    // create list of battles
    _shoots = new Array();
    var eList:Array = _game.playerActive.all;

    // var eList:Array = Player.getEverybody();

    var lftPt:Point2D;
    var rhtPt:Point2D;
    var inc:Point2D;
    var angle:Number;
    var loc:Point2D;
    var range:Number;
    var eLft:Element;   var eRht:Element;
    var target:Element;
    for (var i:Number = 0; i < eList.length; ++i) {

        // engaged units cannot shoot
        if (eList[i].status == "Engaged") { continue; }
```

```
switch (eList[i].type) {
    case "Shot":
        range = 8;
        break;
    case "Archers":
    case "Crossbows":
        range = 24;
        break;
    case "Artillery":
        range = (eList[i].grade == "Inferior") ? 24 : 48;
        break;
    default:                range = 0;
}

// range - 8 if in HIGH WIND.

if (range > 0) {

    loc   = eList[i].grdLoc;
    angle = eList[i].angle;
    lftPt = (isDiagonal(angle) ? new Point2D(-1,
             -3) : new Point2D(-3, -1));
    lftPt.rotate(thetaOf(angle)); lftPt.add(loc);
    rhtPt = (isDiagonal(angle) ? new Point2D(+3,
             +1) : new Point2D(+3, -1));
    rhtPt.rotate(thetaOf(angle)); rhtPt.add(loc);
    inc   = (isDiagonal(angle) ? new Point2D(+1,
             -1) : new Point2D( 0, -1));
    inc.rotate(thetaOf(angle));

    // trace a line from the left and right edge to the
    // element most ahead
    eLft = undefined; eRht = undefined; target = undefined;
    for (var j:Number = 0; j < range; ++j) {
        eLft = _grid.getAt(lftPt);
        eRht = _grid.getAt(rhtPt);
        // unit found
        if (eLft instanceof Element ||
            eRht instanceof Element) { break; }

        lftPt.add(inc); rhtPt.add(inc);
    }

    // remove friendly targets and those engaged in combat
    if (eList[i].isFriendly(eLft) ||
        eLft.status == "Engaged") {
        eLft = undefined;
    }
    if (eList[i].isFriendly(eRht) ||
        eRht.status == "Engaged") {
        eRht = undefined;
    }

    // take the closer of two targets
    if (Utils.isACloser(eList[i].grdLoc, eLft.grdLoc,
        eRht.grdLoc)) {
        target = eLft;
```

```
            } else {
                target = eRht;
            }

            if (target != undefined) {

                // check list to see if targeted already
                var targeted:Boolean = false;
                var j:Number = 0;
                for (; j < _shoots.length; ++j) {
                    if(_shoots[j].target.spriteN ==target.spriteN){
                        targeted = true; break;
                    }
                }

                if ( ! targeted) {

                    // assign the shoot.
                    _shoots.push( { shooter: eList[i],
                                    target: target,
                                      xloc: target.grdLoc.x,
                                  supports: 0} );
                    eList[i].statusShoots();
                    target.statusShoots();

                } else {

                    // more ethan one shooter at target check
                    // if new shooter is better than currentshooter
                    if (eList[i].combatTbl.shootingTally(target) >
               _shoots[j].shooter.combatTbl.shootingTally(target)){
                        // switch the shooters!
                        _shoots[j].shooter.statusNormal();
                        _shoots[j].shooter = eList[i];
                        eList[i].statusShoots();
                    }

                    // incremenent supporting fire counter
                    _shoots[j].supports += 1;
                }
            }
        }
    }

    // sort battles according to whose turn it is
    if (_game.playerTurn) {
        // arrange "west to east" for black
        _shoots.sortOn("xloc", Array.NUMERIC);

    } else {
        // and "east to west" for white
        _shoots.sortOn("xloc", Array.NUMERIC | Array.DESCENDING);
    }

    // check if any unis being shot at
    if (_shoots.length > 0) {
        _shootsPage = 1;
```

```
        drawShootsWindow();

    } else {

        // no shooting, start the next turn
        _game.state = "Battles";
    }
}

//
// drawShootsWindow()
//
//  display the shooting window showing battle _battlePage
//
public function drawShootsWindow():Void {

    _pathCtrl.gotoAndStop("shooting");

    var player:Element = _shoots[_shootsPage - 1].shooter;
    var enemy:Element  = _shoots[_shootsPage - 1].target;
    player.combatTbl.displayOddsV(enemy, _pathCtrl.battle, true,
             _shoots[_shootsPage - 1].supports);

    _pathCtrl.battle._x_of_y.text = _shootsPage +
        " of " + _shoots.length;

    // if on the last battle disable forward and next buttons
    if (_shoots.length == 1) {
        _pathCtrl.battle._next._alpha  = 30;
        _pathCtrl.battle._next.enabled = false;
        _pathCtrl.battle._prev._alpha  = 30;
        _pathCtrl.battle._prev.enabled = false;
    } else {
        // enable the next and prev buttons
        _pathCtrl.battle._next._alpha  = 100;
        _pathCtrl.battle._next.enabled = true;
        _pathCtrl.battle._prev._alpha  = 100;
        _pathCtrl.battle._prev.enabled = true;
    }

    // hide the done and ok buttons
    _pathCtrl.battle._done._visible = false;
    _pathCtrl.battle._ok._visible = false;

    // show the fight button
    _pathCtrl.battle._fight._visible = true;

    // calculate mid position between both elements
    //  and deduce the new map position
    var battle_win_pos:Point2D = new Point2D(Stage.width/2,
             Stage.height/2 - 94);

    var loc:Point2D = player.scrnLoc.clone();
    loc.add(enemy.scrnLoc);
    loc.divide(2);
    var angle:Number = 0 - (player.angle - 270);
    loc.rotate(angle);
```

```
    var mpLoc:Point2D = battle_win_pos;

    // zoom the map
    var distance:Number = player.grdLoc.distance(enemy.grdLoc);
    var zoom:Number;
    if (distance <= 6)       { zoom = 100; }
    else if (distance <= 12) { zoom = 85; }
    else if (distance <= 18) { zoom = 70; }
    else if (distance <= 24) { zoom = 50; }
    else                     { zoom = 35; }

    mpLoc.subtract(loc.multiply(zoom/100));
    _game.changeMap(35, zoom, angle, mpLoc);
}

//
// btnNext()
//
//  flip forward a shoots page
//
public function btnNext():Void {
    if (_pathCtrl.battle._fight._visible == false) {
        btnOK();    // same as OK button if after a battle;
    } else {
        ++_shootsPage;
        if (_shootsPage > _shoots.length) { _shootsPage = 1; }
        drawShootsWindow();
    }
}

//
// btnPrev()
//
//  flip back a shoots page
//
public function btnPrev():Void {
    --_shootsPage;
    if (_shootsPage == 0) { _shootsPage = _shoots.length; }
    drawShootsWindow();
}

//
// btnDone()
//
//  Handle the "Done" button on the battle window
//
public function btnDone():Void {
    _game.state = "Battles";
}

//
// btnOK()
//
//  Handle the "OK" button on the battle window
//
public function btnOK():Void {
    drawShootsWindow();
```

```
}

//
// btnFight()
//
//  Handle the "Fight!" button on the battle window
//
public function btnFight():Void {

    // conduct the battle
    var player:Element = _shoots[_shootsPage - 1].shooter;
    var enemy:Element  = _shoots[_shootsPage - 1].target;
    player.combatTbl.conductBattleV(enemy, _pathCtrl.battle, true);

    // remove the battle from the battles list
    _shoots.splice(_shootsPage - 1, 1);

    // ensure _shootsPage still valid
    if (_shootsPage - 1 == _shoots.length) {
        --_shootsPage;
    }

    // hide the fight button
    _pathCtrl.battle._fight._visible = false;


    if (_shoots.length == 0) {
        // show the done button
        _pathCtrl.battle._done._visible = true;
    } else {
        // show the OK button
        _pathCtrl.battle._ok._visible = true;
    }
}

//
// isDiagonal()
//
//  Tests if angle is a diagonal,
//  needed when tracing lines of shooting
//
//   args:
//       angle -- angle in question
//
//      return -- true if angle is a diagonal
//
private static function isDiagonal(angle:Number):Boolean {
    return (angle%90 != 0);
}

//
// thetaOf()
//
//  Needed when tracing lines of shooting
//
//   args:
//       angle -- angle in question
```

```
    //
    //      return -- "theta" value dependant on angle
    //
    private static function thetaOf(angle:Number):Number {
        var theta:Number = angle + ( isDiagonal(angle) ? 45 : 90);
        theta = Utils.cleanAngle(theta);
        return theta;
    }
}
```

Battles.as

```
////////////////////////////////////////////////////////////////////////
//
//  Battles.as
//
//       Author: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: The Battles state checks all the player's Elements
//               to see if any are engaged in close combat and cues
//               them up in the array _battles.  There are some cases
//               where an Element that was moved by the player during
//               the Movement phase is still moving and has not
//               completed its movement into combat. In such
//               instances a _wait boolean flag is set and the
//               Battles state patiently waits for those Elements to
//               complete their move before checking for engaged
//               Elements.  Once the _battles list is established the
//               Battle combat interface is invoked and the player is
//               allowed to page through the possible battles and
//               conduct them via the "Fight!" button.  The
//               actual battles are resolved using the CombatTable
//               Objects associated with each of the Elements in the
//               combat.
//
//       Method:
//
//         Battles() - Constructor
//          update() - If _wait flag true then wait for all battles to
//                     be triggered before allowing this Battle state
//                     to proceed normally.
//           start() - Establish "Battles" state, namely, generate and
//                     sort a list of battles to be displayed. If there
//                     are still battles yet to be triggered due to
//                     movement then wait for them to trigger by use of
//                     a _wait flag.
//drawBattleWindow() - display battle window showing battle _battlePage
//         btnNext() - flip forward a battle page
//         btnPrev() - flip back a battle page
//         btnDone() - Handle the "Done" button on the battle window
//           btnOK() - Handle the "OK" button on the battle window
//        btnFight() - Handle the "Fight!" button on the battle window
//
//        Notes:
//

class Battles implements IGameState {

    // instance members

    // game object
    private var _game:Chevalier;
```

```
// path to the control movie (usually _root)
private var _pathCtrl:MovieClip;

// list of battles for this turn
private var _battles:Array;

// battle # currently shown, 1 = _battles[0], 2= _battles[1], etc
private var _battlePage:Number = 0;

// if true then must wait for units to engage.
private var _wait = false;

//
//  Constructor
//
//   args:
//      game -- The Chevalier game object is needed
//  pathCtrl -- path to the general control so display frame can
//              be changed
//
public function Battles(game:Chevalier, pathCtrl:MovieClip) {
    _game     = game;
    _pathCtrl = pathCtrl;
}

//
// update()
//
//  If _wait flag true then wait for all battles to be triggered
//  before allowing this Battle state to proceed normally
//
public function update():Void       {
    if(_wait) {
        if (! _game.aboutToEngage()) {
            start();
        }
    };
}
public function mouseDown():Void    {}
public function mouseUp():Void      {}
public function mouseMove():Void    {}
public function keyDown():Void      {}
public function keyUp():Void        {}
public function end():Void          {}

//
// start()
//
//  Establish "Battles" state, namely,
//  generate and sort a list of battles to be displayed.
//  If there are still battles yet to be triggered due to movement
//  then wait for them to trigger by use of a _wait flag
//
public function start():Void {

    _wait = false;
```

```
    // create list of battles
    _battles = new Array();

    var eList:Array = _game.playerActive.all;
    for (var i:Number = 0; i < eList.length; ++i) {
        if (eList[i].status == "Engaged") {
            _battles.push( { element: eList[i],
                     xloc: eList[i].grdLoc.x } );
        }
    }

    // sort battles according to whoes turn it is
    if (_game.playerTurn) {
        // arrange "west to east" for black
        _battles.sortOn("xloc", Array.NUMERIC);

    } else {
        // and "east to west" for white
        _battles.sortOn("xloc", Array.NUMERIC | Array.DESCENDING);

    }

    // check if units have finished moving into battle
    if (_game.aboutToEngage()) {
        _wait = true;
        _game.scaleMapTo(260, 50);
    } else if (_battles.length > 0) {
        _battlePage = 1;
        drawBattleWindow();

    } else {

        // no battles to conduct, start the next turn
        _game.state = "StartTurn";
    }
}

//
// drawBattleWindow()
//
//  display the battle window showing battle _battlePage
//
public function drawBattleWindow():Void {

    _pathCtrl.gotoAndStop("battle");

    var battle_win_pos:Point2D = new Point2D(Stage.width/2,
             Stage.height/2 - 94);

    var player:Element = _battles[_battlePage - 1].element;
    var enemy:Element = player.elementMostInFront();

    player.combatTbl.displayOddsV(enemy, _pathCtrl.battle);

    _pathCtrl.battle._x_of_y.text = _battlePage + " of "
                                    + _battles.length;
```

```
    // if on the last battle disable forward and next buttons
    if (_battles.length == 1) {
        _pathCtrl.battle._next._alpha  = 30;
        _pathCtrl.battle._next.enabled = false;
        _pathCtrl.battle._prev._alpha  = 30;
        _pathCtrl.battle._prev.enabled = false;
    } else {
        // enable the next and prev buttons
        _pathCtrl.battle._next._alpha  = 100;
        _pathCtrl.battle._next.enabled = true;
        _pathCtrl.battle._prev._alpha  = 100;
        _pathCtrl.battle._prev.enabled = true;
    }

    // hide the done and ok buttons
    _pathCtrl.battle._done._visible = false;
    _pathCtrl.battle._ok._visible = false;

    // show the fight button
    _pathCtrl.battle._fight._visible = true;

    // calculate mid position between both elements
    //  and deduce the new map position
    var loc:Point2D = player.scrnLoc.clone();
    loc.add(enemy.scrnLoc);
    loc.divide(2);
    var angle:Number = 0 - (player.angle - 270);
    loc.rotate(angle);
    var mpLoc:Point2D = battle_win_pos;
    mpLoc.subtract(loc);

    // zoom the map
    _game.changeMap(35, 100, angle, mpLoc);
}

//
// btnNext()
//
//  flip forward a battle page
//
public function btnNext():Void {
    if (_pathCtrl.battle._fight._visible == false) {

        // same as OK button if after a battle;
        btnOK();

    } else {

        ++_battlePage;
        if (_battlePage > _battles.length) { _battlePage = 1; }
        drawBattleWindow();
    }
}

//
// btnPrev()
//
```

```
//  flip back a battle page
//
public function btnPrev():Void {
    --_battlePage;
    if (_battlePage == 0) { _battlePage = _battles.length; }
    drawBattleWindow();
}

//
// btnDone()
//
//  Handle the "Done" button on the battle window
//
public function btnDone():Void {

    // start the next player turn
    _game.state = "StartTurn";
}

//
// btnOK()
//
//  Handle the "OK" button on the battle window
//
public function btnOK():Void {

    // display the next battle
    drawBattleWindow();
}

//
// btnFight()
//
//  Handle the "Fight!" button on the battle window
//
public function btnFight():Void {

    // conduct the battle
    var player:Element = _battles[_battlePage - 1].element;
    var enemy:Element = player.elementMostInFront();
    player.combatTbl.conductBattleV(enemy, _pathCtrl.battle);

    // remove the battle from the battles list
    _battles.splice(_battlePage - 1, 1);

    // ensure _battlePage still valid
    if (_battlePage - 1 == _battles.length) {
        --_battlePage;
    }

    // hide the fight button
    _pathCtrl.battle._fight._visible = false;

    if (_battles.length == 0) {
        // show the done button
        _pathCtrl.battle._done._visible = true;
    } else {
```

```
            // show the OK button
            _pathCtrl.battle._ok._visible = true;
        }
    }
}
```

Scroll.as

```
////////////////////////////////////////////////////////////////////////
//
//  Scroll.as
//
//       AUTHOR: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: The Scroll Object is a control object that is
//               specifically used to "google" and move single
//               Elements or Element groups according to specific
//               game rules. In particular, it keeps track of the
//               selected units, displaying information on them in an
//               "Information Scroll" Sprite that it also
//               presides over. When units are selected a Movement
//               Control will appear attached to the Information
//               Scroll and tests are performed for each possible
//               movement to see what moves are legal. There are many
//               issues when checking moves to see if the group will
//               need to "shift" and to snap into contact with
//               other groups or enemy Elements, these tests are
//               performed in the testMoveType(),
//               shiftPt_EnemyContact(), and
//               shiftPt_FriendlyContact() methods.
//
//       Method:
//
//          Scroll() - Constructor
// handleMouseDown() - Handle mouse being pressed, usually to select
//                     an Element on the map, or to drag the scroll
//   handleMouseUp() - Handle mouse release, namely, drop anything
//                     being dragged, including the scroll itself.
//     alphaOthers() - Darken all "other" elements, those not of the
//                     selected element's command
// handleMouseMove() - Handle the mouse moving over to a new
//                     grid location
//     tryToSelect() - Try to add an element to the selection, this
//                     is called by handleMouseMove and also by
//                     marqueeDraw() in the movement object
//        deselect() - Deselect all selected elements and close
//                     the scroll
//          update() - update the scroll, in particular update the
//                     movement tool to reflect the angle of the
//                     selected elements
// scrollConcluded() - Scroll has either finished opening, or finished
//                     closing. This is triggered by the animator
//                     object via the Chevalier object's collision()
//                     method once the scroll animation has finished.
//      openScroll() - The scroll is starting to animate open to
```

```
//                      display statistics for a rolled element
//      scrollOpen() - Calculate PIP (initiative points) cost to
//                      perform a move cmd
//updateScrollText() - Update the text on the scroll to reflect the
//                      rolled element or selected group
//       getStatus() - rtn the collective influence for the rolled
//                      element or selected group
//       getStatus() - rtn the collective status for the rolled
//                      element or selected group
//     closeScroll() - The scroll is starting to animate closed
//    scrollClosed() - The scroll animation has finished closing
//     calcPIPcost() - Calculate PIP (initiative points) cost to
//                      perform a move cmd
//     showPIPcost() - Display PIP (initiative points) cost to perform
//                      a move cmd
//    clearPIPcost() - clear PIP (initiative points) cost display
//   selectElement() - The user has selected the "rolled" element that
//                      the mouse is currently over.
//buildAdjacentList() - Build a list of all the elements adjacent to
//                      the currently selected Elements.  This is needed
//                      as these Elements are eligible to be added to
//                      the selected group of Elements.
// addAdjacentList() - When an Element is added to the list of selected
//                      Elements any elements adjacent to it must also
//                      be added to the adjacentList.
// drawMoveControl() - Draw the movement control from scratch. This
//                      requires testing each possible move type button
//                      with the selected Elements and either 1) storing
//                      the result if a legal move or 2) disabling the
//                      control if an illegal move.  As there are many
//                      potential moves, and every selected element has
//                      to be tested with each move type, drawing the
//                      control is a lengthy process. As such the move
//                      control is only updated once the user has
//                      stopped moving/clicking the mouse.
//    dynamic_ffwd() - Modify the forward command according to
//                      max movement and location of other Elements.
//    dynamic_rert() - Modify the retreat command according to
//                      max movement and location of other Elements.
//smplTestMoveType() - Greatly simplified version of testMoveType used
//                      by dynamic_ffwd() and dynamic_rert()
//  mvControlAngle() - Update the angle of the move control to reflect
//                      the angle of the selected elements being moved.
//                      This is called by both Scroll.update
//                      and drawMoveControl.
//    deducePivots() - Deduce pivot points from scratch, these are the
//                      edge points around which a group of selected
//                      Elements will wheel.
//    testMoveType() - Test if an element can perform a move type.
//                      return [false] if not, and [true, shift] if can,
//                      with 'shift' being a shift amount needed if
//                      there is contact with an enemy or friendly
//                      troops.
//shiftPt_EnemyContact() - Calculate shift pt for shifting to other
//                      enemy Elements
//shiftPt_FriendlyContact() - Calculate shift pt for shifting to other
//                      friendly Elements
```

```
// rotateShiftBack() - The "shift" point must be rotated back to "N"
//                     (270) with normal and "NE" (315) with diagonal
//                     moves for the shift to work with the MoveType
//                     definition.
//      isDiagonal() - Tests if angle is a diagonal
//      rollShadow() - Called by rollOver of the mv buttons.
//                     The shadows of the selected Elements are shown
//                     where the Elements will be if this command is
//                     selected, also, the PIP cost of the move is
//                     displayed.
//     resetShadow() - Called by rollOut of the mv buttons. The shadows
//                     of the selected Elements are reset and the PIP
//                     cost of the move is cleared.
//   dynamic_wheel() - Change the wheel (pivot) movement command for a
//                     specific Element according to to the size and
//                     direction of all (selected) elements being
//                     wheeled.
//        moveThem() - Move all selected Elements according to a
//                     specific movement command.
//
//  Notes:
//

class Scroll {

    // instance members

    // game object
    private static var _game:Chevalier;

    // generic animator
    private static var _a:Animatem;

    // keeps location of game pieces
    private static var _grid:Grid;

    // path to control movie (usually _root)
    private static var _pathCtrl:MovieClip;

    // sprite for the scroll
    private var _sprite:Sprite;

    // sprite number used for the scroll, it need to float over
    // elements so 500
    private var _scrollSprNum:Number = 700;

    // movie clip which is used for the scroll
    private var _scrollMC:MovieClip;

    // sprite used for the map
    private var _mapSpr:Sprite;

    // sprite used for the element picture
    private var _spritePct:Sprite;

    // depth of the element picture, one greater then the scroll
    private var _pctDepth:Number = 701;
```

```
// depth of the insigna graphic
private var _insignaDepth:Number = 702;

// depth of the movement control
private var _mvDepth:Number = 703;

// true if scroll being dragged
private var _scrollDrag:Boolean = false;

// true if the mouse is being held down
private var _mouseDown:Boolean  = false;

// time of the last mouse movement.  Used for updating the move
// control
private var _mouseMovedAt:Number;

// if true then the move control is due for an update
private var _mvCtrNeedsUpdate:Boolean;

// time waited for mouse to stop moving before drawing move control
private var _ctrDrawDelay:Number = 240;

// if true then the current group may be auto deselected for a new
// group
private var _canSelectNewGroup:Boolean;

// grid location of the last update/roll
private var _grdLoc:Point2D;

// element currently being displayed
private var _element:Element;

// list of elements currently selected for movement
private var _selected:Array;

// list of elements adjacent to selected elements
private var _adjacent:Array;

// -1 if selection not a group, else the max movement of the group
private var _grpMv:Number;

// denotes group of "Regular", "Clumsy" or "Regular & Clumsy"
private var _grpType:String;

// PIP cost to move this group
private var _pipCost:Number;

// true if group is all regulars
private var _regulars:Boolean;

// minimum PIPs paid by any element in the group
private var _PIPsPaid:Number;

// true if group is all light troops
private var _light:Boolean;
```

```
// valued use for calculating wheels
private var _lft_pivot:Point2D;

// value used for calculating wheels
private var _rht_pivot:Point2D;

// (0,0) point used to test shifts
private var _pt00:Point2D;

// types of availabile movement and corresponding normal and
// diagonal MoveType objects
private var _moveTypes:Array = ["_ffwd", "_fwd", "_dlft", "_drht",
         "_wlft", "_wrht", "_flip", "_tlft", "_trht", "_rert"];
private var _ffwd:MoveType;      private var _d_ffwd:MoveType;
private var _fwd:MoveType;       private var _d_fwd:MoveType;
private var _dlft:MoveType;      private var _d_dlft:MoveType;
private var _drht:MoveType;      private var _d_drht:MoveType;
private var _wlft:MoveType;      private var _d_wlft:MoveType;
private var _wrht:MoveType;      private var _d_wrht:MoveType;
private var _flip3:MoveType;     private var _d_flip3:MoveType;
private var _flip4:MoveType;     private var _d_flip4:MoveType;
private var _flip7:MoveType;     private var _d_flip7:MoveType;
private var _tlft3:MoveType;     private var _d_tlft3:MoveType;
private var _tlft4:MoveType;     private var _d_tlft4:MoveType;
private var _tlft7:MoveType;     private var _d_tlft7:MoveType;
private var _trht3:MoveType;     private var _d_trht3:MoveType;
private var _trht4:MoveType;     private var _d_trht4:MoveType;
private var _trht7:MoveType;     private var _d_trht7:MoveType;
private var _rert:MoveType;      private var _d_rert:MoveType;

//
//  Constructor
//
//   args:
// chevalier -- The Chevalier game object is needed
//         a -- The animatem animator object for moving Elements
//              around the map
//      grid -- Grid object that stores where Elements are on
//              the map
//    mapSpr -- sprite used my the map
//  pathCtrl -- path to general controls (usually _root)
//
public function Scroll(chevalier:Chevalier, a:Animatem, grid:Grid,
         mapSpr:Sprite, pathCtrl:MovieClip) {

    // assign game object
    _game     = chevalier;

    // assign generic animator
    _a        = a;

    // assign battlefield grid
    _grid     = grid;

    _mapSpr   = mapSpr;
    _pathCtrl = pathCtrl;
```

```
// (0,0) point used to test shifts
_pt00 = new Point2D(0,0);



// Scroll must float above element sprites
var ctrlPath:MovieClip = _game.pathCtrl;
_sprite  = _a.setSpriteN(_scrollSprNum, ctrlPath.scroll,
         ctrlPath, -1);
_sprite.frame = 1;  // set scroll to blank opening frame
_scrollMC      = _sprite.movieClip;

// initialize selection and adjacent lists
_selected = new Array();
_adjacent = new Array();

// modifiers for availabile movement types on a
// horizontal (270 degrees, facing "North") element
// dynamic
_ffwd  = new MoveType("_ffwd", new Point2D( 0, 0), 0,0,false);

_fwd   = new MoveType("_fwd", new Point2D( 0, -1), 1  , 0,
         false);
_dlft  = new MoveType("_dlft", new Point2D(-1, -1), 1.5, 0,
         false);
_drht  = new MoveType("_drht", new Point2D(+1, -1), 1.5, 0,
         false);
_wlft  = new MoveType("_wlft", new Point2D( -1, -2), 3  , -45,
         false);
_wrht  = new MoveType("_wrht", new Point2D( +1, -2), 3  , +45,
         false);

_flip3  = new MoveType("_flip", new Point2D( 0, +2), 2  , -180,
         false);
_flip4  = new MoveType("_flip", new Point2D( 0, +3), 3  , -180,
         false);
_flip7  = new MoveType("_flip", new Point2D( 0, +6), 6  , -180,
         false);

_tlft3  = new MoveType("_tlft", new Point2D(-3, -1), 3  , -90,
         false);
_tlft4  = new MoveType("_tlft", new Point2D(-3, 0), 3  , -90,
         false);
_tlft7  = new MoveType("_tlft", new Point2D(-3, +3), 3  , -90,
         false);
_trht3  = new MoveType("_trht", new Point2D(+3, -1), 3  , 90,
         false);
_trht4  = new MoveType("_trht", new Point2D(+3, 0), 3  , 90,
         false);
_trht7  = new MoveType("_trht", new Point2D(+3, +3), 3  , 90,
         false);

_rert   = new MoveType("_rert", new Point2D( 0, 1), 3  , 0,
         false);

// modifiers for availabile movement types on a
// diagonal (315 degrees, facing "North East") element,
```

```
    _d_ffwd = new MoveType("_ffwd", new Point2D(0, 0), 0, 0, true);
    _d_fwd  = new MoveType("_fwd", new Point2D(+1, -1), 1.5, 0,
              true);
    _d_dlft = new MoveType("_dlft", new Point2D( 0, -1), 1  , 0,
              true);
    _d_drht = new MoveType("_drht", new Point2D(+1,  0), 1  , 0,
              true);

    _d_wlft  = new MoveType("_wlft", new Point2D(+1, -2), 3, -45,
              true);
    _d_wrht  = new MoveType("_wrht", new Point2D(+2, -1), 3, +45,
              true);

    _d_flip3  = new MoveType("_flip", new Point2D(-1, +1), 2, -180,
              true);
    _d_flip4  = new MoveType("_flip", new Point2D(-2, +2), 3, -180,
              true);
    _d_flip7  = new MoveType("_flip", new Point2D(-4, +4), 6, -180,
              true);

    _d_tlft3  = new MoveType("_tlft", new Point2D(-1, -3), 3, -90,
              true);
    _d_tlft4  = new MoveType("_tlft", new Point2D(-2, -2), 3, -90,
              true);
    _d_tlft7  = new MoveType("_tlft", new Point2D(-4, 0),  3, -90,
              true);
    _d_trht3  = new MoveType("_trht", new Point2D(+3, +1), 3, 90,
              true);
    _d_trht4  = new MoveType("_trht", new Point2D(+2, +2), 3, 90,
              true);
    _d_trht7  = new MoveType("_trht", new Point2D( 0, +4), 3, 90,
              true);

    _d_rert   = new MoveType("_rert", new Point2D(-1, +1), 3.5, 0,
              true);
}

//
// handleMouseDown()
//
//  Handle mouse being pressed, usually to select an Element on the
//  map, or to drag the scroll
//
public function handleMouseDown(mseLoc:Point2D):Void {

    if ( sprite.movieClip.hitTest(mseLoc.x, mseLoc.y, true)) {
        // user clicked on the scroll

        // disable animator, allowing traditional dragging
        _sprite.active = 0;
        _sprite.movieClip.startDrag();
        _scrollDrag = true;

        // grab cursor

        // over the scroll
        if (_sprite.movieClip.mv.hitTest(_game.msePt.x,
```

```
                    _game.msePt.y, true) ) {
                // ..but not over the movement control
            } else {
                // otherwise use the hand grab cursor
                _game.freezeCursor("grab");
            }

        } else {

            // user clicked on map
            _mouseDown = true;

            var e = _grid.getAt(_grdLoc);
            if (e == undefined) {

                // user clicked on empty map
                deselect();
            }

            _canSelectNewGroup = true;
          handleMouseMove(_grid.ptToGridLoc(_game.cnvPtToMap(mseLoc)));
            _game.movementObj.marqueeStart();
        }
    }

    //
    // handleMouseUp()
    //
    //  Handle mouse release, namely, drop anything being dragged,
    //  including the scroll itself.
    //
    public function handleMouseUp():Void {

        if (_scrollDrag) {

            // stop dragging the scroll
            _sprite.movieClip.stopDrag();
            _sprite.setLocXY(_sprite.movieClip._x,
                     _sprite.movieClip._y);
            Player.active.scrollPos = new Point2D(_sprite.movieClip._x,
                     _sprite.movieClip._y);
            _scrollDrag = false;
            _sprite.active = -1;

            // over the scroll
            if (_sprite.movieClip.mv.hitTest(_game.msePt.x,
                     _game.msePt.y, true) ) {
                // ..but not over the movement control
                _game.unfreezeCursor("arrow");
            } else {
                // otherwise use the hand cursor
                _game.unfreezeCursor("hand");
            }

        }

        _mouseDown = false;
```

```
    // clear the marquee point
    _game.movementObj.marqueeClear();
}

//
// alphaOthers()
//
//  Darken all "other" elements, those not of the selected
// element's command
//
private function alphaOthers():Void {

    _game.movementObj.selectedCmd = _element.command;

