EECE 478 – Computer Graphics
3D Game Design – Final Report
Futuristic Shooter

Submitted: April-14-08

Rex Chang, 81535049
Martin Ma, 13778030
Derek Tam, 63491047
Sarah Wu, 62473046

Abstract

This report details the nature, the design, and the implementation methods of our EECE 478 game project, which consists of a game world where two players can move around and shoot each other with projectile weapons. The game design is explained with the intention to allow the reader to know why and how the techniques are chosen and utilized in the final implementation. The game demonstrates the basic project requirements, while exploring various advanced features that can enhance the gameplay.

Table of Contents

Abstract 2

Table of Contents 3

List of Figures 4

List of Abbreviations 5

1 Introduction 6

2 Game Description 8

3 Implementation Methods 9

3.1 User Interface 9

3.1.1 Menu 9

3.1.2 Sound 10

3.2 Collision Detection 12

3.2.1 Collision of Two Spherical Models 12

3.2.2 Collision of a Spherical Model and a Bounded Plane 13

3.3 Game Mechanics 15

3.3.1 Movement along the Board 15

3.3.2 Firing an Arrow 16

3.3.3 Health Points and Magic Points Status 17

3.4 Three-Dimensional Models 18

3.4.1 3DS Loaders 19

3.4.2 Terrain Generation 20

3.4.3 Frustum Culling 21

3.5 Animation 25

4 Possible Improvements 29

5 Flow Chart 30

List of Figures

Figure 1 Right-Click Game Menu 9
Figure 2 Collision of Two Spheres 12
Figure 3 Collision Between a Sphere and a Bounded Plane 14
Figure 4 Face Directions before Movement 16
Figure 5 Face Directions after Movement 16
Figure 8: Stick Figure - Front View 18
Figure 9: Stick Figure - Back View 18
Figure 11 A Snowfield Terrain Using 3ds Max Studio 20
Figure 12 Terrain with Bitmap 1 21
Figure 13 Terrain with Bitmap 2 21
Figure 14 Frustum Culling – Example Terrain 22
Figure 15 Before Frustum Culling 23
Figure 16 Area Inside the View Frustum 23

List of Abbreviations

GLUT	OpenGL Utility Toolkit
HP	Health Point
MP	Magic points
SDL_Mixer	Simple DirectMedia Layer Mixer
GLEW	OpenGL Extension Wrangler (Library)

1 Introduction

This report will show the nature of our game project, the design details, and the various features we implemented, along with the design decisions and less successful attempts when exploring more advanced features. The intention is to show the reader what our team had accomplished and how we accomplished them.

Our game basically includes a 3D world with a game field, on which two players can shoot each other with projectile weapons (mainly a bow and arrows). The two players can move within the game field to dodge each other's attacks. There is an HP (Health Point) bar and an MP (Magic Point) bar for each player. If the HP of one player drops to zero, the other player is considered to have won. The MP governs the arrow firing ability of the player.

Our game had met the basic requirements in the project details, which include: (a) 3D viewing and objects with a moving camera; (b) lighting and material variations; and (c) texture mapping. There are advanced features we attempted implementing, including the following:

(a) Sound effects
(b) Collision detection
(c) Physical simulation
(d) Terrain
(e) View frustum culling
(f) Key frame animation

These features are explained in more details in the corresponding sections.

We believe there are many more elements and features we can add to the game to make the game more entertaining, but for the time frame given to complete the project, we are unable to include all of them. These possible improvements are briefly described in section 4 of this report.

2 Game Description

As explained in the introduction, the game world contains a game field on which two character models can move around and shoot projectiles. The game allows two human players to take control of the in-game characters and fight against each other to see who may defeat the opponent first. The arrows can be fired at the expense of some MP, which regenerates over time. When MP drops to zero, no more arrows can be fired until it regenerates. If the arrow hits the opponent, the HP of the opponent will decrease based on the power of the fired arrow. When HP drops to zero, the character is considered dead, and the other player has won. Each of the characters can be controlled by keyboard keys or the joystick (mapped to these keys). The character can move (up, down, left, right), change heading (left or right), change projectile angle (up or down), or modify firing power (1 button). The game menu can be accessed with the right click of the mouse and it can start, pause, or quit the game.

3 Implementation Methods

There are many different aspects of the implementation of this game that must be described to fully understand how this game was produced. This game involves creating a simple user interface, to numerous game mechanics algorithms, to various types of three-dimensional modelling, to several different animation techniques. The following sections with examine all the features of the game in further detail.

3.1 User Interface

For the user interface of this game, a simple menu system was used. Also, while playing the game, specific sounds are associated with different actions. The following sections will describe how the menu was created and how sound was implemented for this project.

3.1.1 Menu

The menu system is build using the OpenGL Utility Toolkit (GLUT) menu functions. GLUT provides a simple mechanism for creating pop-up menus. Each menu item is associated with a specified procedure. Our menu system contains three different selections, Start Game, Pause and Quit Game. The menu can be brought up when the right button of the mouse is clicked.

Figure 1 Right-Click Game Menu

A menu is first created by calling int glutCreateMenu(void (*func)(int value)). This creates a simple menu in which you can add numerous entries for different tasks. Items are added into this menu by calling void glutAddMenuEntry(char* name, int value). In our project, as shown in Figure 1, three different choice selections had been added. To associate the right mouse button with the menu, the button must be assigned to this menu by calling void glutAttachMenu(int button). A menu event function is attached to the menu so that when a selection has been made, the correct action will be executed.

