EECE 478

Tower Defense

Jennifer Chui
40187023

Billy Lam
40308025

Clement Leung
17860032

Neil Pahl
88483045

Table of Content

Table of Figures

1. Introduction

This purpose of this report is to provide an in-depth description of the 3D Tower Defense game that was designed and developed from scratch as part of the project requirement in the EECE 478 Computer Graphics course.

The game mimicks the various 2D tower defense flash games that are on the Internet in which swarms of enemies come at specific time intervals, and as the player, one has to build towers and upgrades to destroy these enemies in order to prevent them from crossing a certain boundary. The 3D Tower Dense game transforms that idea by having swarms of enemies spawning to attack a tower structure, and as a player, one has to stay within the tower boundary to kill as much enemies as possible, gaining upgrades and recoveries along the process. The game features the use of OpenGL to render every component in a 3D environment, incorporating techniques and tricks learned throughout the computer graphics course. These include the basics of drawing simple polygon models using OpenGL commands, utilizing camera angles, texture mapping, setting lights and materials on various objects.

Furthermore, the game contains various advanced features which added a lot more dynamics into the game play itself. These features include the implementations of collision detection using the quad tree technique, physical simulation, sound effects, animations, alpha blending, artificial intelligence, and view frustum culling. The report also contains an appendix which is the development log contributed by the entire project team throughout the entire game development process. It outlines the initial game idea, how it has eveolved and also the roles and responsibilities of each team member at various stages.

2. Game Description

The following section will discuss about game operation and background. The first couple of sections will describe the storyline behind the Tower Defense and how to operate the game. After, the implementation behind game interface, player, and camera angle are explained. At the end, character implementation and special features in the gameplay are discussed.

2.1. Game Storyline

As part of a special Angellonian task force, your team was sent to a far away planet to explore the possibility of expanding your race's presence in the universe. Your team's mission was to plant a transportation crystal that will import fellow Angellonians to this region. However, due to agricultural advances, the need to expand the races territory was no longer existent. From that point on, your task force was left to die…

Many years later, the Angellonian race is looking face-to-face with extinction as a giant meteor was found to be headed straight for their planet. Their only hope of survival is to immigrate to your stationed planet via the transportation crystal.

The evil Devillonians, while inferior in military power, are always attempting to eliminate the Angellonian race. Therefore, upon hearing news of the meteor, the Devillonians realized an opportunity to wipe out the Angellonian race once and for all… by destroying the transportation crystal.

Now, after being abandoned on this planet for years, your team has slowly died off and you are left alone as the sole survivor. You are given a mission of the highest priority. Your mission is to protect the tower at all costs from the Devillonians, until your army can safely transport…

The fate of the Angellonian race depends on you.

2.2. Gameplay and *Game Flow*

The flow of the game consists of four states:

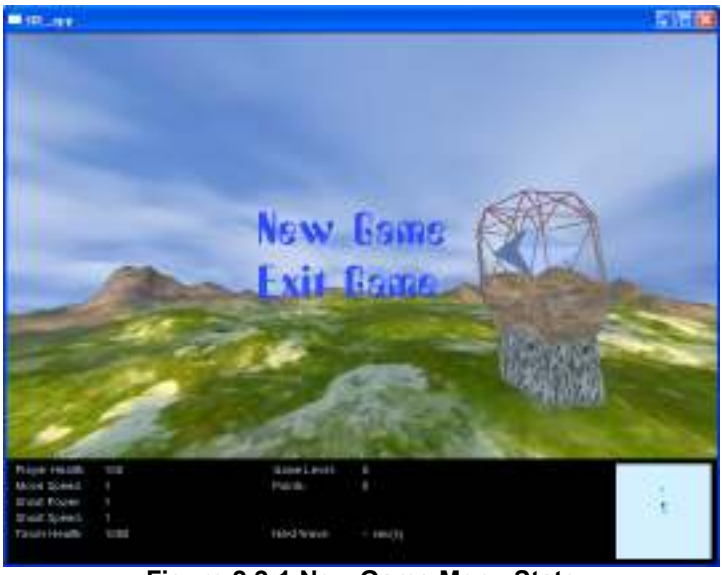


Figure 2.2-1 New Game Menu State

1) New Game Menu – The user is prompted to start a new game or exit the game. In this state the mouse cursor is visible and used to select the menu.

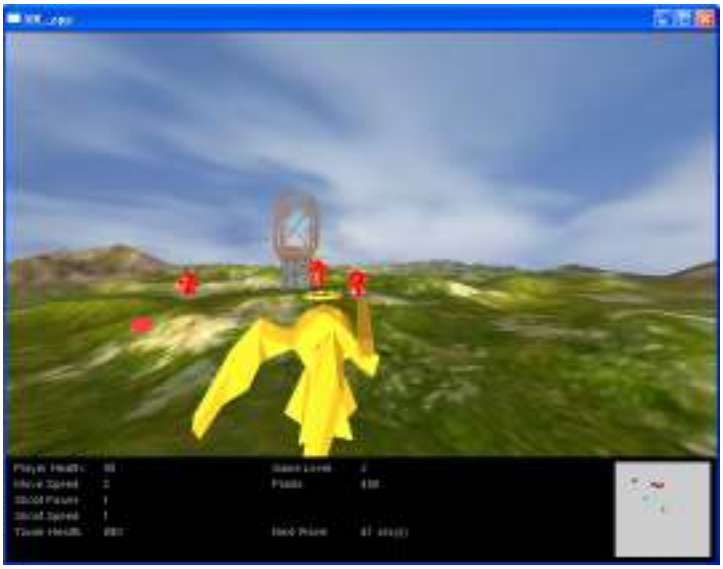


Figure 2.2-2 In Play State

2) In Play – The game character is now being controlled by the user via the mouse and keyboard to play the game.

Figure 2.2-3 Paused Game State

3) Paused Game – The game is paused by pressing the Esc key during ‘In Play’ state. The enemies and objects of the game enter a paused state. The game character is no longer visible, the mouse cursor is visible, and the game menu is available. The game menu now contains a ‘Resume Game’ option.

Figure 2.2-4 Game Over State

4) Game Over – When the player or tower’s health depletes to zero, the game stops and the “Game Over” message appears. Upon clicking the screen, the game will enter the ‘New Game Menu’ state.

2.3. Game Interface

To control the game, the following controls scheme is used:

Figure 2.3-1 User Control Scheme

2.4. Controlling the Player Viewing Direction

The Player Viewing Direction (or Player Aiming Direction) is the direction in which the player is facing (Aiming). This direction is controlled by the user via motion of the mouse. When the mouse is moved, the relative motion of the mouse is interpreted via the SDL Event Handler. Horizontal mouse movements will adjust angle a0, and vertical mouse movements will adjust angle a1. Every render, the player's direction is updated to incorporate the total mouse displacement over that rendering interval.

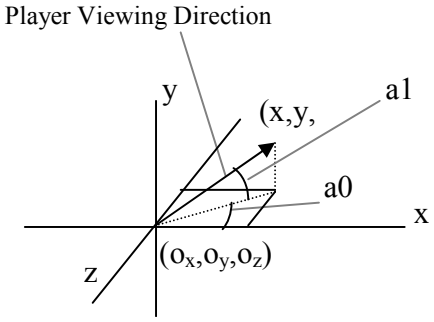


Figure 2.4-1 Player Viewing Direction

Where (o_x,o_y,o_z) is the player position, the players' direction is kept as a unit vector with components <x,y,z> . It is calculated from angles a0 and a1 as follows:

$$x = \cos(a0)$$
$$y = \sin(a1)$$
$$z = \sin(a0)$$

2.5. Player Movement

The game character moves relative to the direction in which the character is facing. The ‘W’ and ‘S’ keys move the player forwards and backwards while ‘A’ and ‘D’ moves the player left and right laterally.

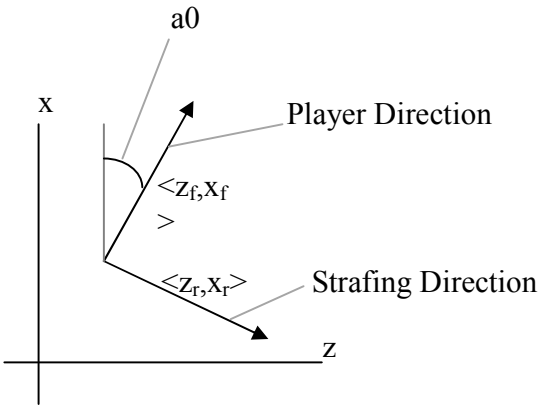


Figure 2.4 Player Viewing Direction

From the 3 dimensional Player Viewing Direction, the forward direction in the z-x plane can be found. To find the lateral motion direction, the following relations are used:

$$z_r = x_f$$
$$x_r = -z_f$$

Every render, the players’ position is updated based on whichever of the following player movement keys are being pressed:

[W]	New Position = Old Position + (Player Direction Vector) x (Player Speed)
[S]	New Position = Old Position - (Player Direction Vector) x (Player Speed)
[A]	New Position = Old Position - (Strafing Direction Vector) x (Player Speed)
[D]	New Position = Old Position + (Strafing Direction Vector) x (Player Speed)

2.6. Camera Angles

Player Cam View

This default view will be created by a camera that is following the player. As described in the following diagram, the camera is located at the affine point,

$$(\text{Player Position}) - (\text{Player Viewing Direction Vector}) \times (\text{Camera Distance}) + j(\text{Camera Height})$$

And aims at the point,

$$(\text{Player Position}) + j(\text{Camera Height})$$

Where the Camera Height and Camera Distance are constants and the Player Position and Player Viewing Direction are updated every render cycle, according to the user input.

