Development of a Simple 3D Game Engine
Table of Contents

1 ABSTRACT .......................................................................................................................... 5
2 INTRODUCTION .................................................................................................................. 6
  2.1 AIMS ............................................................................................................................... 6
  2.2 OBJECTIVES .................................................................................................................. 6
3 PROJECT SCOPE ............................................................................................................ 7
  3.1 MODELS AND MATERIALS ......................................................................................... 7
  3.2 ANIMATION SYSTEM .................................................................................................. 7
  3.3 MATH, PHYSICS AND COLLISION DETECTION ......................................................... 7
  3.4 RENDERING SYSTEM ................................................................................................ 8
  3.5 MODEL EDITOR .......................................................................................................... 8
  3.6 ENGINE VIEWER ....................................................................................................... 8
  3.7 GAMEPLAY DEMO .................................................................................................... 8
4 WORK PLAN .................................................................................................................... 9
  4.1 SEMESTER 1 .............................................................................................................. 9
  4.2 SEMESTER 2 ............................................................................................................ 10
5 LITERATURE REVIEW .................................................................................................... 11
  5.1 INTRODUCTION ........................................................................................................ 11
  5.2 GAME ENGINE ARCHITECTURE ................................................................................. 11
  5.3 3RD PARTY SDKS ...................................................................................................... 13
    5.3.1 Simple DirectMedia Layer (SDL) .......................................................................... 13
    5.3.2 OpenGL .................................................................................................................. 13
    5.3.3 LIB3DS .................................................................................................................. 13
  5.4 PLATFORM INDEPENDENCE LAYER ........................................................................ 14
    5.4.1 Windowing System ............................................................................................. 14
    5.4.2 Rendering System ................................................................................................. 14
      5.4.2.1 Vertex Arrays .................................................................................................. 15
      5.4.2.2 Vertex Buffer Objects (VBOs) ......................................................................... 15
    5.4.3 Input System ........................................................................................................ 15
  5.5 CORE SYSTEM .......................................................................................................... 15
    5.5.1 Math Library ........................................................................................................ 15
    5.5.2 C Math and Standard Template Library (STL) ..................................................... 17
  5.6 RESOURCE MANAGER .............................................................................................. 17
    5.6.1 OBJ File Type ....................................................................................................... 17
    5.6.2 MTL File Type ..................................................................................................... 18
    5.6.3 Texture Loader ..................................................................................................... 19
    5.6.4 Animation System ............................................................................................... 19
      5.6.4.1 Skeletal Animation Tree Structure .................................................................. 19
      5.6.4.2 Morph Targets ............................................................................................... 19
      5.6.4.3 Linear Interpolation ....................................................................................... 20
  5.7 PHYSICS SYSTEM ...................................................................................................... 21
    5.7.1 Bounding Spheres ............................................................................................... 21
    5.7.2 Creating the Collision Sphere ............................................................................. 21
    5.7.3 Sphere - Sphere Collision Detection .................................................................. 21
    5.7.4 Sphere - Plane Collision Detection ................................................................... 22
6 REQUIREMENTS ............................................................................................................... 23
  6.1 MODELS AND MATERIALS ....................................................................................... 23
  6.2 ANIMATION SYSTEM ................................................................................................ 23
  6.3 MATH, PHYSICS AND COLLISION DETECTION ...................................................... 24
  6.4 RENDERING SYSTEM ............................................................................................... 24
  6.5 MODEL EDITOR ......................................................................................................... 24
  6.6 MODEL VIEWER ...................................................................................................... 25
  6.7 GAMEPLAY DEMO ................................................................................................... 25
Figure List

Figure 1: Proposed Work Plan - Semester 1 ................................................................. 10
Figure 2: Proposed Work Plan - Semester 2 ............................................................... 10
Figure 3: Simple 3D Game Engine Architecture, based on Gregory, J, (2009), Game Engine Architecture, page 29 ............................................................... 12
Figure 4: SDL Libraries and Versions Being Used ...................................................... 13
Figure 5: Example C Preprocessor Macro Example .................................................... 14
Figure 6: Formula for Magnitude of a Vector ............................................................. 15
Figure 7: Formula for Dot Product of Two Vectors .................................................... 16
Figure 8: Calculating the Cross Product of Two Vectors ............................................. 16
Figure 9: Calculating the Surface Normal of a Polygon .............................................. 16
Figure 10: Normalizing a Vector .................................................................................. 16
Figure 11: OBJ File Information ................................................................................ 17
Figure 12: MTL File Information ................................................................................ 18
Figure 13: Tree Structure of Animation Structure ...................................................... 19
Figure 14: High Level Morph Target Algorithm .......................................................... 20
Figure 15: Formula for Linear Interpolation ................................................................ 20
Figure 16: Linear Interpolation Example .................................................................... 20
Figure 17: Algorithm for Creating a Bounding Sphere from Model Data ................. 21
Figure 18: Algorithm for Sphere - Sphere Collision Detection ................................ 21
Figure 19: Diagram Showing Sphere - Sphere Collision Detection .............................. 22
Figure 20: Algorithm for Sphere - Plane Collision Detection ...................................... 22
Figure 21: Diagram Showing Sphere - Plane Collision Detection .............................. 22
Figure 22: System Class Diagram ............................................................................. 26
Figure 23: Model3D, Mesh3D and MeshObject Relationship ...................................... 27
Figure 24: Model3D Make-Up ................................................................................... 28
Figure 25: OBJX File Info ......................................................................................... 30
Figure 26: MTLX File Info ......................................................................................... 30
Figure 27: SDL Viewer Keyboard Controls ............................................................... 31
Figure 28: Gameplay Demo Game Pad Controls .......................................................... 32
Figure 29: Model Point Vertex Structure ................................................................. 33
Figure 30: 3D Studio Max OBJ Export Settings ......................................................... 34
Figure 31: Blender OBJ Export Settings .................................................................... 34
Figure 32: OBJ File Parser ....................................................................................... 34
Figure 33: Material Data Structure .......................................................................... 35
Figure 34: MTL File Parser ...................................................................................... 35
Figure 35: Class Structure for Holding Animation Data ............................................. 36
Figure 36: Function to Update an Animation ............................................................. 36
Figure 37: Data Structures for 2D Vectors and 3D Vectors ........................................ 37
Figure 38: Bounding Sphere Data Structure.............................................................. 37
Figure 39: Function to Calculate Magnitude of a 3D Vector ...................................... 37
Figure 40: Function to Normalize a 3D Vector ........................................................... 37
Figure 41: Function to Calculate Cross Product of two 3D Vectors ......................... 37
Figure 42: Function to Calculate Dot Product of two 3D Vectors .............................. 38
Figure 43: Model3D Class function for checking for a Collision with a Second Model3D Object ............................................................. 38
Figure 44: - Model3D with Bounding Sphere Calculated and Rendered in Model Editor ............................................................................................. 38
Figure 45: Triangle Structure and Function for Calculating Triangle Surface Normal ............................................................. 39
Figure 46: Lighting using Surface Normals (Left) and Per Vertex Normals (Right) ........ 39
Figure 47: Identified Problem with Bounding Sphere - Collision Mesh collision detection (Red Polygons have been identified as being hit) ................................................................. 40
Figure 48: Textured OJB Model being displayed using Engine Rendering System using Lighting ............................................................. 40
Figure 49: Final Editor User Interface ...................................................................... 41
Figure 50: Function to Save Material's Diffuse Colour to MeshObject Vertexes ........... 42
Figure 51: Model Rendered Using Lighting (Left) and Model Rendered Without Lighting (Right) ............................................................. 43
Figure 52: Changing the Target OS in XCode ............................................................. 44
Appendix List

Appendix 1: 3D Cube with Corresponding OBJ and MTL Files .................................................53
Appendix 2: Diagram Showing Animation Structure ..................................................................54
Appendix 3: Entity Relationship Diagram showing Relationship between Model3D, Mesh3D and MeshObjects ...........................................................................................................55
Appendix 4: OBJ and MTL Loading Process Diagram ...............................................................56
Appendix 5: Engine Architecture ...............................................................................................57
Appendix 6: Initial Engine Editor Interface Design ..................................................................58
Appendix 7: Initial Engine Editor Functionality .........................................................................59
Appendix 8: Final Engine Editor Interface and Functionality ..................................................60
Appendix 9: ResourceManager and MeshObject Relationship ...............................................61
Appendix 10: Model3D Structure ...............................................................................................62
1 Abstract

The primary objective of this project was to develop a ‘Simple 3D Game Engine’. It was a secondary objective that this engine could be used to develop games for a contemporary portable device running Apple’s iOS Operating System e.g. iPhone, iPad, iPod Touch. Both of these objectives were met.

A Game Engine can be defined as a framework that provides for; the loading and display of 3D models, the animation of 3D models, the loading and display of 2D images, collision detection, and user I/O.

This project delivered as software artifacts; a 3D Model Editor for, a 3D Model Viewer for models exported from the Editor and an Engine framework.

The Model Editor allows the user to create hierarchical, animated 3D models. Additionally, it outputs a custom file type that was developed specifically for this project by the author (.OBJX).

The Model Viewer loads the custom file format and gives a representation of how the developed Engine will render a 3D model.

The Engine framework is a development environment, within which the developer can build a 3D game environment. It also supports models stored in the custom file type and handles physical computations such as collision detection.

An unexpected, but powerful, output of this project is the OBJX file format. Various issues, such as Vendors adding proprietary code to ‘standardized’ 3D file types meant that development of an engine was made unnecessarily difficult. The OBJX file format was developed to solve these problems. Aside from delivering a simple portable file format for 3D models the file type also delivered some other benefits; the processing involved in animating OBJX models is also much less complex than would be needed for a rigged 3D model with skeletal information. Also, OBJX required less processing power making it a file format an option for developing to mobile devices.

---

1 Prior to Summer 2010, the iOS Operating System was called iPhone OS
2 Introduction

This document details the Scope, Research, Design, Implementation and Testing of a Simple 3D Game Engine.

The main goal of this project is to develop a Game Engine that a developer could use to make a simple 3D game targeted at an iOS device (iPhone / iPad / iPod Touch).

This project will produce the following Software Artifacts:

- A 3D Model Editor for creating hierarchical, animated 3D models, manipulating material properties, exporting this model data as a custom file format and generally testing the Engine’s functionality,
- A 3D Model Viewer for testing how models exported in this proprietary form of the Model Editor will be viewed on an iOS device. A second implementation of this Model Viewer Application will be developed that runs for on Windows and Mac OS X environments
- The Engine framework, which will serve as a development environment for a developer that may wish to develop a game using this Engine

2.1 Aims

- Develop an understanding of the core mechanics of a 3D Game Engine
- Further develop knowledge in 3D Math, Physics, C++ and OpenGL
- Develop a better understanding of more efficient and advanced rendering techniques such as Vertex Buffer Objects (VBO), Vertex Arrays, Draw Arrays and Index Arrays
- Develop an understanding of how 3D model data is stored on disk along with their material and animation data
- Develop an understanding of techniques used to effectively break the game objects into smaller sectors to optimize physics testing and rendering
- Gain Experience in developing for iOS mobile devices

2.2 Objectives

- Implement a 3D Model and Material Loader
- Develop a data structure for holding Hierarchical Models
- Create a Simple Animation Structure for hierarchical models
- Develop an Editor that will allow the creation, manipulation and animation of these hierarchical 3D models
- Develop a proprietary file type for saving these 3D models
- Develop a Model Viewer that will allow for the viewing of models saved in a proprietary file format
- Look at most efficient methods for drawing scenery and carrying out physics testing within a 3D virtual world
- Look at most efficient methods of detecting collisions between Bounding Spheres and Collision Meshes
- Develop simple gameplay demo to show the Engine working with a player character moving around and interacting with objects within game world
3 Project Scope

The task of developing a Game Engine, even a relatively simple one, has the potential to become extremely complex and go beyond the scope of what would be expected of a college project. To avoid this happening, this section clearly details parameters of the project. It will be used as a reference to ensure that the scope of the project is maintained and remains manageable.