3.1.2 Sound

The sound used for this project is implemented using the Simple DirectMedia Layer Mixer (SDL_MIXER) library. SDL_mixer is a simple multi-channel audio mixer that supports 8 channels of 16 bit stereo audio, plus a single channel of music, which includes the popular MikMod MOD, Timidity MIDI, Ogg Vorbis, and SMPEG MP3 libraries.

This library divides sound into two categories: music and sound effect. Music constitutes as the sound played on the background and it is not event-driven. The music file needs to either be an mp3 or a MIDI extension. Sound effect is event-driven, thus working closely with mechanics of the game. For example, a sound is played when an arrow is released or when an arrow hits an object. The sound effect file must be a WAV extension.

Here are the steps in using SDL_Mixer in our sound system:

- Open up the audio device
- Load sound effect files into memory
- Play them when necessary
- Clean up

The SDL_MIXER has several music player state functions to control the status of the music, such as int Mix_PlayingMusic() and int Mix_PausedMusic() to play and pause the music. Music player control functions were utilized to control the music playing during the game to play, pause, and resume the music.

The SDL_MIXER has a feature on the sound effect which enables multiple sound effects played at the same time. Multiple sound effects are enabled through the utilization of numerous channels. Mix_PlayChannel(int *channel*, Mix_Chunk **chunk*, int *loops*) is used to control which channel the sound is played on. -1 is passed to *channel* for the first free unreserved channel and 0 is passed to *loops* to play the sound sample only once.

3.2 Collision Detection

Collision detection is required in many aspects of this game to make it perform as it should. A player cannot run through another player nor can it run through the walls of the obstacles. When an arrow hits a player or an obstacle, it must be detected so that the proper action can take place. Without collision detection, the entire game outlined in the game description section would not work. Two types of collision detection were used for this game, collision between two spheres and collision between a sphere and a plane.

3.2.1 Collision of Two Spherical Models

The first type of collision detection implemented for this game was the collision of two spheres. This was used to simulate the collision of two players since each player has a cylindrical body and will only collide into another player at each other's side. To detect a collision of two spherical bodies, the distance between the centres of the two bodies was calculated. If the distance, d, was greater than the sum of the radii of the bodies ($r_1 + r_2$), then they have not collided (

Figure 2). If the distance is equal to the sum of the two radii, then the two bodies are just touching; if the distance is less than the sum of the two radii, then the two bodies have begun to pass through each other.

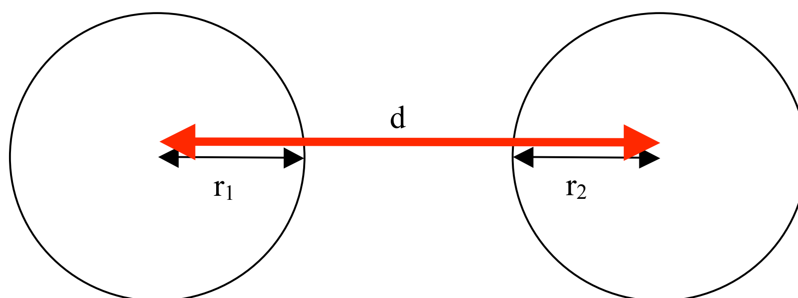


Figure 2 Collision of Two Spheres

3.2.2 Collision of a Spherical Model and a Bounded Plane

The second type of collision detection implement was the collision of a sphere and a plane. This simulates the collision of the body of a player and an obstacle. This second type involved more math than the first type. First the normal of the plane must be determined. This is accomplished by finding the cross product of two vectors on the plane. Vectors on the plane are simply vectors created by points are the plane, of which four are already known since they are the properties that define the plane. Secondly, since a point of the plane is known, a vector from a point on the plane to the center of the body is projected onto the normal of the plane. The length of this projection (Equation 1) is then equivalent to the distance between the center of the body and the plane itself.

$$\left\| \frac{(\mathrm{c} - \mathrm{Q}) \cdot n}{n \cdot n} n \right\|$$

Equation 1 Length of Projection of Line cQ onto *n*

Like the first collision type, if the distance calculated is greater than the radius of the body, then a collision has not occurred. If the distance is equal, then a collision might have occurred. To determine if a collision has indeed occurred, the collision point, P (Equation 2), must be within the boundary of the plane, which is set by the properties of the obstacle.

$$\mathrm{P} = \mathrm{c} + \frac{\mathrm{r}}{\|n\|} n$$

Equation 2 Point of Collision

Figure 3 helps to show how to determine if the collision point is within the boundary. To simplify the calculations for this game, the y values of all the points that define the plane must be equivalent ($Q_y = R_y = S_y = T_y$) so that the height the plane is uniform throughout. This means

values T_y and S_y will be set to zero (the ground of the game) and Q_y and R_y will be set to the same value, the height of the plane. Another predefined property of the plane is that it must be perpendicular to the ground. This means that values Q_x and T_x, R_x and S_x, Q_z and T_z, and R_z and S_z will be equivalent to each other. With these values set into place, it is easy to verify if the collision point is within the boundary.