Figure 2.6-1 Player Cam View

Tower Cam View

The Tower Cam View is simply a camera located at the tower, aiming at the player.

Figure 2.6-2 - Tower Cam View

Birds Eye View

The birds eye is a camera located high above, looking directly downwards on the playing area. This view is shown as follows

Figure 2.6-3 - Birds Eye View

2.7. Enemy Spawning and Power-up Items

At specific intervals during the game, enemies are spawned randomly from different directions. Each enemy wave contains 30% of ranged enemies that fire spears and the rest are enemies with close-ranged combat skills.

As the game level increases, the enemies generated become stronger, and their number also increases. They process higher health points, faster speeds, and deal more damage in their attacks.

2.7.1. Power-up Items

Power-up items are available to the player at a probability of 10% upon an enemy killed. These power-up items are drawn and loaded from blender, and they are made up of the following types, each of which has a different effect on the player's attributes:

HEART boosts the player's health by 10 points
BOMB increases the player's shooting power
ARROW increases the player's shooting frequency
BOOT increases the player's movement speed

The power-up items are subclasses of the object model class. Collision detection technique same as that used for the player and enemies are done on the items to determine whether they have been absorbed, which will be discussed in greater details later in this report.

3. Environment

The environment of this game can be divided into several parts: Terrain, Sky Box, and Tower. Material and Lighting are added to these parts to make the environment appear more realistic. Also, materials were added to objects to make them include realistic interaction from light sources.

3.1. Terrain

The terrain area of the game is generated using a set of height map data stored in a text file. The data file defines the height levels of the ground at specific vertexes in order to create mountains, valleys and a feel of bumpiness throughout the terrain.

At game start, the height map data file is dynamically loaded into memory that can be used to generate the terrain at each render loop. The following sub-sections breaks down and describes how the terrain is actually rendered and the effects that are associated with it

3.1.1. Using the Height Map data

The terrain itself is a set of quad strips drawn in parallel. The function call to render the terrain iterates through the vertexes that need to be drawn and references to the height map data to incorporate the height level at each point. The figure below shows how the quad strips with height values look like in GL_LINE mode.

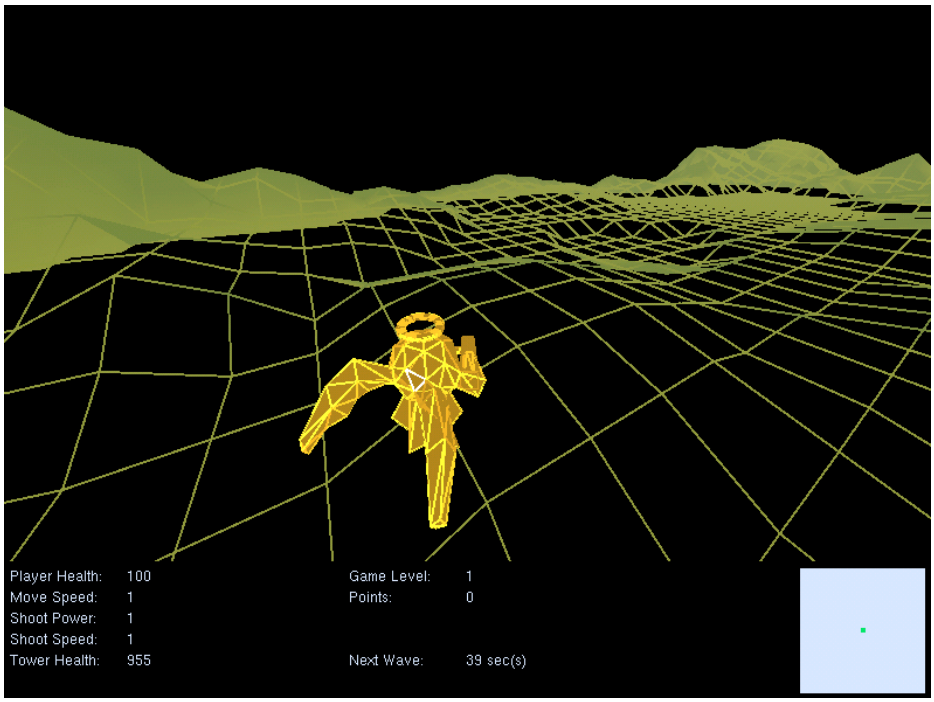


Figure 2.7-2

3.1.2. Mapping the Ground Texture

In order to make the ground look realistic, the entire terrain is texture mapped with a bmp image that incorporates different terrain properties including mountains, grass and rocks. The texture is mapped and scaled according to the position of each vertexes in the quad strips. The screen-shot below shows the same quad strips, only with texture mapped onto them.

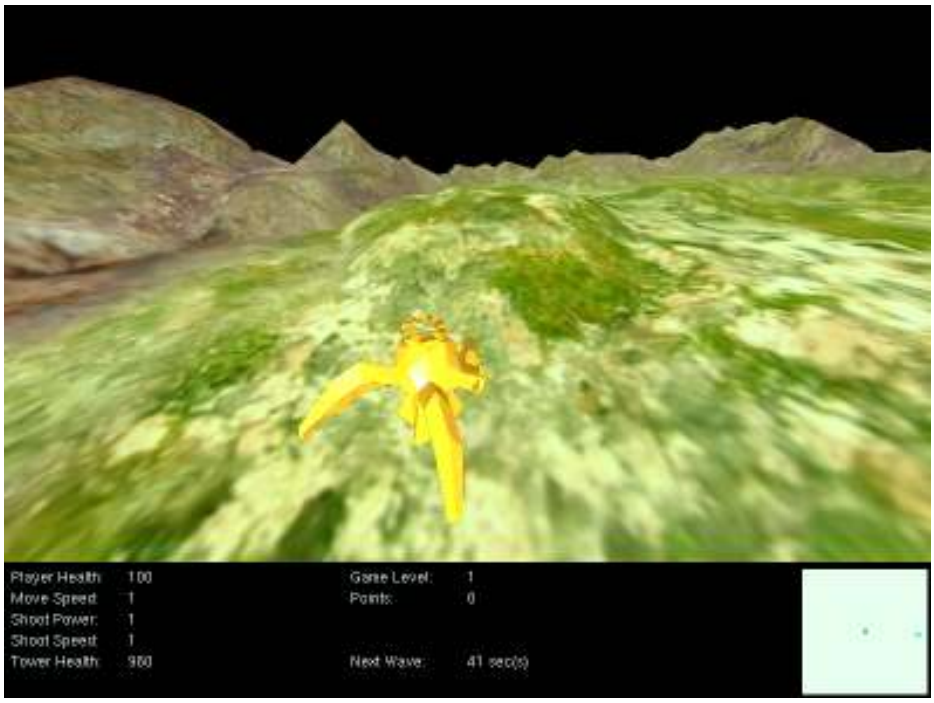


Figure 2.7-3 Ground plane with texturing

3.1.3. Defining Normals

The terrain shown in the previous figure did not have their normal vectors correctly calculated, as they are all set to point upward initially. Therefore, the reflection of light is the same across the entire ground plane. The following screen-shot shows the complete version of the terrain implemented, with proper normals defined for each vertex in the quad strips.

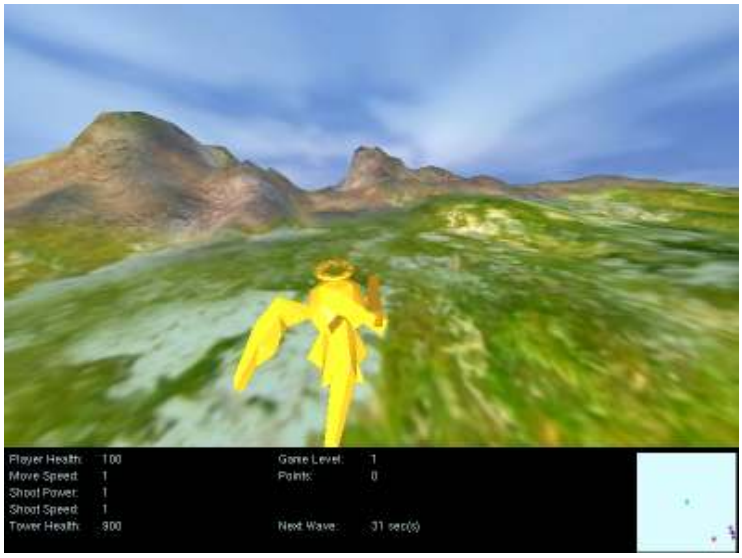


Figure 2.7-4 Ground plane with Defined Normal

Similar to how the height map data are stored, the normal vectors of the terrain are calculated and stored at initialization of the game to save computations during render time. They are then looked up accordingly as each ground plane vertex is drawn along with the texture corrdinates.

Shades and variation of light reflections are clearly visible over the mountains and the bumpy areas, making the scene even more realistic than the previous version displayed. To calculate the normal vector of a particular vertex, all four polygons surrounding the vertex have to be taken into account. This is done by taking two vectors from each polygon, finding its own normal vector, and averaging and normalizing the resulting normals calculated from the four polygons.

Similar to how the height map data are stored, the normal vectors of the terrain are calculated and stored at initialization of the game to say computations during render time. They are then looked up and specified accordingly as each ground plane vertex is drawn along with the texture corrdinates.

