University of British Columbia

Department of Electrical and Computer Engineering

Final Project Report:

Space Invaders 3D

In Partial Fulfillment of the Requirements for:
EECE 478 Computer Graphics

Submitted by:

James Luo (63544043)
Sihanook Fernando (10891034)
Jason Kuo (21006036)
Johnty Wang (16234031)

TABLE OF CONTENTS

List of Figure

1.0 INTRODUCTION

The purpose of this report is to explain the game that was developed as a part of EECE 478 course requirements. The game application tries to encapsulate many 3D rendering and modelling concepts that were taught in class. SDL (Simple DirectMedia Layer) libraries were used along with OpenGL for the implementation.

The report will start off with a description of the game itself, followed by details on game play. The discussion will then move on to the technical aspects of the game which include the software architecture. Various features such as collision detection, audio and procedural modelling that were implemented will also be discussed. Optimization techniques were used in the game development to improve the efficiency in rendering the models. These will also be highlighted. The report will conclude with possible improvements that could be made to the game in the future.

2.0 GAME DESCRIPTION

2.1 What is the game?

The game is based on the popular 2D arcade game, *Space Invaders*, developed in the 70's by Tomohiro Nishikado. There have been several implementations of different version of the game over time. However, there have been few 3D implementations of this game. The implementation discussed in this report aims to produce a 3D version of the game, with a novel user interface that brings more realism to the player. The figure below is a screenshot of one of the 2D versions of the game.

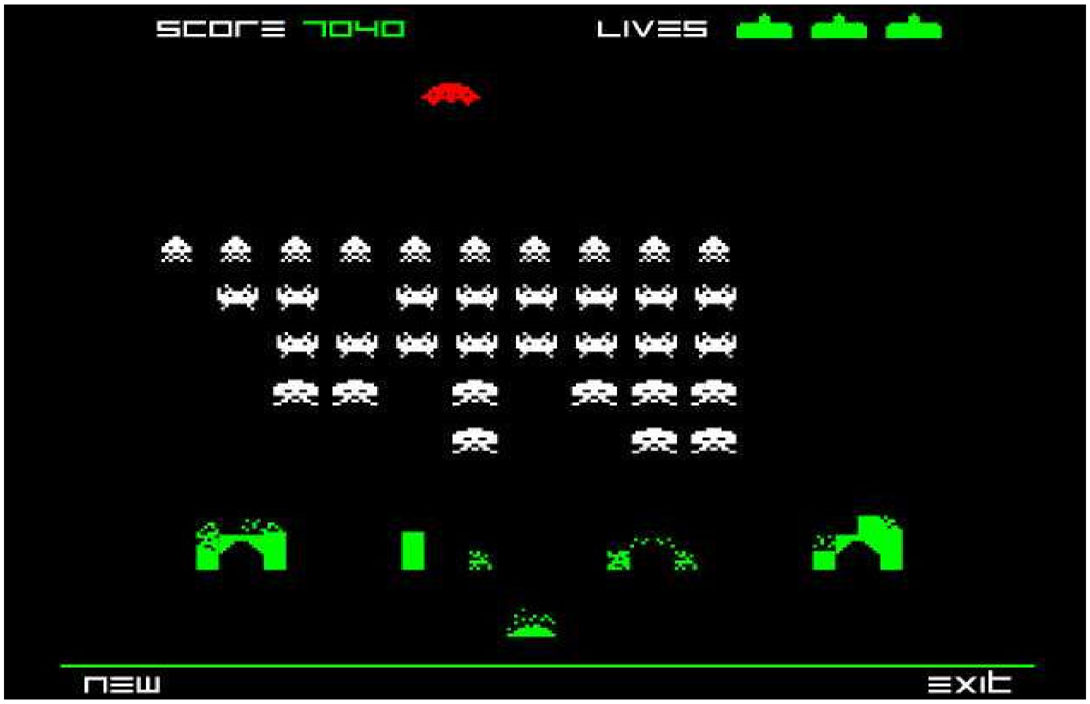


Figure 1: Screenshot of a 2D version of Space Invaders

2.2 Objective of the game

The objective of the game is simply to destroy a formation of advancing enemy spaceships. The player moves on to the next level upon destroying all enemy ships. Each consecutive level gets tougher and tougher with enemies shooting at the player more frequently. The number of points obtained per enemy destroyed increases with every level. The player should aim to maximize his or her score.

2.3 How is this implementation different from others out there?

The versions of *Space Invaders* out there are mostly designed the way shown in Figure 1. Our version of the game has three significant differences compared to the rest:

1. All the models are rendered in 3D space.
2. The game play is from the perspective of the player, rather than from a 3rd person perspective.
3. The game makes use of an infrared head-tracker and audio as input.

We believe that changing the perspective and adding another dimension to the game would improve the realism of the game. By tracking the player's real position adds a new level of involvement to the game play experience.

3.0 GAME PLAY & USER INTERFACE

We will now discuss details on the game play and how exactly the player interacts with the game.

3.1 User Interface

3.1.1 Audio

There are two types of audio used in the game. MP3 tracks for music, and WAVE files for sound effects such as shooting and explosions. The WAVE files are found on www.findsounds.com, an online audio search engine of publicly available sounds. The music was performed and recorded on a synthesizer keyboard by one of the team members.

3.1.2 User Input

One of the novelties of the game is the use of non-standard input methods. While mouse and keyboard input exist, they are used for debugging during development, and only serve as auxiliary forms of input in the final game. The main inputs required for game play consists of controls for player movement, and shooting. Given the parameters of the game, the movement is simply along a single dimension, and a single trigger for shooting is required.

Player Movement

An infra-red head tracking scheme was used to move the player to create an interesting effect: Since the game has a first-person point of view, as the player moves from side to side, the computer screen becomes a window into a virtual world with a perspective that depends on the real physical location of the player. As the player moves in the real world, the head tracker relates the physical movement to the virtual position of the player in the game.

To implement the head tracker, a Nintendo Wii-mote was used. The Wii-mote contains an infra-red camera, and is used in conjunction with the IR sensor bar for aiming at objects on screen. If the Wii-mote is placed at the screen, pointing towards a player equipped with an infra-red source, it is possible for the position of the player to be tracked. After some issues using an open source Wii-mote driver, the GlovePIE scripting environment was used in conjunction with a custom DLL to capture infra-red position data from the Wii-mote.