The collision point, P, is within the defined obstacle boundary if:

- $Q_x \leq P_x \leq R_x$
- and $Q_z \leq P_z \leq R_z$
- and $0 \leq P_y \leq R_y$

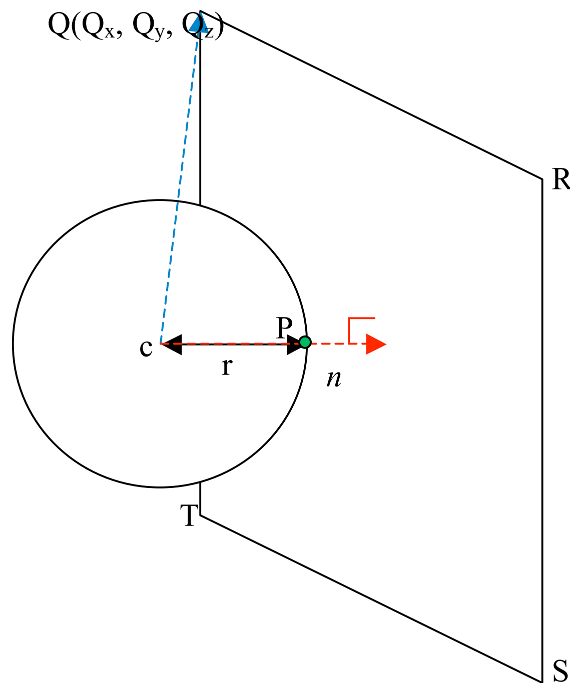


Figure 3 Collision Between a Sphere and a Bounded Plane

3.3 Game Mechanics

There are many components to the mechanics of this game. Different elements enhance different aspects of the game play. These game mechanic components include:

- movement along the board
- firing an arrow
- hit point (HP) status
- magic points (MP) status

Since a number of the mechanics of this game have yet to be implemented, parts of this section will examine the methods of how an element *will be* implemented instead of how it was done.

3.3.1 Movement along the Board

A player may move in four directions on the board, forward, backward, left, and right. This is achieved by pressing four different keyboard keys. If an obstacle or another player is its way, it will be unable to move in that direction. To accomplish this, four flags, one for each direction, are set for each player. If a collision has occurred, the flag will be set and the player will be unable to move in the direction in which the collision has taken place. When the player moves in such a way that it is no longer colliding with an object, the flag is unset.

Originally, to increase the ease with which a player fires an arrow (next section), when either players move, both players will automatically rotate that they are always facing the other player (Figure 4 and Figure 5). This has been changed so that a player may rotate towards whichever direction it prefers to allow for greater manipulability of the player models.

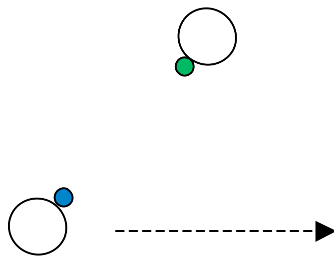


Figure 4 Face Directions before Movement

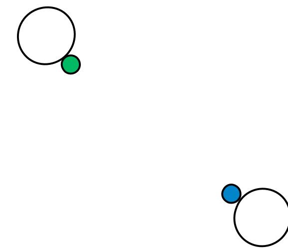


Figure 5 Face Directions after Movement

3.3.2 Firing an Arrow

An arrow in this game contains numerous properties. There are two types of arrows that cause different reactions. The standard arrow just decreases a player's HP, whereas the special arrow will produce a knock-back effect, which causes a player to lose HP and be forced to move back a few steps. Different types of arrows require different MP costs; this enhances the game play because every time an arrow is to be fired, the player will need to determine which arrow will produce the best affect.

Besides the type of arrow affecting the cost of MP, the strength of the shot also has a different cost. A player can pull the arrow longer and, therefore, fire to a greater distance, but this will require a greater amount of MP. The strength of an arrow is displayed on a power bar and is determined by repeatedly pressing the "pull" key. At the desired strength, the player can press "fire" to fire the arrow.

The firing of an arrow in this game is modelled using the projectile motion formulas. The x and z distance of the arrow is determined with Equations 3 and the y is determined with Equation 4. A player may rotate the angle of the arrow to the specific y direction it wants to aim at.

$$d_x = v_x t \text{ and } d_z = v_z t$$

Equation 3 x and z Distance

$$d_y = v_y t + \frac{1}{2}at^2$$

Equation 4 y Distance

3.3.3 Health Points and Magic Points Status

Health points (HP) is used to keep track of the players' health status. When a player is hit by an arrow, its HP will decrease. When a player's HP reaches zero, it is dead and the other player has won. HP is stored in a variable that is updated every time a player is hit by an arrow.

A player's magic points (MP) is what it costs a player to fire an arrow. Magic points decreases every time an arrow is fired and regenerates as time passes. A player cannot attack if it does not have sufficient MP saved up to fire an arrow, in this case, a player must wait for MP to regenerate enough to afford the cost of an arrow. MP is also stored in a variable that is updated every time an arrow is fired or after a specific regeneration interval has passed.

HP and MP status bars are displayed above each player's head. They are drawn as rectangles in red and blue colours for HP and MP, respectively. These bars are placed above the players' head because with the camera of the game moving around, it would be difficult to draw static bars in the frame. Furthermore, placing the hp/mp bars above the models' heads is more convenient for players because they will not be distracted from the gameplay by looking away from the board to view various statuses.