3.1.4. Height Interpolation

During the game play, the player, the enemies and the bullets all react to the height level of the terrain. Both the player and the enemies go over bumps and hills while the bullets disappear when they collide with hills. These details are all done by taking the objects' coordinates on the ground plane and finding their corresponding height values inside the height map data.

Since the height map data only defines height levels for specific vertexes, the values have to be interpolated when the objects' positions do not lie above the defined vertexes, which is usually the case.

The calculation is done by a simple linear interpolation that takes into account the four closest height levels and the object's position relative to those four vertexes. The detailed logic of the height interpolation function is illustrated in the figure and steps below:

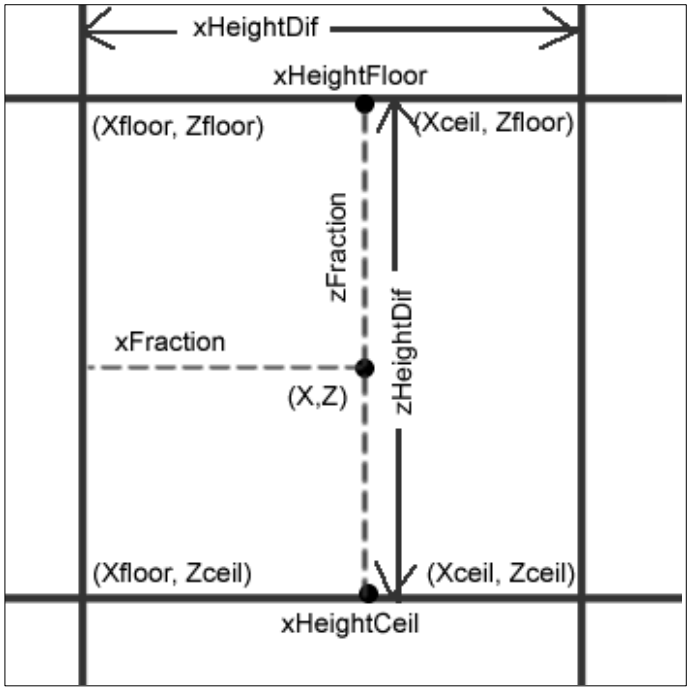


Figure 2.7-5 Height Interpolation Calculation

1. Suppose an object is located at coordinate (x,z) in between four quad strip vertexes on the ground plane, the function first retrieve the height levels from the points (Xfloor, Zfloor), (Xceil, Zfloor), (Xfloor,Zceil) and (Xceil, Zceil) from the height map data.

2. The values xHeightFloor and xHeightCeil in the figure are calculated based on xFraction and the height level difference between the vertex pairs {(Xfloor,Zfloor),(Xceil,Zfloor)} and {(Xfloor,Zceil),(Xceil,Zceil)}, respectively.
3. The value of zHeightDif can then be determined using xHeightFloor and xHeightCeil, and at last the true height value at point (x,z) can be computed by taking into account zFraction.

3.2. Sky Box

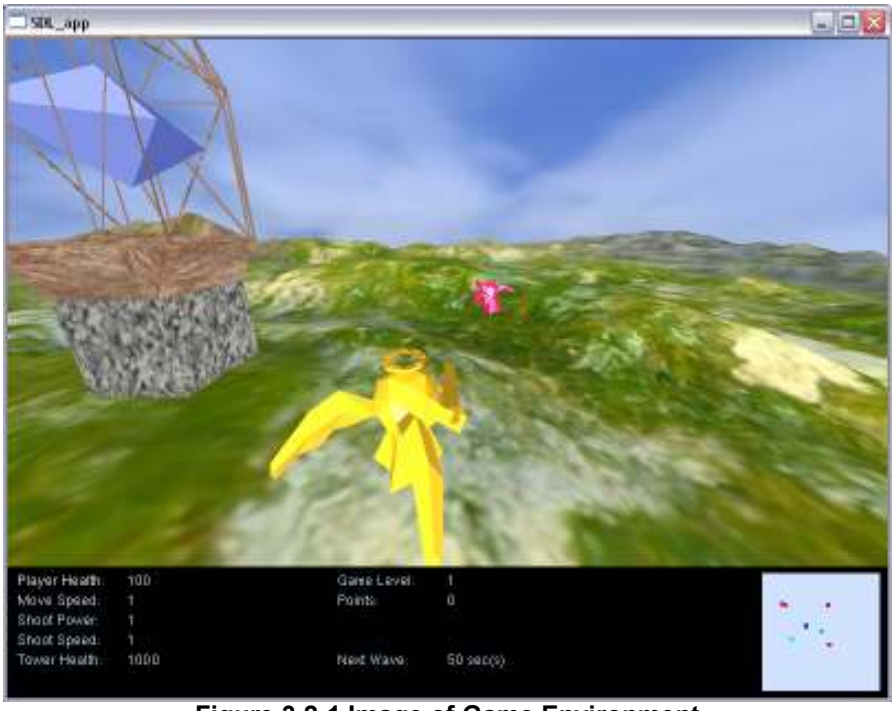


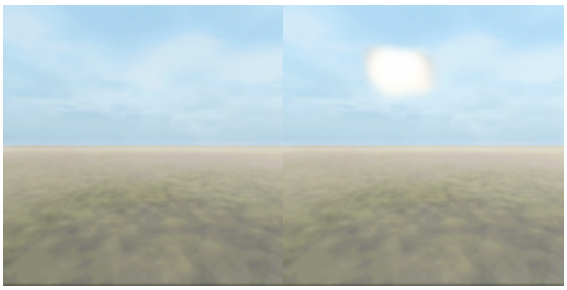
Figure 3.2-1 Image of Game Environment

A sky box is used to help the game appear more realistic by creating the illusion that our game environment is endless. The sky box is created by using two objects, a cube and a sphere, that surrounds the game boundaries. To make the sky box appear even more realistic, a moving sky is also added.

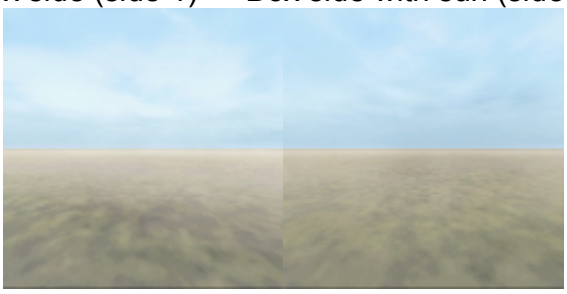
3.2.1. Cube and Game Boundaries

Our game boundaries are place within a cube that provides the backdrop. The inside of the cube is textured with the sky and sun images 1 side and just sky on the other.

Box side up



Box side (side 1) Box side with sun (side 2)



Box side (side 4) Box side (side 5)

Figure 3.2-2 The 5 images that are mapped on to the sky box cube.

3.2.2. Moving Clouds

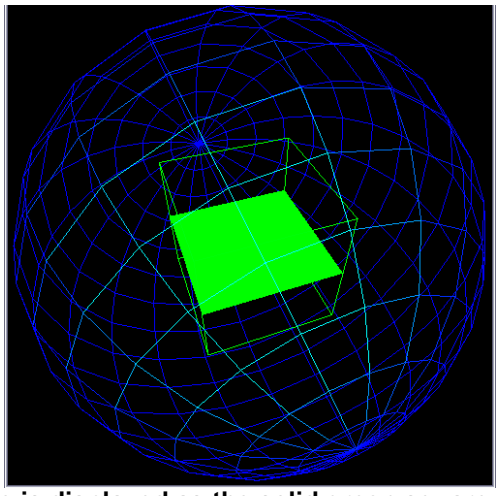


Figure 3.2-3 Ground plane is displayed as the solid green square while the wire cube is the skybox cube and sphere is the cloud sphere.

Our sky box gives off the illusion that the clouds are moving by incorporating blending. A much larger sphere with cloud textured on to it surrounds the cube. The picture above shows how the sphere surrounds the cube (the green cube) and the game plane (the solid green square). Having a much larger sphere helps to make the environment seem more endless. The sphere is alpha blended with the cube which makes the clouds translucent. To make the cloud appear as if it is moving, the sphere is slowly rotated around the center of the game. To make the cloud movement more realistic, the sphere is scaled down vertically so that the sky looks more infinite.

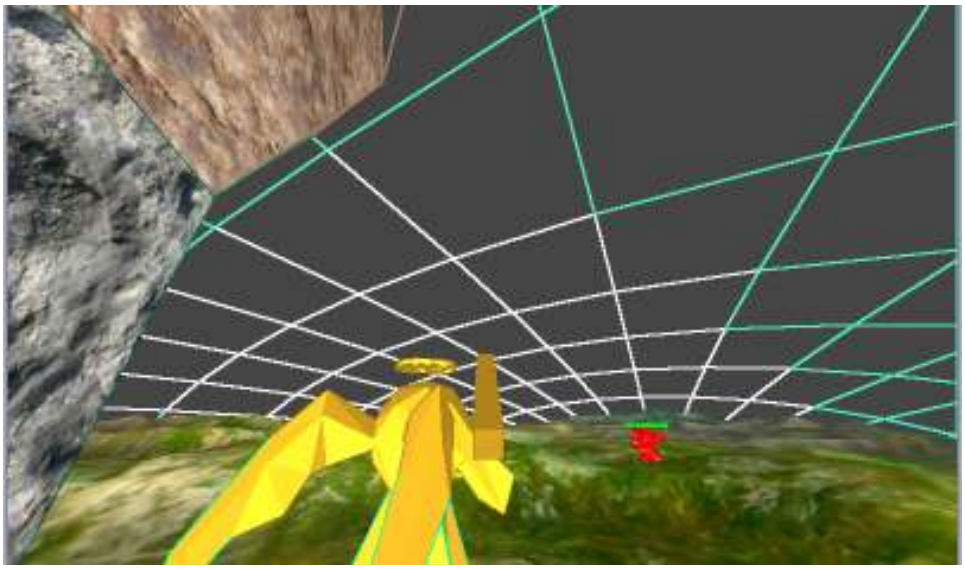


Figure 3.2-4 Flatten sphere makes the sky appear endless.

