Project Number: MXC-1065

Phantasm: A Game Engine

A Major Qualifying Project Report

Submitted to the faculty

of the

Worcester Polytechnic Institute

in partial fulfillment of the requirements for

the Degree of Bachelor of Science

by

Andrew Zafft

Date:

Friday, August 22, 2008

Professor Michael Ciaraldi,
Advisor

(This page is intentionally left blank)

Abstract

A game engine containing OpenGL[1], the Win32 API, C++ and common RPG themes was developed. Research into graphics and physics models was performed and then implemented. Fully customizable graphics, storyline development, and internal objects were created using a model-view-controller architecture.

[1] OpenGL® is a registered trademark of Silicon Graphics Inc

Acknowledgements

I would to thank Professor Mike Ciaraldi for his time and guidance on this project. Without his efforts this project would not have been as successful as it was.

Table of Contents

Chapter 1 Introduction

Games have existed for as long as human beings have lived. The act of entertaining oneself through exercises that serve primarily for self enjoyment is a purely human trait. Through the years, human beings have raised entertainment to an art form. Video games are the latest evolution of the concept. The power of the computer has allowed entertainment and gaming to be taken to a whole new level. We have new technologies that can expand and develop the gaming experience.

Game engines are the core of video games. These pieces of code display the virtual world to the computer screen, capture the frantic mouse clicks and keyboard commands from the user and manage the computer artificial intelligence, all to create an enjoyable experience. Advances in game engines create advances in video games, leading to better and better gaming experiences. In fact, it could be argued that since entertainment is exclusively a human trait, advances in entertainment are actually advances to the human race as a whole. The goal of this project was to expand my own personal understanding of game engines and maybe even advance game engine development.

1.1 Project Description

I have created a fully functioning role playing game engine. Its features include allowing future users to easily design and import custom graphics, objects and storyline development into the game. The game engine was developed with C/C++ code and interacts with the OS through the Win32 API. The 3-D graphical portion of the game was written using the OpenGL libraries. The game engine was developed and tested on Windows based systems, specifically in the Windows 2000 and Windows XP environments. The game engine can load game data from several different customizable configuration files that are easy to expand and develop.

The game engine is designed for use as a single player game. Character development is in the same spirit as the Diablo™[2] series. The player's character advances in power as they interact in evermore difficult environments. A system to develop character growth has been partially created, and future version will need to finish and polish these aspects of the game engine.

The engine presents the game from an overhead camera using 3-D models. The camera can rotate fully around the Z axis, and can both zoom in and zoom out on the Z axis. The main interface for the game is the mouse, but there is some minor keyboard interaction. The game engine loads game data from several configuration files, and has the ability to load pre-generated and randomly generated data. Saving and loading of game progress was to be implemented so that users can quickly and easily enter and exit the game, however this feature is currently not available. The engine can handle a relatively simple series of conditional events as dictated by the configuration files (such as action-response scenarios to be used for storyline generation). A basic physics model was implemented. When joined together, these pieces form the basis of an RPG game engine.

1.2 Goals and Plan of Action

The responsibilities for the engine include: loading graphic files (written for OpenGL), capturing user input, displaying information to the screen, loading pre-generated and randomly generated maps, implementation of a minimal computer artificial

[2] Diablo® is a registered trademark of Blizzard Entertainment

intelligence, development of a physics model, and maintaining the flow of the game during play.

The first step was to research existing game technologies. Using this information, a basic framework of the game engine was developed conceptually. Basic character attributes were planned out. Following this, there was some education into the programming skills that were required for this project. Win32 API was researched. OpenGL was learned. Finally the actual programming framework began. The windows framework was coded first, followed by the display system. Afterwards loading of XML configuration files was added. Following this, the user input was added. Next, topics in game flow were added and continuously tested as development proceeded. Then collision detection was implemented. Next random map generation was added. In game goals were implemented into the game framework. Finally, simple artificial intelligence was added. At the completion of the project, there existed a simple game engine that met all the criteria stated in the project description.

Chapter 2 Research

The first question one might ask is "What exactly is a game engine?" Unfortunately, there is not one clear and concise answer. Within the industry, there are many inconsistencies for what constitutes a "game engine." For some video games, the game engine comprises the entire game. In other games, the game engine solely performs the graphics rendering. And in still more, there is no game engine defined. The game engine is in itself, purely a concept. Games can function without a game engine explicitly defined. In fact, most likely due in part to wide variety of game genres, some games are better off without a game engine. But the lack of consistency does in itself, reveal some information. First and foremost this is a relatively new field. Even more so, this is a new field that is built on a changing foundation. Operating systems, graphics, and game theory are all in various stages of childhood, and have a lot of development left. Additionally, the ultimate goal of the game engine varies from instance to instance. Some are designed to be used and reused, with finely defined elements. Others are one shot deals, thrown together and then thrown away. For the purposes of this project, and to follow proper software engineering design, a reusable game engine was developed.

The game engine is the core software component of the video game. It provides the system that starts, loads, and runs the game. It abstracts the underlying system technologies and work to simplify the overhead needed for game story development and customization. It acts as middleware for the video hardware, operating system and input devices. It provides the structure for the game and takes care of all the various odds and ends. To achieve this, there must be a solid study of the role of this game engine. Interaction between game engine elements can pose significant problems if designs are changed during the course of the project. The complexity of these elements relies heavily on abstract programming and extensive preplanning. Therefore, much thought was devoted to the design phase. To assist in the design, the following sections will examine key topics in additional detail: games in general, writing and working with actual video games, role playing game considerations, graphics and rendering, animation, OpenGL specifics, game physics, scripting, artificial intelligence, Win-32 API concepts and the Model-View-Controller Architecture.

2.1 Games in General

Games have been used to hone skills, provide an outlet for energy or as a way to pass the time. While there are many possible goals for games, the primary goal of most games is entertainment. To do this, there must be a way of hooking the user and keeping them interested. Rules and storylines are two ways of achieving this.

Rules provide a twofold service: first they develop a framework for the game, and second they allow us to build expectations. The framework brings order to chaos and assists us in following along with a game. The framework makes it easier for us to understand the game. Building expectations also allows the user to get more involved with the game. But building expectations goes beyond basic involvement. If the user's expectations are in line with the expectations the game is expressing, then achievements and rewards in the game can be conveyed as achievements and rewards felt by the user. In this concept, the game becomes an extension of the player. This concept is examined in further detail in the Role Playing Games section. Conversely, rules that do not make sense to the logic of the game, or are too complicated to keep track of, detract from a user's expectations. This is important to games because if someone does not like what they expect from a game, they most likely will not continue to play the game.

The storyline is another way of keeping a player's interest, although a storyline in a video game is certainly not necessary. In fact, before computers, most games did not possess a storyline. Now however, games are often expected to include storylines. And the expectation has worked for hooking users. The storyline acts the same in video games as it does in novels, and makes the player/reader feel and grow with the characters in the story. All of these elements make up game elements that are good to incorporate into a game engine.

2.2 Programming Video Games

Computers have revolutionized games. Games can create abstract representations of the world we live in. System modeling and artificial intelligence represent huge, largely untapped, resources for our culture. Equations in physics which could never be calculated by hand in a person's lifetime can be completed within minutes. With proper development, the benefits of these advancements to science could be remarkable. These represent just a fraction of the capabilities of computers.

For now, however, the advancements come with a price. More power means more design considerations and longer implementation times. Luckily, there are quite a few tools out there that were used to assist with this task. Four have been selected to assist with this project: Microsoft Visual Studios 6.0, XML data organization formats, CVS implemented through WinCVS, and DOxygen.

Microsoft Visual Studios 6.0 provides the all important integrated design environment (IDE). The IDE provides a quick and easy platform to navigate the 100+

files needed for this project. Color coordination of code and help files available at the flick of the F1 key eases the strain of writing in a multi-system environment. Additionally, there really is no comparison for a debugger when trying to root out the cause of a programming failure.

XML data organization continues in the same mold. Importing and exporting of data files could exist without it however the complexity and confusion that would be left in its place would become a serious time sink as the size of the project increased. XML allows the creation of clear and easy to understand tags that ensure future users can more easily grasp the design and layout of the configuration files.

CVS implemented on both the development machine and on WPI's Sourceforge helped turn catastrophic crashes into minor inconveniences. When developing in a large systems environment, there really is no excuse for going without a versioning system.

Finally, DOxygen is a documentation building program that helps keep track of what everything is and what everything is supposed to do. This helps all those future users and readers understand what is going on, and for me to keep track of how this project interacts internally. DOxygen provides extra benefit to the programmer during development as they can review what a function does and compare it to what it was intended to do. This can help close up loop holes in my own thinking due to any concept changes that occur over time.

2.3 Role Playing Games

Role playing games (or RPGs) are a specific genre of games. An RPG seeks to transform the player into an alternate personality within the game. The concept has existed for as long as people have been playing games, and can be considered to be in the same vein as theatre or acting. In an RPG, the player adopts an ego (or role) and acts accordingly to how this role should act. The player goes through the game developing this role. Often times, the player actions influence and build the story. An RPG's core element, the telling of a story, is a common element in the history of all cultures across all corners of the world, and therefore can ring deeply with all races. This is opposite from other forms of entertainment, where the player is a "passive observer," and does not interact with or influence the story.[3]

2.4 Game Engine Structure & Elements

A game engine typically includes "a rendering engine ('renderer') for 2D or 3D graphics, a physics engine or collision detection (and collision response), sound, scripting, animation, artificial intelligence, networking, and memory management."[4] However, game engines are typically designed by dozens of programmers over several years. As there is only one programmer for this project, a game engine capable of rendering, animation, the physics engine, scripting, and artificial intelligence was created. Together these constitute a basic game engine. These topics will be examined in more detail in the following sections.

A game engine is only half of the picture however. The game engine controls how the game is processed, but it does not contain details on what the game is about. As this game engine is designed for an RPG, the game is really about the storyline and storyline development. This development is controlled through triggers and quests. A trigger is a conditional event that creates some result when the conditional part is met. A quest is a

[3] Role-playing game. Wikipedia. 26 Mar. 2008 <http://en.wikipedia.org/wiki/Role-playing_game>
[4] Game Engine. Wikipedia. 26 Mar. 2008 <http://en.wikipedia.org/wiki/Game_engine>

collection of triggers used for major storyline development. The details for quests and triggers are initialized from configuration files that are loaded during start up. These concepts will also be examined in more detail in the system design section.

2.5 Rendering

Graphics are a major topic in video games. While games can be created with limited to no graphics, it is commonly accepted today that games contain graphics. Rendering is the act of taking a scene and turning it into the graphics that are displayed on a computer screen. Once gamers were content to have some game genres exist with mediocre images. But, because of the increase in computer-generated imagery (CGI) in Hollywood, gamers are now demanding stunning, state of the art graphics for all types of video games. Much money in research and development has been invested into building better and faster graphics. Often times, teams of scores, or even a hundred, programmers devote multiple years of their lives towards developing the final effects that gamers witness. Unfortunately the scope, time and man-power requirements of this project were in insufficient supply to meet the needs of a complete video game. Therefore, only the basics are addressed and only a thin framework has been built.

The graphics that are displayed on the computer screen come from a few different sources. First, computer graphics these days follow a model similar to the filming of a movie; there is a scene and a camera. The scene is the world and the camera represents the eye of the viewer. During rendering, objects in the scene are rotated, translated and scaled according to their relative positions to the eye of the camera. Finally, the image is turned into pixels (or rasterized) and drawn to the screen's frame buffer. Often, objects are made up from many smaller geometric shapes (usually triangles). The general idea is that if you can approximate reality by creating complex shapes from many smaller geometric primitives. If the shapes are small enough, and if there are enough shapes, then you can create an approximation to almost any object in the physical world, including complex objects like spheres. Once all the objects are drawn to the scene, then the scene is rotated, transposed, and clipped to match the display field of the camera. On top of this, four techniques are commonly used: double buffering, light shading, quadratics and blending. Specifics of how OpenGL renders the model will be covered in the OpenGL section.

Double buffering is the act of having two screen buffers on which the renderer can draw. The general idea is that while one buffer is being displayed, the other buffer is being populated. When the rendering of one buffer is complete, the buffers are swapped and the old drawing buffer becomes the display buffer, and vice versa. The benefit of this method is that you never display an incomplete screen. This is a standard function that has been built into the OpenGL programming language, and was fully utilized by this project.

Another technique common to rendering is the effects of light on objects. Light can be used in a scene to increase the reality of the scene. Light is used in three forms: light source reflecting off an object, the effect of shadows of other objects thrown onto an object, and the dimming of some sides of an object because of a limited source of light. Through a series of complex equations using polygon normal vectors, shadows and the brightness of objects displayed in the scene are manipulated to create realistic looking effects. Unfortunately, the calculation of these equations can be very expensive in terms of CPU processing time. Therefore, it was decided by the designer of this document that there is going to be no use of light and shadows on objects. Any addition of these materials will be left to future designers.

Another technique that is used is for drawing complex shapes. This technique uses quadratic equations. The problem with creating rounded shapes is that they can take a very large number of triangles to achieve. To assist in graphics development, OpenGL has included a utility kit (glu.h) that allows quick and easy creation of rounded shapes through quadratic equations. There are many types of shapes that can benefit from quadratics, however the only ones implemented in this project are sphere, cylinders, and disks.

The final technique to be used during rendering is blending. Blending is the act of making an object appears partially translucent. This can create a dazzling affect (in this writer's opinion). While this technique can be expensive in terms of computer processing power, it is of the view of the designer of this document that the expense is well worth the effort.

2.6 Animation

Animation is all about making objects appear to move. The motion, however, is actually a trick of the animator. Scenes are built using still images only, and then displayed in rapid succession. The images are displayed so quickly that the brain joins the images together. Therefore, animation is actually the act of taking still images and making them appear to move. Animation from still-frames is the core of how all animation occurs, and so it has been around much longer than computers and has a large amount of research available on its practical use. Animation speed is measured in the number of frames displayed per second. Older, analog TVs typically display at 30 frames per second, with newer services displaying at higher rates.[5] An average computer monitor, for example, displays at 60 frames per second (computers also measure frames per second in hertz). Therefore, for a computer game to successfully imitate actual motion, the renderer must create a minimum of 30 frames per second, although a more standard 60 frames per second should be sought after. However, this only governs how often the scene should be drawn, and not what is drawn in the scene. This dilemma is more of a conceptual problem, and so there are a few different ways of achieving the same thing. Almost all concepts lead back to three generalized methods: Frame-by-Frame animation, Key-Frame animation and Parameterized systems.

Frame-by-Frame animation is a technique where an individual picture is used for every frame in the animation, similar to a flip book. As with the flip book, each picture is played in rapid succession and the brain blends the images together creating fluid motion. The major drawback to this form is that it requires a large amount of graphics work to create the motion, has a large space requirement to store the pictures, and is a rigid model that cannot react to unexpected changes.[6]

Key-Frame animation is very similar to the Frame-by-Frame technique. Frames are created, similar to the Frame-by-Frame method. However, instead of creating every single frame, only specific, key frames are created. The game then "morphs" the two consecutive frames together by varying degrees to achieve the illusion of animation. Key-frame animation reduces the time required during development and the disk space

[5] Analog television. Wikipedia, 16 Aug, 2008. <http://en.wikipedia.org/wiki/Analog_television>
[6] Animation. Brooklyn College of the City University of New York. 20 Mar. 2008 <http://acc6.its.brooklyn.cuny.edu/~lscarlat/GUI/animation.htm>

required over frame-by-frame animation however rigidity still exists in the animation model.[7]

A Parameterized system uses "object motion characteristics that are defined via kinematics or dynamics. Object motion can be derived from key frames using inverse kinematics or dynamics through linear interpolation or curved paths."[8] This creates simple, lifelike animations that are flexible and can accurately represent reality. The major drawback is that there must be an intuitive physics model to back up the motions.