GlovePIE is a scripting program that can access most of the Wii-mote's useful features. However, there is no straight forward way of exposing the captured Wii-mote data with another program. GlovePIE does however have the ability to call an external dynamic link library (DLL). Therefore, a simple DLL with a shared memory section was used to expose the values acquired by GlovePIE, and the game simply has to link to this DLL and access the functions to retrieve the values. This allows simple two-way communication between the Wii-mote and the game application. In the main game loop, the player's position given by the IR data is queried, and the value is read by the game and interpreted accordingly.

Firing Mechanism

In most games, the trigger for firing is usually a key (such as space bar) or a mouse click. However, an audio triggered input was developed for this game. Initially it was intended for the firing to be triggered with a certain phrase such as “fire!”, but such analysis of the audio input would create complexities that are beyond the scope of this project. Therefore, the implemented method is simply a peak detection routine that issues a firing command if the current sample is over a certain level. Hence it is possible for the firing to be activated using any sound. However, as mentioned earlier, the ability to fire using a mouse/keyboard still exists if the player opts to do so.

3.1.3 Menu System

A menu system was developed for the player to interact with the game application. There are several menu screens available to the player presented at different situations in the game. These menus are discussed below.

Main Menu

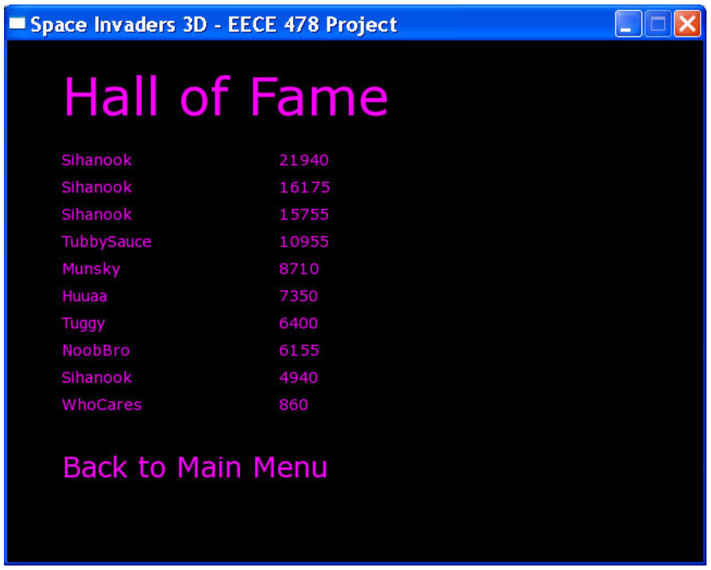
Upon starting the game, the player is presented with the main menu where the player can select one of three options:

- ***Start Game***

  Clicking on this option allows the user to start playing the game.

– ***Hall of Fame***

Clicking on this option takes the player to a screen which displays a list of all time high scores on the game along with the names of those who achieved them. A screenshot of this screen is shown below:


Space Invaders 3D - EECE 478 Project

Hall of Fame

Sihanook	21940
Sihanook	16175
Sihanook	15755
TubbySauce	10955
Munsky	8710
Huuaa	7350
Tuggy	6400
NoobBro	6155
Sihanook	4940
WhoCares	860

Back to Main Menu

Figure 2: Screenshot of Hall of Fame screen

– ***Quit Game***

This option can be selected by the user to exit the game application.

The Main Menu can also be reached by the player by pressing “Escape” key during game play. In that situation, however, the first option (“Start Game”) will be replaced by “Resume” for obvious reasons. A screenshot of the Main Menu shown at the start of the game is shown below.

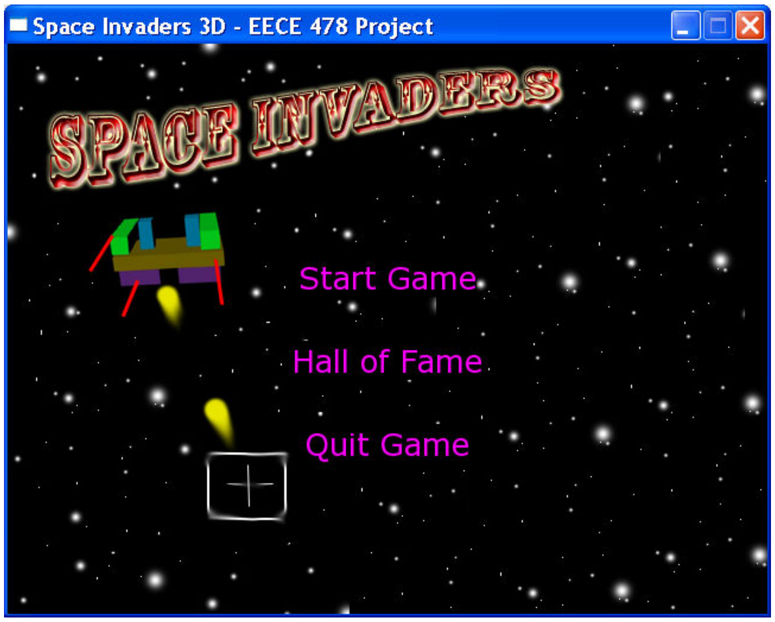


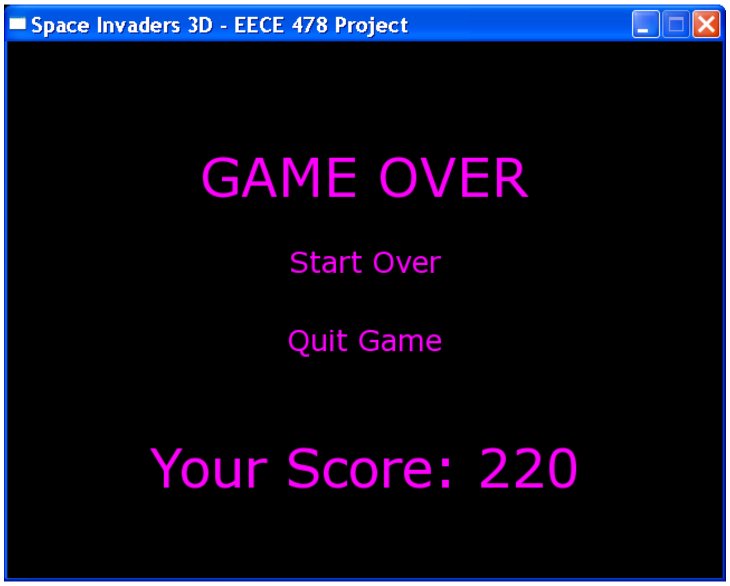
Figure 3: Screenshot of Main Menu shown at the start of the game

Game over Menu

This menu is displayed to the player when the game is over and the player has NOT obtained a high score. The game maybe over for either of two reasons:

- An enemy ship has landed
- Player has run out of lives (4 of them)

The player's score is displayed on the screen and the menu options allow the player to either start the game again or quit the game. A screenshot of the game over menu is shown below.