3.3. Tower

The tower is located at the center of the game. The enemy characters in the game will try to destroy the tower by shooting at the tower. If they succeed, the tower is destroyed. When the tower is destroyed, the tower object is no long rendered. The tower consists of two main parts, the lower base and the diamond on the top. The tower is one of the central objects in our game and needs to be seen from all corners. The glowing diamond on the top of the tower makes it stand out from the terrain.

Figure 3.3-1 Tower

3.3.1. Base

The base of the tower is modeled after the cement block below statues. It consists of rectangular prism at the bottom of the base and a platform on top. The rectangular prism is created by making a quad strip while the platform is made from 4 quads and 4 triangles. The base is texture mapped with marble and rock like material. The texturing mapping will be described in more detail in later sections.

Figure 3.3-2 Tower Base

3.3.2. Diamond

The diamond is located on top of the base and it rotates around the y axis. The diamond is created from a triangle strip with 8 triangles. Emission material property is added to the diamond to make it appear as if it is glowing. This will be discussed in further detail in later sections.

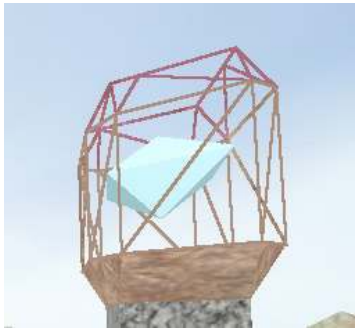


Figure 3.3-3 Diamond in Top part of Tower

3.4. Lighting and Materials

Light and materials are used to help create a more realistic look to our environment. Several lighting is used in our environment. There is a global light source and light 0 and light 1 are enable. Materials are added to both the tower and the character in the game.

3.4.1. Lighting

The global light source is an ambient light that light up the whole scene while light 0 is point source. The global light source is called by:

```
GLfloat global_ambient[] = { 0.2,0.2,0.2,1.0 };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient);
<just global light turned on>
```

It is relative low since we want the other lights to dominant.
Light 0 is located near the tower so that the diamond at the tower will glow. The ambient light is very low while the specular and diffuse is very high so that the diamond will appear shiny and would diffuse light better.

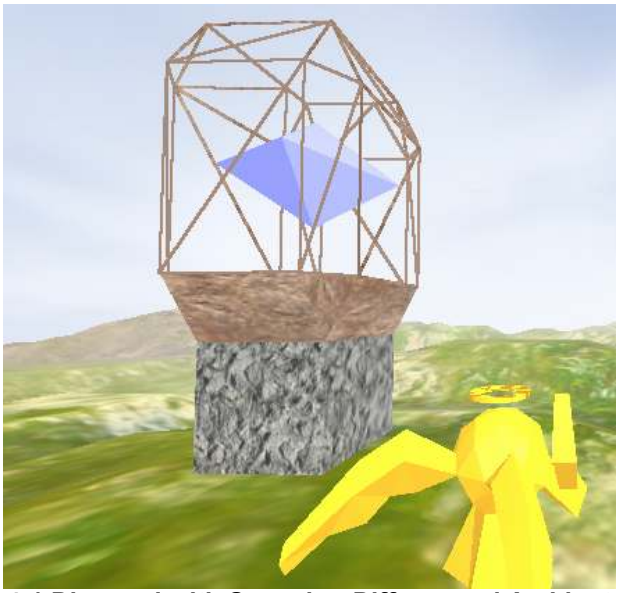


Figure 3.4-1 Diamond with Specular, Diffuse, and Ambient Lighting

3.4.2. Materials

Both the tower and the characters have materials applied to them. The diamond in the tower has both specular, diffuse, and emission material applied to it. The emission material in the diamond gives the diamond a glow. The emission values are changed slowly from blue to green everything the tower is rendered.

Figure 3.4-2 Blue Emission

Figure 3.4-3 Green Emission

Specular light is also added to the diamond. The specular light changed from blue to green also. It can be clearly seen in the screenshot below when emission is turned off.

Figure 3.4-4 Without emission, the blue specular light can be seen.

The characters also have material applied to it. For the angel character, yellow ambient and diffuse material is applied along with shininess. This gives the character a golden shiny look. The enemy has red diffuse and ambient material which gives it the red appearance.

Figure 3.4-5 Enemy Character with Red Ambient Light

Figure 3.4-6 Angel Character with Yellow Ambient and Diffuse Light

3.4.3. Fog

Fog is also added to our environment which provides a light as well. We use a relatively thin fog so that our moving clouds will be more visible.

With Fog

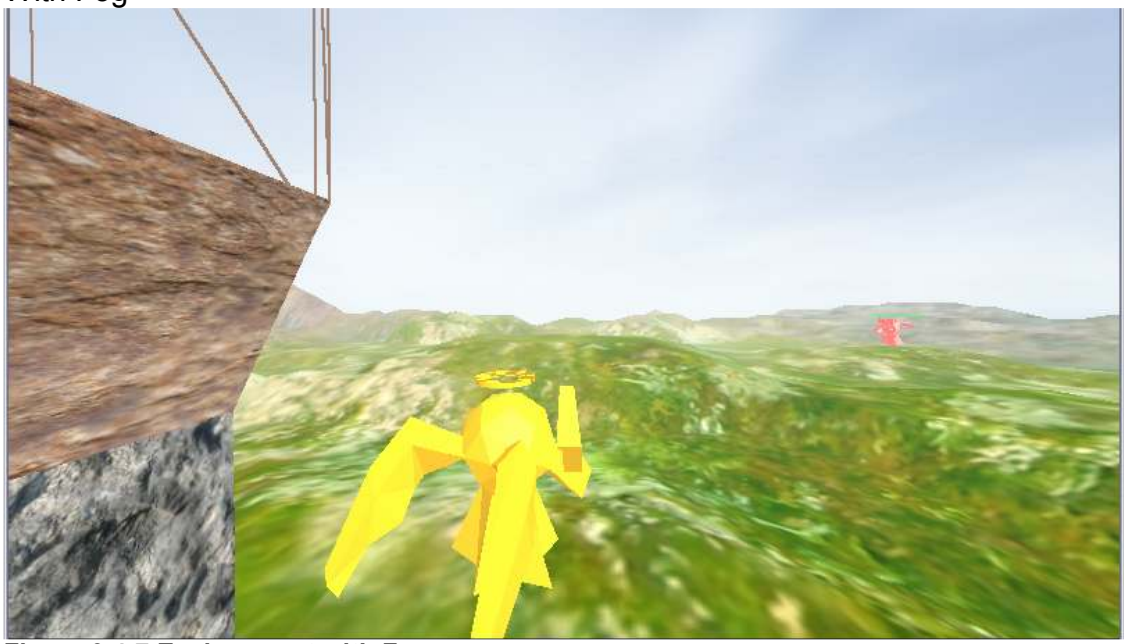


Figure 3.4-7 Environment with Fog

Without Fog

Figure 3.4-8 Environment without Fog

3.5. Texture

Texturing allows our environment to become more realistic and is done on the terrain, tower, and skybox. Some of the texture used the GL_REPLACE mode while others used GL_MODULAR.

3.5.1. Texturing on Skybox

The fours sides and box of the sky box were textured. Four images combined to make 360 degree picture which was used on the four sides. It gives the illusion that the environment is endless. The skybox texture used the GL_replace and CLAMP corners options so that the sky box seems more seamless.

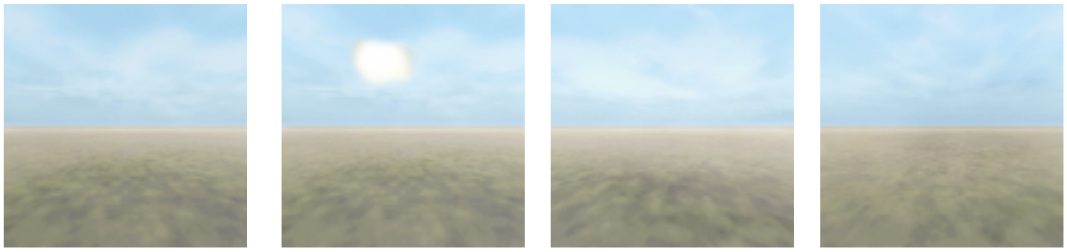


Figure 3.5-1 The 4 images that were textured on to the side of the skybox cube. The 4 images combined to make 1 long horizontal image

3.5.2. Texturing on Clouds Sphere

A large sphere that is outside of the skybox is used to produce a moving clouds effect. Images of the sky with clouds are textured on to the sphere. The sphere is created by making 16 quad strip. Each trip contains 36 quads and the last and first quad will connect to make a cylinder where the circumstance on opposite sides are different. Each strip will produce a horizontal strip of the sphere. To map the texture, the image is divided onto 10x16 grid and each square is mapped to a part on the sphere. The cloud texture used GL_MODULAR so that it can alpha blend with the skybox.