3.4 Three-Dimensional Models

We wanted to make our 3D game look more exciting, so we created our own 3D models with programs such as 3DS max studio to use in our game. However, loading the 3D models that we created in 3DSmax to our OpenGL game is a lot more difficult than we imagined. It requires a thorough understanding of the 3DS chunk structure and writing a 3DS loader to load the models into our game. Figures 8 and 9 show two different views of the static bowmen model that we made in 3DSmax studio.

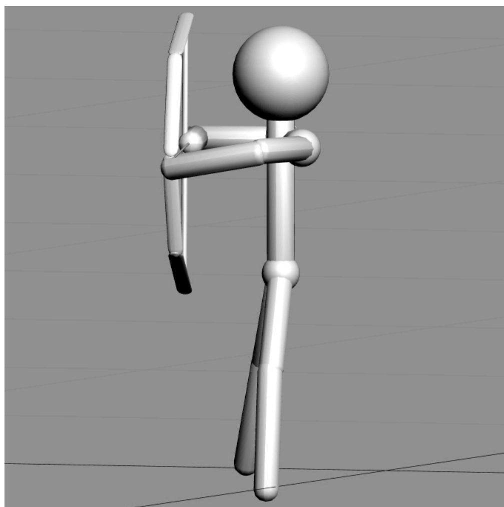


Figure 6: Stick Figure - Front View

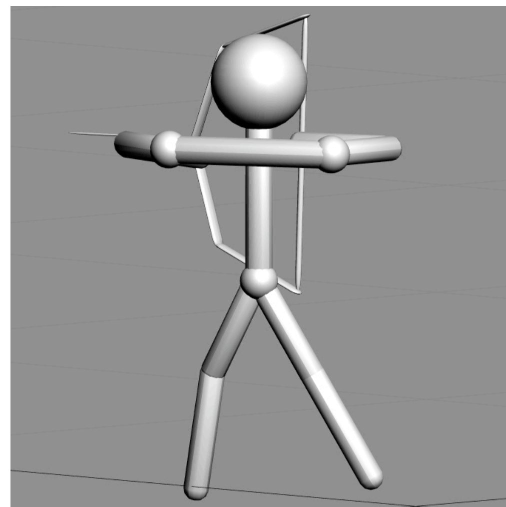


Figure 7: Stick Figure - Back View

Since we were short on time, we tried to locate 3ds loaders that were readily available on the net for use in our project.

We made many attempts to find 3ds loaders for loading our models into the game most of which gave us quite a lot of problems such as missing or outdated header files (glaux.h, glew.h…) or incompatible with our compiler software. We tried quite a few 3ds loaders, most of which did not work entirely the way we expected them to behave. The only two 3ds loaders that successfully loaded our code were: “SpaceSimulator” and “Anim8or”, both of which have their strengths and shortcomings.

3.4.1 3DS Loaders

Lib3ds	"Lib3ds" is one of the most commonly used open source 3ds loaders on the web which is geared for those who are unfamiliar with the 3ds chunk structure. However, our compiler setup seemed to be missing some components that were required to run the source code for this loader. Thus, we were unable to use this loader.
SpaceSimulator	"SpaceSimulator" is a 3d game made with the use of a self-written 3ds loader. But it does not contain some of the 3ds chunks that we needed in our game such as lighting, materials and animation. However, it does provide texture mapping on the object itself.
Anim8or	"Anim8or" is a 3D program which allows importing of 3ds models from 3ds max and has an internal converter to C which extracts all the vertices, normals, information within the 3ds file. This allows us to load the static model for the game. However, similar to SpaceSimulator, Anim8or's exported C format did not contain information on the animation keyframes, lighting and material inside the 3ds file. We did, however, located, an anim8or viewer which provided decent lighting and visibility for our model. If we are not able to find a loader which includes the animation and lighting details within the 3ds file, we will be using a static model exported from Anim8or in our game. Anim8or is by far the best static model loader that we found on the web.
MD2	MD2 is one of the 3d modelling formats that support animation keyframes, lighting and materials. We will be using MD2 for our animated model. Please refer to section 3.5 Animation for more information.

3.4.2 Terrain Generation

We wanted to give a more realistic feel to our game so we created a terrain for our models to walk on. One of the terrain model creation methods that we looked at was using 3DSmax to create a 3DS model and load the terrain into our game as a static model using “Anim8or” or other 3DS model loaders. However, this method poses a problem as the height information which is needed to define the player boundary is not shown in the 3ds format. Thus, we cannot use this method to define our terrain. Figure 11 shows a snowfield terrain that we made in 3DS Max studio.