    // default command center
    var otherOne:Array = Player.active.left;
    var otherTwo:Array = Player.active.right;

    if (_element.command == "Left") {
        var otherOne:Array = Player.active.center;
    } else if (_element.command == "Right") {
        var otherTwo:Array = Player.active.center;
    }

    // darken all elements in other commands
    for (var i:Number = 0; i < otherOne.length; ++i) {
        otherOne[i].alpha();
    }
    for (var i:Number = 0; i < otherTwo.length; ++i) {
        otherTwo[i].alpha();
    }
}

//
// handleMouseMove()
//
//  Handle the mouse moving over to a new grid location
//
//   args:
//    grdLoc -- The new grid location of the mouse
//
public function handleMouseMove(grdLoc:Point2D):Void {

    if ( sprite.movieClip.hitTest(_game.msePt.x, _game.msePt.y,
             true)) {

        // over the scroll
        if (_sprite.movieClip.mv.hitTest(_game.msePt.x,
                 _game.msePt.y, true) ) {
            // ..but not over the movement control
            _game.setCursor("arrow");
        } else {
            // otherwise use the hand cursor
            _game.setCursor("hand");
        }
```

```
} else {

    // if a new location then update _gridLoc
    if (grdLoc != undefined) {
        _grdLoc = grdLoc;
    }

    _mouseMovedAt = getTimer();

    // find what is being rolled
    var roll:Element = _grid.getAt(_grdLoc);

    if (_selected.length == 0) {

        // nothing selected yet,
        // handle roll over of elements

        if (roll == undefined) {

            // nothing rolled.  ensure scroll is clear
            closeScroll();
            _game.setCursor("arrow");

        } else if (roll instanceof Element) {

            // normal the previous rolled element (if any)
            if (_element.spriteN != roll.spriteN) {
                _element.stateNormal();
            }

            // roll display the new one
            openScroll(roll);
            _game.setCursor("google");

        }

    } else if (roll == undefined) {
        _game.setCursor("arrow");
    } else if (_game.isFriendly(roll)) {
        _game.setCursor("crosshair");
    }

    // if the mouse is held down
    //  then try to select whatever is being rolled
    if (_mouseDown && roll instanceof Element) {

        if (_game.isFriendly(_element)) {

            // OK, try to select element
            tryToSelect(roll);

        } else {

            // trying to select an enemy element
            _game.playSnd("denied");
        }
    }
```

```
            _canSelectNewGroup = false;
        }
    }

    //
    // tryToSelect()
    //
    //  Try to add an element to the selection,
    //  this is called by handleMouseMove and also by
    //  marqueeDraw() in the movement object
    //
    //   args:
    //      e -- Element that wants to be added to the selection
    //
    public function tryToSelect(e:Element):Void {

        // exit if element already selected
        //  of e not an element
        if (e.state == "Highlight") { return; }

        var select:Boolean = false;

        if (! select && _selected.length == 0 ) {

            // nothing selected yet, so allow selection
            select = true;
        }

        if (! select && e.command == _selected[0].command) {

            // if the element is in the same command as the first
            // selected element look through adjacent list for the
            // element
            for (var i:Number = 0; i < _adjacent.length; ++i) {

                // if adjacent then allow selection
                if ( _adjacent[i].spriteN == e.spriteN) {
                    select = true;
                    break;
                }
            }
        }

        // if not adjacent but we're ready to select a new group
        if ( ! select && _canSelectNewGroup) {

            // deselect selected elements
            deselect();

            // and allow selection
            select = true;
        }

        if (select) {

            // ensure scroll open for element
```

```
        openScroll(e);

        if (_selected.length == 0 ) {

            // if nothing selected yet then play the click sound
            // and
            //  alpha out the elements in other commands
            _game.playSnd("clk");
            alphaOthers();
        }

        // and add element to selection
        selectElement();
    }
}

//
// deselect()
//
//  Deselect all selected elements and close the scroll
//
public function deselect():Void {

    // tell movement object no command selected
    _game.movementObj.selectedCmd = undefined;

    _game.normalizeElements();
    // clear the selection array
    _selected = new Array();

    // clear the adjacent array
    _adjacent = new Array();

    // close the scroll
    closeScroll();

}

//
// update()
//
//  update the scroll, in particular update the movement tool
//  to reflect the angle of the selected elements
//
public function update():Void {

    // update the movement tool to reflect the angle of selected
    // element
    if (_mvCtrNeedsUpdate) {
        drawMoveControl();
    } else if (_selected.length > 0) {
        mvControlAngle();
    }
}

//
// scrollConcluded()
```

```
//
//  Scroll has either finished opening, or finished closing.
//  This is triggered by the animator object via the Chevalier
//  object's collision() method once the scroll animation has
// finished.
//
public function scrollConcluded():Void {
    if (_sprite.tag == "OPENING")      { scrollOpen(); }
    else if (_sprite.tag == "CLOSING") { scrollClosed(); }
}

//
// openScroll()
//
//  The scroll is starting to animate open to display statistics
// for
//  a rolled element
//
//   args:
//      e -- the Element being displayed by the scroll
//
public function openScroll(element:Element):Void {
    if (_element.spriteN != element.spriteN) {

        // designate new element for rollover display
        _element = element;
        _element.stateRollHL();


        // calc. scrolls opening position
         _sprite.loc = Player.active.scrollPos;


        // tell animator sprite to start opening the scroll
        _sprite.tag = "OPENING";
        _sprite.fPerSec = 20;
        _sprite.cycleType = "DEACTIVATE";
        _sprite.active = -1;

    }
}

//
// scrollOpen()
//
//  Calculate PIP (initiative points) cost to perform a move cmd
//
//   args:
//      cmd -- the movement command being queried
//
public function scrollOpen():Void {

    // ensure on end frame
    _sprite.frame = _sprite.cycleSize;

    _sprite.tag = "OPEN";
```

```
    updateScrollText();

    // keep the scroll active so it may be animated
    _sprite.fPerSec = 0;
    _sprite.frame   = _sprite.movieClip._totalframes;
    _sprite.active  = -1;
}

//
// updateScrollText()
//
//  Update the text on the scroll to reflect the rolled element
//  or selected group
//
public function updateScrollText():Void {

    // If selection is a mixed group then set _grpMv as max
    // movement
    //  of the group and determine _grpType
    _grpMv    = _element.movePts;
    _grpType  = (_element.regular ? "'Regular'" : "'Clumsy'");
    _regulars = _element.regular;
    _PIPsPaid = Number.MAX_VALUE;
    _light    = true;
    var mixed:Boolean      = false;
    var grade:String       = _element.grade;
    var type:String        = _element.type;

    for (var i:Number = 0; i < _selected.length; ++i) {

        // always find the minimum movement of the group
        if (_grpMv > _selected[i].movePts) {
            _grpMv = _selected[i].movePts;
        }

        // find total PIPs paid by the group
        if (_selected[i].PIPsPaid < _PIPsPaid) {
            _PIPsPaid = _selected[i].PIPsPaid;
        }

        // find if all regular
        if (! _selected[i].regular &&
            _selected[i].isGeneral == undefined) {
            _regulars = false;
        }

        // find if all light
        if (! (_selected[i].type == "Lt. Horse" ||
               _selected[i].type == "Lt. Infantry" ||
               _selected[i].type == "Skirmishers")) {
            _light = false;
        }

        // set mixed flag and _grpType
        if (mixed ||
            _selected[i].regular != _regulars ||
            _selected[i].grade   != grade ||
```

```
        _selected[i].type    != type) {

        mixed = true;

        if (_grpType == "'Regular'" && ! _selected[i].regular||
            _grpType == "'Clumsy'" && _selected[i].regular) {
            _grpType = "Regular+Clumsy";
        }
    }
}

_scrollMC._fld_terrain.text   = "Clear";
_scrollMC._fld_status.text    = getStatus();
_scrollMC._fld_influence.text = getInfluence();

if (! mixed) {

    _grpType = "Uniform";

    // fill information fields of scroll
    //  for a uniform group or single element
    _scrollMC._fld_element.text = _element.desc;
    _scrollMC._fld_class.text   = (_element.regular ?
        "Regular" : "Clumsy") + " " + _element.grade;
    _scrollMC._fld_type.text    = "'" + _element.type + "'";


    // display element picture
    var suffix:String;
    if (_element.player.thePlayer == Player.active.thePlayer) {
        // show movement remaining + right side picture
        _scrollMC._fld_movement.text = (_grpMv * 10) +" paces";
        suffix = "_rht";

    } else {
        // always show max move for enemy + left side picture
        _scrollMC._fld_movement.text =(_element.baseMve * 10) +
            " paces";
        suffix = "_lft";
    }

    // display element picture
    _sprite.movieClip._fig.gotoAndStop(_element.picture+suffix)

} else {

    // fill information fields of scroll
    //  for a mixed group
    _scrollMC._fld_element.text = "Group of";
    _scrollMC._fld_class.text   = _grpType;
    _scrollMC._fld_type.text    = _element.command + "Command";
    _scrollMC._fld_movement.text= (_grpMv * 10) + " paces";

    // display mixed group
    _sprite.movieClip._fig.gotoAndStop("mixed_group");
}
```

```
    // display the player insigna
_sprite.movieClip._insignia.gotoAndStop(_element.player.thePlayer);
}

//
// getStatus()
//
//  return - the collective influence for the rolled element or
//           selected group
//
public function getInfluence():String {

    var n:Number = 0;
    if (_selected.length > 0) {
        for (var i:Number = 0; i < _selected.length; ++i) {
            if (n < _selected[i].influence) {
                n = _selected[i].influence;
            }
        }
    } else {
        n = _element.influence;
    }

    switch (n) {
        case 0:         return "None";
        case 0.5:       return "Low";
        case 1:         return "Normal";
        case 2:         return "High";
        case 4:         return "Very High";
    }
}

//
// getStatus()
//
//  return - the collective status for the rolled element or
//           selected group
//
public function getStatus():String {

    var cmdStatus:String =
        _element.player.getCmdStatus(_element.command);
    if (cmdStatus == "Normal") {
        return _element.status;
    } else {
        if (cmdStatus == "Dispirited") {

            // determine the lowest influence in the group
            var n:Number = 4;
            if (_selected.length > 0) {
                for (var i:Number = 0; i < _selected.length; ++i) {
                    if (n > _selected[i].influence) {
                        n = _selected[i].influence;
                    }
                }
            } else {
                n = _element.influence;
```

```
            }

            if (n <= 1) {
                return "Dispirited";
            } else {
                return "Normal";
            }

        } else {
            return cmdStatus;
        }
    }
}

//
// closeScroll()
//
//  The scroll is starting to animate closed
//
public function closeScroll():Void {

    // remove the movement tool
    _sprite.movieClip.mv.removeMovieClip();

    // fix the PIP movement cost
    clearPIPcost();

    // return roll/selected element to normal
    _element.stateNormal();
    _element = undefined;

    // tell animator sprite to close the scroll
    _sprite.tag = "CLOSING";
    _sprite.fPerSec = -30;
    _sprite.cycleType = "DEACTIVATE";
    _sprite.active  = -1;
}

//
// scrollClosed()
//
//  The scroll animation has finished closing
//
public function scrollClosed():Void {
    _element = undefined;
    _sprite.tag = "CLOSED";

    // update with whatever is currently at the _grdLoc
    handleMouseMove();

    // keep the scroll active so it may be animated
    _sprite.fPerSec = 0;
    _sprite.frame   = 1;
    _sprite.active  = -1;
}
```

```
//
// calcPIPcost()
//
//  Calculate PIP (initiative points) cost to perform a move cmd
//
//   args:
//      cmd -- the movement command being queried
//
private function calcPIPcost(cmd:String):Number {
    var pipCost:Number = _light ? 0.5 : 1;

    // clumsy troops pay more for fancy moves
    if (! (cmd == "_ffwd" || cmd == "_fwd") && ! _regulars) {
        pipCost += _light ? 0.5 : 1;
    }

    // deduct pips paid so far
    pipCost -= _PIPsPaid;

    // clear negative cost
    if (pipCost < 0) { pipCost = 0; }

    return pipCost;
}

//
// showPIPcost()
//
//  Display PIP (initiative points) cost to perform a move cmd
//
//   args:
//      cmd -- the movement command being queried
//
private function showPIPcost(cmd:String) {

    _pipCost = calcPIPcost(cmd);

    if (_pipCost > 0) {
        // - _pipCost
       var pips:Number=_game.playerActive["PIPs"+_element.command];

        if (pips < 0) { pips = 0; }
        _pathCtrl.palette["_" + _element.command +
            "_PIPs"].gotoAndStop(pips + "_w");
        _sprite.movieClip.mv[cmd +
            "_cost"].gotoAndStop(_pipCost + "_cost");
        var n:Number = _selected[0].sprite.angle +
            _mapSpr.angle + 180;
        n = Utils.cleanAngle(n);
        _sprite.movieClip.mv[cmd + "_cost"]._rotation =
            - _sprite.movieClip.mv._rotation;
    }
}

//
// clearPIPcost()
//
```

```
//  clear PIP (initiative points) cost display
//
private function clearPIPcost() {
    var pips:Number = _game.playerActive["PIPs" +
        _element.command];
    if (pips < 0) { pips = 0; }
    _pathCtrl.palette["_" + _element.command +
        "_PIPs"].gotoAndStop(pips + "_b");

    // reset all cost rolls from mv controler
    for (var i:String in _moveTypes) {
        _sprite.movieClip.mv[_moveTypes[i] +
            "_cost"].gotoAndStop("none");
    }
}

//
// selectElement()
//
//  The user has selected the "rolled" element
//  that the mouse is currently over
//
public function selectElement():Void {

    // user has selected the rolled element,
    _selected.push(_element);
    _element.stateSelectHL();

    //addAdjacentList(_element);
    buildAdjacentList();
    drawMoveControl();

    // updateScroll text/info
    updateScrollText();
}

//
// buildAdjacentList()
//
//  Build a list of all the elements adjacent to
//  the currently selected Elements.  This is needed
//  as these Elements are eligible to be added to the
//  selected group of Elements
//
private function buildAdjacentList():Void {

    _adjacent = new Array();
    for (var j:Number = 0; j < _selected.length; ++j) {
        var list:Array = _selected[j].adjacent();

        for (var i:Number = 0; i < list.length; ++i) {
            _adjacent.push(list[i]);
        }
    }
}

//
```

```
// addAdjacentList()
//
//  When an Element is added to the list of selected
//  Elements any elements adjacent to it must also be
//  added to the adjacentList
//
//   args:
//      e -- Element being added to selection
//
private function addAdjacentList(e:Element):Void {

    // remove e from the _adjacent list
    for (var i:Number = 0; i < _adjacent.length; ++i) {
        if (_adjacent[i].spriteN == e.spriteN) {
            _adjacent.splice(i, 1);
            --i; // decrement i and keep iterating for more cases
        }
    }

    // add elements adjactent to e to _adjacent list
    var list:Array = e.adjacent();
    for (var i:Number = 0; i < list.length; ++i) {
        _adjacent.push(list[i]);
    }
}

//
// drawMoveControl()
//
//  Draw the movement control from scratch.
//  This requires testing each possible move type button
//  with the selected Elements and either 1) storing the
//  result if a legal move or 2) disabling the control
//  if an illegal move.  As there are many potential moves,
//  and every selected element has to be tested with each
//  move type, drawing the control is a lengthy process.
//  As such the move control is only updated once the user
//  has stopped moving/clicking the mouse.
//
public function drawMoveControl():Void {

    // only update after the mouse has stopped moving.
    if (getTimer() < _mouseMovedAt + _ctrDrawDelay && _mouseDown) {
        _mvCtrNeedsUpdate = true;
        return;
    }

    var price:Number = _light ? 1 : 2;
    var need:Number  = calcPIPcost("_rert");
    _sprite.movieClip.attachMovie("mv", "mv", _mvDepth);

    if (_regulars) {

        // regulars cost 1/2 as much to move
        price /= 2;

        if (need == price) {
```

```
            _sprite.movieClip.mv.gotoAndPlay("1_Regular");
        } else {
            _sprite.movieClip.mv.gotoAndPlay("0_Regular");
        }

    } else {

        if (need == price) {
            _sprite.movieClip.mv.gotoAndPlay("2_Clumsy");
        } else if (need == price/2) {
            _sprite.movieClip.mv.gotoAndPlay("1_Clumsy");
        } else {
            _sprite.movieClip.mv.gotoAndPlay("0_Clumsy");
        }
    }

    _sprite.movieClip.mv._x = 26;
    _sprite.movieClip.mv._y = 210;
    _sprite.movieClip.mv._xscale = 180;
    _sprite.movieClip.mv._yscale = 180;
    mvControlAngle();

    deducePivots();

    // get the movement PIPs for this command
    var PIPs:Number = _game.playerActive["PIPs" +
        _selected[0].command]

    // if PIPs exhausted and all selected are unmoved
    //  then show the "out" symbol for this commands movement,
    //   these guys aren't going anywhere
    if (PIPs == 0 && _PIPsPaid == 0) {
        _pathCtrl.palette["_" + _element.command +
            "_PIPs"].gotoAndStop("out");
    }

    for (var j:String in _moveTypes) {
        var btn:String = _moveTypes[j];
        var btnName:String = btn;

        // set button states
        var mvMC:MovieClip = _sprite.movieClip.mv;
        var enable:Boolean = true;
        var mv:MoveType; var origMv:MoveType; var dynMv:MoveType;

        // shift var needed to calculate contacting enemy
        var shift:Point2D;
        var friendlyShift:Boolean = false;

        // can player afford this move?
        var PIPcost:Number = calcPIPcost(_moveTypes[j]);

        if ( PIPcost >  PIPs ) {
            enable = false;
        } else {

            // OPTIMIZE? I.E. ENGAGED ELEMENTS CAN ONLY RETREAT
```

```
// special case setting of mv for _ffwd
if (btnName == "_ffwd") {
    dynMv = dynamic_ffwd();
} else if (btnName == "_rert") {
    dynMv = dynamic_rert();
}

// test all selected elements
for (var i:Number = 0; i < _selected.length; ++i) {

    // once element has retreated it may not
    //  do any other type of move
    if (_selected[i].lastMvMade == "_rert" &&
        _selected[i].withdrawn &&
        btnName != "_rert") {
        enable = false;
        break;
    }

    // elements can only nudge a certain number of
    // times
    var mxNudges = _selected[i].regular ? 4 : 2;
    if (_selected[i].nudges >= mxNudges &&
        (btnName == "_dlft" ||
         btnName == "_drht")) {
        enable = false;
        break;
    }

    // if engaged then can only retreat
    if (_selected[i].status == "Engaged") {
        if (btnName == "_rert" &&
            _selected[i].baseMve >
            _selected[i].engagingWith().baseMve &&
            _selected[i].leftFlanked() == undefined &&
            _selected[i].rightFlanked() == undefined) {

            // OK, let them retreat
        } else {
            enable = false;
            break;
        }
    }

    // modify _flip and turns according to depth
    if (btnName == "_flip" || btnName == "_tlft" ||
        btnName == "_trht") {
        btn = btnName + (_selected[i].baseDepth);
    }

    // get a fresh MoveType
    if (btnName == "_ffwd" || btnName == "_rert") {
        // special case for "_ffwd" and "_rert"
        mv = dynMv;
    } else {
        mv = this[(_selected[i].diagonal ? "_d" : "")
            + btn];
```

```
                if (btnName == "_wlft" || btnName == "_wrht") {
                    mv = dynamic_wheel(mv, _selected[i]);
                }
            }

            // shift movement to contact enemy or snap to
            // friends
            if (shift != undefined) { mv = mv.fixMove(shift,
                  0); }

            // NOTE: an undefined shift but with a true
            // friendlyShift denotes
            //  a friendly shift that was tried and failed.
            var r:Object = testMoveType(_selected[i], mv,
                  (shift == undefined && ! friendlyShift));

            if (! r.ok ) {

                // if a failing due to friendly shift then
                // negate it and start over,
                //  but keep the friendlyShift tag to denote a
                // failed friendlyShift
                if (friendlyShift) {
                    shift = undefined;
                    i = -1;
                } else {
                    enable = false; break;
                }

            } else {

                if (r.shift != undefined && !
                    r.shift.equal(_pt00)) {

                    if (shift == undefined ||
                        (shift != undefined &&
                         friendlyShift && ! r.friendlyShift)) {

                        // start loop over using the shift
                        i = -1;
                        shift = r.shift;

                        // give prority to non-friendly shifts
                        friendlyShift = r.friendlyShift;

                    } else {

                        // can't 'shift' twice
                        trace ("can't 'shift' twice.");
                        enable = false; break;
                    }
                }
            }
        }
    }

    if (enable) {
```

```
            // mvMC[btnName]._alpha  = 100;
            mvMC[btnName].enabled = true;
        } else {
            mvMC[btnName]._alpha  = 20;
            mvMC[btnName].enabled = false;
        }

    }

    // disable expand line lft and right
    mvMC["_exlft"]._alpha  = 20;
    mvMC["_exlft"].enabled = false;

    mvMC["_exrht"]._alpha  = 20;
    mvMC["_exrht"].enabled = false;

    _mvCtrNeedsUpdate = false;
}

//
// dynamic_ffwd()
//
//  Modify the forward command according to max movement
//  and location of other Elements
//
private function dynamic_ffwd():MoveType {

    var mv:MoveType = this[(_selected[0].diagonal ?
        "_d_ffwd" : "_ffwd")];
    var e:Element;

    // find limit due to move points
    var maxMv:Number  = 32;
    var theDepth:Number = 0;
    for (var i:Number = 0; i < _selected.length; ++i) {
        if (_selected[i]._mvePts < maxMv ) {
            maxMv = _selected[i]._mvePts;
        }
        if (_selected[i].depth > theDepth) {
            theDepth = _selected[i].depth;
        }
    }

    // check for passing through element directly in front
    var skippy:Number = 0;
    for (var i:Number = 0; i < _selected.length; ++i) {
        var tSkip:Number = 0;
        e = _selected[i].elementMostInFront();
        // calc depth to pass through
        while (e != undefined && e.state != "Highlight") {
            // no, can't move through
            if ( ! _selected[i].combatTbl.moveThrough(e)) {
                return mv.nonZero();
            }

            tSkip += e.depth;
            e = e.elementMostInFront();
```

```
        }
        if (tSkip) {
            // if there are units in front then add the depth of
            // mover(s)
            e = _selected[i];
            while (e != undefined && e.state == "Highlight") {
                tSkip += e.depth;
                e = e.elementBehind();
            }
        }
        if (tSkip > skippy) { skippy = tSkip; }
    }

    // passing through an element
    if (skippy) {


        // not enough movement to pass through
        if (skippy > maxMv) { return mv.nonZero(); }

        for (var i:Number = 0; i < skippy; ++i) {
            mv = mv.plusOne();
        }

    } else {

        // moving normally
        var nMv:MoveType = undefined;

        // inc by amount
        var incM = maxMv/(_selected[0].diagonal ? 1.5 : 1);
        incM = Math.round(incM);

        // limit max move to the max base depth
        if (incM > theDepth) { incM = theDepth; }

        var ok:String = "true";
        for (var i:Number = 0; i < incM; ++i) {

            // get potential move
            if (ok == "true") {
                nMv = mv.plusOne();
            } else if (ok == "nudge" && _selected[0].diagonal) {
                // ... so try nudging to the right
                nMv = mv.nudgeRight();

                ok = "nudged";
            } else {
                // if "exit" or "nudged" then break
                break;

            }

            // test for all units
            for (var j:Number = 0; j < _selected.length; ++j) {
                if (! smplTestMoveType(_selected[j], nMv)) {
                    if (ok == "true" &&
```

```
                        nMv.cost <= _selected[j].movePts) {
                        ok = "nudge"; --i;
                    } else {
                        ok = "exit";
                    }
                    break;
                }
            }
            if (ok == "true" || ok == "nudged") {
                mv = nMv;
            }
        }
    }

    // ensure a "zero" move is never passed
    return mv.nonZero();
}

//
// dynamic_rert()
//
//  Modify the retreat command according to max movement
//  and location of other Elements
//
private function dynamic_rert():MoveType {

    var mv:MoveType =
        this[(_selected[0].diagonal ? "_d_rert" : "_rert")];
    var e:Element;

    // find limit due to move points
    var maxMv:Number  = 32;
    var theDepth:Number = 0;
    for (var i:Number = 0; i < _selected.length; ++i) {
        if (_selected[i]._mvePts < maxMv ) {
            maxMv = _selected[i]._mvePts;
        }
        if (_selected[i].depth > theDepth) {
            theDepth = _selected[i].depth;
        }
    }
    --theDepth;

    // check for passing through element directly behind
    var skippy:Number = 0;
    for (var i:Number = 0; i < _selected.length; ++i) {
        var tSkip:Number = 0;
        e = _selected[i].elementMostBehind();
        // calc depth to pass through
        while (e != undefined && e.state != "Highlight") {
            // no, can't move through
            if ( ! _selected[i].combatTbl.moveThrough(e)) {
                return mv;
            }

            tSkip += e.depth;
            e = e.elementMostBehind();
```

```
            }
            if (tSkip) {
                // if there are units in front then add the depth of
                // mover(s)
                e = _selected[i];
                while (e != undefined && e.state == "Highlight") {
                    tSkip += e.depth;
                    e = e.elementInFront();
                }
            }
            if (tSkip > skippy) { skippy = tSkip; }
        }

        // passing through an element
        if (skippy) {


            // not enough movement to pass through
            if (skippy > maxMv) { return mv; }

            // decrement skippy to cover for 1st mv
            --skippy;

            for (var i:Number = 0; i < skippy; ++i) {
                mv = mv.minusOne();
            }

        } else {

            // moving normally
            var nMv:MoveType = undefined;

            // inc by amount
            var incM = maxMv/(_selected[0].diagonal ? 1.5 : 1);
            incM = Math.round(incM) - 1;

            // limit max move to the max base depth
            if (incM > theDepth) { incM = theDepth; }

            for (var i:Number = 0; i < incM; ++i) {

                // get potential move
                nMv = mv.minusOne();

                // test for all units
                for (var j:Number = 0; j < _selected.length; ++j) {
                    if (! smplTestMoveType(_selected[j],
                              nMv)) { return mv; }
                }
                mv = nMv;
            }
        }

        return mv;
    }

    //
```

```
// smplTestMoveType()
//
//  Greatly simplified version of testMoveType used
//  by dynamic_ffwd() and dynamic_rert()
//
private function smplTestMoveType(e:Element, mv:MoveType):Boolean {

    // check if it costs too much
    if (mv.cost > e.movePts) { return false; }

    // test the location
    var nGridLoc:Point2D = e.grdLoc;
    var mvMod:Point2D = mv.loc;
    mvMod.rotate(e.theta);
    nGridLoc.add(mvMod);
    var angle:Number = e.angle + mv.angle;

    if (! e.testLocation(nGridLoc, angle)) { return false; }

    // OK, can do.
    return true;
}

//
// mvControlAngle()
//
//  Update the angle of the move control to reflect the
//  angle of the selected elements being moved.
//  This is called by both Scroll.update and drawMoveControl
//
private function mvControlAngle():Void {
    var n:Number = _selected[0].sprite.angle + _mapSpr.angle;
    n = Utils.cleanAngle(n);
    _sprite.movieClip.mv._rotation = n;
}

//
// deducePivots()
//
//  Deduce pivot points from scratch, these are the edge points
//  around which a group of selected Elements will wheel.
//
private function deducePivots():Void {

    var angle:Number = _selected[0].angle;

    if (angle == 0 || (angle >= 225 && angle <= 315)) {
        _lft_pivot = new Point2D(+1000, +1000);
        _rht_pivot = new Point2D(-1000, -1000);
    } else if (angle == 180 || (angle >= 45 && angle <= 135)) {
        _lft_pivot = new Point2D(-1000, -1000);
        _rht_pivot = new Point2D(+1000, +1000);
    }

    var eInFront:Element;
    for (var i:Number = 0; i < _selected.length; ++i) {
```

```
            eInFront = _selected[i].elementInFront(true);
            if ( eInFront == undefined||eInFront.state!="Highlight") {

                if (angle == 0) {
                    // smallest Y
                    if (_lft_pivot.y > _selected[i].grdLoc.y) {
                        _lft_pivot = _selected[i].grdLoc
                    }

                    // largest Y
                    if (_rht_pivot.y < _selected[i].grdLoc.y) {
                        _rht_pivot = _selected[i].grdLoc
                    }


                } else if (angle >= 45 && angle <= 135) {
                    // largest X
                    if (_lft_pivot.x < _selected[i].grdLoc.x) {
                        _lft_pivot = _selected[i].grdLoc
                    }

                    // smallest X
                    if (_rht_pivot.x > _selected[i].grdLoc.x) {
                        _rht_pivot = _selected[i].grdLoc
                    }


                } else if (angle == 180) {
                    // largest Y
                    if (_lft_pivot.y < _selected[i].grdLoc.y) {
                        _lft_pivot = _selected[i].grdLoc
                    }

                    // smallest Y
                    if (_rht_pivot.y > _selected[i].grdLoc.y) {
                        _rht_pivot = _selected[i].grdLoc
                    }


                } else if (angle >= 225 && angle <= 315) {
                    // smallest X
                    if (_lft_pivot.x > _selected[i].grdLoc.x) {
                        _lft_pivot = _selected[i].grdLoc
                    }

                    // largest X
                    if (_rht_pivot.x < _selected[i].grdLoc.x) {
                        _rht_pivot = _selected[i].grdLoc
                    }
                }
            }
        }
    }

    //
    // testMoveType()
    //
```

```
//  Test if an element can perform a move type. return [false] if
// not,
//  and [true, shift] if can, with 'shift' being a shift amount
// needed
//  if there is contact with an enemy or friendly troops
//
private function testMoveType(e:Element, mv:MoveType,
         noShiftYet:Boolean):Object {

    // check if it costs too much
    if (mv.cost > e.movePts) { return { ok:false }; }

    // limit turns to single elements
    if ( (mv.name == "_tlft" || mv.name == "_trht" ) &&
        _selected.length > 1) { return { ok:false }; }

    // limit flip and rert to single elements or groups of lt.
    // horse or skirmishers
    if ( (mv.name == "_flip" || mv.name == "_rert") &&
        _selected.length > 1) {

        // only light horse and skirmishers can flip or rert in a
        // group
        var allow:Boolean = true;
        for (var i:Number = 0; i < _selected.length; ++i) {
            if ( ! (_selected[i].type == "Skirmishers" ||
                _selected[i].type == "Lt. Horse")) {
                allow = false;
                break;
            }
        }
        if (! allow) { return { ok:false }; }
    }

    // test the location
    var nGridLoc:Point2D = e.grdLoc;
    var mvMod:Point2D = mv.loc;
    mvMod.rotate(e.theta);
    nGridLoc.add(mvMod);
    var angle:Number = Utils.cleanAngle(e.angle + mv.angle);

    // ensure the basic position
    if (! e.testLocation(nGridLoc, angle)) { return { ok:false }; }

