CROSS-PLATFORM ISOMETRIC GAME ENGINE DEVELOPMENT TARGETING ANDROID

THESIS FOR THE DEGREE OF MASTER OF SCIENCE

EMBEDDED SYSTEMS

ANTHONY J. DIPERNA

OAKLAND UNIVERSITY

2013

CROSS-PLATFORM ISOMETRIC GAME ENGINE DEVELOPMENT TARGETING ANDROID

by

ANTHONY J. DIPERNA

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
EMBEDDED SYSTEMS

2013

Oakland University
Rochester, Michigan

APPROVED BY:

Subramaniam Ganesan, Ph.D., Chair	Date
Manohar Das, Ph.D.	Date
Chingseh Wu, Ph.D.	Date

To my loving wife Jenny, and my sons Matthew, Jacob and Zachary

ACKNOWLEDGMENTS

...Thanks to everyone along the way!

Anthony J. DiPerna

ABSTRACT

CROSS-PLATFORM ISOMETRIC GAME
ENGINE DEVELOPMENT TARGETING ANDROID

by

Anthony J. DiPerna

Adviser: Subramaniam Ganesan, Ph.D.

It is becoming increasingly difficult to stay competitive as a solo developer in the Mobile and Social gaming markets. Historically, one of the greatest aspects of software design is that an individual or small team of innovative developers could compete with much larger corporations. This window is may be closing on the mobile market as large corporations attempt to dominate the application releases. This thesis describes a game engine named IsoMob that enables developers to create isometric applications that target the World Wide Web, Desktop PC and Android simultaneously. It accomplishes this with a single code base written in HaXe. HaXe is an abstraction language that is able to compile to various languages such as C++, Actionscript 3.0, PHP, HTML5 and Javascript. The design and implementation of the game engine are described, as well as the performance results of a sample application built using the IsoMob isometric game engine.

TABLE OF CONTENTS

TABLE OF CONTENTS—Continued

TABLE OF CONTENTS—Continued

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Now is a very exciting time for mobile developers. Mobile phones and tablets have never been more popular, and consumers are relying on powerful smart phones to augment their lives. Today's smart phones are incredibly feature rich, they pack features such as: hardware global positioning systems (GPS), accelerometers and high resolution touch screens. This technology is becoming more and more affordable with the rise of low power, low cost, micro controllers and integrated circuits (IC). By the end of 2011, smart phones are expected to surpass non-smart phones in total marketshare [1]. An estimated 91% of americans have cellphones, and of those, 49% have smartphones [1]. This means there are around 150 million people in the United States alone that have smartphones as of July 2011, and according to projections, that number is still increasing. This boom in smartphone usage has also created a software expansion as developers are drawn into the new market of developing mobile games and applications.

1.1 Motivation

The range of applications being deployed to mobile environments deployed today is considerably broader than it was even a few years ago. The huge growth in the mobile market was a combination of cheap hardware, abundant technology, and essentially the iPhone. Since the iPhone was released, Google has introduced the Android operating system, HP acquired Palm (WebOS), Nokia partnered with Windows Phone, and Motorola Spun off Motorola Mobility Holdings[2]. Steve Jobs

himself said "...we brought great software to the smart phone space", and this is truly what started the smart phone surge - great software on a new mobile platform. This breakthrough, in turn, defined a new type of software application almost overnight; the mobile application. Over the last six months (Feb. 2011 - Jul. 2011) there have been, on average, 35,000 new android applications developed per month[3]. The total number of android applications as of August 2011 is over 250,000[4]. Around the same time Facebook also became a viable development platform. Facebook transformed small development shops like Zynga, Playfish and Popcap games into game juggernauts that could compete with industry incumbents such as Electronic Arts (EA). Most mobile game applications work very well on Facebook as well as their native platform. This is because facebook games are essentially mobile applications and share many of the "social gaming" aspects that have become popular in mobile applications.

1.1.1 Why Isometric?

Isometric games are still very popular in today's gaming world. Particularly, adventure and role playing games (RPGs) make heavy use of the isometric formats. This style of game is very good for a mobile application because it gives the illusion of a 3D game, even though the entire back end is made up of 2D calculations. Isometric games also work very well as tile based games. Using tiles to layout a game has been popular for quite a while, and this technique was very prevalent on twenty year old video game technology such as the Super Nintendo Entertainment System (SNES). At this time in video game history platforms did not have processor speeds in the gigahertz range or hundreds of megabytes of memory, instead they used techniques such as tiling to stretch what little resources they had. Isomob is tailored to work well as an isometric game engine for mobile applications and the web. In both

of these industries resources are limited, and the benefits of excess processing power and memory are simply not there. Isometry was chosen as a happy medium between very simple 2D games and much more advanced 3D games. Looking at Figure 1.1 one can see the progression from simple games (2D), to intermediate games (Isometric), to advanced games (3D). Tools such as Unity Pro work very well with the advanced 3D games, but don't have much support for simpler games. This is not to say that a 2D game cannot be made using an advanced tool such as Unity Pro, but it is not the best tool for the job. Under similar circumstances, an engine using a wrapper aproach would be a poor fit for a 3D or even isometric game, and this approach most likely would not even be plausible due to performance constraints. Isomob presents itself as a unique way to develop isometric games for both mobile and web applications with very good performance.

1.1.2 Social Gaming

Social gaming refers to topics that make a user feel good about themselves and their gaming ability. Some common social gaming aspects are:

- Constant feedback
- Incremental progress
- Short sessions
- User Expression
- Staying in the social loop

Constant feedback refers to consistently informing the user about everything they do. Every user action is regarded as important no matter how small of

Figure 1.1: Mobile game examples. Bloons (top left), Target Tap (top right), Voltron (center left), Spectral Souls (center right), Raging Thunder (bottom left), Ravensworld (bottom right).

an impact it may actually make on a game. In the earliest of games constant feedback was something as simple as a scoreboard or sound. In Pac-man each pellet consumed by the user would increase the score and make a munching sound, both forms of constatnt feedback. When the score reached a special value the user would be awarded with an extra life and additional music would play. In Simcity, when a user reaches certain landmarks they are rewarded with special buildings, such as the mayor's mansion or a casino. To summarize, constant feedback is all about making the player feel good about their performance[5].

Incremental progress creates long term goals that push players to keep on playing. Many games have complex goals that are realized only by the completion of many smaller goals. In Simcity, the goal of the game is to develop your land into a booming metropolis. This goal can be achieved in a variety of ways and gives the user freedom to design the land how they wish. Along the way the user will face smaller problems such as: "How do I design the transportation system?", "How will electricity be generated?" and "Will anyone live in the residential areas next to the power plant?". While answering these questions the user can gauge residents satisfaction, industrial efficiency and many other statistics that reflect directly on the users' decisions.

Short sessions allow a user to play a game for a very small amount of time while still feeling satisfied with their progress. At the same time, short sessions will often "tease" the player and always leave them feeling as if they have something to do. In a way, this forms a sort of user addiction to the game and helps ensure they will actually want to keep playing. Simultaneously giving the player a sense of accomplishment while still leaving them longing for more is key for allowing users to play in small chunks of time.

User Expression is in place to make players feel that they are skillful and are excelling at the game in question. If the goal of a game is to win than users want to feel as if they are winning the game. Most people do not like to feel frustrated when they get beat a game. When a user is defeated by another user they probably feel as if they lost because the other player was better than them. The opposite seems to occur when a user is defeated by a programmed game though; in this case a user is more likely to feel as if the opposition was cheating. In general, it is good to let the user, in a sense, show off what they have done in the game. A game consisting of drawing pictures is expected to have a feature that allows a user to save their work and show it other people, even if those other people have never played the game. This technique really capitalizes on how a user sees themselves. People perceive themselves as being smart, funny, creative and as winners, and a social game should strive to provide this. The social loop refers to being able to socialize with other users of the game. Social loop interaction goes beyond just letting users communicate with each other over voice or through text. Instead, it focuses on allowing users to share in game experiences with others. A user sharing an in game gift or visiting another players home is a way for a user to do something nice for another. Humans, being social creatures tend to have a desire to reciprocate these good deeds and want to do something in return for the originator. This creates a web of interaction between others, and gives another users another outlet for sharing their lives with their community. This is a very powerful force because users outside of the game already have a predefined relationship, and if they can strengthen that in the application, then the application becomes a tool for communication. Once a game reaches this state it will draw the users back so they can communicate with each other, even if otherwise, they do not want to play the game[5].

1.1.3 Cross-platform Development

Today there are three places that an application needs to be at once: Online, on the desktop, and on mobile phones. A user developing a social application generally wants to target as many platforms as possible in order to reach many end users. The numerous amounts of mobile phone hardware present similar environments and obstacles to overcome. Even though an Android based phone and an iPhone may run a differant operating system, they still have similar processing power, RAM, flash memory, and input devices. Almost all smart phones have cameras, accelerometers, and touchscreens and these input devices target a different set of API's on the target platform, but are much more alike then different from a programming perspective. An approach that helps abstract out the differences between platforms as well as speed up application development is very much in demand for anyone interested in making a modern day software application.

CHAPTER 2

RELATED WORK

While IsoMob is a new approach to create isometric games primarily targeting smart phones, desktop PCs, and the internet, it is certainly not alone. There exists various other solutions that provide alternate methods to accomplish a similar end goal. That end goals of these existing solutions vary, but most can be summarized as either: targeting as many platforms as possible or creating portable game engines. The approach taken by IsoMob is a sort of hybrid of both of these end goals. Very good performance is essential in a game engine, but another primary concern of IsoMob is helping small group of developers achieve their goal of seeing their software application come to fruition. In order to design this middleware package other competitive solutions were evaluated and listed in detail in the rest of the chapter.

2.1 Cross-Platform Competition

When we look at developing mobile applications one of the bigger obstacles is running the application on multiple platforms. The goal of software is to get other people to use it. If an application does not target a specific platform, then it is already limiting itself in usefulness. Games in particular have almost always been released on various platforms, whether it is Nintendo, Microsoft, or Sony's latest offering. Cross-platform mobile game development is growing very rapidly in order to keep up with growing smartphone popularity. Cross-platform development not only helps development of two distinct platforms, for example iPhone and An-

droid, but also with the fragmentation with the platform itself. An example of this is targeting mobile Apple products; a developer must be concerned with iPhone, iPad, iPad2, iPod Touch, etc. The fragmentation within Android is even more severe as there are numerous companies pushing out smartphones with varying types of software and hardware, as can be seen in Table 2.1. The supported Android API version, overall hardware resources available, and screen resolution vary widely from one product to another. This fragmentation presents the developer with additional challenges in presenting a quality application on a variety of platforms. So, cross-platform development, in a broader sense, is a technique used to make an application consistent across a range of other variations.

With cross-platform mobile development we find that the existing tools for creating an isometric game are either catered to large established game companies or target the simplest of applications. The targeted cross-platform mobile game engines for comparison are shown in Table 2.2. For example, consider a user who

Table 2.1: List of Common Android Based Smartphones.

Name	Android Ver.	ROM (MB)	RAM (MB)	CPU (MHz)	Architecture	Resolution (px)
HTC Desire	2.2	512	576	1000	ARMv7	800x480
HTC Dream	2.1	256	192	528	ARMv6	480x320
HTC Sensation	2.3	1024	768	1200 x 2	ARMv7	960x540
LG Optimus 2X	2.3	8196	512	1000 x 2	ARMv7	800x480
Motorola Droid X	2.3	8196	512	1000	ARMv7	854x480
Motorola Droid 2	2.2	8196	512	1000	ARMv7	854x480
Google Nexus S	2.3	16384	512	1000	ARMv7	800x480
Samsung Transform	2.2	512	256	800	ARMv6	480x320

wants to create a game targeting a mobile application. This user could research and build a game by making a map, creating the enemies, and writing artificial intelligence (AI), but now he wants to add physics to the game. He could then do further research and add physics to the game, but this is not a trivial task and will require an extended amount of time. Once the game is complete, the user decides he wants to port it to another platform, and in doing so, completely re-writes his game in another language for the new target. Ideally, when creating a game, a middleware solution is used that provides much of the "behind the scenes" functionality. By using a game engine a user can focus on actually making a game and not making a middleware solution. By using a cross-platform game engine, a user can now open up the user base to their game and expand many possibilities. To that end, the user needs an engine that allows them these freedoms, and helps them achieve their end goal.

Most existing game engines that provide cross-platform mobile solutions rely on the user to spend a large amount of money or suffer from a lack of features or performance, as shown in Table 2.2. In the "wrapper" approach as seen in Figure 2.1 the framework uses javascript to create an abstraction layer between the user's program and the actual hardware target. This allows user's to use the same application programming interface (API) for any supported platform. This approach works really well for applications that are not very demanding. Applications that allow the user to customize their phone, or work as an informational display widget work well with this approach. The downfall to this approach is that performance will be similar to a pure javascript approach. The performance of javascript has improved quite a bit recently, but it still cannot handle the average 2D game. The HaXe language can output HTML and javascript and full performance analysis is provided, and the results for a javascript based game are simply inadequate. An-

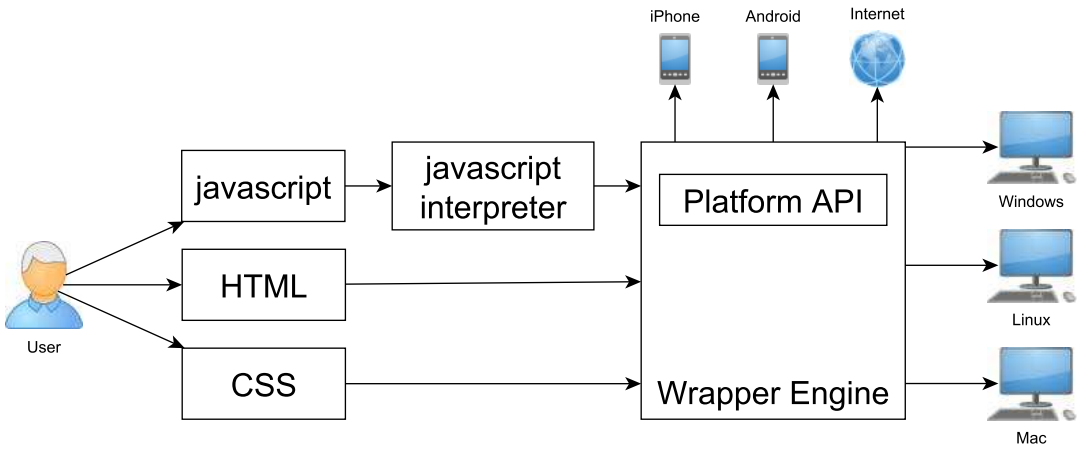

Figure 2.1: The Wrapper approach to cross-platform mobile development.

other deficiency of this method is that the javascript abstraction layer may limit the actual functionality of the user's program. If a new phone feature is made available, a user will not be able to access it until the abstraction layer is updated. This puts the user at the mercy of the framework solution. This delay of features is not a major issue, but it is something to keep in mind.

Other frameworks allow the user to work with a scripting language without ever seeing the underlying middleware layer, as seen in Figure 2.2. This approach provides very good performance because the underlying engine is written in a native language. The scripting language is used to stimulate the underlying frameworks code base while abstracting the underlying programming language. Scripting languages such as Lua are very popular in blockbuster video games. One of the reasons for this is that simply re-compiling all the code in a large scale project can take a significant amount of time. Changing just a few constants will result in a rebuild which uses up precious development time. A great aspect of scripting languages is that the operate in a much higher level(i.e. much more verbose) compared

Table 2.2: Summary of cross-platform mobile game engines

Game Engine	Cost(USD)	iPhone	Android	WebOS	Web	Flash	PC	2D	3D	Isometric	Language	Approach
Unity Pro	5000[a]	Y	Y	N	Y	Y	Y	N	Y	N	C#	Scripting
Corona SDK	350[b]	Y	Y	N	N	N	N	Y	N	N	Lua	Scripting
Cocos2D	0	Y	Y	N	N	N	N	Y	N	N	Java & Obj. C	Same API
Cocos2D-x	0	Y	Y	N	N	N	Y	Y	N	N	C++	Same API
Marmalade	499[c]	Y	Y	N	N	N	N	N	N	N	C++	Same API
Titanium	499[d]	Y	Y	N	Y	N	Y	Y	N	N	JS	Scripting
Phone Gap	0	Y	Y	Y	Y	N	Y	Y	N	N	JS	Wrapper
IsoMob	0	Y	Y	Y	Y	Y	Y	Y	N	Y	HaXe	Abstraction

[a]Pricing is per major release

[b]Pricing is per per year

[c]Pricing is per developer per year for standard edition

[d]Pricing is per developer per year for indie edition

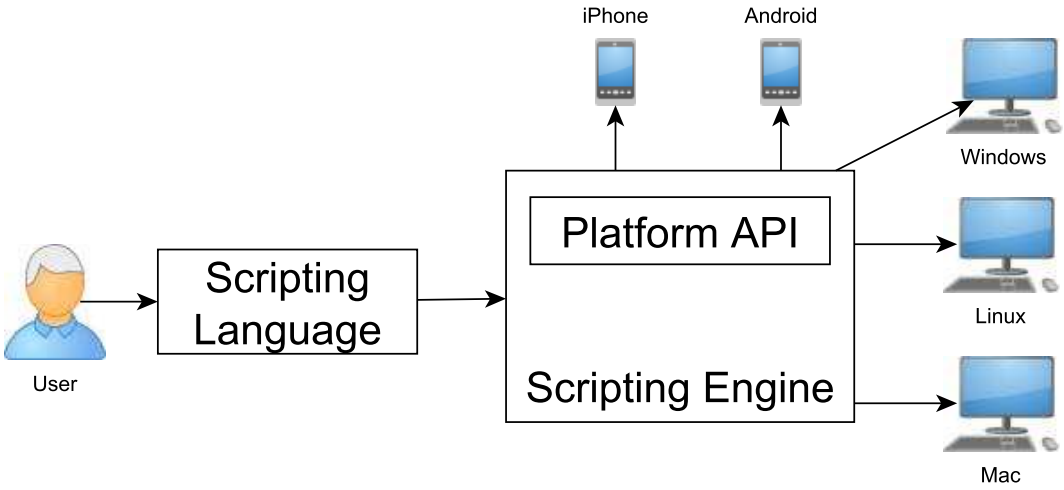

Figure 2.2: The Scripting approach to cross-platform mobile development.

to something such as C++[6]. This can be a large positive for an inexperienced programmer as most scripting languages are much simpler to learn then C++. For a large scale game the benefits of a scripting engine are simply to great to pass up, however, a smaller scale game may not need the features presented by a scripting language. With a smaller project, adding support for the scripting language may actually be more overhead then its worth. In the case of the sample application developed with IsoMob, compilation time was less approximately five seconds for Adobe Flash and double that for C++, hence, reducing compilation length would not be a very productive use of time. This approach has the same limitation as the "wrapper" approach, in that the user must wait for native platform features to be exposed to the scripting interface before they can be used. In that regard, the "scripting" approach is very similar to the "wrapper approach" except that the performance is much better.

The method of using a hardware targets' native language with a common API is also in use and can be seen in Figure 2.3. In this case, an application developed for the iPhone could be ported to Android much easier then if the program was a complete re-write. One of the problems with this approach is the developer needs to be competent in possibly very different languages. In the case of [7]Cocos2D, the developer will need to be comfortable in both Java and Objective C. The requirements for the common API approach are higher then that of other methods, but at the same time, provide very good performance. [7]Cocos2D originally was an iPhone only game engine and was eventually ported to Java as well as C++ (Cocos2D-x). With this in mind, it isn't surprising to see that application performance on the iPhone is much higher then that on Android[8].

Using a programming language that runs in a cross-platform runtime environment such as Adobe Flash is also a valid method. An application developed in flash will run on any platform that supports the flash player. On the desktop, the flash player penetration rate is over 99% in worldwide mature markets as of June 2011[9]. Besides the desktop, flash player is also included in Android version 2.2+,

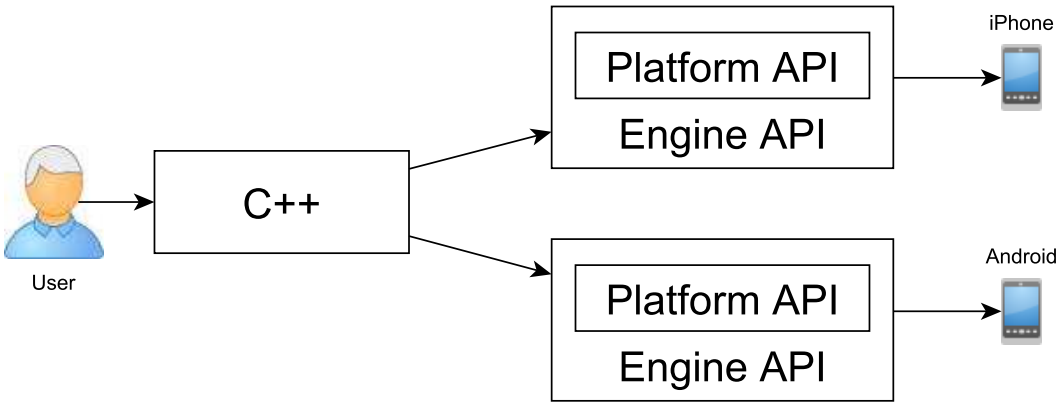

Figure 2.3: The Similar API approach to cross-platform mobile development.

iOS (iPhone and iPad), BlackBerry PlayBook and many more. Using flash to target a native platform is similar to the cross-platform wrapper method, in that flash started as a web technology. The next version of flash (flash player 11) introduces Molehill which are a set of low level, GPU- accelerated 3D APIs. This set of APIs is very similar to OpenGL and is capable of very high performance. Once this technology is released flash will most likely become a much more viable technique for developing a high performance mobile application. The first iteration of IsoMob attempted to use flash to target Android with subpar results. After the realization that flash was not the best solution for the objective at hand, IsoMob was moved in a different direction.

Abstraction is a cross-platform development strategy that adds an abstraction layer above the actual programming language, which can be seen in Figure 2.4. This method allows a single code base to target multiple platforms with their native code. Traditional cross-platform development falls into two categories; building for a single target at a time (OpenGL), and adding an abstraction layer on top of the platform (Java). The abstraction layer used for targeting both Android as well as Flash abstracts the C++ and Actionscript 3.0 code, but uses different build parameters and a few compile switches to fully support both platforms. This technique allows for a code base that is shared between the platforms, meaning that each platform only requires a very small amount of customization code.

Abstraction of the actual programming languages itself is made possible with the HaXe language. HaXe describes itself as a versatile open-source high level language and even goes so far as to call itself a "universal programming language"[10]. Although the HaXe language gives the user the ability to target multiple platforms; it does not provide any resources that aid in creating a game. So, HaXe provides the ability to target multiple platforms, but a game engine written on top of HaXe

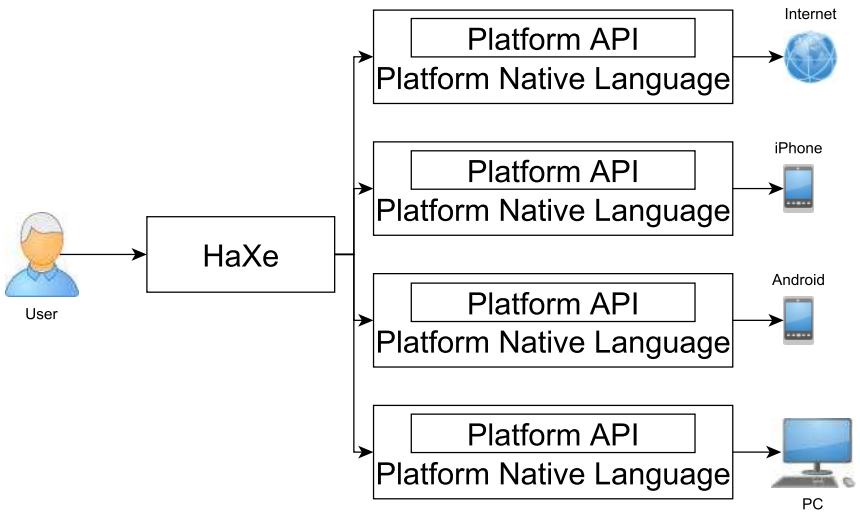

Figure 2.4: The Abstraction approach to cross-platform mobile development.

is needed. The game engine can provide functionality such as: displaying things on screen, in game physics, playing sounds and loading assets. HaXe, being open-source, has a number of library contributions to the project; the most important being neko media engine (NME). NME helps abstract away low level functionality such as drawing the the display buffer, booting the application, and file input and output. Its worth noting that NME is only used for the C++ targets: Android, iOS, webOS. NME works by replicating the Adobe Flash 10 API in C++ in a process very similar to how Simple Direct Media Layer (SDL) wraps OpenGL. This is exactly the same as the "Same API" method using by other cross-platform solutions such as Cocos2D and Marmalade. By leveraging the aspects of the "Same API" and "Scripting" methods; the "Abstraction" work flow is able to hide the platforms' native language and provide a common API at the same time. The IsoMob game engine utilizes both HaXe and NME to achieve great performance with only minor platform specific tweaks.

This thesis presents an isometric game engine that can target multiple platforms with a single code base. The engine is targeted at helping an individual or small team of developers quickly and efficiently build an application that runs on Google Android, Adobe Flash, and on any other C++ target. The smart phone, desktop, and online platforms are all targeted with a cross-platform approach with a very similar code base. There have been recent advances in cross-platform mobile game development, but most of these solutions are targeted at either the very complex or simplest of games, where the IsoMob engine strives to be somewhere in the middle. IsoMob offers unique advantages and disadvantages to other similar solutions, with its main focus being fast and efficient isometric game performance.

CHAPTER 3

ENGINE DESIGN

Rather than creating an entire game engine from scrach, I designed IsoMob based on proven game engine components. The Axonometric projection is based on the very first isometric game "Knight Lore"[11], AI is based on steering behaviors[12], the physics is a mix of Euler and Verlet Integration. This design decision facilitates the adoption of IsoMob with those familiar with game programming experience. The original Filmation engine boasted 128+ rooms, animated sprites, menus, and game logic in less than 48Kb of memory[11]. With this in mind, the engine is designed to work with relatively low resources by todays standards.

3.1 Isometry

An isometric projection is what changes a game from two dimensional (2D) to isometric and is a type of axonometric projection. Isometry is a method for visually representing three-dimensional objects in two dimensions in a chosen medium[13]. In laymans terms this means that isometry is equality within measurement. This equality means that in any isometric representation, all measurements are drawn to scale. In a three-dimensional scene, viewing distance doesn't matter because there is no vanishing point. The lack of a vanishing point means that when an isometric scene is viewed from any point the perspective will be exactly the same. Isomob uses isometry as a parallel projection in order to call itself an isometric engine.