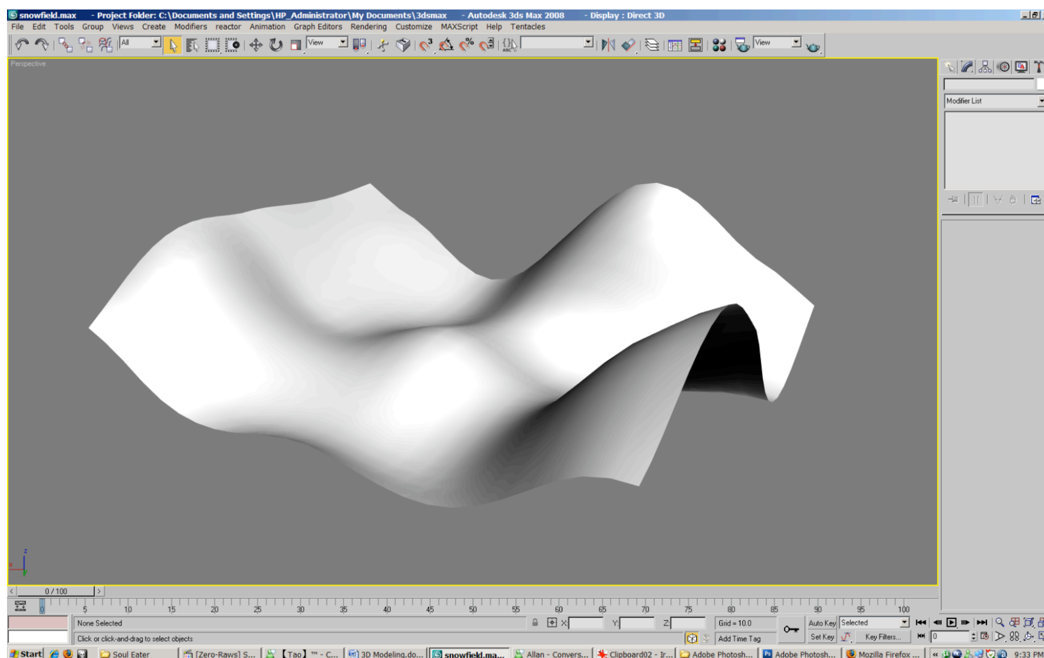


Figure 8 A Snowfield Terrain Using 3ds Max Studio

In the end, we decided to extract the height information from a grayscale bitmap that we created. The bitmap is displayed in a 24 bit 128x128 grayscale format. The terrain loader is written in a way that the program extracts height information from the bitmap based on the color intensity of the bitmap. White represents the maximum height and black represents the minimum height. If

our bitmap consists of regions with colors that are gray, then terrains with heights that are between the maximum and minimum heights will be shown.

The two figures below show two example terrains generated by two different bitmaps.

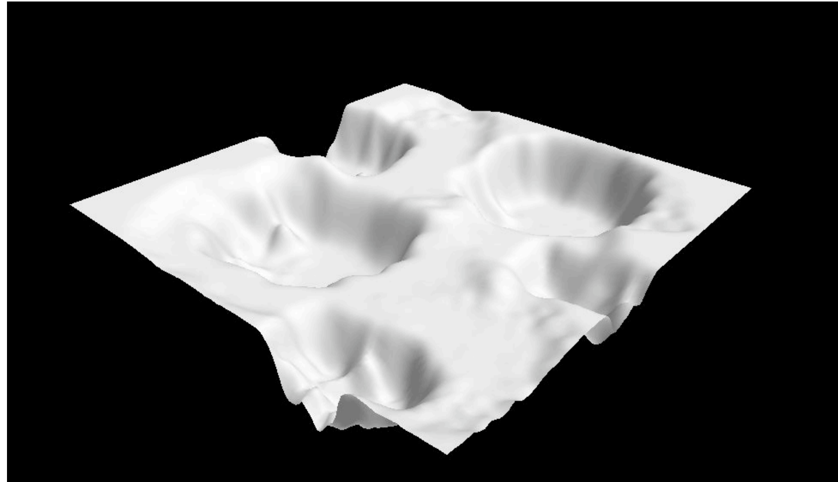
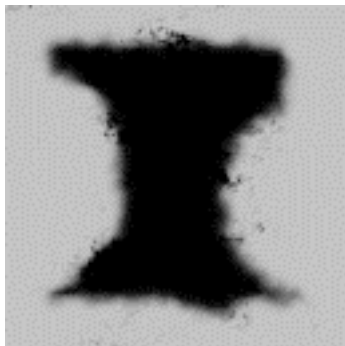


Figure 9 Terrain with Bitmap 1



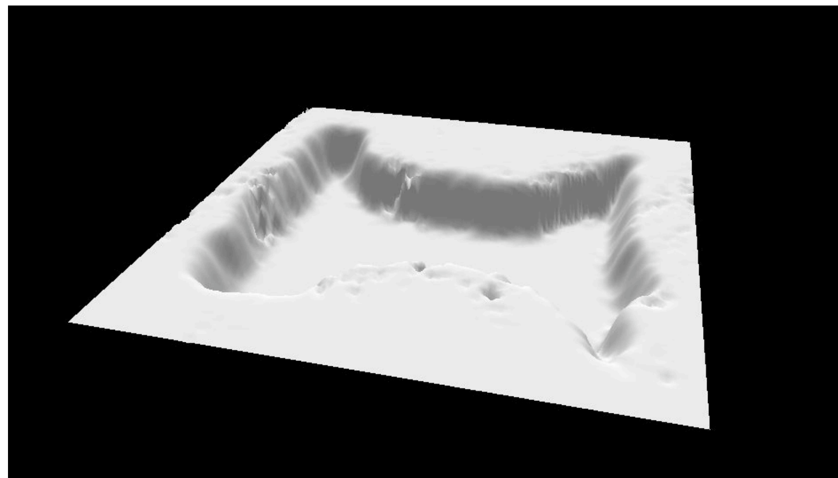


Figure 10 Terrain with Bitmap 2

3.4.3 Frustum Culling

The terrain can easily be the largest model in the game and rendering it without optimization will drastically slow down the overall loading time of the game. Thus, a technique known as "frustum culling" is used to hide certain parts of the terrain that is not visible by the viewer. This technique enables our games to have faster loading times.

Our frustum culling is still in its early development and we hope to have it made ready by the time of our demo.   The figure below shows an explanation of the fundamentals of frustum culling.