This Simple 3D Game Engine will be written in C++ using OpenGL ES 1.1. Its main target platform will be iOS devices (iPhone, iPad, iPod Touch), however the Engine will also be able to be used on Windows and Mac OS X Systems. Its functionality will be limited to:

- Loading and Displaying 2D Images
- Loading and Displaying 3D Models
- Manipulation of Model Material Properties
- Handling Simplistic 3D Model Animations
- A Math Library containing core math functions and structures
- Ability to Handle Input from iOS Device Touch Screen and Game Pad
- Handling Some Basic Collision Detection – Sphere-Sphere Collision Detection, Sphere-Box Collision Detection, Sphere-Collision Mesh Collision Detection
- Handling Some Basic Physics – Gravity, Sphere Collision Response

This project will be divided into seven key areas – Models and Materials, Animations Structure, Math, Physics and Collision Detection, Model Editor, Model Viewer and a Gameplay Demo.

3.1 Models and Materials

This Engine will support the loading of Models and their corresponding Material Properties. Models supported by the engine can be stored in either .obj or .3ds file formats. Material properties for OBJ Models will be read from their corresponding .mtl file. The Engine will contain its own code to load OBJ and MTL files, but the Open Source Lib3DS library will be used to load 3DS files.

3.2 Animation System

The Engine will be able to perform simple model animations. These animations will be similar to what may be seen in older generation games on consoles such as the original Sony Playstation and Nintendo 64. Animations will be made up of rotations about a model’s origin on the X, Y and Z-axis. It is felt that these simple animations will work well on iOS devices due to their limited screen size and processing power.

The Engine will also be able to perform Morph Animations using Morph Targets. Morph Targets are used to give the illusion of one model morphing into another. The engine will be able to morph between two models, provided each model has the same number of vertices.

3.3 Math, Physics and Collision Detection

A Math Library class will be written to hold Mathematical Structures and Functions needed by the Engine. This will include structures for 2D and 3D Vectors and a 3D Vertex Structure to hold data about points in a 3D model. Functions will include methods to calculate Surface Normals, Dot Products and Cross Products.

The Engine will be able to detect and handle some very basic collision detection. This collision detection will consist of:

- Sphere - Sphere Collision Detection
- Sphere – Plane Collision Detection
- Sphere - Collision Mesh Collision Detection
3.4 Rendering System
The Rendering System will do all of the drawing to the screen. It will only use calls available in OpenGL ES 1.1. It will also be designed independently of the rest of the Engine so that it may easily be swapped out so a new rendering system using a different technology or library such as Direct X or OpenGL ES 2.0 could be replace it with minimal problems.

3.5 Model Editor
The Model Editor will allow for the loading and displaying of 3D Models and manipulation of their material properties. It will also allow for the linking together of smaller models to make hierarchical models, which can then be animated. The Editor will be able to export all this data to a file on disc in a proprietary file format that may then used in a game developed using the Engine. It will also serve as a tool for developers to test the functionality of the Engine.

3.6 Engine Viewer
The Engine Viewer will display data files that are exported by the Editor. This will allow a developer using the Engine to see how their hierarchical, animated modes will appear in a game. There will be two different implementations of the Viewer developed. The first will be targeted at Windows and Mac OS X Development Environments. It will be developed primarily using SDL to handle input, texture loading and setting up windows. A second Viewer will be developed to allow for viewing on iOS devices. It will take advantage of Apple’s iOS libraries Cocoa Touch, Quartz and UIKit for setting up a window, texture loading and handling input.

3.7 Gameplay Demo
The Gameplay Demo will be used to demonstrate the functionality of the Engine developed. It will consist of a Model, being controlled by a user, moving across a simple terrain. The Model’s current animations will change as the user does different actions, *i.e.* Idle, Running, Jumping. The Demo will also consist of objects that the Player will not be able to walkthrough to demonstrate the collision detection functionality in the engine.
4 Work Plan

This project will be developed over the next two college semesters. This section details the expected work plan for the project.

4.1 Semester 1

<table>
<thead>
<tr>
<th>Week</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 1</td>
<td>• Prepare list of possible project ideas</td>
</tr>
<tr>
<td>Week 2</td>
<td>• Refine list of project ideas and do further research in these areas</td>
</tr>
<tr>
<td>Week 3</td>
<td>• Decide on project title and prepare project proposal document</td>
</tr>
<tr>
<td></td>
<td>• Identify some key texts that will be of serve as a point of reference for project</td>
</tr>
<tr>
<td>Week 4</td>
<td>• Begin reading text and make notes on book Game Engine Architecture</td>
</tr>
<tr>
<td></td>
<td>• Decide on Scope and deliverables of project</td>
</tr>
<tr>
<td></td>
<td>• Prepare a Project Plan</td>
</tr>
<tr>
<td></td>
<td>• Setup a simple SDL project with a directory structure that will work with both Apple’s XCode and Microsoft’s Visual Studio</td>
</tr>
<tr>
<td></td>
<td>• Setup repository on Google Code</td>
</tr>
<tr>
<td></td>
<td>• Do initial import to code repository</td>
</tr>
<tr>
<td></td>
<td>• Test that can check out work from repository on different machines</td>
</tr>
<tr>
<td>Week 5</td>
<td>• Research information on the OBJ file format</td>
</tr>
<tr>
<td></td>
<td>• From the OpenGL Super Bible and looking at books covering OpenGL ES 1.3 and 2.0 decide on best methods for sending data to graphics card and rendering</td>
</tr>
<tr>
<td></td>
<td>• Look at collision detection algorithms between different 3D shapes</td>
</tr>
<tr>
<td>Week 6</td>
<td>• Research collision physics from Physics for Games Developers and Physics for Games Programmers</td>
</tr>
<tr>
<td></td>
<td>• Look at tree structures used for holding model information and how they are implemented for animation</td>
</tr>
<tr>
<td></td>
<td>• Begin preparing Literature Review</td>
</tr>
<tr>
<td>Week 7</td>
<td>• Look at algorithms for morph targets, linear interpolation and cosine interpolation</td>
</tr>
<tr>
<td></td>
<td>• Research particle systems and useful formula used in particle systems</td>
</tr>
<tr>
<td></td>
<td>• Finish Literature Review</td>
</tr>
<tr>
<td>Week 8</td>
<td>• Begin preparing list of software requirements</td>
</tr>
<tr>
<td>Week 9</td>
<td>• Finalize list of software requirements</td>
</tr>
<tr>
<td></td>
<td>• Look at methods form rendering text to screen in SDL and limitations of this approach</td>
</tr>
<tr>
<td>Week 10</td>
<td>• Begin Design Document</td>
</tr>
<tr>
<td></td>
<td>• Identify some key classes for engine</td>
</tr>
<tr>
<td></td>
<td>• Prepare UML diagram</td>
</tr>
<tr>
<td>Week 11</td>
<td>• Finish Design Document</td>
</tr>
<tr>
<td></td>
<td>• Begin preparing Oral Presentation</td>
</tr>
<tr>
<td>Week 12</td>
<td>• Finish preparation for Oral Presentations</td>
</tr>
<tr>
<td></td>
<td>• Give Oral Presentation</td>
</tr>
</tbody>
</table>
### 4.2 Semester 2

<table>
<thead>
<tr>
<th>Week</th>
<th>Tasks</th>
</tr>
</thead>
</table>
| Week 1 | • Implement OBJ File Loader  
         • Implement MTL file loader  
         • Test OBJ Model Loader with models exported from 3D Studio Max and free models from the internet |
| Week 2 | • Integrate Lib3DS into engine  
         • Develop simple hit test program that’s prompts user when two objects are colliding |
| Week 3 | • With plane collision working implement a full collision mesh  
         • Look at methods of breaking terrain into smaller pieces for optimized collision testing |
| Week 4 | • Produce demo of multiple balls colliding with each other and the terrains collision mesh  
         • Implement Player class and allow user can move a ball around the terrains collision mesh |
| Week 5 | • Implement a tree structure that will allow for the creation of animations for player character  
         • This will produce a program that will allow for the rotation of the player’s limbs and printing put their rotated values. This information will then be used to code the animation |
| Week 6 | • Finish off player animations and prepare demo that will show off player animation state machine by changing player animation on a button press  
         • Implement morph targets into engine  
         • Produce demo that shows morph targets working on a terrain model (my own)  
         • Produce demo that will show morph targets working on a model of a 3D face (by modifying a free model taken from the internet)  
         • Prepare Interim Presentation  
         • Begin preparing Draft Final Report |
| Week 7 | • Give Interim Presentation  
         • Continue work on Draft Final Report |
| Week 8 | • Continue work on Draft Final Report |
| Week 9 | • Finalize gameplay demo of engine  
         • Finalize Draft Final Report |
| Week 10 | • Prepare Final Report |
| Week 11 | • Finalize Final Report  
         • Prepare Oral Presentation |
| Week 12 | • Present Oral Presentation |

*Figure 2: Proposed Work Plan - Semester 2*
5 Literature Review

5.1 Introduction

A Game Engine is a piece of software that aids in the development of a video game. The engine provides developers with some of the core functionality needed to develop their game. This functionality can include the ability to load in 3D Models created in a number of packages such as 3D Studio Max or Maya. It may also include the implementation of some sort of physics system or integration of an existing physics engine such as NVIDIA’s PhysX, Havok or Bullet. The physics system applies physics to objects in the game world making them react with each other when they collide. Some engines can even add Artificial Intelligence to agents in the game making the agents interact with each other and respond to events that may occur during the game or simulation.

From looking at engines that currently in use, three main categories of Game Engines have been identified - Open Source, Commercial and In-House.

Open Source Engines, like all other Open Source software, provide developers with access to all their code and internal workings. This allows developers to optimize the engine for their own needs, or take some code implemented in the Open Source engine and use it in an engine they may be developing themselves. Mostly hobbyist and indie developers will use Open Source engines. Two of the most popular Open Source engines used today are Ogre and Irrlicht.

Commercial engines will usually provide developers with an attractive GUI to setup their game scenes and allow developers to create scripts that control gameplay and elements within the game world. The developers of the commercial engine will sell it to clients, who will then use the engine to create their own games. The engine developers will have some sort of client support setup to aid clients and will constantly seek to improve their engine based on suggestions and feedback. Unlike Open Source engines, the coders developing the game usually do not have access to the engine’s underlying source code. Smaller, start-up game studios are likely to use a commercial engine for their first few game projects. Some popular commercial engines used today are Unity3D, Torque and the Unreal Engine.

In-House engines are developed internally in big games studios like EA and Sega. They will have an entire team dedicated to working and improving their engine, and an entirely separate team working on game titles that utilise their engine. For the latest Sonic the Hedgehog Titles, Sega have developed their own engine, titled the Hedgehog Engine, which they use to aid them when developing their Sonic games. Argonaught was a large, UK based game development studio that had developed their own in-house games engine. As well as having a team of developers working on improving the engine, they also had separate teams using their engine to create games such as iNinja, Catwoman and Harry Potter.

5.2 Game Engine Architecture

The engine produced by this project will be a very simplistic and will not include the advanced features seen in engines used in the industry today. The engine will be able to load 3D models, animate these models using a basic skeleton animation structure and morph targets, create bounding spheres for these models and handle collision between these bounding sphere and level terrain within the game environment. Terrain collision wall be controlled by a collision mesh.

Figure 3 details the architecture of the Simple 3D Game Engine being developed. It is based on a diagram detailing the architecture of a complex game engine in Gregory, J, (2009), Game Engine Architecture, page 29. This diagram will form the basis of the rest of the discussion in this section.
Figure 3: Simple 3D Game Engine Architecture, based on Gregory, J, (2009), Game Engine Architecture, page 29
5.3 3rd Party SDKs

5.3.1 Simple DirectMedia Layer (SDL)

SDL is an Open Source library that allows for setting up a window, the rendering of 2D and 3D graphics and access to low level input devices including keyboard, mouse and game pad. Through SDL’s extension libraries - SDL_Image, SDL_TTF, SDL_Mixer and SDL_NET, the functionality of SDL can be further extended to allow for the loading images of various formats, the rendering of text, playing audio files and networking. Figure 4 details the SDL libraries, extension libraries and their versions that will be used in the development of this engine.