In this project a generalized key-frame approach was used. To achieve this, objects are broken down into sub-parts that are fixed and atomic. The key-frames include locations of where and when these pieces should be drawn for each of the different animation sequences. Animation between key-frames is achieved by "morphing" or comparing the object's current location of each sub-part with the next key-frame location. This was intended to reduce the rigidity that exists in the key-frames system, as the starting point of each atomic piece is not be from a fixed location. Unfortunately, there was not enough time to implement more than a skeletal framework for animation in this project.

2.7 OpenGL Programming Language

OpenGL (or Open Graphics Library) is "a standard specification defining a cross-language cross-platform API for writing applications that produce 2D and 3D computer graphics."[9] OpenGL seeks to be a powerful and versatile language suitable for learning

[7] Animation. Brooklyn College of the City University of New York. 20 Mar. 2008 <http://acc6.its.brooklyn.cuny.edu/~lscarlat/GUI/animation.htm>
[8] Animation. Brooklyn College of the City University of New York. 20 Mar. 2008 <http://acc6.its.brooklyn.cuny.edu/~lscarlat/GUI/animation.htm>
[9] OpenGL. Wikipedia. 13 Aug. 2008 <http://en.wikipedia.org/wiki/OpenGL>

and commercial development. OpenGL operates through the drawing of various primitives including but not limited to: points, lines, triangles, rectangles (also called quads), bitmaps, and quadratics. OpenGL is a state system, in the sense that there are many states which you can put the system into, and these states affect the outcome of the drawing. OpenGL functions with a very specialized drawing pipeline, or rasterization, that will be discussed in the next section.

OpenGL is a very complex language, and elements of its functionality can be discussed at great length and in great detail. However, this is not a paper on using OpenGL, and so if more information is desired, it is suggested that you visit the OpenGL homepage.[10] There are a few elements that do deserve some discussion, as they become integral parts of the game engine. First, is the idea of a camera. This concept was first looked at in the Rendering section. OpenGL fully utilizes this feature. Data structure elements are included that meet this design pattern. The *camera* exists specifically as a class in the game engine. The camera encapsulates several different things, two of which are very important. First is the viewing area in the game, which is the field of view along the x, y and z axes. Second is the OpenGL window size on the desktop, otherwise known as the viewport. The viewport is important because in this game engine, the OpenGL window is actually divided into two separate viewports (one for the *openglwindow* and one for the *variablewindow*). The viewport is important because it controls how much each object is stretched or shrunk in the x and y directions so that the field of view match the viewport. The other two elements that are important in OpenGL are the Rendering Pipeline and Selection. Each of these will be discussed in a separate subsection.[11]

[10] http://www.opengl.org/
[11] OpenGL. Wikipedia. 13 Aug. 2008 <http://en.wikipedia.org/wiki/OpenGL>

2.7.1 Rendering Pipeline

The rendering pipeline is the mechanism of moving from a three-dimensional model to a filled framebuffer. A brief description of the process of the graphics pipeline was taken directly from Wikipedia.com:

1. Evaluation, if necessary, of the polynomial functions which define certain inputs, like NURBS surfaces, quadratics, approximating curves and the surface geometry.
2. Vertex operations, transforming and lighting them depending on their material. Also clipping non visible parts of the scene in order to produce the viewing volume.
3. Rasterisation or conversion of the previous information into pixels. The polygons are represented by the appropriate colour by means of interpolation algorithms.
4. Per-fragment operations, like updating values depending on incoming and previously stored depth values, or colour combinations, among others.
5. Lastly, fragments are inserted into the Frame buffer.[12]

A similar graphic representation of the process is the following:

[12] OpenGL. Wikipedia. 13 Aug. 2008 <http://en.wikipedia.org/wiki/OpenGL>

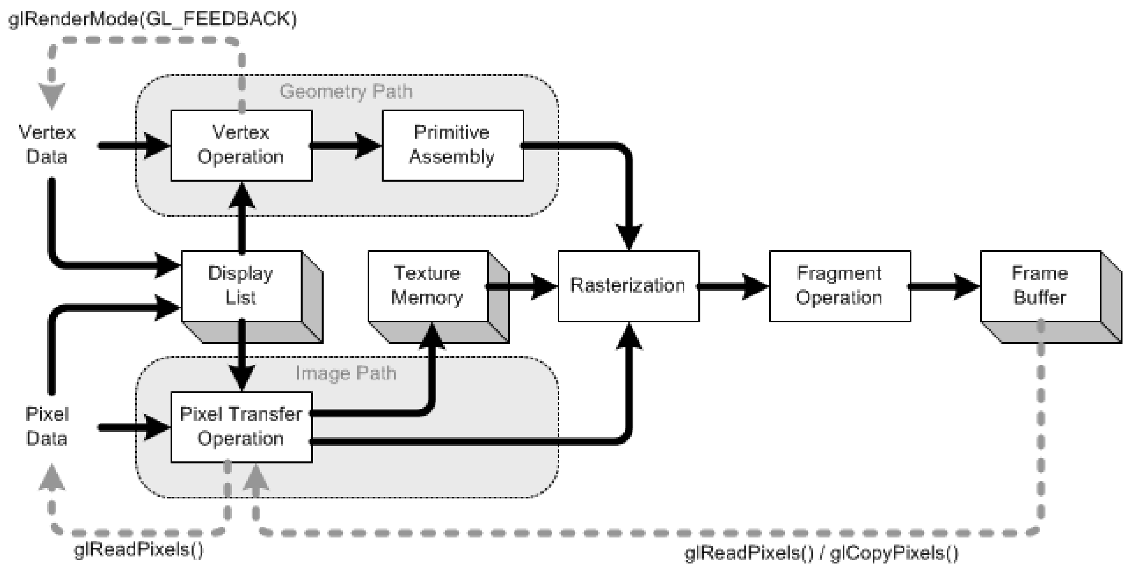


Figure 2.7-1 OpenGL Pipeline[13]

2.7.2 Selection

Selection is a very important part of the OpenGL language. Selection is all about determining what a user clicked on in an OpenGL scene. This can pose a very tricky problem. After all, the user is seeing a two-dimensional representation of a three-dimensional model. Luckily, OpenGL was designed with a solution to this problem. The answer lies in the drawing function.

When a player clicks on the OpenGL screen, the program receives the mouse click message with the coordinates of the click, in windows coordinates. After some coordinate manipulation, and the setting of certain flags, OpenGL can determine what was clicked on. The power comes from three sources. First, OpenGL takes advantage of the fact that it is a state system. The only way to determine what was clicked on is to rebuild the entire scene. However, the goal of this redraw is not to write to the frame buffer. Therefore, expensive texture mapping and blending functions are not needed.

[13] OpenGL Pipeline. 13 Aug. 2008. <http://www.songho.ca/opengl/files/gl_pipeline.gif>

This setting is achieved through telling OpenGL that all primitive drawing is done in SELECT mode, and does not affect the framebuffer. Second, OpenGL is really only interested in a limited portion of the window. Therefore, OpenGL creates a very small viewport which is a 3 pixel box around the very tip of the mouse cursor. Third, OpenGL uses a stack, call the names stack, that contains integer identifiers. During the drawing sequence, all primitives that lie outside the reduced viewport are ignored. If an object has any primitives drawn within the drawing area, then the integer identifier, or name, is pushed onto the names stack. At the close of the redraw, all names currently on the stack are returned to the caller after the drawing mode is changed from SELECT mode. As a side note, the game engine designed in this project takes advantage of this setup. Every object in the game has a unique identifier associated with it. When OpenGL pushes the integer name onto the stack, the program is actually pushing the game engine's unique identifier. In this way, the game knows exactly what object the player clicked on in the game. This list is provided to the control manager to determine what the proper action should be.[14]

2.8 Physics Modeling

A physics engine is "a computer program that simulates Newtonian physics models, using variables such as mass, velocity, friction and wind resistance."[15] As with most video game concepts, implementation varies. Physics model development can be a large time consuming task, a simple collision detection test, or not implemented at all. The focus of this project's physical model is on collision detection and collision response.

[14] Selection and Feedback. SGI. 12 Aug. 2008. <http://www.glprogramming.com/red/chapter13.html>
[15] Physics Engine. Wikipedia. 25 Mar. 2008 <http://en.wikipedia.org/wiki/Physics_engine>

In collision detection, the physics engine attempts to identify any object movement that causes two objects to occupy the same region of space. "Typically most 3D objects…[are] represented by two separate meshes or shapes. One of these meshes is highly complex…which the player sees….[The] second highly simplified invisible mesh is used to represent the object to the physics engine."[16] The use of two different models is simple: determining collisions between two models, each with hundreds of shapes, is very time consuming. Therefore, to maintain game speed, a generalized approximation is used. This approximation is sometimes refered to as a bounding box.

The second focus of a physics model is collision response. Collision response is used after a collision detection has occurred. The goal of collision response is to create an approximate reaction that mimics real world results. Physics engines often times use values such as weight and mutability to determine what consitutes a proper response. The most common response, however, is to back the moving object up just a little bit, so that two objects do not become locked together.

2.9 Artificial Intelligence

Artificial intelligence (AI) is an integral part of video games. AI represents the sole opponent of the player, in single player games. In this case, AI is divided into two groups: action determination and pathfinding. Action determination is the selection of the proper action for a computer controlled character, and pathfinding is used to determine the route between two points.

Action determination is a complex affair. The hope of action determination is to create an action/reaction that is similar to what a human would perform. Incorrect and

[16] Physics Engine. Wikipedia. 25 Mar. 2008 <http://en.wikipedia.org/wiki/Physics_engine>

improper action determination has ruined many video games. To combat this, weighty and complex data structures are often added to a game, with the hope of better approximations and better responses. Because of this, action determination is heavily dependent on the specifics of the system that is being developed. Additionally, because of the massive scope of this project, there was no extensive research into the development of action determination. There is only a very simplistic, "move to attack range and kill," mentality for the AI.

Pathfinding is the other major role of the AI. Pathfinding has the goal of moving an object from point A to point B. In the simplest situation, this is merely a straight line. However, many objects exist in most games, and these objects can block or slow movement. It is the responsibility of the pathfinding system to determine a best path for an object. Furthermore, this has to be determined within a reasonable amount of time. Oftentimes, pathfinding is equated to a sorting problem, and is solved with various search algorithms. This game engine uses a straight line pathfinding algorithm.[17]

2.10 Win32 API Programming Concepts

While Win32 programming does not directly have anything to do with game engines in general, this program was developed in the Windows environment. Therefore, some understanding of the Win32 application programming interface (API) is required. Additionally, as the game engine is developed under the Model-View-Controller architecture (discussed in the next section), the View has a direct responsibility for the Win32 API.

[17] Pathfinding. Wikipedia. 12 Aug, 2008. <http://en.wikipedia.org/wiki/Pathfinding>

There are many concepts and many different elements that are available in the Win32 API, and a whole MQP could be devoted to this subject. However, for the purposes of this project only six elements are important: windows classes, window handles, windows message procedure, the WinMain function and message loop, the graphic device index, and the menu. Each of these will be discussed, but not in any great detail.

Windows classes are a special data structure in the Win32 API. The windows class is used to define common characteristics of groups of windows that you will be creating in the future. All windows must have an associated windows class. The windows class defines, amongst other things, the message procedure that handles all messages being sent to the window. The windows class also defines which messages can be sent to the window. Therefore, this is a very important class, because if it is improperly created, it is possible for the window to not accept mouse and keyboard messages (making for a very dull gaming experience indeed!). Before a windows class can be used, it has to be registered with the windows operating system. The windows class has no physical appearance on the desktop however it does help define new windows that are created. It should be noted that the Win32 API was designed for use in C. Almost all classes are actually defined as structures.

Windows handles are, as one might expect, pointers to windows that have been created. These are the actual representations of the windows that we see and click on every day. A windows handle is associated with a windows class when it is created. Additionally, windows handles requires x and y positions (from the top left of the screen) and lengths and widths for the window. Common Win32 API programming standards require that there is one parent window that all other windows are drawn on top of. The basic Win32 API is a bare framework for windows programming. Elements such as scroll bars and buttons need to be explicitly programmed in order to properly function, and it takes some time getting used to the level of micromanagement needed.

The windows message procedure is the one stop shop for all windows interactions. The message procedure receives all messages (as long as they are allowed in the windows class) sent by the Windows operating system to the windows you have created. Painting, mouse & keyboard processing, and even system standby and screen saver mode are all funneled through the message procedure. Luckily, there are predefined responses built into the system, so one need only override the procedures where the default response is insufficient.

Where the windows message procedure is the heart of the Win32 API, the WinMain function and message loop are the heart of the game engine. Standard C and C++ programs start in the main function. The WinMain function is the equivalent starting point for a Win32 program. Because Windows runs as a shared resource system, the WinMain function begins with some additional parameters that the standard main function does not possess. For the most part these parameters are for backwards compatibility, and all should be circumvented with newer, better methods. The commonly accepted role of the WinMain function is to initialize the program, and enter into the message loop. This is not required, but has become a de facto standard. The message loop is where the game engine action resides. The message loop is actually the initial point of contact for all windows messages to the application, however the first step of the loop is to send all messages to their proper windows message handler. If desired, one could circumvent the standard message processing procedure at this point, but this can result in erratic program behavior. It is within this message loop that we have our controlling code for the game engine. For this design, this controlling code is responsible for calling each of the different phases that will be discussed in more detail in the system design chapter.

The graphic device index (or GDI) is a special element of the Win32 API. This object is used to draw everything that is visible on the screen. This object is very complex, and will be only lightly addressed. The importance of the GDI is two-fold. First, the GDI is written to directly by OpenGL when framebuffers are displayed. Secondly, scrollbars and the system log have to explicitly draw themselves through the GDI in order to be visible on the screen. Specific windows functions are used to draw the desired graphics to the GDI. The exact manner in which these functions accomplish this result are not entirely known by this writer, and for the purpose of this project not entirely important.

The menu is the final element examined. Windows are not required to have a menu bar, however it is very common for a window to possess one on their parent window. Options selected from the menu are sent as messages to the windows message function, and processed according to how the default response has been overridden (and the default responses do have to be overridden in this case, as the default response is to ignore the command).

All six of these elements are used by the game engine. To assist in their use, Visual Studios has included a default file called resource.h. The purpose of this file is to help navigate the various elements that are required to design and build windows.

2.11 Model-View-Controller Architecture

The model-view-controller (MVC) architecture is a common design pattern when programming. The MVC architecture is most useful when you have a complex or detailed system that does not have a linear presentation or when you would like the freedom to present the same dataset in different ways. The MVC architecture is made up of three separate elements (as the name suggests): the model, the view, and the controller. The model is the system you are representing. The view is how you are displaying the model. There can exist multiple views for any given dataset. The controller mediates between the view, the model and the user. Because of this, the MVC architecture is uniquely suited for a game engine. The game data structures become the model. The windows become the view, and the in-game logic becomes the controller. In this project multiple views are utilized. Additionally, because of the separation of the different elements of the architecture, it is theoretically possible for the major subsystems to be individually redesigned without having to redesign the whole architecture, and therefore the MVC architecture can be reused again and again. However, the goal of this project is not to create a template to be reused time and time again, and therefore there is not an extensive amount of time spent ensuring the easy retooling of the major subsystems. While one of the major goals of object oriented programming is to allow the user to benefit in the future from reusable code, the primary goal of this project is to create a solid game engine. Therefore, there was a focus in this project on clear and concise design over portability.[18]

[18] Model-view-controller. Wikipedia. 12 Aug, 2008. <http://en.wikipedia.org/wiki/Model-view-controller>

Chapter 3 Methodology

There was a very simple methodology for this project. Current video games were examined, examples of game engines were read about, and then basic designs were created. Additionally, program concepts in Win-32 and OpenGL had to be tested, as the designer of this project knew little to nothing about 3-D graphics and operating system interaction. Therefore, testing of various tasks in Win32 and OpenGL was performed. Following this, a basic game engine framework was created, keeping in line with a modified Model-View-Controller architecture. Basic objects were developed, and storage devices for these objects were implemented. Next, the whole game engine was reviewed from a customizable perspective, and XML reading and writing was added. Finally, finishing touches were put on for the user interface.