    // get testing points and vars ready for looking for shifts
    var friendlyShift:Boolean = false;
    var ctrPt:Point2D = (isDiagonal(angle) ? new Point2D(+1,
            -1) : new Point2D( 0, -1));
    ctrPt.rotate(thetaOf(angle));
    ctrPt.add(nGridLoc);
    var lftPt:Point2D = (isDiagonal(angle) ? new Point2D(-1,
            -3) : new Point2D(-3, -1));
    lftPt.rotate(thetaOf(angle));
    lftPt.add(nGridLoc);
    var rhtPt:Point2D = (isDiagonal(angle) ? new Point2D(+3,
            +1) : new Point2D(+3, -1));
    rhtPt.rotate(thetaOf(angle));
```

```
    rhtPt.add(nGridLoc);
    var shift:Point2D;

    // check for slide shifts due to contact with enemy
    // ...snap into contact with enemy element
    var shift:Point2D = shiftPt_EnemyContact(e,angle,ctrPt,ctrPt);

    if (shift == undefined) {
        // ...slide into contact with enemy element
        shift = Utils.smallerOfTwoPts(
            shiftPt_EnemyContact(e, angle, lftPt, ctrPt),
            shiftPt_EnemyContact(e, angle, rhtPt, ctrPt));
    }


    // if no enemy shifts and no existing shifts then check for
    // "extra" friendly shifts ...snap to the back of a
    // friendly element
    if (shift == undefined && noShiftYet) {
        friendlyShift = true;
        shift = shiftPt_FriendlyContact(e, angle, ctrPt, ctrPt,
                 nGridLoc, "center");
        // ...or snap to the left or right of a friendly element
        if (shift == undefined) {

            shift=Utils.smallerOfTwoPts(shiftPt_FriendlyContact(e,
                     angle, lftPt, ctrPt, nGridLoc, "left"),
                     shiftPt_FriendlyContact(e, angle, rhtPt,
                     ctrPt, nGridLoc, "right"));
        }
    }

    // OK, can do.
    e.storeMoveResult(mv.name, nGridLoc, angle, mv.cost);
    return { ok:true, shift:shift, friendlyShift:friendlyShift };
}

//
// shiftPt_EnemyContact()
//
//   Calculate shift pt for shifting to other enemy Elements
//
private function shiftPt_EnemyContact(e:Element, angle:Number,
         tstPt:Point2D, ctrPt:Point2D):Point2D {
    var shift:Point2D = undefined;

    // who's at test point?
    var other = _grid.getAt(tstPt);

    //  if an enemy
    if ( other instanceof Element && ! e.isFriendly(other)) {


        // ...facing opposite direction
        if (Utils.compareAngle(angle, other.angle - 180) == 0) {

            shift = other.grdLoc; shift.subtract(ctrPt);
```

```
        // ...facing same direction
        } else if (angle == other.angle) {

            shift = other.footPrint.backInside;
            shift.subtract(ctrPt);
        // ...facing left flank
        } else if (Utils.compareAngle(angle, other.angle + 90)==0){

            //  ... get front rank of enemy block
            other = other.getFrontRankElement();

            shift = other.footPrint.flankLeft;
            shift.subtract(ctrPt);
            shift.limit(isDiagonal(angle) ?  5 : 7);
        // ...facing right flank
        } else if (Utils.compareAngle(angle, other.angle-90)==0) {

            //  ... get front rank of enemy block
            other = other.getFrontRankElement();

            shift = other.footPrint.flankRht;
            shift.subtract(ctrPt);
            shift.limit(isDiagonal(angle) ?  5 : 7);
        }
    }

    return rotateShiftBack(e, shift);
}

//
// shiftPt_FriendlyContact()
//
//   Calculate shift pt for shifting to other friendly Elements
//
private function shiftPt_FriendlyContact(e:Element, angle:Number,
         tstPt:Point2D, ctrPt:Point2D, nGridLoc:Point2D,
         caseOf:String):Point2D {
    var shift:Point2D = undefined;

    // who's at test point?
    var other = _grid.getAt(tstPt);

    // if friendly and not highlighted
    if ( other instanceof Element && e.isFriendly(other) &&
         other.state != "Highlight") {

        // ... and facing the same direction
        if (angle == other.angle) {


            switch (caseOf) {
                case "center":
                    shift = other.footPrint.backInside;
                    break;
                case "left":
                    shift = other.footPrint.shiftRightPt;
                    break;
```

```
                case "right":
                    shift = other.footPrint.shiftLeftPt;
                    break;
            }
            shift.subtract(ctrPt);
        }
    }
    return rotateShiftBack(e, shift);
}

//
// rotateShiftBack()
//
//   The "shift" point must be rotated back to "N" (270) with
//   normal and "NE" (315) with diagonal moves for the shift to
//   work with the MoveType definition
//
//   args:
//       angle -- angle in question
//
//      return -- resultant "shift" point
//
private function rotateShiftBack(e:Element,
         shift:Point2D):Point2D {

    if (shift != undefined) {
        switch (e.angle) {
            case 0:     case 45:    shift.rotate(270);  break;
            case 90:    case 135:   shift.rotate(180);  break;
            case 180:   case 225:   shift.rotate(90);   break;
            case 270:   case 315:   shift.rotate(0);    break;
        }
        shift.round();
    }

    return shift;
}

//
// isDiagonal()
//
//  Tests if angle is a diagonal
//
//   args:
//       angle -- angle in question
//
//      return -- true if angle is a diagonal
//
// MOVE TO UTILS?
private static function isDiagonal(angle:Number):Boolean {
    return (angle%90 != 0);
}

//
// thetaOf()
//
//   args:
```

```
//       angle -- angle in question
//
//      return -- "theta" value dependant on angle
//
private static function thetaOf(angle:Number):Number {
    var theta:Number = angle + ( isDiagonal(angle) ? 45 : 90);
    theta = Utils.cleanAngle(theta);
    return theta;
}

//
// rollShadow()
//
//  Called by rollOver of the mv buttons.
//  The shadows of the selected Elements are shown
//  where the Elements will be if this command is selected,
//  also, the PIP cost of the move is displayed.
//
public function rollShadow(cmd:String):Void {

    var cmdName:String = cmd;

    for (var i:Number = 0; i < _selected.length; ++i) {

        // display the shadow element
        var mvData:Array = _selected[i].getMoveResult(cmd);
        _selected[i].shadowAt(mvData[0], mvData[1]);
    }

    // show cost    MUST FIX THIS WHEN WHEELING...
    _scrollMC._fld_movement.text = (_grpMv * 10) +
                    " (-" + mvData[2] * 10 + ")";

    showPIPcost(cmd);

}

//
// resetShadow()
//
//  Called by rollOut of the mv buttons.
//  The shadows of the selected Elements are reset
//  and the PIP cost of the move is cleared.
//
public function resetShadow():Void {

    for (var i:Number = 0; i < _selected.length; ++i) {
        _selected[i].resetShadow();
    }

    // fix movement cost field
    _scrollMC._fld_movement.text = (_element.movePts*10) +"paces";

    clearPIPcost();

}
```

```
//
// dynamic_wheel()
//
//  Change the wheel (pivot) movement command for a specific
// Element
//  according to to the size and direction of all (selected)
// elements
//  being wheeled
//
//   args:
//       mv -- The base move command
//        e -- The specific Element wheeling
//
//   return -- modified move type specific to e
//
private function dynamic_wheel(mv:MoveType, e:Element):MoveType {

    var angle:Number = e.angle;
    var pivot:Number;
    var inc:Number;
    var locMod:Point2D;
    var bkMod2:Point2D;
    var bkMod3:Point2D;
    var bkMod5:Point2D;

    var bkMod:Point2D = new Point2D(0, 0);

    if (mv.name == "_wlft") {
        if (angle == 0 || angle == 180) {
            pivot  = _lft_pivot.y;
        } else {
            pivot  = _lft_pivot.x;
        }

        if (e.diagonal) {
            inc = 5;
            locMod = new Point2D(+2, -5);

            bkMod2  = new Point2D(+2, +1);
            bkMod3  = new Point2D(+3, +1);
            bkMod5  = new Point2D(+5, +2);

        } else {
            inc = 7;
            locMod = new Point2D(-2, -5);

            bkMod2  = new Point2D(+2, -1);
            bkMod3  = new Point2D(+3, -1);
            bkMod5  = new Point2D(+5, -2);

        }

    } else if (mv.name == "_wrht") {
        if (angle == 0 || angle == 180) {
            pivot  = _rht_pivot.y;
        } else {
```

```
            pivot  = _rht_pivot.x;
        }

        if (e.diagonal) {
            inc = 5;
            locMod = new Point2D(+5, -2);

            bkMod2  = new Point2D(-1, -2);
            bkMod3  = new Point2D(-1, -3);
            bkMod5  = new Point2D(-2, -5);
        } else {
            inc = 7;
            locMod = new Point2D(+2, -5);

            bkMod2  = new Point2D(-2, -1);
            bkMod3  = new Point2D(-3, -1);
            bkMod5  = new Point2D(-5, -2);
        }
    }

    var val:Number;
    var eInFront:Element = e;
    while (eInFront != undefined && eInFront.state== "Highlight") {

        // if not the moving element, "stagger" bkMod according to
        // depth
        if (eInFront.spriteN != e.spriteN) {
            switch (eInFront.diagonalDepth) {
                case 2:  bkMod.add(bkMod2); break;
                case 3:  bkMod.add(bkMod3); break;
                case 5:  bkMod.add(bkMod5); break;
            }
        }

        if (angle == 0 || angle == 180) {
            val = eInFront.grdLoc.y;
        } else {
            val = eInFront.grdLoc.x;
        }
        eInFront = eInFront.elementInFront(true);
    }

    var index:Number = Math.round(Math.abs((val - pivot))/inc);

    locMod.multiply(index);
    locMod.add(bkMod);
    var costMod:Number = 3 * index;

    return mv.fixMove(locMod, costMod);
}

//
// moveThem()
//
//  Move all selected Elements according to a
//  specific movement command
//
```

```
//   args:
//      cmd -- name of movement command
//
public function moveThem(cmd:String):Void {

    _game.playSnd("clk");

    var mvData:Array;

    // some moves, such as wheeling, will cost the whole
    //  group the cost as the furthest moved unit.
    //   so find the most costly move.
    var mvCost:Number = 0;
    for (var i:Number = 0; i < _selected.length; ++i) {
        mvData = _selected[i].getMoveResult(cmd);
        if (mvData[2] > mvCost) { mvCost = mvData[2]; }
    }

    // remove all data for all selected elements from grid
    //  as otherwise elements moved individually
    //   can potentially erase each other
    for (var i:Number = 0; i < _selected.length; ++i) {

        // if engaged then disengage (must be retreating)
        if (_selected[i].status == "Engaged") {
            _selected[i].withdrawn = true;
            _selected[i].elementMostInFront().disengage();
            _selected[i].disengage();
        }

        // if nudging an element then increment its nudge counter
        if (cmd == "_dlft" ||
            cmd == "_drht") {
            _selected[i].nudges += 1;
        }

        // remove data
        _selected[i].removeData();
    }

    // set the new element locations
    for (var i:Number = 0; i < _selected.length; ++i) {

        mvData = _selected[i].getMoveResult(cmd);

        // subtract movement cost and move element
        _selected[i].movePts -= mvCost;

        // ensure no negative values (can happen with wheels)
        if (_selected[i].movePts < 0) { _selected[i].movePts = 0; }

        // set element at new location
        _selected[i].setLoc(mvData[0], mvData[1]);

        // tag elements last move made
        _selected[i].lastMvMade = cmd;
```

```
            // save PIPs paid to move
            _selected[i]._PIPsPaid += _pipCost;

        }

        // pay the PIP price
        _game.playerActive["PIPs" + _selected[0].command] -= _pipCost;

        // update the scroll info
        updateScrollText();

        // draw the movement control
        drawMoveControl();

        // if button still active then update the roll shadow
        if (_sprite.movieClip.mv[cmd].enabled) { rollShadow(cmd); }

        // recreate the adjacent list
        buildAdjacentList();

        // animate elements(s) to their new position
        for (var i:Number = 0; i < _selected.length; ++i) {
            _selected[i].advance();
        }

        clearPIPcost();
    }


    // accessors
    public function get sprite():Sprite { return _sprite; }

    // mutators
}
```

CombatTable.as

```
////////////////////////////////////////////////////////////////////////
//
//  CombatTable.as
//
//       AUTHOR: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: The Player Object creates a CombatTable Object for
//               every Element that it creates and passes this
//               reference to the Element on creation.  CombatTable
//               is a collection of methods specifically to conduct
//               various rule intricacies for combat, such as combat
//               factors, shooting factors, grading factors, battle
//               result strings, and other intricacies such as what
//               Element Types will pursue after battle and which
//               Types can move through which when moving and
//               recoiling.
//
//       Method:
//
//     CombatTable() - Constructor
//        lessThan() - rtn result when score less then enemy of type
//       doubledBy() - rtn result when score doubled by enemy of a type
//      convResult() - Convert battle sybols to String results
//         factors() - Tally Support & Tactical Factors for Element
// shootingFactors() - Tally Shooting factors for this Element
//   shootingTally() - Rough tally of this elements shooting
//                     effectiveness against a specific enemy.  Use by
//                     the Shoots object to determine who should be the
//                     primary shooter at a target
//         grading() - Determine factors dictated by the grading of
//                     both Elements in combat. These Grading Factors
//                     are add AFTER the dice for the battle have been
//                     rolled
//    displayOddsV() - display odds for battle against another element
//  conductBattleV() - conduct battle (roll dice) against another e
//          pursue() - rtn true if Element e will pursue this Element
//     moveThrough() - rtn true if Element e can pass through this El
//
//        Notes:
//

class CombatTable {

    //
    // instance members
    // game object      MAKE STATIC?
    private var _game:Chevalier;

    // element this table belongs to
    private var _element:Element;
```

```
// element type: "Kn", "Cv", "LH", "Sp", etc
private var _type:String;

private var _grade:String;          // element grade
// true if Skirmishers who can support spears, blades or cavalry
private var _supporters:Boolean;

// true if  this Element is mounted
private var _mtd:Boolean = false;

// fighting value against mounted
private var _FVvMtd:Number;

private var _FVvFt:Number;          // fighting value against foot

// Results when combat result is less then enemy v types
//  "K" Killed, "E" Killed if enemy turn, "R" Regroup, "F" Flee
private var _El:String;     private var _db_El:String;
private var _Exp:String;    private var _db_Exp:String;
private var _Kn:String;     private var _db_Kn:String;
private var _Cv:String;     private var _db_Cv:String;
private var _LH:String;     private var _db_LH:String;
private var _Sp:String;     private var _db_Sp:String;
private var _Pk:String;     private var _db_Pk:String;
private var _Sw:String;     private var _db_Sw:String;
private var _Bw:String;     private var _db_Bw:String;
private var _Cb:String;     private var _db_Cb:String;
private var _Ax:String;     private var _db_Ax:String;
private var _AxS:String;    private var _db_AxS:String;
private var _Sk:String;     private var _db_Sk:String;
private var _Art:String;    private var _db_Art:String;
private var _Hd:String;     private var _db_Hd:String;
private var _Bg:String;     private var _db_Bg:String;
private var _Wb:String;     private var _db_Wb:String;
// player's Fighting Value (FV) in displayed battle
private var _atkr_fv:Number;

// enemy's Fighting Value (FV) in displayed battle
private var _dfdr_fv:Number;

//
//  Constructor
//
public function CombatTable(game:Chevalier,
                            type:String,
                            grade:String,
                            supporters:Boolean,
                            fvVMtd:Number,
                            fvVFt:Number,
                            el:String,
                            exp:String,
                            kn:String,
                            cv:String,
                            lH:String,
                            sp:String,
                            pk:String,
                            sw:String,
```

```
                            bw:String,
                            cb:String,
                            ax:String,
                            axS:String,
                            sk:String,
                            art:String,
                            hd:String,
                            bg:String,
                            wb:String,
                            dbEl:String,
                            dbExp:String,
                            dbKn:String,
                            dbCv:String,
                            dbLH:String,
                            dbSp:String,
                            dbPk:String,
                            dbSw:String,
                            dbBw:String,
                            dbCb:String,
                            dbAx:String,
                            dbAxS:String,
                            dbSk:String,
                            dbArt:String,
                            dbHd:String,
                            dbBg:String,
                            dbWb:String) {

_game   = game;

_type   = type;
_grade  = grade;
_supporters = supporters;
_FVvMtd = fvVMtd;
_FVvFt  = fvVFt;

_El     = el;   _db_El  = dbEl;
_Exp    = exp;  _db_Exp = dbExp;
_Kn     = kn;   _db_Kn  = dbKn;
_Cv     = cv;   _db_Cv  = dbCv;
_LH     = lH;   _db_LH  = dbLH;
_Sp     = sp;   _db_Sp  = dbSp;
_Pk     = pk;   _db_Pk  = dbPk;
_Sw     = sw;   _db_Sw  = dbSw;
_Bw     = bw;   _db_Bw  = dbBw;
_Cb     = cb;   _db_Cb  = dbCb;
_Ax     = ax;   _db_Ax  = dbAx;
_AxS    = axS;  _db_AxS = dbAxS;
_Sk     = sk;   _db_Sk  = dbSk;
_Art    = art;  _db_Art = dbArt;
_Hd     = hd;   _db_Hd  = dbHd;
_Bg     = bg;   _db_Bg  = dbBg;
_Wb     = wb;   _db_Wb  = dbWb;

// special case for superior Auxilia
if (_type == "Ax" && _grade == "Superior") {
    _type = "AxS";
}
```

```
    // set mounted flag
    if (_type == "El" ||
        _type == "Exp" ||
        _type == "Kn" ||
        _type == "Cv" ||
        _type == "LH") {
        _mtd = true;
    }
}

//
// lessThan()
//
//  return result when score less then enemy of type
//
public function lessThan(eType:String, myTurn:Boolean,
         terrain:String, shooting:Boolean):String {
    return convResult(this["_" + eType], eType, myTurn, terrain,
             shooting);
}

//
// doubledBy()
//
//  return result when score doubled by enemy of a type
//
public function doubledBy(eType:String, myTurn:Boolean,
         terrain:String, shooting:Boolean):String {
    return convResult(this["_db_" + eType], eType, myTurn, terrain,
             shooting);
}

//
// convResult()
//
//  Convert battle sybols to String results
//
//   args:
//              str  -- combat result symbol
//            eType  -- type of enemy
//           myTurn  -- true if this Elements turn
//          terrain  -- type of terrain battle is in
//         shooting  -- true if a distant shooting combat
//
//   return         -- result of combat, i.e. "Killed", "Recoil"
//
private function convResult(str:String, eType:String,
         myTurn:Boolean, terrain:String, shooting:Boolean):String {

    // shooters can't be killed by non shooters
    if (shooting && ! (eType == "Bw" || eType == "Cb" ||
                       eType == "Sh" || eType == "Art")) {
        return "Stand";
    }

    // Hills count as clear for these results
```

```
if (terrain == "Hill") {
    terrain = "Clear";
}

// if element under flank attack then always a Killed result
if (_element.leftFlanked() instanceof Element ||
    _element.rightFlanked() instanceof Element) {
    return "Killed";
}

// if retreat blocked then always a Killed result
var rearRank:Element  = _element.getRearRankElement();
var elmtsBehind:Array = rearRank.elementsBehind();
for (var i:Number = 0; i < elmtsBehind.length; ++i) {
    if ( _element.isFriendly(elmtsBehind[i])) {
        if (_element.angle != elmtsBehind[i].angle) {
            return "Killed";
        }
    } else  { return "Killed"; }
}

switch (str) {

    // K = Killed
    // K*= Killed if in close combat
    // k = Killed if in clear terrain, otherwise recoil
    case "K": case "K*": case "k":
        if (str == "K*" && shooting) { return "Recoil"; }
        if (str == "k" && terrain != "Clear") {
            return "Recoil";
        }
        return "Killed";

    // E = Killed if enemy's turn
    // e = Killed if enemy's turn and in clear terrain,
    // otherwise recoil
    case "E": case "e":
        if (myTurn) {
            return "Recoil";
        } else {
            if (str == "e" && terrain != "Clear") {
                return "Recoil";
            }
            return "Killed";
        }

    // S = Spent
    // S*= Spent if in close combat
    // s = Spent if in difficult terrain, otherwise killed
    case "S": case "S*": case "s":
        if (str == "S*" && shooting) { return "Recoil"; }
        if (str == "s" && terrain == "Clear") {
            return "Killed";
        }
        return "Spent";

    // F = Flee
```

```
        // F*= Flee if in close combat
        // f = Flee if in difficult terrain, otherwise killed
        case "F": case "F*": case "f":
            if (str == "F*" && shooting) { return "Recoil"; }
            if (str == "f" && terrain == "Clear") {
                return "Killed";
            }
            return "Flee";

        // O = Repulsed if own turn
        // o = Repulsed if own turn and in clear terrain
        case "O": case "o":
            if (myTurn) {
                if (str == "o" && terrain != "Clear") {
                    return "Killed";
                }
                // was "Repulsed"--DBMM SAYS REPULSED
                return "Flee";

            } else {
                return "Killed";
            }

        // R = Repulsed
        // R*= Repulsed
        // r = Repulsed if in clear terrain, otherwise recoil
        case "R": case "R*": case "r":
            if (str == "R*" && shooting) { return "Recoil"; }
            if (str == "r" && terrain != "Clear") {
                return "Recoil";
            }
            // was "Repulsed"--DBMM SAYS REPULSED
            return "Flee";

        case "-": return "Recoil";
    }

    // should never get to here
    throw new Error("Can't understand combat result: " + str);
}

//
// factors()
//
//  Tally Support & Tactical Factors for this Element
//
//   args:
//           enemy  -- enemy Element
// shootingSupport  -- # of other shooters supporting this element
//         terrain  -- type of terrain battle is in
//        shooting  -- true if a distant shooting combat
//
//
private function factors(enemy:Element, myTurn:Boolean,
         terrain:String, shooting:Boolean):Array {

    var s:Number = 0;           // support modifier
```

```
        var str:String = "";        // string with explanations

        // get tables for all possible supporting elements
        // enemy 1st rank
        var other:CombatTable = enemy.combatTbl;

        // enemy 2nd rank
        var oOther:CombatTable = enemy.elementBehind(true).combatTbl;

        // enemy 3rd rank
        var ooOther:CombatTable =
            enemy.elementBehind(true).elementBehind(true).combatTbl;

        // enemy 4th rank
        var oooOther:CombatTable =
enemy.elementBehind(true).elementBehind(true).elementBehind(true).comba
tTbl;

        // supporting 2nd rank
        var support:CombatTable =
            _element.elementBehind(true).combatTbl;

        // supporting 3rd rank
        var sSupport:CombatTable =
            _element.elementBehind(true).elementBehind(true).combatTbl;


        // enemy overlaping on left
        var eLft:Element = _element.leftOverlap();

        // enemy overlaping on right
        var eRht:Element = _element.rightOverlap();


        var eLeftFlank:Element = _element.leftFlanked();
        var eRightFlank:Element = _element.rightFlanked();

        //////////
        //
        // tactical factors

        var eMtd:Boolean = enemy.combatTbl.mtd;

        // +1 if general
        if (_element.isGeneral == "Sub-gen" ||
            _element.isGeneral == "C-in-C") {
            str += "+1 commander\r";
            ++s;
        }

        // -1 if Disheartend or Broken, -2 if Shattered
        switch(_element.player.getCmdStatus(_element.command)) {
            case "Dispirited":
                if (_element.influence <= 1) {
                    str += "- 1 disheartend troops\r";
                    --s;
                }
```

```
            break;
        case "Broken":
            str += "- 1 broken troops\r";
            --s;
            break;
        case "Shattered":
            str += "- 2 shattered troops\r";
            --s; --s;
            break;
    }

    // +1 if opponent is mounted troops overlapped by Elephants.
    var eOlp:Element = enemy.leftOverlap();
    if (eMtd && eOlp.combatTbl.type == "El" &&
                eOlp.status != "Engaged") {
        str += "+1 Elephants aiding v mounted enemy\r";
        ++s;
    }
    eOlp = enemy.rightOverlap();
    if (eMtd && eOlp.combatTbl.type == "El" &&
                eOlp.status != "Engaged") {
        str += "+1 Elephants aiding v mounted enemy\r";
        ++s;
    }

    // - 1 for left flank attacked
    if (eLeftFlank instanceof Element) {
        str +="- 1 left flank attacked-cannot recoil or flee!\r";
        --s;
    } else {

        // - 1 for left flank overlapped
        var ignore:Boolean;
        if (eLft instanceof Element) {

            ignore = false;

            if (eLft.combatTbl.type == "El" && eMtd &&
                other.type != "El") {
                str += "- 0 Elephants can't aid " +
                        enemy.type + "\r";
                ignore = true;
            } else if (eLft.combatTbl.type == "Exp") {
                str += "- 0 Expendables can't aid\r";
                ignore = true;
            } else if (other.type == "Exp") {
                str += "- 0 Expendables can't recieve aid\r";
                ignore = true;
            } else if (_mtd && myTurn && ! eLft.combatTbl.mtd &&
                        ! (eLft.combatTbl.type == "Bw" ||
                           eLft.combatTbl.type == "Cb") ) {
             str+="- 0 left flank can't aid v mounted in enemy turn\r";
                ignore = true;
            }

            if (! ignore) {
                str += "- 1 left flank overlapped\r";
```

```
            --s;
        }
    }
}

// - 1 for right flank attacked
if (eRightFlank instanceof Element) {
  str += "- 1 right flank attacked-cannot recoil or flee!\r";
    --s;
} else {

    // - 1 for right flank overlapped
    if (eRht instanceof Element) {

        ignore = false;

        if (eRht.combatTbl.type == "El" &&
                eMtd && other.type != "El") {
            str += "- 0 Elephants can't aid " +
                    enemy.type + "\r";
            ignore = true;
        } else if (eRht.combatTbl.type == "Exp") {
            str += "- 0 Expendables can't aid\r";
            ignore = true;
        } else if (other.type == "Exp") {
            str += "- 0 Expendables can't recieve aid\r";
            ignore = true;
        } else if (_mtd && myTurn && ! eRht.combatTbl.mtd &&
                ! (eRht.combatTbl.type == "Bw" ||
                   eRht.combatTbl.type == "Cb")) {
      str+="- 0 right flank can't aid v mounted in enemy turn";
            ignore = true;
        }

        if (! ignore) {
            str += "- 1 right flank overlapped\r";
            --s;
        }
    }
}

// - 1  If troops or terrain already in contact with its rear
// edge or rear corner would prevent any recoil
var rearRank:Element  = _element.getRearRankElement();
var elmtsBehind:Array = rearRank.elementsBehind();
for (var i:Number = 0; i < elmtsBehind.length; ++i) {
    if ( _element.isFriendly(elmtsBehind[i])) {
        if (_element.angle != elmtsBehind[i].angle) {
         str+="- 1 retreat blocked-cannot recoil or flee!\r";
            --s; break;
        }
    } else {
        str += "- 1 enemy behind-cannot recoil or flee!\r";
        --s; break;
    }
}
```

```
// Hills count as clear for these results   NO THEY DON'T! +1
// IF ON HIGHER GROUND
if (terrain == "Hill") {
    terrain = "Clear";
}

if (terrain != "Clear") {

    // - 1 If Pikes 'Fast', Swords, Warriors (S) or (O) or
    // Hordes (O) and in close combat against foot while in
    // difficult going.
    if ( ! eMtd && _type == "pk" && _grade == "Fast" ) {
        // "- 1 for Pikes 'Fast' in difficult terrain fighting
        // foot\r"
        str += "- 1 Pikes 'F' v foot in difficult terrain\r";
        --s;
    } else if ( ! eMtd && _type == "Sw" ) {
        //  "- 1 for Swords in difficult terrain fighting
        // foot\r"
        str += "- 1 Swords v foot in difficult terrain\r";
        --s;
    } else if ( ! eMtd && _type == "Wb" &&
              (_grade == "Superior" ||
               _grade == "Ordinary") ) {
        // "- 1 for Warriors 'Superior' or 'Ordinary' in
        // difficult terrain fighting foot\r"
  str+="- 1 Warriors 'S' or 'O' v foot in difficult terrain\r";
        --s;
    } else if(! eMtd && _type == "Hd" && _grade =="Ordinary") {
        // "- 1 for Hordes 'Superior' or 'Ordinary' in
        // difficult terrain fighting foot\r"
    str+="- 1 Hordes 'S' or 'O' v foot in difficult terrain\r";
        --s;
    }

    // - 2 If mounted troops, Spears, Pikes except (F) or train
    // and in close combat in rough or difficult going.
    if (_mtd ) {
        // "- 2 for mounted troops fighting in difficult
        // terrain\r"
        str += "- 2 mounted in difficult terrain\r";
        s -= 2;
    } else if (_type == "Sp") {
        // "- 2 for Spears fighting in difficult terrain\r"
        str += "- 2 Spears in difficult terrain\r";
        s -= 2;
    } else if (_type == "Pk" && _grade != "Fast") {
        // "- 2 for Pikes (except 'Fast') fighting in difficult
        // terrain\r"
        str += "- 2 Pikes (except 'F') in difficult terrain\r";
        s -= 2;
    } else if (_type == "Art") {
        // "- 2 for Artillery fighting in difficult terrain\r"
        str += "- 2 Artillery in difficult terrain\r";
        s -= 2;
    }
}
```

```
//////////
//
// support factors

//
// +1 if
//
// Light Horse (F) supported by a 2nd rank of these against
// foot.
if (other.mtd == false && _type == "LH" && _grade == "Fast" &&
                 support.type == "LH" &&
                 support.grade == "Fast") {
    // "+1 for Lt. Horse 'Fast' supported by 2nd rank of Lt.
    // Horse 'Fast' against foot\r"
    str += "+1 2nd rank of Lt. Horse 'F' v foot\r";
    ++s;
}
//
// Spears supported by a 2nd rank of Spears against Elephants,
// Knights, Swords or Warband.
if (_type == "Sp" && support.type == "Sp") {
    if (other.type == "El" || other.type == "Kn" ||
        other.type == "Sw" || other.type == "Wb") {
        // "+1 for Spears supported by a 2nd rank of Spears
        // against Elephants, Knights, Swords or Warriors\r"
        str += "+1 2nd rank of Spears v " + enemy.type + "\r";
        ++s;
    }
}
//
// Pikes for each supporting consecutive 2nd or 3rd rank of
// Pikes of the same grade
//  - unless fighting against Cavalry, Light Horse, Skirmishers
if (_type == "Pk" && support.type == "Pk" &&
                     support.grade == _grade) {
    if ( ! (other.type == "Cv" || other.type == "LH" ||
            other.type == "Sk") ) {
        // "+1 for Pikes supported by a 2nd rank of Pikes of
        // same grade unless fighting Cavalry, Lt. Horse, or
        // Skirmishers\r"
        str += "+1 2nd rank of Pikes v " + enemy.type + "\r";
        ++s;
       if (sSupport.type == "Pk" && sSupport.grade == _grade) {
            // "+1 for Pikes supported by a 3rd rank of Pikes
            // of same grade\r"
          str += "+1 3rd rank of Pikes v " + enemy.type + "\r";
            ++s;
        }
    }
}