<table>
<thead>
<tr>
<th>Library</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDL</td>
<td>v 1.2.14</td>
</tr>
<tr>
<td>SDL_Image</td>
<td>v 1.2.10</td>
</tr>
<tr>
<td>SDL_TTF</td>
<td>v 2.0.10</td>
</tr>
</tbody>
</table>

Figure 4: SDL Libraries and Versions Being Used

SDL is written in C and can be used to compile for Windows, Mac OS and Linux. Ports also exist that allow for SDL be used to create homebrew (hobbyist / indie) programs and games for the Sega Dreamcast and the Sony PSP. SDL 1.3 is currently in development and it will include support for the Apple iPhone. SDL is best used for 2D applications as it is not designed to take full advantage of the graphics hardware for rendering, however, its windowing system will work with OpenGL. This allows for the development of programs that take advantage of SDL’s windowing, device handling and SDL’s extension libraries as well as OpenGL for rendering.

5.3.2 OpenGL

OpenGL is a cross platform graphics library. It can be used to render 2D and 3D graphics on Windows, Mac and Linux machines, as well as on other devices including the Sony PlayStation 3, Nintendo Wii, PSP, iPhone and Android platforms. OpenGL has been designed to take advantage of the graphics hardware in a device. The graphics card vendor will provide optimised implementations of any OpenGL functions that their card can improve on. This allows for OpenGL programmers to take advantage of any special features of the graphics cards in a machine using standard OpenGL code.

The latest release of OpenGL is OpenGL v 4.0, with 4.1 currently in development. For smaller devices such as mobile phones and portable games consoles such as the PSP, a more compact version of OpenGL is available. This is called OpenGL Embedded System (OpenGL ES). OpenGL ES is essentially a stripped down version of the full OpenGL. It does not include costly rendering functions such as glBegin() and glEnd(), where vertices are sent one by one to the graphics card (immediate mode). Instead the programmer must take advantage of rendering techniques such as Vertex Arrays and Vertex Buffer Objects for optimized rendering. These techniques are discussed later in 5.4.2 Rendering System.

This engine will only use the functions that are available to OpenGL ES 1.1. This will help improve the performance of the engine and ensure that it is kept current with technology that is being used today.

5.3.3 LIB3DS

The 3DS file format is a popular file format for storing models created in 3D Studio Max. It can be used to store information on an entire scene in a game world and even animations. It is a very complex file format and stores a lot of information that is specific to 3d Max and of no use to the engine.

LIB3DS is an Open Source project that allows for the loading of data from a 3DS file into a program. It is then up to the programmer to use this data for physics or convert it into a format that will allow for it to be sent to the graphics card and then drawn to the screen.
5.4 Platform Independence Layer

Most Commercial and In-House Game Engines are able to compile to different target platforms such as Mac, PC, Xbox or PS3, with minimal changes to the original code. This is done by dividing the engine to different sub-systems, with each sub-system being responsible for a different task, such as rendering, handling input devices and handling a windowing system. This keeps a level of abstraction between the different areas of the engine as each sub-system is concerned only with its own internal workings. This helps portability as to change, for example, the engine’s rendering technology, all that needs to be done is replace the current rendering system with a different one, rather than having to dig through all the individual classes and change their render functions one by one.

The same approach will be taken for the development of this engine. Although for this project it will only compile to Mac OS X and Windows environments, using SDL and OpenGL, an aim of this project is to develop an understanding of how a game engine works. Therefore the idea that future revisions, after this project has been completed, may be implemented for the iPhone or PSP will be taken into consideration in the engine architecture.

SDL and OpenGL are cross platform libraries; however each platform has its own implementation of the SDL and OpenGL source files. Using C Preprocessor Macros, the target platform that is compiling the engine will be identified and then the necessary implementation of the library class included.

An example of C Preprocessor Macros for including the necessary SDL header files depending on the target platform can be seen in Figure 5.

```c
#ifdef __APPLE__     //Apple SDL Headers
#include "SDL.h"
#include "SDL_opengl.h"
#else                  //Windows SDL
#include "SDL\SDL.h"
#include "SDL\SDL_opengl.h"
#endif
```

**Figure 5: Example C Preprocessor Marco Example**

This code simply includes the necessary SDL header files when compiling on either Mac or Windows environments.

As detailed in Figure 3, this engine will have three different sub-systems in its Platform Independence Layer – a Windowing System, a Rendering System and an Input System.

### 5.4.1 Windowing System

It will be the responsibility of the Windowing System to setup a window on the target platform, in this case Mac OS or Windows OS. The Windowing System in this engine will use SDL.

### 5.4.2 Rendering System

The Rendering System is responsible for drawing the scene. In this engine the Rendering System will be implemented in OpenGL with the aim that it will be compatible with devices that support OpenGL ES 1.1.

When first learning OpenGL, students typically learn to program in immediate mode. Whilst this is makes OpenGL easier to learn, it is extremely inefficient as each vertex is sent to the graphics card one by one to be rendered. Vertex Arrays and Vertex Buffer Objects are much more efficient rendering techniques. This engine will take advantage of these more advanced and optimised rendering
techniques.

5.4.2.1 Vertex Arrays
With Vertex Arrays, the data needed to render the scene is stored in one or more arrays on the CPU and sent to the GPU in one transfer each time the scene is rendered. Vertex, Texture Coordinate and Normal data that make up model can all be stored in separate arrays or in the same interleaved array before being sent to the GPU.

5.4.2.2 Vertex Buffer Objects (VBOs)
With VBOs, the ownership of the vertices being used to render the scene is passed to OpenGL rather than the actual program. When setting up a VBO the programmer can state whether the vertex information of the VBO will change for animation or if the geometry remains static. Depending on whether the vertex data will change, and if so how frequently, OpenGL will then decide where in memory it should store this data, either on the GPU or the CPU. This provides a massive performance boost for rendering the scene.

5.4.3 Input System
SDL has its own input library that is not very well documented, or efficient for taking game pad input. To try and improve on how the engine will read game pads this system will act as a wrapper class around SDL's game pad functions.

With implementing a specific Input System, should a better, platform-specific library for reading input, such as Direct X's XInput library, be desired in a future revision, this Input System can be overwritten to use the new technology.

5.5 Core System
The Core System contains libraries that will be implemented or existing libraries that will be used to provide the core functionality of the engine. For this project a custom math library will be implemented. The standard C math library and Standard Template Library (STL) will also be used.

5.5.1 Math Library
For this engine a custom Math Library will need to be written. This will provide a number mathematical data structures for use in the engine such as 2D and 3D vertex structures.

From Abrash, M, (1997), Graphics Programming Black Book the functions below have been identified be implemented in this Math Library.

Figure 6 can be used to calculate the magnitude of a vector

\[ \|v\| = \sqrt{(v_x \cdot v_x) + (v_y \cdot v_y) + (v_z \cdot v_z)} \]

Figure 6: Formula for Magnitude of a Vector

Figure 7 can be used to calculate the dot product of two vectors.

\[ u \cdot v = (u_x \cdot v_x) + (u_y + v_y) + (u_z + v_z) \]

and also

\[ u \cdot v = \|u\| \|v\| \cos(\theta) \]

This can be changed to give us the angle between two vectors

\[ \cos(\theta) = (\|u\| \|v\|) / (u \cdot v) \]
Figure 7: Formula for Dot Product of Two Vectors

\[
\begin{align*}
\text{CrossProduct}.x &= (v1.y \times v2.z) - (v1.z - v2.y); \\
\text{CrossProduct}.y &= (v2.z \times v1.x) - (v2.x - v1.z); \\
\text{CrossProduct}.z &= (v1.x \times v2.y) - (v1.y - v2.x);
\end{align*}
\]

Figure 8: Calculating the Cross Product of Two Vectors

Figure 8 shows how to calculate the Cross product of two vectors.

\[
\begin{align*}
\text{CrossProduct}.x &= (v1.y \times v2.z) - (v1.z - v2.y); \\
\text{CrossProduct}.y &= (v2.z \times v1.x) - (v2.x - v1.z); \\
\text{CrossProduct}.z &= (v1.x \times v2.y) - (v1.y - v2.x);
\end{align*}
\]

Figure 9 shows how to calculate the surface normal of a polygon. The surface normal tells what direction the vertex is facing. This is needed for both lighting information and also for collision detection against the polygon.

1. Identify three vertices on the polygon – A, B and C. With A → C being clockwise and A → B being anti-clockwise.

2. Create two new vectors by doing the following:

   \[
   \text{Vector 1} = \text{Vertex B} - \text{Vertex A} \\
   \text{Vector 2} = \text{Vertex C} - \text{Vertex A}
   \]

3. The surface normal is produced by calculating the Cross Product of Vector 1 and Vector 2.

4. With the normal calculated, it then needs to be normalised

Figure 9: Calculating the Surface Normal of a Polygon

Figure 10 shows how to normalize a Vector. Normalizing a vector makes its length equal to one. If a normal vector’s length is not equal to one, it may produce unexpected results when rendered to screen.

\[
\text{Factor} = \sqrt{ (\text{Norm}.x \times \text{Norm}.x) \times (\text{Norm}.y \times \text{Norm}.y) \times (\text{Norm}.z \times \text{Norm}.z) } \\
\text{Norm}.x /= \text{Factor} \\
\text{Norm}.y /= \text{Factor} \\
\text{Norm}.z /= \text{Factor}
\]

Figure 10: Normalizing a Vector
5.5.2 C Math and Standard Template Library (STL)

The C Math Library provides some essential math function such as calculating the cos, sin and tan of an angle. It also provides the inverse of each of these functions.

The Standard Template Library provides a number of data structures such as Vectors and Lists that will be needed in the development of this engine.

5.6 Resource Manager

The Resource Manager is responsible for loading assets such as models, textures, sounds and movies, to be used in the game. This engine’s Resource Manager will be able to load OBJ, MTL and 3DS model files as well as image files of various formats for use as textures.

5.6.1 OBJ File Type

Alias Wavefront developed the OBJ File Format. It is one of the easiest 3D file formats to understand and any OBJ file can be opened and viewed in a standard text editor such as NotePad or TextEdit. Each line begins with a one or two letters that detail what information is being provided about the model on the rest of that line. Figure 11 details possible letter combinations and what they mean. This table was created from analyzing a number of OBJ models in TextEdit and referring to Roy Riggs information on the OBJ Model File (http://www.royriggs.com/obj.html).

Understanding these three bullet points will help in understanding the information provided in Figure 11:

- An OBJ Model is made up of one or more Groups of smaller models.
- A Group is made up of a number of Faces
- A Face is made up of a number if Vertices

<table>
<thead>
<tr>
<th>Starting Characters</th>
<th>Line Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>The Object name</td>
<td>o Sonic</td>
</tr>
<tr>
<td>v</td>
<td>A physical vertex</td>
<td>v -0.269232 0.231753 -2.057379</td>
</tr>
<tr>
<td>vt</td>
<td>A texture coordinate</td>
<td>vt 0.164063 0.507813</td>
</tr>
<tr>
<td>vn</td>
<td>A normal vertex</td>
<td>vn 0.684576 0.714890 0.142436</td>
</tr>
<tr>
<td>g</td>
<td>The name of a group of faces</td>
<td>g Face</td>
</tr>
<tr>
<td>f</td>
<td>A face made of indexed vertices</td>
<td>f 2/30/11 60/31/11 77/3/12</td>
</tr>
<tr>
<td>mtllib</td>
<td>Reference to the OBJ Model’s corresponding MTL file</td>
<td>mtllib Sonic.mtl</td>
</tr>
<tr>
<td>usemtl</td>
<td>Reference to material name in the MTL file that is to be used for the current group of objects</td>
<td>usemtl BlueFur</td>
</tr>
</tbody>
</table>

Figure 11: OBJ File Information

An OBJ file is only used for storing data on a static 3D model; it does not store data on any sort of animation.

When reading through an OBJ file, the first information on the model provided is on the physical vertices that make up the model. These vertices need to be stored by the engine as an array of a custom 3D vector structure.

After reading the vertices that make up the OBJ model, the next information provided by the OBJ file is the model normals. These can either be stored in the same way as the physical vertices or the normals can be calculated by the engine when the model has been entirely loaded. Some models may not provide normal information on the model.
The next piece of data given by the OBJ file is on the texture co-ordinates of the model. These will need to be stored as an array of a custom 2D vector structure. Some models may not provide texture coordinates, meaning the model is not textured.

At this point in the read, all the raw data that makes up the model will be loaded into and store by the engine, the next step is to make some sense of how all this vertex information ties together. After reading in all this raw data the OBJ file then details on the faces and polygon groups that make up the model.