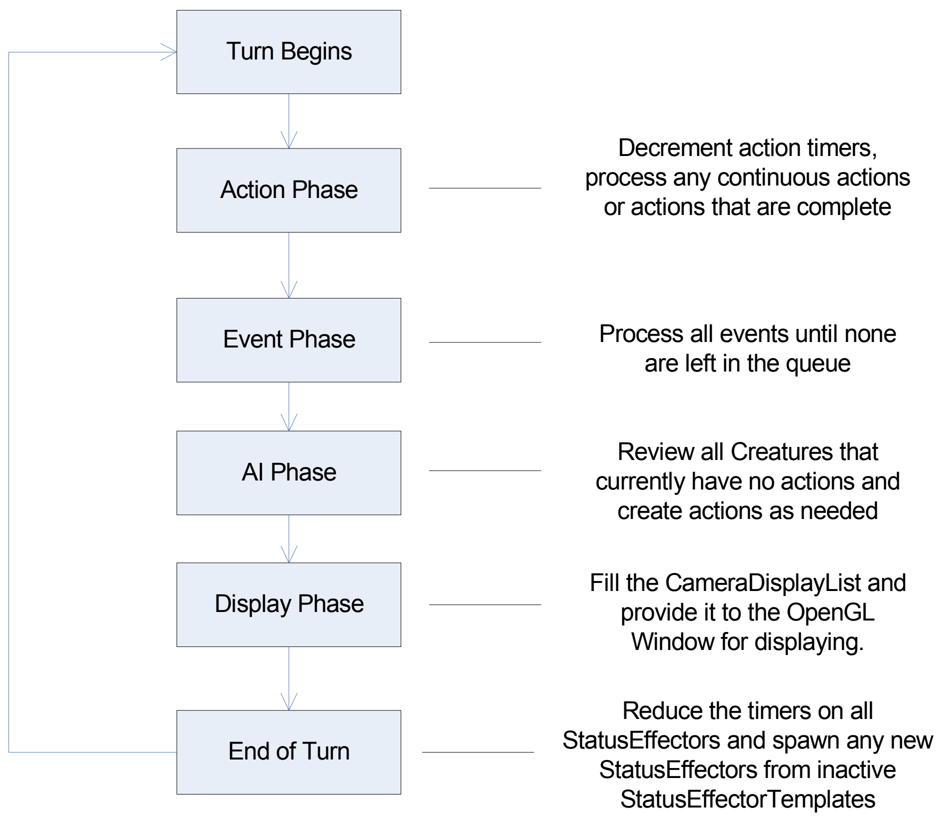
Chapter 4 System Design

The game was created with the Model-View-Controller architecture. As per the description, the Model houses all elements of the world, the View contains all aspects of displaying the Model to the screen and the Controller is responsible for mediating between the Model, the View and the Player, as well as maintaining the flow of the game.

The game engine gets its initialization data from 3 customizable xml files. The configuration files are each responsible for a specific set of data. The *GraphicConfig.xml* file is responsible for storing the file path and name of all the required *GraphicTemplates* that are used to display all objects in the game. The *ObjectTemplate.xml* file is responsible for storing the file path and name of all the *CreatureTemplates*, *BackgroundTemplates*, and *ItemTemplates* that are required for the game. Finally, the *StartupConfig.xml* file is responsible for holding any other required start up information. For future development of the game engine, the configuration files can easily be modified and appended to as needed.

The game is broken down into a continuous series of turns. Each turn consists of several steps. The game continues in this loop until the signal to close the game has been given. The flow of the turns is as follows: First, all current actions are examined and processed as necessary. After this, the game engine evaluates the events. Once event processing is complete, the game enters an AI phase. During this phase the game engine determines any applicable actions that should be added to the action list. Next, the scene is drawn to the screen buffer and the buffers swapped. Finally any clean up is performed and the turn ends. This list can be summarized as below:

Process of a Turn


Turn Begins
Action Phase
Decrement action timers, process any continuous actions or actions that are complete
Event Phase
Process all events until none are left in the queue
AI Phase
Review all Creatures that currently have no actions and create actions as needed
Display Phase
Fill the CameraDisplayList and provide it to the OpenGL Window for displaying.
End of Turn
Reduce the timers on all StatusEffectors and spawn any new StatusEffectors from inactive StatusEffectorTemplates

4.1 Basic Elements

To achieve the Model-View-Controller architecture, many elements, encapsulated by classes, have been designed. First and foremost is the concept of a *thing*. A *thing* is any object in the game that can be interacted, displayed, or targeted. *Things* are subdivided into 3 groups: *items*, *creatures*, and *backgrounds*. *Items* represent all usable, and possibly consumable, objects that cannot perform any independent action. *Creatures* represent any *thing* that can perform independent action. *Backgrounds* represent all *things* that make up the scenery and do not provide interaction beyond boundary detection.

The game is divided into areas called *maps*. All *items*, *creatures* and *backgrounds* for a game level are housed within the *map* collection. The *map* is the heart of the model, and contains the memory storage for every *thing*. *Maps* have their collection of *backgrounds* subdivided into subgroups called *sections*. *Sections* are of two types: rooms and corridors. The separate distinctions of the *section* is used primarily during the random map generation phase, and during random creature and item generation, and will be discussed later.

To continue the focus on randomization, all *things* have a *template* associated with them. You can think of the *template* as a mold. The *template* holds all the information needed to create a new *thing*. Each *template*'s role is to spawn new *things* upon request, with some amount of randomization. One example of this spawning is during the random map generation phase. Every *background* created during the map generation phase is from a corresponding *backgroundtemplate*. This concept is examined in more detail in the random map generation section. Additionally, any *things* explicitly listed in the *StartupConfig.xml* are created from the specified *thingtemplate*.

The graphical display of all *things* is through a series of atomic subdivisions called *BodyParts*. The role of each *bodypart* is to encapsulate all the information that is specific to that subpart of the *thing*. Each *bodypart* holds one *GraphicTemplate*. The *graphictemplate* contains all the information needed for the *bodypart* to be written to the OpenGL buffer. These concepts are examined in more detail in the Graphics section.

Interaction in the game is done through *Actions*. *Actions* are used to encapsulate any activity that a *creature* can perform. *Actions* are broken down into two general types: continuous actions that produce a continuous update (such as moving), and discrete actions where the outcome occurs only after a set amount of time. Furthermore, *actions* are implemented through a series of subclasses. Each subclass is responsible for a different type of *action*. There are currently seven *action* types: move, pick up item, drop item, equip item, unequip item, use item, and attack. One of targets of the future planning section is to expand this list.

A generalized class known as *events* is used to assist in the many tasks required by the game engine. As with *actions*, *events* are created through an inheritance hierarchy. *Events* are used for a large number of situations, primarily to make sure that all checks and balances that need to happen do happen. At their most basic level, *events* are used to pass messages to the *consolewindow*. Additionally, *events* are used to determine the effects of a collision and produce any wrap up information when a *creature* is killed, just to name a few different scenarios. Once again, future development will be responsible for expanding this section.

Objects, called *triggers*, are used to create *events* on behalf of the system. *Triggers* are meant to develop the flow of the storyline in the game. Each *trigger* is made up of a goal, an activation boolean, a unique name, and a series of *events* that are sent to the *EventQueue* when the goal is met. When a *trigger* is in the 'on' state, the *trigger* actively checks for completion of its goal. If a *trigger* is set to the 'off' state, then the *trigger* ignores all system messages. Goals are dependant on the type of *trigger*, and can be anything as simple as immediate completion of the goal to waiting for the *deathevent* of a specific *creature*. *Triggers* are turned 'off' and 'on' through *eventactivations*. *Eventactivations* are the one *event* that *triggers* always hear, regardless of their 'off' or ‘on’ state. The *trigger*’s listening is accomplished through evaluating current *events*, and therefore *triggers* are evaluated during the Event Phase.

Triggers are organized into groups called *Quests*. The player's gaming experience revolves around these larger goals and they are used to control the major elements of the storyline within the game. *Quests* share an activation boolean similar to *triggers*. *Quests* are activated and deactivated through *eventsactivations*, just like *triggers*. When a *quest* is activated, all *triggers* currently stored in the *quest* are sent to the global *triggermanager*. An example of a *quest* would be to rescue the princess from the mighty dragon.

Finally, there is one last base class known as *StatusEffectors*. *Statuseffectors* are used to create a change in the status of a *thing*. These changes can be immediate, such as healing or hurting, or continuous, such as a boost in speed. *Statuseffectors* are created from *StatusEffectorTemplates*. As with other templates in the game engine, the primary role of the template is to spawn new effects with some degree of randomization. Together all these classes make up the base classes required for the game engine to function.

4.2 Organizational Structure

The game engine is organized with the Model-View-Controller programming framework. The MVC framework is encapsulated by the following classes: *controlmanager*, *modelmanager*, and *viewmanager*. The *controlmanager* is responsible for the initial setup and interpretation of commands given by the player. The *controlmanager* houses all the *actions* and *events*. The *controlmanager* is in charge of loading the three configuration files. The *controlmanager* holds the random number generator, and the unique identifier generator (*idgenerator*). The *controlmanager* dictates the flow of the game.

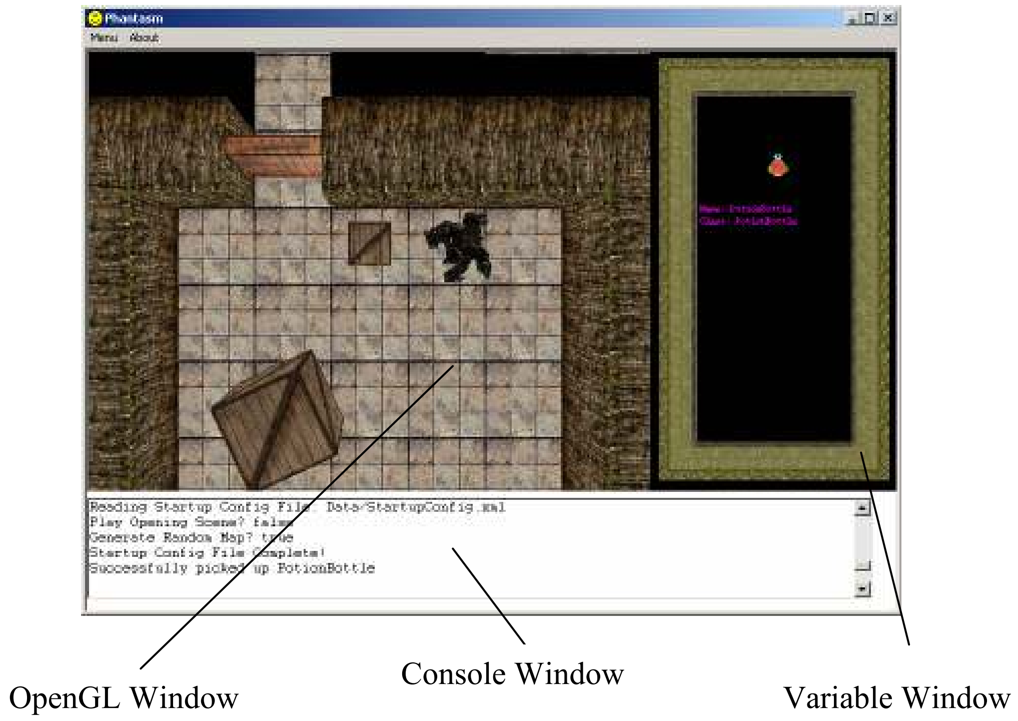
The *modelmanager* houses the *map*, which includes all *things* in the game. The *modelmanager* holds the *triggers* and *quests* for the game. The *modelmanager* contains the connections between the different *maps* in the game. The *modelmanager* holds all the different *thingtemplates*, and can spawn different *things* on request.

The *viewmanager* is responsible for building the user interface and passing the user commands to the *controlmanager*. The *viewmanager* holds the *graphictemplates*, the system *camera* and all necessary elements required to build the Win32 windows. The *viewmanager* also maintains the systems log. The *viewmanager* is responsible, through the *consolewindow*, for outputting the system log to the screen. The *viewmanager* controls the *variablewindow* display, and maintains the different possible *screens* that can be displayed. These will be examined in more detail in the User Interface section.

The *modelmanager*, *viewmanager* and *controlmanager* are defined in a shared global library and can easily interact with each other. Each class utilizes proper data member privatization to limit improper use.

4.3 User Interface

The goal of the user interface is to provide game data in a pleasant and easy to understand way. To accomplish this, some standards from general windows programming are observed. First, the application window has a menu bar and the familiar 'X' button in the top right of the window to close and exit out of the application. Secondly, the application exists on top of one master *backgroundwindow*, which encompasses the entire viewing area. On top of this *backgroundwindow* are 3 smaller windows, each which perform a specific functionality: the *openglwindow*, the *consolewindow* and the *variablewindow*. Please see the picture below:



```
Reading Startup Config File: Data/StartupConfig.xml
Play Opening Scene? false
Generate Random Map? true
Startup Config File Complete!
Successfully picked up PotionBottle
```

The *openglwindow* is the main display for the game, and is where the game play resides. Within this window is displayed the Player's window into the game engine world, and the user is able to use their mouse to click on various elements and perform various *actions* in the game.

The *consolewindow* acts as a textual output to the Player. System notices, information notes and any sort of text data that is sent to the Player is displayed here. The *consolewindow* maintains the system log, and is responsible for controlling the scrolling of the log. The *consolewindow* also holds information on the current system font.

The *variablewindow* is a multipurpose tool for the Player. The window allows the Player to view different facts about their *creature*, what *quests* they have to accomplish, and what *items* their *creature* is carrying. This is accomplished through a class called *Screen*. *Screen* is an inherited class that encapsulates the different views for the *variablewindow*.

To keep with Windows standards, there is a menu available. The menu is the point where Player's can start new games, save existing games, or load old games. The menu displays in the standard, acceptable Windows style. The menu also displays the current game's version number.

The game accepts several types of interactions from the player. The player can click the mouse, press a key, or select an option from the menu. If the player clicks the mouse, the game determines where and what, in game coordinates, the player clicked on. The coordinates are calculated in two forms: the exact position of the click and any *thing* that is under the mouse cursor when the mouse button is clicked. The game attempts to use some intelligence when interpreting mouse clicks. Basically, this means that what the player has clicked on determines what *action* is created. For example, if the player clicks on an *item*, then the game interprets this as a *pickupitem action*. If the player clicks on a *creature*, the system interprets this as an *attackcreature action*. Together, all of these elements make up an intuitive and easy to use user interface.

4.4 OpenGL Graphics

To build a proper model of the world, a few base graphic elements have been created. A *Vertex* houses a three dimensional point. *Shapes* have been created to encapsulate the different OpenGL graphics that can be drawn. *Shapes* are further divided into *discreteshapes* and *quadratic* subclasses. *Discreteshapes* are *triangles* and *quads*. *Quadratic shapes* are *quadraticcylinders*, *quadraticspheres* and *quadraticdisks*.

The OpenGL display of all *things* is through the *graphictemplate*. All *graphictemplates* are independently stored within the *viewmanager* as a central reference point. G*raphictemplates* have been segregated out for separate storage, so that overall program memory usage is reduced. Video games were studied in the Research chapter and it was discovered that the memory requirements for the information on displaying a *thing* far outstripped the memory requirements for the in-game representation of the *thing*. Therefore, independent storage of *graphictemplates* has been added to mitigate the effects of these 3-dimensional graphics. *Graphictemplates* are referenced by use of a unique name in the object configuration files. Furthermore, to simplify animation and reuse of data, *things* have been divided into *bodyparts*. A *bodypart* represents a subsection of a *thing*. Each *bodypart* has one *graphictemplate* it uses to display itself. Each *bodypart* is atomic, in the sense that it moves as a whole, and cannot be subdivided. Animation is performed by moving and rotating the connecting points between connected *bodyparts*.