//
// - 1 if
//
```

```
 // Foot except Swords, Warband or Skirmishers - fighting
 // against enemy Spears supported by a 2nd rank of Spears of
 // the same grade
 if (other.type == "Sp" && oOther.type == "Sp" &&
                          other.grade == oOther.grade &&
     ! _mtd && ! (_type == "Sw" || _type == "Wb" ||
                  _type == "Sk") ) {
     // "- 1 for enemy Spears supported by a 2nd rank of enemy
     // Spears of the same grade against foot other than Swords,
     // Warriors or Skirmishers\r"
     str += "- 1 " + _element.type + " v 2nd rank of Spears\r";
     --s;
 }
 // Foot except Skirmishers - if fighting against Pikes that
// have a supporting consecutive 4th rank of Pikes of same grade
 if (other.type == "Pk" && oOther.type == "Pk" &&
     ooOther.type == "Pk" && oooOther.type == "Pk" &&
     ooOther.grade == oooOther.grade &&
     ! _mtd && _type != "Sk") {
     // "- 1 for enemy Pikes supported by a 4th rank of enemy
     // Pikes of the same grade against foot other than
     // Skirmishers\r" (except Skirmishers)
     str += "- 1 foot v 4th rank of Pikes\r";

     --s;
 }

 //
 // if enemy turn
 if (! myTurn && ! shooting ) {

     //
     // +1 if
     //
     // Swords (except "Fast") supported by a 2nd rank of Swords
     // or Spears against Elephants or Knights
     if (_type == "Sw" && _grade != "Fast" &&
         (support.type == "Sw" || support.type == "Sp") &&
         (other.type == "El" || other.type == "Kn")) {
         // "+1 for Swords (except 'Fast') supported by a 2nd
        // rank of Swords or Spears against Elephants or Knights
         // in enemy turn\r"
         if (support.type == "Sw") {
             str += "+1 2nd rank of Swords v " + enemy.type +
                    " in enemy turn\r";
         } else {
             str += "+1 supporting Spears v " + enemy.type +
                    " in enemy turn\r";
         }

         ++s;
     }
     // Archers supported by a 2nd rank of Archers of the same
     // grade
     if ( _type == "Bw" && support.type == "Bw" &&
         _grade == support.grade) {
         // "+1 for Archers supported by a 2nd rank Archers of
```

```
            // the same grade in enemy turn\r"
            str += "+1 2nd rank of Archers in enemy turn\r";
            ++s;
        }
        // Crossbows supported by a 2nd rank of Crossbows of the
        // same grade
        if ( _type == "Cb" && support.type == "Cb" &&
             _grade == support.grade) {
            // "+1 for Crossbows supported by a 2nd rank Crossbows
            // of the same grade in enemy turn\r";
            str += "+1 2nd rank of Crossbows in enemy turn\r";
            ++s;
        }
        // Swords 'Superior' or 'Ordinary' supported by a 2nd rank
        // of Archer 'Superior' or 'Ordinary': or vice versa
        // - against foot
        if (! other.mtd && _type == "Sw" &&
            (_grade == "Superior" || _grade == "Ordinary") &&
            support.type == "Bw" && (support.grade == "Superior" ||
            support.grade == "Ordinary")) {
            // "+1 for Swords 'Superior' or 'Ordinary' supported by
            // a 2nd rank of Archers 'Superior' or 'Ordinary'
            // against foot in enemy turn\r"
     str+="+1 supporting Archers 'S' or 'O' v foot in enemy turn\r";
            ++s;
        }
        if (! other.mtd && _type == "Bw" && (_grade == "Superior"||
            _grade == "Ordinary") &&
            support.type == "Sw" && (support.grade == "Superior" ||
            support.grade == "Ordinary")) {
            // "+1 for Archers 'Superior' or 'Ordinary' supported
            // by a 2nd rank of Swords 'Superior' or 'Ordinary'
            // against foot in enemy turn\r";
    str += "+1 supporting Swords 'S' or 'O' v foot in enemy turn\r";
            ++s;
        }
        // Swords 'Superior' or 'Ordinary' supported by a 2nd rank
        // of Crossbow 'Superior' or 'Ordinary': or vice versa
        // - against foot
        if (! other.mtd && _type == "Sw" && (_grade == "Superior"||
            _grade == "Ordinary") &&
            support.type == "Cb" && (support.grade == "Superior" ||
            support.grade == "Ordinary")) {
            // "+1 for Swords 'Superior' or 'Ordinary' supported by
            // 2nd rank of Crossbow 'Superior' or 'Ordinary'against
            // foot in enemy turn\r"
            str += "+1 supporting Crossbows 'S' or 'O' v foot in
                    enemy turn\r";
            ++s;
        }
        if (! other.mtd && _type == "Cb" && (_grade == "Superior"||
            _grade == "Ordinary") &&
            support.type == "Sw" && (support.grade == "Superior" ||
            support.grade == "Ordinary")) {
            // "+1 for Crossbow 'Superior' or 'Ordinary' supported
            // by a 2nd rank of Swords 'Superior' or 'Ordinary'
            // against foot in enemy turn\r"
```

```
            str += "+1 supporting Swords v foot in enemy turn\r";
            ++s;
        }
        // Regular Auxilia (S) supported by a 2nd rank of these -
        // against Knights.
        if (other.type == "Kn" && _type=="AxS" &&_element.regular&&
            support.type == "AxS") {
            // "+1 for Regular Lt. Infantry 'Superior' supported by
            // a 2nd rank of Lt. Infantry 'Superior' againstKnights
            // in enemy turn\r"
str+="+1 2nd rank of 'S' Lt. Infantry v Knights in enemy turn\r";
            ++s;
        }
        // Skirmishers supported by a 2nd rank of Skirmishers (O) -
        // against Lt. Horse or Skirmishers
        if (_type == "Sk" && support.type == "Sk" &&
            support.grade == "Ordinary" &&
            (other.type == "LH" || other.type == "Sk")) {
            // "+1 for Skirmishers supported by a 2nd rank of
            // Skirmishers 'Ordinary' against Lt. Horse or
            // Skirmishers  in enemy turn\r"
            str += "+1 2nd rank of Skirmishers 'O' v " + enemy.type
            + " in enemy turn\r";
            ++s;
        }
        // Cavalry supported by a 2nd rank of Skirmishers* (S) or
        // (I) -  against Cavalry
        if (other.type == "Cv" && _type == "Cv" &&
            support.supporters) {
            // "+1 for Cavalry supported by a 2nd rank of
            // Skirmishers* fighting against Cavalry in
            // enemy turn\r"
   str += "+1 supporting Skirmishers* v Cavalry in enemy turn\r";
            ++s;
        }
        // Spears, Pikes, Blades or Lt. Infantry supported by a 2nd
        // rank of Skirmishers* - against Warband or mounted troops
        if ( (other.mtd || other.type == "Wb") &&
             (_type == "Sp" || _type == "Pk" || _type == "Sw" ||
              _type == "Ax") ) {
            if (support.supporters || (support.type == _type &&
                sSupport.supporters)) {
                // "+1 for Spears, Pikes, Swords or Lt. Infantry
                // supported by a rank of Skirmishers* fighting
                // against Warband or mounted troops in
                // enemy turn\r"
                if (other.type == "Wb") {
   str += "+1 supporting Skirmishers* v Warband in enemy turn\r";
                } else {
   str += "+1 supporting Skirmishers* v mounted in enemy turn\r";
                }
                ++s;
            }
        }

        //
        // - 1 if
```

```
        //
        // Foot except Skirmishers - if fighting against Warband
        // that have a supporting 2nd rank of Warband.
        if (other.type == "Wb" && ! _mtd && _type != "Sk") {
            // "- 1 for enemy Warriors supported by a 2nd rank of
            // enemy Warriors against foot other than Skirmishers
            // in enemy turn\r" (except Skirmishers)
          str += "- 1 foot v 2nd rank of Warriors in enemy turn\r";
            --s;
        }
    }

    // +1   If the opposing element is disadvantaged by weather.
    //  disadvantage if Archers, Crossbows, Shot, or Artillery;
    //   and in close combat against mounted.
    if (_game.weather == "Rain" && _mtd && ! shooting &&
       (other.type == "Bw" || other.type == "Cb" ||
        other.type == "Sh" || other.type == "Art")) {
        str += "+1 enemy disadvantaged by rain\r";
        ++s;
    }

    return [s, str];
}

//
// shootingFactors()
//
//  Tally Shooting factors for this Element
//
//   args:
//           enemy  -- enemy Element
// shootingSupport  -- # of other shooters supporting this element
//
public function shootingFactors(enemy:Element,
         shootingSupport:Number):Array {

    var s:Number = 0;           // support modifier
    var str:String = "";        // string with explanations

    // supporting 2nd rank
    var support:CombatTable =
        _element.elementBehind(true).combatTbl;


    //////////
    //
    // shooting factors

    // +1 If a primary shooter aided by another element contiguous
    // behind it.
    if (support.type == "Bw" || support.type == "Cb" ||
        support.type == "Sh" || support.type == "Art") {
        str += "+1 shooting aided by rank behind\r";
        ++s;
    }
```

```
// -1 For each shooting element aiding an enemy primary shooter
//
//  other than from contiguous behind it.
if (shootingSupport == 1) {
    str += "- 1 enemy shooting aided by other shooter\r";
    --s;
} else if (shootingSupport > 1) {
    str += "- " + shootingSupport +
           " enemy shooting aided by " + shootingSupport +
           " other shooters\r";
    s -= shootingSupport;
}

// check support from left
var enemyLft:Element     = enemy.elementToLeft(true);
var enemyLftType:String  = enemyLft.combatTbl.type;
if ((enemyLft.status == "Engaged" ||
     enemyLft.status == "Moving") &&
    (enemyLftType == "Bw" || enemyLftType == "Cb" ||
     enemyLftType == "Sh" || enemyLftType == "Art")) {
    str += "- 1 enemy shooting aided from left by " +
           enemyLft.type + "\r";
    --s;
}
// check support from right
var enemyRht:Element     = enemy.elementToRight(true);
var enemyRhtType:String  = enemyRht.combatTbl.type;
if ((enemyRht.status == "Engaged" ||
     enemyRht.status == "Moving") &&
    (enemyRhtType == "Bw" || enemyRhtType == "Cb" ||
     enemyRhtType == "Sh" || enemyRhtType == "Art")) {
    str += "- 1 enemy shooting aided from right by " +
           enemyRht.type + "\r";
    --s;
}

// +1 If the opposing element is disadvantaged by weather.
// Rain disadvantage if Archers, Crossbows, Shot, or Artillery
// in rain
var enemyType:String = enemy.combatTbl.type;
if (_game.weather == "Rain" &&
   (enemyType == "Bw" || enemyType == "Cb" ||
    enemyType == "Sh" || enemyType == "Art")) {
    str += "+1 enemy shooting disadvantaged by rain\r";
    ++s;
} else {

    // Wind

    //  disadvantage if Archers, Crossbows, Shot, or Artillery
    //   shooting across a strong wind
    var windAngleDiff:Number =
        Utils.compareAngle(_game.windDirection, enemy.angle);
    var enemyType:String = enemy.combatTbl.type;
    if (_game.windStregth == "Strong" &&
        windAngleDiff > 45 && windAngleDiff != 180 &&
       (enemyType == "Bw" || enemyType == "Cb" ||
```

```
            enemyType == "Sh" || enemyType == "Art")) {
            str += "+1 enemy shooting across strong wind\r";
            ++s;
        }

        //  disadvantage if Archers, Crossbows, Shot, or Artillery
        //   shooting into a strong wind
        windAngleDiff = Utils.compareAngle(_game.windDirection,
                _element.angle);
        if (_game.windStregth == "Strong" &&
            windAngleDiff == 180 &&
           (_type == "Bw" || _type == "Cb" || _type == "Sh" ||
            _type == "Art")) {
            str += "-1 shooting into strong wind\r";
            --s;
        }
    }

    return [s, str];
}

//
// shootingTally()
//
//  Rough tally of this elements shooting effectiveness
//  against a specific enemy.  Use by the Shoots object to
//  determine who should be the primary shooter at a target
//
//   args:
//            enemy  -- enemy Element
//
public function shootingTally(enemy:Element):Number {

    var tally:Number = enemy.combatTbl.mtd ? _FVvMtd : _FVvFt;
    // supporting 2nd ranks cbt table
    var support:CombatTable =
        _element.elementBehind(true).combatTbl;

    // +1 If a primary shooter aided by another element contiguous
    // behind it.
    if (support.type == "Bw" || support.type == "Cb" ||
        support.type == "Sh" || support.type == "Art") {
        ++tally;
    }

    return tally;
}

//
// grading()
//
//  Determine factors dictated by the grading of both Elements in
//  combat. These Grading Factors are add AFTER the dice for the
//  battle have been rolled
//
//   args:
//        my_score  -- this Elements dice score
```

```
//      your_score  -- enemy Elements dice score
//        myTurn    -- true if it is this Elements turn
//           other  -- ememy's combat table object
//        shooting  -- true if a distant shooting combat
//
private function grading(my_score:Number, your_score:Number,
    myTurn:Boolean, other:CombatTable, shooting:Boolean):Array {

    switch (_grade) {
        case "Superior":
            if (my_score > your_score && myTurn) {
                // "+1 for Superior who scored more in their own
                // turn"
                return [+1,
                        "+1 'Superior' scoring more in own turn"];
            } else if (my_score < your_score && ! myTurn) {


                if (shooting &&
                     other.grade == "Superior"  &&
                    (other.type == "Bw" ||
                     other.type == "Cb" ||
                     other.type == "Sh" ||
                     other.type == "Art")) {

                    // "+1 for Superior who scored less in enemy
                    // turn" &
                    //  -1  if any troops shot at by Superior (S)
                    // troops whose total score is more.
                    return [+0, "+0 'Superior,
                            ' but shot at by 'Superior' shooters"];

                } else {

                    // "+1 for Superior who scored less in
                    // enemy turn"
                    return [+1,
                       "+1 'Superior' scoring less in enemy turn"];
                }
            }
            break;
        case "Ordinary":
            break;
        case "Inferior":
            if (my_score <= your_score) {
                // "- 1 for Inferior who scored equal to or less"
                return [-1,
                          "- 1 'Inferior' scoring equal or less"];
            }
            break;
        case "Fast":
            if ( ! myTurn && ( (_mtd && other.mtd) || ! _mtd) ) {
                if (! _mtd && ! shooting) {
                    // "- 1 for Fast foot in enemy turn"
                    return [-1, "- 1 'Fast' foot in enemy turn"];
                } else {
                    // "- 1 for Fast mounted fighting mounted in
```

```
                    // enemy turn"
                    return [-1,
                     "- 1 'Fast' mounted v mounted in enemy turn"];
                }
            }
            break;
    }

    // check for superior shooters
    if ( shooting && my_score < your_score &&
         (other.type == "Bw" ||
          other.type == "Cb" ||
          other.type == "Sh" ||
          other.type == "Art") &&
          other.grade == "Superior") {
        //  -1  if any troops shot at by Superior (S) troops whose
        // total score is more.
        return [-1,
          "- 1 shot at by 'Superior' shooters who score more"];
    }

    return [0, ""];
}

//
// displayOddsV()
//
//  display odds for battle against another element
//
//   args:
//           enemy  -- Element being fought
//   battle_window  -- path of the battle or shooting window
//        shooting  -- true if it is a distant shooting combat
// shootingSupport  -- # of other shooters supporting this element
//
public function displayOddsV(enemy:Element,
                  battle_window:MovieClip, shooting:Boolean,
                    shootingSupport:Number):Void {

    // get opposing table
    var other:CombatTable = enemy.combatTbl;

    //////
    //
    // attacker

    // v foot or v mounted?
    _atkr_fv = other.mtd ? _FVvMtd : _FVvFt;

    if (shooting && (other.type == "Bw" || other.type == "Cb" ||
        other.type == "Sh" || other.type == "Art")) {
        battle_window._atkr_base.text = "+" +
            _atkr_fv + " " + _element.type + " v shooting";
    } else {

        // +1 to base if Bows shooting v foot without being shot at
        if (shooting && ! other.mtd && (_type == "Bw" ||
```

```
            _type == "Cb") ) { ++_atkr_fv; }

        battle_window._atkr_base.text = "+" + _atkr_fv +
            " " + _element.type + " v " +
            (other.mtd ? "mounted" : "foot");
    }

    // reset the dice & flags
    battle_window._atkr_dice.gotoAndStop("black");
battle_window._atkr_icon.gotoAndStop(Utils.convDots(_element.type));
    battle_window._atkr_icon._alpha = 100;
    battle_window._atkr_result.gotoAndStop(1);

    battle_window._atkr_type.text  = _element.type;
    battle_window._atkr_grade.text = _grade;

    // display element picture
    var suffix:String;
    if (_element.player.thePlayer == Player.active.thePlayer) {
        suffix = "_rht";
    } else {
        suffix = "_lft";
    }
    battle_window._atkr_fig.gotoAndStop(_element.picture + suffix);

    battle_window._atkr_if_less.text = lessThan(other.type, true,
            enemy.terrain, shooting) + " if less";
    battle_window._atkr_if_less._alpha = 70;
    battle_window._atkr_if_less.textColor = 0x000000;
    battle_window._atkr_if_dbld.text = doubledBy(other.type, true,
            enemy.terrain, shooting) + " if doubled";
    battle_window._atkr_if_dbld._alpha = 70;
    battle_window._atkr_if_dbld.textColor = 0x000000;

    // Support & Tactical Factors note
    // shootingSupport subtracts from defender, not adds
    // to attacker
    var r:Array = factors(enemy, true, enemy.terrain, shooting);
    var r2:Array;
    if (shooting) { r2 = shootingFactors(enemy,
            0); r[0] += r2[0]; r[1] += r2[1]; }
    _atkr_fv += r[0];
    battle_window._atkr_detail.text = r[1];


    battle_window._atkr_total.text = "Total: " + _atkr_fv + " +";
    battle_window._atkr_mod.text = "";
    battle_window._atkr_equals.text = "";

    //////
    //
    // defender

    // v foot or v mounted?
    if (_mtd) {
        _dfdr_fv = other.FVvMtd;
    } else {
```

```
            _dfdr_fv = other.FVvFt;
        }

        if (shooting) {
            battle_window._dfdr_base.text = "+" + _dfdr_fv +
                " " + enemy.type + " v shooting";
        } else {
            battle_window._dfdr_base.text = "+" + _dfdr_fv +
                " " + enemy.type + " v " + (_mtd ? "mounted" : "foot");
        }

        // reset the dice & flags
        battle_window._dfdr_dice.gotoAndStop("black");
      battle_window._dfdr_icon.gotoAndStop(Utils.convDots(enemy.type));
        battle_window._dfdr_icon._alpha = 100;
        battle_window._dfdr_result.gotoAndStop(1);

        battle_window._dfdr_type.text  = enemy.type;
        battle_window._dfdr_grade.text = "'" + other.grade + "'";

        // display element picture
        var suffix:String;
        if (enemy.player.thePlayer == Player.active.thePlayer) {
            suffix = "_rht";
        } else {
            suffix = "_lft";
        }
        battle_window._dfdr_fig.gotoAndStop(enemy.picture + suffix);

        battle_window._dfdr_if_less.text = other.lessThan(_type, false,
                enemy.terrain, shooting) + " if less";
        battle_window._dfdr_if_less._alpha = 70;
        battle_window._dfdr_if_less.textColor = 0x000000;
        battle_window._dfdr_if_dbld.text = other.doubledBy(_type,false,
                enemy.terrain, shooting) + " if doubled";
        battle_window._dfdr_if_dbld._alpha = 70;
        battle_window._dfdr_if_dbld.textColor = 0x000000;

        // Support & Tactical Factors
        //  defender gets no additional shooting supports
        r = other.factors(_element, false, enemy.terrain, shooting);
        if (shooting) { r2 = other.shootingFactors(_element,
            shootingSupport); r[0] += r2[0]; r[1] += r2[1]; }
        _dfdr_fv += r[0];
        battle_window._dfdr_detail.text = r[1];

        battle_window._dfdr_total.text = "Total: " + _dfdr_fv + " +";
        battle_window._dfdr_mod.text = "";
        battle_window._dfdr_equals.text = "";

        // enable fight button
        battle_window._fight._alpha  = 100;
        battle_window._fight.enabled = true;

    }

    //
```

```
// conductBattleV()
//
//  conduct battle (roll dice) against another element
//
//   args:
//         enemy    -- Element being fought
// battle_window    -- path of the battle or shooting window
//      shooting    -- true if it is a distant shooting combat
//
public function conductBattleV(enemy:Element,
         battle_window:MovieClip, shooting:Boolean):Void {

    var fmt18:TextFormat = new TextFormat(); fmt18.size = 14;

    // get opposing table
    var other:CombatTable = enemy.combatTbl;

    // roll them dice
    var atkr_dice:Number = Utils.randomInt(1, 6);
    var atkr_score:Number = _atkr_fv + atkr_dice;
    battle_window._atkr_dice.gotoAndStop("n" + atkr_dice);

    var dfdr_dice:Number = Utils.randomInt(1, 6);
    var dfdr_score:Number = _dfdr_fv + dfdr_dice;
    battle_window._dfdr_dice.gotoAndStop("n" + dfdr_dice);

    // grading Factors
    var atkr_prefix:String = "";
    var atkr_mod = grading(atkr_score, dfdr_score, true, other,
             shooting);
    if (atkr_mod[0] != 0) {
        atkr_prefix = ((atkr_mod[0] > 0) ? "+" : " ") +atkr_mod[0];
        atkr_score += atkr_mod[0];
        battle_window._atkr_mod.text = atkr_mod[1];
    }
    var dfdr_prefix:String = "";
    var dfdr_mod = other.grading(dfdr_score, atkr_score, false,
        this, shooting);
    if (dfdr_mod[0] != 0) {
        dfdr_prefix = ((dfdr_mod[0] > 0) ? "+" : " ") +dfdr_mod[0];
        dfdr_score += dfdr_mod[0];
        battle_window._dfdr_mod.text = dfdr_mod[1];
    }

    battle_window._atkr_equals.text = atkr_prefix +
        " = " + atkr_score;
    battle_window._dfdr_equals.text = dfdr_prefix +
        " = " + dfdr_score;

    // disable fight button
    battle_window._fight._alpha  = 30;
    battle_window._fight.enabled = false;
    var r:String;

    // play the sword/arrow battle sound
    if (shooting) {
        _element.game.playSnd("arrow");
```

```
} else {
    _element.game.playSnd("sword");
}

if (atkr_score == dfdr_score) {

    /////
    // tie

    battle_window._atkr_result.gotoAndPlay("tie");
    battle_window._dfdr_result.gotoAndPlay("tie");
    battle_window._atkr_if_less._alpha = 20;
    battle_window._atkr_if_dbld._alpha = 20;
    battle_window._dfdr_if_less._alpha = 20;
    battle_window._dfdr_if_dbld._alpha = 20;
    battle_window._atkr_equals.textColor = 0x000000;
    battle_window._dfdr_equals.textColor = 0x000000;

} else if (atkr_score > dfdr_score) {

    /////
    // attacker wins

    // disengage elements
    if (! shooting) {
        _element.disengage();
        enemy.disengage();
    }

    battle_window._atkr_result.gotoAndPlay(
        (_element.game.playerTurn ? "black" : "white") +
            "_flag");
    battle_window._atkr_if_less._alpha = 20;
    battle_window._atkr_if_dbld._alpha = 20;
    battle_window._atkr_equals.textColor = 0x000000;
    battle_window._dfdr_equals.textColor = 0x990000;
    // battle_window._dfdr_icon._alpha = 20;

    if (dfdr_score*2 <= atkr_score) {

        // defender doubled
        battle_window._dfdr_if_less._alpha = 20;
        battle_window._dfdr_if_dbld._alpha = 100;
        battle_window._dfdr_if_dbld.textColor = 0x990000;
        battle_window._dfdr_if_dbld.setTextFormat(fmt18);

        r = other.doubledBy(_type, false, "Clear", shooting);

    } else {

        // defender less
        battle_window._dfdr_if_dbld._alpha = 20;
        battle_window._dfdr_if_less._alpha = 100;
        // 66
        battle_window._dfdr_if_less.textColor = 0x990000;

        battle_window._dfdr_if_less.setTextFormat(fmt18);
```

```
            r = other.lessThan(_type, false, "Clear", shooting);
        }

        if (r == "Killed" || r == "Spent") {
            battle_window._dfdr_result.gotoAndPlay("death");
        } else if (r == "Flee" || r == "Repulsed") {
            battle_window._dfdr_result.gotoAndPlay("flee");
        }

        var eFlank:Element;     // recoil any flanking elements
        eFlank = _element.leftFlanked();  if (eFlank != undefined){
            eFlank.recoil();
        }
        eFlank = _element.rightFlanked(); if (eFlank != undefined){
            eFlank.recoil();
        }
        eFlank = _element.rearAttacked(); if (eFlank != undefined){
            eFlank.recoil();
        }

        switch (r) {
            case "Stand":
                break;
            case "Recoil":
                enemy.recoil(shooting ? 0.8 : 0.1);
                pursue(enemy, true);
                break;
            case "Repulsed":
                _element.game.playSnd("drums");
                enemy.repulsed();
                _element.stateNormal();
                break;
            case "Flee":
                _element.game.playSnd("runaway");
                enemy.flee();
                pursue(enemy, true);
                break;
            case "Spent":
                enemy.spent();
                _element.stateNormal();
                break;
            case "Killed":
                enemy.killed(false, shooting ? 0.8 : 0.1);
                pursue(enemy, true);
                break;
        }

    } else {

        /////
        // defender wins

        // disengage elements
        if (! shooting) {
            _element.disengage();
            enemy.disengage();
```

```
}

battle_window._dfdr_result.gotoAndPlay(
    (_element.game.playerTurn ? "white" : "black") +
    "_flag");
battle_window._dfdr_if_less._alpha = 20;
battle_window._dfdr_if_dbld._alpha = 20;
battle_window._atkr_equals.textColor = 0x990000;
battle_window._dfdr_equals.textColor = 0x000000;
// battle_window._atkr_icon._alpha = 20;

if (atkr_score*2 <= dfdr_score) {

    // attacker doubled
    battle_window._atkr_if_less._alpha = 20;
    battle_window._atkr_if_dbld._alpha = 100;
    battle_window._atkr_if_dbld.textColor = 0x990000;
    battle_window._atkr_if_dbld.setTextFormat(fmt18);
    r = doubledBy(other.type, true, "Clear", shooting);

} else {

    // attacker less
    battle_window._atkr_if_dbld._alpha = 20;
    battle_window._atkr_if_less._alpha = 100;
    battle_window._atkr_if_less.textColor = 0x990000;
    battle_window._atkr_if_less.setTextFormat(fmt18);
    r = lessThan(other.type, true, "Clear", shooting);
}

if (r == "Killed" || r == "Spent") {
    battle_window._atkr_result.gotoAndPlay("death");
} else if (r == "Flee" || r == "Repulsed") {
    battle_window._atkr_result.gotoAndPlay("flee");
}

var eFlank:Element;     // recoil any flanking elements
eFlank = enemy.leftFlanked();  if (eFlank != undefined)  {
    eFlank.recoil();
}
eFlank = enemy.rightFlanked(); if (eFlank != undefined)  {
    eFlank.recoil();
}
eFlank = enemy.rearAttacked(); if (eFlank != undefined)  {
    eFlank.recoil();
}

switch (r) {
    case "Stand":
        break;
    case "Recoil":
        _element.recoil(shooting ? 0.8 : 0.1);
        other.pursue(_element, false);
        break;
    case "Repulsed":
        _element.game.playSnd("drums");
        _element.repulsed();
```

```
                enemy.stateNormal();
                break;
            case "Flee":
                _element.game.playSnd("runaway");
                _element.flee();
                other.pursue(_element, false);
                break;
            case "Spent":
                _element.spent();
                enemy.stateNormal();
                break;
            case "Killed":
                _element.killed(false, shooting ? 0.8 : 0.1);
                other.pursue(_element, false);
                break;
        }
    }
}

//
// pursue()
//
//  rtn true if Element e will pursue this Element
//
//   args:
//      e   -- Element being tested against
// myTurn   -- true if it is this Elements turn
//
//   return -- true if the element e should pursue
//
public function pursue(enemy:Element, myTurn:Boolean):Void {

    // get opposing table
    var other:CombatTable = enemy.combatTbl;

    var pursue:Boolean = false;

    // only certain elements pursue,
    // Elephants, Knights, Pikes, Swords, Spears, Expendables,
    // Warriors,
    //  Irregular elements of - Light Horse (S), Swords (S) or (F),
    //   Spears (O), Hordes (S) or (F).
    if (_type == "El" || _type == "Kn" || _type == "Pk" ||
        _type == "Sw" || _type == "Exp" || _type == "Wb") {
        pursue = true;
    } else if ( ! _element.regular) {
        if (_type == "LH" && _grade == "Superior") {
            pursue = true;
        } else if (_type == "Sw" && (_grade == "Superior" ||
            _grade == "Fast")) {
            pursue = true;
        } else if (_type == "Sp" && _grade == "Ordinary") {
            pursue = true;
        } else if (_type == "Hd" && (_grade == "Superior" ||
            _grade == "Fast")) {
            pursue = true;
        }
```

```
    }

    // unless...
    //  (a) Knights who fought in an enemy bound unless against
    // Knights,
    //  (b) foot who fought against mounted in an enemy bound
    //  (c) foot against Light Horse
    //  (d) regular Swords or Spears who fought against foot
    if ( (! myTurn && _type == "Kn" && other.type != "Kn") ||
         (! myTurn && ! _mtd && other.mtd) ||
         (! _mtd && other.type == "LH") ||
         (_element.regular && ! other.mtd && (_type == "Sw" ||
            _type == "Sp")) ) {

        // in which case pursuit is OPTIONAL, so the computer will
        //  have to decide, FOR NOW, DO NOT PURSUE

        pursue = false;
    }

    if (pursue) {
        _element.pursue(enemy.depth);
    }
}

//
// moveThrough()
//
//  rtn true if Element e can pass through this element
//
//   args:
//      e   -- Element being tested against
//
//   return -- true if the element e can pass through
//
public function moveThrough(e:Element):Boolean {

    // get e's table
    var other:CombatTable = e.combatTbl;

    // must not be in combat and must be friendly
    if (e.status == "Engaged" || ! _element.isFriendly(e)) {
        return false;
    }

    // ensure same or opposite direction
    if (! (_element.angle == e.angle ||_element.angle ==
        Utils.cleanAngle(e.angle - 180)))  { return false; }

    // Skirmishers can pass through any land troops
    if (_type == "Sk" || other.type == "Sk") { return true; }

    // Mounted can pass through Lt. Horse or any foot except Pikes
    // or Hordes.
    if (_mtd && other.type == "LH") { return true; }

    // Mounted can pass through any foot except Pikes or Hordes.
```

```
        if (_mtd && ! other.mtd && ! (other.type == "Pk" &&
            other.type == "Hd")) { return true; }

        // Regular Swords can pass through regular Swords
        if (_element.regular && _type == "Sw" && e.regular &&
            other.type == "Sw") { return true; }

        // Regular Lt. Horse or Regular Cavalry can pass through
        // regular Cavalry... FACING THE OPPOSITE DIRECTION. (?)
        if (_element.regular && (_type == "LH" || _type == "Cv") &&
            e.regular) { return true; }

        // Swords can be passed through by Lt. Inf or Bows if facing
        // the same direction
        if (_element.angle == e.angle && other.type == "Sw" &&
            (_type == "Bw" || _type == "Cb" || _type == "Ax" ||
                _type == "AxS")) { return true; }