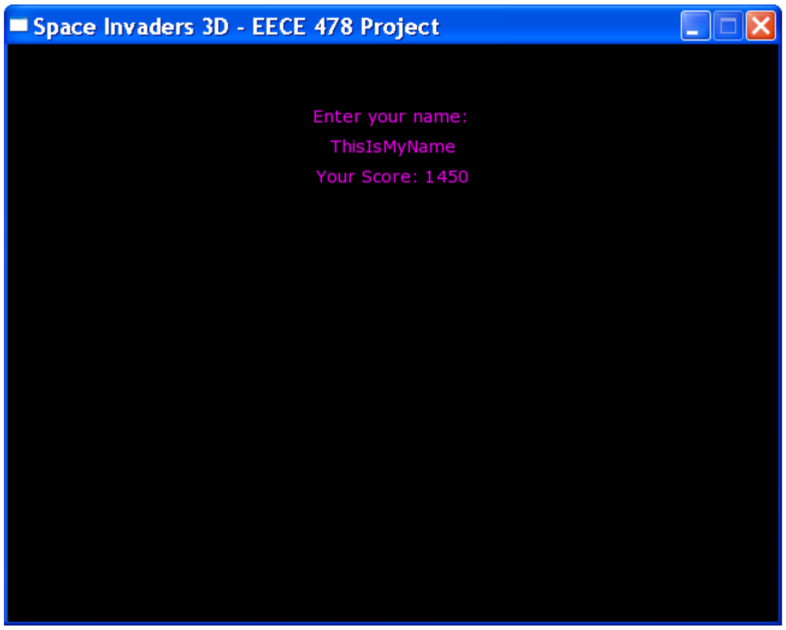


Your Score: 220

Figure 4: Screenshot of Game over Menu

High Score Menu

This screen allows the player to enter his or her name upon achieving a high score. The player enters his or her name and presses the Carriage Return key to save the data. The latest scores and names are shown on the Hall of Fame menu. A screenshot of the High Score Menu is shown below.



Enter your name:
ThisIsMyName
Your Score: 1450

Figure 5: A screenshot of the High Score Menu

4.0 SOFTWARE ARCHITECTURE

This section discusses in detail the technical aspects of the game's implementation.

4.1 Class Diagram

The following diagram illustrates the class hierarchy within our code.

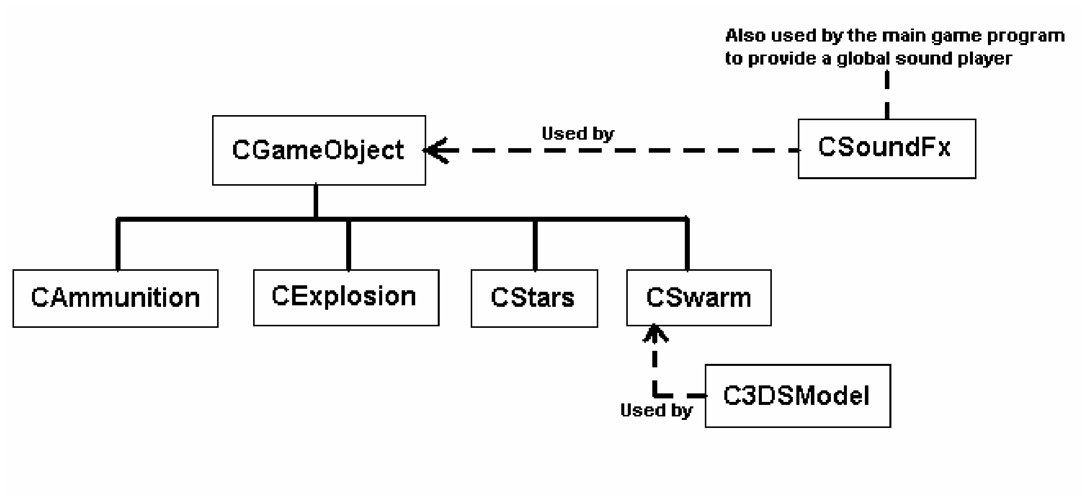


Figure 6: Diagram illustrating the different classes within the code

4.2 Class Description

Following are the descriptions of the various classes that were developed by our group.

- **C3DSModel**

  This class encapsulates the functionality to load 3D models and render them when required. The initial implementation performed the rendering on a vertex by vertex basis. As the number of models that the game contains increased, this approach proved to be extremely inefficient and therefore another approach was selected: vertex arrays.

  In order to render using vertex arrays, the C3DSModel class builds arrays of vertices, indices and colors when constructing its instances. When rendering, these arrays are used in appropriate OpenGL calls. Vertex

Arrays helped improve the performance of the game by more than 3 times. Vertex Arrays will be discussed in a little more detail under "Optimization Techniques Used".

Note that C3DSModel class still provides the functionality to render on a vertex by vertex basis if that becomes necessary for any reason.

In addition to loading and rendering 3D objects, C3DSModel class also contains the functionality to help with collision detection. More specifically, it builds a bounding box for the model when loading the model and provides an accessor to obtain the box's coordinates.

- **CGameObject**

  The game objects in the game refer to the major moving objects in the game. These include the enemy ships, enemy lasers, player's bullets, and explosion bits. The CGameObject class is a base class that is adopted by three other classes: CAmmunition, CSwarm, and CExplosion.

  In the CGameObject class, two pure virtual functions are declared. First, there is the function that helps generate the next-frame's coordinates of all game objects. This function, advanceTime(Uint32 dT), is the basis of our game's animation. dT is a measure of time in milliseconds between two frames. This is extracted with SDL_GetTicks(), which gets the number of milliseconds since the SDL library initialization. This value is passed to advanceTime function of other classes derived from CGameObject class to generate the object coordinates depending on each object's own speed.