The OBJ model is made up or one, or more Group(s) of vertices. Each Group will be given a name either by the creator of the 3D model or by the modeling package when the model was exported. After providing the Group name, information on the faces that make up the group will be provided. The engine will need to store each Group either in a custom structure or class.

Each Face is either a quad or a triangle. Each point of the face will contain a normal coordinate, a texture coordinate and a physical coordinate. This data is passed in as three indexes into to each of the arrays created earlier when loading in the raw vertex data. Each Face will be stored as an array of a custom face structure.

It should be noted that the indexed vertices referred to by the OBJ file start at index 1; however arrays in C++ start at index 0. To accommodate for this one will need to be subtracted from each index as it is loaded into the engine.

### 5.6.2 MTL File Type

Each OBJ file will have a corresponding MTL file. Like the OBJ file it too can be viewed in a text editor. Again each line will begin with one or two letters that detail what information is on each line in the file. Figure 12 details the possible letter combinations and their meanings in the MTL file. This table was created from analysis of a number of MTL files and referring to [http://www.fileformat.info/format/material/](http://www.fileformat.info/format/material/).

<table>
<thead>
<tr>
<th>Starting Characters</th>
<th>Line Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>newmtl</td>
<td>The current material name</td>
<td>newmtl BlueFur</td>
</tr>
<tr>
<td>Ns</td>
<td>The shininess of the material</td>
<td>Ns 86.274510</td>
</tr>
<tr>
<td>Ka</td>
<td>The ambient colour of the Material</td>
<td>Ka 0.000000 0.0000000 0.000000</td>
</tr>
<tr>
<td>Kd</td>
<td>The diffuse colour of the Material</td>
<td>Kd 0.351994 0.284977 0.720587</td>
</tr>
<tr>
<td>Ks</td>
<td>The specular colour of the Material</td>
<td>Ks 0.062500 0.062500 0.062500</td>
</tr>
</tbody>
</table>

Figure 12: MTL File Information

An OBJ model's corresponding MTL file is detailed in its line beginning `mtllib`.

The MTL file will contain information on a number of materials that are to be applied when rendering the OBJ file. Each material will have a name as well as diffuse, ambient, specular and shininess properties.

As a Group is declared in the OBJ file, the next line will Each group in the OBJ file will refer to a Material in the MTL file through the line beginning `usemtl`.

Appendix 1 shows a 3D cube along with its corresponding OBJ and MTL file being viewed in a text editor.
5.6.3 Texture Loader
Textures used in the engine will be in PNG format. SDL's SDL_ImagLib will allow for the loading of images in a number of different formats for use as textures or UI elements in the engine.

5.6.4 Animation System
There will be two separate animation systems in place in the engine.

5.6.4.1 Skeletal Animation Tree Structure
To allow 3D models to move and animate, a tree-node structure similar to Kasper Peeter's tree.hh structure (http://tree.phi-sci.com/) will be implemented. OBJ models for each body part (hand, head, torso leg, etc) will be stored as separate OBJ files. The animation tree-node structure will then be used to connect these body parts together. Original image Taken from - http://www.the-eleventh.com/node/57
Appendix 2 and Figure 13 show how a standard human model would be made up using this animation structure. A similar animation system can be seen in Campbell, J.G. (2009). Computer Graphics using OpenGL, Section 14-5.

The Torso will be the top level of the tree with child nodes for the head, upper arms and upper legs. The lower arms would then be child nodes of the upper arms. The hands are then child nodes of the lower arms. The same applied for the upper legs, lower legs and feet.

![Figure 13: Tree Structure of Animation Structure](http://example.com/image.jpg)

Rotations on a parent body node will affect only itself and its children nodes. For instance a rotation of 33.0 degrees of the upper arm will affect the lower arm and the hand, but not the torso or any other body parts. For animations each body part will have an animation time and an angle it needs to be adjusted too. A class will have to be written to hold information on an OBJ Model. To allow for animation, a separate class will have to be written that will hold the OBJ object for a body part along with a number of different animations values (rotations and times) for that body part.

5.6.4.2 Morph Targets
Morph Targets are used to animate between different 3D models. A constraint that this engine will have is that the 3D models must be made up of the same number of points. Morph Targets are extremely effective at animating facial expressions and basic water such as the sea.

From Akenine-Moller, T, Haines, E, Hoffman, N, (2008). Real-Time Rendering, Figure 14 has been constructed as a high level algorithm for implementing morph targets into the engine.
1. Compare two models and check they have the same number of vertices.
2. Make a list of all vertices that are different in the two models.
3. Using some form of interpolation, create a third model that is made up from the interpolated points.
4. When the interpolation is complete check the vertices against the next model in the animation sequence and repeat.

**Figure 14: High Level Morph Target Algorithm**

### 5.6.4.3 Linear Interpolation

There are a number of different methods of interpolation. The method of interpolation that will be used in this engine's implementation of Morph Targets will be Linear Interpolation. It was considered to use both Linear Interpolation and Cosine Interpolation for morphing, however, during research it has been decided that the added complexity of implementing Cosine Interpolation would not be justified by the end result.

From Dorn, O (2010), [http://www.maths.manchester.ac.uk/service/MATH29641/2Q1Lecture1.pdf](http://www.maths.manchester.ac.uk/service/MATH29641/2Q1Lecture1.pdf), Figure 15 has been created to show the formula for Linear Interpolation.

\[
p = p_1 + t(p_2 - p_1)
\]

Where - \( p \) is the current position  
- \( p_1 \) is the starting position  
- \( p_2 \) is the desired position  
- \( t \) is the elapsed time / total morph time (must be a value between 0.0 and 1.0)

**Figure 15: Formula for Linear Interpolation**

This must be carried out for the x, y and z position of each vertex to be morphed.

For example, for a point on the x-axis with starting position of 23.0 and a destination point of 65.0 and a morph time of 30 seconds this above formula old look like:

\[
p = p_1 + t(p_2 - p_1) \\
p_X = 23.0 + t(65.0 - 23.0) \\
p_X = 23.0 + (\text{elapsed_time} / \text{total_time}) \times (65.0 - 23) \\
p_X = 23.0 + (\text{elapsed_time} / 30.0) \times (65.0 - 23)
\]

**Figure 16: Linear Interpolation Example**
5.7 Physics System

The Physics System will detect and handle collision between the planes that make up the collision mesh and bounding spheres of objects in the game world.

5.7.1 Bounding Spheres

There are two main reasons why Bounding Spheres are being used for collision detection for moving objects in the engine, rather than Bounding Boxes:

1. From research in collision detection it has been found that Bounding Spheres require less processing time for collision detection than Bounding Boxes
2. The math for collision detection between axis-aligned Bounding Boxes is quite simple, however the math involved for rotated bounding boxes, where the rotation angle modulus 90 does not evaluate to 0, can become quite complex and would take too long to implement into the engine. With Bounding Spheres there is not need to worry about the rotation of the spheres involved as the collision detection code remains the same.

5.7.2 Creating the Collision Sphere

With the mesh data for the model loaded into the engine, based on Ericson, C, (2005), Real Time Collision Detection, Figure 17 can be used to calculate the bounding sphere of a model. The only value needed to create the sphere is its radius. Its origin point will be the same as the models.

```plaintext
Float min_x, max_x, min_y, max_y, min_z, max_z;
for(all verts in model)
   Compare x, y and z values and decide if need to be changed
   xRadius = max_x - min_x;
   yRadius = max_y - min_y;
   zRadius = max_z = min_z;
return largest of Radius values;
```

Figure 17: Algorithm for Creating a Bounding Sphere from Model Data

5.7.3 Sphere - Sphere Collision Detection

Based on formula from Ericson, C, (2005), Real Time Collision Detection, page 88, Figure 18 and Figure 19 have been derived to detect whether two spheres are colliding.

```plaintext
Vector d = sphere_A.centre - sphere_B.centre;
float dist_squared = |d| * |d|;
This evaluates to the square of the distance between the spheres

float radius = sphere_A.radius + sphere_B.radius;
The reason we work with squares is to accommodate for negative values, should sphere be below, in-front or to the left of sphere_A

return ( dist_squared <= (radius * radius) );
If this evaluates to true then the spheres are intersecting, otherwise they are not
```

Figure 18: Algorithm for Sphere - Sphere Collision Detection
5.7.4 Sphere - Plane Collision Detection

Based on formula from Ericson, C, (2005), *Real Time Collision Detection*, page 160, Figure 20 and Figure 21 have been derived to detect whether two spheres intersect.

```c
float distance = DotProduct(sphere.centre, plane.normal) - plane.position;
The plane must be normalized (magnitude of the plane's normal must be 1)

return (distance <= sphere.radius);
```

Figure 20: Algorithm for Sphere - Plane Collision Detection

Figure 19: Diagram Showing Sphere - Sphere Collision Detection

Figure 21: Diagram Showing Sphere - Plane Collision Detection
6 Requirements

This project breaks down into seven key systems, each providing a different aspects of functionality. These seven key areas are Models and Materials, Animations Structure, Math, Physics and Collision Detection, Model Editor and Tester, Model Viewer and a Gameplay Demo.

6.1 Models and Materials
R 1. Must be able to Load OBJ Model Files
R 2. Must be able Load OBJ Models exported from Blender and 3D Studio Max
R 3. Must be able to Load 3DS Model using LIB3DS
R 4. Must Load a Models Corresponding MTL Material file
R 5. Textures referenced in the MTL file must also be loaded into the Engine and stored, assuming the texture is contained in the Project Package
R 6. In cases where lighting is disables a model’s colour will be decided from the diffuse property from its MTL file
R 7. Must load texture coordinates from file
R 8. Normal data must be loaded correctly
R 9. A Model Tree Structure/Class must be devised to hold together different models to make up a single, more complex model

6.2 Animation System
R 10. An Animation Structure must be devised that will store animation data for a given model
R 11. Any rotations applied to a parent model object must also be applied to any corresponding child objects.
R 12. Animation rotations must occur smoothly using Linear Interpolation over a set time period
R 13. Animated Rotations must be able to occur on X, Y and Z axis simultaneously
R 14. Animation Sequences must be able to contain any desired number of rotations around each axis
R 15. Model Animations can be one of 3 types - Stop, Reverse, Loop
R 16. Stop Animations must Stop when they reach the end of their Animation Sequences
R 17. Reverse Animations must animate backward through the animation sequence when they reach their last animation rotation, effectively reversing their animation
R 18. Loop animations must repeat from the beginning when the animation sequence is repeated
R 19. Models set to use morph targets must morphs over a specified time using linear interpolation.
6.3 Math, Physics and Collision Detection

R 20. Must be have function to calculate the magnitude of a vector

R 21. Must have function to calculate the dot product of two vectors

R 22. Must have function to calculate the surface normal of a face

R 23. Must have function to normalize a vector

R 24. Must be able to generate s Model’s Collision Sphere

R 25. Math Library calculated normals for a collision mesh must be used for collision mesh collision detection

R 26. Must be able to detect Sphere – Sphere Collisions

R 27. Must be able to detect Sphere – Collision Mesh Collisions

R 28. Collision sphere must collide respond to collision against a collision mesh

6.4 Rendering System

R 29. Rendering System code must be kept independent of rest of program to allow for different implementations to be easily produced if desired in future revisions

R 30. All objects are to be rendered using Vertex Buffer Objexts (VBOs) or a Vertex Array to keep with current standards and to optimize rendering

R 31. In the final Release build, only OpenGL calls available in OpenGL ES 1.1 are to be used

R 32. Renderer must draw 3D Models Correctly on screen

R 33. Renderer must display Material Information Correctly on screen

R 34. Renderer must display textures correctly and maps them correctly to a model’s texture coordinates

R 35. Renderer must use display Model normals correctly under lighting

6.5 Model Editor

R 36. Allow for the manipulation of Model Material Properties

R 37. Allow the linking of simple models to create a hierarchial models

R 38. Allow for the User to create Animations for these hierarchial models

R 39. Export hierarchial model data

R 40. Export animation data

R 41. Toggle the drawing of Bounding Spheres

R 42. Allow user to Scale Models

R 43. Allow user to Rotate Models
R 44. Allow user to Translate Models

6.6 Model Viewer
R 45. Display Model data exported by the Engine Editor
R 46. Animate Model data exported by the Engine Editor
R 47. Windows/OS X SDL Viewer will allow for toggling rendering of Bounding Spheres
R 48. Windows/OS X SDL Viewer will switch between Model animations on key presses
R 49. iOS Viewer will switch between Model Animations on key presses

6.7 Gameplay Demo
R 50. Player must be controlled using a game pad
R 51. Player must be able to jump and fall due to gravity
R 52. Player will collide with other objects in the game world
R 53. Player will change animation depending on what they are doing – Idle, Running, Jumping
7 Design

This section details the Design of this Simple 3D Games Engine. It discusses how different sections of the Engine work together and how the user interacts with different elements within the Engine. Again, like in previous sections, the Engine Design has been divided into the seven key areas of this project - Models and Materials, Animations Structure, Math, Physics and Collision Detection, Model Editor and Tester, Model Viewer and a Gameplay Demo.