        // otherwise no
        return false;
    }


    // accessors
    public function get type():String        { return _type; }
    public function get grade():String       { return _grade; }
    public function get supporters():Boolean { return _supporters; }
    public function get mtd():Boolean        { return _mtd; }
    public function get FVvMtd():Number      { return _FVvMtd; }
    public function get FVvFt():Number       { return _FVvFt; }

    // mutators
    public function set element(val:Element):Void { _element = val; }
}
```

Presentation Objects

All Presentation Objects are generic objects for use in any Flash program.

Animatem.as

```
////////////////////////////////////////////////////////////////////////
//
//  Animatem.as
//
//       Author: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: The Animatem object oversees multiple animating sprite
//               objects to yield solid, smooth, fractional of a
//               pixel animation. Each sprite has various properties
//               such as velocity, friction, fps for cell animation,
//               and collision detection.  All of these properties
//               need to be frequently updated for smooth animation.
//               To do this Animatem maintains an update() method
//               that is called as frequently as possible by the
//               programs main loop.
//
//               The update() method simply records the amount of time
//               passed since the last update(), and instructs each
//               sprite being controlled to update according to such
//               time passed. Time is measured in ticks, which are
//               1/60ths of a second, which was deemed a manageable
//               yet fine enough granularly. For instance, a sprite
//               with a velocity of  0.5 pixels per tick along the x
//               plane receives an update from Animatem informing it
//               that it has been three ticks since the last update.
//               The Sprite then knows to draw itself (0.5 * 3 =) 1.5
//               pixels further along the x axis.  Many sprites can
//               flexibly animate simultaneously and interactively,
//               and each respond to each other and the changing
//               environment.  Velocities and other parameters can be
//               instantly changed at the will of the environment,
//               often according to highly randomized factors.
//
//               Flash 8 demands that all movies to play at the same
//               constant speed (FPS) as the first root movie. It is
//               for this reason that Animatem works best with the
//               root movie set with a high FPS. Individual animating
//               Sprite objects may then each then be assigned their
//               own slower FPS.  This gives Animatem much more
//               animation flexibility and power than inherently
//               present Flash, as animations can be played at
//               different and precise rates, and even backwards.
//
//      Methods:
```

```
//
//        Animatem() - Constructor
//          update() - Called by the main loop of the program.
//                     Updates the current _updateTime and determines
//                     the amount of time passed since the last update.
//                     This time_passed is sent to each active sprite
//                     telling it to update its current position
//                     accordingly.
//       addSprite() - Adds a new sprite to the animator, as a specific
//                     channel is not specified the next available
//                     channel is used.
//      addSpriteN() - As for addSprite() but a specific channel n
//                     is specified.
//       setSprite() - As for addSprite(), but an actual MovieClip is
//                     given instead of a link name.
//      setSpriteN() - As for setSprite(), but a specific channel n
//                     is specified.
// clearAllSprites() - Empties the animator of all sprites. This is
//                     often useful when an environment resets.
//   releaseUpdate() - When an excessive amount of time passes between
//                     updates it is necessary to ignore the break,
//                     otherwise a very visible jump is seen in
//                     the animation. By setting the _updatePrev to 0
//                     the animator knows to do the next update with a
//                     minimal time_passed value.
//     clearSprite() - Clears a specific sprite from the animator,
//                     removing it from both the _sprites list and the
//                     _spriteList and deleting its attached movie from
//                     the main movieclip.
//         reserve() - Specifies a sprite channel as reserved and not
//                     to be recycled.
//     notReserved() - Checks if a channel is marked as reserved.
//
//     ...message passing methods to the controlling object
//       collision() - Passes collision messages to controlling object
//     deactivated() - Passes deactivated messages to controlling obj
//
//     ...general processes that can be performed on sprite objects
//    goToLocAtSpd() - Animate sprite to a point at set speed
//    goToLocInTme() - Animate sprite to point in set time
//    rotateInTime() - Rotate sprite to an angle in set time
//     scaleInTime() - Scale sprite to a size in a specific time
//   addDropShadow() - Adds a drop shadow filter to an animating sprite
//        addBevel() - Adds a bevel filter to an animating sprite
//         addGlow() - Adds a glow filter to an animating sprite
//    removeFilter() - Removes the last applied filter
//
//  Notes: A) Animatem has accessors and mutators to a sprite as
//         referenced by the sprites channel, but generally it's better
//         to get a reference to the sprite and access it directly.
//         The only important sprite mutator that Animatem needs to be
//         aware of is setClip().
//         B) List of controlled sprites is stored in two forms,once
//         in _sprites where the index represents a sprite's channel
//         again in _spriteList, which is a list of sprites currently
//         assigned.
//
```

```
class Animatem {

    // instance members

    // Object that spawned this Animator
    private var _obj:Object;

    // Movie clip to which sprites are attached and animated
    private var _path:MovieClip;

    // starting layer depth at which sprites are attached
    private var _depth:Number;

    // duration of Animatem tick is 1/60th of a second (60 fps), in
    // milliseconds (1/1000ths) that's 16.666
    private var _tick:Number = 1000 / 60;

    // maximum duration allowed between an updates (1/4 of a second)
    private var _timeCap:Number = 15;

    // time when update was called
    private var _updateTime:Number;

    // time of previous update
    private var _updatePrev:Number;

    // Array of sprite objects
    private var _sprites:Array;

    // list of reserved sprites
    private var _reserved:Array;

    // list of assigned sprites
    private var _spriteList:Array;

    //
    //  Constructor
    //
    //   args:
    //      path  -- MovieClip that new sprites are attached to,
    //               usually _root
    //      depth -- starting depth at which sprites are attached
    //      obj   -- Master object that spawed this animator, and to
    //               which all extenal calls are sent, such as when
    //               collisions and deactivations occur
    //
    public function Animatem(path:MovieClip, depth:Number,
             obj:Object) {

        _path       = path;
        _depth      = depth;
        _sprites    = new Array();
        _spriteList = new Array();
        _reserved   = new Array();
        _obj        = obj;
    }
```

```
//
// update()
//
//  Called by the main loop of the program.
//  Updates the current _updateTime and determines the amount of
//  time passed since the last update. This time_passed is sent to
//  each active sprite telling it to update its current position
//  accordingly.
//
public function update():Void {

    // an update tick = 1/60th of a second
    _updateTime = getTimer() / _tick;

    // execute only once we have a previous time
    if (_updatePrev > 0) {

        var time_passed:Number = _updateTime - _updatePrev;

        if (time_passed > _timeCap) {
            //  limit greatest time jump.
            time_passed = _timeCap;
        }

        // animate each sprite
        for(var i:Number = 0; i < _spriteList.length; ++i) {
            if (_sprites[_spriteList[i]].active) {
                _sprites[_spriteList[i]].update(_updateTime,
                         time_passed);
            }
        }
    }
    _updatePrev = _updateTime;
}

//
// addSprite()
//
//  Adds a new sprite to the animator, as a specific
//  channel is not specified the next available channel is used.
//
//   args:
//      clip -- the link name of the MovieClip to used
//         X -- x location of the sprite
//         Y -- y location of the sprite
//    active -- The active setting to be used,
//                       1 = animating
//                       0 = not animating
//                      -1 = animating but NOT detecting for
// collisions
//
//   return -- a refernce to the new sprite
//
public function addSprite(clip:String, X:Number, Y:Number,
         active:Number):Sprite {

    // find an empty sprite
```

```
    var n:Number = 0;
    for (; n < _sprites.length ; ++n) {
        if (_sprites[n] == undefined && notReserved(n)) { break; }
    }

    return addSpriteN(n, clip, X, Y, active);
}

//
// addSpriteN()
//
//  As for addSprite() but a specific channel n is specified
//
public function addSpriteN(n:Number, clip:String, X:Number,
        Y:Number, active:Number):Sprite {

    if (_sprites[n] == undefined) {

        _path.attachMovie(clip, n.toString(), _depth + n);
        _sprites[n] = new Sprite(this, n, _path[n], _path, X, Y,
                active);
        // tell the sprite the clip linkage name
        _sprites[n]._clip = clip;
        _spriteList.push(n);

    } else {
        throw new Error("Can't add sprite#" + n);
    }

    return _sprites[n];
}

//
// setSprite()
//
//  As for addSprite(), but an actual MovieClip is given instead
//  of a link name
//
public function setSprite(path:MovieClip, root:MovieClip,
         active:Number):Sprite {

    // find an empty sprite
    var n:Number = 0;
    for (; n < _sprites.length ; ++n) {
        if (_sprites[n] == undefined && notReserved(n)) { break; }
    }

    return setSpriteN(n, path, root, active);
}

//
// setSpriteN()
//
//  As for setSprite(), but a specific channel n is specified
//
public function setSpriteN(n:Number, path:MovieClip,root:MovieClip,
         active:Number):Sprite {
```

```
    if (_sprites[n] == undefined) {
        _sprites[n] = new Sprite(this, n, path, root, path._x,
                 path._y, active);
        _spriteList.push(n);
    } else {
        throw new Error("Can't add sprite#" + n);
    }

    return _sprites[n];
}

//
// clearAllSprites()
//
//  Empties the animator of all sprites.
//  This is often useful when an environment resets
//
public function clearAllSprites():Void {

    // remove all sprites in animator
    while (_spriteList.length > 0) {
        clearSprite(_spriteList[0]);
    };
}

//
// releaseUpdate()
//
//  When an excessive amount of time passes between updates
//  it is necessary to ignore the break, otherwise a very visible
//  jump is seen in animation. By setting the _updatePrev to 0 the
//  animator knows to do the next update with a minimal
//  time_passed value.
//
public function releaseUpdate():Void {
    _updatePrev = 0;
}

//
// clearSprite()
//
//  Clears a specific sprite from the animator, removing it from
//  both the _sprites list and the _spriteList and deleting its
//  attached movie from the main movieclip.
//
//   args:
//      n  -- channel number o the sprite to be removed
//
public function clearSprite(n:Number):Void {

    // find and delete from sprite list
    for(var i:Number = 0; i < _spriteList.length; ++i) {
        if (_spriteList[i] == n) {
            _spriteList.splice(i, 1);
            break;
        }
```

```
    }

    var spriteRoot:MovieClip = _sprites[n].path;
    spriteRoot[n].removeMovieClip();
    delete _sprites[n];
}

//
// reserve()
//
//  Specifies a sprite channel as reserved and not to be recycled
//
//   args:
//          n -- channel to be reserved
//
public function reserve(n:Number) { _reserved.push(n); }

//
// notReserved()
//
//  Checks if a channel is marked as reserved
//
//   args:
//          n -- channel to be checked
//
//   return -- true if channel is not reserved
//
private function notReserved(n:Number):Boolean {

    // check n is not on reserved list
    for (var i:Number = 0; i < _reserved.length; ++i) {
        if (_reserved[i] == n) { return false; }
    }
    return true;
}

//
// message passing from sprite object to the controlling object
//

//
// collision()
//
//  Passes collison messages to the controlling object
//
//   args:
//      src -- source sprite channel that triggered the call
//      trg -- target sprite channel that was collided with
//      str -- string describing the type of collision
//
public function collision(src:Number, trg:Number,
         str:String):Void {
    _obj.collision(src, trg, str);
}

//
// deactivated()
```

```
//
//  Passes deactivated messages to the controlling object
//
//   args:
//      n -- sprite channel number that deactivated
//
public function deactivated(n:Number):Void {
    _obj.deactivated(n);
}

//
// processes that can be performed on sprites
//

//
// goToLocAtSpd()
//
//  Animate sprite to a point at set speed
//
//   args:
//         n -- channel number of the sprite
//      dest -- point destination where the sprite is to move to
//       spd -- velocity at which the sprite should move, as
//                measured in pixels per tick (1/60ths of a
//                second), this number is usually small
//
public function goToLocAtSpd(n:Number, dest:Point2D,
         spd:Number):Void {

    if (_sprites[n] == undefined) {
        // no such sprite
        return;
    }

    var pt:Point2D = dest.clone();
    pt.subtract(_sprites[n].loc);
    var angle:Number = Utils.ptToAngle(pt);
    _sprites[n].vel  = Utils.angleToPt(angle, spd);
    _sprites[n].dest = dest;

    // if inactive then activate.
    if (_sprites[n].active == 0) { _sprites[n].active = 1; }
}

//
// goToLocInTme()
//
//  Animate sprite to point in set time
//
//   args:
//         n -- channel number of the sprite
//      dest -- point destination where the sprite is to move to
//       spd -- duration of the animation measured in
//                ticks (1/60ths of a second)
//
public function goToLocInTme(n:Number, dest:Point2D,
         t:Number):Void {
```

```
    var vel:Point2D = dest.clone();
    vel.subtract(_sprites[n].loc);
    vel.divide(t);

    _sprites[n].vel  = vel;
    _sprites[n].dest = dest;
}

//
// rotateInTime()
//
//  Rotate sprite to an angle in specified time
//
//   args:
//             n -- channel number of the sprite
//  targetAngle -- Angle in degrees to which sprite should rotate
//             t -- duration of the rotation measured in
//                  ticks (1/60ths of a second)
//
public function rotateInTime(n:Number, targetAngle:Number,
         t:Number):Void {

    // keep angle between 0 and 360
    targetAngle = Utils.cleanAngle(targetAngle);

    var vel:Number;
    var startAngle = _sprites[n].angle;

    // find the difference in both directions
    var diffA:Number = (targetAngle - startAngle);
    if (startAngle < targetAngle) {
        startAngle += 360;
    } else { targetAngle += 360; }
    var diffB:Number = (targetAngle - startAngle);

    var vel:Number = ((Math.abs(diffA) <
                       Math.abs(diffB)) ? diffA : diffB)/t;

    _sprites[n].angularVel  = vel;
    _sprites[n].targetAngle = targetAngle;
}

//
// scaleInTime()
//
//  Scale sprite to a size in time
//
//   args:
//           n -- channel number of the sprite
//       scale -- Scale to which sprite should expand or contract to
//           t -- duration of the scale measured in
//                ticks (1/60ths of a second)
//
public function scaleInTime(n:Number, scale:Number, t:Number) {
    var vel:Number = (scale - _sprites[n].scale)/t;
    _sprites[n].scaleVel    = vel;
```

```
    _sprites[n].targetScale = scale;
}

//
// addDropShadow()
//
//  Adds a drop shadow filter to an animating sprite
//
//   args:
//          n -- channel number of the sprite
//   distance -- distance of the drop shadow
//
public function addDropShadow(n:Number, distance:Number) {
    var mc:MovieClip = _sprites[n].movieClip;
    var dropFilter = new flash.filters.DropShadowFilter();
    if (distance != undefined) { dropFilter.distance = distance; }
    var filters:Array = mc.filters;
    filters.push(dropFilter);
    mc.filters = filters;
}

//
// addBevel()
//
//  Adds a bevel filter to an animating sprite
//
//   args:
//          n -- channel number of the sprite
//   distance -- distance of the bevel
//  blurXandY -- amount to blur bevel in x and y
//   strength -- strength of the bevel
//    quality -- quality of the bevel (1 = low, 2 = medium,
//               3 = high)
//    filters -- if deffined then the bevel is added to
//               the filter array list instead of the sprite
//
//  return -- returns the filter array list (if deffined)
//
public function addBevel(n:Number,
                         distance:Number, blurXandY:Number,
                         strength:Number, quality:Number,
                                 filters:Array):Array {

    var bevelFilter = new flash.filters.BevelFilter();
    if (distance != undefined) {
        bevelFilter.distance = distance;
    }
    if (blurXandY != undefined) {
        bevelFilter.blurX = blurXandY;
        bevelFilter.blurY = blurXandY;
    }
    if (strength != undefined) {
        bevelFilter.strength = strength;
    }
    if (quality != undefined) {
        bevelFilter.quality = quality;
    }
```

```
    if (filters == undefined) {
        var mc:MovieClip = _sprites[n].movieClip;
        filters = mc.filters;
        filters.push(bevelFilter, 4);
        mc.filters = filters;
    } else {
        filters.push(bevelFilter, 4);
    }

    return filters;
}

//
// addGlow()
//
//  Adds a glow filter to an animating sprite
//
//   args:
//          n -- channel number of the sprite
//      color -- color of the glow
//  blurXandY -- amount to blur glow in x and y
//   strength -- strength of the glow
//    quality -- quality of the glow (1 = low, 2 = medium,
//               3 = high)
//    filters -- if deffined then the glow is added to
//               the filter array list instead of the sprite
//
//  return -- returns the filter array list (if deffined)
//
public function addGlow(n:Number,
                        color:Number, blurXandY:Number,
                        strength:Number, quality:Number,
                        filters:Array):Array {

    var glowFilter = new flash.filters.GlowFilter();
    if (color != undefined) { glowFilter.color = color; }
    if (blurXandY != undefined) {
        glowFilter.blurX = blurXandY;
        glowFilter.blurY = blurXandY;
    }
    if (strength != undefined) {
        glowFilter.strength = strength;
    }
    if (quality != undefined) {
        glowFilter.quality = quality;
    }

    if (filters == undefined) {
        var mc:MovieClip = _sprites[n].movieClip;
        var filters:Array = mc.filters;
        filters.push(glowFilter);
        mc.filters = filters;
    } else {
        filters.push(glowFilter);
    }
```

```
    return filters;
}

//
// removeFilter()
//
//  Removes the last applied filter
//
//   args:
//          n -- channel number of the sprite
//
//  return -- returns the filters array of the sprite
//
public function removeFilter(n:Number):Array {

    // removes the last filter applied to a sprite
    var mc:MovieClip = _sprites[n].movieClip;
    var filters:Array = mc.filters;
    filters.pop();
    mc.filters = filters;

    return filters;
}

//
// accessors
public function get tick():Number {
    return _tick;
}
public function get spritelist():Array {
    return _spriteList;
}
public function get path():MovieClip {
    return _path;
}
public function getDefined(n:Number):Boolean {
    if (_sprites[n] == undefined) {
        return false;
    } else {
        return true;
    }
}
public function getActive(n:Number):Number {
    if (_sprites[n] == undefined) {
        return 0;
    } else {
        return _sprites[n].active;
    }
}
public function getClip(n:Number):String {
    return _sprites[n].clip;
}
public function getLoc(n:Number):Point2D {
    return _sprites[n].loc;
}
public function getVel(n:Number):Point2D {
    return _sprites[n].vel;
```

```
}
public function getAngle(n:Number):Number {
    return _sprites[n].angle;
}
public function getRadius(n:Number):Number {
    return _sprites[n].radius;
}
public function getTag(n:Number) {
    return _sprites[n].tag;
}
public function getTTime(n:Number):Number {
    return _sprites[n].tTime;
}
public function getCycleType(n:Number):String {
    return _sprites[n].cycleType;
}
public function getCycleDirection(n:Number):Number {
    return _sprites[n].cycleDirection;
}
public function getMovieClip(n:Number):MovieClip {
    return _sprites[n].movieClip;
}
public function getSprite(n:Number):Sprite {
    return _sprites[n];
}

//
// mutators
public function setClip(n:Number, clip:String):Void {
    var spriteRoot:MovieClip = _sprites[n].path;
    spriteRoot.removeMovieClip();
    spriteRoot.attachMovie(clip, n.toString(), _depth + n);
    _sprites[n].movieClip = spriteRoot[n];

    // tell the sprite the new clip linkage name
    _sprites[n]._clip = clip;
}
public function setActive(n:Number, val:Number):Void {
    _sprites[n].active = val;
}
public function setLoc(n:Number, val:Point2D):Void {
    _sprites[n].loc = val;
}
public function setVel(n:Number, val:Point2D):Void {
    _sprites[n].vel = val;
}
public function setLocXY(n:Number, X:Number, Y:Number):Void {
    _sprites[n].setLocXY(X, Y);
}
public function setVelXY(n:Number, X:Number, Y:Number):Void {
    _sprites[n].setVelXY(X, Y);
}
public function setAngle(n:Number, val:Number):Void {
    _sprites[n].angle = val;
}
public function setRadius(n:Number, val:Number):Void {
    _sprites[n].radius = val;
```

```
    }
    public function setFriction(n:Number, val:Number):Void {
        _sprites[n].friction = val;
    }
    public function setMaxVel(n:Number, val:Number):Void {
        _sprites[n].maxVel = val;
    }
    public function setWallType(n:Number, val:String):Void {
        _sprites[n].wallType = val;
    }
    public function setTag(n:Number, val):Void {
        _sprites[n].tag = val;
    }
    public function setFrame(n:Number, val:Number):Void {
        _sprites[n].frame = val;
    }
    public function setCycleType(n:Number, val:String):Void {
        _sprites[n].cycleType = val;
    }
    public function setFPerSec(n:Number, val:Number):Void {
        _sprites[n].fPerSec = val;
    }
    public function setDisplayTop(n:Number, val:Number):Void {
        _sprites[n].displayTop = val;
    }
    public function setDisplayLeft(n:Number, val:Number):Void {
        _sprites[n].displayLeft = val;
    }
    public function setDisplayBottom(n:Number, val:Number):Void {
        _sprites[n].displayBottom = val;
    }
    public function setDisplayRight(n:Number, val:Number):Void {
        _sprites[n].displayRight = val;
    }
    public function setTTime(n:Number, val:Number):Void {
        _sprites[n].tTime = val;
    }
    public function setCollision(n:Number, val:Number, s:String):Void {
        _sprites[n].setCollision(val, s);
    }
    public function clearCollisions(n:Number):Void {
        _sprites[n].clearCollisions();
    }
    public function reverseCycle(n:Number):Void {
        _sprites[n].reverseCycle();
    }
    public function set path(val:MovieClip):Void {
        _path = val;
    }
}
```

Sprite.as

```
////////////////////////////////////////////////////////////////////////
//
//  Sprite.as
//
//       Author: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: The Sprite object contains numerous member properties
//               that dictate how a sprite should behave when
//               animating. The update() method called by Animatem is
//               divided into five procedures for readability,  1)
//               physics, 2) cycles, 3) walls, 4) collisions, and, 5)
//               drawing. The whole object is really one long update
//               method, utilizing a large set of member variables
//               that determine behavior.  Accessors and mutators are
//               provided for all variables to give easy direct
//               access and allow highly interactive manipulation.
//
//       Method:
//
//          Sprite() - Constructor
//          update() - Do/calculate everything needed to update
//                     this sprite.
//       doPhysics() - Deal with physics on the sprite, specifically,
//                     friction, maximum velocity, velocity, scaling
//                     velocity, angular velocity, destination points,
//                     and termination time (time to live).
//        doCycles() - Deal with cycling frames.
//                     Sprites can animate through fame "cells" forward
//                     and backwards and at a rate specified by
//                     _tPerFrame which is really a specified Frames
//                     Per Second (FPS).  This is a powerful feature as
//                     traditionally Flash restricts playing of movie
//                     frames to strictly forward and only at the
//                     FPS defined by the whole movie.
//       capCycles() - Ensure that a frame cycle "cell" does not exceed
//                     cycle end or start.
//         doWalls() - Deal with sprite boundaries ("walls"). The
//                     sprite has a display rect that it should animate
//                     within, if it exceeds this Rect then something
//                     should happen, as defined by _wallType.
//    doCollisions() - Deal with collision detection buy searching
//                     through the _collisionList for sprites flagged
//                     as a collision hazard.
//          doDraw() - Once all factors have been taken into account,
//                     tell Flash to actually draw the sprite at a
//                     location with a rotation, using a scale, at at a
//                     specific frame.
// clearCollisions() - Clear all collisions from this sprite.
//                     Useful after a sprite has fulfilled its purpose.
//      deactivate() - Sprite has triggered a deactivation, but cannot
//                     deactivate unless all deactivation triggers have
```

```
//
//                      been satisfied.
//
//   Notes:
//

class Sprite {

    // instance members
    // controlling ancestor object
    private var _a:Animatem;

    // 1 = active, 0 = inactive, -1 = active but no collisions
    private var _active:Number          = 0;

    // sprite channel number, also determines depth
    private var _number:Number          = -1;

    // name of movie clip (linkage) associated with this sprite
    private var _clip:String;

    // movie clip associated with this sprite
    private var _movieClip:MovieClip;

    // path or "root" of this movieClip
    private var _path:MovieClip;

    // location of sprite x, y
    private var _loc:Point2D;

    // velocity of sprite x, y
    private var _vel:Point2D;

    // facing angle in degrees
    private var _angle:Number           = 0;

    // angular velocity;
    private var _angularVel:Number      = 0;

    // target angle, used when animating to a specific angle
    private var _targetAngle:Number     = -1;

    // everything starts at 100%
    private var _scale:Number           = 100;

    // velocity of scaling
    private var _scaleVel:Number        = 0;

    // target scale, used when animating to a specific scale
    private var _targetScale:Number     = -1;

    // surface friction on object
    private var _friction:Number        = 0;

    // height(y) of sprite
    private var _h:Number               = 0;
```

```
// width(x) of sprite
private var _w:Number               = 0;

// time of last update
private var _updateTime:Number      = 0;

// time passed since previous update
private var _timePassed:Number      = 0;

// radius used for collision detection
private var _radius:Number          = 0;

// use to cap movement
private var _maxVel:Number          = 0;

// number of images in cycle
private var _cycleSize:Number       = 0;

// image currently displayed
private var _frame:Number           = 1;

// time between frame changes
private var _tPerFrame:Number       = 0;

// number of triggered operations that can deactivate sprite
private var _deactivateFlags:Number = 0;

// how sprite handles walls
private var _wallType:String        = "NONE";

// What to do at the end of a cycle
private var _cycleType:String       = "NONE";

// misc. label, tag is usually used in conjunction with collisions
private var _tag:String             = "NONE";

// list of collisions to check for
private var _collisionList:Array;

// display area for this sprite
private var _display:Rect;

// terminate time when which sprite deactivates
private var _tTime:Number           = 0;

// destination point at which sprite deactivates
private var _dest:Point2D;



//
// constructor
//
//  Constructs Sprite
//
//   args:
//      animator -- The animatem object controlling this sprite
```

```
//             n -- the channel number of this sprite
//     movieClip -- the MovieClip graphic used for this sprite
//          path -- path to the movieClip graphic
//             x -- location of sprite in x
//             y -- location of sprite in y
//        active -- 1 is active,
//                  0 is inactive,
//                 -1 is active but not detecting collisions
//
public function Sprite(animator:Animatem, n:Number,
         movieClip:MovieClip, path:MovieClip, x:Number,
                        y:Number, active:Number) {

    if (x == undefined) { x = 0; }
    if (y == undefined) { y = 0; }
    if (active == undefined) { active = 1; }

    _display        = new Rect(0, 0, Stage.width, Stage.height);
    _a              = animator;
    _number         = n;
    _path           = path;
    _loc            = new Point2D(x, y);
    _vel            = new Point2D(0, 0);
    _collisionList  = new Array();
    this.movieClip  = movieClip;
    _active         = active;

    if ( _active ) {
        // place sprite on the stage for first time
        update(getTimer() / _a.tick, 0);
    } else {
        // place sprite (usually offscreen)
        _movieClip._x = x;
        _movieClip._y = y;
        _movieClip.gotoAndStop(Math.round(_frame));
    }
}

//
// update()
//
//  Do/calculate everything needed to update this sprite.
//
//   args:
//      updateTime  -- the current update time as specified by
// Animatem
//      timePassed  -- time passed since the last Animatem update
//
public function update(updateTime:Number, timePassed:Number):Void {

    _updateTime = updateTime; // save updatetime
    _timePassed = timePassed;

    doPhysics();              // handle physics of movement
    doCycles();               // handle cycling frames
    doWalls();                // check display limit
    doCollisions();           // handle collisions
```

```
    doDraw();
}

//
// doPhysics()
//
//  Deal with physics on the sprite, specifically,
//  friction, maximum velocity, velocity,
//  scaling velocity, angular velocity, destination points,
//  and termination time (time to live).
//
private function doPhysics():Void {

    // surface friction
    var friction:Number = _friction * _timePassed;
    if (friction != 0) {

        var nx:Number = Math.abs(_vel.x) - friction;
        var ny:Number = Math.abs(_vel.y) - friction;

        // FLASH BUG if = 0.. ?
        if (nx < 0) { nx = 0.00000000000000000000001; }

        // FLASH BUG if = 0.. ?
        if (ny < 0) { ny = 0.00000000000000000000001; }


        if (_vel.x < 0) { nx = -nx; }
        if (_vel.y < 0) { ny = -ny; }

        _vel.x = nx;
        _vel.y = ny;
    }

    // limit velocity to _maxVel
    if (_maxVel && _vel.length > _maxVel) {
        var angle:Number = Utils.ptToAngle(_vel);
        _vel = Utils.angleToPt(angle, _maxVel);
    }

    // add velocity
    _loc.x += _vel.x*_timePassed;
    _loc.y += _vel.y*_timePassed;

    // add scaling velocity
    _scale += _scaleVel*_timePassed;

    // check if at target scale
    if (_targetScale != -1) {

        // check if reached target scale
        if ( (_scaleVel > 0 && _scale >= _targetScale) ||
             (_scaleVel < 0 && _scale <= _targetScale) ||
              _scaleVel == 0) {
            _scale       = _targetScale;
            _scaleVel    = 0;
            _targetScale = -1;
```

```
            deactivate();
        }
    }


    // add angular velocity and check if at a target angle
    var nAngle:Number = _angle + _angularVel*_timePassed;
    if (_targetAngle != -1 ) {

        // adjust target due to looping nature of degrees, 0 = 360
        // and vis-a-vis
        var target:Number = _targetAngle;
        if (_angularVel > 0 && target < _angle) {
            target += 360;
        } else if (_angularVel < 0 && target > _angle) {
            target -= 360;
        }
        if ( ( _angle <= target && target <= nAngle) ||
             ( nAngle <= target && target <= _angle) ) {
            // reached target angle
            nAngle = _targetAngle;
            _angle = _targetAngle;
            _angularVel  = 0;
            _targetAngle = -1;

            deactivate();
        }
    }
    // keep angle between 0 and 360
    nAngle = Utils.cleanAngle(nAngle);
    _angle = nAngle;

    // check if at destination point
    if (_dest != undefined) {
        if ((_vel.x < 0 && _loc.x < _dest.x) ||
            (_vel.x > 0 && _loc.x > _dest.x)) {
            _loc.x = _dest.x; _vel.x = 0;
        }
        if ((_vel.y < 0 && _loc.y < _dest.y) ||
            (_vel.y > 0 && _loc.y > _dest.y)) {
            _loc.y = _dest.y; _vel.y = 0;
        }
        if (_loc.x == _dest.x && _loc.y == _dest.y) {
            _vel.x = 0; _vel.y = 0;
            _dest = undefined; deactivate();
        }
    }

    // check for termination time
    if ( _tTime && getTimer() > _tTime) {
        _tTime = 0;
        deactivate();
    }


}
```

```
//
// doCycles()
//
//  Deal with cycling frames
//  Sprites can animate through fame "cells" forward and
//  backwards and at a rate specified by _tPerFrame which is really
//  a a specified Frames Per Second (FPS)
//  This is a powerful feature as traditionally Flash restricts
//  playing of movie frames to strictly forward and only at the
//  FPS defined by the whole movie.
//
private function doCycles():Void {