  The other virtual function, renderObject(), is the rendering function that allows each class derived from CGameObject class to determine how its respective model object will be rendered.

- **CAmmunition**

  The CAmmunition class creates objects that are used both by the enemy swarm and the player. Two types of ammunition are provided in the ammunition class: a laser like weapon representing the enemy fire, and a sphere-shaped bullet representing player's game ammunition. By calling either fire(…) or enemy_fire(…) public functions in the class, each different ammunition types can be initialized.

Similar to CSwarm class described next, CAmmunition class uses a linked list of structs to manage all the bullets/lasers in the game. The struct is shown below.

```
struct gameBullet
{
    GLfloat posX;
    GLfloat posY;
    GLfloat posZ;
    gameBullet * nextBullet;
};
```

posY is the same thoughout the game as the player is only allowed one degree of freedom: x-direction. The player is not allowed to move up or down. posX determines where bullets are fired from and posZ deteremines how far bullets are moved as the game animates. Ammunition moves only in z-direction and the speed with which the enemy bullets move changes as the game progresses through levels, thus allowing us to vary the level of difficulty between levels.

One extra attribute is assigned to the CAmmunition class which contains the following information:

```
struct BulletBounds
{
    GLfloat xMin, xMax;
    GLfloat yMin, yMax;
    GLfloat zMin, zMax;
};
```

The BulletBounds define the dimension of the bullets, which will be used while rendering to check for a target hit.

Type 1 enemy ammunition has a long bounding box since the laser is defined as a long cylindrical object. Type 2 player ammunition has a cubic bounding box that encloses the spherical bullets. When the bounding box collides with another object's bounding box, say enemy swarm, the ammunition object will be removed from the link-list, and enemy object is assigned the invisible state (thus essentially destroying the enemy). Please refer to the discussion on collision detection for further information.

Unlike in the CSwarm class, the number of enemy ships generated is limited by our compile time game constants, we can pass in different values to enemy ammunition initiating function, enemy_fire(…), to define how frequent the enemy fires a shot at the player. As level increases, more enemy bullets can exist at once, making the game harder to play. The maximum number of bullets is defined in enemy_bullets(). As the different level changes and game playing time elapses, the maximum number of bullets increases.

The game is built in such a way that different ammunitions do not annihilate each other, and ammunition objects are destroyed if they pass a certain Z bound in the game's world coordinate space.

- **CSwarm**

Swarm refers to the enemy ships that need to be destroyed in order to score in the game. The swarm class controls what enemy ships to load, and how those enemy ships move in the game.

The 3D model that represents each enemy in the swarm and the initial properties of each enemy ship in the class is first created when the class is initialized. A total of 50 ships are created as a linked list, represented by 10 columns and 5 rows. Properties of each ship are stored in a struct which is shown below:

```
struct enemyObject
{
    GLfloat posX;
    GLfloat posY;
    GLfloat posZ;
    bool isVisible;
    enemyObject * nextEnemy;
};
```

It is these structs that form the linked list. The struct describes whether the ship is visible, its coordinates, and the next ship in the linked list.

The boolean variable “isVisible” assigned to each enemyObject allows the program to determine if the enemy ship is hit, or if it should fire a laser. An invisible state deems the object as being “destroyed” and thus it doesn’t need to be rendered. This approach, rather than destroying the model object altogether, means that we don’t need to load the models again and thus saves processing time.

Movement of the swarm is quite simple. It moves across the screen. When it passes a certain bound, it will move closer to the user by one ship’s depth. However, speed of the swarm moving across the screen increases as they advances forward. Starting at 100pts/sec the horizontal speed inreases by 30% each time.

The class also has a regenerateEnemies() function which is used to recreate the swarm after the user advances to the next level or restarts the game.

- **CExplosion**

This class is derived from CGameObject class and implements a particle system that simulates explosions. The constructor of the class allows us to specify the location of the explosion. It then creates a series of vertices at the exploding location and stores it in a custom array along with random directions in which the particles move.

Just as any other class that is derived from CGameObject, this class provides an advance time and a rendering function which allow us to move the particles in space and then render them on to the screen. The class also stores a fading factor which decreases the visibility of the particles with time. Once they become completely invisible to the user, the main program can destroy the CExplosion objects.

Note: Invisibility of the particles in this case is simply achieved by gradually turning them black. Since our background is mainly black, this allows us to achieve invisibility of the particles without much processor overhead that would be involved if we were to use Alpha Blending.

The following is a screenshot of explosions that occur during game play when an enemy ship is destroyed.

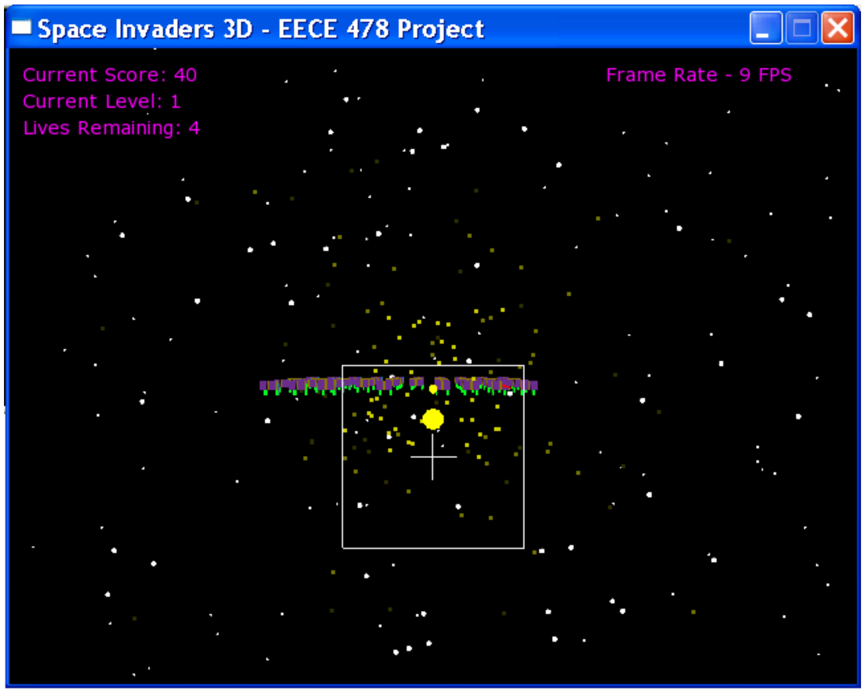


Figure 7: Screen of Explosions when enemy ships are hit

- **CStars**

Since the game is about "Space Invaders", a starry night sky would be a fitting background. Instead of using a static image, it was decided to procedurally generate the stars to allow for more interesting effects. This way, they could be dynamically modified to create more realism and intrigue. A few groups of stars, classified by size, are randomly scattered across the 'sky'. The stars also "twinkle" at random intervals, simulating the effect seen by an observer on a planet with an atmosphere. The entire sky also rotates slowly (although at a much faster rate compared to what is observed on earth) to simulate another physically observed phenomenon.