Figure 3.5-2 Texture Image for Cloud Sphere

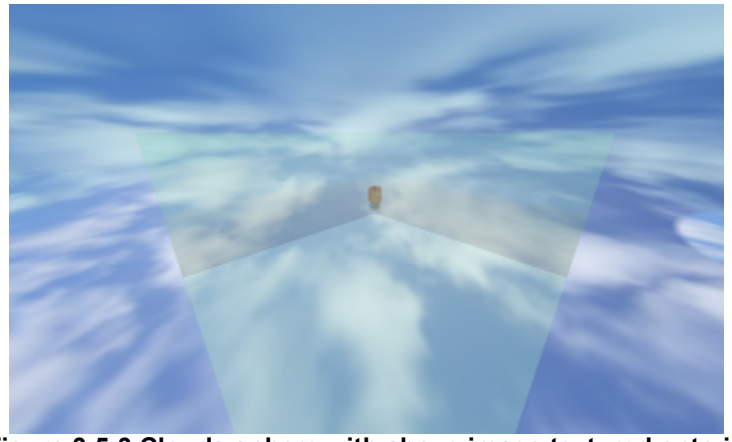


Figure 3.5-3 Clouds sphere with above image textured onto it.

3.5.3. Texturing on Tower

The Tower is texture with several different textures. The lower half of the tower uses two different textures. The bottom of the lower half of the tower is created using only 1 quad strip. The quad strip was textured by repeating only the horizontal part of the texture. The texture on the top of the lower half of the tower is mapped by using the s,t coordinate to map to vertices of the polygon.

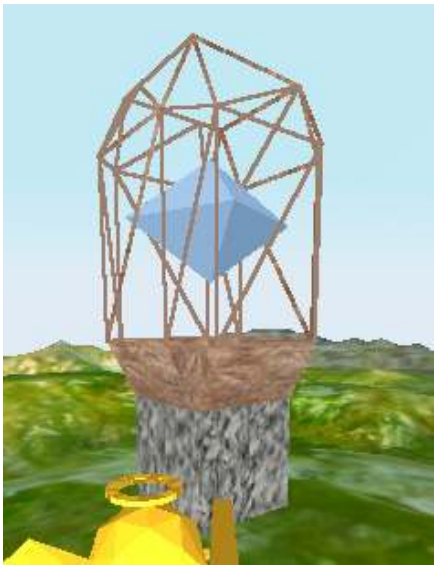


Figure 3.5-4 Tower with Texture

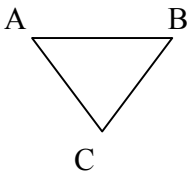


Figure 3.5-5 Triangle on the Side of the Upper Bottom Half of the Tower

The image above represents a triangle on one of the sides of the tower. To texture map the triangle, point A is mapped to (1,0) of the texture image while point B is mapped to (1,1). The third point C is mapped as 0, 0.5. This ensures that the texturing of the triangle does not look distorted. The texture for the tower uses GL_REPLACE and repeat so that the texture can be repeated on the tower.

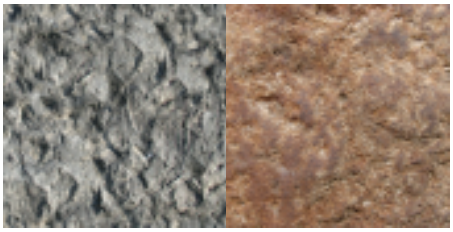


Figure 3.5-6 Tower Texture 1 **Figure 3.5-7 Tower Texture 2**

3.5.3.1. Texturing on terrain

The terrain image is also 1 large image as well. Terrain Texturing are described in further detail in the terrain section. The terrain is made of quad strips. The image is divided into squares where each square will map to a quad in one of the quad strips. The texture image for the terrain contains darker rock material on the border and lighter grass material inside. This improves the appearance of the terrain. The border of the terrain is a bit steeper and the rock texture will make it more realistic. The terrain texture used the GL_replace and CLAMP corners options so that it seems more seamless, and the colour of the polygon that is being mapped will not affect the appearance of the texture.

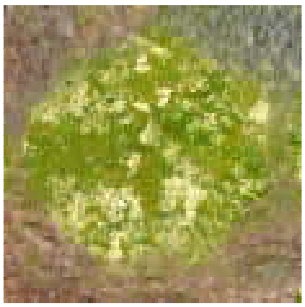


Figure 3.5-8 Texture of the Terrain

4. Advance Features

The following sections will describe the advance features that are added to the game. These features include the implementations of collision detection using the quad tree technique, physical simulation, sound effects, animations, alpha blending, artificial intelligence, and view frustum culling.

4.1. Collision Detection

The collision detection implementation of our game uses the strategy of creating bounding boxes around each object rendered in the game. We optimized the efficiency of our collision detection algorithm by implementing a quad tree class from scratch. Both techniques are explained further below.

4.1.1. Bounding Box Volume for Collision Detection

Each object introduced into the game has an appropriate bounding box associated to it. The bounding box is defined by three variables: the width, height, and depth of the object. A pictorial representation of this can be seen below. The below diagram shows two objects (object A and object B) encapsulated in their respective bounding boxes:

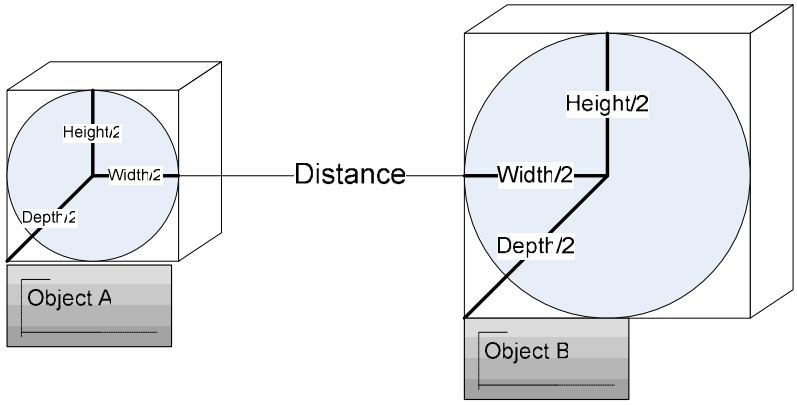


Figure 4.1-1 Two objects (A and B) shown with their bounding boxes used for collision detection.

The basic collision detection algorithm does the following series of steps for a comparison between two objects:

Calculates the absolute difference value between the two object centers in the x, y and z direction, i.e.

$$\Delta x = abs(objA_x - objB_x)$$
$$\Delta y = abs(objA_y - objB_y)$$
$$\Delta z = abs(objA_z - objB_z)$$

Calculate and store the distance value at which the bounding boxes of each object are touching (the maximum distance between the two objects in which collision still occurs):

$$width = \frac{objectA_{width}}{2} + \frac{objectB_{width}}{2}$$
$$height = \frac{objectA_{height}}{2} + \frac{objectB_{height}}{2}$$
$$depth = \frac{objectA_{depth}}{2} + \frac{objectB_{depth}}{2}$$

if $(\Delta x < width) \&\& (\Delta y < height) \&\& (\Delta z < depth)$
Collision has occurred between the two objects.

Only when step 3 evaluates to true do we have the two objects intersecting each other's bounding boxes. Thus, when step 3 evaluates to true we know a collision has occurred between the two objects. Upon detecting a collision, a function named "uponCollision" is used to execute an action for each object, depending on what type of collision has occurred (i.e. player to enemy collision, bullet to enemy collision, spear to player collision etc.). In our game there are many different types of collisions that can occur. The different types of actions that happen after a collision are handled by the uponCollision function. The following is a high level view of some of the major actions that happen after a collision between different objects. For a more in depth look into the different actions (events) please refer to the uponCollision method of each object class (Enemy, Player, Bullet etc.).

Collision Type	Action
Enemy to Player projectile collision	Enemy loses HP, projectile gets destroyed. Note: Enemy gets destroyed if Enemy HP equals zero.
Enemy projectile to player or tower collision	Player or Tower loses HP (game over if either HP becomes zero). Enemy projectile gets destroyed.
Enemy to Enemy collision	Enemies rotate to avoid interfering with each other's paths. Please see AI section for more detail.
Player to power up collision	Player powers up (HP increase, speed increase, weapon power increase, or shot speed increase depending on power up icon). Power up icon object gets destroyed.