    // special case for NONE
    if (_cycleType == "NONE") { return; }

    // special case for ANGLE
    if (_cycleType == "ANGLE" || _cycleType == "S_ANGLE") {

        var target:Number = Utils.angleToCell(_angle, _cycleSize);

        if (_cycleType == "S_ANGLE") {
            // SHOULD BE UTILIZING _tPerFrame as angular velocity
            _frame += Utils.incOrDec(_frame, target, _cycleSize);

            if (_frame <= 0) {
                _frame = _cycleSize;
            } else if (_frame > _cycleSize) {
                _frame = 1;
            }
        } else {
            _frame = target;
        }

    // time to change cells?
    } else if (_tPerFrame) {

        // get the next cell
        _frame = _frame + _timePassed/_tPerFrame;

        if (_frame > _cycleSize || _frame <= 0) {
            if (_cycleType == "END") {
                capCycles();
                _cycleType == "NONE";
            } else if (_cycleType == "WRAP") {
                if (_frame > _cycleSize) {
                    _frame -= _cycleSize;
                } else if (_frame <= 0) {
                    _frame += _cycleSize;
                }
                capCycles();
            } else if (_cycleType == "REVERSE") {
                capCycles();
                _tPerFrame = - _tPerFrame;
            } else if (_cycleType == "DEACTIVATE") {
                capCycles();
```

```
                deactivate();
            }
        }
    }
}

//
// capCycles()
//
//  Ensure that a frame cycle "cell" does not exceed
/// cycle end or start
//
private function capCycles():Void {
    if (_frame <= 0) {
        _frame = 1;
    } else if (_frame > _cycleSize) {
        _frame = _cycleSize;
    }
}

//
// doWalls()
//
//  Deal with sprite boundaries ("walls")
//  The sprite has a display rect that it should animate within,
//  if it exceeds this Rect then something should happen, as
//  defined by _wallType
//
private function doWalls():Void {

    if (_wallType == "NONE") { return; }

    if (_wallType == "REFLECT") {

        if (_loc.x > _display.x2 - _w/2) {
            _loc.x = _display.x2 - _w/2;
            _vel.x = 0 - _vel.x;
        } else if (_loc.x < _display.x1 + _w/2) {
            _loc.x = _display.x1 + _w/2;
            _vel.x = 0 - _vel.x;
        }
        if (_loc.y > _display.y2 - _h/2) {
            _loc.y = _display.y2 - _h/2;
            _vel.y = 0 - _vel.y;
        } else if (_loc.y < _display.y1 + _h/2) {
            _loc.y = _display.y1 + _h/2;
            _vel.y = 0 - _vel.y;
        }
    } else if (_wallType == "WRAP") {

        if (_loc.x > _display.x2 + _w/2 + 1) {
            _loc.x = _display.x1 - _w/2;
        } else if (_loc.x < _display.x1 - _w/2 - 1) {
            _loc.x = _display.x2 + _w/2;
        }

        if (_loc.y > _display.y2 + _h/2 + 1) {
```

```
            _loc.y = _display.y1 - _h/2;
        } else if (_loc.y < _display.y1 - _h/2 - 1) {
            _loc.y = _display.y2 + _h/2;
        }
    } else if (_wallType == "DEACTIVATE") {

        if (_loc.x < _display.x1 ||     // USE INSIDE??
            _loc.x > _display.x2 ||
            _loc.y < _display.y1 ||
            _loc.y > _display.y2) {
            deactivate();
        }
    } else if (_wallType == "BLOCK") {

        if (_loc.x > _display.x2 - _w/2) {
            _loc.x = _display.x2 - _w/2;
        } else if (_loc.x < _display.x1 + _w/2) {
            _loc.x = _display.x1 + _w/2;
        }
        if (_loc.y > _display.y2 - _h/2) {
            _loc.y = _display.y2 - _h/2;
        } else if (_loc.y < _display.y1 + _h/2) {
            _loc.y = _display.y1 + _h/2;
        }
    }
}

//
// doCollisions()
//
//  Deal with collision detection buy searching through
//  the _collisionList for sprites flagged as a collision hazard.
//
private function doCollisions():Void {

    // only check collisoins with _active == 1 sprites
    if (_active != 1) { return; }

    // beware, _collisionList can change size during loop
    for (var i:Number = 0; i < _collisionList.length; ++i) {

        var collision:Pair = _collisionList[i];
        var spr:Sprite = _a.getSprite(collision.number);

        if (spr.active > 0 &&
            _loc.distance(spr._loc) < _radius + spr.radius) {

            // send collision detected
            _a.collision(_number, collision.number,
                     collision.string);
        }
    }
}

//
// doDraw()
//
```

```
//  Once all factors have been taken into account,
//  tell Flash to actually draw the sprite at a location
//  with a rotation, using a scale, at at a specific frame.
//
private function doDraw():Void {
    // time to draw the sprite, but first check if still active,
    // the sprite could have been switched off due to a collision
    // or similar

    if (_active) {

        // draw the sprite-- Note: rounded numbers should give
        // faster drawing
        _movieClip._x = _loc.x; // Math.round(_loc.x)
        _movieClip._y = _loc.y; // Math.round(_loc.y)
        _movieClip._rotation = _angle;
        _movieClip._xscale   = _scale;
        _movieClip._yscale   = _scale;
        _movieClip.gotoAndStop(Math.round(_frame));
    }
}

//
// clearCollisions()
//
//  Clear all collisions from this sprite
//  Useful after a sprite has fulfilled its purpose
//
public function clearCollisions():Void {
    delete _collisionList;
    _collisionList  = new Array();
}

//
// deactivate()
//
//  Sprite has triggered a deactivation, but cannot deactivate
// unless
//  all deactivation triggers have been satisfied
//
private function deactivate():Void {
    --_deactivateFlags;
    // if out of deactiveate flags then, well, deactivate
    if (_deactivateFlags <= 0) {

        this.active = 0;
        _deactivateFlags = 0;
        _a.deactivated(_number);
    }
}

// accessors
public function get number():Number       { return _number; }
public function get active():Number       { return _active; }
public function get angle():Number        { return _angle; }
public function get scale():Number        { return _scale; }
public function get loc():Point2D         { return _loc.clone(); }
```

```
public function get vel():Point2D         { return _vel.clone(); }
public function get display():Rect        {return_display.clone();}
public function get radius():Number       { return _radius; }
public function get tag():String          { return _tag; }
public function get cycleType():String    { return _cycleType; }
public function get tTime():Number        { return _tTime; }
public function get dest():Point2D        { return _dest; }
public function get clip():String         { return _clip; }
public function get movieClip():MovieClip { return _movieClip; }
public function get angularVel():Number   { return _angularVel; }
public function get scaleVel():Number     { return _scaleVel; }
public function get path():MovieClip      { return _path; }
public function get frame():Number        { return _frame; }
public function get cycleSize():Number    { return _cycleSize; }


// mutators
public function set active(val:Number):Void {
    // ensure final positions before deactivating,
    if (val == 0) { doDraw(); }

    _active = val;
}
public function set vel(val:Point2D):Void {
    _vel = val.clone();
}
public function set display(val:Rect):Void {
    _display = val.clone();
}
public function set friction(val:Number):Void {
    _friction = val;
}
public function set radius(val:Number):Void {
    _radius = val;
}
public function set maxVel(val:Number):Void {
    _maxVel = val;
}
public function set wallType(val:String):Void {
    if (_wallType  != "DEACTIVATE" && val == "DEACTIVATE") {
        ++_deactivateFlags;
    } _wallType  = val;
}
public function set tag(val):Void {
    _tag = val;
}
public function set cycleType(val:String):Void {
    if (_cycleType != "DEACTIVATE" && val == "DEACTIVATE") {
        ++_deactivateFlags;
    } _cycleType = val;
}
public function set frame(val:Number):Void {
    _frame = val;
    _movieClip.gotoAndStop(_frame);
}
public function set tTime(val:Number):Void {
    if (_tTime == 0) {
```

```
            ++_deactivateFlags;
        }
        _tTime = val;
    }
    public function set dest(val:Point2D):Void {
        if (_dest == undefined) {
            ++_deactivateFlags;
        }
        _dest = val;
    }
    public function set targetScale(val:Number):Void {
        if (_targetScale == -1) {
            ++_deactivateFlags;
        } _targetScale = val;
    }
    public function set targetAngle(val:Number):Void {
        if (_targetAngle == -1) {
            ++_deactivateFlags;
        }
        _targetAngle = Utils.cleanAngle(val);
    }
    public function set angularVel(val:Number):Void {
        _angularVel  = val;
    }
    public function set scaleVel(val:Number):Void {
        _scaleVel = val;
    }
    public function set clip(val:String):Void {
        _a.setClip(_number, val);
    }
    public function set loc(val:Point2D):Void {
        _loc = val.clone();
        _movieClip._x = _loc.x;
        _movieClip._y = _loc.y;
    }
    public function set fPerSec(val:Number):Void {
        _tPerFrame = Utils.FPS_to_Ticks(val);
        if (_cycleType == "NONE") {
            _cycleType = "WRAP";
        }
    }
    public function set angle(val:Number):Void {
        _angle = val;
        if (_angle >= 360) {
            _angle -= 360;
        } else if (_angle < 0) {
            _angle += 360;
        }
        // _cycleType "ANGLE" and "S_ANGLE" adjust the
        // _movieClip._rotation slowly
        if ( ! (_cycleType == "ANGLE" || _cycleType == "S_ANGLE")) {
            _movieClip._rotation = _angle;
        }
    }
    public function set scale(val:Number):Void {
        _scale = val;
        _movieClip._xscale = _scale;
```

```
        _movieClip._yscale = _scale;
    }
    public function set movieClip(movieClip:MovieClip):Void {
        _movieClip = movieClip;
        _cycleSize = movieClip._totalframes;
        _h         = movieClip._height;
        _w         = movieClip._width;
        _frame     = 1;
        _cycleType = "NONE";          // default cycleType
    }
    public function reverseCycle():Void {
        _tPerFrame = -_tPerFrame;
    }
    public function setCollision(n:Number, s:String):Void {
        _collisionList.push(new Pair(n, s));
    }
    public function setLocXY(x:Number, y:Number):Void {
        _loc = new Point2D(x,y);
    }
    public function setVelXY(x:Number, y:Number):Void {
        _vel = new Point2D(x,y);
    }
}
```

PlaySnd.as

```
////////////////////////////////////////////////////////////////////////
//
//  PlaySnd.as
//
//      AUTHOR: Russell Lowke
//              © Copyright 2006 Russell Lowke
//              All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: PlaySnd uses an update() method called by the
//               program's main loop. PlaySnd maintains a list of
//               loaded Sound objects ready to be triggered at any
//               time.  The list of sounds to be loaded are specified
//               as a parameter in PlaySnd's constructor.
//               PlaySnd's main function is to allow for delayed
//               and staggered sounds, a feature not supplied by
//               Flash. Looping sounds are also catered for as well
//               as specific sound volume.
//
//      Methods:
//
//         PlaySnd() - Constructor
//            play() - Sets a sound to play using a specific volume,
//                     delay and stagger
//            loop() - Sets a sound to loop
//          update() - Called by the main loop of the program
//                     monitors sound progress
//    triggerSound() - Triggers sound at a volume
//
//  Notes: I'm not using the inbuilt loop function because it does
//         not work with streaming audio.
//

class PlaySnd {

    // instance members
    private var _sounds:Array;      // list of loaded sounds
    private var _cue:Array;         // list of cued sounds to be played
                                    //  at a certain time
    private var _looping:String;    // name of looping audio, if any


    //
    // constructor
    //
    //  Builds _sounds list of Sound objects as specified by arg
    //  soundList Initializes _cue to be used for sounds waiting to
    //  be played.
    //
    //   args:
    //       soundList  -- String list of sounds to be loaded
    //
    public function PlaySnd(soundList:Array) {
```

```
    _sounds = new Array();
    _cue    = new Array();

    // load sounds
    for (var i:String in soundList) {
        var snd:Sound = new Sound();
        snd.attachSound(soundList[i]);
        _sounds[soundList[i]] = snd;
    }
}

//
// play()
//
//  Sets a sound to be play using a specific volume, delay and
// stagger
//  If a delay or stagger is specified then the sound is added to
// _cue
//   to be played at a specific time.
//
//   args:
//          sound   -- name of sound to be played
//       atVolume   -- volume to play the sound 0-100 %
//          delay   -- in seconds, delay before sound is played
//        stagger   -- in seconds, maximum randomized stagger
// before sound
//                       is played.
//
public function play(sound:String,
                     atVolume:Number,
                     delay:Number,
                     stagger:Number):Void {

    // get time this sound was triggered
    var time:Number   = getTimer();
    var atTime:Number = time;

    // calculate the play time cue for sound by,
    //  converting delay from seconds to millionths of a second
    //  and adding a randomized stagger time
    //  that's also to millionths of a second
    if (delay != undefined) {
        atTime += delay*1000;
    }
    if (stagger != undefined) {
        atTime += Utils.randomInt(1, stagger*1000);
    }

    if(atTime <= time) {
        _sounds[sound].setVolume(atVolume);
        _sounds[sound].start();

    } else {
        // push an anonymous object with the
        //  sound, atTime and vloume as parameters
        _cue.push({sound:sound, atTime:atTime, atVolume:atVolume})
    }
```

```
}

//
// loop()
//
//  Sets a specific sound to loop
//
//   args:
//          sound   -- name of sound to be played
//       atVolume   -- volume to play the sound 0-100 %
//
public function loop(sound:String, atVolume:Number):Void {

    // switch off previous looping audio
    if (_looping!= undefined && _looping != sound) {
        _sounds[_looping].stop();
    }

    // set new looping audio
    if (sound != undefined) {
        _looping = sound;
        triggerSound(sound, atVolume);
    }
}

//
// update()
//
//  Called by the main loop of the program.  Monitors _cue list
//   checking for specific times to trigger delayed sounds
//
public function update():Void {

    // get the time at this instant
    var time:Number = getTimer();

    // check delay cue list for sounds to play
    for (var i:Number = 0; i < _cue.length; ++i) {
        if (_cue[i].atTime < time) {

            // time to play the sound
            triggerSound(_cue[i].sound, _cue[i].atVolume);
            _cue.splice(i, 1);
            --i;

        }
    }

    // check for end of looping audio
    if (_looping != undefined) {
        if (_sounds[_looping].position >=
            _sounds[_looping].duration) {
            triggerSound(_looping);
        }
    }
}
```

```
    //
    // triggerSound()
    //
    //  Trigger sound at a volume
    //
    //   args:
    //          sound   -- name of sound to be played
    //       atVolume   -- volume to play the sound 0-100 %
    //
    private function triggerSound(sound:String, atVolume:Number):Void {

        // handle volume setting
        if (atVolume != undefined) {
            _sounds[sound].setVolume(atVolume);
        }
        _sounds[sound].start();
    }
}
```

MMatrix.as

```
////////////////////////////////////////////////////////////////////////
//
//  MMatrix.as
//
//       Author: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: Base class used to store 2D information as a matrix.
//               For the convenience of matrix multiplication, data
//               is usually broken into columns first then rows.
//
//      Methods:
//
//         MMatrix() - Constructs MMatrix to s et width and height
//                     using Array data. Data is broken into columns
//                     first, and then rows.
//       congruent() - Tests to see if another matrix is congruent with
//                     this one. Matrices are congruent if their height
//                     and width are the same.
//           equal() - Tests to see if data in other matrix is the same
//                     as the data in this one. Matrices must be
//                     be congruent to be equal.
//             add() - Adds two matrices together or, if val is a
//                     Number, adds that, Number to each cell in this
//                     MMatrix.
//        subtract() - Subtracts a MMatrix from this MMatrix or, if val
//                     is a Number, subtracts that Number from each
//                     cell in this MMatrix.
//        multiply() - Multiplies two matrices together or, if val is a
//                     Number, multiplies that Number from each cell in
//                     this MMatrix. If the width of this matrix equals
//                     the height of the other matrix, then true matrix
//                     multiplication is performed.
//       mMultiply() - Performs true matrix multiplication, to do this
//                     the number of columns in this Matrix must equal
//                     the number of rows in other.
//          divide() - Divides this matrix by another or, if val is a
//                     Number, divides that Number from each cell in
//                     this MMatrix.
//           print() - Prints the rows and columns of this MMatrix
//
//  Notes: Flash does not have method or operator overloading
//         which is why the matrix operations accept an undefined
//         value which is expected to be either another MMatrix or
//         a Number. Perhaps it would be better to have two separate
//         methods one accepting a MMatrix and the other a Number.
//

class MMatrix {

    // instance members
    private var _w:Number;                  // width of Matrix
```

```
private var _h:Number;                  // height of Matrix
private var _data:Array;                // data for Matrix

//
// constructor
//
//  Constructs MMatrix to a set width and height using Array data
//  Data is broken into columns first, and then rows
//
//   args:
//       w  -- width of matrix data
//       h  -- height of matrix data
//    data  -- array of data for matrix, can be a 1D or 2D
//
public function MMatrix(w:Number, h:Number, data:Array) {

    _w = w;         // set width
    _h = h;         // set height

    if (data instanceof Array) {

        if (data[0] instanceof Array) {

            // looks like it's a 2D Array, assign it as is
            _data = data;
        } else {

            // looks like it's a 1D Array, break apart
            _data = new Array(_w);
            for (var x:Number = 0; x < w; ++x) {
                _data[x] = new Array(_h);
                for (var y:Number = 0; y < h; ++y) {
                    _data[x][y] = data[ w * y + x ];
                }
            }
        }
    } else {

        // initialize array and with undefined
        _data = new Array(_w);
        for (var x:Number = 0; x < _w; ++x) {
            _data[x] = new Array(_h);
        }
    }
}


//
// comparison operators
//


//
// congruent()
//
//  Tests to see if another matrix is congruent with this one.
//  Matrices are congruent if their height and width are the same
```

```
//
//   args:
//          other   -- MMatrix being compared
//
public function congruent(other:MMatrix):Boolean {
        return (_w == other._w && _h == other._h);
}

//
// equal()
//
//  Tests to see if data in another matrix is the same as the
//  data in this one. Matrices must (obviously) be congruent.
//
//   args:
//          other   -- MMatrix being compared
//
public function equal(other:MMatrix):Boolean {
    if (! congruent(other)) { return false; }
    for (var x:Number = 0; x < _w; ++x) {
        for (var y:Number = 0; y < _h; ++y) {
            if (_data[x][y] != other._data[x][y]) {  // check data
                return false;
            }
        }
    }
    return true;
}

//
//  matrix operations
//

//
// add()
//
//  Adds two matrices together or, if val is a Number, adds that
//  Number to each cell in this MMatrix.
//
//   args:
//          val     -- MMatrix or Number to be added
//
public function add(val):MMatrix {

    if (val instanceof MMatrix) {       // another MMatrix
        if (congruent(val)) {
            for (var x:Number = 0; x < _w; ++x) {
                for (var y:Number = 0; y < _h; ++y ) {
                    _data[x][y] += val._data[x][y];
                }
            }
        }
    } else {
        for (var x:Number = 0; x < _w; ++x) {
            for (var y:Number = 0; y < _h; ++y ) {
                _data[x][y] += val;
            }
```

```
        }
    }

    return this;
}

//
// subtract()
//
//  Subtracts a MMatrix from this MMatrix or, if val is a Number,
//  subtracts that Number from each cell in this MMatrix.
//
//   args:
//          val     -- MMatrix or Number to be subtracted
//
public function subtract(val):MMatrix {

    if (val instanceof MMatrix) {       // another MMatrix
        if (congruent(val)) {
            for (var x:Number = 0; x < _w; ++x) {
                for (var y:Number = 0; y < _h; ++y ) {
                    _data[x][y] -= val._data[x][y];
                }
            }
        }
    } else {

        for (var x:Number = 0; x < _w; ++x) {
            for (var y:Number = 0; y < _h; ++y ) {
                _data[x][y] -= val;
            }
        }
    }

    return this;
}

//
// multiply()
//
//  Multiplies two matrices together or, if val is a Number,
//  multiplies that Number to each cell in this MMatrix.
//  If the width of this matrix equals the height of the
//  other matrix, then true matrix multiplication is performed.
//
//   args:
//          val     -- MMatrix or Number to be multiplied
//
public function multiply(val):MMatrix {

    if (val instanceof MMatrix) {       // another MMatrix
        // if # of columns equals # of rows in other
        if (_w == val._h) {

            // then perform proper multiplication of matrices
            return mMultiply(val);
```

```
        } else if (congruent(val)) {
            for (var x:Number = 0; x < _w; ++x) {
                for (var y:Number = 0; y < _h; ++y ) {
                    _data[x][y] *= val._data[x][y];
                }
            }
        }
    } else {
        for (var x:Number = 0; x < _w; ++x) {
            for (var y:Number = 0; y < _h; ++y ) {
                _data[x][y] *= val;
            }
        }
    }

    return this;
}

//
// mMultiply()
//
//  Performs true matrix multiplication, to do this the number of
//  columns in this Matrix must equal the number of rows in other.
//
//   args:
//          other   -- MMatrix be multiplied by
//
public function mMultiply(other:MMatrix):MMatrix {
    var height:Number = _h;
    var width:Number  = other._w;
    var r:MMatrix = new MMatrix(width, height);
    var n:Number = 0;
    for (var x:Number = 0; x < width; ++x) {
        for (var y:Number = 0; y < height; r._data[x][y] = n, ++y,
                n = 0 ) {
            for (var i:Number = 0; i < other._h && i < _w; ++i ) {
                n += (_data[i][y] * other._data[x][i]);
            }
        }
    }
    return r;
}

//
// divide()
//
//  Divides this matrix by another or, if val is a Number,
//  divides that Number from each cell in this MMatrix.
//
//   args:
//          val     -- MMatrix or Number to be divided
//
public function divide(val):MMatrix {

    if (val instanceof MMatrix) {       // another MMatrix
        if (congruent(val)) {
            for (var x:Number = 0; x < _w; ++x) {
```

```
                    for (var y:Number = 0; y < _h; ++y ) {
                        _data[x][y] /= val._data[x][y];
                    }
                }
            }
        } else {
            for (var x:Number = 0; x < _w; ++x) {
                for (var y:Number = 0; y < _h; ++y ) {
                    _data[x][y] /= val;
                }
            }
        }

        return this;
    }

    //
    // print()
    //
    //  Prints the rows and columns of this MMatrix
    //
    public function print():Void {
        var pString:String = "";
        for (var y:Number = 0; y < _h; ++y) {
            pString += "[ ";
            for (var x:Number = 0; x < _w; ++x) {
                pString += _data[x][y] + " ";
            }
            pString += "]" + " r";
        }
        trace(pString);
    }

    //
    // accessors
    public function get width():Number {
        return  _w;
    }
    public function get height():Number {
        return  _h;
    }
    public function get data():Array {
        return  _data;
    }
    public function getAt(x:Number, y:Number) {
        return  _data[x][y];
    }

    //
    // mutators
    public function setAt(x:Number, y:Number, val) {
        _data[x][y] = val;
    }
}
```

Grid.as

```
////////////////////////////////////////////////////////////////////////
//
//  Grid.as
//
//       Author: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: Grid object extends MMatrix adding functionality to
//               define cell blocks with a width and height that can
//               be referenced to screen coordinates and vice versa.
//               This is very useful for implementing game boards and
//               terrain.
//
//       Method:
//
//            Grid() - Constructs Grid, defining cell size, location,
//                     height and width.
//     ptToGridLoc() - Convert a pt location, usually a screen
//                     location, to a grid cell location.
//     gridToPtLoc() - Convert a grid cell location to a local
//                     coordinate.
//
//        Notes: Attempting to access or mutate grid cells outside the
//               grid will result in the nearest edge cell being
//               returned/modified
//

class Grid extends MMatrix {

    // instance members
    // symbols wide that the grid is
    private var _w:Number;

    // symbols high that the grid is
    private var _h:Number;

    private var _block_w:Number;        // width of each location
    private var _block_h:Number;        // height of each location
    private var _dim:Point2D;           // width & height dimension
    // 1/2 width & height dimension, used in calculations
    private var _half:Point2D;


    // X modifier used so center of grid is 0,0
    private var _xModifier;

    // Y modifier used so center of grid is 0,0
    private var _yModifier;


    private var _tpLft:Point2D;         // top left point of grid
    private var _rect:Rect;             // rect of the grid
```

```
//
// constructor
//
//  Constructs Grid, defining cell size, location, height and
// width.
//
//   args:
//      block_w -- width of cell block in pixels
//      block_h -- height of cell block in pixels
//        tpLft -- top left point of Grid
//            w -- width of grid (number of cell columns)
//            h -- height of grid (number of cell rows)
//         data -- raw data to initialize MMatrix with
//
public function Grid(block_w:Number, block_h:Number, tpLft:Point2D,

                     w:Number, h:Number, data:Array) {

    super(w, h, data);

    _block_w = block_w;
    _block_h = block_h;
    _tpLft   = tpLft.clone();

    // it's useful to have on hand the dimensions of a cell as a
    // Point2D
    _dim     = new Point2D(_block_w, _block_h);

    // it's also useful to have those dimesions halved.
    _half    = new Point2D(_block_w/2, _block_h/2);

    // The entire of rectangle of the grid is often needed to be
    // known
    _rect    = new Rect(_tpLft.x, _tpLft.y, _tpLft.x + _w *
               _block_w, _tpLft.y + _h * _block_h);
}

//
// ptToGridLoc()
//
//  Convert a pt location, usually a screen location,
//  to a grid cell location
//
//   args:
//          pt  -- location being checked for
//
public function ptToGridLoc(pt:Point2D):Point2D {

    // make sure we are working with a copy
    pt = pt.clone();

    // adjust the point by the top left of the grid
    pt.subtract(_tpLft);

    // add half a unit to get center
    pt.add(_half);
```

```
    // divide the point by the dimension
    pt.divide(_dim);

    // round off the point
    pt.round();

    // if location is outside the dimensions of the board then
    // constrain
    if (pt.x < 1) { pt.x = 1; } else if (pt.x > _w) { pt.x = _w; }
    if (pt.y < 1) { pt.y = 1; } else if (pt.y > _h) { pt.y = _h; }

    return pt;
}

//
// gridToPtLoc()
//
//  Convert a grid cell location to a local coordinate.
//
//   args:
//          pt  -- grid cell being converted
//
public function gridToPtLoc(pt:Point2D):Point2D {

    // make sure we are working with a copy
    pt = pt.clone();

    // multiply the point by the dimension
    pt.multiply(_dim);

    // adjust the point by the top left of the grid
    pt.add(_tpLft);

    // subtract half a unit to get center
    pt.subtract(_half);

    return pt;
}

// accessors
public function getAt(loc:Point2D) {
    return getAtXY(loc.x, loc.y);
}
public function getAtXY(x:Number, y:Number) {

    // return -1 if outside bounds of grid
    if (x < 1 || y < 1 || x > _w || y > _h) { return -1; }

    // make [1, 1] top left, not [0, 0]
    --x; --y;

    // give the data
    return _data[x][y];
}
public function get blockWidth():Number {
    return _block_w;
```

```
    }
    public function get blockHeight():Number {
        return _block_h;
    }
    public function get width():Number {
        return _w;
    }
    public function get height():Number {
        return _h;
    }

    // mutators
    public function setAt(loc:Point2D, val) {
        setAtXY(loc.x, loc.y, val);
    }
    public function setAtXY(x:Number, y:Number, val) {

        // constrain location to dimensions of grid
        if (x < 1) { x = 1; } else if (x > _w) { x = _w; }
        if (y < 1) { y = 1; } else if (y > _h) { y = _h; }

        // make [1, 1] topleft, not [0, 0]
        --x; --y;

        // set the data
        _data[x][y] = val;
    }
}
```

Point2D.as

```
////////////////////////////////////////////////////////////////////////
//
//  Point2D.as
//
//       Author: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: Used to describe a point location in two dimensional
//               space. Has useful Point2D to Point2D operations and
//               includes distance, rounding and rotation methods.
//
//      Methods:
//
//          Point2D() - Constructs Point2D, defining x and y
//            equal() - Test if another point has the same x and y as
//                      this one.
//              add() - Adds two Point2Ds together or, if val is a
//                      Number, adds that Number to both x and y in this
//                      Point2D.
//         subtract() - Subtracts a Point2D from this Point2D or, if val
//                      is a Number, subtracts that Number from both x
//                      and y in this Point2D.
//         multiply() - Multiplies two Point2Ds together or, if val is a
//                      Number, multiplies that Number to both x and y
//                      in this Point2D.
//           divide() - Divides this Point2D by another or, if val is a
//                      Number, divides both x and y by that Number
//            round() - Rounds both x and y values.  This is often
//                      needed as Flash often keeps decimal places but
//                      without displaying them when a trace() call
//                      is used!
//         distance() - Calculates the distance between this point
//                      and another.
//           rotate() - Rotates this point around (0, 0) by an angle.
//            limit() - Ensures neither x nor y are ever greater than
//                      limiting value.
//            clone() - Clones this Point2D, making a safe copy than can
//                      be manipulated.
//            print() - Prints this Point2D.
//
//         Notes:
//

class Point2D {

    private var _x:Number;
    private var _y:Number;

    //
    //  Constructs Point2D, defining x and y.
    //
    //   args:
```

```
//      x -- x location
//      y -- y location
//
public function Point2D(x:Number, y:Number) {
    _x = x;
    _y = y;
}

//
// equal()
//
//  Test if another point has the same x and y as this one.
//
//   args:
//          other   -- other pt being tested
//
public function equal(other:Point2D):Boolean {
    if (_x == other.x &&
        _y == other.y) {

        return true;
    }
    return false;
}

//
// add()
//
//  Adds two Point2Ds together or, if val is a Number, adds that
//  Number to both x and y in this Point2D.
//
//   args:
//          val     -- Point2D or Number to be added
//
public function add(val):Point2D {
    if (typeof(val) == "object") {
        _x += val.x;
        _y += val.y;
    } else {
        _x += val;
        _y += val;
    }
    return this;
}

//
// subtract()
//
//  Subtracts a Point2D from this Point2D or, if val is a Number,
//  subtracts that Number from both x and y in this Point2D.
//
//   args:
//          val     -- Point2D or Number to be subtracted
//
public function subtract(val):Point2D {

    if (typeof(val) == "object") {
```

```
        _x -= val.x;
        _y -= val.y;
    } else {
        _x -= val;
        _y -= val;
    }

    return this;
}

//
// multiply()
//
//  Multiplies two Point2Ds together or, if val is a Number,
//  multiplies that Number to both x and y in this Point2D.
//
//   args:
//          val     -- MMatrix or Number to be multiplied
//
public function multiply(val):Point2D {

    if (typeof(val) == "object") {
        _x *= val.x;
        _y *= val.y;
    } else {
        _x *= val;
        _y *= val;
    }

    return this;
}

//
// divide()
//
//  Divides this Point2D by another or, if val is a Number,
//  divides both x and y by that Number.
//
//   args:
//          val     -- Point2D or Number to be divided
//
public function divide(val):Point2D {

    if (typeof(val) == "object") {
        _x /= val.x;
        _y /= val.y;
    } else {
        _x /= val;
        _y /= val;
    }

    return this;
}

//
// round()
//
```

```
//  Rounds both x and y values.  This is often needed as a Flash
//  bug keeps decimal places but without always displaying them
//  when a trace() call is used.
//
public function round():Void {
    _x = Math.round(_x);
    _y = Math.round(_y);
}