The positions are initialized as random X and Y values in the range that is visible to the player, at the same Z distance. To implement variations in the star size, 3 distinct values are used from 1.0 to 3.0. A twinkling effect is achieved by randomly changing the size of a particular group of stars.

The CStars class implements the star field background used in the game. Four groups of stars, 2 small, 1 medium and 1 large are implemented as lists of positions. The positions are vertices with random X and Y values corresponding to the visible part of the 'sky' as seen by the user, and a constant Z position that defines how far away the sky is. After certain frame intervals, a random number is generated for each group of stars. If the number is above a certain value, the size of that star group is modified before rendering. This effectively creates the 'twinkling' effect. The gradual rotation of the entire 'sky' is implemented with a simple rotation transformation before rendering the star field.

- **CSoundFx**

For the playback of music and audio, the CSoundFx class was created. This class uses SDL_MIXER to load and playback audio assets. WAVE files for effects are loaded from disk during initialization and stored within the class, since they are relatively small and take little memory. Additionally, it is important that they can be accessed very quickly on demand since the timing of the effect is crucial. For music files, the timing requirement is not as stringent, and the MP3 files are loaded from disk when the track is requested. The filenames are stored in the game constants header file, and are loaded by this class on initialization. Any game object that requires the use of sounds can instantiate an instance of this class to implement audio output capabilities.

4.3 External Libraries

We have used several external open source libraries for developing our version of *Space Invaders*. Following is a list and a brief explanation of the various libraries used.

- **SDL_Mixer**

  This library handles playback of WAV and MP3 files. It dynamically allocates and de-allocates channels for audio playback. Up to 8 channels of 16 bit stereo audio is supported, as well as a single channel of stereo music.

- **SDL_AudioIn**

  Provides audio input support for SDL. Used to capture samples from the microphone to implement sound triggered shooting.

- **SDL_Image**

  This library is used to load files of different types to the game for texturing purposes. This saves us the time to write various custom functions that would do this.

- **Lib3DS**

  This library is used to load the various 3D models that our game requires. We selected the 3DS file format for the models used in our game as there are lots of resources available on the internet to handle 3DS files. The Lib3DS library takes care of loading the various meshes and error handling and, therefore, allows us to concentrate on the more important aspects of the game.

- **FreeType**

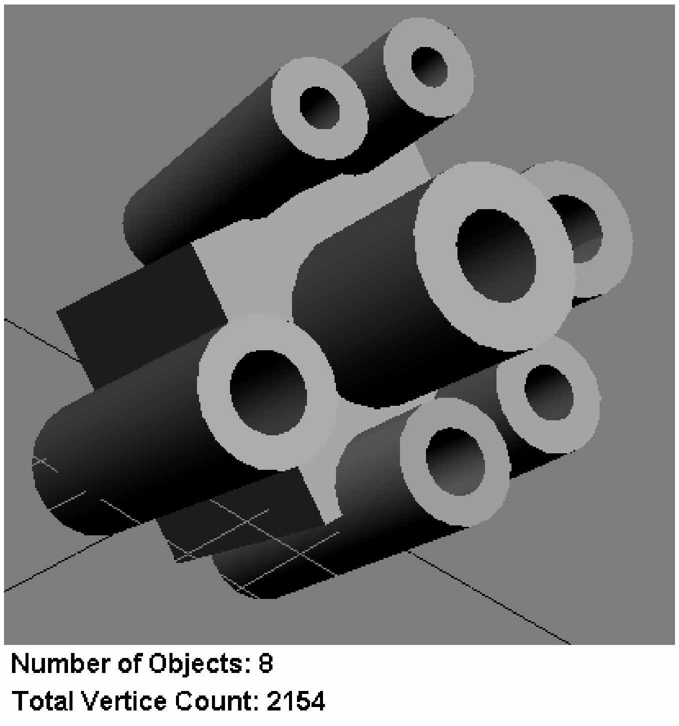
  This is a highly customizable and efficient font engine that is used by the “SDL_TTF” library described below.

- **SDL_TTF**

  This library allows us to render fonts on to the screen. It provides sufficient features for us to be able to render fonts for our game menus as well as for showing the status of the player on screen.

4.4 3D Models

There were a few attempts at modelling the enemy ships for the game. The figure below illustrates our first attempt along with information about the model:



Number of Objects: 8
Total Vertice Count: 2154

Figure 8: Initial Enemy Model

As can be seen from the figure, although the model looks appealing, it contained 2154 vertices which proved to be too many when considering the fact that we had 50 such models to be rendered each frame in addition to the rest of the graphics that make up the game. Therefore, we made another attempt at designing a model. This is shown below.