Figure 22 shows the Class Diagram for this Engine. 7.1 provides Class Descriptions for these classes.

Figure 22: System Class Diagram

2 A larger version of this diagram can be found the accompanying CD with this document at location CD://ClassDiag.jpg
7.1 Class Descriptions

The class descriptions below refer to the class diagram in

1. MathLib – Contains Needed Mathematical Structures and Functions
2. Material – Holds data on a Material
3. MeshObject – Holds data on the smallest possible mesh held by the Engine
4. Mesh3D – Holds a Collection of MeshObject’s together to make up a Mesh
5. ModelAnimation – Holds Animation Data for a Mesh3D object
6. Camera – The camera for viewing the scene
7. Renderer – Does all drawing
8. Model3D – Holds a number of Mesh3D objects in place to make a hierarchical model
9. ResourceManager – Handles Loading of files into Engine
10. CollisionMesh – Used to store collision data for a Mesh
11. Pad – Reads and maps input from a Game Pad
12. Engine – This class would be replaced by a developer using the Engine to make their game
13. OpenGLView – Allows for the Engine to be Viewed in an Objective C Interface on OS X
14. GUIController – Handles communication between Objective C GUI code and Engine C++ code
15. SDLWindow – SDL Model Viewer Interface for Windows and OS X
16. iOS Controller – Handles running of Engine on iOS device. Passes touch input and iOS alerts to Engine.

7.2 Models and Materials

In the Engine, objects that hold data on loaded individual OBJ Models are referred to as Mesh3D objects. A number of Mesh3D objects linked together, along with some data detailing how they are linked together, make up a complex model object, referred to in the Engine as a Model3D object.

OBJ Models (Mesh3D Objects) themselves are made up of a number of groups of vertexes. In the Engine an individual group of vertexes is stored as a MeshObject.

Figure 23 and Figure 24 represents this information visually and Appendix 3 shows an Entity Relationship Diagram showing the relationship between these different classes.

Original Bowser Jr. Image from Super Mario 64, Nintendo, June 1996

Figure 23: Model3D, Mesh3D and MeshObject Relationship
Appendix 4 shows the Process Diagrams for loading OBJ and MTL files.
7.3 Animation System

Each Mesh3D object will contain a number of Animations, which when animated and linked together with other Mesh3D objects will produce an Animated Complex Model3D object.

Animations themselves are made up of a number of rotations about the Mesh3D’s origin on the X, Y and Z axis. An Animation can animate through a number of angle rotations. These rotations are stored in an Array of 3D Vectors.

When animating, the Mesh3D’s current rotation about each axis will be calculated using Linear Interpolation. The Animation itself will track its progress through its Array of Animation Angles.

Animations can be one of three types – Stop, Repeat or Loop.

Stop Animations will play once, and then stop. This would be used for animations such as a model jumping, where the first few seconds of the action need to be animate, but for the remainder of the action the model remains static.

Reverse Animations will reverse back through the animation sequence when it has completed. This will give the illusion of a smooth, repeating animation sequence such as when a character is running.

Loop Animations will play once and then animate back to their starting sequence and repeat the animation. This is used typically for one-off action animations such as when a character reaches out to grab something.

7.4 Math, Physics and Collision Detection

The MathLib class contains all the mathematical structures and functions needed by the Engine. These structures will include mathematical variables such as 3D and 3D Vectors and the needed mathematically functions to manipulate these Vectors.

7.5 Rendering System

The Rendering System does all Rendering in the Engine. It will be developed so that it can easily be swapped out and a new Rendering System may be written in OpenGL ES 2.0, Direct X or another such technology.

In the Engine Editor and Engine Viewer (on OS X and Windows) the Rendering System takes advantage of the full OpenGL library and the GLUT library. In the iOS Viewer and final Core Engine Framework, only calls available in OpenGL ES 1.1 are made, as this is the general implementation of OpenGL that is supported by most devices such as iOS devices, the PSP, the PS3 and Android OS devices.

Appendix 5 shows the Engine Architecture and it can be seen that the Rendering System could be swapped out and replaced with a new Renderer with minimal changes to the code. The new Renderer would simply need to ensure that it followed the same class design laid out in the Renderer’s header file. This demonstrates Requirement R29.

7.6 Model Editor

The Model Editor allows the user to load and create hierarchical 3D models made up of smaller OBJ Meshes. These models can then be animated and their material properties manipulated. This complex Model along with its Animation data can all be exported from the Engine Editor as a custom file format developed for this project - .objx. Material data can also be exported as a second custom file format - .mtlx.
7.6.1 OBJX and MTLX File Formats

As mentioned in 5.6.1 OBJ File Type, OBJ models are stored by writing each individual piece of data on the Model to a line in the file. Each line starts with a letter or character sequence detailing what the data on that line represents.

The OBJ file format is a custom file format that has been developed to hold exported Model3D data from the Engine Editor. An OBJ file, along with its corresponding MTLX file, contains all data to make up a complex 3D Model, its model structure, animation data and material properties.

The OBJX and MTLX file formats have both been developed based on the OBJ and MTL file formats. Like the OBJ and MTL file formats, in OBJX and MTLX files, individual pieces of model data are stored on each line of a file. Each line begins with a letter or character sequence that is used to describe what the data on that line represents.

Figure 25 and Figure 26 detail the character sequences used in OBJX and MTLX files and their meanings.

<table>
<thead>
<tr>
<th>Letter or Character Sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>The name of the Model3D object</td>
</tr>
<tr>
<td>o</td>
<td>A Mesh3D object name</td>
</tr>
<tr>
<td>p</td>
<td>The Name of the Mesh3D’s parent model</td>
</tr>
<tr>
<td>g</td>
<td>A MeshObject object’s name</td>
</tr>
<tr>
<td>usemtl</td>
<td>A MeshObject object’s material name</td>
</tr>
<tr>
<td>pos</td>
<td>A Mesh3D object’s position</td>
</tr>
<tr>
<td>rot</td>
<td>A Mesh3D object’s rotation</td>
</tr>
<tr>
<td>v</td>
<td>A position vector</td>
</tr>
<tr>
<td>vt</td>
<td>A texture coordinate</td>
</tr>
<tr>
<td>vn</td>
<td>A normal vector</td>
</tr>
<tr>
<td>a</td>
<td>Animation Name</td>
</tr>
<tr>
<td>as</td>
<td>Animation Start Angle</td>
</tr>
<tr>
<td>ae</td>
<td>Animation End Angle</td>
</tr>
<tr>
<td>at</td>
<td>Animation End Time</td>
</tr>
<tr>
<td>mtllib</td>
<td>The model’s corresponding MTLX file</td>
</tr>
</tbody>
</table>

Figure 25: OBJX File Info

<table>
<thead>
<tr>
<th>Letter or Character Sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>newmtl</td>
<td>Name of a New Material</td>
</tr>
<tr>
<td>Ka</td>
<td>Ambient colour of Material</td>
</tr>
<tr>
<td>Kd</td>
<td>Diffuse colour of Material</td>
</tr>
<tr>
<td>Ks</td>
<td>Specular colour of Material</td>
</tr>
<tr>
<td>Kt</td>
<td>The texture location/name of the Materials texture (if the material uses a texture)</td>
</tr>
</tbody>
</table>

Figure 26: MTLX File Info
7.6.2 Engine Editor GUI

The Engine Editor has a full GUI that allows provides a friendly, easy-to-use interface for the user to interact with Models. It allows for the following functionality through its interface:

- Rotate, Scale and Translate models using the Mouse
- Zoom In/Out using the mouse
- Toggle the rendering of Bounding Spheres
- Load in any OBJ, OBJX, MTL or MTLX files into the Editor for use
- Lighting can be enabled and disabled
- Switch between the current Mesh3D object and MeshObject that they are changing
- It displays data on the currently selected object
- Animations can be Created and Modified
- Preview Model Animations
- Prompts the User when the do something or there is an error
- Meshes can be linked together to make complex Models
- OBJX and MTLX files can be exported
- Material Properties can be manipulated

Appendix 6 shows the initial design of the Engine Editor and Appendix 7 details its planned functionality. Appendix 8 shows the final design of the Editor along with its final functionality.

7.7 Model Viewer

The Model Viewer allows the viewing of models loaded from an OBJX file along with their properties from their corresponding MTLX file. Two different Viewers are developed – one targeted at Windows and Mac OS X Operating Systems, and a second targeted at iOS devices.

The Model Viewer allows developers to test models they export from the Model Editor. It allows them to know how well their exported models and animations will work in a game using this Engine on their desired target device.

7.7.1 Windows and Mac OS X Viewer

The Windows Mac and OS X Viewer is developed using SDL for its front end windowing system, texture loading and for handling input from the Keyboard and Game Pad Figure 27 details how the user can interact with this Viewer App.

<table>
<thead>
<tr>
<th>Key Press</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>Move Camera Up</td>
</tr>
<tr>
<td>A</td>
<td>Move Camera Left</td>
</tr>
<tr>
<td>S</td>
<td>Move Camera Down</td>
</tr>
<tr>
<td>D</td>
<td>Move Camera Right</td>
</tr>
<tr>
<td>Z</td>
<td>Zoom Camera In</td>
</tr>
<tr>
<td>X</td>
<td>Zoom Camera Out</td>
</tr>
<tr>
<td>N</td>
<td>Next Animation</td>
</tr>
<tr>
<td>P</td>
<td>Previous Animation</td>
</tr>
<tr>
<td>Up</td>
<td>Move Model Up</td>
</tr>
<tr>
<td>Down</td>
<td>Move Model Down</td>
</tr>
<tr>
<td>Left</td>
<td>Move Model Left</td>
</tr>
<tr>
<td>Right</td>
<td>Move Model Right</td>
</tr>
</tbody>
</table>

Figure 27: SDL Viewer Keyboard Controls
7.7.2 iOS Viewer

In the iOS viewer, models are also loaded into the centre of the world. These models rotate about the Y-Axis by themselves, however the user can touch the screen to change the model’s current animation. It would have been nice to allow the user to interact with the model using touch gestures, however this was beyond the scope of this project.

7.8 Gameplay Demo

In the Gameplay Demo the Player uses a Game Pad to control a complex model moving around a Simple 3D scene. The Player’s Model has Idle, Running and Jumping animations. The demo also demonstrates simple collision detection between the Player’s model and other Models in the scene. Figure 28 details the Game Pad controls for the Player in the Simple Game Play demo.

<table>
<thead>
<tr>
<th>Key Press</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-Pad</td>
<td>Move Player</td>
</tr>
<tr>
<td>A (Xbox Pad)</td>
<td>Jump</td>
</tr>
<tr>
<td>Triggers</td>
<td>Rotate Camera</td>
</tr>
</tbody>
</table>

Figure 28: Gameplay Demo Game Pad Controls
8 Implementation

This section details how some of the various elements of this Simple 3D Game Engine were developed. As in previous sections it is divided into seven sections - Models and Materials, Animations Structure, Math, Physics and Collision Detection, Model Editor and Tester, Model Viewer and a Gameplay Demo.

8.1 Models and Materials

8.1.1 ResourceManager Class (R5)

For this Engine, a ResourceManager class has been developed. The ResourceManager class is responsible for the loading of files that can be used by the engine. It supports the loading of OBJ, OBJX, MTL and MTLX files. It also supports the loading of a number of different image files, however the files supported depends on its underlying technology. In the SDL View Application, SDL’s SDL Image Library is used to load images, in the iOS Viewer Apple’s UIImage is supported to load images and in the Engine Viewer Apple’s UIKit framework is used to load images. The image file formats supported varies with each of these different technologies.