//
// distance()
//
//  Calculates the distance between this point and another
//
//   args:
//          val -- Point2D distance is being checked between
//
public function distance(val:Point2D):Number {
    return Math.sqrt(Math.pow(val.x - _x, 2) +
        Math.pow(val.y - _y, 2));
}

//
// rotate()
//
//  Rotates this point around (0, 0) by an angle
//
//   args:
//          angle -- Angle in degrees by which to rotate point
//
public function rotate(angle:Number):Void {

    var theta:Number = Utils.degreesToRadians(angle);
    var x:Number = Math.cos(theta)*_x + -Math.sin(theta)*_y;
    var y:Number = Math.sin(theta)*_x +  Math.cos(theta)*_y;

    _x = x;
    _y = y;
}

//
// limit()
//
//  Ensures neither x nor y are ever greater than limiting value
//
//   args:
//          val -- limiting value.
//
public function limit(val:Number):Void {
    _x = _x%val;
    _y = _y%val;
}

//
// clone()
//
//  Clones this Point2D, making a safe copy than can be manipulated
```

```
    //
    //   return --  copy of this Point2D
    //
    public function clone():Point2D {
        return new Point2D(_x, _y);
    }

    //
    // print()
    //
    //  Prints this Point2D
    //
    public function print():Void {
        trace("(x:"+ _x +" y:"+ _y +")");
    }

    // accessors
    public function get length():Number {
        return Math.sqrt(Math.pow(_x, 2) + Math.pow(_y, 2))
    }
    public function get x():Number {
        return _x;
    }
    public function get y():Number {
        return _y;
    }

    // mutators
    public function set x(val:Number) {
        _x = val;
    }
    public function set y(val:Number) {
        _y = val;
    }
}
```

Rect.as

```
////////////////////////////////////////////////////////////////////////
//
//  Rect.as
//
//       Author: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: Used to describe a rectangle in two dimensional space
//               as described by a top left point and a bottom right.
//               Has useful operations to detect intersections with
//               other Rects and if a Point2D is inside the area
//               bounded by the Rect.
//
//      Methods:
//
//            Rect() - Test if another point has the same x and y as
//           equal() - Test if another Rect has the same x1, y1, x2, y2
//
//                     as this one.
//             add() - Adds two Rects together or, if val is a Number,
//                     adds that Number to all four points in this Rect
//        subtract() - Subtracts a Rect from this Rect or, if val is a
//                     Number, subtracts that Number from all four
//                     points in this Rect.
//        multiply() - Multiplies two Rects together or, if val is a
//                     Number, multiplies that Number to all four
//                     points in this Rect.
//          divide() - Divides this Rect by another or, if val is a
//                     Number, divides all four points by that Number.
//           round() - Rounds all four values.  This is often needed as
//                     a Flash bug keeps decimal places but without
//                     always displaying them when a trace() call
//                     is used.
//           clone() - Clones this Rect, making a safe copy than can
//                     be manipulated
//          inside() - Tests if a point is inside this rect
//       intersect() - Tests if val Rect intersects with this Rect
//         between() - Tests if val is between a and b
//           print() - Prints the four points for this Rect
//
//        Notes:
//

class Rect {

    private var _x1:Number;
    private var _y1:Number;
    private var _x2:Number;
    private var _y2:Number;

    //
    //  Constructs Rect, defining x1, y1, x2, y2
```

```
//
//   args:
//      x1 -- left location
//      y1 -- top location
//      x2 -- right location
//      y2 -- bottom location
//
public function Rect(x1:Number, y1:Number, x2:Number, y2:Number) {
    _x1 = x1;
    _y1 = y1;
    _x2 = x2;
    _y2 = y2;
}

//
// equal()
//
//  Test if another Rect has the same x1, y1, x2, y2 as this one.
//
//   args:
//          other   -- other Rect being tested
//
public function equal(other:Rect):Boolean {
    if (_x1 == other.x1 &&
        _y1 == other.y1 &&
        _x2 == other.x2 &&
        _y2 == other.y2) {

        return true;
    }
    return false;
}

//
// add()
//
//  Adds two Rects together or, if val is a Number, adds that
//  Number to all four points in this Rect.
//
//   args:
//          val     -- Rect or Number to be added
//
public function add(val):Rect {
    if (typeof(val) == "object") {
        _x1 += val.x1;
        _y1 += val.y1;
        _x2 += val.x2;
        _y2 += val.y2;
    } else {
        _x1 += val;
        _y1 += val;
        _x2 += val;
        _y2 += val;
    }
    return this;
}
```

```
//
// subtract()
//
//  Subtracts a Rect from this Rect or, if val is a Number,
//  subtracts that Number from all four points in this Rect.
//
//   args:
//          val     -- Rect or Number to be subtracted
//
public function subtract(val):Rect {

    if (typeof(val) == "object") {
        _x1 -= val.x1;
        _y1 -= val.y1;
        _x2 -= val.x2;
        _y2 -= val.y2;
    } else {
        _x1 -= val;
        _y1 -= val;
        _x2 -= val;
        _y2 -= val;
    }

    return this;
}

//
// multiply()
//
//  Multiplies two Rects together or, if val is a Number,
//  multiplies that Number to all four points in this Rect.
//
//   args:
//          val     -- MMatrix or Number to be multiplied
//
public function multiply(val):Rect {

    if (typeof(val) == "object") {
        _x1 *= val.x1;
        _y1 *= val.y1;
        _x2 *= val.x2;
        _y2 *= val.y2;
    } else {
        _x1 *= val;
        _y1 *= val;
        _x2 *= val;
        _y2 *= val;
    }

    return this;
}

//
// divide()
//
//  Divides this Rect by another or, if val is a Number,
//  divides all four points by that Number.
```

```
//
//   args:
//          val     -- Rect or Number to be divided
//
public function divide(val):Rect {

    if (typeof(val) == "object") {
        _x1 /= val.x1;
        _y1 /= val.y1;
        _x2 /= val.x2;
        _y2 /= val.y2;
    } else {
        _x1 /= val;
        _y1 /= val;
        _x2 /= val;
        _y2 /= val;
    }

    return this;
}

//
// round()
//
//  Rounds all four values.  This is often needed as a Flash bug
//  keeps decimal places but without always displaying them when a
//  trace() call is used.
//
public function round():Void {
    _x1 = Math.round(_x1);
    _y1 = Math.round(_y1);
    _x2 = Math.round(_x2);
    _y2 = Math.round(_y2);
}

//
// clone()
//
//  Clones this Rect, making a safe copy than can be manipulated
//
//   return --  copy of this Rect
//
public function clone():Rect {
    return new Rect(_x1, _y1, _x2, _y2);
}

//
// inside()
//
//  Tests if a point is inside this rect
//
//   args:
//          pt  -- Point2D being tested
//
public function inside(pt:Point2D):Boolean {
    return (insideXY(pt.x, pt.y));
}
```

```
public function insideXY(x:Number, y:Number):Boolean {

    return (between(x, _x1, _x2) &&
            between(y, _y1, _y2))
}

//
// intersect()
//
//  Tests if val Rect intersects with this Rect
//
//   args:
//          val -- Rect being tested
//
public function intersect(val:Rect):Boolean {

    // check horizontals
    if ( between(_x1, val.x1, val.x2) ||
         between(_x2, val.x1, val.x2) ||
         between(val.x1, _x1, _x2) ||
         between(val.x2, _x1, _x2) ) {

        // check verticals
        if ( between(_y1, val.y1, val.y2) ||
             between(_y2, val.y1, val.y2) ||
             between(val.y1, _y1, _y2) ||
             between(val.y2, _y1, _y2) ) {

             return true;
        }
    }
    return false;

}

//
// between()
//
//  Tests if val is between a and b
//
//   return -- true or false
//
private static function between(val:Number, a:Number,
         b:Number):Boolean {
    return (a < b) ? (val >= a && val <= b):(val >= b && val <= a);

}

//
// print()
//
//  Prints the four points for this Rect
//
public function print():Void {
    trace("(x1:"+ _x1 +" y1:"+ _y1 +" x2:" +
                  _x2 + " y2:"+ _y2 +")");
}
```

```
    //
    // accessors

    //
    // return a point for the center of the rect
    public function get center():Point2D {
        return new Point2D( ((x1 < x2) ? x1: x2) + this.width/2,
                            ((y1 < y2) ? y1: y2) + this.height/2);
    }
    public function loc():Point2D { return center(); }
    public function get width():Number  { return Math.abs(x2 - x1); }
    public function get height():Number { return Math.abs(y2 - y1); }
    public function get x1():Number { return _x1; }
    public function get y1():Number { return _y1; }
    public function get x2():Number { return _x2; }
    public function get y2():Number { return _y2; }
    public function get firstPt():Point2D {
        return new Point2D(_x1, _y1);
    }
    public function get secondPt():Point2D {
        return new Point2D(_x2, _y2);
    }

    //
    // mutators
    public function set x1(val:Number) { _x1 = val; }
    public function set y1(val:Number) { _y1 = val; }
    public function set x2(val:Number) { _x2 = val; }
    public function set y2(val:Number) { _y2 = val; }
}
```

General Utilities

Utils.as

```
////////////////////////////////////////////////////////////////////////
//
//  Utils.as
//
//       Author: Russell Lowke
//               © Copyright 2006 Russell Lowke
//               All rights reserved.
//
//         Date: May 1st 2006
//
//  Description: General purpose static utility functions.
//
//      Methods:
//
//    setXMLreader() - Attach generic XML reader to an XML, triggered
//                     by onLoad.
//       randomInt() - Generate a random integer from "low" value to
//                     "high" value.
//      cleanAngle() - Set variable so it ranges between 0 and 359
//        makeRect() - Given an array of points give general rectangle
//    compareAngle() - Give the difference between two angles
// smallerOfTwoPts() - Return the smaller of two points
//       isACloser() - Return true if ptA is closer to origin than ptB
//          parity() - Determine even/odd parity
//    FPS_to_Ticks() - Convert frames per second to ticks (1/60ths of a
//                     second) per frame.
//       ptToAngle() - Return an angle in degrees given a velocity
//                     vector
//       angleToPt() - Return a velocity vector as a Point2D when
//                     given an angle.
//     angleToCell() - Convert an angle (in degrees) to an animation
//                     cell number where a facing of 0 degrees (which
//                     is an East facing, or rather facing the right
//                     screen edge) will yield the first cell number,
//                     while a  359 degree facing will yield the last
//                     cell number
//        incOrDec() - Given a looping sequence in both directions with
//                     a range of max if at n which direction (+ or -)
//                     is quickest to get to dest.
//radiansToDegrees() - Convert radians to degrees.
//degreesToRadians() - Convert degrees to radians.
//
//  Notes:
//

class Utils {

    //
    // setXMLreader()
    //
    //  Attach a generic XML reader to an XML, triggered by onLoad
    //
```

```
//   args:
//              xml  -- xml object being read
//           object  -- global object to which parsed data is sent
//           method  -- method in object to which parsed data is
// sent
//
public static function setXMLreader(xml:XML, object:String,
         method:String):Void {

    xml.onLoad = function(success:Boolean) {

        if (success) {

            // anonymous object used to pass name value data pairs
            var data:Object;

            // variables for name value pairs
            var name:String;
            var value:String;

            var items:Array = this.firstChild.childNodes;
            var nItems:Number = items.length;
            for (var i = 0; i < nItems; i++) {

                // annon. object to carry XML parameters
                data = new Object();

                var parts:Array   = items[i].childNodes;
                var nParts:Number = parts.length;
                for (var j = 0; j < nParts; j++) {
                    name  = parts[j].nodeName;
                    value = parts[j].childNodes[0].nodeValue;

                    // nodeType
                    data[name] = value;
                }

                // pass parsed data to a designated method
                // of a globally accessible object
                _global[object][method](data);

                // _global.gChevalier.addElement(data);
            }
        }
    };
}

//
// randomInt()
//
//  Generate a random integer from "low" value to "high" value
//
//   args:
//              low  -- lowest value generated
//             high  -- highest number generated
//
//           return  -- randomized value
```

```
//
// WHAT IF -'ve?  WHAT IF 0.06 ?
public static function randomInt(low:Number, high:Number):Number {
    var range:Number = (high + 1) - low;
    var r:Number = Math.floor(Math.random() * range) + low;

    return r;
}

//
// cleanAngle()
//
//   Clean angle variable so it ranges between 0 and 359
//
//   args:
//            angle  -- angle to be cleaned
//
//          return  -- angle value between 0 and 359
//
public static function cleanAngle(angle:Number) {
    angle %= 360;

    if (angle < 0) {
        return 360 + angle;
    }
    return angle;
}

//
// makeRect()
//
//   Given an array of points give general rectangle
//
//   args:
//              pts  -- array of pts to constuct Rect from
//
//          return  -- a Rect encompassing all points
//
public static function makeRect(pts:Array):Rect {

    var x1:Number = Number.MAX_VALUE;
    var y1:Number = Number.MAX_VALUE;
    var x2:Number = Number.MIN_VALUE;
    var y2:Number = Number.MIN_VALUE;
    for (var i:Number = 0; i < pts.length; ++i) {
        if (pts[i].x < x1) { x1 = pts[i].x; }
        if (pts[i].y < y1) { y1 = pts[i].y; }
        if (pts[i].x > x2) { x2 = pts[i].x; }
        if (pts[i].y > y2) { y2 = pts[i].y; }
    }
    return new Rect(x1, y1, x2, y2);
}

//
// compareAngle()
//
//   Give the difference between two angles
```

```
//
//   args:
//          angleA  -- 1st angle
//          angleB  -- 2nd angle
//
//       return     -- difference angle value between 0 and 359
//
public static function compareAngle(angleA:Number,
         angleB:Number):Number {
    return cleanAngle(Math.abs(angleA - angleB));
}

//
// smallerOfTwoPts()
//
//   Return the smaller of two points
//
//   args:
//          ptA     -- 1st pt
//          ptB     -- 2nd pt
//
//       return     -- the smaller of the two points to (0, 0);
//
public static function smallerOfTwoPts(ptA:Point2D,
         ptB:Point2D):Point2D {

    if (ptA == undefined) {
        return ptB;
    } else if (ptB == undefined) {
        return ptA;
    }

    if (ptA.length < ptB.length) {
        return ptA;
    } else {
        return ptB;
    }
}

//
// isACloser()
//
//   Return true if ptA is closer to origin than ptB
//
//   args:
//       origin     -- origin pt being tested against
//          ptA     -- 1st pt
//          ptB     -- 2nd pt
//
//       return     -- true if ptA is closer
//
public static function isACloser(origin:Point2D, ptA:Point2D,
         ptB:Point2D):Boolean {

    if (ptA == undefined) {
        return false;
    } else if (ptB == undefined) {
```

```
            return true;
        }

        return(origin.distance(ptB)>origin.distance(ptA))? true:false;
    }

    //
    // parity()
    //
    //   Determine even/odd parity
    //
    //   args:
    //          str -- source string being modified
    //
    //       return -- resultant parity
    //
    public static function parity(val:Number):Boolean {
        return (val%2 == 0);
    }

    //
    // FPS_to_Ticks()
    //
    //   Convert frames per second to ticks (1/60ths of a second) per
    // frame
    //
    //   args:
    //          fps     -- desired number of frames per second
    //
    //           return -- resultant ticks (1/60ths of a sec) per frame
    //
    public static function FPS_to_Ticks(fps:Number):Number {
        if (fps == 0) {
            return 0;   // avoid divide by zero
        } else {
            return (60.0 / fps);
        }
    }

    //
    // ptToAngle()
    //
    //   Return an angle in degrees given a velocity vector
    //
    //   args:
    //          vector  -- velocity vector stored as a Point2D
    //
    //           return -- the resultant angle
    //
    public static function ptToAngle(vector:Point2D):Number {

        var X:Number  = vector.x;
        var Y:Number  = vector.y;

        // avoid error due to divide by zero.
        if (X == 0) {
```

```
        if (Y > 0) { return 90; }   //  down
        else { return 270; }        //  up

    } else {                        // quadrants

        var degrees:Number = radiansToDegrees(Math.atan(Y/X));
        if (X > 0) {
            // if (degrees < 0) { return 360 + degrees; } else {
            // return degrees; }
            return 360 + degrees;
        } else { return 180 + degrees; }
    }
}

//
// angleToPt()
//
//   Return a velocity vector as a Point2D when given an angle
//
//   args:
//          degrees -- the angle in degrees
//            speed -- the speed of the vector in that direction
//
//           return -- the resultant velocity vector as a Point2D
//
public static function angleToPt(degrees:Number,
         speed:Number):Point2D {

    if (speed == undefined) { speed = 1; }

    var radians:Number = degreesToRadians(degrees);
    var vector:Point2D   = new Point2D(Math.cos(radians)*speed,
             Math.sin(radians)*speed);

    return vector;
}


//
// angleToCell()
//
//  Convert an angle (in degrees) to an animation cell number
//  where a facing of 0 degrees (which is an East facing,
//  or rather facing the right screen edge) will yield the first
//  cell number, while a 359 degree facing will yield the last
//  cell number.
//
//   args:
//                angle -- the angle in degrees
//           totalCells -- total number of cells in the sequence
//
//             return   -- the resultant cell number
//
public static function angleToCell(angle:Number,
         totalCells:Number):Number {

    // find angle size of each segment
```

```
    var segment:Number = 360.0 / totalCells;

    // augment the angle by half a segment
    angle += segment/2;

    // loop at 360 back to 0 degrees
    angle = cleanAngle(angle);

    // find the cell
    var cell:Number = Math.floor(angle/segment) + 1;

    return cell;
}

//
// incOrDec()
//
//  Given a looping sequence in both directions with a range of max
//  if at n which direction (+ or -) is quickest to get to dest
//
//   args:
//               n -- the cuurent position in sequence
//            dest -- the desired destination
//             max -- the maximum value in the looping sequence
//
//              return   -- the resultant cell number
//
public static function incOrDec(n:Number, dest:Number,
         max:Number):Number {

    var inc:Number = 0;
    var dec:Number = 0;

    if (n == dest) {
        return 0;
    } else if (n < dest) {
        inc = dest - n
        dec = n + (max - dest);
    } else if (n > dest) {
        inc = dest + (max - n);
        dec = n - dest;
    }

    if (inc > dec) {
        return -1;          // decrement
    } else {
        return +1;          // increment
    }
}

//
// radiansToDegrees()
//
//  Convert radians to degrees
//
//   args:
//              rad -- radians
```

```
    //
    //           return -- equiv. value in degrees
    //
    public static function radiansToDegrees(rad:Number):Number {
        // 2*PI = 6.28318530717959
        return ((rad*360)/6.28318530717959);
    }

    //
    // degreesToRadians()
    //
    //  Convert degrees to radians
    //
    //   args:
    //               deg -- degrees
    //
    //           return -- equiv. value in radians
    //
    public static function degreesToRadians(deg:Number):Number {
        // 2*PI = 6.28318530717959
        return ((6.28318530717959*deg)/360);
    }
}
```

Asteroids Game

Asteroids was used to create and test the *Animatem* engine. *Asteroids* is available online at www.mocaz.com/games/Asteroids.html.

Asteroids.as

```
//
//
//      Author: Russell Lowke
//        Date: Nov 25th 2005
//
//      Asteroids © Copyright 2005 Russell Lowke
//      All rights reserved
//

class Asteroids {

    // instance members

    // window width
    private var _wWidth:Number;

    // window height
    private var _wHeight:Number;

    // animator object for game
    private var _a:Animatem;

    private var _mouseX:Number         = Stage.width/2;
    private var _mouseY:Number         = Stage.height/2;

    // player level
    private var _theLevel:Number       = 0;

    // player's score
    private var _score:Number          = 0;

    // clip that contains score text
    private var _scoreClip:MovieClip;

    // clip that contains the crosshairs
    private var _hairsClip:MovieClip;

    // number of ships left
    private var _ships:Number          = 0;

    // number of asteroid items left.
    private var _nItems:Number         = 0;

    // false if game in progress
    private var _levelDone:Boolean     = true;

    // true if the title banner has been drawn
```

```
private var _titleBanner:Boolean   = false;

// time after which banner may be clicked thru
private var _holdBanner:Number     = 0;

// time after which another star is auto collected at level end
private var _holdStar:Number       = 0;

// vars for player ship
// clock time at which next player ship appears
private var _shipInAt:Number       = 0;

// player ship has immunitty for first second or so
private var _immuneTil:Number      = 0;

// time of last shot. -1 if not shooting
private var _shoot:Number          = 0;

// speed of each shot
private var _shotSpeed:Number      = 5;

// delay between bullets, in milliseconds
private var _shotDelay:Number      = 145;

// duration of bullet, in seconds.
private var _shotDuration:Number   = 1.03;

// top speed of the ship
private var _topSpeed:Number       = 2.5;

// friction ship experiences
private var _friction:Number       = 0.55;

// pts when bonus ship appears
private var _bonusAt:Number        = 500;

// vars for enemy ship
// speed of enemy ship
private var _enemySpeed:Number     = 0.7;

// additional speed for small saucer
private var _saucerExtraSp:Number  = 0.3;

// rate of enemy fire
private var _eRateOfFire:Number    = 1.9;

// enemy shot speed
private var _eShotSpeed:Number     = 2.6;

// objective sprite of the enemy, -1 == player ship
private var _objective:Number      = -1;

// time enemy will next shot
private var _enemyLShoot:Number    = 0;

// starting HP of enemy ship
private var _enemyHPbase:Number    = 4;
```

```
// HP of current enemy ship
private var _enemyHP:Number;

// specific sprite layers used
private var _banners:Number      = 202;
private var _ship_counter:Number = 201;

// sprite used for player ship
private var _ship:Number         = 200;

// sprite used for player crosshairs
private var _crosshairs:Number   = 199;

private var _bullets:Array       = new Array(198, 197,196,195,194,
        193, 192);
// sprite used for first spiked mine
private var _mineOne:Number      = 191;

// sprite used for second spiked mine
private var _mineTwo:Number      = 190;

// sprite used for enemy ship
private var _enemy:Number        = 189;

// sprite used for the enemy bullet
private var _eBullet:Number      = 188;

// level specific members
// number of stars on this level
private var _nStars:Number;

// number of mines on this level (1 or 2)
private var _nMines:Number;

// type of enemy ship player must defeat to complete level
private var _typeEnemy:Number;

// number of asteroids + stars cleared when enemy appears
private var _enemyInAt:Number;

// number of asteroids starting
private var _nAsteroids:Number;

// number of medium asteroid split from a large
private var _medAstSplit:Number;

// number of small asteroids split from a medium
private var _smAstSplit:Number;

// sounds
private var _snd_enemy_appears:Sound;
private var _snd_enemy_fire:Sound;
private var _snd_extra_ship:Sound;
private var _snd_game_over:Sound;
private var _snd_get_star:Sound;
private var _snd_kill_asteroid:Sound;
```

```
private var _snd_kill_enemy:Sound;
private var _snd_shot_enemy:Sound;
private var _snd_level:Sound;
private var _snd_mine_appears:Sound;
private var _snd_ship_dead:Sound;
private var _snd_shoot:Sound;
private var _snd_shot_mine:Sound;
private var _snd_shot_star:Sound;
private var _snd_shot_extra:Sound;
private var _snd_got_extra:Sound;
private var _snd_start_1:Sound;
private var _snd_start_2:Sound;
private var _snd_start_3:Sound;
private var _snd_start_4:Sound;
private var _snd_start_5:Sound;
private var _snd_start_6:Sound;
private var _snd_start_7:Sound;
private var _snd_start_8:Sound;


// constructor
public function Asteroids(wWidth:Number, wHeight:Number,
         path:MovieClip) {

    // window parameters
    _wWidth   = wWidth;
    _wHeight  = wHeight;
    _quality = "MEDIUM";

    // load sounds
    _snd_enemy_appears = new Sound();
    _snd_enemy_appears.attachSound("enemy_appears");
    _snd_enemy_fire    = new Sound();
    _snd_enemy_fire.attachSound("enemy_fire");
    _snd_extra_ship    = new Sound();
    _snd_extra_ship.attachSound("extra_ship");
    _snd_game_over     = new Sound();
    _snd_game_over.attachSound("game_over");
    _snd_get_star      = new Sound();
    _snd_get_star.attachSound("get_star");
    _snd_kill_asteroid = new Sound();
    _snd_kill_asteroid.attachSound("kill_asteroid");
    _snd_kill_enemy    = new Sound();
    _snd_kill_enemy.attachSound("kill_enemy");
    _snd_shot_enemy    = new Sound();
    _snd_shot_enemy.attachSound("shot_enemy");
    _snd_level         = new Sound();
    _snd_level.attachSound("level");
    _snd_mine_appears  = new Sound();
    _snd_mine_appears.attachSound("mine_appears");
    _snd_ship_dead     = new Sound();
    _snd_ship_dead.attachSound("ship_dead");
    _snd_shoot         = new Sound();
    _snd_shoot.attachSound("shoot");
    _snd_shot_mine     = new Sound();
    _snd_shot_mine.attachSound("shot_mine");
    _snd_shot_star     = new Sound();
```

```
        _snd_shot_star.attachSound("shot_star");
        _snd_shot_extra    = new Sound();
        _snd_shot_extra.attachSound("shot_extra");
        _snd_got_extra     = new Sound();
        _snd_got_extra.attachSound("got_extra");
        _snd_start_1       = new Sound();
        _snd_start_1.attachSound("start_1");
        _snd_start_2       = new Sound();
        _snd_start_2.attachSound("start_2");
        _snd_start_3       = new Sound();
        _snd_start_3.attachSound("start_3");
        _snd_start_4       = new Sound();
        _snd_start_4.attachSound("start_4");
        _snd_start_5       = new Sound();
        _snd_start_5.attachSound("start_5");
        _snd_start_6       = new Sound();
        _snd_start_6.attachSound("start_6");
        _snd_start_7       = new Sound();
        _snd_start_7.attachSound("start_7");
        _snd_start_8       = new Sound();
        _snd_start_8.attachSound("start_8");

        _a = new Animatem(path, 50, this);

        // reserve special channels
        _a.reserve(_ship_counter);
        _a.reserve(_ship);
        _a.reserve(_crosshairs);
        _a.reserve(_mineOne);
        _a.reserve(_mineTwo);
        _a.reserve(_enemy);
        _a.reserve(_eBullet);
        for (var i:Number = 0; i < _bullets.length; ++i) {
            _a.reserve(_bullets[i]);
        }

        // create a dummy level for opening
        for (var i:Number = 0; i < 5; ++i) {
            addAsteroid();
        }
        for (var i:Number = 0; i < 15; ++i) {
            addMedAsteroid(new Point2D(Utils.randomInt(0, _wWidth),
                    Utils.randomInt(0, _wHeight)));
        }
        for (var i:Number = 0; i < 20; ++i) {
            addSmAsteroid(new Point2D(Utils.randomInt(0, _wWidth),
                    Utils.randomInt(0, _wHeight)));
        }
        addMine();
        bannerAsteroids();

    }

    public function update():Void {

        // record mouse position for this update
        _mouseX = _xmouse;
```

```
_mouseY = _ymouse;

// check if need to add player ship
if (_shipInAt && getTimer() > _shipInAt) {
    _shipInAt = 0;
    addShip();
}


// check if level clear
if (! _nItems && _holdStar < getTimer()) {
    _a.setActive(_ship, 0);       // switch off the ship
    // remove crosshairs
    Mouse.show();
    _a.clearSprite(_crosshairs);
    _levelDone = true;            // level is done
    // set to next level
    ++_theLevel;

    levelData();                  // get level data
    // flag items as done
    _nItems = -1;

    _holdStar = 0;
    bannerPrepFor();
    _snd_level.start();
}

// update player ship
handleShip();

_a.update();                      // update sprites

// update the score
if (_theLevel > 0) {
    _scoreClip.ScoreFld.text = "LEVEL: " +
    _theLevel + "    SCORE: " + _score;

    if (_score > _bonusAt && _nItems-1 > _nStars) {
        _bonusAt += 500; addExtraShip();
    }
}

// introduce enemy, if any
if (_typeEnemy && _enemyInAt <= 0) {
    addEnemy(); _typeEnemy = 0;
}

// enemy fires
if (_enemyLShoot &&  _enemyLShoot < getTimer()) {
    if (_a.getTag(_ship) == "NORMAL") { addEBullet(); }
    _enemyLShoot = getTimer() + _eRateOfFire * 1000;
}

// check if only stars left. If so, get bonus on one of them
if ( _nStars != 0 && _nItems ==
    _nStars && _holdStar < getTimer()) {
```

```
        for (var i:Number = 0; i < _nStars; ++i) {
            if (_a.getTag(_a.spritelist[i]) == "NORMAL") {
                _holdStar = getTimer() + (0.4/_nStars) * 1000;
                gotStar(_a.spritelist[i], 5 * _theLevel);
                break;
            }
        }
    }
}

public function handleMouseDown():Void {

    // block mouseDowns if banner just displayed
    if (_holdBanner && _holdBanner > getTimer()) { return; }
    _holdBanner = 0;

    if (_levelDone && ! _theLevel) {
        _theLevel = 1;
        _titleBanner = true;
        levelData();
        bannerPrepFor();
    } else if (_levelDone) {
        startLevel();
    } else {
        _shoot = 0;     // > -1 then shooting
    }
}

public function handleMouseUp():Void {
    _shoot = -1;        // < 0 then not shooting
}

public function handleMouseMove():Void {
    // move crosshairs
    if (_a.getDefined(_crosshairs)) {
        _a.setLocXY(_crosshairs, _xmouse, _ymouse);
        // _hairsClip._x = _xmouse;
        // move them now for faster redraw
        // _hairsClip._y = _ymouse;
    }
}

public function collision(src:Number, trg:Number, str:String) {


    if (str == "EXTRA_V_BULLET") {
        killExtra(src);
        killBullet(trg);
    } else if (str == "EXTRA_V_SHIP") {
        gotExtra(src);
    } else if (str == "EBULLET_V_BULLET") {
        killEBullet();
        killBullet(trg);
    } else if (str == "ENEMY_V_SHIP") {
        if (_a.getTag(src) == "NORMAL") {
            _enemyHP = 0;
            killEnemy(src, trg);
```

```
            if (_a.getTag(trg) == "NORMAL") { killShip(); }
        }
    } else if (str == "ENEMY_V_BULLET") {
        killEnemy(src, trg);
        killBullet(trg);
    } else if (str == "EBULLET_V_SHIP") {
        if (_a.getTag(_eBullet) == "EBULLET") {
            killShip();
            killEBullet();
        }
    } else if (str == "STAR_V_SHIP") {
        // gotStar(src, 15);
    } else if (str == "STAR_V_MINE") {
        killStar(src);
    } else if (str == "STAR_V_ENEMY") {
        killStar(src);
    } else if (str == "STAR_V_BULLET") {
        killStar(src);
        killBullet(trg);
    } else if (str == "MINE_V_MINE") {
        _score += 20;   // special case, award pts.
        ++_score;       // recoup pt for lost bullet
        killMine(src);
        killMine(trg);
    } else if (str == "MINE_V_SHIP") {
        if (getTimer() > _immuneTil) {
            if (_a.getTag(_ship) == "NORMAL") {
                killMine(src);
            }
            killShip();
        }
    } else if (str == "MINE_V_BULLET") {
        shotMine(src, trg);
        killBullet(trg);
    } else if (str == "ASTEROID_V_BULLET") {
        killAsteroid(src);
        killBullet(trg);
    } else if (str == "ASTEROID_V_SHIP") {
        if (getTimer() > _immuneTil) {
            killAsteroid(src);
            killShip();
        }
    }
}
public function deactivated(n:Number):Void {

    if (n == _enemy ) {
        var lb:String = _a.getTag(n);
        if (lb == "NORMAL") {
            attackTheHuman(); return;
        } else if (lb == "LEAVING") {
            // saucer escaped!
            --_nItems; _enemyLShoot = 0;
        }
    }
    _a.clearSprite(n);
}
```