The differences between perspective and isometric projection can be seen in Figure 3.1. Figure 3.1 shows perspective projection on the left (a) and isometric

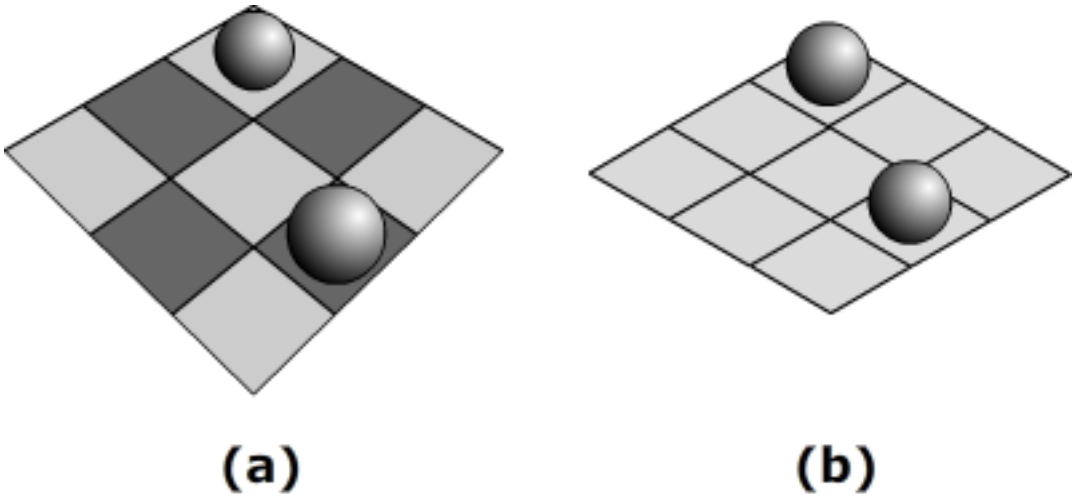

Figure 3.1: Comparison of perspective (a) and isometric (b) based projection.

projection on the right (b). In the isometric projection its clear that lines running in the same direction are parallel. Another point to notice is that the two spheres are exactly the same size, illustrating that viewing distance has no role in isometry. Looking at the perspective projection you can see almost the opposite of the isometric projection. The lines are not parallel. The lines converge at two seperate vanishing points (two-point perspective) off in the distance outside of the grid. The size of the spheres in the perspective illustration clearly show that viewing distance alters the objects. As a sphere gets further from the viewer the distance gets smaller. This is why perspective projection is summarized as an image as seen by the eye.

3.1.1 Isometric Limitations

A true isometric projection has a 30° angle between the x- and y-axis and the cartesian x-axis. Figure 3.2 shows the standard isometric projection developed

by the chemist William Farish. This projection is very popular in engineering drawings[14] because of the convenience of $\sin 30^\circ = \frac{1}{2}$. For humans attempting to create isometric drawings with a rational fraction simplifies the process, but in computer graphics this convenience actually hinders the final output.

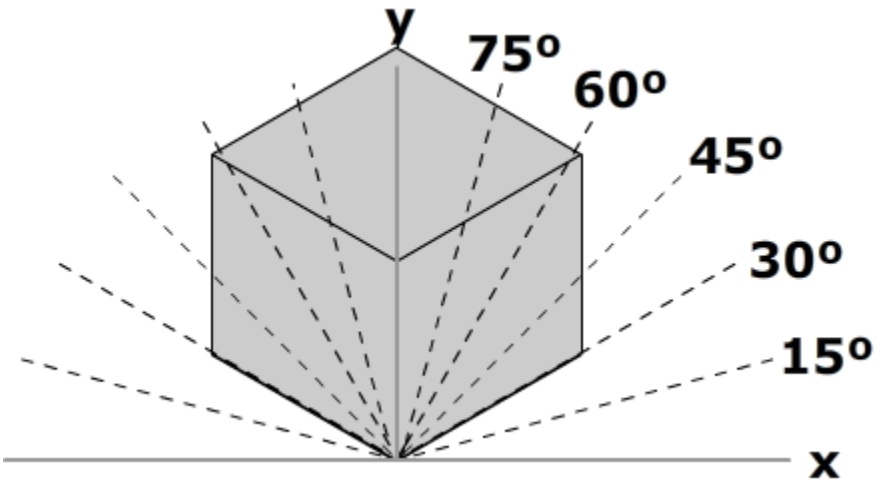

Figure 3.2: Standard isometric projection developed by William Farish.

When a face of the 30° isometric cube is broken down into triangles, we find that each triangle that has angle ratios of 1:2:3. Because we know the triangle is of 30-60-90 form, we also know that the sides will have a 1:$\sqrt{3}$:2 ratio as seen in Figure 3.3. With this information we can deduce that the height to width ratio of an isometric tile is 1:$\sqrt{3}$, meaning that the width is 1.73 times greater than the height. This ratio is impractical for computer graphics. In some implementations, such as Bresenham's line algorithim, pixel data must be defined as whole numbers, but in systems that support floating point pixel values this ratio would still give non-ideal results as seen in Figure 3.4. Looking at the rasterized line of pixel data one can

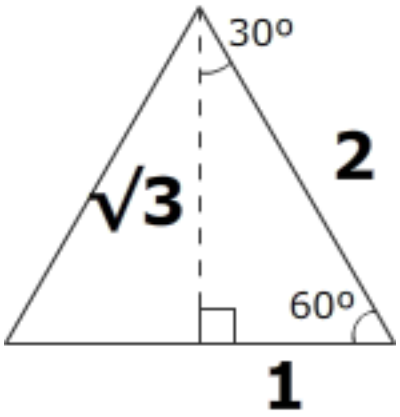

Figure 3.3: Ratios of 30-60-90 triangle sides.

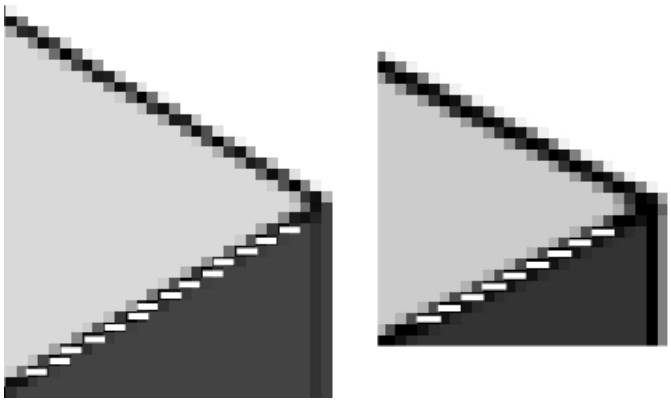

Figure 3.4: Two thousand percent magnification of the isometric and dimetric cubes seen in Figure 3.6. Notice how the dimetric line (right) has a clean 2 to 1 stepping ratio, while the isometric line (left) has a more sporadic 1.73 to 1 ratio. The white lines were added to help visualize the steps.

see that the steps generated by the line drawing algorithim are irregular even after anti-aliasing was applied. This will occur in any system where the display pixels are scare because the height to width ratio involves a non-integer. A solution for the step irregularity is to use a different type of projection, which is in turn is what "isometric" really means in most computer applications.

3.1.2 Dimetric Approach

The accepted approach to isometry in computer graphics is to use a type of axonometric projection that provides an easy to work with height to width ratio. The Dimetric projection, which means that only two of the angles between the axis are the same, provides a good solution. The desired dimetric projection angle for computer graphics is $\arctan \frac{1}{2} = 26.565°$. Figure 3.6 illustrates the difference between a cube projected in isometric versus dimetric projection. Using 26.565° as the angle between the cartesian x- and y-axis (versus 30° for true isometric) gives a 2:1 width to height ratio, as can be seen in Figure 3.5.

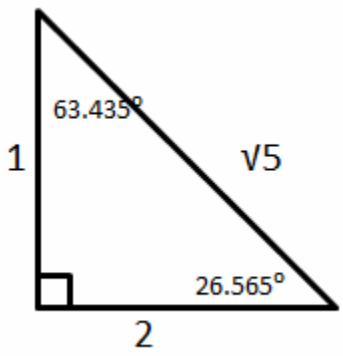

Figure 3.5: Dimetric Projection (26.565°) Triangle.

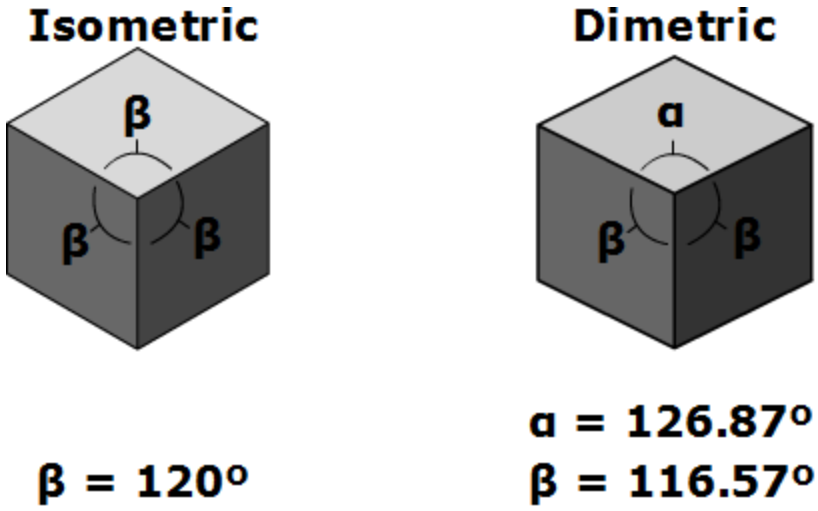

Figure 3.6: Difference between an isometric and dimetric transformation.

3.1.3 Isometric Transform

Once the type of isometry is chosen, in this case dimetric, the scene geometry transformation matrix can be determined. A 3x3 transformation matrix defined will be applied to all in game geometry in order to convert it to isometric form, the form of the matrix can seen in equation equation 3.1. Equation 3.1 shows that a standard 3x3 matrix is used for all 2D affine transformations. Because we have simplified IsoMob to eliminate variations in height along the z-axis, we can eliminate the bottom row of the matrix. The matrix elements *uvw* would be necessary in a 3D enviroment, but for our pseudo 2D purposes they can be assumed respectively as 001.

$$A = \begin{bmatrix} a & c & tx \\ b & d & ty \\ u & v & w \end{bmatrix} = \begin{bmatrix} a & c & tx \\ b & d & ty \\ 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

The linear transformations required to generate the desired isometric projection are presented by a 3x3 matrix. This transformation matrix allows: scaling, rotation, skewing and translation. Each of these linear transformations is used throughout IsoMob, but to generate the desired isometric projection only rotation and scaling are necesssary. The rotation transformation allows a transformation of an object around a fixed point. The input required for a rotation is the angle θ (in degrees) that will rotate the object counter clockwise about its origin. The matrix representation for a rotation transformation is shown in equation 3.2. The matrix in equation 3.2 represents the functional form of $x\prime = x\cos\theta - y\sin\theta$ and $y\prime = x\cos\theta + y\sin\theta$, an example of this rotation can be seen in Figure 3.7. Figure 3.7 illustrates a rotation transformation where $\theta = 45°$ on object A resulting in object B.

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

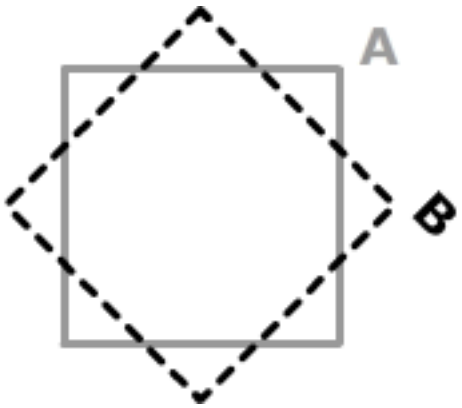

Figure 3.7: Example of a Rotation Linear Transformation.

The scaling transformation handles the second step of the isometric transform and is represented in equation 3.3. In equation 3.3 modifying a and d causes each pixel of an object to be multipled by x_{scale} along the x-axis and y_{scale} along the y-axis. When $a = d$ the object is uniformly scaled on both the x and y directions. In the opposite case of $a \neq d$ the object is scaled non-uniformly and the original shape of the object maybe compromised as shown in equation 3.3 . In Figure 3.8 (a) represents a uniform scaling transformation in which $x_{scale} = y_{scale} = 2$, and a non-uniform transformation where $x_{scale} = 3$ and $y_{scale} = 2$. Shape A represents the original square and shape B represents the shape after the scaling has been applied, and in the non-uniform case it is clear that shape B is no longer square.

$$\begin{bmatrix} x_{scale} & 0 & 0 \\ 0 & y_{scale} & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

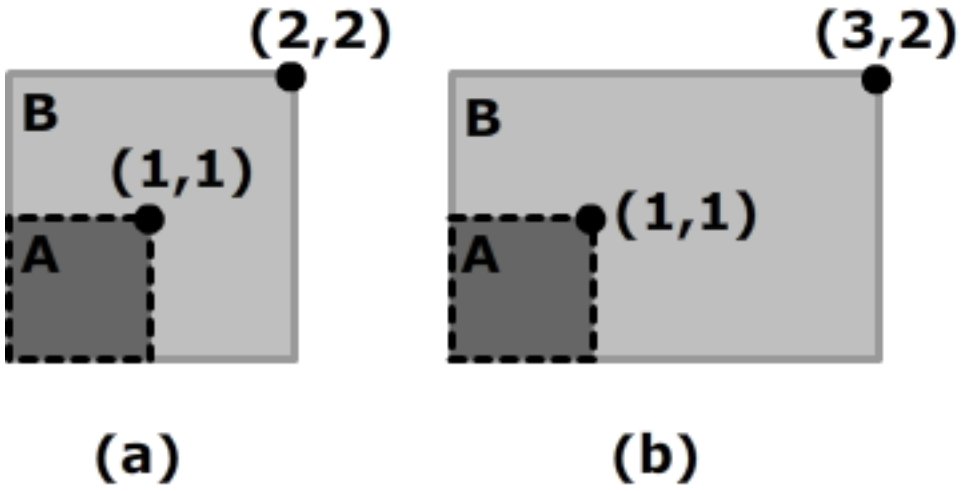

Figure 3.8: Example of a Uniform and Non-uniform Scaling Linear Transformation

The steps required for the isometric transformation are visualized in Figure 3.9. Starting with the identity matrix we perform each step in Figure 3.9 to get the resultant matrix, M_{xy}. First, a rotation of 45°is applied, followed by a height reduction scale of 50%, as shown in equation 3.4. The matrix multiplication and resultant matrix M_{xy}(isometric transform in the x-y plane) is detailed in 3.4.

$$
\begin{aligned}
M_{xy} &= M_{identity} * M_{rotation} * M_{scaling} \\
M_{xy} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos 45° & -\sin 45° & 0 \\ \sin 45° & \cos 45° & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
M_{xy} &= \begin{bmatrix} \cos 45° & -\sin 45° & 0 \\ \frac{1}{2} * \sin 45° & \frac{1}{2} * \cos 45° & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
M_{xy} &= \begin{bmatrix} 0.707106 & -0.707106 & 0 \\ 0.353553 & 0.353553 & 0 \\ 0 & 0 & 1 \end{bmatrix}
\end{aligned}
\tag{3.4}
$$

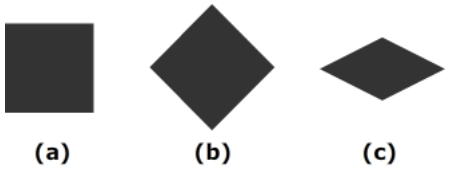

Figure 3.9: Steps required to perform the isometric transform. (a) Begin with a 100x100 px square. (b) Rotate the square from its center point by 45°CW. (c) Scale the height to 50%. In (c) the dimensions of the rhombus are 141.4x70.70 px, which gives the desired 2 to 1 ratio of width to height.

The isometric transform detailed in equation 3.4 takes care of all projections in the x-y plane, but we need still need transformations for the x-z and y-z planes. Because these transforms are derived in the exact same method as the x-y plane, the deriviation is not listed. Instead, the final transformation matrices are shown in equation 3.5 and equation 3.6.

$$M_{yz} = \begin{bmatrix} 0.707106 & 0.353553 & 0 \\ 0 & Deform & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.5}$$

$$M_{xz} = \begin{bmatrix} 0.707106 & -0.353553 & 0 \\ 0 & Deform & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.6}$$

3.1.4 Coordinate Spaces

Once the fundamental aspects of isometrics are grasped a method for translating three-dimensional world coordinates to two-dimensional screen coordinates is required. Once this translation is created all other in game calculations can be easily obtained as if the isometric transform did not exist. Note that the inverse of this transformation, converting three-dimensional world coordinates to two-dimensional screen coordinates is impossible. There is simply no way to output three coordinates x_{world} y_{world} z_{world} from two coordinates $x_{cartesian}$ $y_{cartesian}$ of input data, unless one of the axis is fixed. If the z_{world} is fixed, i.e. $z_{world} = 0$, then the calculation is possible and provides the ability to map any point in the world's x-y plane back to the screen coordinates.

Calculating the transformation from world coordinates to screen coordinates is the process of mapping the isometric axis back to cartesian axis. We already know that the angle between the cartesian x- and y-axis is set to arctan $\frac{1}{2} = 26.565°$, so we can derive the necessary equations. Figure 3.10 shows the orientation of both the isometric world axis as well as the cartesian axis. Equation 3.7 shows the derived equation for the world to screen coordinate transformation. We will then use equation 3.7 and fix $z_{world} = 0$ as seen in equation 3.8. We then perform algebra in order to get x_{world} and y_{world} in a state where we can easily add or subtract both equations as seen in equation 3.9. Now that we have equation 3.9 we can add the equations in order to solve for x_{world} as shown in equation 3.10, and subtract for y_{world} as in equation 3.11.

$$\begin{aligned} x_{cartesian} &= (x_{world} + y_{world}) * \cos\theta \\ y_{cartesian} &= z_{world} + (x_{world} - y_{world}) * \sin\theta \end{aligned} \tag{3.7}$$

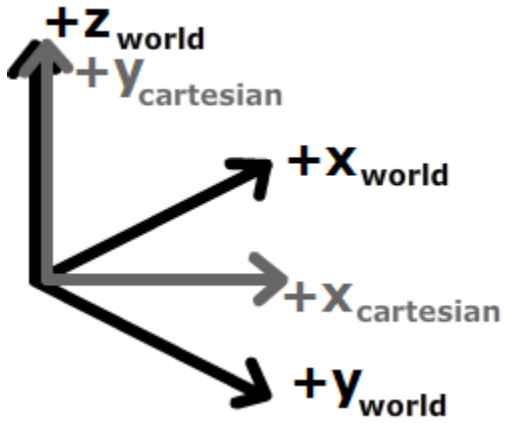

Figure 3.10: Converting isometric coordinates from screen to world.

$$\begin{aligned} x_{cartesian} &= (x_{world} + y_{world}) * \cos\theta \\ y_{cartesian} &= 0 + (x_{world} - y_{world}) * \sin\theta \end{aligned} \tag{3.8}$$

$$\begin{aligned} \frac{x_{cartesian}}{\cos\theta} &= x_{world} + y_{world} \\ \frac{y_{cartesian}}{\sin\theta} &= x_{world} - y_{world} \end{aligned} \tag{3.9}$$

$$\frac{x_{cartesian}}{\cos\theta} + \frac{y_{cartesian}}{\sin\theta} = 2x_{world} \tag{3.10}$$

$$\frac{x_{cartesian}}{\cos\theta} - \frac{y_{cartesian}}{\sin\theta} = 2y_{world} \tag{3.11}$$

$$\begin{aligned} x_{world} &= \frac{\frac{x_{cartesian}}{\cos\theta} + \frac{y_{cartesian}}{\sin\theta}}{2} \\ y_{world} &= \frac{\frac{x_{cartesian}}{\cos\theta} - \frac{y_{cartesian}}{\sin\theta}}{2} \end{aligned} \tag{3.12}$$

3.2 Scenes

The IsoMob engine's base class is IsoEngine. The IsoEngine is responsible for creating, destroying, showing, and hiding game scenes. An Isomob scene consists of geometry and objects. The geometry of an IsoScene consists of a two types planes, IsoFloors and IsoWalls, while scene objects consist of static IsoObjects and dy-

namic IsoCharacters. A UML class diagram illustrating the structure of the IsoEngine class is shown in Figure 3.11.

The IsoScene is the most feature rich class in the IsoMob engine. The UML diagram detailing the IsoScene class is shown in Figure 3.12. The design goal of an IsoScene is to manage an isometric scene. The scene management includes logic such as: rendering, camera management and keeping track of all game objects. Rendering is a large topic that is covered in Section 3.6. Camera management allows a viewport to be assigned to a position on the screen for rendering. Only elements that fit inside the viewport are rendered. In most cases, the viewport is setup to follow around the end user, doing this ensures that the end users character is always on the screen. Managing game objects encompasses logic to create, destroy and position all elements in a scene.

An IsoScene uses two primary data structures hash tables and lists. Hash tables are used to provide easy lookups for characters, objects, walls and floors. A List in HaXe is similar to an array but does not include indexed access. In HaXe, the List class is faster than the Array class so it is used for bullets since there can be a large amount of bullets in a scene. Game objects are created and destroyed by calling the appropriate methods of the IsoScene class, and scenes are handled with the appropriate methods of the IsoEngine class. By using a class that behaves

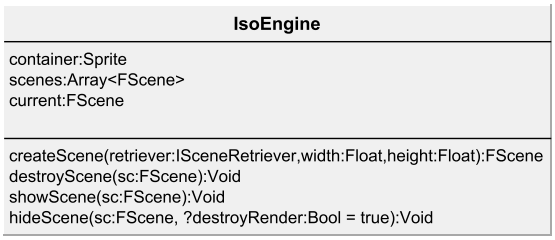

Figure 3.11: UML class diagram of IsoEngine.

Figure 3.12: UML class diagram of IsoScene.

similarly to that of one designed using a factory pattern all game objects can be managed easily. The IsoScene class is summarized in Table 3.1.

The architecture of how IsoScenes interact with the IsoEngine is shown in Figure 3.13. The developer should instantiate the IsoEngine class one time, and then use that to create various IsoScenes to represent different portions of the game.

The floors exist on the x-y plane, while the walls exist on both the x-z plane and y-z plane. Visualizations for all in game planes can be visualized in Figure 3.14. The planes in a scene are the only entities that actually use the isometric transform, objects are displayed without any deformation. An IsoScene encompasses game elements such as: objects, characters, bullets, walls, and floors.

3.2.1 Extensible Markup Language (XML)

Creating scenes is made much simpler when the required input is a human readable format. The chosen method for IsoMob is XML. An IsoMob XML file contains all the necessary info to create scene geometry, entities, triggers and graphics as seen in listing 3.1. This format is not only easy to read, but makes it very convienient to build an external scene editor. Scene creation can easily be off loaded

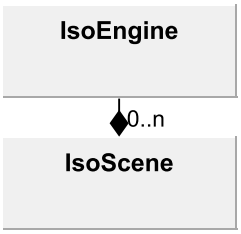

Figure 3.13: UML class diagram showing interaction of the IsoEngine with IsoScenes.

Table 3.1: Summary of properties and methods of an IsoScene.

Property / Method	Description
id	Text identifier for the scene, used for hash table lookups.
xmlObj	XML node that describes this element in the XML definition file.
uniqueId	ID number used for IsoScene hash table lookups.
x	Isometric x-coordinate.
y	Isometric y-coordinate.
z	Isometric z-coordinate.
controller	A generic controller that can add additional functionality to an element.
cell	The cell of the grid overlay used in the IsoScene, based on its x-y-z coordinates.

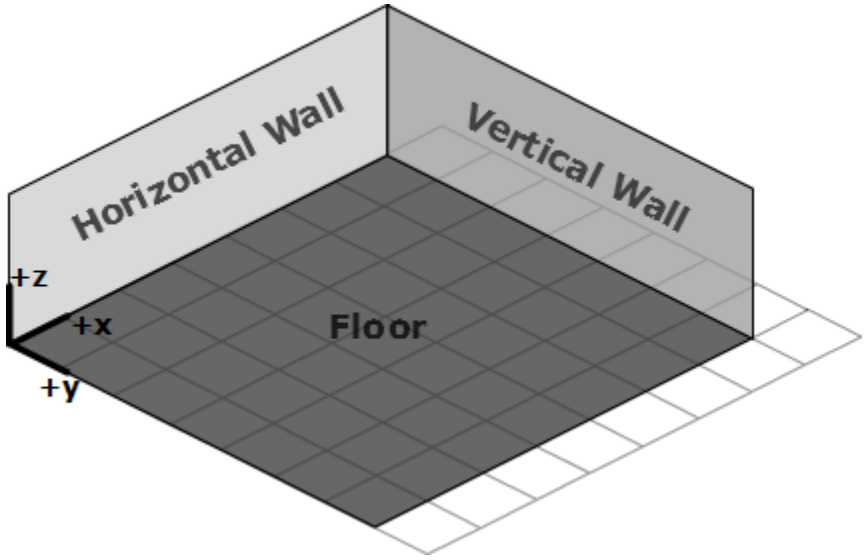

Figure 3.14: Dimetric grid and axis.

to an external editor since all that is required is the ability to read and write XML files.

The format of the scene XML definitions consists of relatively few tags. The `<scene />`, `<head />`, and `<body />` tag are the only requirements necessary for recreating a scene. The `<head />` and `<body />` tag format is a form of the head-body XML design pattern[15], and it also matches an HTML webpage setup. , so it should be familiar to anyone with web design experience. The first tag in the file is `<scene />`, this tag indicates to IsoMob that a scene is defined within this tag and is required in all cases. The `<scene />` tag has two XML properties that define parameters, `gridsize` and `levelsize` that are neccessary in order to create the the scene's grid overlay as seen in Figure 3.14 and also in the XML code shown below.

```
<scene gridsize="64" levelsize="64">
  <head>
```

```
    <name>Demo Scene</name>
    <description>Demo Level</description>
    <definitions src="../definitions/demo_assets.xml"/>
  </head>
  <body>
    <wall id="W1" x="0" y="0"   z="0" dir="vert"  size="768"
       height="64" src="tex1" />
    <wall id="W2" x="0" y="960" z="0" dir="horiz" size="832"
       height="64" src="tex2" />
    <floor id="F2" x="768" y="0" z="0" width="320" height="960
       " src="tex3" />
    <object id="O1" x="490" y="423" z="0" def="demoObj"
       orientation="0" />
    <character id="Player" x="900" y="450" z="0" def="demoChar
       " orientation="0" />
  </body>
</scene>
```

Listing 3.1: Example XML to generate an IsoScene

The gridsize specifies the size of the grid overlay in both the x- and y-directions, while levelsize specifies the height of the grid cells in the z-direction. The `<scene />` tag requires two children, both part of the XML head-body pattern, `<head />` and `<body />`. The first tag defines exactly what it implies; header information. The header information describes the name, while the `<body />` tag explicity defines scene geometry and in game entities or in other words all of the content within the scene. Scene content consists of walls, floors, non-movable and movable objects.

Walls and floors are generated as planes within the isometric world, and all objects are created in the world without any sort of projection applied. All plane dimensions are defined in each geometry tag, x y z specifies the starting point (bottom left corner) of the shape, while `height` always refers to the length of the plane along the z-axis. Walls have a single unique property that helps differentiate them from floors; the property `size`. The parameter `size` represets the length of the plane in its occupied space, either x-z or y-z. The `<wall />` tag allows creation of planes along both the x-z and y-z axis. The type of plane generated is determined by the property `dir`, which represents direction. When `dir=vert`, the wall is created as a vertical wall in the scene and is generated as a y-z plane. The other type of wall, horizontal, is created when `dir=horiz.` A horizontal wall generates a plane existing in x-z space, and besides the plane it occupies it is exactly the same as a vertical wall in every way. The `<floor />` tag is very similar to the `<wall />` tag, it generates a plane occupying x-y space and shares the x y z parameters that specify the start point. Floors also have a single unique property, `width`, that is very similar to the `size` property of the wall. Floor `width` represents the length of the shape in the x-y plane. All types of scene geometry are visualized in Figure 3.14.