Animation in the game is done with the Key-Frame animation style, and controlled by the *animationmanager*. Furthermore, Animation is divided into several different groups, identified by a type of *action* the *creature* is currently performing. When drawing a *thing* to the *openglwindow*, the *thing*'s current *action* (if any) dictates which animation is used. Each animation is subdivided into several steps, called *animationsteps*. An *animationstep* is a single pose for the *thing*. When multiple *animationsteps* are run concurrently, the effect is something similar to a pencil and paper flip book. To reduce the amount of graphics programming needed, the *animationsteps* do not define every *shape* at every moment in time. Instead, the *animationstep* tells the system where every *bodypart* should begin painting and the desired rotation. The *thing* keeps track of where each of its *bodyparts* reside. During a drawing period, the next *animationsteps* in the animation is compared to the *thing*'s record of its *bodyparts*. If the *bodyparts* are out of line, then the *thing*'s adjusts its *bodyparts* to more closely match the *animationsteps*. If the *animationsteps* destination *bodypart* location(s) are reached, then the *animationstep* is incremented to the next *animationstep* and the process continues. Unfortunately, there was insufficient time to properly develop animation into the game engine, and this section is left for future development.

During each round, the *openglwindow* displays all *things* within the viewable range. This is accomplished through the class *cameradisplaylist*. The *cameradisplaylist*'s only role is to collect a copy of all objects to be displayed. The *cameradisplaylist* holds all *bodyparts*, and their relative position from the center of the *camera*. During the Display Phase, the model is translated and the *bodypart* are drawn. To assist in the drawing processes, all *bodyparts* that use the same *graphictemplate* are stored in adjacent nodes. Additionally, because blending has special requirements, all *bodyparts* that contain blending are drawn last. The final result is a scene displayed in OpenGL.

4.5 Physics Modeling

A basic physics model has been created in the game engine. Collision detection and collision response are its primary roles. There is no specific, separate physics engine within the game engine per se, as most of the elements of the physics engine are accomplished in the Action Phase and the Event Phase.

Collision detection features are implemented through a bounding object format. This bounding object is a single, invisible structure that completely covers the *thing*. This structure is called the *CollisionModel*. To create some variability, three types of bounding objects have been created: a sphere, a flat plane, and a box. A *collisionmodel* encapsulates each of these three bounding methods. The *collisionmodel* is associated directly with the *thing*. The primary responsibility of *collisionmodel* is to compare itself to other *collisionmodels* and determine whether or not a collision has taken place. This comparison takes place during the Action Phase. If a collision does occur, a flag is raised and the details of the collision are added to the *eventqueue*.

Collision response is implemented in a round-a-bout fashion. Collision response is evaluated through *collisionevents* during the Events Phase. Each flagged collision examines factors (such as weigh and mutability) and determines a response that best models commonly accepted physics principals. The collision response is only a skeletal section of the code, and unfortunately will have to left for future development. Together, these elements make up the physics model.

4.6 Artificial Intelligence

Artificial intelligence consists of two separate areas: pathfinding and *action* determination of computer controlled creatures. Pathfinding is accomplished through search algorithms, and benefits largely from the fact that the game is navigable in mostly two dimensions. The other side of AI is performed during the AI phase. There is a very simple AI. *Creatures* will store the current *action* they are performing. The AI will look for any *creatures* with no current *action* and determine the appropriate *action*. Initially, all *creatures* will be hostile to the player. The addition of neutral and friendly creatures will be in the future planning section. Also, there was unfortunately not enough time to program even a basic AI model, and so any AI development will be left for future development.

4.7 Game Elements

The game engine has a very robust development system. There is an intuitive skill system. Most *actions* are associated with a *skill*. Each *creature* has a pool of *skills*. When an *action* is executed, the *skill* level of the *creature* is evaluated against a randomly generated number to determine success or failure, or possibly additional benefits or failures based on the degree of the *creature's skill*. The consequences are determined and applied during the Event Phase. Unfortunately, only a skeletal skill system was developed. The goal of future planning will be to expand this list. More details on skills will be given in Chapter 5.

One of the major elements of the game is attacking. Attacking is accomplished through the *attackaction*. *Attackactions* consist of the *creature* doing the attack, the target of the attack, and the type of *attack*. Types of available attacks are stored on the *creature*, and represents *attacks* that are available from two separate sources. First if the *creature* has any innate *attacks* (such as claws or teeth) the *attack* is defined in the *creaturetemplate*. Secondly, if the *attack* originates from an *item* being used, or wielded, by the *creature*, then the *attack* is defined by the *itemtemplate*. Each *attack* has a few aspects defined. First, there is a *skill* associated with the *attack*. Second, there is an attack speed, which gauges how quickly an *attack* occurs. Third, there is a group of *statuseffectorstemplates*. During an *attackaction*, if the attacking *creature* successfully strikes the target, the *statuseffectortemplate* spawns *statuseffectors* onto the target. This is the general method of how damage is transferred to the target. Furthermore, all available *attacks* for a creature are stored in a class called *AttackMaster*. During the AI phase, the *controlmanager* is responsible for selecting which *attack* is appropriate.

4.8 Random Map Generation

Having the entire game predefined, while possible, would result in a very dry and dull experience. Therefore, some thought has been put into adding an element of randomness. *Maps* can be randomly generated. The algorithm to generate them is fairly simple. The developer tells the *controlmanager* how many rooms they would like in a map. The *controlmanager* builds the *map* one room at a time. For each room, the *controlmanager* randomly selects a base room to build off. The *controlmanager* randomly chooses a direction and a distance, and attempts to create a temporary room. If the space is unoccupied the *controlmanager* creates a corridor connecting the new room with the base room. If the corridor cannot be created without crossing over an existing *section*, the *controlmanager* randomly selects another base room and repeats. The *controlmanager* continues until the desired number of rooms is reached, the maximum number of rooms is reached, or the *controlmanager* cannot successfully place another room after 20 iterations. It should be noted that a *map* can develop infinitely in any direction. There are no maximum or minimum boundaries covering the entire world during map generation.

To assist in the random generation, a few extra elements are needed. First are the *backgroundtemplates*. *Backgroundtemplates* exist only to generate new *backgrounds*.

Each *backgroundtemplate* has a designation: floor, wall, corner, wall ending, and door. This designation is found in the objectemplate file. As the *map* is being generated, the *controlmanager* requests a type of *backgroundtemplate*. The *map* selects from its list of available *backgroundtemplates*, and provides the *controlmanager* with a newly created *background*. In a similar sense to the *backgroundtemplate* are two more classes, *CreatureTemplates* and *ItemTemplates*. Each of template classes is used to randomly creating a *creature* or an *item*. The templates work by defining base values plus a range of values for each attribute that are required by the *thing*. To generate a new *creature* or *item*, the template randomly selects values within the predefined range and adds them to the base, and voila, a randomly generated *thing* is created. Together, these three template types are used to randomly generate maps and *things* specified in the startupconfig file.

Chapter 5 Manual

5.1 User Interface & Commands

Control of the game is through two sources, mouse and keyboard. They are broken down into basic view commands, mouse commands, and more detailed keyboard commands.

Basic View:

Left/Right/Up /Down Arrow - these controls move the *openglwindow* viewing area

Control-Left/Right Arrow - these controls rotate the *openglwindow's* view around the Z-axis

Control-Up/Arrow Down – these controls zoom in & zoom out of the *openglwindow's* view along the Z axis.

Mouse Commands:

Mouse - Left Click - this selects a 'valid' *thing* in the game. 'Valid' *things* are non-*background* objects. This also sets the *variablewindow* to the *thingselectionscreen*.

Mouse - Right Click – empty *background*- as long as you right click on any *background* in the map, this creates a *MoveAction* for your small crate, and the *MoveAction* is added to the *ActionQueue*. The *Action* is then processed as the game turns continue...until you get to your destination or there is a collision detected.

Mouse - Right Click – on an *item* - This creates a *PickUpItemAction* command. Your character moves to the spot of the *item*, and the *item* is moved from the *map* into your *inventorymanager*.

Mouse - Right Click – on an *creature* - This creates an *AttackAction* command. Your character moves to within attack range of the target *creature* and issue an *attack*.

Screen Control:

The window on the right of the screen is a *variablewindow*. There are 3 different types of views available: *MainPlayerScreen*, *ThingSelectionScreen*, and *InventoryScreen*.

MainPlayerScreen - display details about your *creature*. If you lose sight of this *screen*, you can always pull it up again by pressing 'M' or 'm'. Also, if you right click anywhere on the *variablewindow*, this screen is displayed.

ThingSelectionScreen - displays details about any 'valid' *thing* you have left clicked in the regular game.

InventoryScreen - displays all *items* in the *inventorymanager* of your *creature*. You can get to this *screen* at any time by pressing the 'i' or 'I' button. You can arrow up & down your *inventorymanager* if you have more than one page of *items*. Also, if you single left click on an *item*, the *item* becomes selected. If you double left click on an *item*, the *variablewindow* displays the *item* in the *thingselectionscreen*. Finally, if you left click on an *item* and drag it into the *openglwindow*, the *item* is dropped at the feet of the player’s *creature*. Also, if you highlight an *item* in your inventory, you can perform two additional options. If you press the ‘e’ or ‘E’ key, then a command to equip the *item* is issued. If you press ‘u’ or ‘U’ key, then you issue a command to unequip the *item.*

5.2 *Creatures*

5.2.1 Basic Attributes

Attributes represent the basic characteristics of a *creature*. Currently, only two attributes exist: hit points and speed. Hit points represent the amount of life a creature has. When hit points reach zero, the game throws a death event and the creature dies. Speed is a general value used to influence how fast a *creature* can perform an *action*. These attributes are needed to be expanded in future versions of this game, as this represents only an extremely basic framework.

5.2.2 Advancement & Experience

Unfortunately, due to time constraints, advancement and experience could not be addressed. It was the hope of this writer that a system similar to the Dungeons & Dragons®[19] system could have been developed, with experience points and levels of power. This unfortunately, has to be left to the future plans section.

5.3 *Items*

Items represent all things incapable of independent action. Furthermore, *items* are subdivided into two major subclasses. These are *UsableItem* and *EquippableItem*. *Usableitems* are all *items* that can be used to create the *UseItemAction*. When used, these *items* create some sort of effect. *Equippableitems* are any *item* that can be worn or wielded. These are further broken down into *WeaponsItem* and *ArmorItem*. When equipped, these *items* provide some sort of additional benefit to the wearing *creature*.

[19] Dungeons & Dragons® is a registered trademark of Wizards of the Coast Inc, a subsidiary of Hasbro International.

All *items* are capable of having *statuseffectors* on them. These *statuseffectors* can be activated in 3 different scenarios: activate on pickup, activate on wear, and activate on use. It is up to the creator of the *item* to make sure that the benefits of these *items* are reasonable, and do not destroy the game play of the game.

5.4 Skills

Skills are used for a variety of different actions. The *skills* are based on a hundred-point system, with zero being the worst and one-hundred being the best. Each *action* has a *skill* linked to it. As the *action* progresses, the benefits of the *action* are determined by the *skill* of the initiating creature. A base skill framework has been created, and needs to be extended in the future plans section.

5.5 XML File Usage

There are two general types of files: configuration files and object files. Configuration files have very little data in them and primarily serve as a collection point for the name and path of object files to be loaded. Object files contain the actual data for the *GraphicTemplate's* and *ThingTemplate's*. One quick note, order is important. The game engine makes heavy use of inheritance, and so children classes first load parent values from the XML before loading their class specific values. While some levels can have their values in any order, other levels require a specific order. It is best to follow the framework of the existing file structure and include every element that I've included in my XML files. When in doubt, just follow an existing model.

5.5.1 Configuration Files

GraphicConfig.xml – Responsible for loading all Graphic Templates. The GraphicConfig file uses the standard element name for the base, <ConfigFile>, and consists of 1 or more GraphicTemplate.xml files to be loaded. This is all that is required for the GraphicConfig.xml file. To add a new GraphicTemplate to the game engine, simply add in the new <GraphicTemplate> and <FileName> elements. Please see Section 5.5.4 for more details on adding new GraphicTemplates.

ObjectConfig.xml – This file is responsible for linking all CreatureTemplate, ItemTemplate, and BackgroundTemplate files. This file uses the standard base element name, <ConfigFile>. The file consists of a 1 or more <Template> elements. Each of the <Template> elements has 2 subelements. The first is <Type> which consists of 3 possible values: ItemTemplate, CreatureTemplate or BackgroundTemplate. This value must correspond with the ObjectTemplate type. The second part of the <Template> element is the <FileLocation> which contains the location of the ObjectTemplate.xml file to be loaded. Please see Section 5.5.4 for more details on adding new ObjectTemplates.

StartupConfig.xml - This file contains all the other variables that are needed for the game engine to initializing. As with the other two configuration files, this file uses the base element <ConfigFile>. This configuration has two roles. First, it defines some system values that are used by the game engine. These are <DisplayOpeningScene> and <RandomlyGenerateMap>. Please leave these at the default values. Secondly, this file defines some nonrandom elements for the map, which are stored under the <Map> element.

<Object> - This is a Thing created from a specified ThingTemplate. The <Object> has 3 sub-elements: <Type>, <ObjectTemplate>, and <Location>. <Type> can be Player (for the Player's Character), Creature, Item or Background. This type should match the type specified in the ObjectTemplate referenced. <ObjectTemplate> is the unique name of the ThingTemplate to be used to create the Thing. This name is defined in the ObjectTemplate.xml file. Finally, there is the element <Location>. <Location> has 4 sub-elements, the x, the y, the z and a rotation around the z axis. The rotation is in degrees, with 0 causing the thing to face the default graphic template direction.

5.5.2 Object Files

Graphic Template – This file defines a specific *GraphicTemplate*. The document uses the standard base element for non-configuration files, namely <Object>. The document has a few different option permutations. The following elements exist for a GraphicTemplate xml document

<Class> - Always GraphicTemplate

<Type> - Either Background, Item or Creature. Redundant and not fully used.

<Name> - The name referenced by the ObjectTemplate file using this GraphicTemplate.

<Graphic> - The name and path of any bitmap used for texturing

<Blending> - Optional. A boolean. True if any shape uses an alpha value. Alpha values are used to calculate the transparency of an object. A 1.0 means completely opaque.

<ScaleFactor> - Optional. Used to reduce the size of the GraphicTemplate and CollisionModel of a Thing. This has an x, y, and z component

<HasQuadratic> - Optional. Required if any shapes in the BodyPart are quadratics.

<BodyPart> - The definition of the GraphicTemplate shapes. This consists of a few different values. First there is a <NumberOfShapes>, which is the number of shapes used to make up the GraphicTemplate. Then, there is each shape that is used. See <Shape> below.

Base Template - There are certain common characteristics of all ItemTemplates, CreatureTemplates and BackgroundTemplates. While all these characteristics are combined in the Template file, the top portion of the template file pertains to these base characteristics. The required elements are <Type>, <Name>, <NumberOfBodyParts>, <BodyPart>, <CollisionModel>, <CanBlockMovement>

<Type> - The type of ObjectTemplate. This must be in the ItemTemplate, BackgroundTemplate, or CreatureTemplate families.

<Name> - This is the unique name of the Template. This name is referenced by the StartupConfig file to select a specific template to create a Thing from.

<NumberOfBodyParts> - This is the number of *BodyParts* associated with the Thing. This must be at a positive number at least greater than or equal to 1.

<BodyPart> This designates the beginning of a *BodyPart* description. The number of BodyParts and the count of <BodyPart> must be the same.

<GraphicTemplateName> - A sub-element of BodyPart. This is the unique name of the GraphicTemplate used to draw the BodyPart.

<CollisionModel> - This is used to declare the CollisionModel used by the ObjectTemplate. See CollisionModel definition below.

<CanBlockMovement> - A boolean determining if Thing's spawned from the ThingTemplate can block another Thing from moving into the space.

Item Template - This has the base Template elements, plus:

<ClassDisplayName> - This is the name of the type of Item.