The following is an explanation of the fundamentals of frustum culling. Total number of faces that need to be rendered without frustum culling is 1458 (Figure 11).

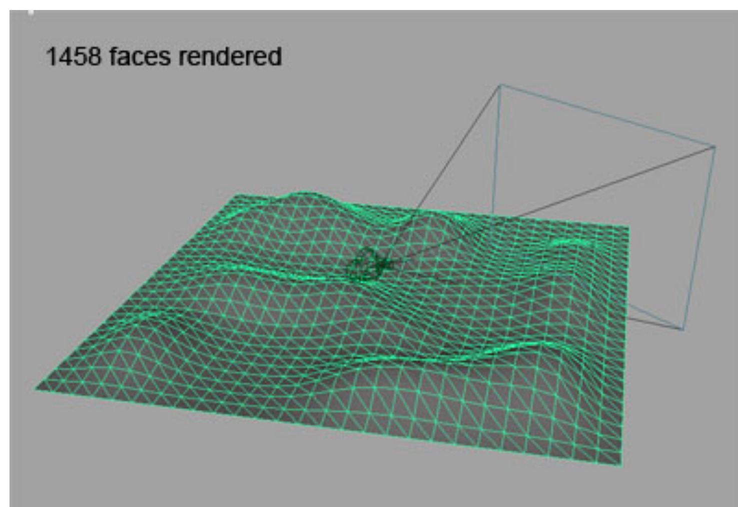


Figure 11 Frustum Culling – Example Terrain

Source: http://accad.osu.edu/~aprice/courses/BVE/frustum/frustum.html

When modeling the terrain, we can cut the model down to multiple parts that match together. The above image shows the terrain cut into nine sections (Figure 12).

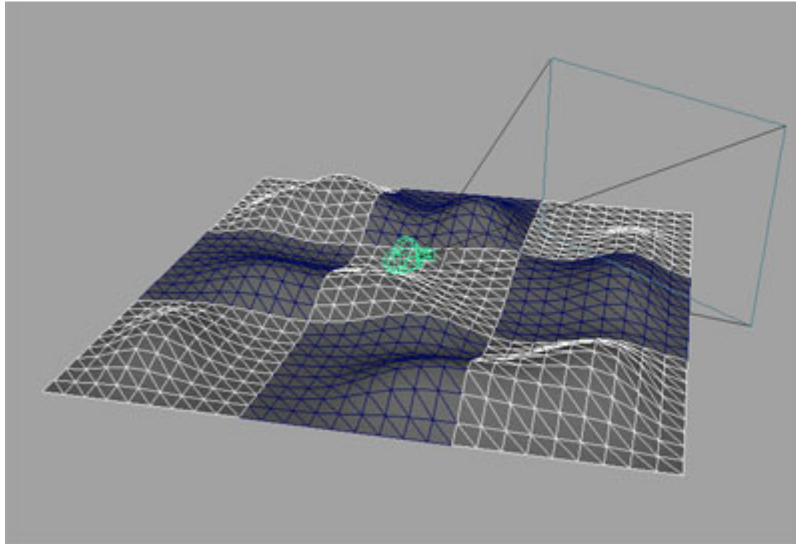


Figure 12 Before Frustum Culling

In Figure 13, the highlighted portion of the terrain model is the only part that's detected by the terrain engine as it is the only area that is inside the view frustum. Thus, the highlighted portion is the only area that is rendered.

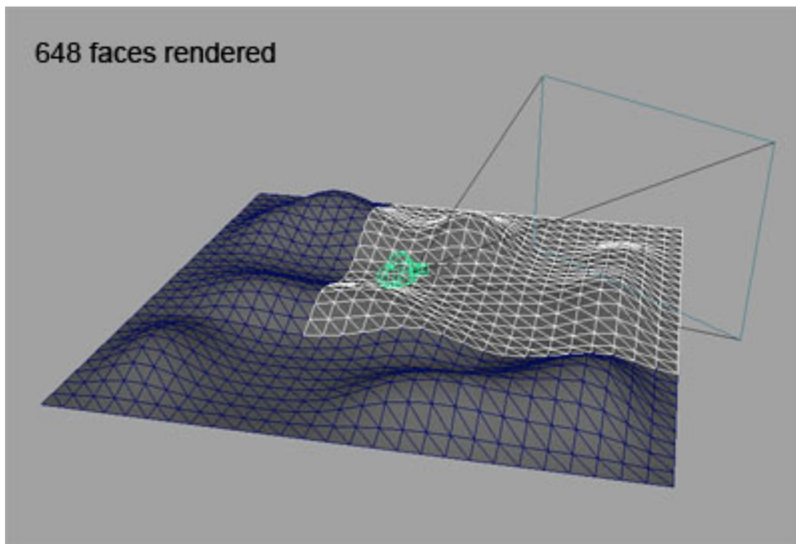


Figure 13 Area Inside the View Frustum

This optimization technique allows the terrain engine to render 648 faces as opposed to 1458 faces on the whole terrain.

3.5 Animation

Originally we decided to work with 3ds (3ds Max) models for animation. Though there are many online guides on the chunk structure of the 3ds model, most of the loaders available on the internet do not support key frame animation. Therefore, we can only easily use them to load static 3ds models into our game. Although it is possible to load a series of static models and animate them inside our program, this approach is quite inefficient and requires a lot of bookkeeping by our game engine. After some research and experimenting with different model files, we have decided to use the md2 (Quake 2) model files for character animation because it supports the necessary features we need. There are also many available resources online about the model structure and loaders that can be integrated with our code. In addition, the game Quake 2 is open-source, so we can study how the original game utilizes the md2 models.