All in game objects can be classified as either moving or non-moving and are differentiated by the `<object />` and `<character />` tags. Objects are static, and are used for background items that are meant as scene props. Characters extend objects and add the ability to move dynamically at run time. Characters are used for both the player of the game as well as all of the enemies. Both characters and objects require a starting isometric world coordinate represented by x y z as well as the type of graphic used to represent the object. The object's graphic points to any asset that has been loaded into the engine, full details of assets are found in Section 3.4.

3.2.2 A Simple Tile Map

In many tile based games it is common to have a tile map generate the game world. Tile maps originally started as an answer to RAM limitations on early video games. Game worlds consist of many objects and textures and tile maps were found to be a an efficient way to transfer graphics assets to the screen. The simplest way to create a tile map is with a simple text file that maps textures to a specific tile. In Figure 3.15, each index represents a different game graphic.

In this example, zero represents a light gray tile and one represents a dark gray tile. The rows of indexes represent the x-axis and the columns represent the y-axis. In Figure 3.16 one can see exactly how the tile map indexes map to the rendered isometric grid.

For simple scenes that consist of only a few types of tiles this method is quick, and works well. The problem with this type of tile map is that it doesn't scale very well when additional graphic textures are added. One way to solve this problem is to assume a default tile type and fill the entire map, and then define differenent tiles as sets of rectangles. Another alternative is to create a world editor

```
1 1 1 1 1 1 1 1
1 0 0 1 1 0 0 1
1 0 0 0 0 0 0 1
1 1 0 1 1 0 1 1
1 0 0 0 0 0 0 1
1 0 0 0 0 0 0 1
1 0 0 1 1 0 0 1
1 1 1 1 1 1 1 1
```

Figure 3.15: A simple 8x8 tile map text file tile map. Each index represents a different game texture. In this example, 0 represents a light gray tile and 1 represents a dark gray tile. See Figure 3.16 for a rendering of this tile map.

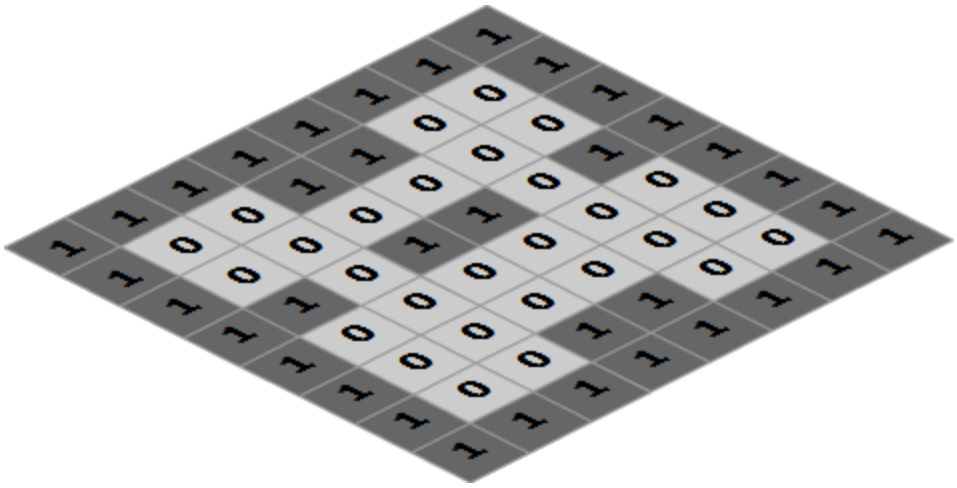

Figure 3.16: A render of the tilemap in Figure 3.15 with an overlay of the tile map values.

that uses a guided user interface (GUI) that will not only create the tile map but also the necessary metadata required to render the map.

The approach taken by IsoMob is to use a set of rectangles to define associate graphics with tiles. The simple tile map in Figure 3.15 doesn't work well for our case because we draw on more then one coordinate plane (for walls and floors). Multiple planes is difficult to indicate with 1:1 tile mapping. Instead we define sets of planes and paint textures to apply graphics. This allows the engine to build actual scene geometry, instead of using only static bitmap images which do not contain any information except for pixel data. The method for creating scene geometry is detailed in the scene XML as seen in Listing 3.1.

3.3 Dynamic Entities

IsoMob can also generate characters dynamically (at runtime) as well as two other entity types that cannot be created statically. Bullets and empty containers can be created and added to the engine at runtime to enable extra functionality. Bul-

lets provide a method to create fast moving projectiles that use a special type of collision detection to interact with other game objects. Bullets cannot collide with each other, but they can collide with any other game object such as: walls, floors, objects, and characters. Containers are used to create in game items and message boxes. Using containers allows IsoMob to position non-interactive entities in isometric world space.

3.4 Resources

All games require resources in order to stimulate the player as well as relay information to them. IsoMob supports two-dimensional bitmap graphics such as: .png, .jpg, and .gif files. The engine needs an efficient way to store graphic assets, and a very good tool for the job is a spritesheet. A spritesheet is a collection of individual bitmaps merged together into one larger bitmap. This data structure provides a number of advantages over using individual images for graphic assets.

3.4.1 Sprite Sheets

A major limitation of rendering performance, especially on mobile platforms, is the graphics processing unit (GPU). The GPU works best with a small number of large jobs, and individual images provide the complete opposite. The bus traffic to transfer many small individual images is much less efficient than that of a single spritesheet containing the images due to the overhead of transferring data on a bus. Also, the GPU has a limited capacity for storing textures at any one time. Designing a system with many small textures fills the limited capacity of the GPU and most likely will result in a high rate of texture swapping. A high amount of texture swapping is bad for rendering performance and is referred to as “texture thrashing”. This term means that the nearest graphics data is very often not in

the nearest source, and time is wasted removing the occupying texture and fetching the required graphics. The sources where texture data can be stored, sorted from fastest to slowest is as follows: GPU texture cache, GPU texture buffer (GPU RAM) and system memory (system RAM). Thrashing occurs quite frequently between the texture cache and texture buffer, but these memories are at such a high-speed that it does not cause any bottlenecks. The real problem is having to fetch textures out of system memory, as this is significantly slower than other sources. Another problem to be aware of is when the total amount of texture graphics exceeds the capacity of the GPU memory. In this case, it is impossible for the entire set of textures to be stored on the GPU, meaning that some portion of the texture data will be stored in slow to reach system memory. Another source of rendering efficiency is minimizing the actual draw calls sent to the GPU. The GPU needs to be informed of exactly what textures to draw at any one time. Without a spritesheet each display texture would require its own GPU draw call which is an expensive operation. Even though the GPU may be storing the texture, there still needs to be a hook that triggers drawing a specific texture to a location on the screen. For example, when rendering font, the spritesheet technique would allow an entire sentence of font to be drawn to the screen with a single GPU draw call. In contrast, the method of individual images would require that each letter in the font was rendered one after another. This last reason, minimizing GPU draw calls, is the most important aspect in maxizing rendering performance, especially on the mobile platforms. A summary of the spritesheet and individual images method can be seen in Table 3.2, while an in depth analysis of texture swapping on the GPU is presented in Section 5.2.

Table 3.2: Summary of spritesheet features.

Individual Images	Spritesheet
+No prepwork	-Requires prepwork to convert individual images
+Low chance of reaching texture threshold	-High chance of reaching texture threshold
-GPU Texture thrashing	+Eliminates GPU Texture thrashing
-Unneccessary overhead	+Eliminates power of 2 overhead
-High graphics bus traffic	+Low graphics bus traffic
-Many graphics output calls	+Single graphics output call

3.4.2 Sound

The game engine does not interact with sound at all, this is directly handled by the target's API for sound.

3.4.3 Textures

Textures are used to apply an image to the IsoFloors and IsoWalls within an IsoScene. There are only two different texture types currently supported in the IsoMob engine: tiled and non-tiled textures. The key difference between the two texture types is that tiled textures will have an offset applied based on the target plane's location in an IsoScene. This means that adjacent planes with the same texture will integrate seemlessly. When two non-tiled textures are placed on adjacent planes there the textures will not align correctly and look visually unpleasing to the user. An example of each type of texture is shown in Figure 3.17.

This shows that two non-tiled textures will not align correctly because the second polygon does not fall on an even division of the texture size. In this example, the texture size is 128 x 128 pixels and each wall polygon is 192 x 128 pixels. In order for the texture to align correctly it needs an offset of 64 pixels applied, instead the non-tiled texture does not apply an offset and disrupts the pattern with

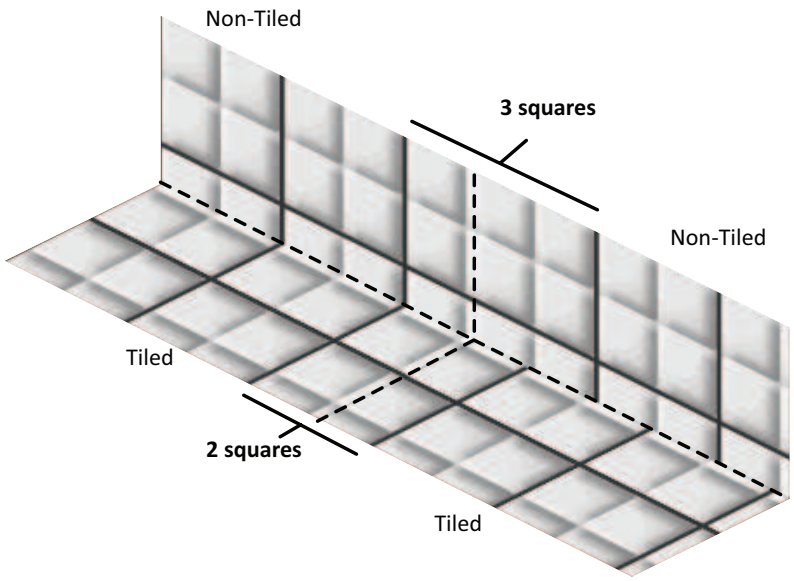

Figure 3.17: The two different types of textures: tiled (floor) and non-tiled (wall).

three squares in a row. The floor has the same texture applied, and the floor polygons have the same dimensions of 192 x 128 pixels. Physically the .png files representing these textures are exactly the same, the only difference is the `Tiled` keyword was appended to the filename. The texture applied to the wall is named texture.png, and results in a non-tiled texture application. The texture applied to the floor is named textureTiled.png and results in a tiled texture application with the correct 64 pixel offset. The texture types are defined by adding the keyword `Tiled` to the asset's file name. The game engine will automatically parse the name of a texture and classify it correctly for class instantiation as long as the proper naming convention is followed. Full details of the asset naming format is listed in Table 3.3.

The actual .png assets are prepared for game engine usage with a script also written in HaXe. Generating the necessary meta data for graphics begins to take a long time to do manually once the number of assets grows significantly large. Generating the necessary .xml nodes to represent assets for the game engine is a te-

Table 3.3: Asset naming convention, note all assets must follow camel case capitalization rules.

Type	Description	Format	Example
tex	A texture applied to walls or floors (non-tiled)	tex<name>	texGrass,texRoad
tex	A texture applied to walls or floors (tiles)	tex<name>Tiled	texBlueTiled, texFlowersTiled
obj	A stationary game object	obj<name>	objBlock,objBush
item	A game item that can be picked up	item<name>	itemFruit,itemBook
bullet	A projectile with collision detection	bullet<name>	bulletIce,bulletFire
“”	An animated game character	<name><animation><frame>	humanNorth001,catWest001

dious process that is easily automated, so a script is a great solution. The script searches for the keywords in Table 3.3 using regular expressions and automatically generated the necessary .xml data. The .xml data is then written to an external file, and that very same file is used by the engine to define the necessary assets in a scene. The source code for the script is included in

3.5 Game Objects

Game objects refer to characters, objects, and textures that are used to makeup an IsoMob scene. All graphics inside of IsoMob are represented as rasterized bitmap images, inside of spritesheets as explained in Section 3.4.1. The spritesheet represents numerous bitmap graphics that can be used for the game objects of IsoMob. These bitmap assets are loaded at runtime, as determined by their associated XML definitions. The scene XML references the XML defintions, and in turn, these XML definitions define every object, character and texture in the scene. The XML definitions are setup this way so that they may contain more then just raw bitmap graphics. The structure of a typical XML definition is shown in Listing 3.2.

```
<!-- Point to Spritesheet .xml and .png -->
<PATH_TO_ASSETS>

<!-- Define Textures -->
<TEXTURE_DEFINITION>
<TEXTURE_DEFINITION>
<TEXTURE_DEFINITION>

<!-- Define Objects -->
<OBJECT_DEFINITION>
```

```
<OBJECT_DEFINITION>
<OBJECT_DEFINITION>

<!-- Define Characters -->
<CHARACTER_DEFINITION>
<CHARACTER_DEFINITION>
<CHARACTER_DEFINITION>
```

Listing 3.2: Example XML definition used to characters, objects and textures.

The XML definition can be broken down into two distinct portions: a section that points to raw bitmap assets and a section that defines how those raw bitmaps are built into game objects. The section that points to bitmap assets is exactly as it sounds, it simply refers to the .xml and .png files of a spritesheet. The other section that builds the game objects includes important meta data that is used to properly reconstruct the necessary game engine classes, and enable the user to spawn game objects at runtime with very simple commands. Once this structure is complete a user can simply say create(gameObject), and the game engine will handle all the work of exactly how to create it. The meta data in the XML definition provides enough information to produce a generic game object at runtime, but objects and characters still require additional logic in order to give them in game behavior. The additional game behavior is described in Chapter 4.

The IsoElement class provides information that allows an object to be identified and placed in the IsoEngine. A description of the class properties is found in Table 3.4.

The IsoElement class has only a few methods, and most of them are used as abstract methods; similar to virtual functions in C++. IsoElement serves as an abstract class that is meant to be extended by other classes such as: IsoRen-

Table 3.4: Summary of properties of an IsoElement .

Class Property	Description
id	Text descriptor referencing a definition in the XML file.
xmlObj	XML node describing the element in the XML file.
uniqueId	ID used for IsoScene hash table lookups.
x	Isometric x-coordinate.
y	Isometric y-coordinate.
z	Isometric z-coordinate.
controller	Class that may add additional element functionality.
cell	The grid cell in the IsoScene, based on its x-y-z coordinates.

derableElement and IsoCamera, and never directly instantiated. It is worth noting that an IsoElement will never be displayed on the screen, as in, it is not available to the engine renderer. In order to be renderable, an element must extend the IsoRenderableElement class, which extends IsoElement, and is the true base class of every game object seen by an end user in an IsoScene. The base class for all game objects, IsoElement is detailed in Figure 3.18.

The IsoElement class serves as the base class for the IsoRenderableElement as well as the IsoCamera. The class relationships for IsoScene game objects is shown in Figure 3.19.

3.5.1 Objects

IsoObjects are game objects that cannot be generated dynamically at runtime. An IsoObject can only be created if it is included in the IsoScene's XML definition. An IsoObject is a non-moving entity that represents an obstacle or decoration in an IsoScene. Some examples of IsoObjects are chairs, tables, desks, and trees. IsoObjects are generally defined as solid objects in order to enable collision detection in a scene which allows interaction with IsoCharacters and IsoBullets.

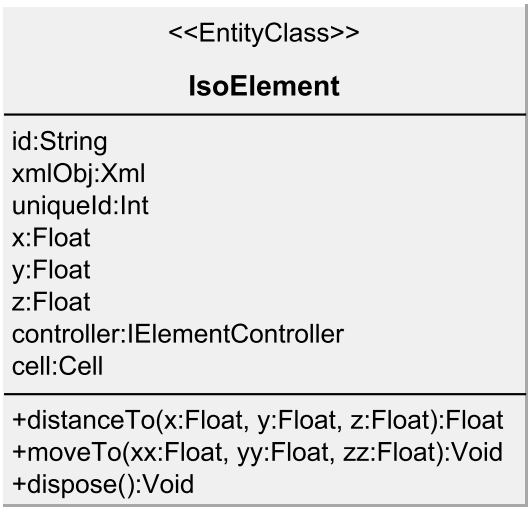

Figure 3.18: UML class diagram of IsoElement structure.

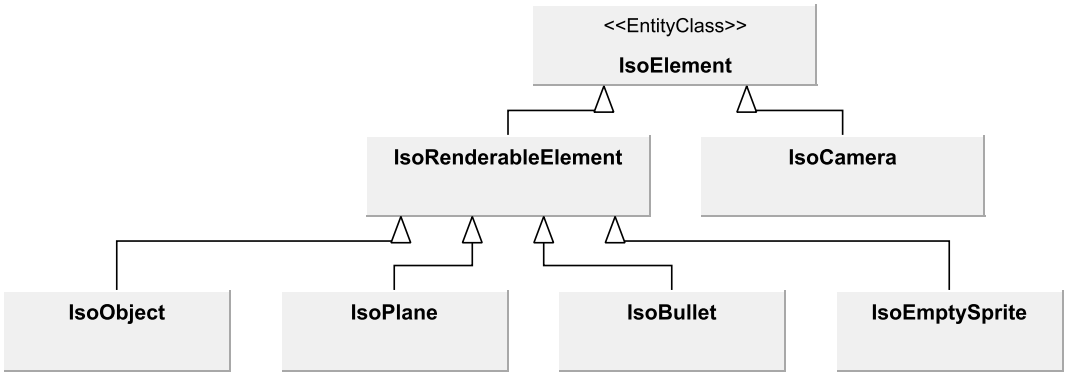

Figure 3.19: UML class diagram illustrating the relationships between the IsoElement and its child classes.

When defined this way they will be given a collision circle to interact with other collision enabled game objects. Full details of how collision detection works is in Section 4.1.3. A UML diagram showing class details for IsoObject is shown in Figure 3.20.

The IsoObject class inherits from the IsoRenderableElement class and adds animation and collision detection. Animation is defined by a sequence of bitmap images stored in an array of IsoSpriteDefinitions. The IsoSpriteDefinition class is used as a data stucture to store the actual string of bitmap images, and some additional meta data regarding how they should be displayed. In example use case of using the IsoSpriteDefinition class is an instance for the walking animation and a seperate instance for the standing animation. IsoObjects are always defined as static objects inside of an IsoScene, anything that needs to move should be defined as an IsoCharacter, which is explained in the Section 3.5.2.

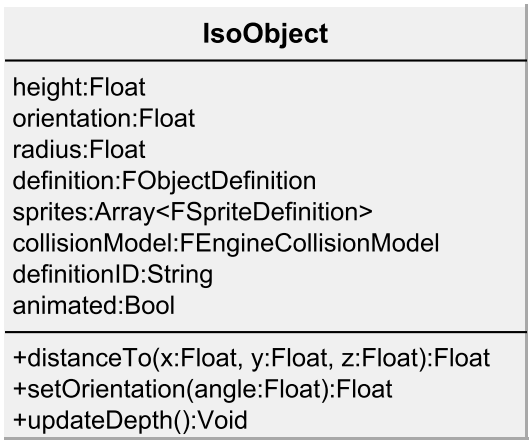

Figure 3.20: UML class diagram of an IsoObject.

3.5.2 Characters

Characters are game objects that can be generated at runtime, and have the ability to move around an IsoScene. Characters are used for the end users "hero" as well as for all enemies in the game. All in game enemies are based on a generic character class with various graphics and AI applied to make them appear different to the end user. The basic structure of a character allows it to move around a scene, change direction, and collide with other game objects.

3.6 Rendering

Arguably, the most important aspect of any game engine is the rendering performance. Without an efficient renderer a game will run at a less than ideal frame rate, and possibly become unplayable to the end user. A frame rate of approximately 30 frames per second (FPS) will be perceived as fluid motion to most users. When the frame rate dips below 30 FPS a number of detrimental events may occur. The most noticeable effect will be all in game animations will slow down, and the graphics will appear choppy since the game can not keep up with the ever changing data. Another possible effect is that the internal game timing will lose synchronization, depending on the how the main loop is defined. If the loop that controls game functionality is based on the framerate then all operations within that loop with slowdown as the framerate drops. A common occurrence is game engines will write engines that generate timing based on a specific FPS; usually 30. Many older console games on the NES and SNES suffered from this issue. If the frame rate dropped, the game would stutter and slowdown, but the opposite was also true. If a user modified the binary of a game, a very popular activity with emulators, then the entire game would actually speed up along with the framerate, again making the game unplayable to the end user.

The concerns for efficient rendering performance are always prevalent in game engine design, but mobile applications and web applications present additional obstacles that must be accounted for. In flash, the developer has two main methods of generating objects on the screen. The first is to use the built in Sprite class. This class allows a user to render either vector or rasterized (bitmap) data. Vector graphics occupy very little memory since they are represented by a series of vertices, but this trade off in RAM reduction comes with a cost. As with most software based optimizations, processing power is traded for a reduction in memory usage, and vice versa. Using vector graphics results in using a fraction of the RAM required for bitmaps, but it is heavily CPU dependant. For mobile platforms, vector rendering performance is simply too taxing on the hardware to be useful. On the desktop as well as on the internet (flash), vector rendering can be used, but is still noticeably slower then rendering bitmap data when a large amount of objects are simultaneously displayed. IsoMob uses Bitmap graphics for all of its on screen assets and a full performance analysis is presented in Chapter 5.

3.6.1 Depthsorting

Depthsorting is a feature that is necessary in any three-dimensional or pseudo three-dimensional landscape. Keeping track of where a unit lies on the third dimension is usually handled by a z-buffer. The buffer gets its name because it is based on the depth of a pixel into the monitor, and this axis is generally labeled as the z-axis. The game engine uses a simulated z-buffer in order to depth sort all in game

entities. The index of an entity along the z-axis is known as the z-index, and is calculated as equation 3.13.

$$z_{index} = \frac{grid_{height}\left((grid_{width} - cell_x + 1) + (cell_y * grid_{width} + 2)\right) + cell_z}{grid_{width} * grid_{height} * grid_{depth}} \quad (3.13)$$

The idea behind calculating the z-index is that display depth increases in the positive y and negative x direction. This is visualized in Figure 3.21.

As the unit moves closer to zero along the x-axis the $cell_x$ decreases, which explains why the value is negative in the equation. At the same time, the $cell_y$ value is positive in the equation because the depth increases as an entities position along the y-axis increases. The constants in the equation are present so that when an entity is positioned at (0,0) it will not be placed in the very back of the depth sort. This is applied because (0,0) is not furthest back in the depth sort. The location in any scene with the lowest depth is $(x_{maximum}, 0)$ where $x_{maximum}$is the max value along the x-axis.

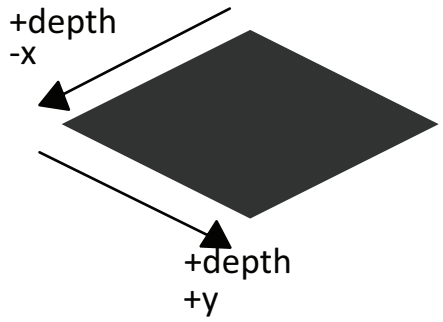

Figure 3.21: Calculating depth with a z-index.

CHAPTER 4

ARTIFICIAL INTELLIGENCE

IsoMob is designed as a single player game, and as such, requires an intelligence system that can challenge the end user. All characters in the game, except for the playable hero, have a subset of the available engine AI in order to perform their required functionality. The AI in IsoMob can be broken down in simple descriptions as shown in Table 4.1. These AI descriptions were made popular in the boids demonstration by Craig Reynolds in 1986[16]. The basic AI elements that drive the enemies are wrapped inside of a finite state machine (FSM). The FSM acts as the brain of the enemy and instructs AI behaviors based on information found in the surroundings. For example, an enemy will test the distance between itself and the user's character. If that distance is within a certain range and the enemy is in good shape, it will choose to seek towards, and attack the hero. If the enemy is weak, and has a very small chance of defeating the hero then the enemy will run away.

Table 4.1: Overview of AI actions.

Description	AI command
"Move enemy towards location (x,y,z)"	Seek
"Move enemy away from location (x,y,z)"	Flee
"Move enemy to location if they do not run into a wall"	Wall Avoidance
"Move enemy to location if they do not run into an object"	Object Avoidance
"Keep enemy A and it's heading close to that of enemy B"	Formations
"Move enemy to point A, then to point B"	Path Finding

4.1 Steering Behaviors

The force behind moving all in game AI controlled enemies is the steering force. The steering force is influenced by indivdual steering behaviors. The concept of steering behaviors is to produce emergant behavior. Emergant behavior is described as behavior that is not explicity defined by a rule or the system itself, but still occurs. An example of emergant behavior is an ant colony. In ant colonies, ants react to chemical stimulation. They may not know where they are going or how far they have to go, but by following chemical scents the entire colony can travel in what appears to be a very organized line. The steering behaviors themselves are very simply but when compunded together they create a competant AI.

4.1.1 Seek

The first AI component required is a method to move an entity towards a location. Seek allows any IsoCharacter to move to a point in the isometric world. Seek by itself does not provide any sort of intelligent movement mechasims, meaning that it will travel in a straight line to the specified destination. The Seek behavior provides a force to apply to an in game entity in order to steer it towards the destination. Determining this force involves finding the vector between the entity and the target location as seen in Figure 4.1.

The first thing the seek behavior does is calculate the velocity required to put the entity at the target location. This velocity $\overrightarrow{Velocity}_{desired}$, is calculated by subtracting the entites current location $\overrightarrow{Position}_{current}$ from the target location $\overrightarrow{Position}_{target}$ as seen in equation 4.1.

$$\vec{Velocity}_{desired} = \vec{Position}_{target} - \vec{Position}_{current} \tag{4.1}$$

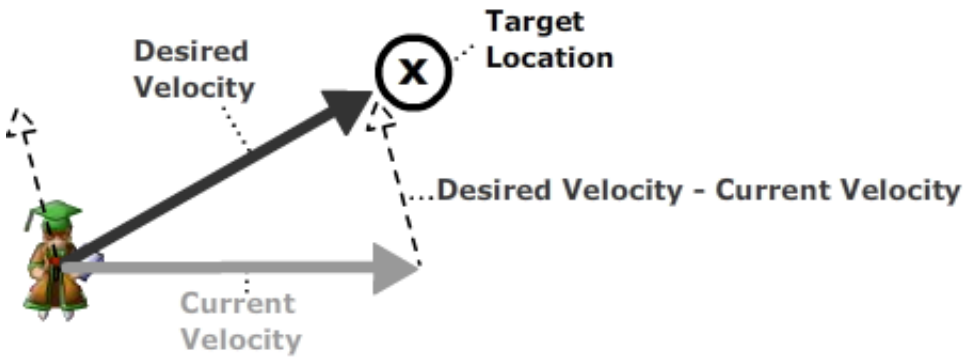

Figure 4.1: The seek behavior calculates the desired velocity to a target location as well as the steering force required to move towards that target.