```
private function handleShip():Void {

    if (_a.getTag(_ship) == "NORMAL") {

        var loc:Point2D = _a.getLoc(_ship);

        // aim the ship towards the crosshairs
        var angle:Number =
            Utils.ptToAngle(new Point2D(_mouseX - loc.x,
                                        _mouseY - loc.y));

        _a.setAngle(_ship, angle);
        // angle the crosshairs
        _a.setAngle(_crosshairs, angle);


        // drive the ship
        var modifier:Number = 28.0;
        var dir:Number = 0;

        // handle ship movement
        var dif:Number = _mouseY - loc.y;
        if (dif > 0) {
            dir = +1.0;
        } else {
            dir = -1.0;
        }

        dif = Math.abs(dif);

        var basicVel:Number = dif / modifier;
        var velY = basicVel * dir;

        dif = _mouseX - loc.x;
        if (dif > 0) {
            dir = +1;
        } else {
            dir = -1;
        }

        dif = Math.abs(dif);
        basicVel = dif / modifier;
        var velX = basicVel * dir;

        _a.setVelXY(_ship, velX, velY);

        // handle guns
        var timeIs:Number = getTimer();
        // shoot!
        if (_shoot != -1 && _shoot + _shotDelay < timeIs) {

            _shoot = timeIs;
            addBullet();
        }
    }
}
```

```
private function killAsteroid(n:Number) {

    var loc:Point2D  = _a.getLoc(n);      // asteroid location
    var lbl:String = _a.getTag(n);        // asteroid tag

    if (lbl == "LARGE") {

        _snd_kill_asteroid.start();
        --_nItems;
        _score += 4;
        ++_score;       // recoup pt for lost bullet
        if (_medAstSplit == 0) {
            // explode like a small asteroid
            lbl = "SMALL";

        } else {
            if (_a.getClip(n) == "ice_rock") {
                var nRocks:Number = 10 + _theLevel * 2;
                for (var i:Number = 0; i < nRocks; ++i) {
                    // surprise asteroid
                    addSmAsteroid(loc);

                }
            } else {
                for (var i:Number = 0; i < _medAstSplit; ++i) {
                    // split into medium asteroids
                    addMedAsteroid(loc);

                }
            }
            _a.setActive(n, 0);
        }
    } else if (lbl == "MEDIUM") {

        _snd_kill_asteroid.start();
        --_nItems;
        _score += 5;
        ++_score;       // recoup pt for lost bullet
        if (_smAstSplit == 0) {
            // explode like a small asteroid
            lbl = "SMALL";

        } else {
            for (var i:Number = 0; i < _smAstSplit; ++i) {
                // split into small asteroids
                addSmAsteroid(loc);

            }
            _a.setActive(n, 0);
        }
    } else if (lbl == "SMALL") {

        _snd_kill_asteroid.start();
        --_nItems;
        _score += 6;
        ++_score;       // recoup pt for lost bullet
```

```
            --_enemyInAt;
        }


        // bang, kill asteroid.
        if (lbl == "SMALL") {

            var spr:Sprite = _a.getSprite(n);
            spr.tag = "EXPLODE";
            spr.clip = "explode";
            spr.fPerSec = 45;
            spr.cycleType = "DEACTIVATE";
            spr.wallType = "NONE";
            spr.active = -1;
        }
    }

    private function killShip():Void {
        var spr:Sprite = _a.getSprite(_ship);
        if (spr.tag == "NORMAL") {
            _snd_ship_dead.start();
            spr.tag = "EXPLODE";
            spr.clip = "ship_explode";
            spr.fPerSec = 30;
            spr.cycleType = "DEACTIVATE";
            spr.active = -1;
            --_ships;

            // remove crosshairs
            Mouse.show();
            _a.clearSprite(_crosshairs);

            if (_ships == 0) {
                gameOver();
            } else {
                // wait 2.5 seconds before next ship
                _shipInAt = getTimer() + 2.5 * 1000;
            }
        }
    }

    private function killBullet(n:Number):Void {
        // remove bullet sprite
        _a.clearSprite(n);
    }

    private function shotMine(src:Number, trg:Number):Void {
        var spr:Sprite = _a.getSprite(src);
        if (spr.tag == "NORMAL") {
            _snd_shot_mine.start();
            ++_score;       // recoup pt for lost bullet
            var bulletV:Point2D  = _a.getVel(trg);
            var bumpV:Point2D    =
                Utils.angleToPt(Utils.ptToAngle(bulletV), 0.5);
            var loc:Point2D = spr.loc;
            var vel:Point2D = spr.vel;
            loc.x += bulletV.x;
```

```
            loc.y += bulletV.y;
            vel.x += bumpV.x;
            vel.y += bumpV.y;
            spr.loc = loc;
            spr.vel = vel;
        }
    }

    private function killMine(n:Number):Void {
        var spr:Sprite = _a.getSprite(n);
        if (spr.tag == "NORMAL") {
            _snd_kill_asteroid.start();
            spr.tag = "EXPLODE";
            spr.clip = "ship_explode";
            spr.fPerSec = 50;
            spr.cycleType = "DEACTIVATE";
            spr.active = -1;
        }
    }

    private function killEnemy(source:Number, target:Number):Void {
        var spr:Sprite = _a.getSprite(_enemy);

        if (spr.tag == "NORMAL" || spr.tag == "LEAVING") {
            --_enemyHP;
            if (_enemyHP > 0) {

                // recoup pt for lost bullet
                ++_score;

                // make him recalc his attack due to timeout.
                spr.tTime = getTimer();

                _snd_shot_enemy.start();
                var bulletV:Point2D  = _a.getVel(target);
                var bumpV:Point2D    =
                    Utils.angleToPt(Utils.ptToAngle(bulletV), 1.4);
                var loc:Point2D = spr.loc;
                var vel:Point2D = spr.vel;
                loc.x += bulletV.x;
                loc.y += bulletV.y;
                vel.x += bumpV.x;
                vel.y += bumpV.y;
                spr.loc = loc;
                spr.vel = vel;

            } else {
                --_nItems; _enemyLShoot = 0;
                _score += 50;
                ++_score;       // recoup pt for lost bullet
                _snd_kill_enemy.start();
                spr.tag = "EXPLODE";
                spr.clip = "ship_explode";
                spr.fPerSec = 40;
                spr.cycleType = "DEACTIVATE";
                spr.active = -1;
                spr.tTime = 0;
```

```
        }
    }
}

private function addShipCounter():Void {

    var spr:Sprite = _a.addSpriteN(_ship_counter,"ship_counter",15,
             Stage.height - 25);
    spr.frame  = _ships;
    _scoreClip = spr.movieClip;
}

private function addShip():Void {

    var spr:Sprite = _a.addSpriteN(_ship, "ship", _mouseX,
            _mouseY);
    spr.setVelXY(0, 0);
    spr.tag       = "NORMAL";
    spr.friction    = _friction;
    spr.maxVel = _topSpeed;
    spr.wallType    = "WRAP";
    spr.cycleType   = "S_ANGLE";
    spr.radius      = 16;
    spr.display = (spr.display).add(new Rect(48, 48, -48, -48));

    // ship gets 1.5 second of immunity
    _immuneTil = getTimer() + (1.5 * 1000);

    switch (Utils.randomInt(1, 8)) {        // random start sound
        case 1: _snd_start_1.start(); break;
        case 2: _snd_start_2.start(); break;
        case 3: _snd_start_3.start(); break;
        case 4: _snd_start_4.start(); break;
        case 5: _snd_start_5.start(); break;
        case 6: _snd_start_6.start(); break;
        case 7: _snd_start_7.start(); break;
        case 8: _snd_start_8.start(); break;
    }

    _a.setFrame(_ship_counter, _ships);

    // add the cross hairs
    Mouse.hide();
    spr = _a.addSpriteN(_crosshairs, "crosshairs", _mouseX,
            _mouseY);
    spr.radius = 5;
    _hairsClip = spr.movieClip;
}

private function addEBullet():Void {
    // can't fire if exploding.
    if (_a.getTag(_enemy) != "NORMAL") { return; }

    // determine bullet velocity
    var loc:Point2D  = _a.getLoc(_enemy);
    var trg:Point2D  = _a.getLoc(_ship);
    var ang:Number  = Utils.ptToAngle(new Point2D(trg.x - loc.x,
```

```
            trg.y - loc.y));
    var vel:Point2D   = Utils.angleToPt(ang, _eShotSpeed);

    _snd_enemy_fire.start();
    // remove old bullet if still in play
    _a.setActive(_eBullet, 0);


    var spr:Sprite = _a.addSpriteN(_eBullet, "eBullet", loc.x,
            loc.y);
    spr.vel   = vel;
    spr.wallType   = "DEACTIVATE";
    spr.radius     = 4;
    spr.angle      = ang;
    spr.tag      = "EBULLET";
    for (var i:Number = 0; i < _bullets.length; ++i) {
        spr.setCollision(_bullets[i], "EBULLET_V_BULLET");
    }
    spr.setCollision(_ship, "EBULLET_V_SHIP");

    _enemyLShoot = getTimer() + _eRateOfFire * 1000;
}

private function killEBullet():Void {
    _snd_kill_asteroid.start();
    var spr:Sprite = _a.getSprite(_eBullet);
    spr.tag = "EXPLODE";
    spr.clip = "ship_explode";
    spr.fPerSec = 60;
    spr.cycleType = "DEACTIVATE";
    spr.active = -1;
    spr.wallType = "NONE";
}

private function addBullet():Void {

    // can't fire if exploding.
    if (_a.getTag(_ship) != "NORMAL") { return; }

    // get a bullet channel, if any
    var sprite:Number = 0;
    for (var i:Number = 0; i < _bullets.length && ! sprite; ++i) {
        if ( ! _a.getActive(_bullets[i])) {
            sprite = _bullets[i];
        }
    }

    // determine bullet x & y velocity
    var ang:Number = _a.getAngle(_ship);
    var vel:Point2D  = Utils.angleToPt(ang, _shotSpeed);
    var gunB:Point2D  = Utils.angleToPt(ang, 24);

    // ensure no bullets with 0 velocity
    if (vel.x == 0 && vel.y == 0) { sprite = 0; }

    if (sprite) {
        var loc:Point2D  = _a.getLoc(_ship);
```

```
        var spr:Sprite = _a.addSpriteN(sprite, "bullet",
                 loc.x + gunB.x, loc.y + gunB.y);
        spr.vel   = vel;
        spr.wallType   = "DEACTIVATE";
        spr.cycleType  = "NONE";
        spr.radius     = 2;
        spr.tTime      = _shotDuration * 1000 + getTimer();
        spr.tag      = "BULLET";
        spr.angle      = ang;

        _snd_shoot.start();
        // reduce score by 1 if score > zero, player penalized for
        // shooting too much.
        if (_score > 0) { --_score;  break; }
    }
}

private function addMine():Void {

    // add sprike to one of two sprite channels
    var sprite:Number = 0;
    if ( ! _a.getActive(_mineOne)) {
        sprite = _mineOne;
    } else if ( ! _a.getActive(_mineTwo)) {
        sprite = _mineTwo;
    }

    var spr:Sprite = _a.addSpriteN(sprite,"mine",Utils.randomInt(0,
             _wWidth), Utils.randomInt(0, _wHeight));

    var speedX:Number = (Utils.randomInt(8, 18) / 10.0);
    if ( Utils.randomInt(0, 1) == 1) { speedX = -speedX; }
    var speedY:Number = (Utils.randomInt(8, 18) / 10.0);
    if ( Utils.randomInt(0, 1) == 1) { speedY = -speedY; }


    spr.setVelXY(speedX, speedY);
    spr.tag    = "NORMAL";
    spr.wallType = "REFLECT";
    spr.cycleType = "WRAP";
    spr.fPerSec = Utils.randomInt(2, 30);   // random spin speed

    // spin half the asteroids the other way
    if ( Utils.randomInt(0, 1) == 1) {

        spr.reverseCycle();
    }
    spr.radius = 16;

    spr.setCollision(_ship, "MINE_V_SHIP");
    for (var i:Number = 0; i < _bullets.length; ++i) {
        spr.setCollision(_bullets[i], "MINE_V_BULLET");
    }
    if (sprite == _mineTwo) { spr.setCollision(_mineOne,
             "MINE_V_MINE"); }
}
```

```
// add an asteroid to the playing field
private function addAsteroid():Void {

    var type:String = "big_rock";
    if (_theLevel > 3 && Utils.randomInt(1, 8) == 1) {
        type = "ice_rock";
    }


    var wMrg:Number = _wWidth  * 0.08;      // edge margin 8%
    var hMrg:Number = _wHeight * 0.10;      // edge margin 10%

    var spr:Sprite = _a.addSprite(type,
                                  Utils.randomInt(0 + wMrg,
                                         _wWidth - wMrg),
                                  Utils.randomInt(0 + hMrg,
                                         _wHeight - hMrg));
    ++_nItems;

    var speedX:Number = Utils.randomInt(2,
             4 + Math.round(_theLevel/3)) / 10.0;
    if ( Utils.randomInt(0, 1) == 1) { speedX = -speedX; }
    var speedY:Number = Utils.randomInt(2,
             4 + Math.round(_theLevel/3)) / 10.0;
    if ( Utils.randomInt(0, 1) == 1) { speedY = -speedY; }

    spr.tag = "LARGE";
    spr.setVelXY(speedX, speedY);
    spr.wallType = "WRAP";
    spr.fPerSec = Utils.randomInt(2, 30);   // random spin speed
    // spr.angle    = Utils.randomInt(0, 360); // random angle
    if ( Utils.randomInt(0, 1) == 1) { spr.reverseCycle(); }
    spr.cycleType = "WRAP";
    spr.radius    = 16;

    spr.setCollision(_ship, "ASTEROID_V_SHIP");
    for (var i:Number = 0; i < _bullets.length; ++i) {
        spr.setCollision(_bullets[i], "ASTEROID_V_BULLET");
    }
    spr.setCollision(_eBullet, "ASTEROID_V_BULLET");
}

// add an asteroid to the playing field
public function addMedAsteroid(loc:Point2D):Void {

    var spr:Sprite = _a.addSprite("medium_rock", loc.x, loc.y);
    ++_nItems;

    var speedX:Number = Utils.randomInt(1,
             7 + Math.round(_theLevel/2)) / 10.0;
    if ( Utils.randomInt(0, 1) == 1) { speedX = -speedX; }
    var speedY:Number = Utils.randomInt(1,
             7 + Math.round(_theLevel/2)) / 10.0;
    if ( Utils.randomInt(0, 1) == 1) { speedY = -speedY; }

    spr.tag = "MEDIUM";
    spr.setVelXY(speedX, speedY);
```

```
        spr.wallType = "WRAP";
        spr.fPerSec = Utils.randomInt(2, 35);    // random spin speed
        // spin half the asteroids the other way
        if ( Utils.randomInt(0, 1) == 1) {

            spr.reverseCycle();
        }
        spr.cycleType = "WRAP";
        spr.radius = 11;

        spr.setCollision(_ship, "ASTEROID_V_SHIP");
        for (var i:Number = 0; i < _bullets.length; ++i) {
            spr.setCollision(_bullets[i], "ASTEROID_V_BULLET");
        }
        spr.setCollision(_eBullet, "ASTEROID_V_BULLET");
    }

    private function addSmAsteroid(loc:Point2D):Void {

        var spr:Sprite = _a.addSprite("small_rock", loc.x, loc.y);
        ++_nItems;

        var speedX:Number = Utils.randomInt(1,
                10 + Math.round(_theLevel)) / 10.0;
        if ( Utils.randomInt(0, 1) == 1) { speedX = -speedX; }
        var speedY:Number = Utils.randomInt(1,
                10 + Math.round(_theLevel)) / 10.0;
        if ( Utils.randomInt(0, 1) == 1) { speedY = -speedY; }

        spr.tag = "SMALL";
        spr.setVelXY(speedX, speedY);
        spr.wallType = "WRAP";
        spr.fPerSec = Utils.randomInt(10, 40);  // random spin speed
        // spin half the asteroids the other way
        if ( Utils.randomInt(0, 1) == 1) {

            spr.reverseCycle();
        }
        spr.cycleType = "WRAP";
        spr.radius = 8;

        spr.setCollision(_ship, "ASTEROID_V_SHIP");
        for (var i:Number = 0; i < _bullets.length; ++i) {
            spr.setCollision(_bullets[i], "ASTEROID_V_BULLET");
        }
        spr.setCollision(_eBullet, "ASTEROID_V_BULLET");
    }

    // add a star to the playing field
    private function addStar():Void {

        var wMrg:Number = _wWidth  * 0.08;      // edge margin 8%
        var hMrg:Number = _wHeight * 0.10;      // edge margin 10%

        var spr:Sprite = _a.addSprite("star",
                Utils.randomInt(0 + wMrg, _wWidth - wMrg),
                Utils.randomInt(0 + hMrg, _wHeight - hMrg));
```

```
    spr.cycleType = "END";
    spr.fPerSec   =  Utils.randomInt(5, 8);

    ++_nItems;
    spr.tag    = "NORMAL";

    // spr.setCollision(_ship, "STAR_V_SHIP");
    for (var i:Number = 0; i < _bullets.length; ++i) {
        spr.setCollision(_bullets[i], "STAR_V_BULLET");
    }
    spr.setCollision(_mineOne, "STAR_V_MINE");
    spr.setCollision(_mineTwo, "STAR_V_MINE");
    spr.setCollision(_enemy,   "STAR_V_ENEMY");
    spr.radius = 13;
    spr.active = 1;

}


//
// add the enemy ship
private function addEnemy():Void {

    ++_nItems; _enemyHP = _enemyHPbase;
    _snd_enemy_appears.start();

    var rndY = Utils.randomInt(32, _wHeight - 32);
    var offScrnX = -32; var onnScrnX = 32 * 3;
    if (_a.getLoc(_ship).x < _wWidth/2) {
        // come in on right
        offScrnX = _wWidth - offScrnX;
        onnScrnX = _wWidth - onnScrnX;
    }

    var enemy:String = "enemy";
    if (_typeEnemy == 2) { enemy = "ssaucer"; }

    var spr:Sprite = _a.addSpriteN(_enemy, enemy, offScrnX, rndY);
    spr.tag     = "NORMAL";
    spr.wallType  = "NONE";
    spr.cycleType = "WRAP";
    spr.fPerSec  = Utils.randomInt(20, 45); // random spin speed

    if (_typeEnemy == 2) {
        spr.radius = 10;
        _enemySpeed += _saucerExtraSp;
        _enemyLShoot = 0;
    } else {
        spr.radius = 16;
        _enemyLShoot = getTimer() + _eRateOfFire/2 * 1000;
    }

    spr.setCollision(_ship, "ENEMY_V_SHIP");
    for (var i:Number = 0; i < _bullets.length; ++i) {
        spr.setCollision(_bullets[i], "ENEMY_V_BULLET");
    }
```

```
    _a.goToLocAtSpd(_enemy, new Point2D(onnScrnX, rndY),
              _enemySpeed * 5);
}


private function objectiveGone():Void {
    if ( _a.getTag(_enemy) == "NORMAL" ) {
        attackTheHuman();
    }
}
private function attackTheHuman():Void {

    if ( _nStars != 0) {

        // attack closest star
        var dist:Number = 10000000000;
        var n:Number    = 0;
        var loc:Point2D   = _a.getLoc(_enemy);
        var starCount:Number = _nStars;
        for (var i:Number = 0; i < starCount; ++i) {
            var sNum:Number = _a.spritelist[i];
            if (_a.getTag(sNum) == "NORMAL") {
                var sLoc:Point2D = _a.getLoc(sNum);
                var sDist:Number = loc.distance(sLoc);
                if (sDist < dist) { dist = sDist; n = i; }
            } else {
                // not a normal star, need to look for one more
                ++starCount;

            }
        }

        _objective = _a.spritelist[n];
        _a.goToLocAtSpd(_enemy, _a.getLoc(_objective),
                 _enemySpeed);

    } else {

        if (_a.getClip(_enemy) == "ssaucer" && _theLevel > 0 ) {

            // if a small saucer, time to escape
            var loc:Point2D = _a.getLoc(_enemy).clone();
            // exit left
            loc.x = -32;

            if (_a.getLoc(_enemy).x > _wWidth/2) {
                // exit right
                loc.x = _wWidth + 32;

            }
            _a.goToLocAtSpd(_enemy, loc, _enemySpeed);
            _a.setTag(_enemy, "LEAVING");

        } else {

            // otherwise attack player
            if (_a.getTag(_ship) == "NORMAL") {
```

```
                _objective = -1;
                _a.goToLocAtSpd(_enemy, _a.getLoc(_ship),
                        _enemySpeed);
            } else {

                // otherwise control center
                _a.goToLocAtSpd(_enemy, new Point2D(_wWidth/2,
                        _wHeight/2), _enemySpeed);
            }
        }
    }
}

private function addExtraShip() {

     _snd_extra_ship.start();

    var rndYst  = Utils.randomInt(0, _wHeight);
    var rndYend = rndYst + _wHeight/1.5;
    if (rndYst > _wHeight/2) {
        rndYend = rndYst - _wHeight/1.5;
    }
    var offScrnXst = -32; var offScrnXend = _wWidth +32;
    if (_a.getLoc(_ship).x < _wWidth/2) {
        offScrnXst  = _wWidth - offScrnXst;     // come in on right
        offScrnXend = - offScrnXend;            // leave on left
    }

    var spr:Sprite = _a.addSprite("ship", offScrnXst, rndYst);
    spr.tag     = "NORMAL";
    spr.wallType  = "NONE";
    spr.cycleType = "WRAP";
    spr.fPerSec  = Utils.randomInt(20, 35); // random spin speed
    spr.radius = 16;

    spr.setCollision(_ship, "EXTRA_V_SHIP");
    for (var i:Number = 0; i < _bullets.length; ++i) {
        spr.setCollision(_bullets[i], "EXTRA_V_BULLET");
    }

    _a.goToLocAtSpd(spr.number, new Point2D(offScrnXend, rndYend),
             _topSpeed - 0.5);
}

private function killExtra(n:Number):Void {

    var spr:Sprite = _a.getSprite(n);
    if (spr.tag == "NORMAL") {
        // shot by player
        //
        _snd_shot_extra.start();

        spr.tag = "EXPLODE";
        spr.clip = "ship_explode";
        spr.fPerSec = 30;
        spr.cycleType = "DEACTIVATE";
        spr.active = -1;
```

```
    }
}

private function gotExtra(n:Number):Void {
    _a.setActive(n, 0);
    _snd_got_extra.start();
    ++_ships;  _a.setFrame(_ship_counter, _ships);
}

private function gotStar(n:Number, val:Number):Void {
    var spr:Sprite = _a.getSprite(n);
    if (spr.tag == "NORMAL") {
        --_nItems; --_nStars; --_enemyInAt;
        _snd_get_star.start();
        _score += val;

        spr.clip = "blue_star";
        spr.movieClip._starPts.text = "+" + val;
        spr.tag       = "GOT_IT";
        spr.frame     = 1;
        spr.fPerSec   = 24;
        spr.cycleType = "DEACTIVATE";
        spr.active    = -1;
        if (n == _objective) { objectiveGone(); }
    }
}

private function killStar(n:Number):Void {
    var spr:Sprite = _a.getSprite(n);
    if (_a.getTag(n) == "NORMAL") {
        _snd_shot_star.start();             // star shot by player

        --_nItems; --_nStars; --_enemyInAt;
        _score -= 5;

        if (_score < 0) { _score = 0; }     // score never < 0

        spr.clip  = "red_star";
        spr.tag       = "EXPLODE";
        spr.frame     = 1;
        spr.fPerSec   = 28;
        spr.cycleType = "DEACTIVATE";
        spr.active    = -1;
        if (n == _objective) { objectiveGone(); }
    }
}


private function startLevel():Void {

    // clear all sprites in the animator
    _a.clearAllSprites();
    _levelDone = false;
    _nItems    = 0;

    // build level
    // crystals are added first as they must always
```

```
    // draw underneath asteroids
    var i:Number;
    for (i = 0; i < _nStars; ++i)      { addStar(); }
    for (i = 0; i < _nMines; ++i)      { addMine(); }
    for (i = 0; i < _nAsteroids; ++i)  { addAsteroid(); }
    addShip();
    addShipCounter();
}

// set the parameters for the level
private function levelData():Void {
    switch (_theLevel) {
        case 1:
            // new game!
            _score        = 0;
            _ships        = 5;

            _nStars      = 0;
            _nMines      = 0;
            _typeEnemy   = 0;
            _enemyInAt   = 0;
            _nAsteroids  = 2;
            _medAstSplit = 2;
            _smAstSplit  = 3;
            break;
        case 2:
            _nStars      = 8;
            _nMines      = 0;
            _typeEnemy   = 0;
            _enemyInAt   = 0;
            _nAsteroids  = 2;
            _medAstSplit = 2;
            _smAstSplit  = 3;
            break;
        case 3:
            _nStars      = 10;
            _nMines      = 0;
            _typeEnemy   = 2;
            _enemyInAt   = 4;
            _nAsteroids  = 3;
            _medAstSplit = 2;
            _smAstSplit  = 3;
            break;
        case 4:
            _nStars      = 12;
            _nMines      = 1;
            _typeEnemy   = 2;
            _enemyInAt   = 4;
            _nAsteroids  = 3;
            _medAstSplit = 2;
            _smAstSplit  = 4;
            break;
        case 5:
            _nStars      = 14;
            _nMines      = 1;
            _typeEnemy   = 1;
            _enemyInAt   = 4;
```

```
            _nAsteroids  = 3;
            _medAstSplit = 3;
            _smAstSplit  = 3;
            break;
        case 6:
            _nStars      = 16;
            _nMines      = 1;
            _typeEnemy   = 1;
            _enemyInAt   = 6;
            _nAsteroids  = 3;
            _medAstSplit = 3;
            _smAstSplit  = 4;
            break;
        default:            // level 7 and higher
            var inc = _theLevel - 7;
            _nStars      = 18 + inc*2;
            _nMines      = 1 + Math.round(_theLevel%2);
            _typeEnemy   = 1 + Math.round(_theLevel%2);
            _enemyInAt   = 6 + inc;
            _nAsteroids  = 4 + Math.floor((inc + 1) / 2);
            _medAstSplit = 3 + Math.floor((inc + 3) / 5);
            _smAstSplit  = 4 + Math.floor((inc + 2) / 3);
            _enemySpeed += 0.1; // saucer gets a little faster
            ++_enemyHPbase;     // ...and a little tougher

            break;
    }
}

private function gameOver():Void {
    // setting level to zero signifies end of the game
    _theLevel  = 0;
    _levelDone = true;
    _snd_game_over.start();
    bannerGameOver();
}

private function bannerAsteroids():Void {

    // opening "Asteroids" banner
    var ymodif:Number  = _wHeight/3 + 10;
    _a.addSpriteN(_banners, "b_asteroids", _wWidth/2, ymodif);
}

private function bannerGameOver():Void {

    var ymodif:Number  = _wHeight/3 + 10;
    _a.addSpriteN(_banners, "b_game_over", _wWidth/2, ymodif);

    _holdBanner = getTimer() +  0.7 * 1000;
}

private function bannerPrepFor():Void {

    // get placement for banner
    var bHeight:Number = 40;
    var ymodif:Number  = _wHeight/3;
```

```
var sprite:Number  = _banners;

// ensure first banner cleared
_a.clearSprite(_banners);

ymodif -= ((_nStars > 0) * bHeight/2);
ymodif -= ((_nMines > 0) * bHeight/2);
_a.addSpriteN(sprite, "b_prep_for", _wWidth/2, ymodif);
_a.getMovieClip(sprite).LevelFld.text =
    "Prepare for Level " + _theLevel;

if (_nStars) {
    ymodif += bHeight; ++sprite;
    _a.addSpriteN(sprite, "b_star", _wWidth/2, ymodif);
    _a.getMovieClip(sprite)._pts.text = (_theLevel * 5) +"pts";

    ++sprite;
    var spr:Sprite = _a.addSpriteN(sprite, "star",
              _wWidth/2 - 170, ymodif);
    spr.cycleType = "END";
    spr.fPerSec   =  4;

}
if (_nMines) {
    ymodif += bHeight; ++sprite;
    _a.addSpriteN(sprite, "b_mine", _wWidth/2, ymodif);

    ++sprite;
    _a.addSpriteN(sprite, "mine",  _wWidth/2 - 170, ymodif);
    _a.setCycleType(sprite, "WRAP");
    _a.setFPerSec(sprite, 8);
}
if (_typeEnemy) {
    ymodif += bHeight; ++sprite;
    _a.addSpriteN(sprite, "b_enemy", _wWidth/2, ymodif);

    ++sprite;

    var enemy:String = "enemy";
    if (_typeEnemy == 2) { enemy = "ssaucer"; }

    _a.addSpriteN(sprite, enemy, _wWidth/2 - 170, ymodif);
    _a.setCycleType(sprite, "WRAP");
    _a.setFPerSec(sprite, 8);
}
if (_nAsteroids) {
    ymodif += bHeight; ++sprite;
    _a.addSpriteN(sprite, "b_large_asteroid", _wWidth/2,
             ymodif);

    ++sprite;
    _a.addSpriteN(sprite, "big_rock", _wWidth/2 - 170, ymodif);
    _a.setCycleType(sprite, "WRAP");
    _a.setFPerSec(sprite, 8);
}
if (_medAstSplit) {
    ymodif += bHeight; ++sprite;
```

```
            _a.addSpriteN(sprite, "b_medium_asteroid", _wWidth/2,
                    ymodif);

            ++sprite;
            _a.addSpriteN(sprite, "medium_rock",  _wWidth/2 - 170,
                    ymodif);
            _a.setCycleType(sprite, "WRAP");
            _a.setFPerSec(sprite, 10);
        }
        if (_smAstSplit) {
            ymodif += bHeight; ++sprite;
            _a.addSpriteN(sprite, "b_small_asteroid", _wWidth/2,
                    ymodif);

            ++sprite;
            _a.addSpriteN(sprite, "small_rock", _wWidth/2 - 170,
                    ymodif);
            _a.setCycleType(sprite, "WRAP");
            _a.setFPerSec(sprite, 12);
        }

        // add bottom of banner
        ymodif += 25; ++sprite;
        _a.addSpriteN(sprite, "b_bottom", _wWidth/2, ymodif);

        if (_theLevel != 1) { _holdBanner = getTimer() + 0.7 * 1000; }
    }
}
```