ItemUsableTemplate - This has the Item Template elements, plus:

<StatusEffectorTemplate> - see StatusEffectorTemplate below

<ClassDisplayName> - the name of the class of items. Duplicate to ItemTemplate.

ItemWeaponTemplate - This has the Item Template elements, plus:

<AttackManager> - The object holding the description of attacks.

<NumberOfAttacks> - A positive integer containing the number of attacks

<Attack> - see Attack below

<SkillUsed> - the unique name of the skill associated with using this weapon

Creature Template - This has the Base Template's elements, plus:

<ClassDisplayName> - The name of the class of Creature (such as race or the like)

<HasInventory> - A boolean value for if the spawned Creature can have an inventory

<HitPoints> - a <Base> and <Variable> value representing the minimum and maximum hit point values of spawned creatures

<SkillManager>

<NumberOfSkills> - the total number of skills that the creature has.

Background Template - This has the Base Template's elements, plus

<RMGClass> - This is used during Random Map Generation. Available choices: Corner, Door, Floor, Wall, and Edge.

5.5.3 Common Base Objects

Shape – this is a subpart of GraphicTemplate. There are a number of different permutations depending on the <Type> of shape. The different types are QuadTexture, TriangleTexture, Quad, Triangle, QuadraticDisk, QuadraticSphere, and QuadraticCylinder. Additionally, any of these shapes can have an optional <Alpha> value after the <Type>.

<Type> Quad - has only the 4 vertex coordinates.

<Type> QuadTexture – has 4 vertex coordinates and 4 texture coordinate pairs.

<Type> Triangle - has only the 3 vertex coordinates.

<Type> TriangleTexture - has 3 vertex coordinates and 3 texture coordinate pairs.

<Type> QuadraticDisk - has a Translation coordinate, an InnerRadius, an Outter Radius, SegmentA and SegmentB which define the number of shapes used to approximate the quadratic.

<Type> QuadraticSphere - has a Translation coordinate, a Radius, SegmentA and SegmentB which define the number of shapes used to approximate the quadratic.

<Type> QuadraticDisk - has a Translation coordinate, a BottomRadius, a Top Radius, a Height, SegmentA and SegmentB which define the number of shapes used to approximate the quadratic.

CollisionModel – broken down into at least 2 parts.

<ModelName> - One of 3 different models: sphere, flat and box.

<width> - Used for all three. This represents the width/radius of the model.

<height> - Used by flat and box, ignored for sphere This is the height of the model.

<length> - Used only by box, ignored for flat and sphere This is the length of the model

Attack – A combination of StatusEffectorTemplates that occur on a successful Attack

<NumberOfStatusEffectors> - a positive integer containing the total number of StatusEffectorTemplates associated with this Attack

<StatusEffectorTemplate> - see below. There is one StatusEffectorTemplate for each number of StatusEffectors.

<AttackRange> - the maximum distance, in game space, that the creature must be from its target to be able to attack. 1.0 is equivalent to the width of a background.

<AttackTime> - the time it takes to process an attack before the result occurs. This amount is divided by the speed of the creature to determine the total number of game turns that must elapse before the attack is evaluated

StatusEffectorTemplate – A common storage device for creating StatusEffectorTemplates.

<Type> - The type of StatusEffectorTemplate (SET) that is being created. Currently only HealingStatusEffectorTemplate

<Base> - An integer base value of the effect

<RandomAmt> - An integer random value of each effect

<DelayBefore> - An integer in turns before the effect becomes active after adding

<RandomDelayBefore> - A random addition integer in turns before the effect becomes active after adding.

<IntervalBetween> - An integer in turns between each effect

<IntervalBetweenRandom> - A random integer in turns between each effect

<LengthOfEffect> - The number of iterations of the effects

<LengthRandom> - A random integer in iterations of the effects

<ActivateOnPickup> - Whether or not this effect activates on the owner when picked up

<ActivateOnUse> - Whether or not this effect actives on the owner when used

<ActivateOnEquip> - Whether or not this effect actives on the owner on equipping

<OneUse> - Whether or not this SET expires immediately after activating

<SENotifyOnSET> - Whether or not the spawned StatusEffector should notify its SET when it finishes

Skill – a Skill used by the Creature

<Name> - the unique name of the skill that is referenced by the ItemWeapon

<Value> - the adeptness of the skill, on a scale of 0 to 100.

Quest – a storage device for Triggers.

<Title> - A descriptive identifier for the Quest

<UniqueName> - the unique name of the Quest. This is referenced by the EventActivation.

<ActivateOnLoad> - True if this Quest is immediately activated and all triggers are sent to the global Trigger Manager

<NumberOfTriggers> - The total number of triggers associated with this quest

Trigger – a conditional object for storyline development

<Type> - The type of Trigger. Can be Trigger or TriggerDeath

<UniqueName> - The unique name of the Trigger that is referenced by the eventactivation.

<ActivateOnLoad> - True if the trigger is active immediately after being sent to the global trigger manager by the quest.

<NumberOfEvents> - the number of events that are sent after the trigger is completed. Currently a maximum of 3.

<CompleteImmediately> - if set to true, this sends all events to the eventqueue when this trigger is added to the global trigger manager

TriggerDeath – a conditional storyline event that looks for the death of a specific creature. It includes all the elements of Trigger

<TargetName> - the unique name of the creature that will satisfy this trigger

Event – an object that performs a wide variety of tasks. This is a base class for EventActivation and EventGeneration

<Type> - Either Event, EventActivation, EventGeneration

<Message> - A descriptive message that is displayed when the Event is evaluated

EventActivation – a subclass of Event that includes all xml structures as event and the following xml. This is used to activate and close triggers and quests

<ActivatedName> - the unique name of the trigger or quest

<ActivateTrigger> - notifies the name trigger to activate

<ActivateQuest> - notifies the name quest to activate

<CloseTrigger> - forces a completion of the named trigger

<CloseQuest> - forces a completion of the name quest

EventGeneration – a subclass of event that includes all exml structures as event and the following xml. This is used to generate new creatures, backgrounds and items from specified templates.

<TemplateType> - one of the following: CreatureTemplate, ItemTemplate, BackgroundTemplate

<TemplateName> - the unique name of the template

<NumberGenerated> - the number of things that are created

5.5.4 Adding New Object Files

How to add a new GraphicTemplate?

Create a new GraphicTemplate.xml file, and fill it out according to the GraphicTemplate object file above (listed in section 5.5.2). Make sure that the name used is unique amongst all the other GraphicTemplates. Then, add the file name and path to the GraphicConfig.xml file.

How to add a new BackgroundTemplate/CreatureTemplate/ItemTemplate?

There are two different types of template addition. You can use an existing class and change some of the preset values, or you can extend an existing class and add new features. If you find an existing class that has all the features and attributes that you need, then please use the class and follow the instructions below. If, however, you want to add attributes that do not currently exist, or implement new functionality that is not currently in the system, then you will have to do some coding. All template classes use an inheritance hierarchy. So pick a base class for inheritance. Please review my existing templates for more details on what needs to be completed in the new class.

Once you have found (or possibly created) the template class you want to use, do the following: create a new Background/Creature/ItemTemplate file and fill it out according to the object file above (listed in section 5.5.2). Make sure that the template name is unique amongst all types of templates (BackgroundTemplates, CreatureTemplates, and ItemTemplates). Add the file name and path to the ObjectConfig.xml file. If testing, add the name of the template file to the

StartupConfig.xml file, along with a location, and verify that the Template properly spawned a Thing.

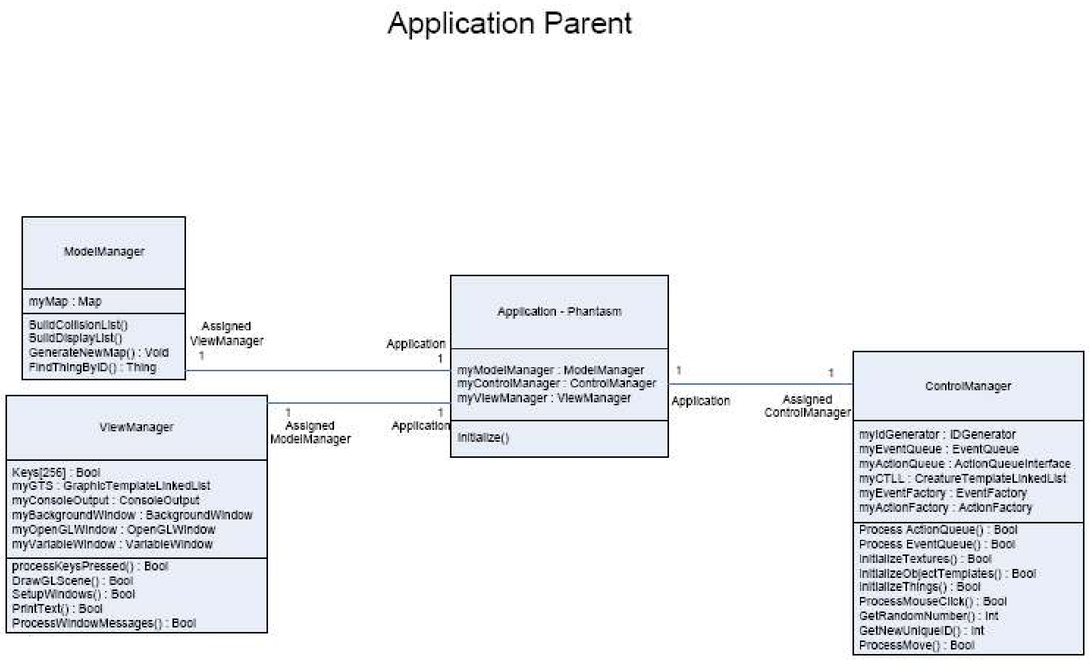
Chapter 6 Future Plans

A lot has been accomplished in this project. Game engine structure, physics modeling, graphics, Win32 structures and XML loading were all implemented, just to name a few features. And while a simple RPG was created, there are still many elements left to be developed or expanded upon. First, AI was never fully implemented and for this to become a fully functional game engine AI must be present. Next, saving and loading of game data should be added. Gone are the days when an RPG would not support saving. Animation has a basic framework only, and therefore needs to be implemented and tested. Additionally, player advancement, while not necessary for all games types, is certainly a staple of every good RPG. Therefore, more time should be allocated to hammer out an intelligent and unique plan for advancement, and then this plan should be implemented. Related to this is the development of new skills and creature attributes. As it stands now, only a minimal set of attributes and skills have been implemented.

Some game elements exist in a semi-complete form. Triggers, quests, events, and actions are currently insufficient for a finished and polished RPG. More thought needs to be given to the development of triggers, as they drive the game's story development. There should be the addition of more screens for the variable window, such as a map screen and a quest screen. Additionally, there needs to be some sort of connection points between different maps. Collision models and collision responses need to be fully implemented. Finally, the game needs to incorporate non-hostile creatures for advanced story development. Upon completion of all these elements, there should be a fully functional game engine

Appendix A UML

A.1 Class Models



Application Parent
ModelManager
myMap : Map
BuildCollisionList()
BuildDisplayList()
GenerateNewMap() : Void
FindThingByID() : Thing
Assigned ViewManager
1
Application
1
Application - Phantasm
myModelManager : ModelManager
myControlManager : ControlManager
myViewManager : ViewManager
Initialize()
ViewManager
Keys[256] : Bool
myGTS : GraphicTemplateLinkedList
myConsoleOutput : ConsoleOutput
myBackgroundWindow : BackgroundWindow
myOpenGLWindow : OpenGLWindow
myVariableWindow : VariableWindow
processKeysPressed() : Bool
DrawGLScene() : Bool
SetupWindows() : Bool
PrintText() : Bool
ProcessWindowMessages() : Bool
1
Assigned ModelManager
1
Application
1
Application
1
Assigned ControlManager
ControlManager
myIdGenerator : IDGenerator
myEventQueue : EventQueue
myActionQueue : ActionQueueInterface
myCTLL : CreatureTemplateLinkedList
myEventFactory : EventFactory
myActionFactory : ActionFactory
Process ActionQueue() : Bool
Process EventQueue() : Bool
InitializeTextures() : Bool
InitializeObjectTemplates() : Bool
InitializeThings() : Bool
ProcessMouseClick() : Bool
GetRandomNumber() : Int
GetNewUniqueID() : Int
ProcessMove() : Bool