4.1.2. Quad Tree

To minimize the number of comparisons needed between objects for collision detection during each render, we have implemented a Quad Tree class to group objects into their respective regions. A Quad Tree is a data structure that partitions a space recursively by subdividing the space into four quadrants or regions. By using a Quad Tree, we reduce the number of comparisons between objects during collision detection dramatically by only comparing objects with other objects that are within the same region. Objects that are situated on the boundaries of regions are compared within all regions that touch the aforementioned boundary. For example, object A in the below diagram lies on the boundary of two regions. Therefore object A will need to be compared with object B and C during collision detection. We find that a recursion depth of 2 would be appropriate for our game. Note however that our Quad Tree implementation allows the recursion depth to be readily increased or decreased. A Quad Tree with a depth of 2 gives us 16 regions. Below is a pictorial representation of a depth 2 Quad Tree:

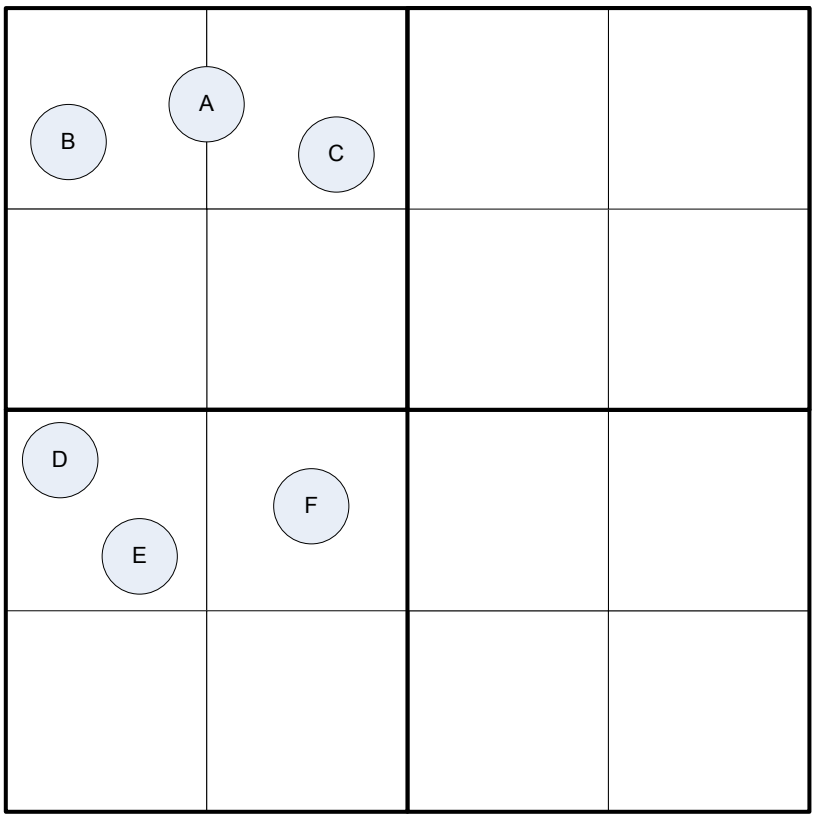


Figure 4.1-2 Depth 2 quad tree implementation. Objects are only concerned with other objects that are within the same region during collision checking. For example, objects D and E only have to check against each other for collision.

The following is a flowchart diagram that illustrates at a high level the full collision detection algorithm:

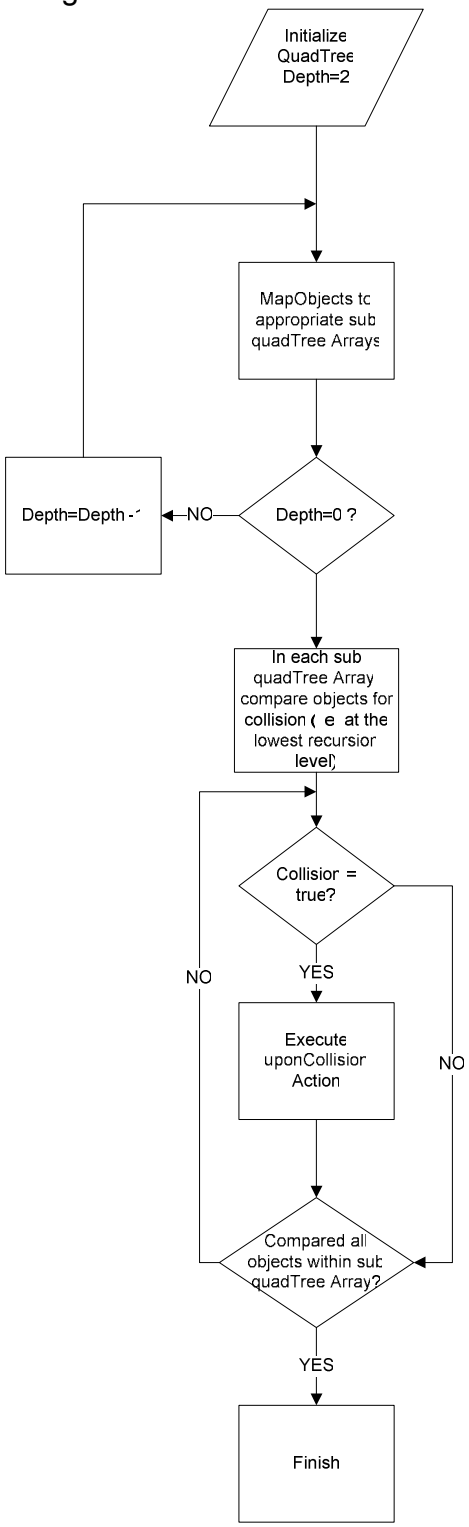


Figure 4.1-3 High level flow diagram of the overall collision detection implementation

4.2. Physical Simulation

We have implemented the physics of projectile motion in our game. Each projectile fired (bullet by the player, or spear by the enemy) is dictated by the laws of the game's gravity component.

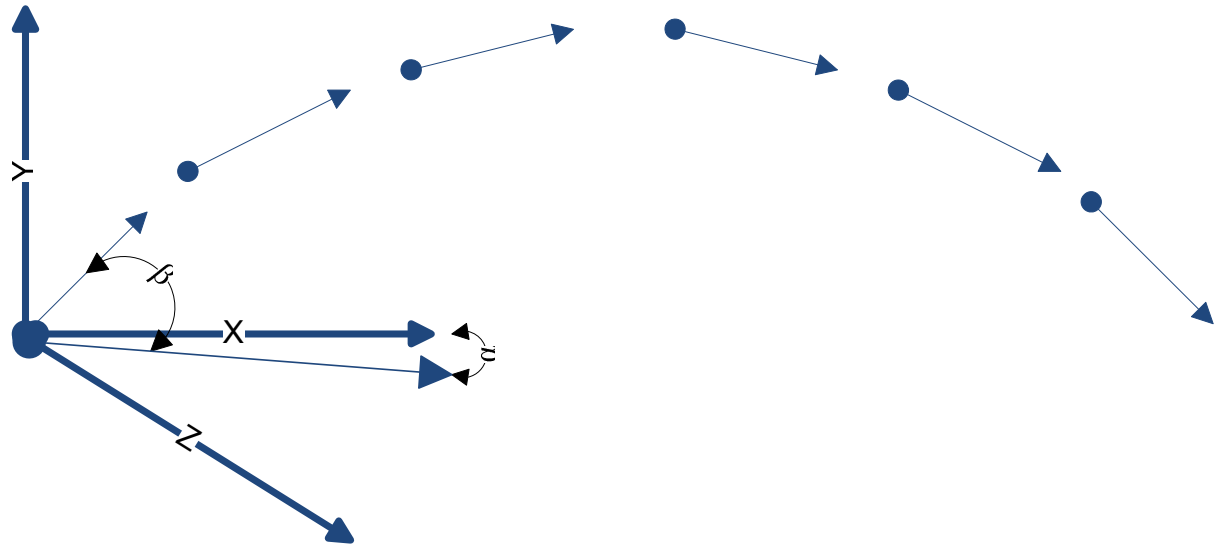


Figure 4.2-1 A Diagram that displays the physics of projectile motion.

There are two types of projectiles in our game: the spear, shot by the enemies, and the bullet, shot by the player character. The main difference between the two types of projectiles is that the direction and angle in which the spears are shot by the enemy are fixed, whereas the user has the ability to change the player's shooting direction and angle. Another difference is the speed of each type of projectile, explained in more detail below.

Upon creation of a projectile object (whether it be by pressing the shoot button, or automatically through enemy AI), the angle of the projectile is first calculated, namely alpha and beta (as seen in the above diagram) are calculated. For the player's bullet, these angles are calculated via the unit direction vectors of the player (the direction in which the player is aiming the gun). For the enemy, this is simply a fixed forward direction. The following equations are used to calculate the angles:

$$\alpha = \tan^{-1} \frac{z\ direction}{x\ direction}$$

$$\beta = \tan^{-1} \frac{y\ direction}{\sqrt{(x\ direction)^2 + (y\ direction)^2}}$$

For each subsequent render after the projectile has been fired, the projectile's position is incremented in the desired direction. The rate at which the projectile translates is dependent upon the product of the projectile's direction magnitude and a speed constant (the speed constant is a weight used to adjust the projectile's translation speed). This speed constant is used to create a different projectile speed for the spear and the bullet, as mentioned above. To simulate projectile motion, the Y direction component of the projectile object is decreased gradually in each subsequent render using the gravity constant. The aforementioned calculations computed for each projectile object results in the simulation of real world projectile motion in our game.

4.3. Sound Effects

Sound effects are implemented using the SDL_Mixer library. The advantage of the SDL_Mixer library is that it allows our game to play multiple sounds concurrently. We have initialized our SDL_Mixer to use an audio rate of 44100 Hz, the default audio format, and a channel value of 2 (stereo). Audio files that are used for sound effects are in .wav format. The audio files have either been created by group members, or have been downloaded via online free sources. Audio files are made available in the game by using the Mix_LoadWAV function to load the audio file into a Mix_Chunk object pointer. The Mix_LoadWAV function accepts the file path of the desired audio file as an argument. The audio file is then readily played by calling the function Mix_PlayChannel with the desired Mix_Chunk passed in as an argument. The following lists the sound effects that have been implemented in the game.