A Game developed using the Engine will contain one ResourceManager Object that it will use to load all assets needed in the game.

8.1.2 OBJ File Loader (R1, R2, R7, R8, R32 – R35)

Each point on a 3D model is made up of a 3D position co-ordinate, a 3D normal co-ordinate, a 2D texture co-ordinate and a 3D colour co-ordinate. Figure 29 below shows the Data Structure that is used to store this data for each vertex in a 3D model.

```
/* - - Model Vertex Structure - - */
typedef struct{
    Vector3f position;
    Vector2f tex;
    Vector3f normals;
    Vector3f colour;
} Vertex;
```

![Figure 29: Model Point Vertex Structure](image)

In OpenGL ES only Triangular polygons can be rendered, as opposed to the full version of OpenGL in which Quad polygons may also be rendered. A result of this, standard OBJ Meshes that may be loaded into the Engine must be made up of Triangles. Figure 30 and Figure 31 below detail the settings used to export models from 3D Studio Max and Blender so they will be compatible with this Engine.
It was found when loading in experimental OBJ files, that different model creation packages added their own additional data to the OBJ format. When the Engine attempted to load models with these extra pieces of package data the Engine would crash or produce distorted models, as it did not know how to handle the information being passed into it.

To solve this problem, a file parser was created. The file parser scans through the OBJ file before loading the model, ensuring that it only contains recognized characters sequences. Any data unrecognized is removed from the file. This ensures that only data the engine can handle is being read in.

Figure 32 below shows the main code from this OBJ File Parser function that decides what data to remove from the file. Essentially all the parser does is read the OBJ file line by line and checks that it begins with a recognized character sequence. Lines beginning with invalid sequences are removed from the file. This is called just before the model data is actually loaded.

```cpp
switch(tmp_line.at(0)) {
    //Only Valid Start Characters Stored
    case 'o': case 'g': case 'u': case 'v': case 'm': case 'f':
        parse_output << tmp_line << "\n";
        break;
    default:
        std::cout << "Parser Data Dropped : " << tmp_line << "\n";
```

8.1.3 MTL File Loader (R4, R5)

Model Materials each contain diffuse, specular and ambient colour data. Each Material also contains a name, its texture name and a reference to the texture loaded by OpenGL. Figure 33 shows this Material structure.

```c
/* - - Material Structure - - */
typedef struct{
    std::string mat_name;
    std::string tex_name;
    Vector3f ambient;
    Vector3f diffuse;
    Vector3f specular;
    bool textured;
    unsigned int texture;
};
```
Each MeshObject has a Material. It was found in development that a number of MeshObject’s where each using the same Material, but where each individually storing this data. Whilst in an example with only a few models the memory wasted holding the same Material data was not noticeable, in a game with a number of Models, or a number of the same model, this would become a real problem.

To address this issue of the same Material being stored multiple times, the ResourceManager class is used to store a bank of textures in an Array. When loading a Material into the Engine, it is checked that the same Material does not already exist in the ResourceManager’s bank of Materials. Each MeshObject now, instead of holding actual Material data, contains a reference to the Memory Address of its matching Material in the ResourceManager’s bank of Materials.

Appendix 9 details this relationship between the MeshObject and the ResourceManager.

When loading MTL files, like found with OBJ files, model creation programs added their own extra data and character sequences to the MTL files. Again, as was done with OBJ files, a MTL File Parser had to be written. This parser removes all data that will not be understood by the Engine. Figure 34 shows the core functionality of this MTL file parser.

### 8.2 Animation System (R10 – R18)

As mentioned before, Animations performed by this Engine are similar to what may be seen in older generation consoles such as the Sony PSOne and the Nintendo 64. Each Mesh3D object within a Model3D object contains the same number of animations. When these animations are animated together they perform an animation for the whole Model.

Figure 35 below shows the Class Structure that has been designed to hold information on an Animation. Each Mesh3D Object holds Animations in an Array. It will will have a marker indicating the index of the current animation within this Array. Changing this index will change hows the Mesh3D Object is currently being animated.
float elapsed_time;
bool playing;
};

Figure 35: Class Structure for Holding Animation Data

Figure 36 shows the how linear interpolation is used to calculate the current rotation of a Mesh3D object.

```c
//update the animation angle
anim.current_angle[X_POS] = (anim.elapsed_time / anim.anim_time) * (anim.start_angle[X_POS] - anim.end_angle[X_POS]);
```

Figure 36: Function to Update an Animation
8.3 Math, Physics and Collision Detection

The Engine can carry out some functions for Vector mathematics and collision detection between Bounding Spheres.

8.3.1 MathLib Class (R20, R21, R23)

The MathLib class contains a number of Vector Math functions. Figure 37 below shows the data structures for 2D and 3D Vectors.

```c
/* --- Structures for 2D and 3D Vectors --- */
typedef float Vector3f[3];
typedef float Vector2f[2];
```

**Figure 37:** Data Structures for 2D Vectors and 3D Vectors

The MathLib class contains a structure to hold Bounding Sphere data. Each Model3D object contains a function to check for a collision between its own Bounding Sphere, and the Bounding Sphere of another Model3D object. Figure 38 shows this Bounding Sphere structure.

```c
/* --- Bounding Spheres --- */
typedef struct{
    Vector3f position;
    float radius;
} BoundingSphere;
```

**Figure 38:** Bounding Sphere Data Structure

Figure 39, Figure 40, Figure 41 and Figure 42 show some Vector Math functions.

```c
float GetVectorMagnitude(Vector3f v){
    return sqrt((v.x * v.x) + (v.y * v.y) + (v.z * v.z));
}
```

**Figure 39:** Function to Calculate Magnitude of a 3D Vector

```c
void NormalizeVector(Vector3f &v){
    float mag =GetVectorMagnitude(v);
    v.x /= mag;
    v.y /= mag;
    v.z /= mag;
}
```

**Figure 40:** Function to Normalize a 3D Vector

```c
Vector3f CalculateCrossProduct(Vector3f a, Vector3f b){
    Vector3f cross;
    cross.x = (a.y * b.z) - (b.y * a.z);
    cross.y = (a.z * b.x) - (b.z * a.x);
    cross.z = (a.x * b.y) - (b.x * a.y);
    return cross;
}
```

**Figure 41:** Function to Calculate Cross Product of two 3D Vectors
Development of a Simple 3D Game Engine

```
float CalculateDotProduct(Vector3f a, Vector3f b)
{
    return (a.x * b.x) + (a.y * b.y) + (a.z * b.z);
}
```

Figure 42: Function to Calculate Dot Product of two 3D Vectors

### 8.3.2 Sphere – Sphere Collision Detection (R24, R26, R41)

Each Model3D has a bounding Sphere that encapsulates the Model3D Object. Each Model3D has a function to check for a collision between its own Bounding Sphere and a second Model’s Bounding Sphere. Figure 43 shows this function.

```
bool Model3D::HitTest(BoundingSphere &b)
{
    //Sphere Radius's
    float radius_sum = bs.radius + b.radius;
    //Distance Between Centres
    float xDist, yDist, zDist;
    xDist = b.position[X_POS] - bs.position[X_POS];
    yDist = b.position[Y_POS] - bs.position[Y_POS];
    zDist = b.position[Z_POS] - bs.position[Z_POS];
    float dist = sqrt((xDist * xDist + yDist * yDist) + zDist * zDist);
    //Return If Colliding or Not
    return (dist < radius_sum);
}
```

Figure 43: Model3D Class function for checking for a Collision with a Second Model3D Object

Figure 44: - Model3D with Bounding Sphere Calculated and Rendered in Model Editor
8.3.3 Sphere – Collision Mesh Collision Detection (R22, R25, R27, R28)

It was originally planned that Sphere - Collision Mesh collision detection would be implemented into the Engine, however this functionality has had to be discarded. An attempt has been made at this functionality, however it is not complete.

It was planned that a Model’s Bounding Sphere could have been tested for a collision against every triangle plane in a simple Mesh structure. Based on the surface normal coming out of the triangle and the position of the Bounding Sphere, it is possible to test for a collision. Figure 45 below show the data structure for holding data on a Triangle and the function for calculating the Surface Normal of a Triangle.

```c
typedef struct
    { Vector3f p1;
    Vector3f p2;
    Vector3f p3;
    } Triangle;

Vector3f CalculateTriangleNormal(Triangle t)
    {
    Vector3f u = SubtractVectors(t.p2, t.p3);
    Vector3f v = SubtractVectors(t.p1, t.p3);
    Vector3f n = CalculateCrossProduct(u, v);
    return n;
    }
```

Figure 45: Triangle Structure and Function for Calculating Triangle Surface Normal

Figure 46 below shows a Mesh being rendered per Vertex Normals compared to this Per Surface Normals used for Collision Detection.

![Figure 46: Lighting using Surface Normals (Left) and Per Vertex Normals (Right)](image)

In the attempt at developing at Sphere – Collision Mesh collision detection, the Bounding Sphere is checked for a collision against each triangle in the Collision Mesh. Colliding Triangles are stored in a Vector. This is where the function failed.

Figure 47 shows the triangles (in red) that are being detected as colliding with the Bounding Sphere. It clearly shows that some triangles lying outside of the bound being detected as colliding.
The next stage of implementing this Collision Mesh would be to identify the closest Triangle to the Plane, and using that triangle, re-adjust the Model’s position.

**8.4 Rendering System (R1 – R8, R29 – R35)**

The Rendering system used in the Engine Editor and SDL Viewer takes advantage of the full OpenGL library. The iOS Viewer is limited to only using calls available in OpenGL ES 1.1. Models are drawn using Vertex Arrays.

*Figure 48* displays a textured, OBJ Model being displayed under lighting correctly using its own per-vertex normal data. The OBJ model itself was exported from Blender.

**8.5 Engine Editor and Tester**

The Engine editor allows for the loading, viewing, editing, animating and saving of models and their material properties. These exported models and materials will be in the custom designed OBJX and MTLX file formats, which are compatible with this Simple 3D Engine.

The initial plan was to allow the User to only interact with the Editor using the keyboard, however when beginning to design and implement this system, it was decided that the desired functionality of the Editor was too complex to only allow interactions using the keyboard. It was decided that the best method for the User to interact with the editor was to develop a full User Interface.

*Figure 49* below shows the final visual layout of the Engine Editor. *Appendix 8* shows the final Editor along with the functionality of its buttons.
The User Interface was developed using Apple’s Interface Builder and Objective C.

As developing a full UI was not in part of the original development plan, the decision to develop this meant that some of the planned functionality had to be left out of the project due to time restraints.

The decision to develop a UI for Engine Editor also restricted what platform the editor would work on as there are very few cross-platform UI platforms. It was decided that the Editor would be developed for Mac OS X as that was the primary platform for development, and that a viewer program would be developed for Windows to view the output of the Editor.

OS X’s IDE, XCode, provides an application to develop Program Interfaces called Interface Builder. This tool allows the interface to be setup easily, by allowing the user to drag elements onto a window layout (much like Visual Studios provide for Visual Basic and C# Applications).

A problem with Interface Builder was that it is designed for Objective C applications rather than C/C++ development. To get the Engine to talk to the User Interface a wrapper class had to be written that handled calls from the Objective C User Interface, to the C++ Engine Framework. As there was no prior knowledge of Objective C, this also extended the time spent developing the User Interface, which contributed to the decision to drop some of the Engine’s proposed functionality. It is however felt that this User Interface provides more value to the overall project than has been taken away by the discarded functionality.

Figure 49: Final Editor User Interface
8.5.1 OBJX Model Format (R7 – R9, R32 – R35, R37 – R40, R42 - R44)

As detailed in before in 7.6.1 OBJX and MTLX File Formats, the OBJX file format is a custom file format that has been devised for this Engine. It holds data on a complex model and its animations. Essentially all data stored in a Model3D object is exported to an OBJX file from the Editors.