Figure 2 Application Parent

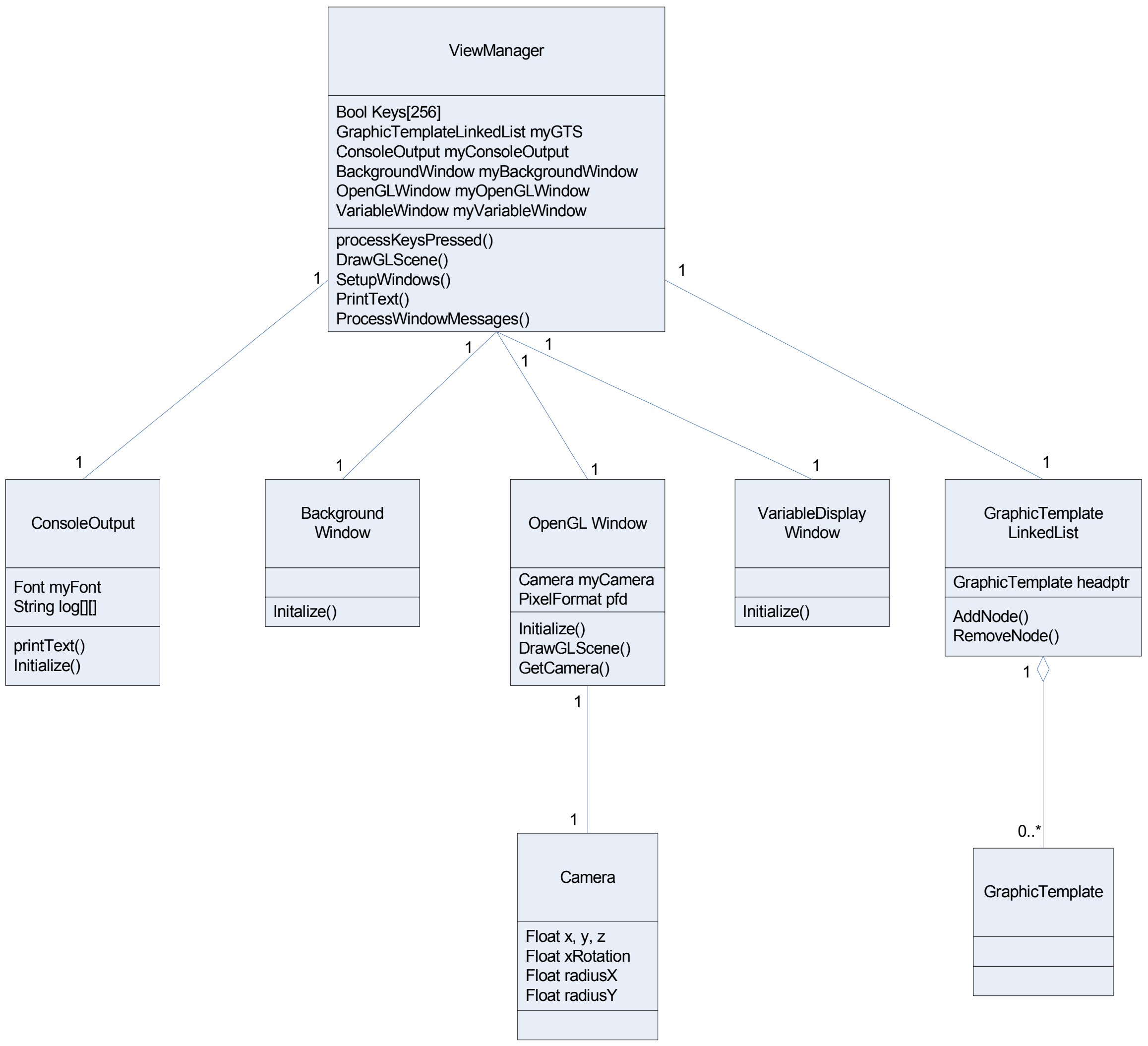


Figure 3 ViewManager Class

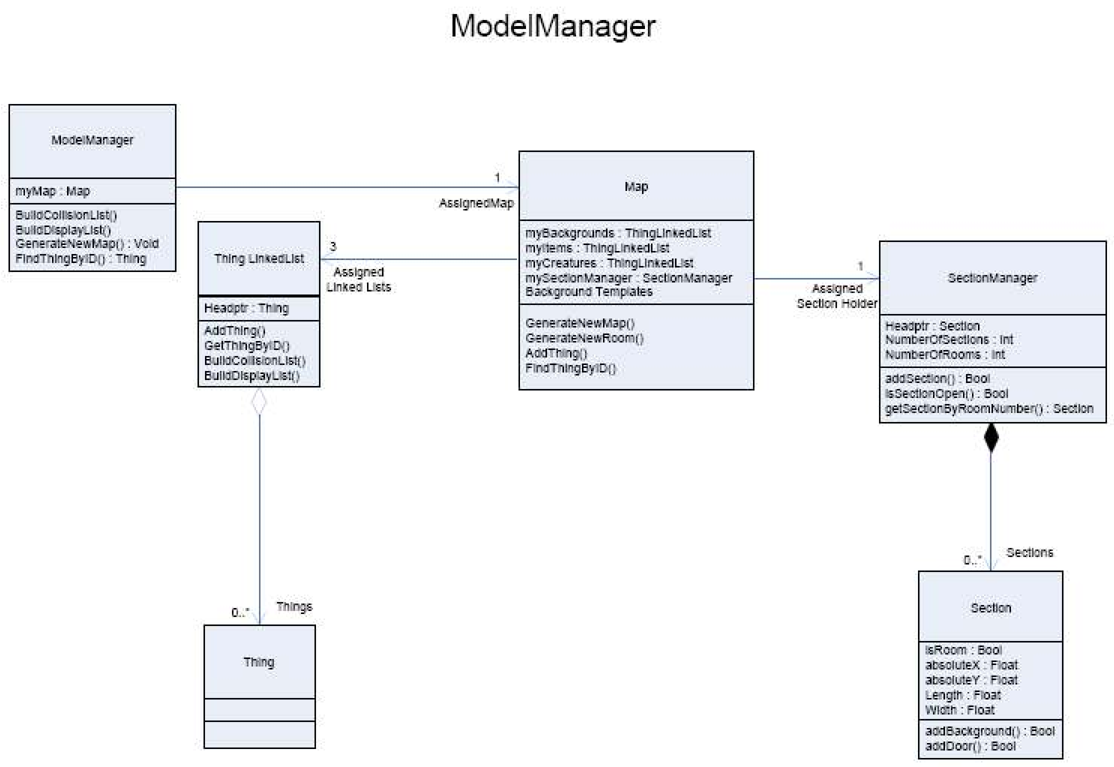


Figure 4 ModelManager Class

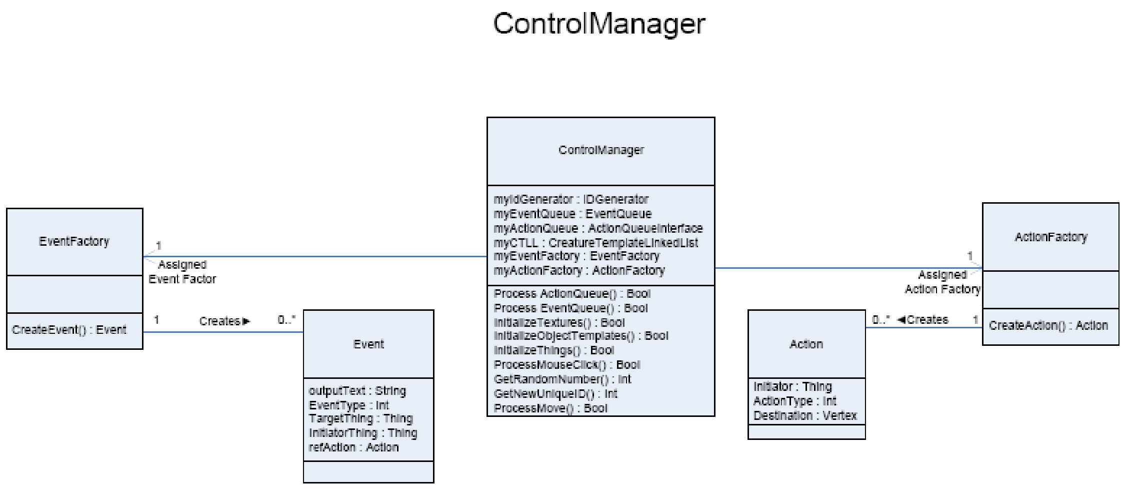


Figure 5 ControlManager Class

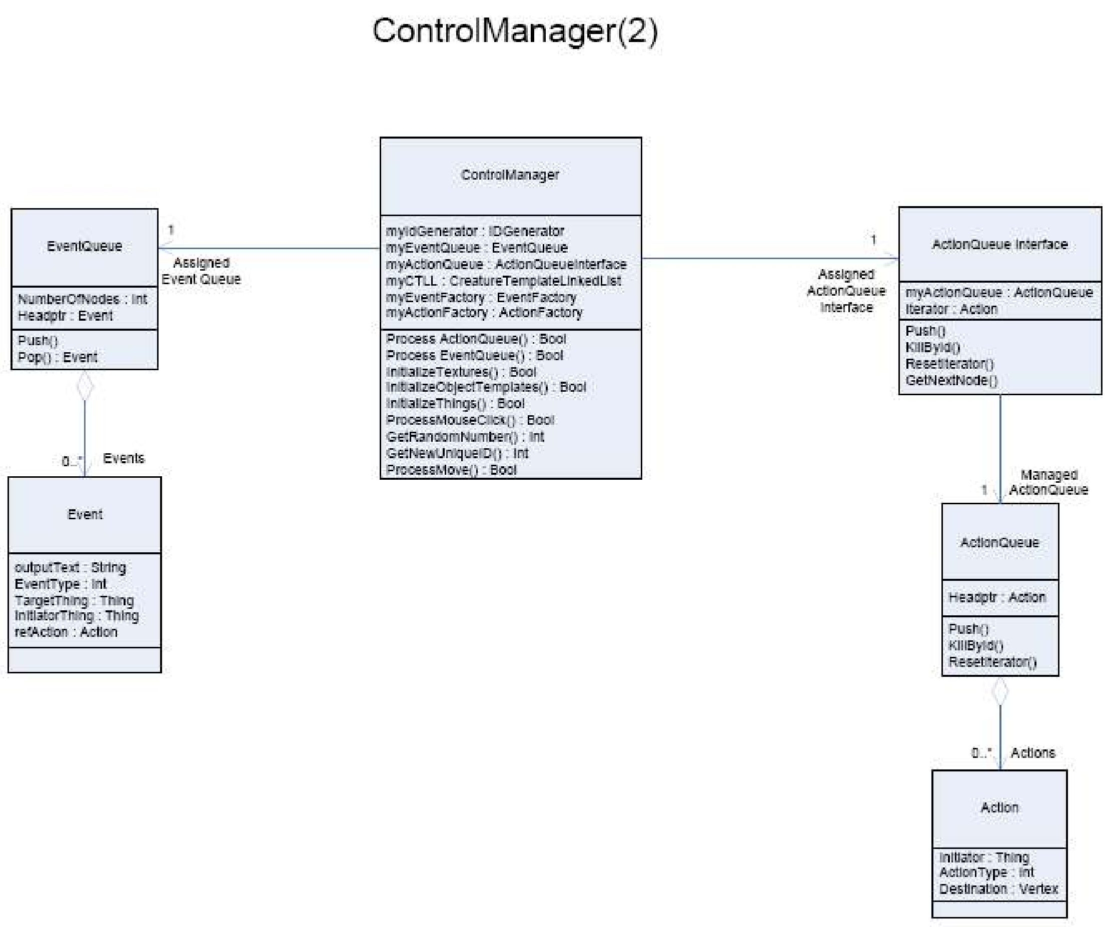


Figure 6 ControlManager Part 2

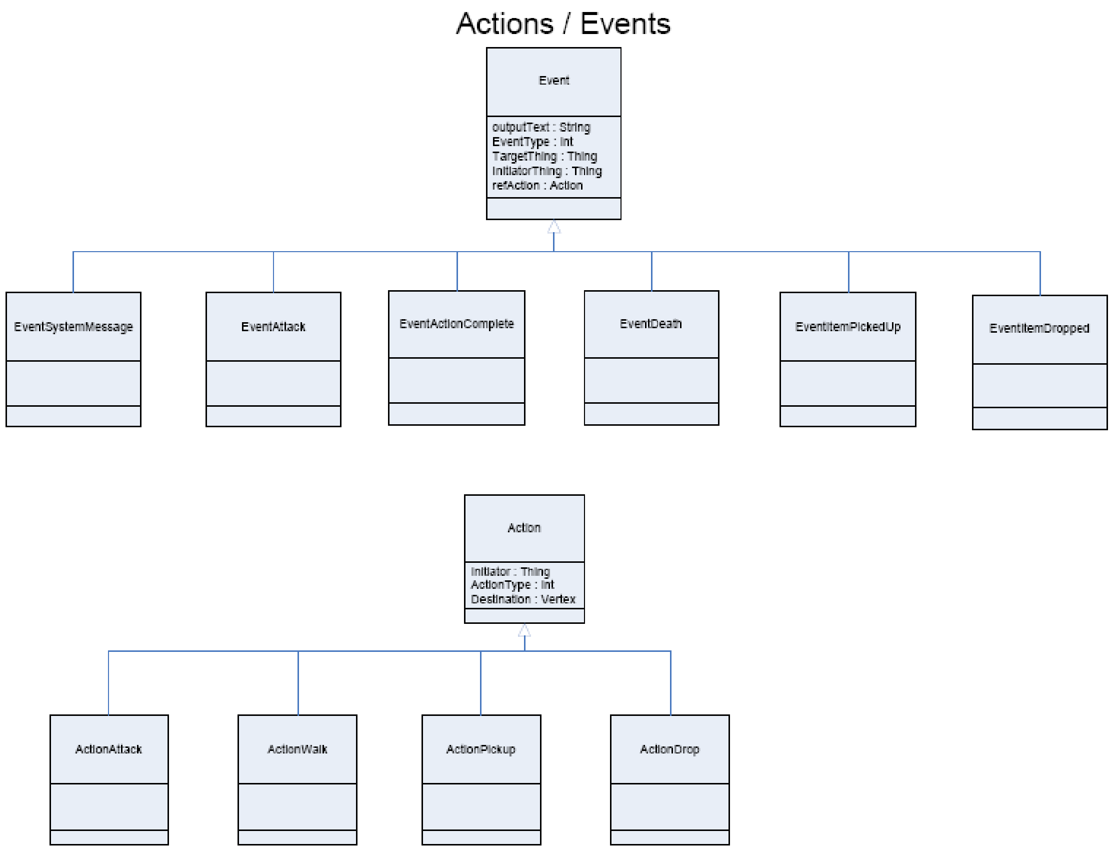


Figure 7 Actions and Events

Thing

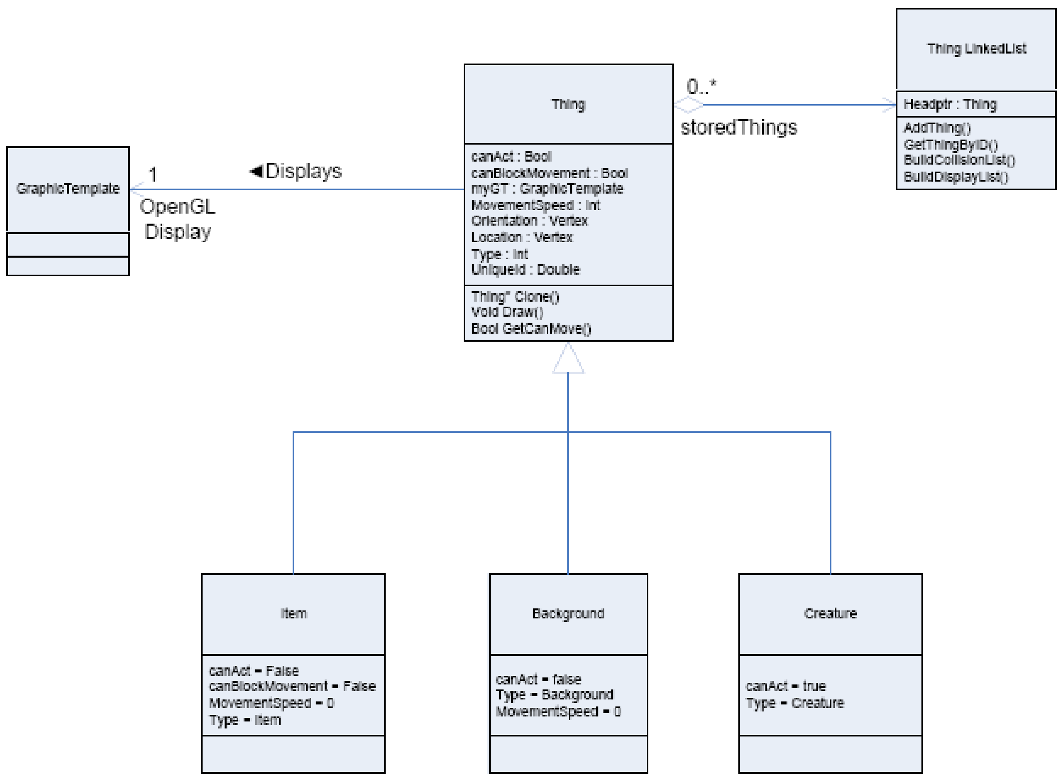


Figure 8 Thing Class

Appendix B Use Case Scenarios

Possible Actors: Developer, Player, Computer Controlled Creature

B.1 User Interface Use Cases

Use Case:	**Start a new game**
Actor:	Player
Goal:	Player desires to start a Level
Precondition:	The executable is currently running
Flow of Events:	1) Player clicks on the Menu at the top
	2) The Player selects the New Game option
	3) Player clicks on the desired options and clicks Play
Exit Condition:	4) Player begins playing the level

Use Case:	**Save an existing game**
Actor:	Player
Goal:	Player desires to save the game.
Precondition:	A game has been started
Flow of Events:	1) Player clicks on the Menu at the top
	2) The Player selects the Save Game option
	3) The Player selects the name of the saved data file
	4) The game pauses while data is saved
Exit Condition:	5) The game is saved

Use Case:	**Load a previously saved game**
Actor:	Player
Goal	Player desires to load a saved game
Precondition:	The executable is currently running.
Flow of Events:	1) Player clicks on the Menu at the top
	2) The Player selects the Load Game option
	3) The Player selects which data file to load
	4) The game pauses while data is loaded
Exit Condition:	5) Previous play location is loaded and Player resumes playing game

Use Case:	**Quit the game**
Actor:	Player
Goal:	Player desires to quit the game
Precondition:	The executable is currently running
Flow of Events:	1) Player clicks on the Menu at the top
	2) The Player selects the Exit Game option
	3) The Player is prompted to save the game before exiting
Exit Condition:	4) The game is closed

Use Case:	**Rotate the display screen**
Actor:	Player
Goal:	The Player desires to rotate the screen around the Z-axis

Precondition	A game has been started
Flow of Events:	1) The Player holds down the control key and presses the left or right arrow
Exit Condition:	2) The screen has been rotated to the desired position

Use Case:	**Zoom in or zoom out the display screen**
Actor:	Player
Goal:	The Player desires to zoom the screen in or out
Precondition:	A game has been started
Flow of Events:	1) The Player holds down the control key and presses the up or down arrow
Exit Condition:	2) The screen has been zoomed in to the desired position