Object Type	Event	Sound Effects
Player	Upon player firing weapon	Upon firing a projectile, the player's weapon produces a gunshot sound.
	Wing flap	The player's wings are constantly in motion (please see animation section for more detail). As the player's wings flap, an audio file is played in the background to give the wings a flapping sound effect.
	Upon being hit	Upon being hit by an enemy projectile or enemy melee attack, the player emits a sound effect signifying that he has been hurt.
Enemy	Upon being hit	Upon being hit by a player projectile, the enemy emits a unique sound effect that signifies that the enemy has been hit.
Tower	Upon being hit by an enemy	Upon being hit by an enemy projectile or enemy melee attack the tower emits a sound signifying that it has been hit.
	Upon being hit by player projectile	The tower emits a unique sound signifying friendly fire.
Item	Upon player picking up item	When the player picks up an item, a sound effect is produced to signify that the item has been taken by the player.

4.4. Dashboard and Radar

The dashboard and radar are both implemented by giving each component (dashboard and radar) its own viewport. The following section gives a high level explanation of the implementation of the dashboard and radar interface. The following is a diagram of the dashboard and radar system implemented in the game:

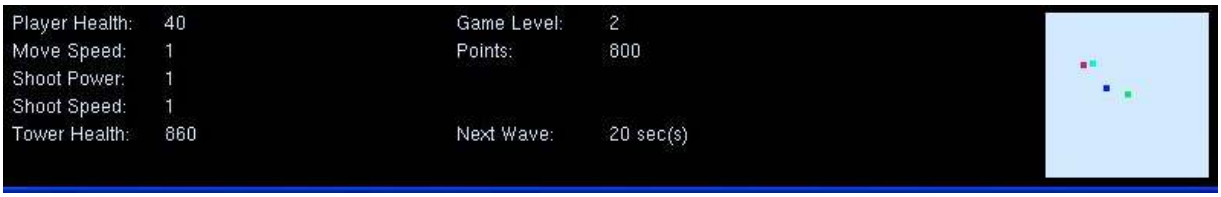


Figure 4.4-1 Dashboard

4.4.1. Dashboard

The dashboard displays to the user up-to-date game information. Information made available to the user on the dashboard includes player health, player speed, player shooting power, player shooting speed, tower health, current game level, points obtained, and the time left before the next wave of enemies arrive. Please see the above diagram of the dashboard for more detail. The dashboard information is rendered using a function that performs the following steps:

Specifies the location of the text to be rendered using glRasterPos2f
Renders the desired text using glutBitmapCharacter

4.4.2. Radar

The radar is a 2 dimensional representation of the game's environment. The radar allows the user to keep track of his/her position. The player object is represented by a green pixel and is always kept centered in the radar viewport. The radar also displays objects that are near the player. Each object's position is extracted and interpolated to be displayed in the radar (relative to the player's position). The following is a list of objects displayed in the radar with their corresponding colors. Please refer to the diagram located in the "Dashboard and Radar" section for an example of the radar GUI.

Object Type	Pixel Color Representation
Player	Green
Enemy	Red
Items	Light Blue
Tower	Blue
Projectile	Black

Note that the radar window also rotates in accordance with the player's rotation. For example, if the player's view were to rotate left, all pixels in the radar (that represent the different objects) would rotate in accordance with the player's rotation to maintain the player's relative position and orientation.

4.5. Object Modeling and Animation

Simple character models have been incorporated into the game to add dynamics. The models were drawn in blender, animated, and had the key frames exported into the 3ds model format. The open source LIB3DS library is used specifically for loading 3ds files. The lib3ds library comes with a sample code tutorial. We have implemented our object model loader functions based on the sample code. Two functions have been implemented for loading and rendering 3ds models, namely load_model(Lib3dsFile*& file) (for loading the models), and render_node(Lib3dsNode *node, Lib3dsFile*& file), used for rendering the model. Shown below is a screen-shot of the player model done in blender using bones.

Figure 4.5-1 Character in Blender

4.5.1. Flying Characters – Key Frame Animation

Both the player and enemies in the game have a wing flapping motion associated with them. This animation is done by loading few key frames of the same models drawn in blender into the program. At different times, the program determines which key frames would be rendered next based on the objects' current states. The animation starts all over again for a character when a character completes one flap motion, and therefore only a few key frames have to be loaded when a character is created.

4.5.2. Enemy Movement and AI

Upon creation, an enemy is assigned with a movement speed, its destination coordinates and the enemy type, which determines rather it processes range or close-combat attacks. Since an enemy's target can switch back and forth between the tower and the player himself, its direction of movement is calculated dynamically at every render loop relative to its current position and the destination.

In the game, the enemies are programmed to attack the tower structure as they spawn. However, they do have artificial intelligence implemented to direct them to react upon different circumstances. The sub-sections below describe how the enemies react when are hit or when the player is nearby, and also when there are obstacles that block their paths to the destination.

4.5.3. Player within Range

The enemies constantly check whether the player is around them. They become alert when the player draws within a certain distance and change their destinations accordingly. The enemies then chase the player for a fixed amount of time before turning their attention back to the tower again.

Depending on the enemy type, an enemy have a range value which determines if they are close enough to the player to fire a shot. For the range-attacking enemy, they can spot up from a distance and fire a spear in the player's direction. On the other hand, the range value of a close-combat enemy is fairly short. In order to save implementation changes for the two enemy types, close combat enemies fire out spears as well, but they are rendered transparent to achieve the close combat attack effect. Therefore, both enemy types' attacks can use the same collision detection technique to check if the player has been hit.

4.5.4. Collision with Bullet

Similar to how it would react when the player is within range, the enemies would chase after the player when it is hit by the player's arrow for a fixed amount of time, and after that, it changes its direction to the tower's location.

4.5.5. Going around Obstacles

As the number of enemies during the game play, they will inevitably run into each other going towards the same destination. A mechanism is implemented to ensure that these enemies would not simply collide with one another and stop their movement. In other words, they learn to change their direction of movement for a brief amount of time when the current direction is blocked by obstacles, until they find an opening to go to their desired destinations again.

This is achieved cross-checking an enemy's current direction of movement and the direction flags that are set in the collision detection algorithm. When a collision is detected, an enemy responds accordingly to the state that it is current in. The mechanism is designed so that when the enemy's forward direction is blocked, it always tries to go around by turning to either the left or right side. However, if even those directions are blocked, it would turn to the opposite direction of its original movement, hoping that its path would clear itself after a certain amount of time when it turns back. If all else fails, the enemy simply stops moving and check the possibilities again at the next render loop.

The mechanism performs fairly well as enemies were able to go around and find room close enough to attack their target in a short amount of time.

4.6. View Frustum Culling

To implement view frustum culling, we used a common method to determine whether or not objects were completely outside our frustum. Objects that are found to be outside the frustum are then excluded from rendering.

More specifically, we used the clipping space approach. The concept of this approach is to compare a point and the planes of the frustum in a normalized clipping space. The transformation to clipping space is accomplished by using the transformation matrix, A, found by the product of the model view matrix, M, and the projection matrix, P.

$$A = MP = \begin{bmatrix} a_{11} & \cdots & a_{41} \\ \vdots & \ddots & \vdots \\ a_{14} & \cdots & a_{44} \end{bmatrix}$$

The point,

$$p = (x, y, z, 1)$$

Will be transformed to

$$p_{clipping} = pA = pMP = (xc, yc, zc, wc)$$

Which is normalized to give

$$p_{normclip} = \left(\frac{xc}{wc}, \frac{yc}{wc}, \frac{zc}{wc}, 1\right) = (x', y', z')$$

We know that $p_{normclip}$ is inside the frustum as long as

$$\begin{aligned} -1 < x' < 1 \\ -1 < y' < 1 \\ -1 < z' < 1 \end{aligned}$$

Because in normalized clipping space the walls of the frustum are located at

$$\begin{aligned} x' &= -1, 1 \\ y' &= -1, 1 \\ z' &= -1, 1 \end{aligned}$$

Which, if not normalized, is

$$\begin{aligned} xc &= -wc, wc \\ yc &= -wc, wc \\ zc &= -wc, wc \end{aligned}$$

Therefore, we can find the equations of the frustum walls by writing $p_{clipping}$ in terms of the point p and transformation matrix A.

For example, to find the left plane,

$$\begin{aligned} xc &= xa_{11} + ya_{12} + za_{13} + a_{14} \\ wc &= xa_{41} + ya_{42} + za_{43} + a_{44} \end{aligned}$$

$$xa_{11} + ya_{12} + za_{13} + a_{14} = -(xa_{41} + ya_{42} + za_{43} + a_{44})$$
$$xa_{11} + ya_{12} + za_{13} + a_{14} + xa_{41} + ya_{42} + za_{43} + a_{44} = 0$$

$$x(a_{11} + a_{41}) + y(a_{12} + a_{42}) + z(a_{13} + a_{43}) + (a_{14} + a_{44}) = 0$$

Now, to test whether a point is to the right of the left frustum wall, $xc > -wc$. Therefore, the following inequality must be true,

$$x(a_{11} + a_{41}) + y(a_{12} + a_{42}) + z(a_{13} + a_{43}) + (a_{14} + a_{44}) > 0$$

Otherwise, the point is not in the frustum.

Performing the same check for all of the frustum walls allows us to know if an object is inside the frustum.