We primarily used the tutorial available on the website "The Quake II's MD2 file format" (http://tfc.duke.free.fr/old/models/md2.htm) by David Henry. The loader available on the same website is also incorporated into our code. Most of the following diagrams and codes depicting the data structure and the algorithms are also from the website.

The key frame animation structure is embedded in the MD2 model, as shown in figure 17 below:

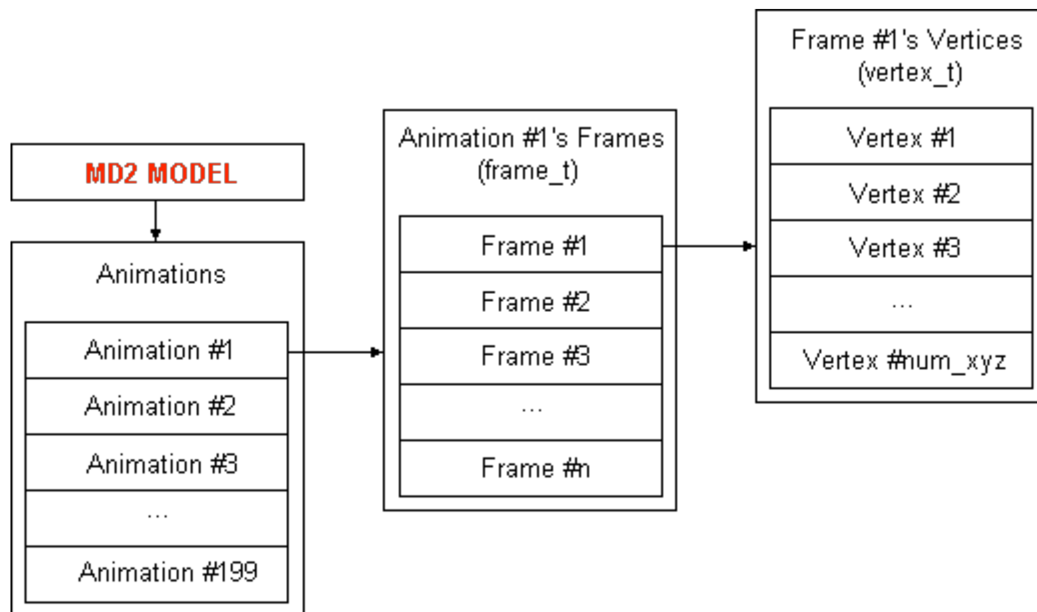


Figure 17 Animation Data Structure in the MD2 model (from "The Quake II's MD2 file format")

As shown in the figure, each animation sequence contains a number of frames, and each frame contains the vertices required to display. We can identify these animation sequences by their starting and ending frames. The following code segment demonstrates how we identify the different sequences common in most MD2 models:

```
anim_t CMD2Model::animlist[ 21 ] =
{
    // first, last, fps

    {   0,  39,  9 },   // STAND
    {  40,  45, 10 },   // RUN
    {  46,  53, 10 },   // ATTACK
    {  54,  57,  7 },   // PAIN_A
    {  58,  61,  7 },   // PAIN_B
    {  62,  65,  7 },   // PAIN_C
    {  66,  71,  7 },   // JUMP
    {  72,  83,  7 },   // FLIP
    {  84,  94,  7 },   // SALUTE
    {  95, 111, 10 },   // FALLBACK
    { 112, 122,  7 },   // WAVE
    { 123, 134,  6 },   // POINT
    { 135, 153, 10 },   // CROUCH_STAND
    { 154, 159,  7 },   // CROUCH_WALK
    { 160, 168, 10 },   // CROUCH_ATTACK
    { 196, 172,  7 },   // CROUCH_PAIN
    { 173, 177,  5 },   // CROUCH_DEATH
    { 178, 183,  7 },   // DEATH_FALLBACK
    { 184, 189,  7 },   // DEATH_FALLFORWARD
    { 190, 197,  7 },   // DEATH_FALLBACKSLOW
    { 198, 198,  5 },   // BOOM
};
```

To make these sequences easier to call, enumerated types are introduced so the main program can easily access them, as shown in the following code segment:

```
typedef enum {
    STAND,
    RUN,
    ATTACK,
    PAIN_A,
    PAIN_B,
    PAIN_C,
    JUMP,
    FLIP,
    SALUTE,
    FALLBACK,
    WAVE,
    POINT,
```

```
    CROUCH_STAND,
    CROUCH_WALK,
    CROUCH_ATTACK,
    CROUCH_PAIN,
    CROUCH_DEATH,
    DEATH_FALLBACK,
    DEATH_FALLFORWARD,
    DEATH_FALLBACKSLOW,
    BOOM,

    MAX_ANIMATIONS

} animType_t;
```

In addition to draw the frames stored in each sequence, we need to also interpolate between frames to make the animation smooth. The linear interpolation model in the MD2 loader is described in a simple formula:

$$X_{interpolated} = X_{inital} + InterpolationPercent * (X_{final} - X_{inital})$$

We will be able to create any arbitrary number of intermediate frames necessary for our program to smoothly render the animation.