The seek steering force $\overrightarrow{Velocity}_{seek}$ is then calculated by subtracting $\overrightarrow{Velocity}_{current}$from the desired velocity $\overrightarrow{Velocity}_{desired}$ as shown in equation 4.2. This force is then applied to an entities overall steering force, by doing so at a cyclic rate the target will reach its destination.

$$\overrightarrow{Velocity}_{seek} = \overrightarrow{Velocity}_{desired} - \overrightarrow{Velocity}_{current} \tag{4.2}$$

4.1.2 Flee

The flee component is very similar to the seek behavior, except that, it is the exact opposite. The calculation for flee is the exact same as that of seek as seen in equation 4.2, the only difference is that $\overrightarrow{Velocity}_{flee}$ is the additive inverse of $\overrightarrow{Velocity}_{seek}$. The calculation for flee is defined in equation 4.3.

$$\overrightarrow{Velocity}_{flee} = -\overrightarrow{Velocity}_{seek} \tag{4.3}$$

4.1.3 Avoiding obstacles

The ability to move objects around the isometric world is a great starting point for enemy AI, but with seek and flee alone an entity has no way of interacting with its enviroment. Without looking for walls, objects and other entities an IsoEnemy will move through all items in its path as it seeks and flees. For example, if a pit of spikes is in an entites seek path then the entity will fall into the pit of spikes, as shown in Figure 4.2.

The end user will view this action as very "dumb" AI, as no human player would ever blindly walk into a pit of spikes. The solution to this is to have the entities scan their surroundings for potential obstacles. This can be broken down into two categories, wall avoidance and object avoidance.

All walls in IsoMob are represented by a vector $\overrightarrow{V}_{wall}$ along either the y-z or x-z plane, so collision detection can be modeled as an intersection between two vectors. This is achieved in a similar manner to how a cat uses whiskers to navigate in the dark. Game entities project out whiskers in order to feel around for walls inside an IsoScene. Figure 4.3 shows an example of a cat projecting three feelers in order to search for any potential nearby walls.

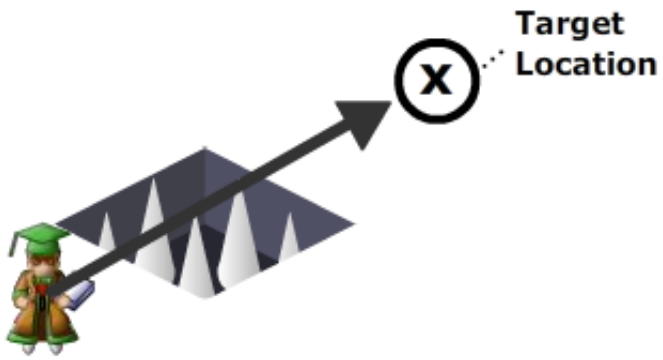

Figure 4.2: Movement without enviromental knowledge.

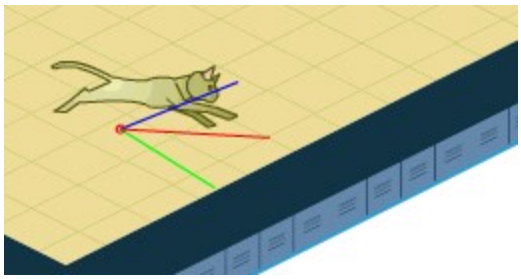

Figure 4.3: Feelers used to detect an intersection between two lines.

The first feeler is set as $\overrightarrow{Velocity}_{heading}$ since we know that $\overrightarrow{Velocity}_{heading}$ determines what direction the entity is facing. This feeler then multiplies $\overrightarrow{Velocity}_{heading}$ by a static value $length_{feeler}$ in order to ensure the feeler projected far enough away from the entity to appear realistic as shown in equation 4.4.

$$Feeler_b = \overrightarrow{Velocity}_{heading} * length_{feeler} \tag{4.4}$$

Additional feelers are then created by rotating $\overrightarrow{Velocity}_{heading}$ at any desired angle. The feelers in Figure 4.3 are each rotated 45° from $\overrightarrow{Velocity}_{heading}$, the first is rotated counter-clock wise and the second is rotated clockwise using a rotation matrix as shown in equation 4.5 and equation 4.6.

$$\begin{bmatrix} x_{feeler_a} \\ y_{feeler_a} \end{bmatrix} = \begin{bmatrix} x_{feeler_b} \\ y_{feeler_b} \end{bmatrix} * \begin{bmatrix} \cos(-45°) & -\sin(-45°) \\ \sin(-45°) & \cos(-45°) \end{bmatrix} \tag{4.5}$$

$$\begin{bmatrix} x_{feeler_c} \\ y_{feeler_c} \end{bmatrix} = \begin{bmatrix} x_{feeler_b} \\ y_{feeler_b} \end{bmatrix} * \begin{bmatrix} \cos 45° & -\sin 45° \\ \sin 45° & \cos 45° \end{bmatrix} \tag{4.6}$$

There is no limit on the number of feelers used, or exactly how they are projected. It's up to the developer to choose feeler length, number of feelers, and angle of feeler projections to best fit their game design. Although there is no limit on the

number of feelers remember that each feelers requires an additional line segment intersection calculation, so it is potentially limited for performance reasons.

When an intersection ocurs between feelers and a wall a steering force is generated to move the entity away from the wall. The magnitude of the force the wall applies to the entity is determined by the normal of $\overrightarrow{V}_{wall}$. Collisions occur in three dimensions so calculation of the wall normal $\overrightarrow{V^{\perp}}_{wall}$ can either be an inward-pointing normal or an outer-pointing normal as shown in Figure 4.4.

Determing the force of a wall on an entity when a collision occurs is determined by how a feeler penetrates a wall. In order to find this force we first must test for segment intersections of the feeler and the wall. We create line segments from the entities position $P_{entityPosition}$ to its feeler extension $P_{entityFeeler}$, and also from the start point of a wall $P_{wallStart}$ to its end point $P_{wallEnd}$. Using these line segements we then calculate the intersection point if it exists and the distance from the entities position$P_{entityPosition}$ to the intersection point. We store the distance so

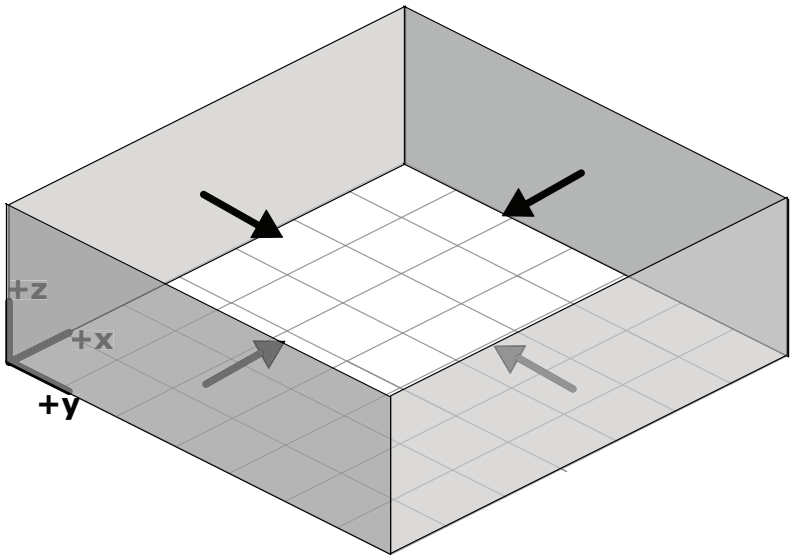

Figure 4.4: Normal Vectors for each wall type.

that we can check against all nearby walls and keep track of which wall collision is closest to the entity. Only the closest wall will apply its steering force back on the entity, and all other collision information will be discarded. A visualization of this-model is seen in Figure 4.5.

For the equations below the following is defined:

$$\begin{aligned} P_1 &= P_{wallStart} \\ P_2 &= P_{wallEnd} \\ P_3 &= P_{entityPosition} \\ P_4 &= P_{entityFeeler} \end{aligned}$$

The equations of the lines are defined in equation 4.7 and equation 4.8, where there are two unknowns u_a and u_b.

$$line_a = P_1 + u_a (P_2 - P_1) \tag{4.7}$$

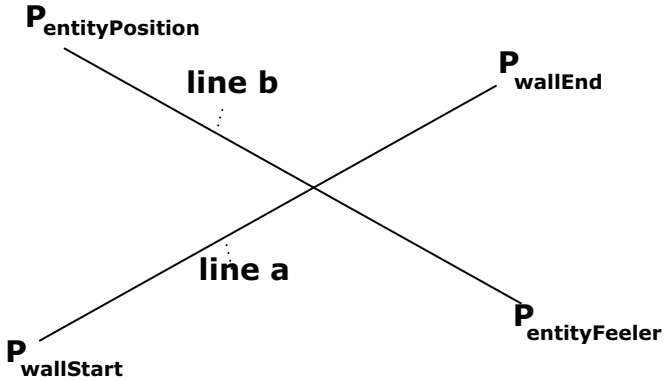

Figure 4.5: Two dimensional line segment intersection.

$$line_b = P_3 + u_b\,(P_4 - P_3) \tag{4.8}$$

The unknowns in the equation represent a percentage along their corresponding lines, an example for $line_a$ is seen in Figure 4.6.

We then solve for the point where $line_a = line_b$, which will solve for the two unknowns u_a and u_b.

$$x_1 + u_a\,(x_2 - x_1) = x_3 + u_b\,(x_4 - x_3) \tag{4.9}$$

$$y_1 + u_a\,(y_2 - y_1) = y_3 + u_b\,(y_4 - y_3) \tag{4.10}$$

Solving then gives the following expressions for u_a and u_b and gives equation 4.11 and equation 4.12.

$$u_a = \frac{(x_4 - x_3)\,(y_1 - y_3) - (y_4 - y_3)\,(x_1 - x_3)}{(y_4 - y_3)\,(x_2 - x_1) - (x_4 - x_3)\,(y_2 - y_1)} \tag{4.11}$$

$$u_b = \frac{(x_2 - x_1)\,(y_1 - y_3) - (y_2 - y_1)\,(x_1 - x_3)}{(y_4 - y_3)\,(x_2 - x_1) - (x_4 - x_3)\,(y_2 - y_1)} \tag{4.12}$$

Substituting either u_a or $u_b$into its corresponding equation all posibilites of line types. Note that the denominators for both u_a and u_b are the same, i.e. $u_{a_{denominator}} = u_{b_{denominator}} = u_{denominator}$. When the denominator is equal to

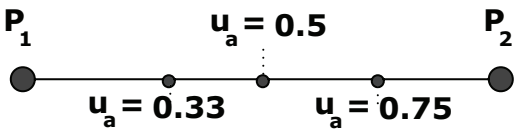

Figure 4.6: Different values of u_a representing potential intersection points.

zero,$u_{denominator} = 0$, the two lines are parallel and no intersection occurs. This case is handled by exiting the intersection test with a null value. If we perform the parallel line check then we do not need to check that the two lines are coincident. The two lines will be coincident when the numerator and denominator are equal to zero, so the lines can not be coincident without being parallel. We also need to check that the intersection actually falls on the tested line segment and not outside of it. This is done by checking the values of the unknowns u_a and u_b and is summarized in Table 4.2.

Once we have the intersection point we determine its distance from the entity performing the collision represented by $P_{entityPosition}$. The distance is calculated by finding the Euclidian distance between the entity $P_{entityPosition}$ and intersection $P_{intersection}$ points multiplied by u_a as seen in equation 4.13.

$$distance = \sqrt{(x_{intersection} - x_{entityPosition})^2 + (y_{intersection} - y_{entityPosition})^2} * u_a \quad (4.13)$$

The four types of wall collisions are first seperated by determining if the wall is aligned horizontally or vertically. After we know the wall orientation then we test which direction the entity is traveling in when it hits the wall. By using the entites

Table 4.2: Determining when an intersection lies within or outside of a line segment.

u_a	u_b	Intersection type
$1 > u_a > 0$	$1 > u_b > 0$	Within both line segments
$u_a \geq 1$	$1 > u_b > 0$	Outside of line segment A
$u_a \leq 0$	$1 > u_b > 0$	Outside of line segment A
$1 > u_a > 0$	$u_b \geq 1$	Outside of line segment B
$1 > u_a > 0$	$u_b \leq 0$	Outside of line segment B

heading vector $\overrightarrow{V}_{heading}$ we can determine the direction of the traveling entity. The x-component of $\overrightarrow{V}_{heading}$ is tested against vertical walls. When $\overrightarrow{V}_{heading_x} < 0$ then we know the collision is occuring against a southern wall, and when $\overrightarrow{V}_{heading_x} \geq 0$ the collision occurs on a northern wall. Similarly, when $\overrightarrow{V}_{heading_y} < 0$ then we know the collision is occuring against a eastern wall, and when $\overrightarrow{V}_{heading_y} \geq 0$ the collision occurs on a western wall. These calculations give us the direction of the steering force that moves the entity away from a wall, preventing the collision and are summarized in Table 4.3.

The normals of all IsoWalls are stored as normalized vectors, meaning the magnitude of the steering force needs to be determined seperately. The magnitude of the force is calculated by how deep the projected feeler $\overrightarrow{V}_{feeler}$ penetrates the vector that represents the wall $\overrightarrow{V}_{wall}$ as illustrated in Figure 4.7.

This calculation is done by subtracting the intersection $\overrightarrow{V}_{intersection}$ from the feeler $\overrightarrow{V}_{feeler}$, resulting in the overshoot vector $\overrightarrow{V}_{overshoot}$ as seen in equation 4.14.

$$\overrightarrow{V}_{overshoot} = \overrightarrow{V}_{feeler} - \overrightarrow{V}_{intersection} \tag{4.14}$$

Table 4.3: Summary of steering force directions applied to an entity during a wall collision.

Steering force direction	Wall direction	Entity heading
$\vec{V}_{direction} = -\vec{V}^{\perp}_{wall}$	vertical	$\vec{V}_{heading_x} < 0$
$\vec{V}_{direction} = \vec{V}^{\perp}_{wall}$		$\vec{V}_{heading_x} \geq 0$
$\vec{V}_{direction} = -\vec{V}^{\perp}_{wall}$	horizontal	$\vec{V}_{heading_y} < 0$
$\vec{V}_{direction} = \vec{V}^{\perp}_{wall}$		$\vec{V}_{heading_y} \geq 0$

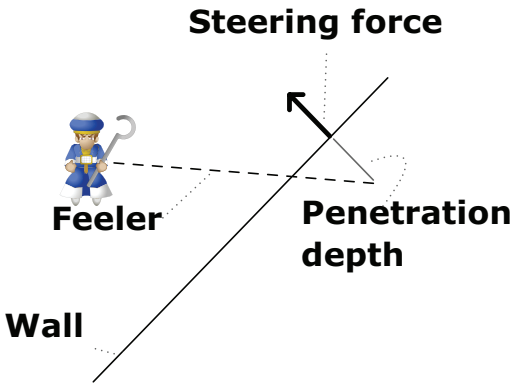

Figure 4.7: Detecting wall collisions. The magnitude of the steering force applied by the wall to the entity is equal to the penetration depth of the feeler $\overrightarrow{V}_{feeler}$ into the wall vector $\overrightarrow{V}_{wall}$.

The magnitude of the steering force is then calculated by multiplying the steering force by the length of the overshoot as seen in equation 4.15.

$$\overrightarrow{V}_{steeringForce} = \overrightarrow{V}_{direction} * length_{overshoot} \tag{4.15}$$

The steering force $\overrightarrow{V}_{steeringForce}$ is the force that will be applied to the entity in order to avoid the wall collision.

Detecting wall collisions is based on intersections between two vectors, but using a vector to model a game entity does not work very well. A better solution is to represent a game entity in a bounding box. The bounding box is then tested for collisions in a similar manner to that of walls. The entity testing for collision will again project a feeler that tests against all other entities, represented as circles. This modeling is visualized in Figure 4.8.

Using circles for collision detection between game entities is very fast and reasonably accurate. For a game world with mostly open spaces and no tight corridors this method works very well. If a tight path around objects is required this

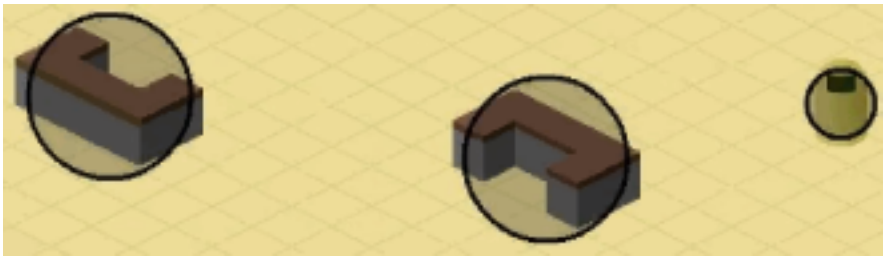

Figure 4.8: Modeling of in game entities as circles for object collision detection.

approximation method will perform poorly. The major issue with this method is that entities will get stuck when there are small spaces in between obstacles. In this case the entity may behave erratically when trying to move away from obstacles. From Figure 4.8 one can see that the method of using circles for collision detection is not perfect. The circles may protrude past the actual objects or conversely may not cover the entire object they are approximating. These small inconsistencies can be overlooked since an entity will always try to maintain the feeler distance away from the object. If a circle fails to cover enough area so that the feeler length cannot perform sufficiently then the collision detection will fail.

The steering force involved with object collisions is calculated in a similar manner to that of a wall collision. Instead of finding the intersection between two line segments we test for an intersection of a line segment and a circle. Each game entity defines a radius to use for its bounding collision circle to simply object collisions. This circle is then used by other entities to test for a collision as seen in Figure 4.9.

This is a common simplification that makes testing for object collision with many obstacles feasible. This method, is not looking for and, does not produce pixel perfect collision. Circle approximations are used to get a general idea of where something is in the world, and to stay away from it when navigating with AI. Other

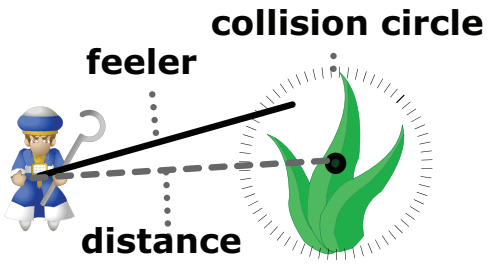

Figure 4.9: Detecting object collisions.

methods such as bitmap based collision detection will give very accurate results, at the cost of a large performance hit. A feeler $\overrightarrow{V}_{feeler}$ is again projected in front of an entity aligned with its heading vector $\overrightarrow{V}_{heading}$. The feeler $\overrightarrow{V}_{feeler}$ checks for intersections between a circle and a vector. When a collision occurs the direction of the steering force is perpendicular to the heading vector of the entity. The magnitude in this case is calculated the same way as it is for wall collisions as seen in equation 4.14 and equation 4.15.

In order to find the distance between an entity and the object its trying to avoid we calculate the in between vector $\overrightarrow{V}_{between}$ is shown in equation 4.16.

$$\overrightarrow{V}_{between} = \overrightarrow{V}_{object} - \overrightarrow{V}_{entityPosition} \tag{4.16}$$

After $\overrightarrow{V}_{between}$ is calculated we find the dot product between the entities heading $\overrightarrow{V}_{heading}$ and $\overrightarrow{V}_{between}$ as seen in equation 4.17.

$$direction = \overrightarrow{V}_{heading} \bullet \overrightarrow{V}_{between} \tag{4.17}$$

Before we test for line segment to circle intersections we first create a list of object candidates. The list of candidates is created by verifying that a game object is currently in front of the entity performing the collision detection. This is easy to

perform with a vector dot product. The dot product of an entities heading vector $\overrightarrow{V}_{heading}$ and the vector between the entity and object $\overrightarrow{V}_{between}$ will be positive if the object is forward of the facing plane, and negative if behind. A summary for how objects are determined to be collision candidates is shown in Table 4.4.

Once the list of candidates is built each object is tested for collision with the searching entity. The game engine will keep track of the closest candidate to the entity and only apply the steering force $\overrightarrow{V}_{steeringForce}$ from said closest object. In order to calculate the collision radius $r_{collision}$ necessary to actually test for a collision we obtain the intermediate vectorsthe scalar value of the dot product is multiplied to the entities heading vector $\overrightarrow{V}_{heading}$. The closest object is determined by getting the distance from the entity then testing for a line-segment to circle collision as seen in equation 4.18.

$$distance_{objFeeler} = \sqrt{(x_{object} - x_{feeler})^2 + (y_{object} - y_{feeler})^2} \tag{4.18}$$

The next step is to project vector between the entity and object $\overrightarrow{V}_{between}$ on the entities heading vector $\overrightarrow{V}_{heading}$ using the scalar value *direction* calculated us-

Table 4.4: Using the dot product to determine if an object is in front of an entity during collision detection.

direction	Angle between vectors	Result	Candidate?
$direction > 0$	$0^o \leq \theta \leq 90^o$	object is in front	yes
$direction < 0$	$90^o \leq \theta \leq 180^o$	object is behind	no
$direction = 0$	$\theta = 90^o$	object is to the side	no

ing the dot product in equation 4.17. This produces the projection vector $\overrightarrow{V}_{projection}$ as seen in equation 4.19.

$$\overrightarrow{V}_{projection} = \overrightarrow{V}_{heading} * direction \tag{4.19}$$

Figure 4.10 shows how the *distance* is projected along the heading $\overrightarrow{V}_{heading}$ of the entity. The projected feeler has the same direction as the heading $\overrightarrow{V}_{heading}$ with a larger magnitude in order to help feel around for objects.

Once we have the projection vector $\overrightarrow{V}_{projection}$ we calculate the radius of collision $r_{collision}$by calculating the distance between the center of the object's circle approximation and the end point of the projection $\overrightarrow{V}_{projection}$ as seen in Figure 4.10.

The examples shown in Figure 4.11 list the cases that can occur in game when deciding if an object collision occurs. Note that the projection $\overrightarrow{V}_{projection}$ will never be in the a situation where the radius check passes, but the length check fails. In other words, case d in Table 4.5 can never occur due to how the projection $\overrightarrow{V}_{projection}$ is created.

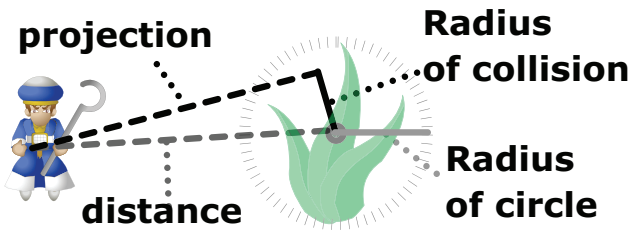

Figure 4.10: Detecting object collisions. Projecting distance along the entities heading $\overrightarrow{V}_{heading}$ to find the radius of collision $r_{collision}$.

Table 4.5: Descriptions for object collision examples from Figure 4.11.

Case	$l_{proj} < l_{feeler}$	$r_{collision} < r_{circle}$	Details
a	no	yes	Collision check fails
b	yes	yes	A valid collision occurs
c	no	no	Collision check fails
d	yes	no	This will never occur

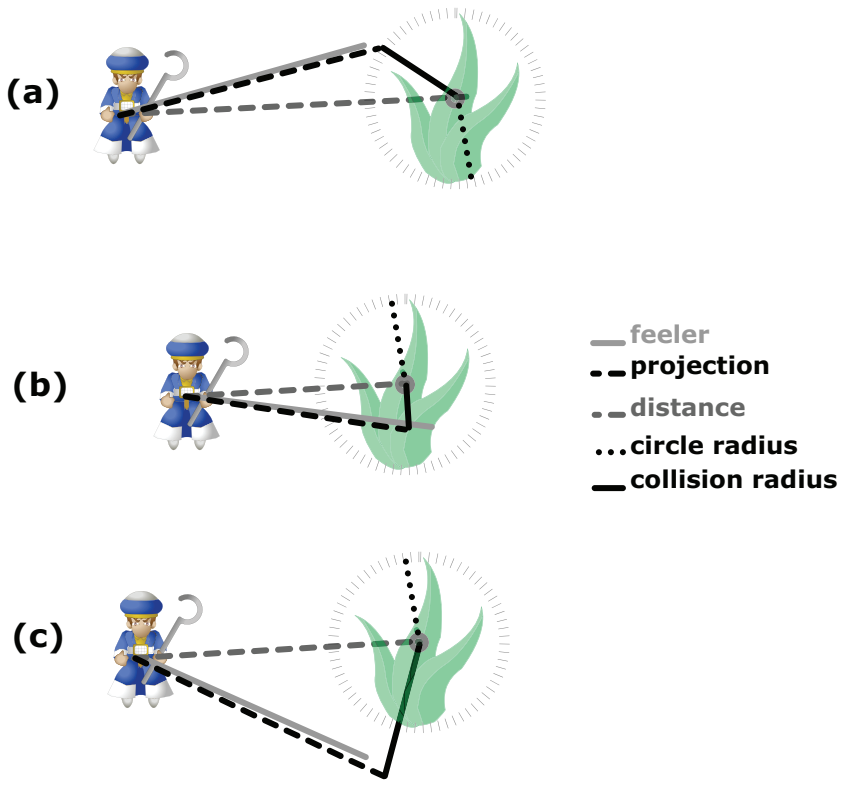

Figure 4.11: Example cases of detecting object collisions.

4.1.4 Autonomous Movement

Now that entites can navigate the environment while avoiding walls and objects they need a way to automatically wander around. If an entity can wander by itself while avoiding obstacles, then we've created a sufficient default AI behavior. Its worth pointing out that these methods can be improved on quite a bit. Currently a single feeler is used for object collisions, this can easily be extended to use three feelers as the wall detection does. During testing it was found that a single feeler provided relatively good performance, so thats why that design decision was made.

Wandering is useful in that it produces random motion that appears to be relatively intelligent. As mentioned earlier, when coupled with collision detection wandering will enable entities to navigate game worlds. One of the problems with random motion is that it can appear very jittery. The simplest approach of generating a random number each step update will produce very jitery motion that will appear very unatural to the end user. Other options are available that produce smooth random movement, such as perlin noise, but calculating this is a very CPU intensive operation. The solution presented by Craig Reynolds in the boids simulation, is that of projecting a circle in front of an entity and having that entity steer towards a random point constrained to that circle[16]. Every step update, a random displacement is applied to the constrained point, and over time, the point will move back and forth around the perimeter of the circle. This calculation does not hit the CPU very hard in terms of calculation time, and it produces a range of random motion. Three parameters drive the wander calculation; maximum jitter per step j_{step}, wander distance w_d and wander radius w_r. The wander parameters are visualized in Figure 4.12, and the jitter j_{step} defines how far the target can move in any one step.

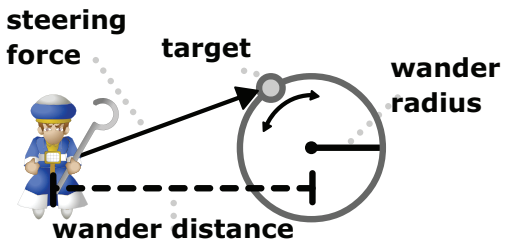

Figure 4.12: Producing random wander movement by projecting a circle in front of an entity.