The Model3D class is designed to hold together all the items are needed to make up a complex 3D model. It mainly consists of a number of Mesh3D objects with some link information of how these models are connect to makeup the complex model. Mesh3D objects needed to make up a complex model are stored together as an Array in the Model3D object. Each Mesh3D objects then contains a Mesh3D memory pointer (*) to its Parent Mesh in the Model3D’s Mesh3D Array. Each Mesh3D object also then contains an Array of Mesh3D pointers to its own Children Mesh in the Model3D’s Mesh3D Array. The Mesh3D element in the Model3D’s Array with a NULL parent pointer is the Parent/Base Mesh3D of that model. The Model3D object itself contains a Mesh3D pointer to its Base Mesh3D object. Appendix 10 visually represents this description, which perhaps may be easier to understand.

As mentioned before OBJX files are loaded into the Engine to become Model3D objects. These objects have a heavy dependency on storing memory addresses on different pieces of data loaded from the OBJX file (referencing). In some of the first attempts at developing this model loader, memory referencing was being done in the middle of the OBJX file load. As different pieces of data where shuffled in Array (std::vector) reallocation these memory references became invalid, resulting in the program to crash or produce unexpected models.

These memory referencing problem shave been resolved by revising the OBJX/Model3D structures and ensuring that no memory referencing is done until all data from the OBJX file has been read and the elements needed to make up the Model3D object are set up.

8.5.2 MTLX Format (R6, R36)

As mentioned previously, Each OBJX model file has a corresponding MTLX file that holds details on the OBJX model’s material properties. As with Materials loaded from a MTL file, MTLX materials are saved in the same bank of Materials in the ResourceManager class when loaded into the Engine.

When a Material is assigned to a MeshObject, each vertex in the MeshObject saves the assigned Material’s diffuse information. This vertex colour property is then used in rendering for coloring the model when lighting is disables in the Engine. Figure 50 shows the function to save the Diffuse Material colour properties to each Vertex in the MeshObject. Figure 51 demonstrates the same model being drawn with lighting using its Material Properties (Left) and without lighting using its Vertex Colour Properties (Right). This demonstrates Requirement R6.

```cpp
Vector3f tmp_colour;
//Store MeshObject Material Colour Info
for(int i = 0; i < vertexs.size(); i++)
{
    tmp_colour = material->diffuse;
    vertexs[i].colour.r = tmp_colour.x;
    vertexs[i].colour.g = tmp_colour.y;
    vertexs[i].colour.b = tmp_colour.z;
    vertexs[i].colour.a = 0.0;
}
```

Figure 50: Function to Save Material's Diffuse Colour to MeshObject Vertexes
There were some initial problems in the development of the OBJX and MTLX loaders and savers. Each MeshObject expects to be given a material. In earlier implementations, the program would crash if a MeshObject could not find its corresponding material in the ResourceManager’s bank of materials. To fix this, should a MeshObject not find its material in the ResourceManager, the ResourceManager will automatically create a new Material for this MeshObject and add it to its Array of materials and return the memory address of this newly added Material to the MeshObject.
8.6 Model Viewer (R45 – R49)

The Model Viewer Application has been developed to test the OBJX and MTLX files exported from the Engine Editor. It allows for these hierarchical models to be tested on their target platform. The Model Viewer shows how these models will appear on these devices.

As mentioned before there have been two different Viewer Applications developed. The first is targeted at Windows and Mac OS X Systems. It uses SDL as a front-end windowing system for the Engine, handling input and texture loading. It is written entirely in C++.

There are currently some problems with SDL on the Apple’s OS X Operating System. Apple’s OS X is currently on version 10.6.7, with 10.7 to be released in the very near future. The latest version of SDL is SDL 1.2.14. SDL 1.2.14 is quite old and is targeted on the Mac at OS X version 10.4. OS X Version 10.4’s last major update to 10.4.11 occurred in November 2007.

Since OS X 10.4 Apple has made some major changes in its Operating System’s architecture. The main changes being a focus on 64-bit processors and GPU processing. SDL is targeted at 32 bit development. XCode natively builds for the current release of OS X the development machine is running. The recently released XCode 4 has no obvious backward support for OSX 10.4. As a result of this when compiling under XCode 4, and releases of XCode 3 without the installed support for OSX 10.4, will result in a number of linker errors when compiling the viewer project. On some of the newer releases of XCode 3, you have to specifically tell XCode to build for OS X 10.5 for this project to compile correctly. Figure 52 shows this.

The second Model Viewer Application is an iOS viewer. It displays OBJX models on iOS devices as they will appear in a game developed using the engine. Apple’s iOS is based closely to Apple’s OS X. It contains a stripped down implementation of OS X’s functionality, much in the same that J2ME contains a stripped down version of J2SE’s provided functionality. Earlier iOS devices only support OpenGL ES 1.1, however more recent iOS devices support OpenGL ES 2.0. As this project has been developed sing OpenGL ES 1.1 it has no problems running on iOS devices.

The Editor and the SDL viewer both contain the entire OpenGL library, as well as the GLUT library. The iOS Viewer does not support this full functionality because of its restrictions to only use calls available in GLES 1.1. The core rendering is all done by the engine using this standard however, for debugging and testing items such as Bounding Spheres, World Axis and other items that would not be visible in the final release of a game the iOS viewer will not render.
8.7 Gameplay Demo (R50 – R53)

The Gameplay Demo has been built using the Engine framework. It demonstrates a hierarchal animated model being controlled by the user. The model animation changes as the user moves about the game world and carries out different actions. The demo also demonstrates the user colliding and stopping when they try and walk into another object in the game world.
9 Testing

This section details the testing of this project to ensure that it meets the planned set of requirements. Like in the Requirements, the testing section is divided into seven sections - Models and Materials, Animations Structure, Math, Physics and Collision Detection, Model Editor and Tester, Model Viewer and a Gameplay Demo. Each Test has a Test Number, a Description, a Requirement Met and a whether its Passed or Failed.

9.1 Models and Materials

<table>
<thead>
<tr>
<th>Test No</th>
<th>Description</th>
<th>Req Met</th>
<th>Pass/Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>OBJ Models Load Correctly</td>
<td>R1</td>
<td>Pass</td>
</tr>
<tr>
<td>T2</td>
<td>OBJ Models Exported from 3DS Max and Blender Load Correctly</td>
<td>R2</td>
<td>Pass</td>
</tr>
<tr>
<td>T3</td>
<td>3DS Files Load Correctly</td>
<td>R3</td>
<td>See 10.1</td>
</tr>
<tr>
<td>T4</td>
<td>OBJ File’s corresponding MTL file loads correctly</td>
<td>R4</td>
<td>Pass</td>
</tr>
<tr>
<td>T5</td>
<td>Textures referenced in the MTL file are loaded with MTL file</td>
<td>R5</td>
<td>Pass</td>
</tr>
<tr>
<td>T6</td>
<td>Model’s Material colour remains the same when lighting disabled</td>
<td>R6</td>
<td>Pass</td>
</tr>
<tr>
<td>T7</td>
<td>Model Tex Co-ordinates Load Correctly</td>
<td>R7</td>
<td>Pass</td>
</tr>
<tr>
<td>T8</td>
<td>Model Normals Load Correctly</td>
<td>R8</td>
<td>Pass</td>
</tr>
<tr>
<td>T9</td>
<td>Model Structure devised to link together smaller models to make a single, larger, more complex Model</td>
<td>R9</td>
<td>Pass</td>
</tr>
</tbody>
</table>

9.2 Animation System

<table>
<thead>
<tr>
<th>Test No</th>
<th>Description</th>
<th>Req met</th>
<th>Pass/Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>T10</td>
<td>Animation Structure holds Model Animation Data</td>
<td>R10</td>
<td>Pass</td>
</tr>
<tr>
<td>T11</td>
<td>In a hierarchical Model, Rotations Applied to Parent Models also effect Models at Lower Levels</td>
<td>R11</td>
<td>Pass</td>
</tr>
<tr>
<td>T12</td>
<td>Animations animate using Linear Interpolation</td>
<td>R12</td>
<td>Pass</td>
</tr>
<tr>
<td>T13</td>
<td>Animations occur on X, Y and Z axis simultaneously</td>
<td>R13</td>
<td>Pass</td>
</tr>
<tr>
<td>T14</td>
<td>Animations contain a number of rotation points</td>
<td>R14</td>
<td>Pass</td>
</tr>
<tr>
<td>T15</td>
<td>Animations can be one of Stop, Reverse or Loop</td>
<td>R15</td>
<td>Pass</td>
</tr>
<tr>
<td>T16</td>
<td>Stop Animations Stop once complete</td>
<td>R16</td>
<td>Pass</td>
</tr>
<tr>
<td>T17</td>
<td>Reverse Animations repeat backward through Animation angles when complete</td>
<td>R17</td>
<td>Pass</td>
</tr>
<tr>
<td>T18</td>
<td>Loop Animations repeat from beginning when Animation completes</td>
<td>R18</td>
<td>Pass</td>
</tr>
<tr>
<td>T19</td>
<td>Models Morph correctly</td>
<td>R19</td>
<td>See 10.2</td>
</tr>
</tbody>
</table>

9.3 Math, Physics and Collision Detection

<table>
<thead>
<tr>
<th>Test No</th>
<th>Description</th>
<th>Req met</th>
<th>Pass/Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>T20</td>
<td>Calculates Vector Magnitude</td>
<td>R20</td>
<td>Pass</td>
</tr>
<tr>
<td>T21</td>
<td>Calculates Dot Product of Two Vectors</td>
<td>R21</td>
<td>Pass</td>
</tr>
<tr>
<td>T22</td>
<td>Calculates Surface Normal</td>
<td>R22</td>
<td>Pass</td>
</tr>
<tr>
<td>T23</td>
<td>Normalizes Vectors</td>
<td>R23</td>
<td>Pass</td>
</tr>
<tr>
<td>T24</td>
<td>Generates Model’s Collision Spheres</td>
<td>R24</td>
<td>Pass</td>
</tr>
<tr>
<td>T25</td>
<td>Calculates normals used for Collision mesh Collision Detection</td>
<td>R25</td>
<td>See 10.3</td>
</tr>
<tr>
<td>T26</td>
<td>Detects Sphere – Sphere collisions</td>
<td>R26</td>
<td>Pass</td>
</tr>
<tr>
<td>T27</td>
<td>Detects Sphere – Collision Mesh collisions</td>
<td>R27</td>
<td>Pass</td>
</tr>
<tr>
<td>T28</td>
<td>Collision Sphere responds to collision with Collision Mesh</td>
<td>R28</td>
<td>See 10.3</td>
</tr>
</tbody>
</table>
## 9.4 Rendering System

<table>
<thead>
<tr>
<th>Test No</th>
<th>Description</th>
<th>Req met</th>
<th>Pass/Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>T29</td>
<td>Rendering System kept Independent of rest of program</td>
<td>R29</td>
<td>Pass</td>
</tr>
<tr>
<td>T30</td>
<td>All Rendering done using VBOs and Vertex Arrays</td>
<td>R30</td>
<td>Pass</td>
</tr>
<tr>
<td>T31</td>
<td>Only calls available in OpenGL 1.1 used in Viewer and Gameplay Demo</td>
<td>R31</td>
<td>Pass</td>
</tr>
<tr>
<td>T32</td>
<td>3D Models rendered correctly</td>
<td>R32</td>
<td>Pass</td>
</tr>
<tr>
<td>T33</td>
<td>Material Info displayed correctly</td>
<td>R33</td>
<td>Pass</td>
</tr>
<tr>
<td>T34</td>
<td>Textures display and are mapped correctly</td>
<td>R34</td>
<td>Pass</td>
</tr>
<tr>
<td>T35</td>
<td>Model Normals displayed correctly under lighting</td>
<td>R35</td>
<td>Pass</td>
</tr>
</tbody>
</table>

## 9.5 Model Editor

<table>
<thead>
<tr>
<th>Test No</th>
<th>Description</th>
<th>Req met</th>
<th>Pass/Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>T36</td>
<td>Can Manipulate Model and Material properties</td>
<td>R36</td>
<td>Pass</td>
</tr>
<tr>
<td>T37</td>
<td>Can Link together single Models to make larger more complex Models</td>
<td>R37</td>
<td>Pass</td>
</tr>
<tr>
<td>T38</td>
<td>User can create animations for these complex models</td>
<td>R38</td>
<td>Pass</td>
</tr>
<tr>
<td>T39</td>
<td>Complex Model data can be exported and loaded</td>
<td>R39</td>
<td>Pass</td>
</tr>
<tr>
<td>T40</td>
<td>Model Animation data can be exported and loaded</td>
<td>R40</td>
<td>Pass</td>
</tr>
<tr>
<td>T41</td>
<td>Can toggle drawing of Bounding Spheres</td>
<td>R41</td>
<td>Pass</td>
</tr>
<tr>
<td>T42</td>
<td>User can Scale Models</td>
<td>R42</td>
<td>Pass</td>
</tr>
<tr>
<td>T43</td>
<td>User can Rotate Models</td>
<td>R43</td>
<td>Pass</td>
</tr>
<tr>
<td>T44</td>
<td>User can Translate Models</td>
<td>R44</td>
<td>Pass</td>
</tr>
</tbody>
</table>