Use Case:	**Examine all Items carried by the Player's Creature**
Actor:	Player
Goal:	Player wishes to examine what Items they are carrying
Precondition:	A game has been started
Flow of Events:	1) The Player pressed the “I” or “i” key
	2) The Variable Display window loads the Inventory screen
Exit Condition:	3) Items carried are displayed in the Variable Display window

Use Case:	**Examine the Player's Creature**
Actor:	Player

Goal:	The Player wishes to examine the stats of their Creature
Precondition:	A game has been started
Flow of Events:	1) The Player presses the “C” or “c” key
	2) The Variable Window displays the MainPlayer screen
Exit Condition:	3) The stats of the Player’s Creature are displayed in the Variable Window

Use Case:	**Examine the Player's Creature’s Equipment**
Actor:	Player
Goal:	The Player wishes to examine the equipment of their Creature
Precondition:	A game has been started
Flow of Events:	4) The Player presses the “E” or “e” key
	5) The Variable Window displays the Equipment screen
Exit Condition:	6) The equipment of the Player’s Creature are displayed in the Variable Window

Use Case:	**Examine a Creature or Item in the Map**
Actor:	Player
Goal:	The Player wishes to examine a Creature or Item in the Map
Precondition:	A game has been started
Flow of Events:	1) The Player left mouse clicks on the Creature or Item
	2) The Variable Window targets the Creature or Item
	3) The Variable Window displays the Thing screen

Exit Condition:	4) The Thing's stats are displayed in the Variable Display window (a more limited view than the MainPlayer Screen)

B.2 In-Game Use Cases

(all have a precondition of a game running)

Use Case:	**Move the Player's Creature**
Actor:	Player
Goal:	Player Desires to move his creature
Entry Condition:	1) Player right mouse clicks on the desired destination.
Flow of Events:	2) The system checks to see if the destination is in the game window.
	a) If destination is not on screen, command is ignored. End use case.
	3) The system checks to see if there is an Item or a Creature at the location
	a) If there is this becomes a "Pick Up an Item" Use Case or an "Attack a Creature" Use Case. End use case
	4) A MoveAction is requested from the ActionFactory
	5) The Player's Creature and the target model coordinates are added to the new Action.
	6) The Action is added to the ActionQueue
	7) …Wait until Action is popped from the Action Queue

8) Compare Creature coordinates to destination Coordinates. Determine direction from coordinates and distance moved from speed.
9) Apply direction & distance to Creature coordinates and create a potential move
10) Check to see if potential move collides with any existing Things
    a) If no collision, apply new coordinates to Creature
    b) If collision, Create a CollisionEvent with Action, Creature, and collision object. Add to Event Queue. End use case.
11) Check to see if Creature location is at destination location
    a) If not at destination, Action is pushed back onto ActionQueue
12) Continue Processing Action Queue…

Exit Condition: 13) Creature reaches destination or a CollisionEvent occurs

Use Case:	**Pick up an Item**
Actor:	Player or Computer Controlled Creature
Goal:	The Player or Computer Controlled Creature decides to pick up an Item on the ground in the Map.

Entry Condition:	The Player right mouse clicks on an Item on the ground, or Artificial Intelligence decides a Computer Controlled Creature wants to pick up the item.

Flow of Events:

1. A Pickup Item Action is requested from the ActionFactory
2. Pointers to the Creature initiating and the targeted Item are added to the Action
3. The Action is added to the Global ActionQueue
4. …Wait until Action is popped from the ActionQueue
5. Action is popped off ActionQueue
6. System checks if the Creature is within pickup range of the Item (pickup range to be determined later)
   a) If not, the Action is processed as a Move Action with the destination being the Item's location. Original Action pushed back onto the ActionQueue, return to Step 4.
7. System checks if Creature has a larger Carrying Capacity than the Weight of the Item (weight check)
   a) If weight is too great, Action is discarded. Event System Message created with "Failure to pick up due to weight." Event added to the EventQueue. Exit Use Case.
8. Item removed from Map's Thing LinkedList
9. Item added to Creature's InventoryManager
10. Event System Message created with "Item successfully picked up." Event added to the EventQueue.

Exit Condition:	11. The Item is picked up or a failure message is displayed

Use Case:	**Drop an Item that is carried**
Actor:	Player
Goal:	The Player wishes to drop and Item carried by their Creature
Flow of Events:	1) The Player selects the Inventory tab from the Variable Display window
	2) The Player left clicks & drags the desired Item to be dropped
	3) The Player releases the left mouse button somewhere over the Map
	4) A DropItem Action is requested from the ActionFactory
	5) The DropItem Action is given the Item desired to be dropped and a location on the Map. The Action is added to the Action Queue
	6) …Wait until Action is popped from the Action Queue
	7) Verify that the Item is still in the possession of the Creature
	8) The Item is moved from the Player's inventory to the Map
	9) Discard the Action
Exit Condition:	10) The Item appears at the destination location

Use Case:	**Equip an Item that is carried in the Player's Creature Inventory**
Actor:	Player

Goal:	The Player wishes to equip an Item that is carried by their Creature
Precondition:	The Variable Window is displaying the Inventory screen

Flow of Events:

1) The Player left clicks the Item they wish to equip
2) The Player left clicks the Equip button
3) If the Item has more than one Equip Location, the Player is prompted to select which location
4) The system checks to see if the Creature can equip the Item at the specified location
   a) If the Creature cannot equip the Item, then create a SystemMessage Event communicating failure and add to the EventQueue. End use case.
5) The system checks to see if the Creature already has an Item at the specified location
   a) If the Creature already has an Item equipped there, then create a SystemMessage Event communicating failure and add to the EventQueue. End use case.
6) The EquipItem Action is requested from the ActionFactory
7) The Action is given the Creature, the Item being Equipped, and the Equipped Location
8) The Action is added to the ActionQueue
9) ….Wait until the Action is popped from the ActionQueue
10) Verify that the Item is still in the Creature's possession.

11) Add the Item to the Creature’s EquipmentManager

12) Recalculate benefits from Equipment

13) The Action is discarded

14) Create a SystemMessage Event communicating success and add to the EventQueue

Exit Condition: 15) The Item is equipped or a failure message is displayed

Use Case: **Unequip an Item that is being equipped by a Creature**

Actor: Player

Goal: The Player wishes to unequip an Item that is being equipped by their Creature

Precondition: The Player is viewing the Equipment screen

Flow of Events:

1) The Player left clicks the Item
2) The Player selects the Unequip button
3) A new UnequipItemAction is requested from the ActionFactory
4) A new Action is given the Creature and Item to be unequipped
5) …Wait until Action is popped from the Action Queue
6) The Item is moved from the EquipmentManager to the InventoryManager on the Creature
7) The benefits from Equipment are recalculated

8) A SystemMessage Event is created communicating success, and is added to the EventQueue

9) The Action is discarded

Exit Condition: 10) The Item is unequipped and a success message is displayed

Use Case: **Use an Item carried by the Player's Creature**

Actor: Player

Goal: The Player desires to use an Item carried by their Creature

Precondition: The Player is viewing the Inventory screen

Flow of Events:

1) The Player selects the Item in the Variable Display window
2) The Player presses the Use Item button
3) If the Item requires a target, the Player left clicks on the target for the Item
4) A UseItem Action is requested from the ActionFactory
5) A new Action is given the Item, Creature and possibly the target
6) …Wait until Action is popped from the Action Queue
7) Verify that the Item is still in the Creature's inventory.
    a. If not then create a SystemMessage Event and add to the Event Queue. End use case
8) Verify that the Creature is within range of the Target. If not, then treat as a movement Action and go back to step 6)

9) Create a UseItem Event and add the Action to the Event. Add the Event to the EventQueue
10) ....Wait for the Event to be popped from the EventQueue
11) Perform the action requested by the Item (Item.DoAction())
12) If applicable, remove the Item from the Creature's inventory
13) Create a SystemMessage Event with the results. Send to the EventQueue
14) Discard the Action

Exit Condition: 15) The Item is used or an error message is displayed

Use Case: **Open a Door**

Actor: Player

Goal: The Player desires to open a door in the game

Flow of Events:
1) The Player right clicks on the closed door in the game
2) A OpenDoor Action is requested from the ActionFactory
3) The new Action connected with the Player and the door and added to the Action Queue
4) …Wait until Action is popped from the Action Queue
5) If Creature is out of range of the door, then treat as a movement Action and go to Step 4)
6) Check to see if the door is unlocked.
    a) If locked, then create a SystemMessage Event communicating the locked status and add to the

	EventQueue. End Use Case.
	7) Change the Door to an Opened State.
	8) The Action is discarded
Exit Condition:	9) The door is open or an error message is displayed

Use Case:	**Close a Door**
Actor:	Player
Goal:	The Player desires to close a door in the game
Flow of Events:	1) The Player right clicks on the open door in the game
	2) A CloseDoor Action is requested from the ActionFactory
	3) The new Action connected with the Player and the door and added to the Action Queue
	4) …Wait until Action is popped from the Action Queue
	5) If Creature is out of range of the door, then treat as a movement Action and go to Step 4)
	6) Change the Door to a closed state.
	7) The Action is discarded
Exit Condition:	8) The door is closed or an error message is displayed

Use Case:	**Unlock a Door**
Actor:	Player
Goal:	The Player desires to unlock a door in the game
Flow of Events:	1) The Player right clicks on the closed door in the game

	2) An UnlockDoor Action is requested from the ActionFactory
	3) The new Action connected with the Player and the door and added to the Action Queue
	4) …Wait until Action is popped from the Action Queue
	5) If Creature is out of range of the door, then treat as a movement Action and go to Step 4)
	6) Check to see if the door is unlocked.
	a) If unlocked, treat as an OpenDoor Use Case. End Use Case.
	7) Check to see if the Creature has the right Item in their Inventory to unlock the door
	a) If they do not, then create a SystemMessage Event communicating the lack of a proper unlocking Item. Send Event to EventQueue and End Use Case
	8) Change the Door to an unlocked State.
	9) The Action is discarded
Exit Condition:	10) The door is unlocked or an error message is displayed

Use Case:	**Lock a Door**
Actor:	Player
Goal:	The Player desires to lock a door in the game
Precondition:	The Variable Window is displaying the Inventory screen
Flow of Events:	1) The Player left clicks the Item to be used to Lock the Door

2) The Player selects the Use Item button
3) The Player left clicks on the closed door in the game
4) A LockDoor Action is requested from the ActionFactory
5) The new Action connected with the Player and the door and added to the Action Queue
6) …Wait until Action is popped from the Action Queue
7) If Creature is out of range of the door, then treat as a movement Action and go to Step 4)
8) Check to see if the door is already locked.
    b) If locked, create a SystemMessage Event communicating the already locked status. Add Event to EventQueue. End Use Case.
9) Check to see if the Item is the right Item to lock the door
    a) If it is not, then create a SystemMessage Event communicating the lack of a proper locking Item. Send Event to EventQueue and End Use Case
10) Change the Door to a Locked State.
11) The Action is discarded

Exit Condition: 12) The door is locked or an error message is displayed

Use Case:	**Attack a Creature**
Actor:	Player
Goal:	Player wishes to attack a Creature

Flow of Events:

1) Player right clicks a hostile Creature
2) An AttackAction is requested from the ActionFactory
3) The new Action is added the Player's Creature and the target Creature.
4) The new Action is added to the ActionQueue
5) …Wait until Action is popped from the Action Queue
6) If Player's Creature is too far away from the target Creature to attack, then treat as a MovementAction and return to Step 5)
7) Check to see if the Player's Attack is successful
    a) If successful, then create a new AttackEvent with the Action and add to the Event Queue. Keep Action in ActionQueue
    b) If unsuccessful, create a SystemMessage Event containing the failure. Add Event to ActionQueue. Return to Step 4). Keep Action in ActionQueue.
8) ...Wait until AttackEvent is popped from the Event Queue
9) Determine amount of damage applied to target Creature.
10) Create a SystemMessage Event containing the success of the AttackAction and the damage applied. Add Event to EventQueue.

	11) Determine if damage is enough to kill Creature
	a) If not, Go back to Step 5
	b) If so, Create a DeathEvent with the Creature and Add to EventQueue.
	12) …Wait again for Death Event to be selected from the Event Queue
	13) Create a SystemMessage Event communicating Death of Creature. Send Event to EventQueue
	14) Determine outcome & gains from Death (experience)
	15) Kill any Actions with the initiator being the dead Creature
	16) Kill any Actions targeting the dead Creature
	17) If the dead Creature is the Player's Creature, then create EndGameEvent. Send EndGameEvent to EventQueue.
Exit Condition:	18) Player is informed of target Creature's death

B.3 Developer Use Cases

Use Case:	**Develop a new Creature**
Actor:	Developer
Goal:	Developer wishes to add a new Creature
Flow of Events:	1) Developer fills out the statistics per the documentation for a CreatureTemplate

	2) Developer links the CreatureTemplate BodyParts to a Graphic Template
	3) Developer adds a link in the Object Template file to the newly created creature template file
	4) Developer adds a link in the Map Configuration file to the newly created creature template file
Exit Condition:	5) A new Creature has been added to the game

Use Case:	**Develop a new Item**
Actor:	Developer
Goal:	Developer wishes to add a new Item
Flow of Events:	1) Developer fills out the statistics per the documentation for an ItemTemplate
	2) Developer links the new ItemTemplate BodyParts to a Graphic Template
	3) Developer adds a link in the Object Template file to the newly created item template file
	4) Developer adds a link in the Map Configuration file to the newly created item template file
Exit Condition:	5) A new Item has been added to the game

Use Case:	**Develop a new Background**
Actor:	Developer

Goal:	Developer wishes to add a new Background
Flow of Events:	1) Developer fills out the statistics per the documentation for a BackgroundTemplate
	2) Developer links the BackgroundTemplate BodyParts to a Graphic Template
	3) Developer adds a link in the Map Configuration file to the newly created background template file
Exit Condition:	4) A new Background has been added to the game

Use Case:	**Develop new Graphic Template**
Actor:	Developer
Goal:	Developer wishes to add a new Graphic Template
Flow of Events:	1) Developer translates desired shape of item into a file containing triangles and quadrangles per documentation
	2) Developer links the newly created Graphic Template to the Graphic Configuration file.
Exit Condition:	3) A new Graphic Template has been added to the game

Appendix C Class Descriptions

As stated earlier, the game is programmed with the Model-View-Controller architecture. A *ModelManager* class encapsulates the model, a *ViewManager* class encapsulates the view, and a *ControlManager* class encapsulates the controller. This is only a partial list. Please review the documentation that goes along with this document for a full view of the classes and their functions.

C.1 Model

The model holds all the data elements of the game. In reality, this covers the following classes: *things, backgrounds, items, creatures, thinglinkedlist, section, sectionmanager, map, quest, questmanager, trigger, triggermanager,* and *modelmanager.*

Things, Backgrounds, Items, Creatures

Things, and its subclasses *background*, *item*, and *creatures*, is encapsulated by classes with their respective names. To ease programming constraints, the three subclasses inherit from the *thing* master class. The majority of elements are the same throughout the parent and its subclasses. Each *thing* has a link to its graphical display (the *graphic template*), and can clone itself. Information is kept on specifics of its graphical display (the *body parts*) along with any custom colors the *thing* possesses. Each *thing* has a base movement, a *vertex* for its location on the *map*, and its orientation on the x-y-z axis. Each *thing* also has a set of boolean values that designate whether it can create *actions* or block movement. Additionally, each *thing* has an identification number which uniquely identifies it. Also, the *thing* has a type-identifier which distinguishes to which subclass it belongs.

Creatures have some additional implementation notes. *Actions* that are not to begin immediately are stored inside the *creature*'s personal *actionqueue*. This is used during the artificial intelligence phase of the game.

Thing Linked List

Groups of *things* are stored in *ThingLinkedLists*. As the name suggests, the storage method for this class are in a linked list. The linked list is responsible for adding and removing *things* from its data pool, and providing *things* on request. The *thinglinkedlist* is also responsible for populating the *printobjectlinkedlist* and *collisionlinkedlist*.