To implement this in software we begin by creating the wander target $\overrightarrow{w_t}$. The wander target $\overrightarrow{w_t}$ will be updated each step by a random value multiplied by the maximum amount of jitter allower per step j_{step}. We multiply the x and y component by random numbers in order to produce the next wander target position $\overrightarrow{w_t}'$ as seen in equation 4.20.

$$\overrightarrow{w_t}' = \overrightarrow{w_t} + \begin{bmatrix} x * j_{step} \\ y * j_{step} \end{bmatrix} \tag{4.20}$$

$$\begin{aligned} where: -j_{step} &\leq x \leq j_{step} \\ -j_{step} &\leq y \leq j_{step} \end{aligned}$$

Now we have the information we need to update the wander target $\overrightarrow{w_t}$ and begin generating the steering force $\overrightarrow{V}_{steeringForce}$ to direct the entity towards the new displacement of the wander target w_t. We begin by discarding the magnitude of the wander target $\overrightarrow{w_t}$, by normalizing it. The only pertinent information is the direction of the vector, since that is used as the projection on the unit circle. Using

the normalized vector we can then project it onto the chosen circle that constrains the wander target. This is done by multiplying the normalized wander target $\overrightarrow{\hat{w}_t'}$ by the magnitude of the wander radius $\|w_r\|$ as seen in equation 4.21.

$$\overrightarrow{V}_{steeringForce} = \frac{\overrightarrow{w_t'}}{\left\|\overrightarrow{w_t'}\right\|} * \|w_r\| \tag{4.21}$$

The steering force $\overrightarrow{V}_{steeringForce}$ can then be applied to the entity and it will randomly wander around in game. Note that wandering without object and wall collisions will lead to the entity being able to walk through any object in the game.

4.1.5 PathFinding

The basis for all IsoMob formations are that of flocking birds. The classic rule set for boids is defined as follows[16]:

- Seperation - if an agent is too close to another agent, then steer away.
- Alignment - steer towards the average heading of the nearest agents.
- Cohesion - adjust the speed of an agent to match that of the nearest agents.

These rules along with the other AI methods listed above provide all the functionality needed to simulate a real group of objects. These methods enable the engine to handle instructions such as "group up with the other enemies and attack", "all enemies retreat and regroup!" and "protect me from attacks!". For example, these high level coordination techniques could be used by an in game boss to control the enemies to do his bidding.

4.1.6 Finite State Machine

The finite state machine groups all of the above mentioned AI techniques into a simulated "brain" for IsoMob enemies. The enemies use data gathered from the player, other enemies, and other aspects of the game world in order to make decisions. These decisions break down into nothing more then sequences of the simple AI techniques. For example, a possible state in the FSM could be to retreat. Retreating may involve attempting to put any object in between the retreating enemy and its target it deems as a threat. The first action the entity may take is to find the nearest locations that provide cover. Once a suitable location is found the enemy will find a path to the desired location, and finally, seek along the points of the path to reach an end goal. An example of a higher level AI FSM is shown in Figure 4.13.

This FSM could is used in the sample application to control enemy movement. Enemies will wander around searching for the player, attack the player, and run the opposite direction when their health is low. Finite state machines work very well for controlling enemy AI since they can easily be extended with additional states and scale very well. The AI itself can be broken down into multiple level's of

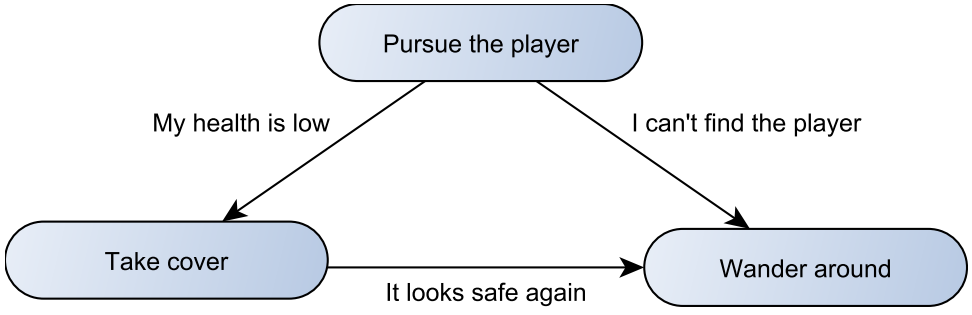

Figure 4.13: Example of a high level state machine to control enemy AI.

state machines starting with very high level and moving down to the lowest level necessary. An example of the logic inside of a FSM state is shown in Figure 4.14.

It is important to note that the built in AI methods in IsoMob are meant to be controlled in some sort of FSM. The skeleton structure of the FSM is included in the game engine, and it is up to the end user to decide exactly how they want their game enemies to function.

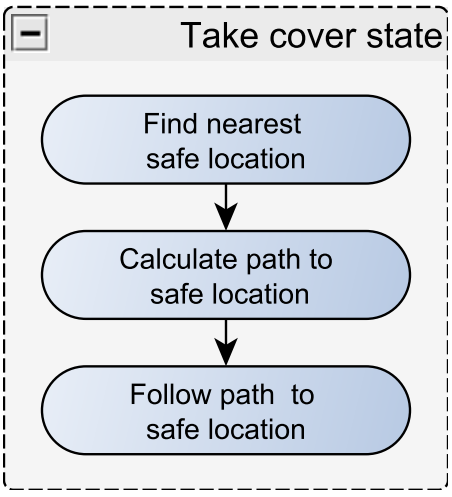

Figure 4.14: Example logic inside a high level AI state machine state.

CHAPTER 5

IMPLEMENTATION

5.1 Design Goal

The goal of IsoMob is to run an isometric game with multiple on screen entities at an effective frame rate of 30 frames per second (fps). With this in mind a sample application was built on top of IsoMob in order to evaluate how well the engine performs. The application is a game that utilizes the engine in order to present short bursts of play time, to coincide with other mobile applications. The game itself allows a user to play on the internet, on a desktop PC, and on an Android smartphone with the exact same codebase. All platform dependancies are satisfied by the IsoMob engine itself.

The most important aspect of a game engine is the performance. The key areas of interest regarding performance for IsoMob are total CPU utilization, RAM in use, and GPU texture swaps. For mobile development in particular, deciding how to store textures in GPU memory is especially important. The design of how to store textures, and when to perform swaps becomes a key point of interest. This turns out to be the largest drive in overall frames per second achieved.

5.1.1 Motivation

The sample application is an isometric shooter game that is similar to some of the very first isometric games. The game is not a clone of any other game, but if it had to be compared, it would be a cross between zaxxon and smash tv. The iso-

metric view point is very interesting in games, it gives a pseudo three-dimensional feel while keeping many of the simplicities of a pure two-dimensional game. Because IsoMob aims to be used for mid level applications (see Section 1.1) the sample application includes a large number of active game enemies and even more projectiles. At certain points in the sample application there are more then two hundred bullets on screen at the same time with approximatley 10 enemies performing collision detection and running AI state machines. The large number of active objects is meant to realisticaly model a real mobile application that someone would find on the Android Market. In summary, the application is meant to prove that IsoMob is a viable platform for mobile game development.

5.2 Optimizations

Back in the 1980's video game designers had a relatively small amount of resources to work with. The original Isometric engine, Filmation's first game KnightLore, worked with 48kb of ROM and significantly less RAM [11]. In order to operate at a high framerate the filmation engine relied on many tricks in order to conserve ROM and RAM. The design consisted of data that was split into numerous sections, stored in bit masks and contains functionality embedded into the data itself [17]. The design conserved every last bit and implemented game data structures with very little overhead. Similar, less extreme, measures were taken with the IsoMob engine in order to ensure great performance on the mobile platforms.

One issue that come up with the IsoMob design is that a scene could be filled with a very large number of bullets. The number of active characters an in Isomob scene is, on average, five. On a regular basis, the number of active bullets in a scene reached around two hundred. The original design for all in game bullets was to use a class to represent the bullet type. This was found to be a large impact

on performance, and instead, bullets are planned to be managed with a single array of integers. The data inside this array of integers would be parsed, and in a similar manner to the original filmation engine, the bullets functionality is embedded in the data itself.

5.3 Frames Per Second (FPS)

Numerous tests meant to simulate a real application built on top of the IsoMob engine were conducted to determine overall performance. The standard for games is frames per second and all code profiling and optimization were based around increasing the performance in terms of FPS. Optimizing in terms of FPS means two things: increasing the FPS, and increasing the number of game objects in use without impacting FPS.

The first method was tested with a static IsoMob scene with the exact same amount of characters, objects, textures, and bullets for each test run. The minimum, maximum, and average FPS after 120 seconds of game play were calculated in order to evaluate performance. If the performance benefit resulted in a simultaneous degradation in game play then the code change was rejected.

The second method for testing FPS involved adding additional game objects to an IsoMob scene while attempting to minimize the drop in FPS. For this case, the goal was to allow an increased amount of game objects to exist while minimizing the impact on FPS. These tests were run in order to help determine maximum values for various aspects of the game engine. For example, using this test method, it was possible to determine the maximum number of bullets simultaneously active in a scene at any one time.

5.4 Centralized Game Loop

The IsoMob game engine needed some to synchronize various events during game play. A centralized game loop exists in the engine in order to facilate events that happen in steps over a period of time. In the sample application, the frame rate was locked at 30 FPS, and this timing (33 milliseconds) was used as the base game counter. An easy way to visualize the game counter is that it is logically equivalent to a timed ISR. Generally most embedded systems are driven by an interrupt that fires every x seconds in order to provide the base counter for a simple task scheduler or an operating system; the game counter does exactly that for the IsoMob engine.

The central game loop allows a user to easily add timed events with a call to the function `add(task:IPeriodicTask)`. All tasks must implement the interface `IPeriodicTask`, so they can be managed regardless of exactly what needs to run. The structure of the tasks is shown in Figure 5.1

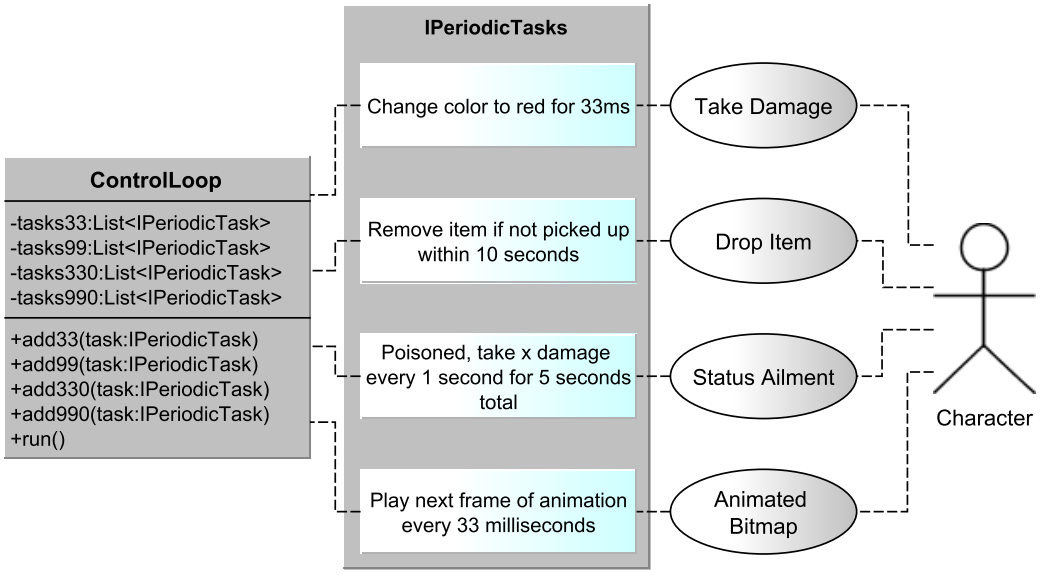

Figure 5.1: How tasks are added to the centralized game loop.

The `IPeriodicTask` contains only an `update():Bool` function that is called at the desired interval. In the case of the sample application the interval choices are: 33,99,330, and 990 milliseconds. If the `update()` function returns a value of 1 then it means the task is complete, otherwise the task will run again on the next interval. The game loop class itself, `ControlLoop`, automatically disposes of tasks that are no longer needed. This is performed by reading the return value of the `update()` function, as seen in Figure 5.2.

The game loop class is a singleton object and by using the singleton design pattern all game modules can easily get the single created instance. The game loop provides a single interface for all things that need to be updated within the IsoMob game engine, and allows for easy creation of periodic tasks.

5.4.1 Registering Game Events

Along with the game loop which provides consistent, periodic, execution; numerous events are also required. An event is a pseudo-interrupt in that when an event is broadcast all objects listening for this event will immediatley have their event handler (think interrupt service handler) executed. This allows the game engine to do away with polling for things to occur. Many of the periodic tasks themselves are started once a certain game event takes place. For instance, a character is notified they have taken damage when a bullet object determines it has collided with said character. In this case the bullet dispatches an event straight to the character involved in the incident to let them know they have been damaged. This same event is also being listened to by the IsoScene, which will remove the bullet from the render list once it has hit something. A diagram showing this in detail is shown in Figure 5.3.

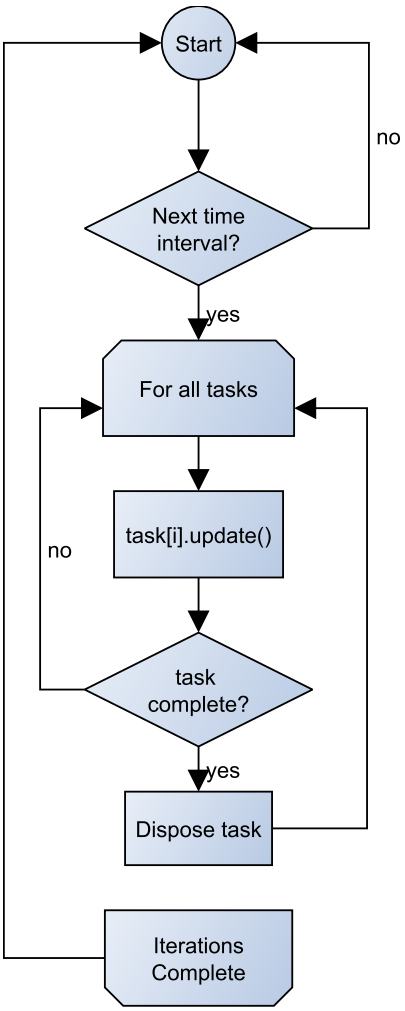

Figure 5.2: How the game loop manages tasks.

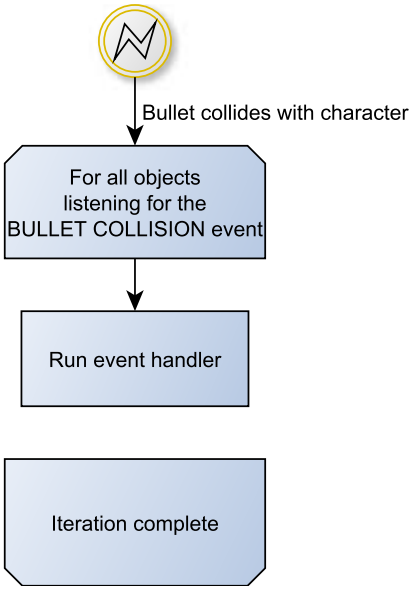

Figure 5.3: How events are handled in HaXe.

The event system is built into the HaXe language itself, and is simply used by the IsoMob engine. The engine does define specific types of events used throughout, but these are mere extensions of the built in `event` class. An example implementation with source code is shown in Listing 5.1

```
//Listen for the event to occur, multiple objects
//can listen for the same event.
//Parameter 1 = Event Type (string)
//Parameter 2 = Event Handler (function)
objA.addEventListener(BulletEvent.COLLIDE, takeDamage);
objB.addEventListener(BulletEvent.COLLIDE, showRiccochet);
scene.addEventListener(BulletEvent.COLLIDE, removeBulletFromRender)
//...

//Collision occurred!
bullet.dispatchEvent(new BulletEvent(BulletEvent.COLLIDE));

//Run objA.takeDamage()
//Run objB.showRiccochet()
```

```
//Run scene.removeBulletFromRender()
```

Listing 5.1: Example source code using events.

The source code shows a number of objects listening to the same collision event broadcast by the bullet object. Using events eliminates the need for polling, and allows multiple objects to be loosely coupled by simply listening for broadcasted events. Developing an Android application requires listening for broadcast events (detailed in Appendix C), and the method for doing so is exactly the same as what IsoMob uses within the game engine.

5.5 Performance Tests

Two types of performance tests were run on the sample application. Each test case is meant to gauge a particular aspect of the engine. The first test spawns hundreds of bullets on the screen at the same time and tests rendering performance as well as handling a scene with a very large number of objects. The second test is meant to push the AI code by spawning many enemies that wander around the stage and perform collision detection. The test cases are meant to find some maximum values for the two aspects that are most demanding: AI and rendering. Knowing these maximum values will help a user determine how they can best design their application with out worrying about performance. The test specs for each of the platforms is listed in Table 5.1.

The software packages for very sleepy and flash develop were used to help profile the code. They provided information such as: how many objects were created, how often is a function called, and how long does it take to execute a function. The software package Vizzy enabled writing trace outputs from the Flash target into a text file. This helped make the the chart graphics that are shown below; trace data was imported into excel and then used to generate the graphs.

Table 5.1: Test specifications for each of the three platforms: Adobe Flash, Windows, and Android.

<table>
<tr><td></td><td colspan="2">Adobe Flash</td><td>Windows</td><td>Android (Tablet)</td><td>Android (Phone)</td></tr>
<tr><td>Hardware</td><td colspan="3">AMD Phenom II x3 720
3GB DDR3
Nvidia Geforce 240GT</td><td>Acer Iconia A500
NVIDIA Tegra 2
Dual ARM A9 @ 1GHz
1GB RAM</td><td>HTC Incredible
QSD8650 @ 1GHz
512MB RAM</td></tr>
<tr><td rowspan="3">Software</td><td colspan="3">Windows 7 32-bit</td><td rowspan="3">Android 3.2</td><td rowspan="3">Android 2.3.4</td></tr>
<tr><td colspan="2">Adobe Flash debug player 10.3
Flash Develop 3.3 (Profiler)
Vizzy 3.8 (Trace)</td><td>Very Sleepy 0.7 (Profiler)</td></tr>
<tr><td colspan="3">Nvidia 275.33 graphic drivers</td></tr>
<tr><td>Compiler</td><td colspan="5">HaXe 2.07
NME 3.1</td></tr>
<tr><td>Resolution</td><td colspan="5">800x480</td></tr>
</table>

All performance tests were run with debug information enabled for each of the targets. Adding debug information to the various HaXe targets does add overhead, but these tests are meant to gauge how viable it would be to create an application with the IsoMob engine. Keep in mind that the non-debug version of the software will perform slightly better than these tests show. Debug information was necessary in order to collect some of the data for the tests, which is why it was compiled into the final executable.

5.6 Bullet Performance

The first test case filled the IsoScene with bullets and attempted to push the limits of the engine with sheer quanitity. Bullets performance collision detection and need to be rendered each frame since they move rapidly. The test was conducted with a special test level that fired a set of bullets from the player every 500 milliseconds. The number of bullets varied between tests and was increased until the FPS dropped significantly below the target 30 FPS goal. Each test strives to stay at 30 frames per second and can not exceed this value. An upper limit is in place for each test so the performance may be much higher than 30 FPS for some of the less stressful tests. The test level involves a single animated character (the player) in a sparsely populated room firing bullets in a full circle. The bullets are spaced evenly in the firing radius (360 degrees). This means that if we fire 18 bullets per round they will travel as a vector rotated 20 degrees from the previous bullet. This test is visualized in Figure 5.4.

Each test involved test code that would automatically fire the bullets. In order to gather the data trace statements were used to output the current frames per second calculation. The FPS value is calculated by simply incrementing a counter after each game loop where each full game loop is equivalent to a single frame. The

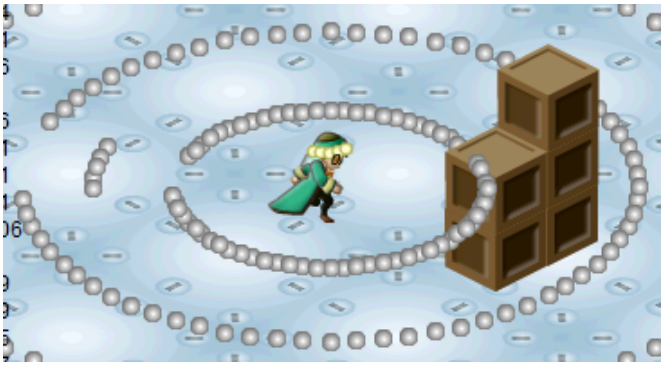

Figure 5.4: Bullet test where `x` bullets are fired per round.

FPS output is then simply the counter divided by the total time elapsed as seen in Equation 5.1.

5.6.1 Android tablet

$$FPS = counter_{frames}/time_{elapsed} \tag{5.1}$$

The results for the first bullet test run on the Android tablet can be seen in Figure 5.5.

When 18 bullets are fired each round the test application presents a very playable scenario. The FPS hovers near 30 FPS making the animation look fluid and controls responsive. Performance is very good in this test case. The next test case increases the bullets fired per round to 36 as seen in Figure: 5.6.

This test scenario presents a problem for the Android tablet target. With nearly 300 bullets on screen at the same time, the game engine can simply no longer complete what it needs within 33 milliseconds. The application is not quite a slide show but is rather unplayable. Performance at this level is not acceptable. There

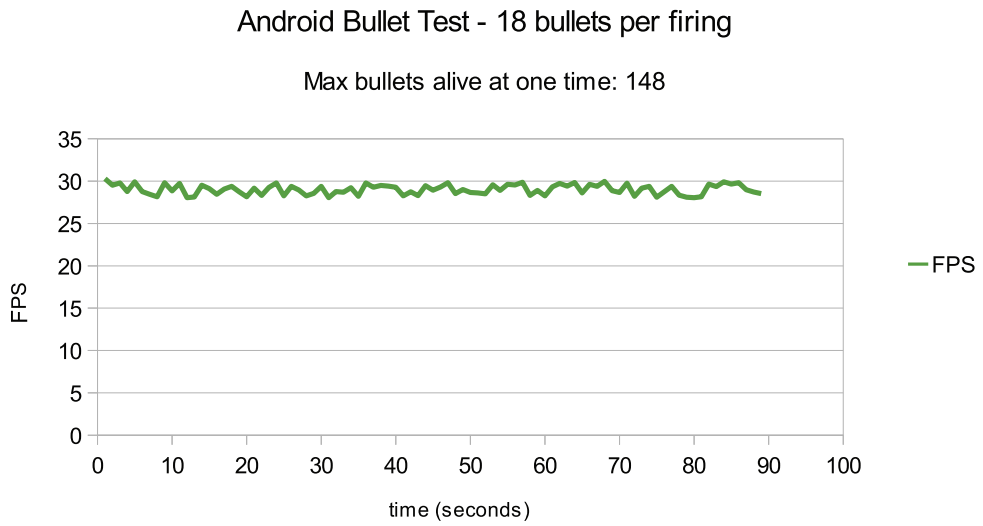

Figure 5.5: Android tablet bullet test - 18 per round.

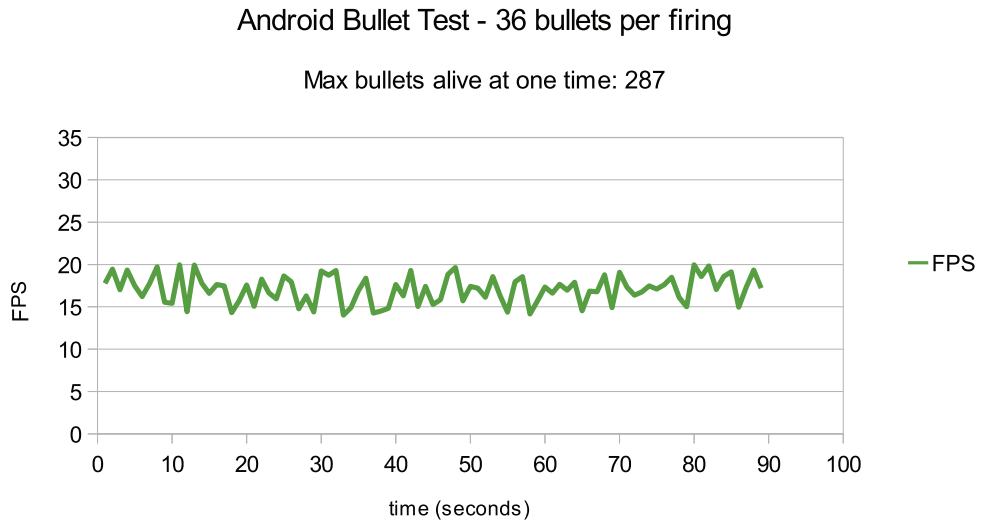

Figure 5.6: Android tablet bullet test - 36 per round.

is little hope for the test where bullets increase to 64 per round. The results are shown in Figure 5.7.

The most stressing test in this scenario is simply too much for the Android tablet target. With over 500 bullets on screen at the same time there are far too many calculations to perform in the alotted time. Luckily, this scenario would most likely never occur in a normal isometric game. There are so many bullets on screen that it is impossible to find a safe spot where your character would not collide with a bullet. Performance at this level is dismal, and its slow to even close the application.

5.6.2 Android phone

The same tests were run on the HTC Incredible with much lower overall performance results. The 18 bullet per round test results are shown in Figure 5.8.

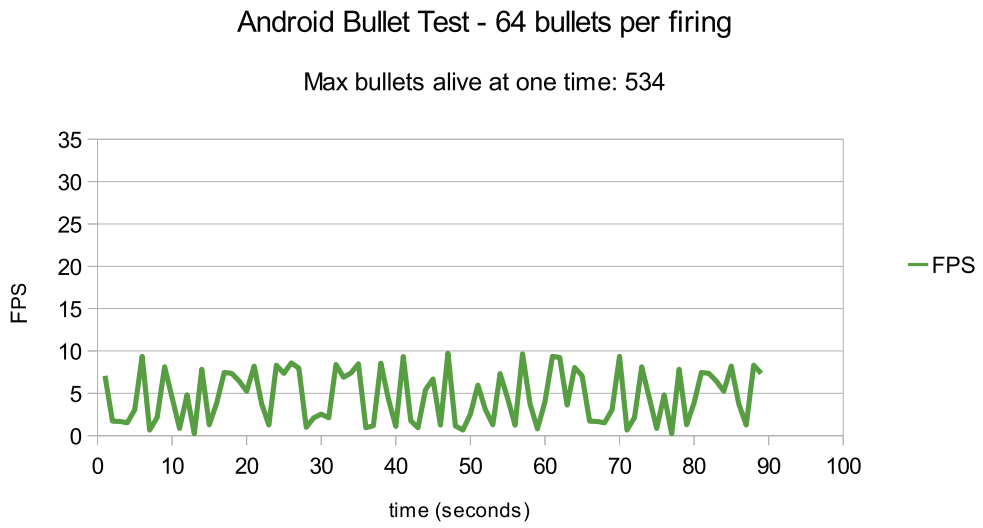

Figure 5.7: Android tablet bullet test - 64 per round.