4.7. Blending

Blending is used in both the skybox and for objects in the game. The alpha value allows us to specify certain lighting parameters. The alpha value is used to combine the colour value of the fragment being processed with the pixel that is already in the framebuffer. The object will appear see-through. Blending will need to be enabled in order for the blending to work. It will also need to be disabled right after or else all other objects that are rendered will also blend together.

4.7.1. Blending Enemy

When the Enemy's HP depletes to 0, the Enemy object starts to gradually fade away. This effect is done via blending the Enemy's color with the environment's color. When an Enemy's HP drops to 0, the enemy color's alpha value, initially at 1.0, is decremented on subsequent renders by 0.02. The enemy object is eventually destroyed when the alpha value reaches 0 (when the enemy object has taken on the environment's color). In addition, during Enemy blending (when the enemy's HP is 0), we enable blending and disable depth test when we render the Enemy to produce the blending effect. After rendering the enemy (in each frame) we disable blending and enable depth test.

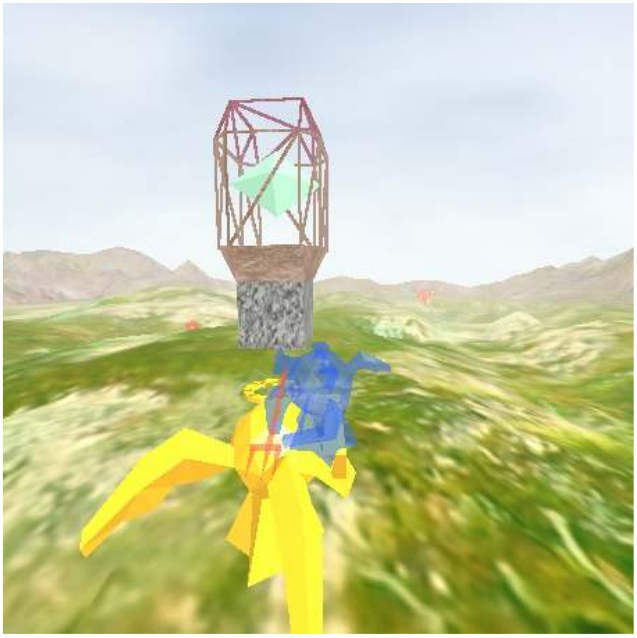


Figure 4.7-1 Enemy is dying and blends to the environment

4.7.2. Blending Clouds

The clouds on the outer sphere of the skybox are blended with the skybox cube to make the clouds appear translucent. After the cube is draw, Blending is enabled and depth test is turned off and the sphere is drawn. After the sphere is drawn, blending is turned on and depth test is turned on again. The clouds on the sphere will blend into the skybox cute and will appear translucent.

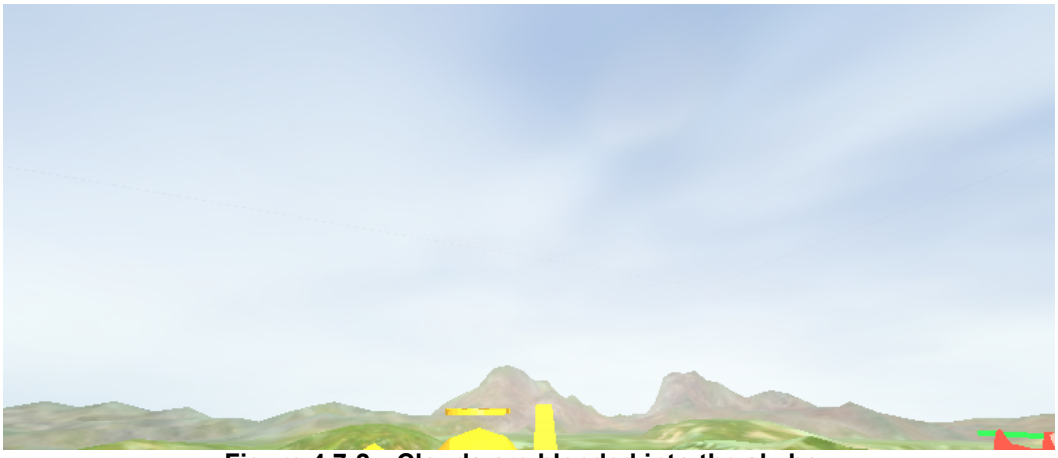


Figure 4.7-2 Clouds are blended into the skybox

Appendix A: Development Log

Initial Planning

- defending the castle
- download 3d max

goal: to defend a small tower with crytals
timeline: fantasy world

- like gaint rock man

tower life:
details:

- need to go out the tower to defend
- multi weapon
- attack players in the power
- weapons are outside the power so need to run outside
- u can buy amo

Stage 1: Defend around the castle

- 1 enemy
- 1 gun
- Defend around castle/tower

Stage 2: inside tower

- more enemy
- shoot at tower

Stage 3: building stuff
Things we need to do:

- collision
- surroundings
- shooting
- kinetics
- physics of

Sunday, March 09, 2008
Things done (Jen)
- work on getting sky effects on the background
- got sky box to work and texture mapped plane

Need to work on (Jen)
- Jpeg to texture map converter
- getting clouds effect

Saturday, March 08, 2008
-implemented collision detection algorithm, using quadtree approach for optimization (Clement)
-created first iteration of tower object (Jen)
-multiple button pressing capability logic (Neil)
-started implemented bullet events (disappears on collision with anything) (Clement)
-started implemented enemy events (at the moment disappears upon collision with bullet) (Clement and Billy)

Thursday, March 13th, 2008
- quadTrees, Enemy AI, collision detection (Billy & Clement)

Things to get done: (Billy)
- quadTree (ok)
- collision detection (ok)
  - check against all objects, perform action upon collison (ok)
- enemy
  - 3 spawn spots
  - walk towards one destination (ok)
  - walk towards player when near (ok)
  -stop when reach destination and attack (ok)
  - if hit bullet, descrease hp (ok)
  - if hit enemy, change direction
  - if hit player/tower, stop and attack

Saturday March 15th, 2008
(Billy, Clement, Jen and Neil)
- character modeling
- environment
- character aiming
- shooting bullet
- delete bullet when it goes out of bound (ok)
- enemy needs to turn when collide with one another
- set up viewport/radar
- moutain terrain

Sunday March 16th, 2008
-Enemy to Enemy Collision event (enemies now move away from each other upon collision) (Clement)
-Dashboard now displays player Health, and points scored (Clement)

Monday, March 17th, 2008
-Enemy to Tower Collision event, tower loses HP (Clement)
-Shooting: Bullets now come from the player (Clement)

Tuesday, March 18th, 2008
- Camera follows player and rotates with mouse motion. player direction and aiming is also rotated with mouse. (Neil)
- w,s,a,d are will move the player forward, backward, lateral left, lateral right with respect to the direction that the player is facing - straf logic. (Neil)
todo: -rotate players lateral direction during rendering to face same direction that the player faces.(Neil)
- create a gun for the player which rotates up/down with mouse movement. (Neil)

Thursday, March 20th, 2008
-Incorporated 3ds loader (Clement)
-Enhanced bullet object to be a projectile, mimics real life physics as projectile now rises and falls when shot, with its nose pointing accordingly. (Clement)
-included radar viewport (Clement)
-enemies turn red and fade back to original color when hit by bullet (Clement)

TODO: enemy spawning, different levels, radar, game player

Tuesday, March 25th, 2008

-added menu functionality (restart game, pause, exit). Still need to create headlines for menu options (Neil)
need to discuss:
- what is remaining
- go thur requirement list
- what else we want to do

-Frustum Culling implemented (though not sure how much computation this thing saves), multiplied ModeView with Projection matrix to get clipping matrix. Then calculated the 6 planes. Compared each object to the planes to see if we should render or not.
-Foundation for sound implmented, also some sound components have been added (such as bullet shooting, and player steps)

Saturday, March 29th., 2008
-Implemented fading (alpha blending) for enemies when they die (Clement)
-improved AI of enemy 1 (melee attack enemy) for upon collision with tower/player event. The enemy does not constantly decrement the player/tower's health, rather in small increments now, in accordance with attack motion.(Clement)

Tuesday, April 01, 2008
-fog (billy)
-radar upgrade to rotate and center player in middle of radar (billy)
-player weapon power increase when absorbing power upgrades (billy)
-ground texture (billy)
-terrain bumps and objects moving in accordance with terrain bumps (billy)

Sunday, April 05, 2008
-models implemented in blender (Billy)
-projectile objects implemented in openGL (Clement)
-lighting
-tower is complete with rotating diamond, and emission (Jen)
-menu finished, resume, restart, and exit options (Neil)
-Texture mapping on tower finished (Jen)
-Sky box implemented with rotating clouds using blending (Jen)

Tuesday, April 07, 2008
-Debugging (Everyone)
-Fine tuning of physical simulation variables, player speed, enemy spawn rate etc. (Everyone)

Sunday, April 13, 2008
- Game Balance (Neil and Billy)
- Lighting (Jen)
- Sound and Music (Clement)
- Report Compiling and Editing (Everyone)

References

[1] Basic Collision Detection:
http://pheatt.emporia.edu/courses/2003/cs410s03/hand27/hand27.htm

[2] SDL_Mixer Tutorial:
http://gpwiki.org/index.php/C:Playing_a_WAV_Sound_File_With_SDL_mixer

[3] LIB3DS Library:
http://lib3ds.sourceforge.net/

[4] View Frustum Culling:
http://www.lighthouse3d.com/opengl/viewfrustum/
http://www.crownandcutlass.com/features/technicaldetails/frustum.html