## 9.6 Model Viewer

<table>
<thead>
<tr>
<th>Test No</th>
<th>Description</th>
<th>Req met</th>
<th>Pass/Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>T45</td>
<td>Must display model data exported by Editor</td>
<td>R36</td>
<td>Pass</td>
</tr>
<tr>
<td>T46</td>
<td>Must animate models using data exported from Editor</td>
<td>R37</td>
<td>Pass</td>
</tr>
<tr>
<td>T47</td>
<td>SDL Viewer can toggle rendering of a Model’s Bounding Sphere</td>
<td>R38</td>
<td>Pass</td>
</tr>
<tr>
<td>T48</td>
<td>SDL Viewer changes current Model Animation on Key Press</td>
<td>R39</td>
<td>Pass</td>
</tr>
<tr>
<td>T49</td>
<td>iOS Viewer changes current Model animation on Touch</td>
<td>R40</td>
<td>Pass</td>
</tr>
</tbody>
</table>

## 9.7 Gameplay Demo

<table>
<thead>
<tr>
<th>Test No</th>
<th>Description</th>
<th>Req met</th>
<th>Pass/Fail</th>
</tr>
</thead>
<tbody>
<tr>
<td>T50</td>
<td>Player controls using Game Pad</td>
<td>R36</td>
<td>Pass</td>
</tr>
<tr>
<td>T51</td>
<td>Player can jump and fall due to gravity</td>
<td>R37</td>
<td>Pass</td>
</tr>
<tr>
<td>T52</td>
<td>Player can collide with other models in Game world</td>
<td>R38</td>
<td>Pass</td>
</tr>
<tr>
<td>T53</td>
<td>Model animations change depending on what Player is doing – Idle, Running, Jumping</td>
<td>R39</td>
<td>Pass</td>
</tr>
<tr>
<td>T54</td>
<td>iOS Viewer changes current Model animation on Touch</td>
<td>R40</td>
<td>Pass</td>
</tr>
</tbody>
</table>
10 Discarded Functionality

Due to time restraints and problems encountered during development, some of the planned functionality of this project had to be discarded. This section details what pieces of functionality were not implemented.

10.1 3DS Model Support

Initially the Open Source LIB3DS library was to be integrated into the Engine to support the loading of 3DS model files. As the UI became more complex, to the point where the user was able to interact and set properties for Models loaded into the Ending Editor, it was decided that to then incorporate the LIB3DS library would be too complex for all the minimal functionality that it would add to the project.

10.2 Morph Targets

The steep learning curve brought by the introduction of a full UI and learning pieces of Objective C meant that an element in the project had to be dropped. It was decided that Morph Targets, whilst pleasing to look at, brought fairly little added depth to the project and thus was discarded from the project.

10.3 Sphere – Collision Mesh Collision Detection

An attempt was made at implementing this functionality. This is discussed in 8.3.3. It was felt that the time it would have taken to fully implement a Collision Mesh could have been better spent working on other areas of the project.
11 Conclusion

Over the last year at college the ‘Simple 3D Game Engine’ was developed. The Engine is targeted primarily at iOS devices, however it will also work on Windows and Mac OS X Operating Systems. The Engine has been designed to be easily ported to target other devices.

This project has achieved the following outputs;

- An Engine Model Editor
- An Engine Model Viewer
- The Engine Framework
- A Game Demo

In the Model Editor a user can load, view, manipulate and animate 3D Models. This data can then be exported to disk in a proprietary file format developed specifically for this Engine.

The Model Viewer allows the user to view models saved in this proprietary on a desired platform. Implementations of the Viewer have been developed for Windows, Mac OS X and iOS Systems.

The Engine Framework is what a developer would use as their development environment to develop a Simple 3D Game. The Game Demo was produced using the framework. It shows model animations created in the Editor being used in a game environment and demonstrates the capability of the Engine’s built in Collision Detection.

This Engine Framework, along with its corresponding Model Editor and Model Viewer has the following functionality:

- Load and Display OBJ Models along with their Material and Texture data
- File Parsers have been developed in an attempt to ensure that the Engine will not try and read data from a file it cannot handle
- The User can interact with these 3D Models
- A Camera class has been developed that allows for the movement around these models in a 3D scene and allows the user to Zoom in and out of the scene
- The Engine Editor allow the user to create complex, animated 3D models from smaller models loaded from OBJ files.
- The Engine Editor allows for the manipulation of these models Material Properties
- The Engine Editor has a full User Interface developed in Objective C
- Data can be exported from the Engine in a custom file format (OBJX and MTLX)
- A Viewer Application has been developed that allows for the user to view the output of the Engine Editor on Windows, Mac OS X and iOS systems.
- Bounding Spheres are calculated automatically as a model is loaded
- The Engine can perform simple Sphere – Sphere collision detection in 3D
- An Attempt has been made at implementing Bounding Sphere - Collision Mesh collision detection
- A Math Library class has been developed that holds a number of mathematical structure and functions needed by the Engine
- The Engine can read input from Game Pads and iOS devices
- A Gameplay Demo has been developed that shows a user-controlled model traveling across a plane
- In the Gameplay Demo the current model animation changes depending of what the user is doing –Idle, Running, Jumping

In review, the initial functional specification for the Engine was too ambitious. Several functional aspects had to be discarded or only partially implemented. These are discussed in 10 Discarded Functionality.

A major hurdle in the development of this project turned out to be the development of the User Interface for the Engine Editor. The interface was developed using XCode’s Interface Builder. Interface Builder uses Objective C code to interact with an application. This was my first time using Interface Builder and Objective C. With reference to some of Apple’s ADC (Apple Developer
Connection) documentation I was able get familiar with the Interface Builder classes and learn enough Objective C to create this interface. Additionally, there were the issues with the design of the UI. 3D software has to deliver some quote abstract functionality to the user, deciding on visual solutions for this functionality took up considerably more time than was initially planned.

Another technical problem encountered during the development of the User Interface was interfacing Objective C code with the Game Engine’s C++ Code. To allow these different languages to interact with each other, Objective C header files had to end with a .hpp extension as opposed to .h extension. Objective C code files had to end with the extension .mm as opposed to the standard .m. Finally C++ code files too had to use the .mm extension as opposed to the standard .cpp. The .mm and .hpp extensions inform the compiler that it is compiling Objective C and C++ code, this allows the two to interact.

When considering which 3D file format to use several were evaluated in terms of their flexibility and capacity to store 3D model and animation data. There were pro’s and con’s to each file type considered. Generally, they are very poorly documented without a set standard for storing data. As a result, different application vendors add in their own additional data when saving models in these formats making them unreadable in different modeling environments. Effectively, many vendors subvert the ‘standardised’ file type to something more proprietary.

For example, some vendors add additional data to the standardized OBJ file format. This required a pre-loader to be written to validate an OBJ file before it is loaded. This pre-loader scans the file and removes any data that it knows the loader will not be able to handle. Further details on this pre-loader can be seen in section 8.1.2 OBJ File Loader (R1, R2, R7, R8, R32 – R35).

Another problem with the available file formats is that their entire internal file structure changes through different revisions of the model format. Typically file types are not backwards compatible, meaning a new loader will have to be written for each iteration of the model format. Alias’s FBX file format has been completely changed between the 2010 release and the first 2011 release of their publicly available FBX SDK. Function calls have been completely renamed, with some taking different arguments. The result is that for a developer to use the updated FBX file formats they have to completely rewrite the entire loader code, with reference to very little documentation on the changes between code iterations.

These problems above have introduced a major hurdle for amateur developers looking to create a game on a budget. They may not have the money to license a major game engine with built in model loading functionality, or contract an experienced programmer to write a model loader for them. Generally in major studios they use a customized file format for storing 3D models, designed specifically for the target platform and game genre to avoid potential problem.

This project presents a sample solution for these amateur developers. Using this project, a developer may create their 3D character in a modeling package such as Maya, 3D Studio Max or Blender. The limbs of this model can then exported and loaded into the developed Model Editor Application to be linked together to create a hierarchical model. This hierarchical model can then be animated and a bounding sphere for the model created. This data can be exported as an OBJX model and using the developed Engine the model can be used in a 3D game.

The processing involved in animating these OBJX models is also much less complex than would be needed for a rigged 3D model with skeletal information. These simplistic animations and requirements for less processing power make the OBJX file format an option for developing on a mobile device.

There are a number of optimizations that could be made to the OBJX file format such as indexing vertices as they are saved to disc and saving the file in binary format as opposed to ASCII text format. These modifications would result on OBJX files requiring smaller space for storage further optimizing the file format for use on a mobile device. As an unexpected output of this project the OBJX file type offers possibilities for Masters level research and a solution for amateur 3D developers and programmers.
References


Ericson, C, (2005), *Real Time Collision Detection*, Morgan Kaufmann


Sabbarton, R. [http://www.fullonsoftware.co.uk/snippets/content/Math_-_Calculating_Face_Normals.pdf](http://www.fullonsoftware.co.uk/snippets/content/Math_-_Calculating_Face_Normals.pdf) (07/11/2010)

Dorn, O (2010), University of Manchester, Lecture 1 – Interpolation, [http://www.maths.manchester.ac.uk/service/MATH29641/2Q1Lecture1.pdf](http://www.maths.manchester.ac.uk/service/MATH29641/2Q1Lecture1.pdf) (09/11/2010)
12 Bibliography


SDL Project. (2010). http://www.libsdl.org/ (01/12/10)

Appendix 1: 3D Cube with Corresponding OBJ and MTL Files
Appendix 2: Diagram Showing Animation Structure

Original image Taken from - http://www.the-eleventh.com/node/57
Appendix 3: Entity Relationship Diagram showing Relationship between Model3D, Mesh3D and MeshObjects
Appendix 4: OBJ and MTL Loading Process Diagram
Appendix 5: Engine Architecture

OS - Mac OSX / iOS / Windows
QuickTime™ and a decompressor are needed to see this picture.

Appendix 6: Initial Engine Editor Interface Design
Below details the functionality of the numbered elements in **Appendix 6: Initial Engine Editor Interface Design.**

1) Import OBJX File  
2) Import OBJ File  
3) Import MTL File  
4) Import MTLX File  
5) Save OBJX and MTLX  
6) Set Parent of Currently Selected Mesh  
7) Toggle Bounding Sphere Renderer On/Off  
8) Set Previous Animation  
9) Set Next Animation  
10) Play Animation  
11) Pause Animation  
12) Changes the Currently Selected Mesh to its Parent  
13) Changes the Currently Selected Mesh to its First Child  
14) Selects the Previous Mesh on the Current Mesh’s Level  
15) Selects the Next Mesh on the Current Mesh’s Level  
16) Translate the Current Mesh  
17) Rotate the Current Mesh  
18) Scale the Current Mesh  
19) Any additional buttons needed for the Editor will be placed here  
20) Displays info and allows for interaction on the currently selected Model and Mesh. Info includes info such as Model and Mesh names, position, rotation, scales, and child meshes.  
21) Displays info and allows for interaction on the currently selected MeshObject. This info includes info such as the MeshObjects available for selection, the current MeshObject’s name and material colour. The Material can be modified and info is saved in an MTLX file when exported.  
22) Displays information and allows for the creation and modification of the Current Model’s animations  
23) This is where the OpenGL Rendering is done. What is displayed in this widow is similar to how the scene will appear in the Engine Viewer App.

**Appendix 7: Initial Engine Editor Functionality**
Appendix 8: Final Engine Editor Interface and Functionality
Appendix 9: ResourceManager and MeshObject Relationship
Appendix 10: Model3D Structure
14 Code
Development of a Simple 3D Game Engine