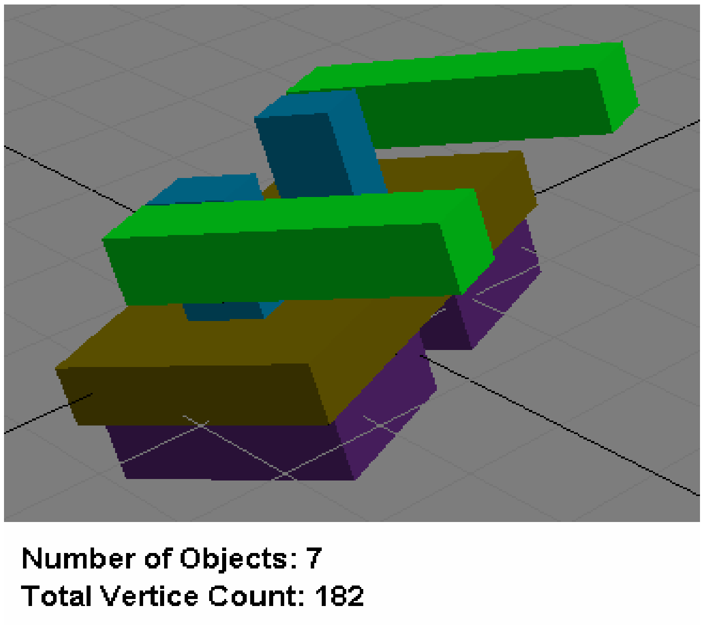

Number of Objects: 7
Total Vertice Count: 182

Figure 9: Final enemy model

As can be seen, this is a huge reduction in the number of vertices that needs to be rendered per frame. This design is based on simplicity, with only a body and 2 legs. Upon using this model, we noticed a large increase in our frame rate, which seemed to be acceptable to us. Therefore, we decided that this was to be our final model for the enemies.

4.5 Audio Triggered Firing by Player

Implementation

To implement the audio activated firing mechanism, audio input capabilities are required. The SDL_audioin library was used. After initialization, the audio input triggers a call-back function when the specified number of samples has been captured by the input. The sum of squares (effectively the power of the signal) is then found, and compared to a reference value. The reference value is found by trial and error due to the dependence of the captured level on a number of parameters outside of immediate control (microphone/sound card specifications, system sound level settings, etc). In a more thorough implementation, a calibration feature could be used to find this value.

Latency

Since the audio input call-back function is triggered only after a certain number of samples have been captured, there will be some latency between the voice command and processing of the sample. Given 512 samples per capture, there will be a latency of at least 11ms for a sampling frequency of 44100Hz. Due to inherent latencies of the WDM driver used by Windows sound cards, the actual latency is usually higher (and not constant). There will also be a slight delay while the signals are summed up, but this was found during debugging to be less than 1ms. Overall, the latency was qualitatively found to be acceptable.

Issues

While amusing and certainly entertaining, there are certain drawbacks with audio as a form of input (especially for shooting). One of the major problems with detecting audio input power as a firing mechanism is its dependence on the volume of sound effects from the game. If sound effects are louder than the capture threshold, audio from the game can trigger unintended firing. A compromised solution is to turn down the game audio, or to simply require the player to wear headphones.

4.6 Enemy Firing

The speed of enemy ammunition needs to be altered as the game progresses in order to affect the level of difficulty observed by the player. Therefore, we developed the game such that an enemy that is picked at random from the remaining enemies fires with a speed that is directly dependent on the duration of the game. Then, as (1 / game level) seconds goes by, the total number of enemy ammunition allowed increases by 1. So at level 1, there will be 1 enemy bullet for the first second, 2 enemy bullets for the next second, and so forth. Since we determine the way enemy shoots by means of maximum enemy bullets allowed instead of the number of enemies remaining, there will be a massive amount of fire power generated when there are little aliens left in the game. So never underestimate the power of the last remaining enemy! The flowchart below illustrates this sequence.

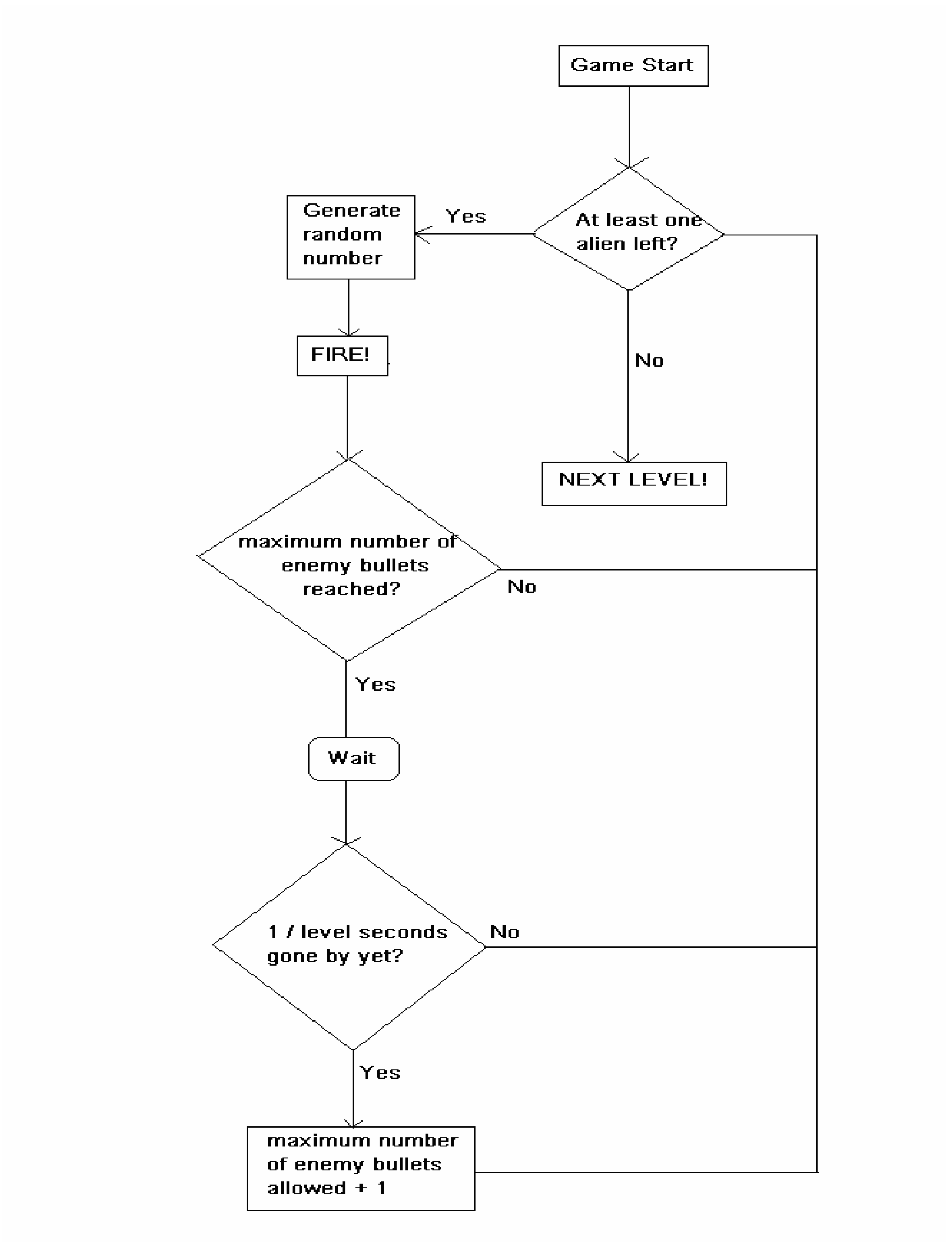

Game Start
At least one alien left?
Yes
Generate random number
FIRE!
No
NEXT LEVEL!
maximum number of enemy bullets reached?
No
Yes
Wait
1 / level seconds gone by yet?
No
Yes
maximum number of enemy bullets allowed + 1