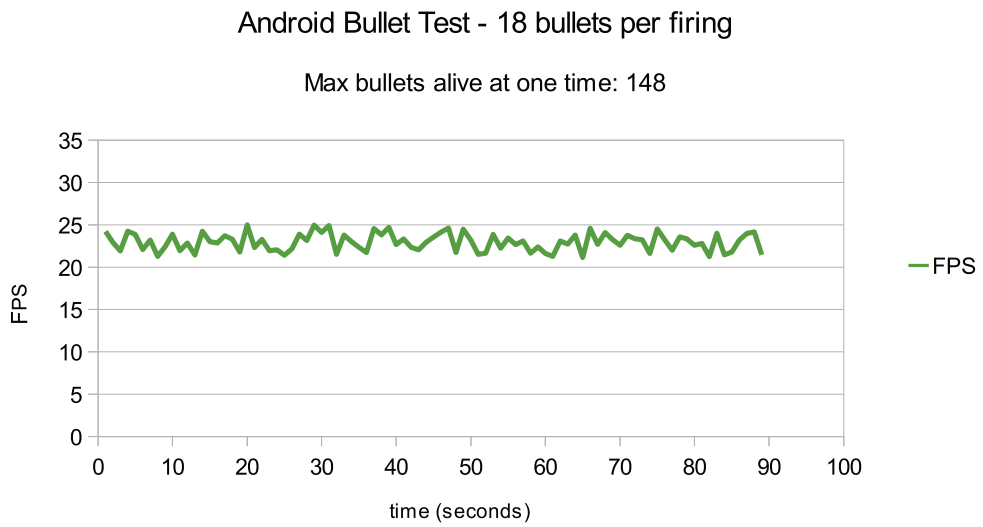

Figure 5.8: Android phone bullet test - 18 per round.

This test gives passable performance on the phone. One can already see the impact of adding this many bullets to an IsoScene from the start. The Android phone is the weakest target in terms of hardware, and the tests prove this. The next test fires 36 bullets per round, and is shown in Figure 5.9.

The phone cannot keep up with the number of on screen bullets in this test. The main suspicion on why the phone lags behind the tablet in performance is the amount of video memory. The HTC Incredible uses an Adreno 200 GPU, while the tablet uses the an ultra low power GeForce chip. The GeForce chip is much more powerful as the Adreno 200 GPU is a very poor performer. The Adreno 205 was released slightly after the 200 and it vastly increased performance.

5.6.3 Adobe Flash

The next target for testing is Adobe Flash. The Flash platform performs mostly software rendering of the scene where as the Windows and Android targets

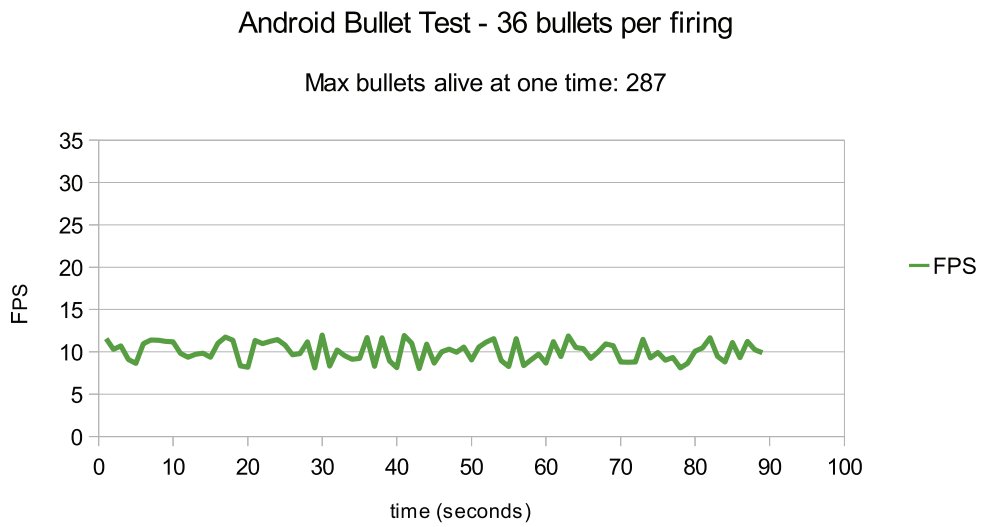

Figure 5.9: Android phone bullet test - 36 per round.

use hardware (OpenGL) in order to render the images. Flash is one of the best ways to run an online application that doesn't require a user to physically download the game files. The results actually show the frame rate going over 30 FPS, this is because of how Flash internally regulates the frame rate. The results of the first bullet test are shown in Figure 5.10.

The test for firing 18 bullets per round is simple for the Flash target, as it handles the test with ease. The test is smooth with no performance hiccups in sight. The next test is more difficult to run and the results are shown in Figure 5.11.

The test for firing 36 bullets per round runs great on the Flash target. The performance shows that this test is close to the sweet spot for balancing performance and frame rate. It's likely that an increase in bullets after this point will dramatically lower the frame rate. The results are shown in Figure 5.12.

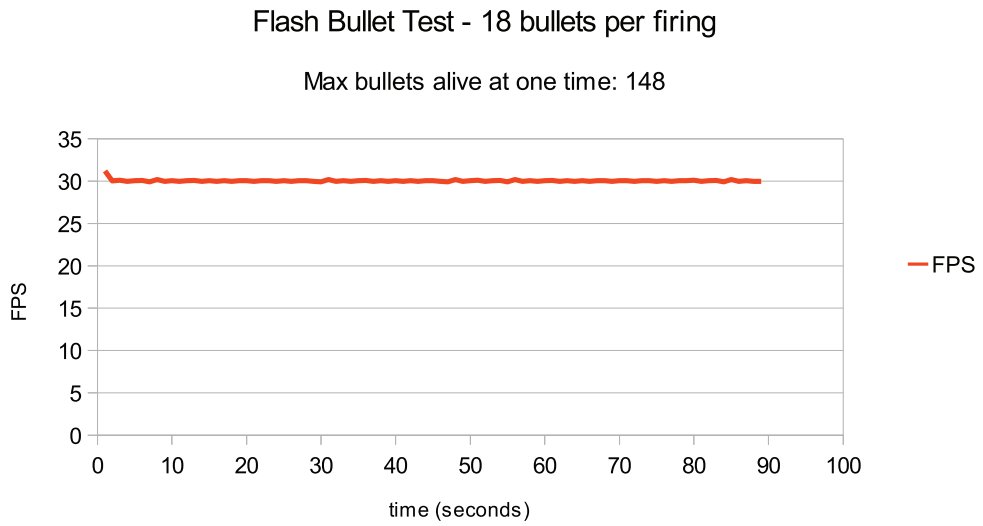

Figure 5.10: Flash Bullet Test - 18 per round.

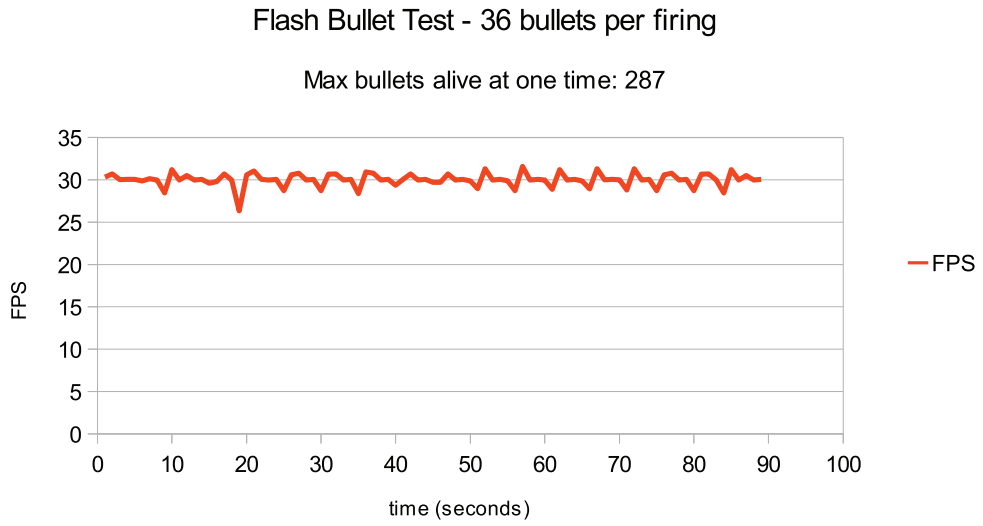

Figure 5.11: Flash Bullet Test - 36 per round.

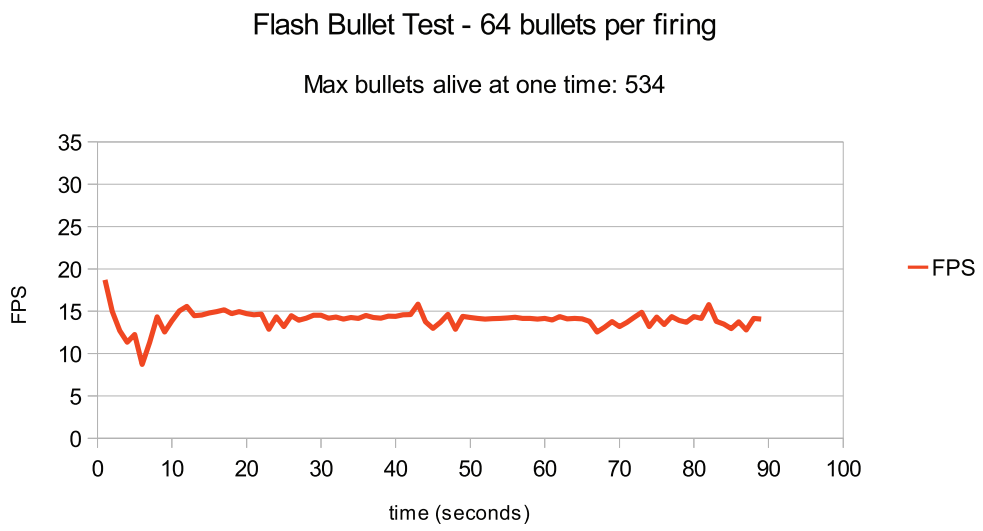

Figure 5.12: Flash Bullet Test - 64 per round.

The Flash target running on a desktop PC outperforms the Android target, but at 64 bullets per round, this test again renders the application almost unusable. The biggest slowdown occurs each time the 64 bullets are created. During these tests each bullet is represented by an object, which while fairly light in complexity, still presents a challenge when creating a large number of instances. Next up is the Windows C++ target.

5.6.4 Windows

Theoretically the Windows target should give the best performance. The desktop PC used in these tests is quite a bit more powerful than the Android tablet. Both the Windows and Android target utilize C++ as well as OpenGL to perform scene rendering. Because of these similarities it is reasonable to believe whatever is running on the faster overall hardware will perform better. The test results begin with Figure 5.13.

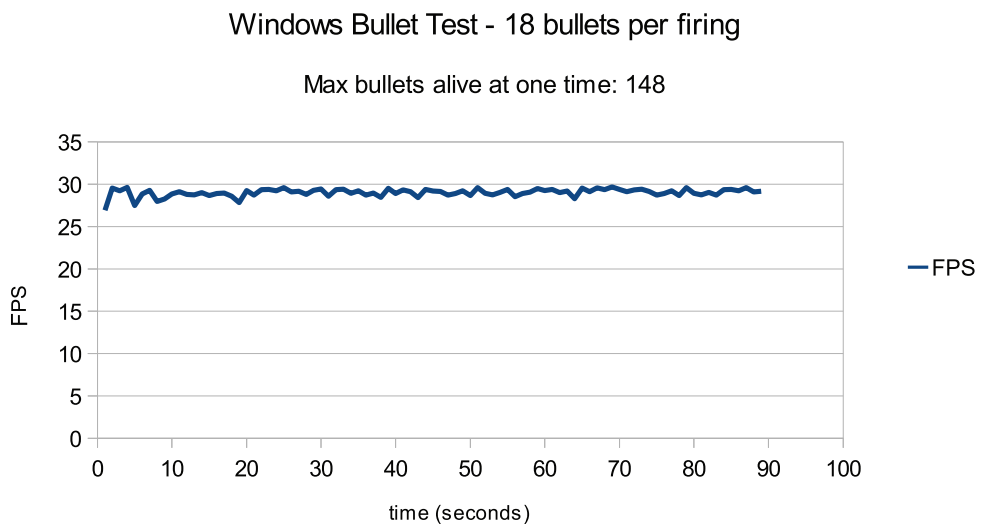

Figure 5.13: Windows Bullet Test - 18 per round.

This test, as expected, is simple for the Windows target. Both the Flash and Android target easily passed this test with great results and the Windows target is no exception. The next test is firing 36 bullets per round and is shown in Figure 5.14.

Again this test presents little trouble for the Windows target. The frame rate stays very close to 30 FPS and the game is very playable. It's interesting to see that the Flash target seems to actually perform better in this test. The results between the two platforms are very close and they easily trounce the performance obtained on the Android target. The final bullet test, with 64 bullets per round is shown in Figure 5.15.

Again, this test starts to degrade the overall performance compared to the previous test. The drop occurred on the Windows target is the least of all targets, but the decrease is still apparent. The application remains responsive, and this test still represents a passable frame rate for a real application. An additional test

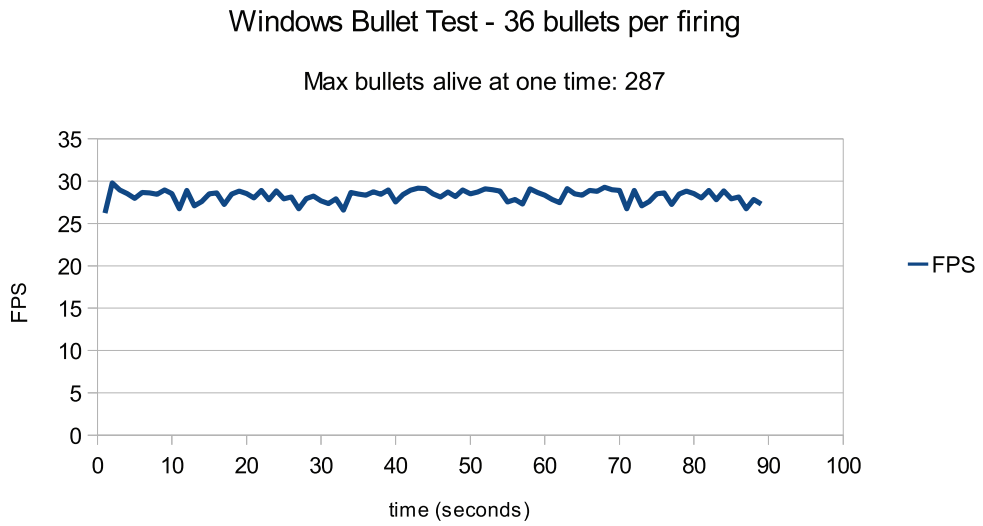

Figure 5.14: Windows Bullet Test - 36 per round.

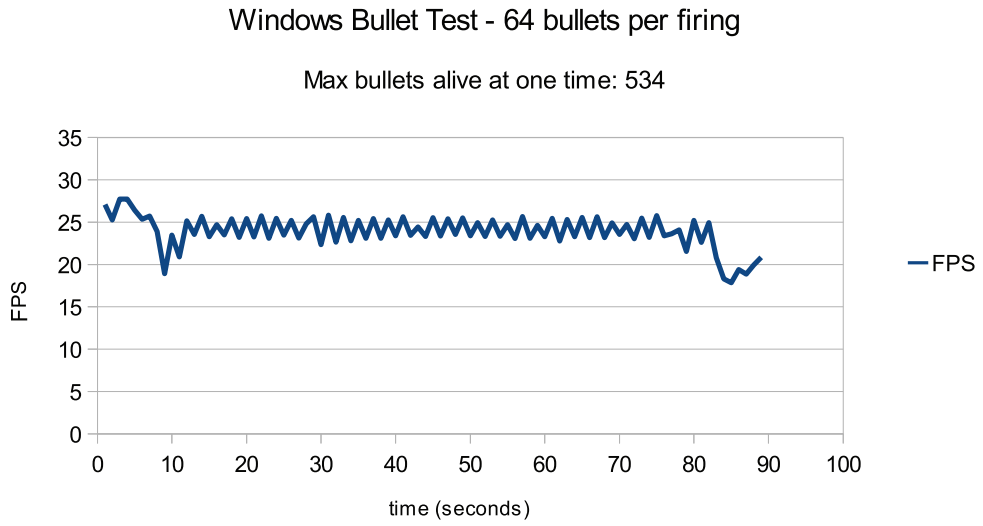

Figure 5.15: Windows Bullet Test - 64 per round.

was run on the Windows target to see how much further the performance could be pushed. This test, 72 bullets per round, is shown in Figure: 5.16.

This test shows that the Windows target was reaching a ceiling in terms of performance with the previous test. The decrease in frame rate is very close to overall percentage of total number of active bullets in the scene. This test brings the performance down far enough that this scenario is not viable for a real application. A frame rate in the low twenties is only passable in a turn based game, anything meant to play in real time will be an exercise in frustration for the end user. In this scenario there are almost 600 active bullets that require updating each frame. Each bullet has a display component (a sprite) as well as the data components which must test for collisions. This is an impressive number of instanced game objects, and even though it does not easily relate to a real world test, it gives good indication of what the engine is capable of.

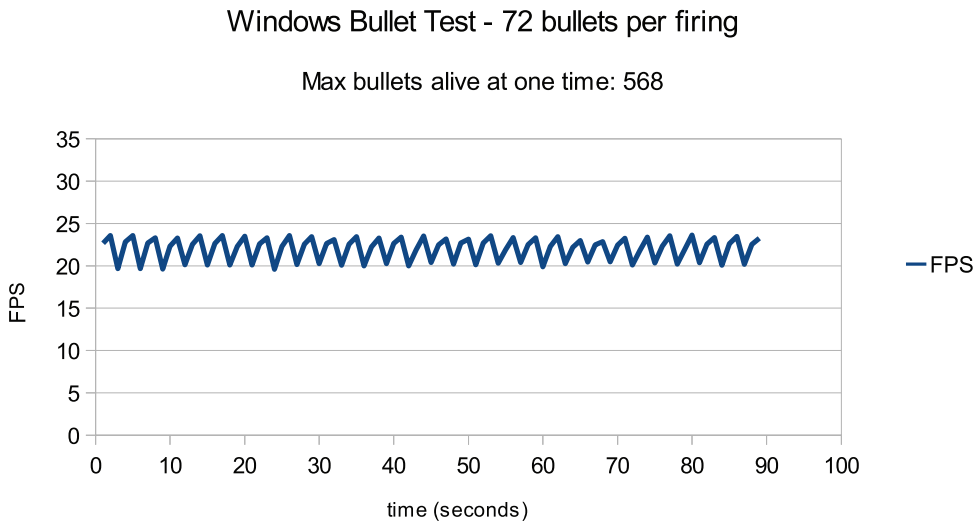

Figure 5.16: Windows Bullet Test - 72 per round.

5.7 Unit Performance

The next test is meant to stress IsoMob with the game entity that is the most complex; enemies. Enemy Characters utilize various types of AI, apply collision detection, contain animations, and broadcast numerous events. This test spawns a large number of enemies and lets them roam the IsoScene. Roaming is one of the more resource intensive AI methods, and represents a good fit when attempting to stress the game engine. No bullets are used in this scene, as it is important to try and find what is the larger bottle neck in frame rate performance. The level design is exactly the same as the previous test, the only change is instead of spawning many bullets we spawn many enemies. A screenshot showing what the chaos looks like is shown in Figure 5.17.

5.7.1 Android tablet

The Android tablet showed very good performance in the bullet tests, and looks to show equally impressive results with the unit test. The first set of test results is shown in Figure 5.18.

Figure 5.17: Unit test meant to stress the engine with `x` number of simultaneous enemies.

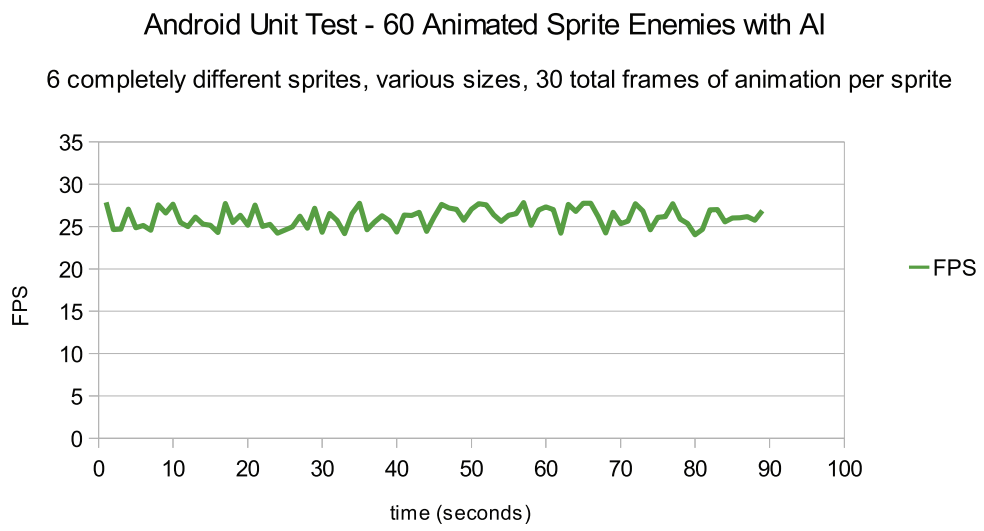

Figure 5.18: Unit test with 60 active enemies on the Android tablet.

The test results show that the Android tablet can handle around 60 active enemies at one time. In a real application this amount of enemies is unrealistic in that they block almost the entire screen. An IsoMob application would likely have five to ten simulatenous enemies at one time, anything more becomes very difficult to play against in an isometric game with a phone's touchscreen. The next test ramps up the number of enemies to 115, and the results are shown in Figure 5.19.

The number of enemies shown in this test is beyond excessive for the Android tablet. The tablet was at the borderline for acceptable performance when 60 enemies occupied the screen, so it is not suprising that 115 enemies brings it to it's knees. Again, this is not a real world situation and shows that even with an overly excessive scenario the tablet still performs decently.

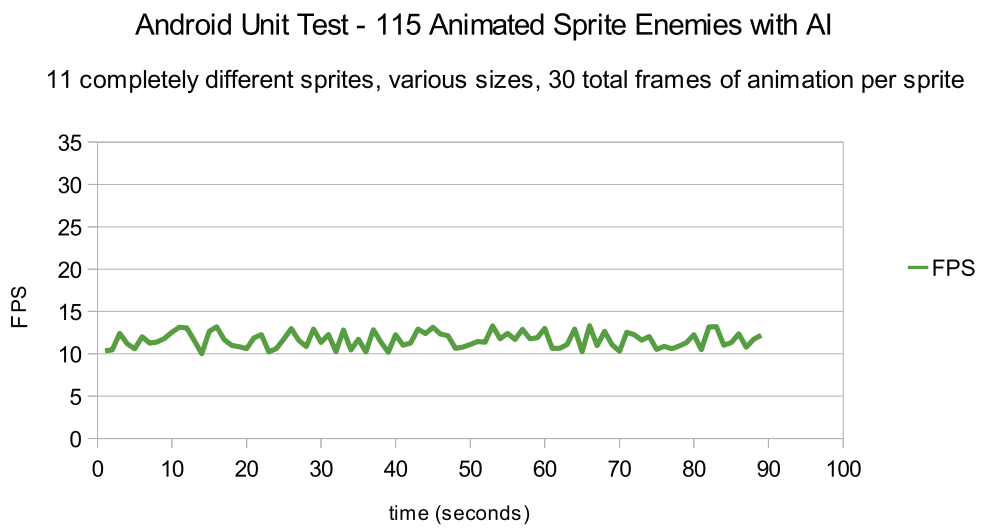

Figure 5.19: Unit test with 115 active enemies on the Android tablet.

.

5.7.2 Android phone

The Android phone didn't fare nearly as well as the tablet. The test results for the unit performance tests are similar to that of the bullet test results; very low. Because of the much lower performance two additional tests were run on the phone with less units to gauge where the phone reaches acceptable performance. The first set of results is shown in Figure 5.20.

The test results here show that the phone is capable of pushing 20 active enemies in a scene, but the performance is already beginning to fall away from the desired 30 frames per second rate. This trend continues with the next test. The results are shown in Figure 5.21.

The phone performance with 30 enemies is adequate, and at the point where the phone can not handle much more. Further tests show that the framerate suffers

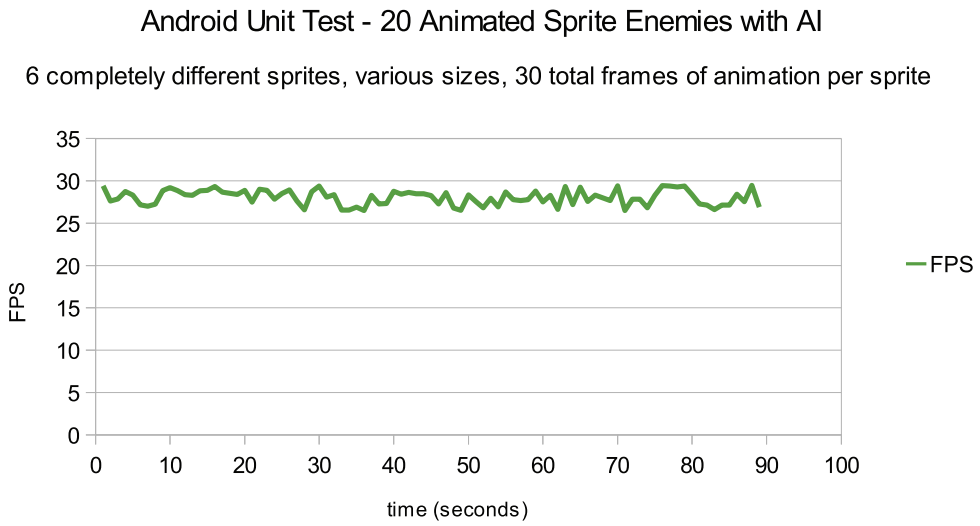

Figure 5.20: Unit test with 20 active enemies on the Android phone.

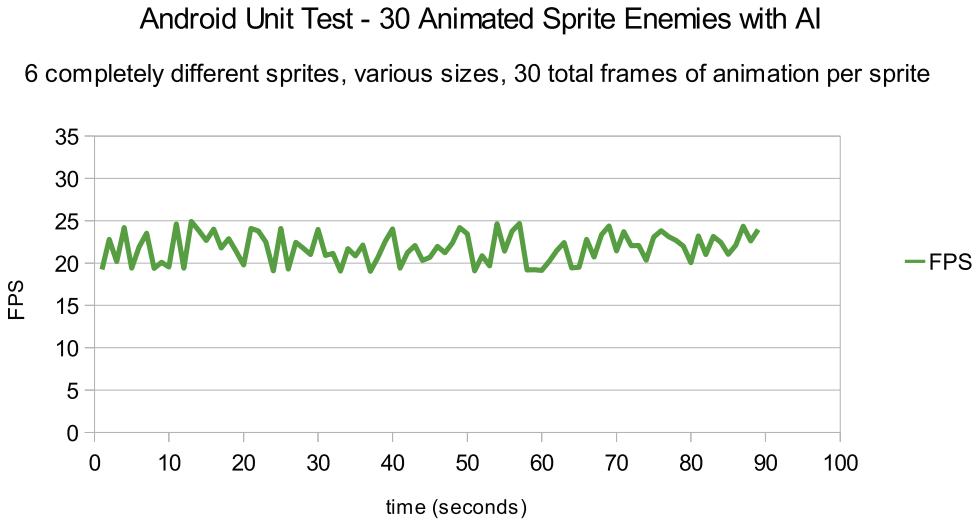

Figure 5.21: Unit test with 30 active enemies on the Android phone.

quite a bit when jumping to the next test. The results for the 60 enemy test are shown in Figure 5.22.