The following figure shows the MD2 model loader rendering a Gunner model in Quake 2:

Figure 18 The Gunner model in Quake 2

The following 2 figures show 2 frames in the “Attack” animation sequence:

Figure 19 Attack animation: Loading   Figure 20 Attack animation: Ready to Fire

4 Possible Improvements

There are many improvements that can make our game more entertaining. The following list includes some major ones:

(a) Network play: This allows the two human players to play remotely against each other rather than having to be both physically present at one place to play the game.

(b) AI: This allows the human player to fight against a computer character, or even allow computer vs. computer gameplay for viewing. This is also potentially useful in team matches.

(c) Team Play: This allows more than 2 human players to join the game and form 2 teams to compete with each other.

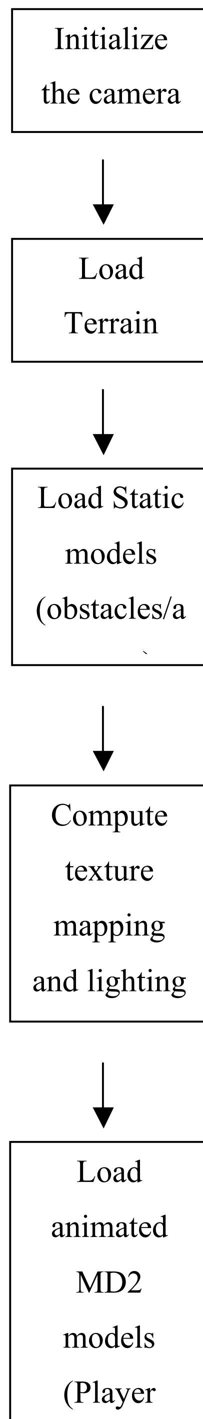
(d) Shadow: The shadow generation of the models will also give the game a more realistic feel of the characters.

(e) Skybox: This simulates a more realistic environment for the game field.

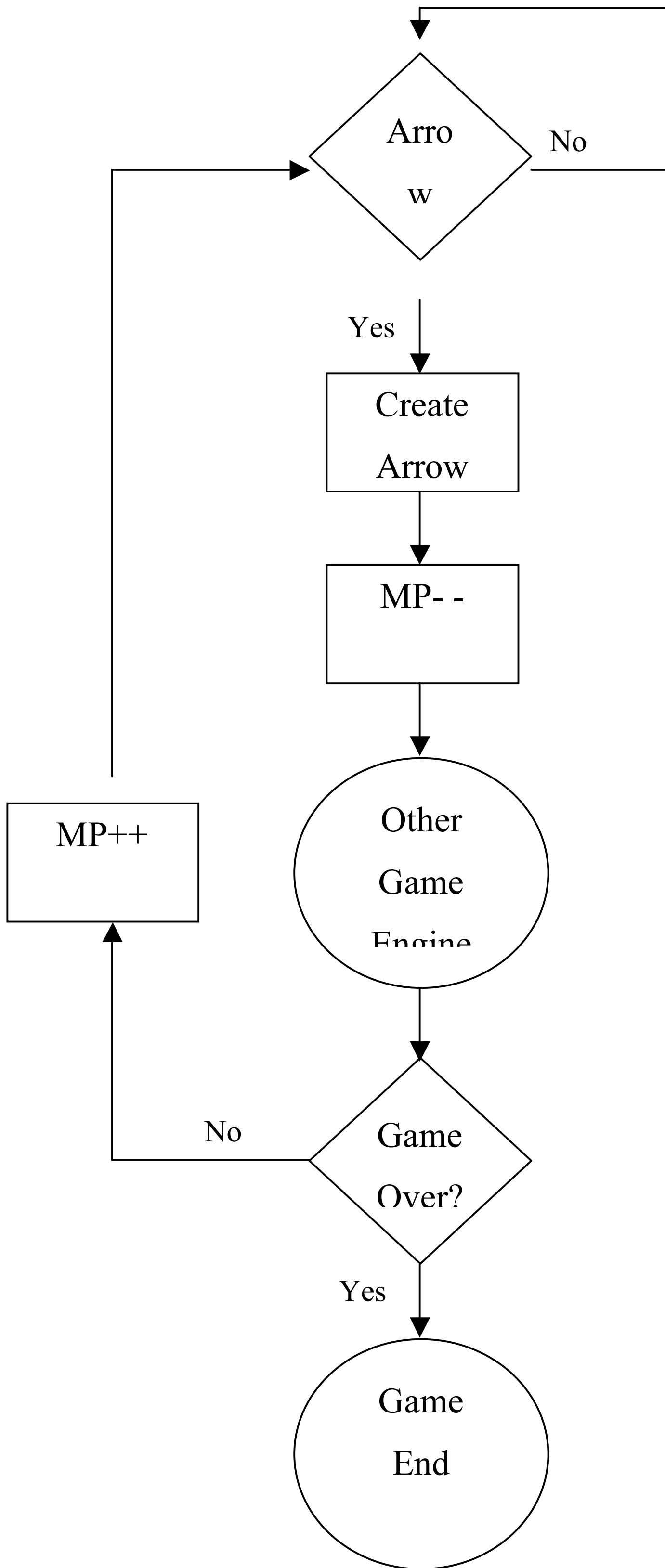
5 Flow Chart

The following figures are the flow charts of the various components of our game engine:

Game Graphic Initialization



MP Bar



HP Bar