Figure 10: Flow-chart showing enemy fire sequence

4.7 Handling Menu Interface

The menu options on screen can be selected by the player by clicking using a mouse. This is achieved by using OpenGL picking technique.

A separate drawing function was developed to draw each menu screen. Each clickable menu item is named using the functions provided by OpenGL. Upon clicking on the screen, a small portion of the screen around the location where the mouse was clicked is redrawn, keeping track of which objects are drawn at the location based on the name assigned to each item. This allows us to figure out which menu item was clicked and thus take appropriate action.

The following is a snippet of code showing OpenGL picking that was used by us to translate mouse clicks on the menus.

```
void handleMenuClick(int x, int y)
{
   GLuint buff[64] = {0};
   GLint hits, view[4];
   glSelectBuffer(64, buff);
   glGetIntegerv(GL_VIEWPORT, view);
   glRenderMode(GL_SELECT);
   glInitNames();
   glPushName(0);
   glMatrixMode(GL_PROJECTION);
   glPushMatrix();
      glLoadIdentity();
      gluPickMatrix(x, y, 1.0, 1.0, view);
      glOrtho(-100.0f, 100.0f, 0.0f, 100.0f, 0.0f, 100.0f);
      drawStartMenu();
   glPopMatrix();

   hits = glRenderMode(GL_RENDER);
   GLuint hitName = buff[3];

   if (hitName == 7)
   {
      //... perform action here for item named "7"
```

```
    }
    else if (hitName == 14)
    {
        //... perform action here for item named "14"
    }
    //... do the same for other items
}
```

4.8 Collision Detection

In our game, there are mainly just two objects that generate ammunition: the player and the enemy alien. When starting the game, all the enemy models are loaded and their bounding boxes are calculated. A bounding box is also associated with the enemy and player bullets. To detect if an enemy is hit, all the needs to be done is to determine if the bounding box of the bullet intersects with that of any of the enemies. The following snippet of code from the detectEnemyHit function illustrates the process further.

```
while (curBullet) {
        nextBullet = curBullet->nextBullet;

        // locate the bounding box of the bullet to the
        // current bullet location
        GLfloat bMinX = curBullet->posX + bulletBox.xMin;
        GLfloat bMaxX = curBullet->posX + bulletBox.xMax;
        // ... similarly for bMinY, bMaxY, bMinZ, bMaxZ

        while (curEnemy) {
            if (curEnemy->isVisible) {
                bool isHit = false;
                BoundingBox enemyBox =
                        Swarm->swarmEnemyModel->getBoundingBox();
                // locate the bounding box of the enemy to the
                // current enemy location
                GLfloat eMinX = curEnemy->posX + enemyBox.xMin;
                GLfloat eMaxX = curEnemy->posX + enemyBox.xMax;
                // ... similarly for eMinY, eMaxY, eMinZ, eMaxZ

                // Detect Collision Between Bullet and Enemy
                if (bMinX > eMinX && bMinX < eMaxX) {
                    if (bMinY > eMinY && bMinY < eMaxY) {
                        if (bMinZ > eMinZ && bMinZ < eMaxZ) {
                            curEnemy->isVisible = false;
                            isHit = true;
                        }
                        else if (bMaxZ > eMinZ && bMaxZ < eMaxZ) {
```

```
                        curEnemy->isVisible = false;
                        isHit = true;
                    }
                }
                else if (bMaxY > eMinY && bMaxY < eMaxY) {
                    // Similarly check if bullets' max X lies inside
                    // the enemy
                }
            }
            else if (bMaxX > eMinX && bMaxX < eMaxX) {
                // Similar to above code not shown here
            }
            if (isHit) {
                // Enemy has been hit
            }
        }
        // Move onto the next enemy
        curEnemy = curEnemy->nextEnemy;
    }
    //.. Start over with the enemies and the next bullet
}
```

While the above is to detect elimination of enemy, to determine whether the user is hit or not, we need another function. This function is called detectIamHit(). This function does exactly the same thing as detectEnemyHit except that it checks whether an enemy bullet reached an area around the player's current location.

4.9 Optimization Techniques Used

There were several optimization and smoothing techniques that were used to ensure that the game play was possible efficiently. They are discussed below.

1. **Variable Frame Rate Rendering**

   Since the performance of such a game is dependent on the system's capabilities (especially the graphics card), it is likely to see different frame rates on differently equipped systems. In order for the game dynamics to operate at a constant speed, the time step between each frame render is found. This change in time between frames is used to update the game objects to ensure close to constant movement rates at different rendering rates. On a faster system the time between frames will be small, and so will the movement between each frame; On a slower system, the time between frames will be larger, causing larger changes (less smooth) from one frame to the next. It is possible that on a slower system that the time steps are so great that the game is no longer playable. Care must be made to ensure that the game runs at reasonable rates on most modern systems.

2. **Vertex Arrays**

   The game requires that lots of models be rendered on the screen simultaneously. If the rendering of these models were done on a vertex by vertex basis using a multiple number of OpenGL calls, the efficiency and performance of the game suffers heavily as we discovered. Vertex Arrays allow us to replace hundreds of OpenGL calls with only a very few calls that work on arrays of data related to the vertices. This improves the efficiency greatly and in our case, we discovered that the frame rate more than tripled after introducing Vertex Arrays.