The phone simply cannot keep up with this many enemies on the screen at one time. Even though this example is far from a real world application, it is a nice target to shoot far when applying further optimization. Eventually, I would like to push the engine to support this test at 30 FPS on the phone. Out of curiosity the test consisting of spawning 115 enemies was run. The phone immediatley crashed with an out of memory error when attempting to start the application. As expected, the phone simply cannot handle a scene as large as this.

5.7.3 Adobe Flash

The Flash target has shown better performance than both Android targets up to this point, and the trend continues with the unit tests. The results of the first test are shown in Figure 5.23.

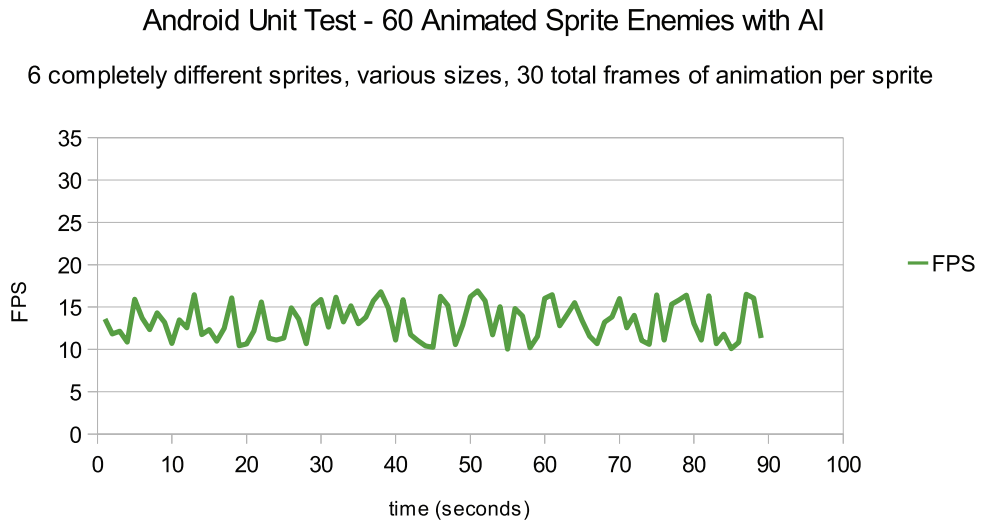

Figure 5.22: Unit test with 60 active enemies on the Android phone.

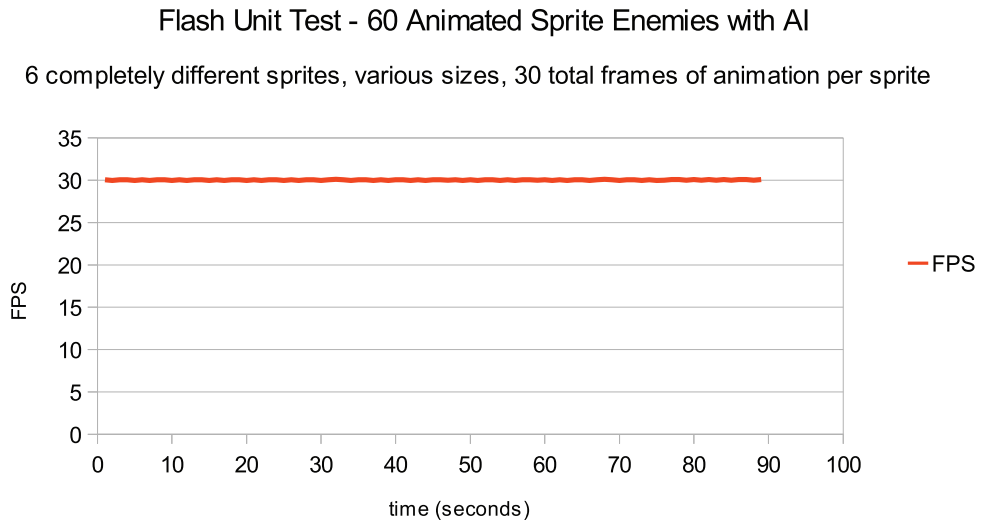

Figure 5.23: Unit test with 60 active enemies in Adobe Flash.

Flash easily handles the 60 unit test and does not falter from 30 frames per second for even a moment. The next set of test results, spwaning 115 enemies, is seen in 5.24.

The unit ceiling in Adobe Flash is not a real issue as the limit is very high. Performance is very good in Adobe Flash and an application containing a large amount of enemies is quite possible.

5.7.4 Windows

The Windows target proved it was the fastest of all targets, and the trend continues with the unit tests. The test results are shown in Figure 5.25 and 5.26.

These tests are no match for the Windows HaXe target. The frame rate is locked at 30 FPS and shows no signs of changing. The Windows target is by far the fastest of the group and needs little optimization when it comes to handling large amounts of IsoCharacters.

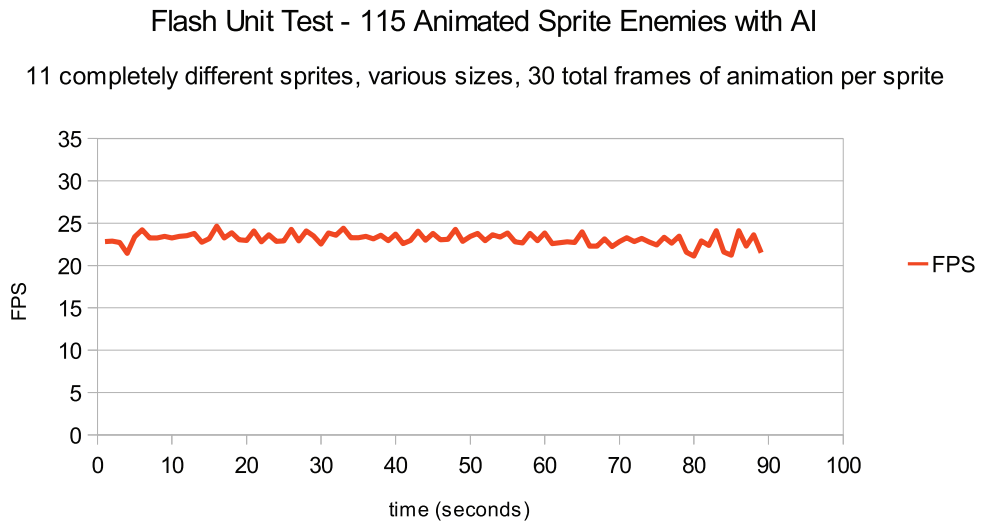

Figure 5.24: Unit test with 115 active enemies in Adobe Flash.

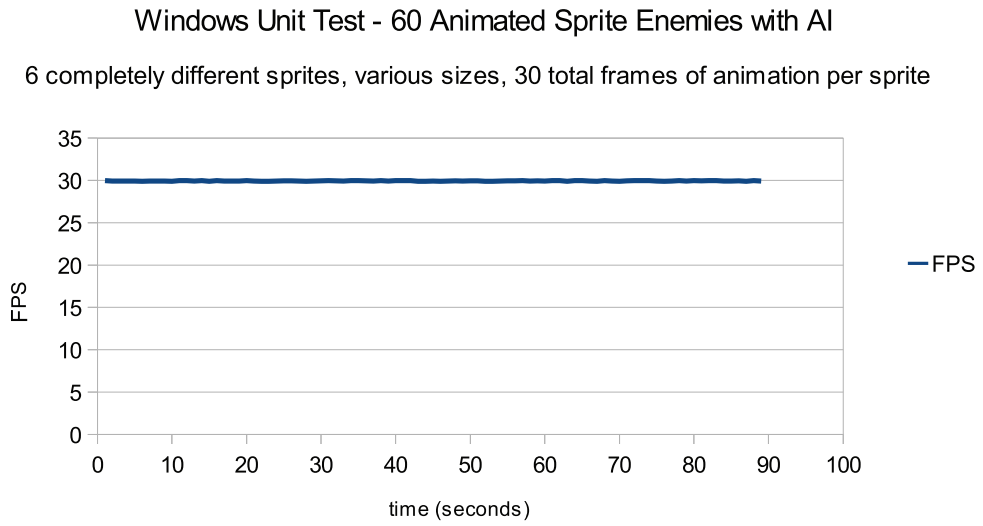

Figure 5.25: Unit test with 60 active enemies in Windows.

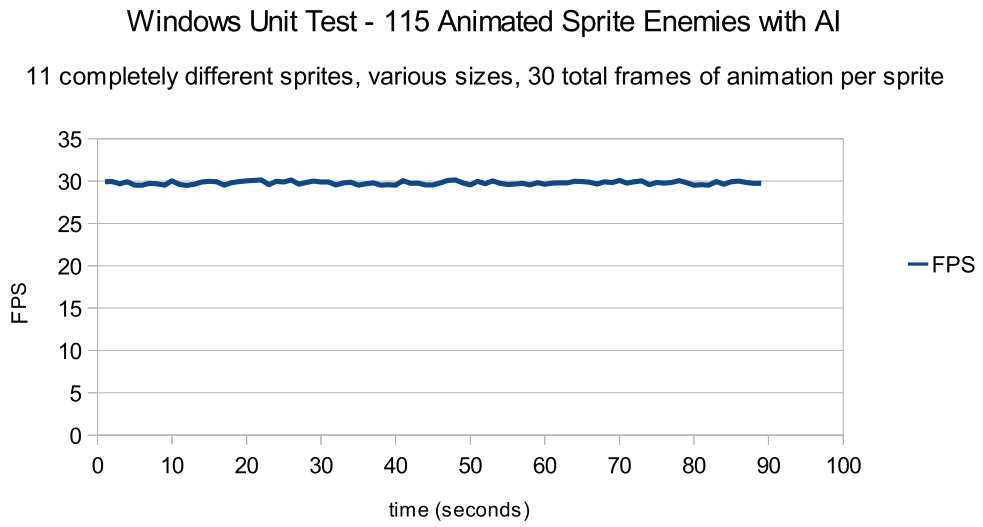

Figure 5.26: Unit test with 115 active enemies in Windows.

CHAPTER 6

CONCLUSION

IsoMob enables users to develop isometric games for a variety of platforms while still providing the necessary performance. It is currently the only game engine designed with the HaXe language that is optimized for both internet (Adobe Flash) and mobile platforms (Google Android). Utilizing the unique ability of HaXe to target multiple platforms with a single code based allows IsoMob to compile to actionscript 3.0, as well as C++. IsoMob leverages the NME library in order to reproduce the Flash 10 API in C++ making the cross-platform targeting a reality. Around 95% of the code is completely the same between various targets, the main differences being how graphics are rendered on screen and how assets are loaded into the game. The HaXe language is growing rapidly and hopefully the IsoMob contribution will help other users in the community and attracts new users to HaXe.

6.1 Contributions

IsoMob is an isometric game engine that runs on Windows, Adobe Flash, and on Google Android. With a little extra effort any game made using IsoMob will also run on iOS and webOS. The game engine itself is the first of its kind for HaXe, and enables developers to concentrate on creating their game while IsoMob handles working with the lower level software layers. Rendering performance is optimized per target, and provides an environment on mobile platforms where hundreds of sprites can occupy the screen at 30 FPS.

6.1.1 Open Source Game Engine

IsoMob is open source, and will be added to Google code very soon. The end goal is that this game engine can help other developers speed up the game development process when attempting to build for mobile platforms. There are alternative game engines out there but none that fill the exact same market as IsoMob.

6.2 Future Work

IsoMob works well for what it does, but it still needs quite a few features to make it truly robust.

6.2.1 Robustness

The game engine itself has a few small bugs present. One of the problems is how planes are rendered. When two planes with textures examined closely, small lines can be seen seperating them. This occurs do to pixel rounding errors in the rendering itself. The problem occurs because individual planes have the isometric transform applied to them, and once the planes are in isometric space they no longer perfectly line up along pixels, as they do before the transform takes place. The solution to this is to perform the isometric transform on all planes in a given coordinate space at the same time. Unfortuneatley, the rendering is setup in a manner that implementing this would require a rewrite of a large portion of code. The lines do not produce any ill-effects behind the scenes in the game engine, but they are unappealing to the end user.

6.2.2 User Interface

Currently creating maps inside of IsoMob is not the easiest task. A GUI level editor was in the works during the completion of this thesis, but it still not

in a finished state. The editor can create basic maps but it is currently filled with bugs and lacking in features. This is another feature that was left out due to time constraints. At first I thought it would be easy to extend the game engine in order to build maps visually, but this turned out to be much more work then planned. Particularly guarding against wrong end user behavior, in this case my own behavior. Handling user interaction such as random mouse clicks turned out to produce subtle and hard to fix bugs. Eventually I decided to stop working on the editor to continue working on IsoMob, but I would still like to see it completed.

6.2.3 Extensions to AI

There are many AI techniques that were simply too advanced for someone with no prior AI development experience. Techniques such as neural networks, fuzzy logic, and path planning would be great to have in the game engine but require alot of effort to implement. Path planning in particular would really help entities navigate tight corridors and large spaces. The current method of moving entities with steering forces is good for small open worlds but it tends to drop off in effectiveness as the worlds get larger and more closed in. Coordination between enemies and leaders could be done using neural networks in order to give enemies a collective intelligence and allow them to make smarter decisions. Fuzy logic would be very useful for defining the state machine that directs enemies to use the current steering behaviors or any other form of movement. One of the key areas where the AI is lacking is gathering information about the world and making a decision based on that information.

6.2.4 Extensions to Physics

One of the features that was left out due to time constraints was integrating a large physics engine such as Box2D or chipmunk. Both of these physics engines are open source, but neither of them at the time had been ported to HaXe, and I wasn't able to dedicate enough time to port over libraries with 100,000 lines of code into a new langauge. Having a "real" physics engine that is capable of calculating broadphase collisions, convex polygon collision bounds and AABB would really step up the IsoMob AI. Many corners were cut in order to provide good performance in regards to collision detection. The performance is very good but some of the approximations are less than ideal.

6.2.5 Performance

There are a few areas of the game engine that could use a boost in speed. One area, talked about briefly earlier, is with bullets. In the sample application bullets outnumbered characters almost 20 to 1, and because of this large ratio are prime candidates for optimization. The entire structure of the bullets could use a rewrite focusing on keeping things as simple as possible, so simple, that OOP practices would not be used. There are scenarios where it is still valuable to follow the game programming techniques from much less powerful systems, and I think that bullets is one area in need of these optimizations.

The rendering performance is very subpar on the particular Android phone that I have to test on. The HTC Incredible is a feature packed phone, but its severly lacking in the GPU area compared to other modern Android phones. The GPU weakness has proven very poor performance in most of the tests, but at this point I'm not quite sure how to improve things. Getting multiple pieces of hardware to test on, would hopefully gain additional insight into what may be the problem. The

GPU itself is quite abstracted by Android, and I would like to dig deeper to determine how things really work in order to increase rendering performance.

6.2.6 Evaluation

Overall, I'm happy with how IsoMob turned out. Prior to this project I had zero experience programming any type of game. I had written code in the past for a variety of things such as device drivers and network stacks but I quickly realized that programming a game was much different. The biggest obstacle I had at first was relearning much of the linear algebra and geometry that I didn't realize I had forgotten. The other issue was deciding how to create something that could target both the internet and mobile phones effectively. I didn't want to create an overly simple application because the performance requirements would be so low that any solution would be viable. IsoMob is a functional isometric game engine that provides exactly what it was intended to do, great performance on mobile platforms and the internet for fairly complex game using a single code base.

APPENDIX A

DEVELOPMENT ENVIRONMENT

Targeting various platforms with HaXe requires each of the platform dependant programs to be installed and setup. First time HaXe users may find this a bit overwhelming but recent additions such as the NME install tool have made things much easier. It should be said that other game engines require setup installing programs, and setting environment variables and such, so this requirement is not unique to choosing HaXe and NME.

A.1 HaXe Configuration

- The first step to using HaXe is to, of course, install it. HaXe can be installed from:
  - `http://haxe.org/download`.
- After HaXe is downloaded test it by going to the command prompt and typing the following:
  - `haxe`
- Next, install the necessary libraries to target Android. The first is Hxcpp, the C++ target for HaXe is still fairly new and is not yet included as an official part of HaXe. At the command prompt type the following:
  - `haxelib install hxcpp`
- The next library to install is NME. This is installed the same way as Hxcpp:
  - `haxelib install nme`

At this point we can build HaXe programs for Flash since it can build for the Flash target by default in the 2.07 release. In the next release the C++ target

will be bundled as well, so downloading Hxcpp will not be necessary. Now we can begin setting up for the Windows and Android targets.

A.1.1 Visual Studio Configuration

- The first step is to setting up the Windows C++ target is download and install the Visual Studio 2010 C++ package, so that we can compile C++ code.
  - `http://www.microsoft.com/express/Downloads/#2010-Visual-CPP`
- After Visual Studio is installed we will use the compiler `cl.exe` from the command line. A convenient shortcut to access the necessary Visual Studio information is to type the following at the command line (The path may be different if you are not using Windows 7):
  - `c:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\Tools\vsvars32.bat`
- Once that command is run, then the Visual Studio compiler can be invoked from the command line, and building for the HaXe C++ target on Windows will work.

One thing to note is that the `vsvars32.bat` command needs to be run in any command prompt window that you plan to build HaXe projects in, so its common to include calling this batch file in another batch file. In a future version of HaXe XML configuration will be present so that no script file will have to be written, as HaXe will cover the necessary dependancies.

A.2 Android Configuration

In order to use HaXe with Android we first need to setup the system to allow building Android applications.

- The first step is to download and install the Java Development Kit (don't use the x64 version):
  - `http://www.oracle.com/technetwork/java/javase/downloads/`
- Next, install the Android SDK, all code in IsoMob was tested against API version 2.2.
  - `http://developer.android.com/sdk/index.html`
- After the SDK is installed, we install the NDK:
  - `http://developer.android.com/sdk/ndk/index.html`
- After the development kits are installed we install Cygwin utilities as these are required by the Google build tools.
  - `http://www.cygwin.com/install.html`
  - Add Cygwin to the `PATH`: `c:\<install-dir>\cygwin\bin`
- Next, we install ANT so that we do not need to install Eclipse. If you prefer using Eclipse then ANT does not need to be installed.
  - `http://ant.apache.org/bindownload.cgi`
- Before building for Android we have a few more environment variables to set that are necessary for HaXe. Type the following in your opened command prompt window, where `<install-dir>` is where you installed each program:
  - `set ANT_HOME=<install-dir>/ant`
  - `set ANDROID_SDK=<install-dir>android-sdk`

- `set ANDROID_NDK_ROOT=<install-dir>android-ndk`
- `set JAVA_HOME=<install-dir>\jdk1.6.0_24`

All install and setup requirements are done at this point. As mentioned earlier, these options can be setup directly in HaXe as of the next release with the file .hxcpp_config.xml. Using the command prompt we can now build some of the sample projects included with NME. The commands to build with NME for some of the targets are summarized in Table A.1. The file `project.nmml` refers to the name of your project that you will be creating. This is similar to a project file such as *.sln for visual studio.

An example of the the project.nmml file is detailed with comments below in listing A.1.

```
<?xml version="1.0" encoding="utf-8"?>
<project>

    <app
      file="thesisProject"
      title="IsoMobDemo"
      package="diperna.tony.IsoMob"
      version="1.0"
      company="Oakland University"
```

Table A.1: Summary of building HaXe and NME for various platforms.

Command	Target
`haxelib run nme test project.nmml flash`	Flash
`haxelib run nme update project.nmml ios`	iPhone
`haxelib run nme test project.nmml webos`	webOS
`haxelib run nme test project.nmml android`	Android
`haxelib run nme test project.nmml cpp`	PC 32-bit
`haxelib run nme test project.nmml cpp -64`	PC 64-bit

```
    main="Main"
  />

  <window
    width="800"
    height="480"
    orientation="landscape"
    fps="30"
    background="0xffffff"
    resizeable="true"
    hardware="true"
  />

  <classpath name="." />
  <haxelib name="nme" />

  <assets>
    <asset name="../assets/characters_badguys.png" />
    <asset name="../assets/characters_badguys.xml" />
  </assets>

  <ndll name="std" />
  <ndll name="regexp" />
  <ndll name="zlib" />
  <ndll name="nme" haxelib="nme" nekoapi="1"/>

  <certificates>
    <certificate file="signature.cer" name="mysign" password="abc" />
  </certificates>

</project>
```

Listing A.1: Example *.nmml file

APPENDIX B

HOW HAXE WORKS

When a developer hears HaXe they may have no idea just what it entails. HaXe is a high-level multiplatform programming language described as a "universal language"[10]. HaXe currently has a much smaller community of users but that community is growing quickly. As of a few years ago HaXe did not have a c++ target, and many of the other targets, and was really used as good alternative to the adobe actionscript compiler. It evolved outside of flash and added addditional targets such as Javascript, PHP, and C++. Other soon to come targets include Java as well as C#. The number of targets HaXe supports and overall amount of functionality is growing very rapidly. The C++ target is mainly used for this project because the C++ is used to for the desktop PC target, as well as the Google Android and Apple iPhone targets.

B.1 Adobe Flash

Adobe Flash is targeted with the HaXe flash target although it doesn't have to be. The flash target is unique that NME was based around the flash API, so much of the NME functionality is already built into flash. One of the advantages of using NME with flash is loading external assets. Adobe flash SWF files cannot access the local file system so they load everything remotely using a `URLRequest`, which is similar to an `HTTP` request. If NME is used, then assets can be directly embedded into the resulting output SWF file. This can be done without NME as well, but outside of the Flash IDE it is a tedious process. The structure for how HaXe and NME target flash is detailed in Figure B.1. Adobe Flash doesn't use OpenGL or DirectX for rendering, instead it uses its own internal rendering process. Currently that rendering is limited to mostly software based rendering with some GPU optimization, but full GPU rendered flash content is not yet available. With the ad-

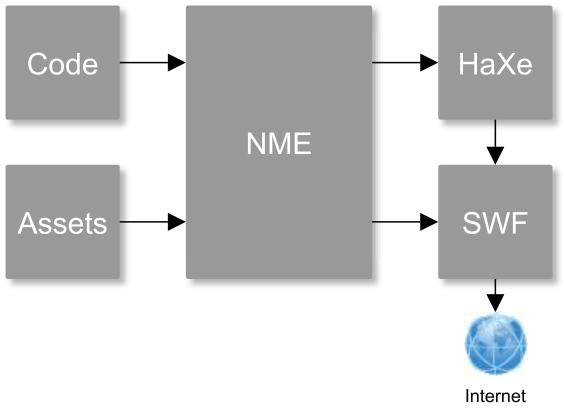

Figure B.1: Targeting Adobe Flash with HaXe and NME.

dition of the Molehill API in Flash player 11, Flash will have the added capabilities of rendering straight from the GPU with its own shader language[18].

B.2 PC (Windows)

Windows is targeted using the HaXe C++ target, and the Visual Studio C++ (VSC++) compiler is used to build all generated C++ code. Its worth nothing that other C++ compilers will work just fine, and that HaXe C++ builds on Linux 32-bit, Linux 64-bit and OSX varieties, I just don't have a PC with any of those operating systems installed. A diagram illustrating how HaXe works with the Windows C++ environment is shown in Figure B.2. In the PC environment NME renders directly to OpenGL and provides wrapper functions very similar to SDL. These wrapper functions make it trivial to get things on screen quickly, which is one of the biggest advantages to using Adobe Flash. NME essentially took the ease of creating graphics in flash and ported it to C++.

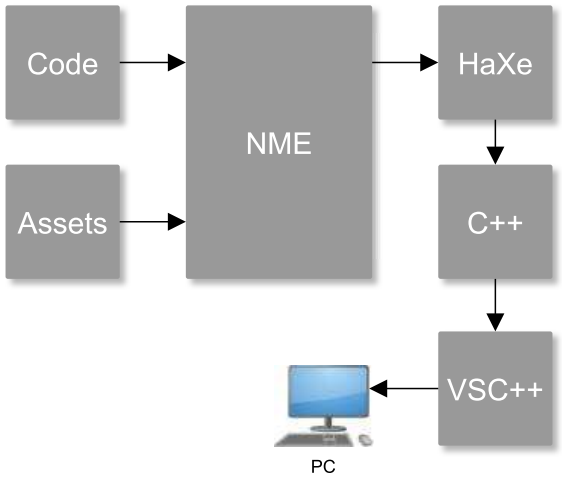

Figure B.2: Targeting a PC with HaXe and NME.

B.3 Android

Android is targeted using HaXe and Google's native development kit (NDK) as shown in Figure B.3. HaXe's interaction with Android is similar with the PC environment in that it again wraps OpenGL (OpenGL ES this time!) in order to perform graphics rendering. The traditional way to develop for Android is to use the SDK and develop applications using Java. Not to long after the SDK came out, the lower level NDK was released. At first the NDK lagged far behind the SDK in features and usability, but Google found that the NDK was much more popular with users then they had originally imagined. The NDK now has just about all of the features of the SDK, meaning that Android applications developed in C++ have access to the same APIs as the SDK and can provide better performance than Java applications. Generally, non-trivial games require all the performance help they can get so using the NDK is a good idea for critical portions of code such as rendering. Using HaXe almost all of the code is straight C++, so as long as the game en-

gine renders efficiently performance will be very good. When targeting the Android NDK with C++ there is still some Java code present. The main view of an Android application can only be created in Java. Once the main view is created then Java native interface (JNI) calls will provide all the interface requirements needed to get all of the C++ content on screen.

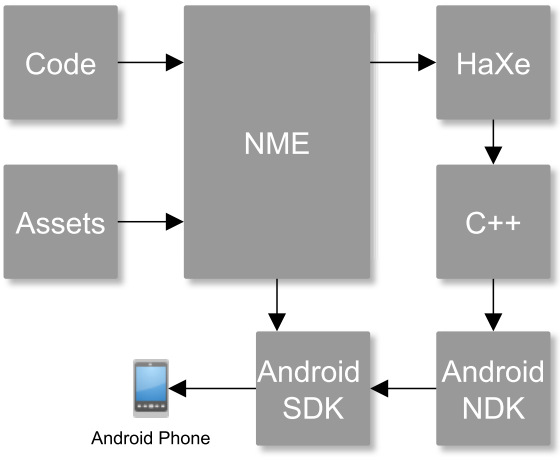

Figure B.3: Targeting Android with HaXe and NME.

APPENDIX C

ANDROID NDK TARGETING

Using the Android SDK to create an application is much simpler then trying to use the NDK. Applications will still behave the same way but there is quite a bit more initial setup with the NDK. Most of the setup involves setting up the Java to C++ interface. The Java Native Interface (JNI) is a programming framework that allows Java code to interact with native applications written in other languages. JNI has been in use for a long time and is not a new feature added to Java just for Android. Going into the details of how JNI works is outside the scope of this project, instead this appendix will give a brief overview of an Android application as well as how to use JNI in order to get a C++ application running on Android.

C.1 Android Application Structure

Android is a middleware software stack built on top of Linux. Each application runs in its own linux process, and each process has its own virtual machine (VM). This means that your application runs in isolation from all other running applications. Android applications are unlike traditional programs in that there is no main() function. Applications are made up of sequences of `Activity` objects that communicate with each other using `Intent` messages. An `Activity` is a user interface screen, something a user can interact with, but its not an entire application. An `Intent` is a bundle of information that drives structured actions. It is a message passes between `Activity` objects and can be local or global to the current application. Some `Intent` examples are: "send an email", "place a call" and "draw a map". Applications are also built with only four types of components, two of them being `Activity` and `Service` objects, the other two being `Content Providers` and `Broadcast Receivers`.