Sections

Sections are a method to subdivide a *map*. Its initial purpose is during random map generation, however it is used to help ease collision detection, artificial intelligence, and work with the model during random *creature* and *item* generation. A *section* possesses a width, height as well as an x-y coordinate representing the bottom left hand corner of the *section.* *Sections* are separated into rooms and corridors. The delineation is used primarily during the random map generation process. The value is stored in a boolean variable. *Sections* contain a *thinglinkedlist* with links to every *background* within its area.

Section Manager

The *SectionManager* stores all *sections* on a given *map*. The *sectionmanager* stores these *sections* in a linked list fashion. The *sectionmanager* is a key player in the random map generation phase. The *sectionmanager* contains a counter for the number of rooms and corridors it currently has in its possession.

Map

The *Map* is an important part of the game engine. This contains three *thinglinkedlists*, one for each of the *thing* subclasses. In the case of *backgrounds*, this results in a redundant link held by the *map* and the *section*. The map is responsible for ensuring that a complete *printobjectlinkedlist* and *collisionlinkedlist* are built for the controller. The *map* is the other key player during random map generation. It achieves this by keeping several *backgrounds* with its collection. These *backgrounds* are cloned to create the floors, walls, doors and other various *background* elements that exist in a *map*. Finally, there is a connection point stored by the *map* which allows it to be linked to other *maps*.

Quest

The *Quest* is an import part of developing the storyline for the game. From a game play perspective, this section is one of the keys to keeping the player interested in the game. The *quests* themselves are fairly straight forward. They contains a unique identifier, a boolean status, a type identifier to identify the manner the quest is completed, and several backup fields used for different *quest* types. An example of some of the back up field is a *thing* field which designates a specific target for the *quest* to be completed.

Additionally there may be a string field to help identify the *quest*. It should be noted that these *quests* are specifically for the player's creature.

Quest Manager

The *QuestManager* class holds all the current and completed *quests* that the player's creature has been assigned. The primary role of this class is to hold new *quests* and retrieve the states of *quests* in its possession.

Trigger

Trigger represents conditional *actions* and *events* that are executed by the system.

Trigger Manager

The *TriggerManager* stores all the current *triggers* for a given *map*.

Model Manager

The *ModelManager* is the housing for the *triggermanager*, *questmanager*, and *map* for the game. All together, the *modelmanager* encompasses all the elements for the model in the MVC architecture. Additionally, the *modelmanager* maintains a special link to the player's creature. The *modelmanager*'s main functions are generating new *maps*, providing the *background* templates to build new *maps*, adding new *things*, and retrieving *things* on request. Additionally, the *modelmanager* passes along any requests to build *printobjects* or *collisionlists* to the *map* in its possession. Basically, any request for a specific *thing* has to pass through the *modelmanager*.

C.2 View

The view holds all the visual elements of the game. In reality, this covers the following classes: *vertex, textcoord, shape, triangle, quad, texturetriangle, texturequad, bodypart, animation, animationstep, graphictemplate, graphictemplatelinkedlist, backgroundwindow, openglwindow, consoleoutputwindow, variableoutputwindow, camera, printobject, printobjectlinkedlist, collisionmodel, collisionlinkedlist,* and the *viewmanager.*

Vertex

The *Vertex* class is the most basic of all the graphics classes. This class encapsulates a three-dimensional point. The only structural data elements of this class are three floating point numbers representing the x, y, and z of the *vertex*. The *vertex* coordinates are all relative to the base of the *bodypart*.

TextCoord

The *TextCoord* class is the most basic of all the graphics classes that use textures. As with the *vertex* class, this class encapsulates a point. Texture coordinates are on a 2 dimensional bitmap, so the only data elements of this class are an x and y floating point number.

Shapes

Shape represents the core of the basic graphic display of every *thing* in the game. *Shape* has 2 inherited classes: *discreteshape* and *quadraticshape*.

Discrete Shape, Triangles, Quads, Texture Triangle, Texture Quad

DiscreteShape, and its subclasses *triangle*, *quad*, *texturetriangle* and *texturequad*, make up the basic graphic display of every *thing* in the game. As with the *thing* class, *shape* is the parent class and the four subclasses inherit from *shape*. *DiscreteShape* exists as an abstract class, and therefore no actual *shapes* are created. *DiscreteShape* possesses four data elements and one function. The first is sides, which is the number of sides for the *shape*. By definition, *triangles* have three sides and *quads* have four sides. The second data type is an array of *vertices*. The array length is the same size as sides. The third data type is an option data type. It is an array of *textcoords*, with it having the same number of elements as size. Finally, there is a boolean data element labeled hasTemplate whose value determines whether or not the shape has any textcoords. Everything but the values of the *vertices* and *textcoords* are determined by the object class. The hasTemplate value is only true for the *texturetriangle* and *texturequad.* There exists only one function for *shape* and all it's subclasses, named draw(). When draw is invoked, the system uses OpenGL libraries to write the current shape, and if applicable the texture, into the graphics buffer.

QuadraticShape, QuadraticCylinder, QuadraticDisk, QuadraticSphere

QuadraticShape, and its subclasses *quadraticcylinder, quadraticdisk and quadraticsphere,* encapsulate the basic graphic display for every quadratic shape in the game. Again, there exists only one function named draw(). When draw is invoked, the system uses OpenGL libraries to write the current shape, and if applicable the texture, into the graphics buffer.

Body Part

BodyPart represents an atomic element of the *thing*. The *bodypart* is made up of one *graphictemplate*. The *bodypart's* primary function is the draw function. When this function is invoked, the current *bodypart* is moved as per the *animationstep* and the *graphictemplate* is drawn.

Animation Step

AnimationStep encapsulates one key-frame in a *bodypart*'s *animation*. *AnimationStep* contains a list of *bodyparts* and where they should begin their drawing sequence.

Animation

Groups of *animationsteps* are stored in the class *animation*. This class uses a type identifier to determine which real life action this animation mimics.

Collision Model

The *CollisionModel* is used by the physics model to detect collisions within the game. The primary role of the *collisionmodel* class is to detect whether or not it has collided with another *collisionmodel*. The *collisionmodel* is an over simplification of the space occupied by the *thing*, and used to limit the number of geometric calculations required by a system to detect collisions. The *collisionmodel* is a bounding box and exists in one of three different forms: plane (2-dimensional), sphere (simple 3-dimensional) and box (complex 3-dimensional). The *collisionmodel* maintains which of the different forms it exists as. Depending on which form, the *collisionmodel* has a radius, length, width or height. Additionally, the *collisionmodel* maintains a simplified length used to quickly judge whether a collision is a reasonable possibility. This is stored as the object's longest edge, though in fact it is just a sphere representation of the *collisionmodel* that fully encompasses its length, width and height.

Graphic Template

GraphicTemplate encapsulates all information necessary to display a *bodypart* in a three dimensional fashion. The *graphictemplate* is made up of a series *shapes*. There is no limit to the number of *shapes* existing within the *graphictemplate*. The *graphictemplate* consists of 3 data elements, an array of *shapes*, the total number of shapes, and a boolean value storing whether or not it has a texture. *GraphicTemplates* have a name that is referenced by the *ItemTemplate*, *CreatureTemplate,* and configuration files. The *graphictemplate's* primary function is the draw function. When invoked, the system parses through each of the *shapes* stored and invokes their draw function.

Graphic Template Linked List

GraphicTemplateLinkedList stores all the *graphictemplates* on a given map. Its role is to add, remove, and return requested *graphictemplates*. Additionally, the *graphictemplatelinkedlist* stores the longest of all the *collisionmodel* edges. Using this longest edge can feasibly reduce the total number of possible collisions that need to be tested. As the name states, the *graphictemplates* are stored in a linked list fashion.

Background Window

BackgroundWindow encapsulates the main window that the GUI resides upon. The *backgroundwindow* represents the base Win32 API element for the game engine. All future GUI elements are written on top of this object. The *backgroundwindow* maintains a connection to the windows element in the system's memory (also known as a window handle).

Camera

The *Camera* is a display tool used by the view. The *camera* represents the location of the "eye" that is viewing the OpenGL world. The *camera* stores a series of 8 floating point numbers. Three floating point numbers represent where the *camera* sits, three floating point numbers represent what direction the *camera* is facing, and two floating point numbers capture the width and height of the viewing area.

Camera Display Node

The *CameraDisplayNode* is a median class that holds objects that need to be displayed in the *openglwindow*. In addition to the *thing* to be displayed, the *cameradisplaynode* stores a relative location for the object. This relative location is in camera coordinates. This simplifies the eventual display of the *things*. The *cameradisplaynode* also contains a boolean value which says whether or not an adjacent *cameradisplaynode* contains the same texture as the current *cameradisplaynode.* This value is set by the *cameradisplaylist*.

Camera Display List

CameraDisplayList contains all the *cameradisplaynode* that are to be displayed in a given turn. As the name suggests, the *cameradisplaynode* are stored in a linked list fashion. The list is responsible for adding and removing all *cameradisplaynode* in its possession. Additionally, before printing, the *cameradisplaylist* organizes the all *cameradisplaynodes* so that *things* with the same *graphictemplate* lie in adjacent nodes.

OpenGL Window

The *OpenGLWindow* class encapsulates the child window that displays the model. As with the other children windows, the *openglwindow* is displayed on top of the *backgroundwindow*. The *openglwindow* also contains a handle to the window representation in memory. The *openglwindow* possesses the camera, as well as some OpenGL specific variables. The primary role of the *openglwindow* class is to draw the OpenGL scene to the screen. The function takes a *printobjectlinkedlist* and parses through the list displaying each object.

Console Output Window

The *ConsoleOutputWindow* class encapsulates the child window that displays the system log. As with other children windows, the *consoleoutputwindow* is displayed on top of the *backgroundwindow*. This window also contains a handle to the window representation in memory. The *consoleoutputwindow* maintains a system log as well as a specific font. As new messages are sent to it, it parses the data and separates it out into lines. Finally, it displays these lines to the screen, scrolling the vertical scrollbar as necessary. The primary function is the print text function, which writes the text to the log.

Screen

The *screen* is a virtual class used by the *variabledisplaywindow*. The *screen* houses all the elements need to display itself. The *screen*'s primary function is to receive commands form the user interface, and display various elements as selected by the user. The *screen* is broken down into three subclasses, and can be extended in the future as needed: *ThingSelectionScreen*, *InventoryScreen*, and *MainPlayerScreen*.

Variable Display Window

The *variabledisplaywindow* is a subset of the OpenGL window, and is used to govern what is displayed in the variable display. The Variable display is made up into a series of different screens. Each screen has a different purpose. The *variabledisplaywindow* maintains the current screen, and passes messages from the *ViewManager* to the various screens.

Collision Linked List

The *CollisionLinkedList* stores all potential collisions between a base *thing* and all other *things* held by the *map*. These collisions are examined in closer detail, and if any collisions occur then they are added to the *eventqueue*.

View Manager

The *ViewManager* class encapsulates the View. The *viewmanager* holds the *graphictemplatelinkedlist*, *backgroundwindow, consoleoutputwindow, openglwindow,* and *variabledisplaywindow* classes. Additionally, it contains a 256 array of booleans that represent each of the possible keys. If there is a true value for the corresponding ASCII value of a key then that the key is being pressed. These keys are tested periodically during the game. The *viewmanager* also acts as a conduit for display text to the log, adding, removing and retrieving *graphictemplates*, and as the initial receiving point of messages from the Windows operating system to the game engine.

C.3 Controller

The controller maintains the system elements of the game. In reality, this covers the following classes: *idgenerator, creaturetemplate, itemtemplate, thingtemplatelinkedlist, action, actionqueue, event, eventqueue,* and the *controllmanager*.

ID Generator

The *IDGenerator* is a special part of the controller. This class's only responsibility is to ensure that request identifiers for *things* are unique. Therefore, the class only contains a counter representing the last value submitted, and a function for getting new values.

Thing Templates, Creature Templates, Item Templates, BackgroundTemplate

CreatureTemplate, *ItemTemplate*, and *BackgroundTemplate* classes are using during random generation. These classes attempt to represent a truly generalized *creature* or *item*, or *background*. Their only role is to spawn new *things*. They work by having base attributes and a range for each attribute.

Thing Template Linked Lists, Background Template Linked List

The *ThingTemplateLinkedList* and it's subclass, *BackgroundTemplateLinkedList* hold all the templates that exist for a given map

Action

Action is a virtual class to represent all future actions that are currently being done by a *creature*. *Action* is an inherited class, with several subclasses.

Action Queue

ActionQueue holds all the currently running *Actions*.

Event

Event is a virtual class to represent all actions that are happening this second. *Event* is an inherited class with several subclasses.

Event Queue

EventQueue holds all the currently active *Events*.

Control Manager

ControlManager is the control portion of the MVC architecture

Appendix D Files

File Name	License / Notes
GoldArrow.bmp	Provided as a free sample from the Jupiterimages Corporation (http://www.mediabuilder.com/webl3.html)
RedFluid.bmp	This is actually a glass of red juice that I took and then cropped out the edges of the glass.
SideBorder.bmp	Unknown. They were released under a free, not to be used for commercial license, but the vendor can no longer be found.
Tinystr.h & Tinystr.cpp	Provided courtesy of Yves Berquin under the GNU license (www.sourceforge.net/projects/tinyxml).
Tinyxml.h & tinyxml.cpp	See tinystr.h above
Tinyxmlerror.cpp	See tinystr.h above
Tinyxmlparser.cpp	See tinystr.h above
Door1GT.bmp	Provided courtesy of the Brisbane, Australia Mayors office, http://joe_kelso.tripod.com/brisbane/kelsolandmanor.htm. Government office, so property of the public.
Floor1GT.bmp	Provided courtesy of kittyispretty69 for free use http://www.modthesims2.com/showthread.php?t=178728
Wall1GT.bmp	Unknown. They were released under a free, not to be used for commercial license, but the vendor can no longer be found.
Hobgoblin MD2	Hobgoblin Md2 file and bmp provided from DigiBen from www.gametutorials.com and cannot be used for any commercial reasons
Hobgoblin1GT.bmp	See above note
Crate1GT.bmp	Provided courtesy of Jeff Molofee from http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=07 under a limited use noncommercial license.

What are the important directories and files?

Directory	
Documentaton\	This directory holds all DOxygen html pages
Executable\	For those who do not want to compile the program, this contains the executable and all the necessary data files
PhantasmEngine\	This is the game code, and includes all data files
PhantasmEngine\Data\	This folder contains the 3 xml configuration files

How do I compile the program?

If you have Visual Studios – load the .dsw file in the PhantasmEngine directory. This will load all the necessary information into Visual Studios. From here, select the "Build" menu, and then "Rebuild All". This will compile all the necessary files. You can then run the program by pressing the red exclamation point.

If you do not have Visual Studios – I have exported a dependencies file (PhantasmEngine.dep) and a make file (PhantasmEngine.mak). I have not tested these files, nor do I know how to use them. Not only this, but I'm not positive that the library and header files I've referenced in my code can be compiled by any compiler other than Microsoft's compiler. Good luck to you if you have to go this route.

How do I recreate the DOxygen documentation?

I have used no special settings or build requirements for DOxygen. Just set up to output in html (or whatever output format you would like to use), use the default settings, point it at the code directory and run.

How do I run the program?

There are two ways of doing this. 1) If you want to play the game engine immediately, then run the PhantasmEngine.exe in the executable directory. 2) If you are using Visual Studios, you can load the code and press Control-F5 or click the red exclamation point. This will automatically run the code. On a side note, the location of the executable built from Visual Studios can be confusing. Visual Studios will put the executable in the debug or release directories (depending on your active configuration) when you compile the file. The game will execute fine when run from the Visual Studios window. However, to run the code outside of Visual Studios, you have to move the .exe file up to the base directory. Basically, the Data\ directory has to be the next directory down from the executable in order for the game engine to function.