3. **Model Complexity**

   With the original unnecessarily complex models (which are very computationally challenging for integrated graphics cards), it was shown that the frame rate was less than 10 frames/sec, which made the game unplayable. After the new models were introduced, the frame rate went up to over 40 FPS, even on a laptop's integrated graphics card.

5.0 Recommendations

There is a lot of room for improvements in this version of the game. This section discusses some of them.

- **Better Models**

  The models used for the enemies in this version of the game is quite simple and don't really look all that appealing. The time constraints and our need to stick to simple models to achieve a high rendering speed limited us in this respect. However, it must be mentioned that better models can be realised without affecting the speed of rendering too much, given enough time.

- **Rendering based on level of detail**

  It is possible to achieve greater performance with more complex models if we control the level of detail in models being rendered depending on where they are located in world space.

  For example, we could have models with different numbers of vertices. When the models are far away from the player, we could render the model with the least number of vertices and as they get closer to the player, we could gradually use models with higher number of vertices. This could potentially improve the experience of the player while making sure that performance is not greatly affected.

- **Occlusion Culling and Clipping**

  It may be possible to achieve greater performance by implementing an algorithm to detect and not render vertices that are not visible i.e. covered by other polygons. We believe that this potential exists since there are a large number of enemy models rendered per frame and most of them are covered by those models that are closer to the player.

- **Lighting**

  The current implementation does not utilize any of the lighting capabilities that OpenGL provides. We felt that it was possible to achieve a good player experience without introducing any complexities associated with lighting.

  However, introducing material properties along with lighting could improve the experience. For example, specular lighting properties would improve the laser firing of the enemy bullets. Explosions could also be done more realistically if lighting was introduced.

- **More options during Game Play**

  There are many options and features that can be added during game play. For example, shooting certain enemies could provide higher scores. Objects could be programmed to fall from outer-space that provides bonus points and special capabilities to the player such as better weapons. The concept of a "mother ship" that was a part of the original "Space Invaders" could also be introduced to the game.

6.0 Conclusion

We believe that this game has most of the important features that make up an enjoyable game. Some of the features are explosion effects, sound effects and a good visual environment. Needless to say, however, that there is a lot of other features that can be added and further improvements that can be made.

It must also be said that this implementation of “Space Invaders” is very much different from the original version, both visually and in the way it is played. Therefore, even though the game is based on the original idea of “Space Invaders”, we believe that this version has enough differences to qualify as a new game that preserves some of the elements of the original version.

Appendix A: System Flowchart

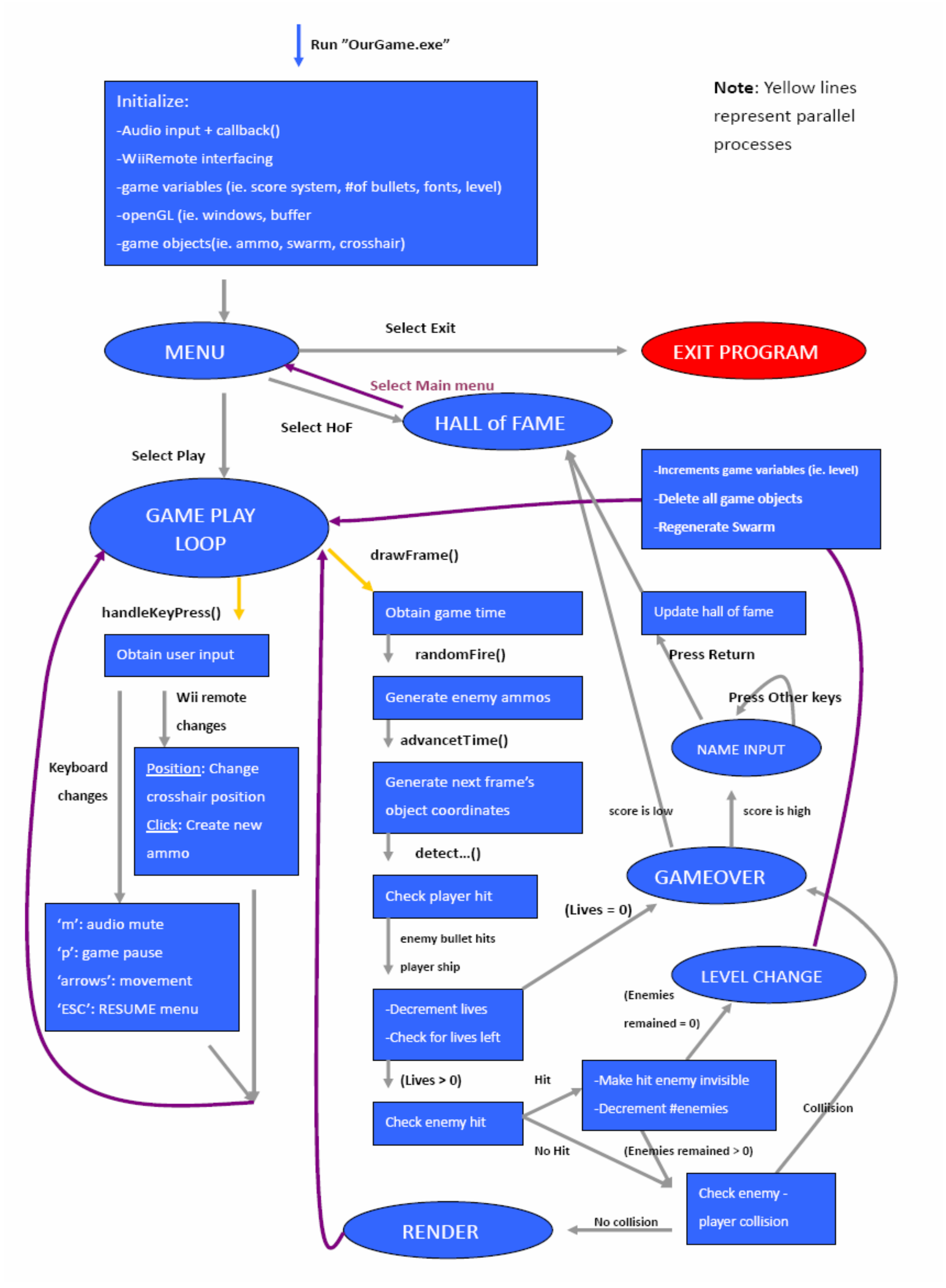


Figure 11: Flow-chart Showing Overall Game Design