The components of an Android application are detailed in Figure C.1. The component that the user can see in an Android application is the `Activity` class.

The main `Activity` class acts as the entry point to an Android application. The application `Activity` objects are meant to be designed as highly reusable. The accepted method to design an Android application is to utilize `Activities` from other applications in order to reach an end goal. For example, if one would like to get user input they may display the pop up keyboard `Activity`. There is no reason to create a seperate pop up keyboard because the user is already familiar with this user input method, as it is used in many other applications. This design methodology makes the overall user interface on an Android phone consistent and familiar. There is no reason to force a user to learn yet another new function unless it is neccessary, as it may frustrate them due to differences in design. In many cases `Activities` need to communicate with each other, and similar to network protocols a defined bundle of information is passed between `Activity` objects. This bundle of information is an `Intent` and it consists of predefined commands that allow `Activities` to communicate with a consistent API. Its worth noting that `Intents` can be passed locally (from within your application) or globally, by an `Activty` in another running application. Examples of `Intents` are: send an email, place a call and view website. Some examples of `Activities` are: a contact list, the pop up keyboard, and a list of new emails.

The `Service` class is meant to run background processes that may take a long time to complete. Under no circumstance should an application lock up the user interface while working on another task. In order to avoid an interface lockup the `Service` class can process things at a lower priority while the user is free to navigate through the rest of the application. A `Service` is meant for operations that may take a long time to run, examples are: connecting to a database and loading file over the internet. The `Content Provider` class is used to query sources of data. Examples of `Content Providers` are: a mysql database, a sqlite database,

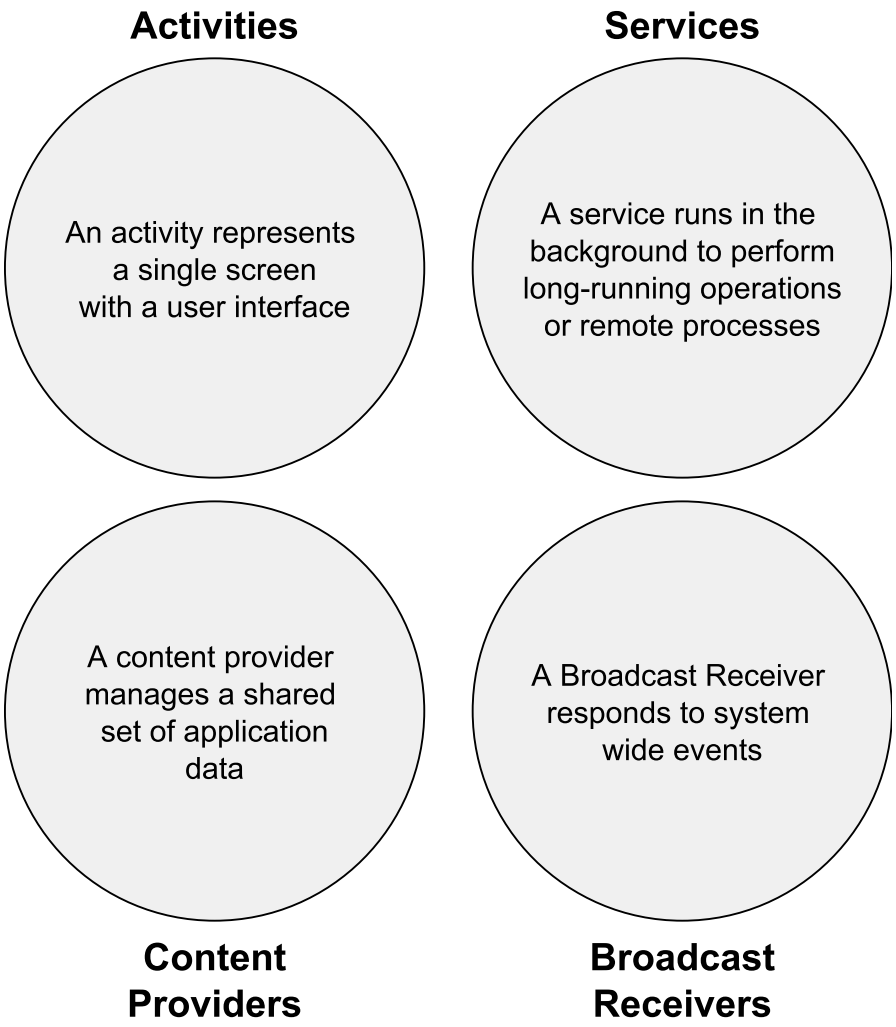

Figure C.1: Android application structure overview.

a XML file and other types of data files. A `Content Provider` shares a common interface regardless of the source of the data, and because of this abstraction, allows all types of data to be accessed in the same manner. The last component, the `Broadcast Receiver`, is used to indicate system wide events. This class behaves similarly to a hardware interrupt in that an interrupt handler will be called immediatley after the interrupt conditions are met. In this case, an event handler is called as soon as a system wide event occurs. Example events are: battery low, file downloaded and incoming call. These events much be handled properly by your application in order to prevent entering an unknown state.

Using the NDK and building an Android application in C++ at best allows you to produce a library and not an executable. The only way to run an application on Android is to use Java. This is where the JNI like bootloader is needed. The JNI code is used to create an initial `Activity` that subsequently loads the C++ produced library (.so) file. The IsoMob game engine uses Java in order to produce an OpenGL ES window which is then used for the entire application. This Java initialization is provided by the HaXe C++ target, and is the same generic routine that is provided as boiler plate code by Google in the Hello-JNI-Example [19].

APPENDIX D

ASSET PARSING SCRIPT

This script consists of two classes `GraphicsParser` and `LevelInfo`. The `GraphicsParser` is responsible for creating the valid types for the IsoMob game engine: Characters, Objects and Textures. The `LevelInfo` class consists of the keywords in the actual .png file names of the asset files. This script performs a regular expression search for each keyword and then classifies the asset as the correct type of IsoMob game object. Both classes can be easily extended, and Figure D.1 shows an example of asset file structure used by the script.

The script works by parsing the full path of the raw .png assets and creates the necessary .xml files that instruct the engine exactly how to use the assets when creating in game objects.

The source code for the asset parsing script that creates the necessary meta data is detailed with comments in listing D.1.

```
import haxe.io.Eof;
import neko.io.File;
import neko.Lib;

/**
* Tony DiPerna
*/
class GraphicsParser
{
//Output definition files
inline private static var DEFINITIONS_LEVEL_0:String = "level_0.xml";
inline private static var DEFINITIONS_LEVEL_1:String = "level_1.xml";
inline private static var DEFINITIONS_LEVEL_2:String = "level_2.xml";
inline private static var DEFINITIONS_LEVEL_3:String = "level_3.xml";
inline private static var DEFINITIONS_LEVEL_4:String = "level_4.xml";
inline private static var DEFINITIONS_LEVEL_5:String = "level_5.xml";
inline private static var DEFINITIONS_LEVEL_6:String = "level_6.xml";
inline private static var DEFINITIONS_LEVEL_7:String = "level_7.xml";
inline private static var DEFINITIONS_LEVEL_8:String = "level_8.xml";
inline private static var DEFINITIONS_LEVEL_9:String = "level_9.xml";
```

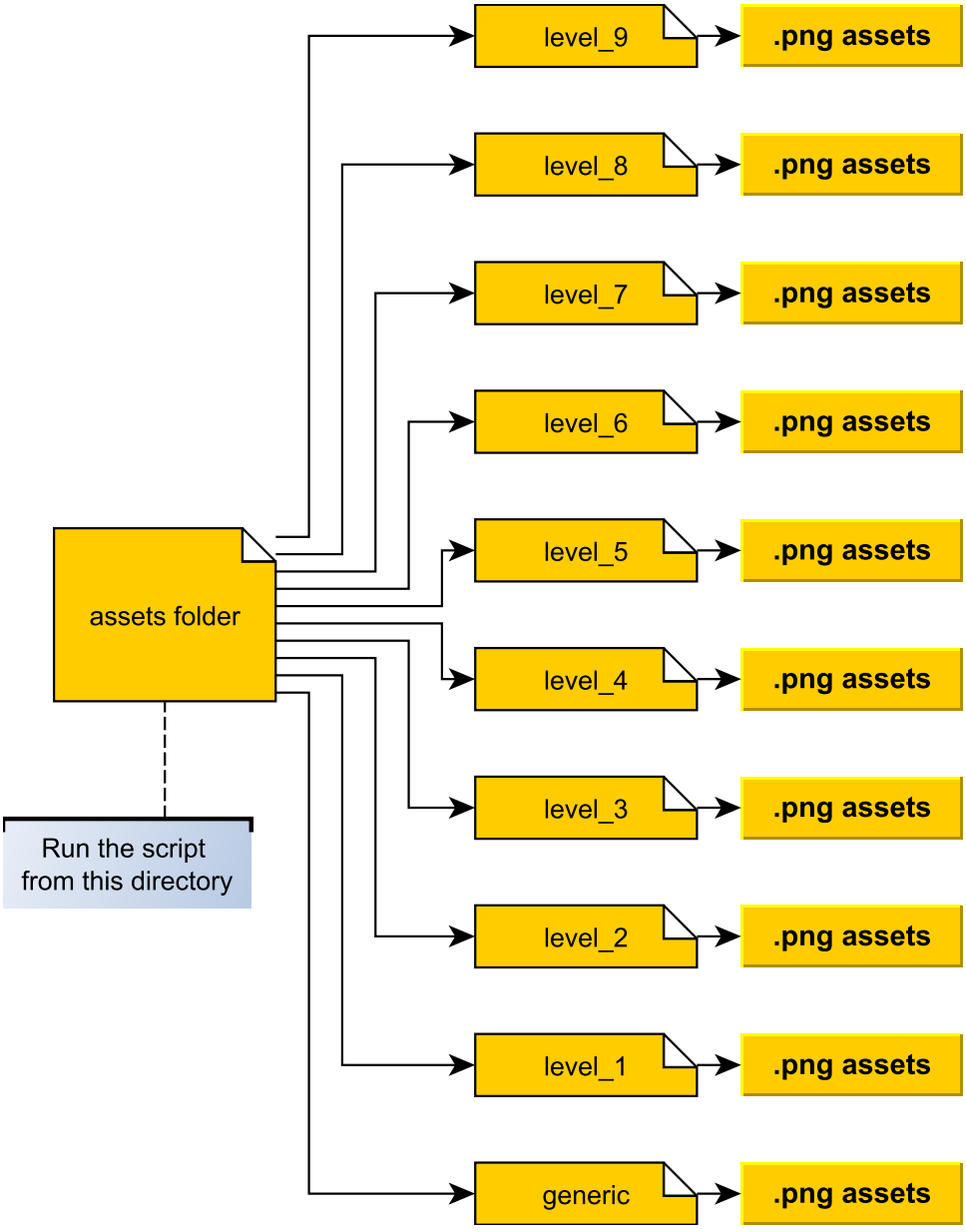

Figure D.1: Example folder structure in regards to the asset parsing script.

```
//Key words to parse input file for
inline private static var SEARCH_LEVEL_0:String = "generic";
inline private static var SEARCH_LEVEL_1:String = "Level 1";
inline private static var SEARCH_LEVEL_2:String = "Level 2";
inline private static var SEARCH_LEVEL_3:String = "Level 3";
inline private static var SEARCH_LEVEL_4:String = "Level 4";
inline private static var SEARCH_LEVEL_5:String = "Level 5";
inline private static var SEARCH_LEVEL_6:String = "Level 6";
inline private static var SEARCH_LEVEL_7:String = "Level 7";
inline private static var SEARCH_LEVEL_8:String = "Level 8";
inline private static var SEARCH_LEVEL_9:String = "Level 9";

private static var searchWords:Array<String> =
[
SEARCH_LEVEL_0,
SEARCH_LEVEL_1,
SEARCH_LEVEL_2,
SEARCH_LEVEL_3,
SEARCH_LEVEL_4,
SEARCH_LEVEL_5,
SEARCH_LEVEL_6,
SEARCH_LEVEL_7,
SEARCH_LEVEL_8,
SEARCH_LEVEL_9
];

private static var levels:Array<Array<String>> =
[
new Array<String>(),    //0
new Array<String>(),    //1
new Array<String>(),    //2
new Array<String>(),    //3
new Array<String>(),    //4
new Array<String>(),    //5
new Array<String>(),    //6
new Array<String>(),    //7
new Array<String>(),    //8
new Array<String>()             //9
```

```
];

private static var levelInfos:Array<LevelInfo> =
[
new LevelInfo(),        //0
new LevelInfo(),        //1
new LevelInfo(),        //2
new LevelInfo(),        //3
new LevelInfo(),        //4
new LevelInfo(),        //5
new LevelInfo(),        //6
new LevelInfo(),        //7
new LevelInfo(),        //8
new LevelInfo(),        //9
];

private static var definitionNames:Array<String> =
[
"level_0.xml",
"level_1.xml",
"level_2.xml",
"level_3.xml",
"level_4.xml",
"level_5.xml",
"level_6.xml",
"level_7.xml",
"level_8.xml",
"level_9.xml",
];

static function main()
{
  //parse the input file (list.txt) and split into information by level
  parseInput("list.txt");

  //Search the level arrays and split into information by type
  for (i in 0...levels.length)
  {
    //parse the level data and break it into groups
```

```
    levelInfos[i].parseIntoTypes(levels[i]);

    //Generate definition files
    createDefinitions(definitionNames[i],levelInfos[i],i);

    //Generate the level templates
    //No need for level 0 since its the common files
    if (i > 0)
    {
          createLevelTemplates(i);
    }
  }
}

private static function createLevelTemplates(levelIndex:Int):Void
{
  var fName = "Level_" + levelIndex + "_template.xml";
  var fout = neko.io.File.write(fName, false);

  fout.writeString('<scene gridsize="64" levelsize="64">\n');
  fout.writeString('    <head>\n');
  fout.writeString('            <name>Level'+levelIndex+' Template </name>\n');
  fout.writeString('            <description>template</description>\n');
  fout.writeString('            <definitions src="definitions/'+definitionNames[0]+'"
       />\n');
  fout.writeString('            <definitions src="definitions/'+definitionNames[
      levelIndex]+'" />\n');
  fout.writeString('    </head>\n');
  fout.writeString('    <body>\n');
  fout.writeString('    </body>\n');
  fout.writeString('</scene>\n');
}
private static function createDefinitions(outName:String,levelInfo:LevelInfo,
    levelIndex:Int):Void
{
  //WRITE
  //open file for writing
  var fout = neko.io.File.write(outName, false);
```

```
fout.writeString("<definitions>\n");

fout.writeString("<!--\n|\n");
fout.writeString("|  M  E D I A\n");
fout.writeString("||||||||||||||||||||||||||||||||||||||||||||||||||||--->\n");
fout.writeString('<media src="assets/level'+levelIndex+'" />');


//<materialDefinition name="matYellowTileFloor" type="tile">
//    <diffuse>matYellowTileFloor</diffuse>
//</materialDefinition>
//output materials
fout.writeString("\n");
fout.writeString("<!--|\n");
fout.writeString("|  M  A  T  E  R  I  A  L  S\n");
fout.writeString("||||||||||||||||||||||||||||||||||||||||||||||||||||--->\n");
for (mat in levelInfo.materials)
{
fout.writeString('<materialDefinition name="'+mat+'" type="tile"><diffuse>'+mat+'</
    diffuse></materialDefinition>');
fout.writeString("\n");
}

//<objectDefinition name="objBookshelfSE" solid="true" animated="false">
//   <displayModel><sprite angle="0"   src="objBookshelfSE"/></displayModel>
//   <collisionModel><cilinder radius="20" height="20"/></collisionModel>
//</objectDefinition>
//output objects
fout.writeString("\n");
fout.writeString("<!--\n|\n");
fout.writeString("|  O  B  J  E  C  T  S\n");
fout.writeString("||||||||||||||||||||||||||||||||||||||||||||||||||||--->\n");
for (obj in levelInfo.objects)
{
  fout.writeString('<objectDefinition name="' + obj + '" solid="true" animated="
      false"><displayModel><sprite angle="0" src="' + obj + '" /></displayModel><
      collisionModel><cilinder radius="20" height="20"/></collisionModel></
      objectDefinition>');
  fout.writeString("\n");
```

```
}

//<objectDefinition name="bastetcase" solid="true" animated="true">
//   <displayModel>
//      <sprite angle="45"   src="bastetcaseEast"/>
//      <sprite angle="225"   src="bastetcaseWest"/>
//   </displayModel>
//   <collisionModel><cilinder radius="20" height="20"/></collisionModel>
//</objectDefinition>
//output characters (SIDE)
fout.writeString("\n");
fout.writeString("<!--\n|\n");
fout.writeString("|  C H A R A C T E R S   *SIDE*\n");
fout.writeString("||||||||||||||||||||||||||||||||||||||||||||||||||-->\n");
for (char in levelInfo.charactersSide)
{
  fout.writeString('<objectDefinition name="' + char + '" solid="true" animated="
      true"><displayModel><sprite angle="45" src="' + char + 'East" /><sprite angle
      ="225" src="' + char + 'West" /></displayModel><collisionModel><cilinder
      radius="20" height="20"/></collisionModel></objectDefinition>');
  fout.writeString("\n");
}

//<objectDefinition name="bastetcase" solid="true" animated="true">
//   <displayModel><sprite angle="45"   src="bastetcaseEast"/>
//      <sprite angle="225"   src="bastetcaseWest"/>
//   </displayModel>
//   <collisionModel>
//      <cilinder radius="20" height="20"/>
//   </collisionModel>
//</objectDefinition>
//output characters (NORTH / SOUTH)
fout.writeString("\n");
fout.writeString("<!--\n|\n");
fout.writeString("|  C H A R A C T E R S   *NORTH/SOUTH*\n");
fout.writeString("||||||||||||||||||||||||||||||||||||||||||||||||||-->\n");
for (char in levelInfo.charactersUpDown)
{
```

```
    fout.writeString('<objectDefinition name="' + char + '" solid="true" animated="
        true"><displayModel><sprite angle="315" src="' + char + 'North" /><sprite
        angle="135" src="' + char + 'South" /></displayModel><collisionModel><
        cilinder radius="20" height="20"/></collisionModel></objectDefinition>');
    fout.writeString("\n");
  }

  fout.writeString("</definitions>");

  fout.close();
}

private static function parseInput(fName:String):Void
{
  // open and read file line by line
  var fin = neko.io.File.read(fName, false);
  var svnCheck : EReg = new EReg("svn","");

  try
  {
    neko.Lib.println("file content:");
    var lineNum = 0;
    while( true )
    {
      var line = fin.readLine();

      if (!svnCheck.match(line))
      {
        //loop through all search words
        for (word in searchWords)
        {
          var r : EReg = new EReg(word,"");

                  //Search each line of the file for the SEARCH WORDS
                  if (r.match(line))
                  {
                    //A match was found!
                    //get the index of the SEARCH WORD
                    var index = Lambda.indexOf(searchWords, word);
```

```
                    //Extract the name from the path
                    //Subtract 4 from len because everything ends with ".png"
                    var startPos = line.lastIndexOf("\\")+1;
                    var strName = line.substr(startPos, line.length - startPos - 4);

                    //Strip all numbers out of the name
                    r = ~/[0-9]+/;
                    strName = r.replace(strName, "");

                    //Don't allow duplicate data.  For animations, multiple lines
                        will
                    //be in the .txt file, we only care about one of these
                    //And since we strip out the numbers above we will have
                    //the same data in multiple places if we allow it.
                    //i.e.
                    //charA0000.png
                    //charA0001.png
                    //...
                    //charA0030.png
                    if (Lambda.indexOf(levels[index], strName) == -1)
                    {
                      //push the line into the appropriate array
                      //trace("strName: " + strName);
                      levels[index].push(strName);
                    }
                  }
                }
          }
        }
  }
catch( ex:haxe.io.Eof )
{}
fin.close();
}
}

class LevelInfo
{
```

```
  inline private static var TYPE_MATERIAL:String   = "mat";
  inline private static var TYPE_OBJECT:String     = "obj";
  inline private static var TYPE_CHAR_STAND:String = "Stand";
  inline private static var TYPE_CHAR_SIDE:String  = "Side";
  inline private static var TYPE_CHAR_NORTH:String = "North";
  inline private static var TYPE_CHAR_SOUTH:String = "South";

  public var materials:Array<String>;
  public var objects:Array<String>;
  public var charactersSide:Array<String>;
  public var charactersUpDown:Array<String>;

public function new()
{
  materials         = new Array<String>();
  objects               = new Array<String>();
  charactersSide        = new Array<String>();
  charactersUpDown      = new Array<String>();
}

public function parseIntoTypes(data:Array<String>):Void
{
  var material:EReg  = new EReg(TYPE_MATERIAL, "");
  var object:EReg    = new EReg(TYPE_OBJECT, "");
  var charSide:EReg  = new EReg(TYPE_CHAR_SIDE, "");
  var charStand:EReg = new EReg(TYPE_CHAR_STAND, "");
  var charNorth:EReg = new EReg(TYPE_CHAR_NORTH, "");
  var charSouth:EReg = new EReg(TYPE_CHAR_SOUTH, "");

  for (line in data)
  {
    //Determine the type
    if (material.match(line))
    {
      //This is a material
      if (Lambda.indexOf(materials, line) == -1)
      {
        materials.push(line);
      }
```

```
    }
    else if (object.match(line))
    {
      //This is a object
      if (Lambda.indexOf(objects, line) == -1)
      {
        objects.push(line);
      }
    }
    else if (charSide.match(line))
    {
      //This is a character
      //filter out "Side"
      var filteredLine = charSide.replace(line, "");

      if (Lambda.indexOf(charactersSide, filteredLine) == -1)
      {
        charactersSide.push(filteredLine);
      }
    }
    else if (charNorth.match(line) || charSouth.match(line))
    {
      //This is a character
      //filter out "North" "South"
      var filteredLine = charNorth.replace(line, "");
      filteredLine = charSouth.replace(filteredLine, "");

      if (Lambda.indexOf(charactersUpDown, filteredLine) == -1)
      {
        charactersUpDown.push(filteredLine);
      }
    }
    else
    {
          //...do nothing
    }
  }
}
```

```
}
```

Listing D.1: Asset parsing script file

BIBLIOGRAPHY

[1] R. Entner. (2010, Mar.) Smartphones to overtake feature phones in u.s. by 2011. [Online]. Available: http://blog.nielsen.com/nielsenwire/consumer/smartphones-to-overtake-feature-phones-in-u-s-by-2011/

[2] R. Kim. (2011, Jun.) How apple's iphone changed everything. [Online]. Available: http://gigaom.com/2011/06/29/the-iphone-effect-how-apples-phone-changed-everything/

[3] AndroLib. (2011, Jul.) Number of new applications in android market by month. [Online]. Available: http://www.androlib.com/appstats.aspx

[4] AppBrain. (2011, Mar.) Android market statistics overload. [Online]. Available: http://blog.appbrain.com/2011/03/new-android-market-statistics-overload.html

[5] S. Sivak. (2011, Jul.) What aspects of social games do you see as key to their additctive nature? [Online]. Available: http://www.quora.com/Social-Games/What-aspects-of-social-games-do-you-see-as-key-to-their-addictive-nature.

[6] M. Buckland, *Programming Game AI by Example*. Wordware, 2005.

[7] (2011, Aug.) cocos2d. [Online]. Available: http://cocos2d.org/

[8] B. Zhang. (2011, Apr.) Cocos2d-x performance results. [Online]. Available: http://www.cocos2d-x.org/projects/cocos2d-x/wiki/Performance

[9] Adobe. (2011, Jun.) Flash player version penetration. [Online]. Available: http://www.cocos2d-x.org/projects/cocos2d-x/wiki/Performance

[10] Twin-Motion. (2011, Sep.) Haxe. [Online]. Available: http://haxe.org/

[11] N. Walker. (2004, Jun.) On filmation. [Online]. Available: http://retrospec.sgn.net/users/nwalker/filmation/

[12] C. Reynolds, "Steering behaviors for autonomous characters," Sony Computer Entertainment of America, 1999, a strong form of the prime number theorem, 19th century. [Online]. Available: http://www.red3d.com/cwr/steer/gdc99/

[13] M. Beckler. (2009, Sep.) Inkscape: Isometric projection. [Online]. Available: http://www.mbeckler.org/inkscape/isometric_projection/

[14] J. Krikke, "Axonometry: A matter of perspective," *IEEE Computer Graphics and Applications*, 2000.

[15] R. Jelliffe, *The XML and SGML Cookbook : Recipes for Strucutred Information.* Prentice Hall, 1998.

[16] C. Reynolds. (2007, Jun.) Swarm intelligence. [Online]. Available: http://opensteer.sourceforge.net/doc.html

[17] C. Wild. Knight lore data format. [Online]. Available: http://www.icemark.com/dataformats/knightlore/index.html

[18] T. Imbert. (2011, Jan.) Introducing molehill: 3d apis for adobe flash player and adobe air. [Online]. Available: http://www.adobe.com/newsletters/inspire/january2011/articles/article1/index.html

[19] Google. (2010, Oct.) Hello jni example. [Online]. Available: http://developer.android.com/sdk/ndk/overview.html#samples

[20] J. Fitzpatrick, "An interview with steve furber," *Communications of the ACM*, 2011. [Online]. Available: http://cacm.acm.org/magazines/2011/5/107684-an-interview-with-steve-furber/fulltext

[21] K. Peters, *AdvancED Actionscript 3.0 Animation.* friendsofED, 2008.

[22] ——, *AdvancED Game Design with Flash.* friendsofED, 2010.

[23] F. Dunn and I. Parberry, *3D Math Primer for Graphics and Game Development.* Wordware, 2002.

[24] R. Meier, *Professional Android 2 Application Development.* Wordware, 2002.

[25] Appcelerator. (2011, Sep.) Titanium. [Online]. Available: http://appcelerator.com/

[26] PhoneGap. (2011, Sep.) Phonegap. [Online]. Available: http://phonegap.com/

[27] I. Ansca. (2011, Sep.) Cornoa sdk. [Online]. Available: http://anscamobile.com/corona/

[28] T. PA. (2003, Jul.) Tile based games. [Online]. Available: http://www.tonypa.pri.ee/tbw/start.html

[29] Z. Games. (2011, Sep.) Moai. [Online]. Available: http://getmoai.com/

[30] B. Snow. (2011, Aug.) Why most people don't finish video games. [Online]. Available: http://gigaom.com/2011/06/29/the-iphone-effect-how-apples-phone-changed-everything/

[31] B. Gatliff, "What is google android? an introduction to android programming," Jun. 2011, eSC Chicago.

[32] S. M. Blackburn and K. S. McKinley, "Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance," 2008.

[33] T. Krazit. (2006, Oct.) Armed for the living room. [Online]. Available: http://news.cnet.com/ARMed-for-the-living-room/2100-1006_3-6056729.html

[34] T. Riemersma. (2001, Jan.) Axonometric projectuins - a technical overview. [Online]. Available: http://www.gamedev.net/page/resources/_/reference/programming/isometric-and-tile-based-games/298/axonometric-projections-a-technical-overview-r1269

[35] S. Jobs. (2010, Jul.) Apple's iphone 4 press conference. [Online]. Available: http://www.engadget.com/2010/07/16/live-from-apples-iphone-4-press-conference/