University of British Columbia

EECE 478 – Computer Graphics

Game Design – 3D Labyrinth

Phoebe Hsu

Neville Kwong

George Lee

Patricia Mo

Owen Yang

Date submitted: April 14, 2008

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF ABBREVIATIONS

GDC	GameData class

1.0 INTRODUCTION

This project discusses the development of an action puzzle solver computer game, 3D Labyrinth, by using OpenGL. In this game, player will be asked to use mouse to rotate the box to move the ball from the starting point to the ending point. Our goal of this project is to learn the game developing process and create a final product that is both exciting and fun to play. Contrary to the traditional labyrinth game which has only one layer, our maze will consist of multiple layers that will allow bigger motion. To imitate the feeling of the drop-down motion of a ball in the real world, complicated physics engine is also implemented.

The report divides into the following primary sections:

- Game Architecture
- Game Engine and Game Logic
- Calculation Module
- Front End
- RTU
- Future Improvement

2.0 PURPOSE

The purpose of the project is to utilize OpenGL API to create a 3D computer game. The game that we chose is a 3D version of Labyrinth. The player has to rotate a 3D maze to maneuver the ball to a set destination while avoiding obstacles. The 3D nature of the game makes the game a lot more difficult and more enjoyable then the original 2D version of the game.

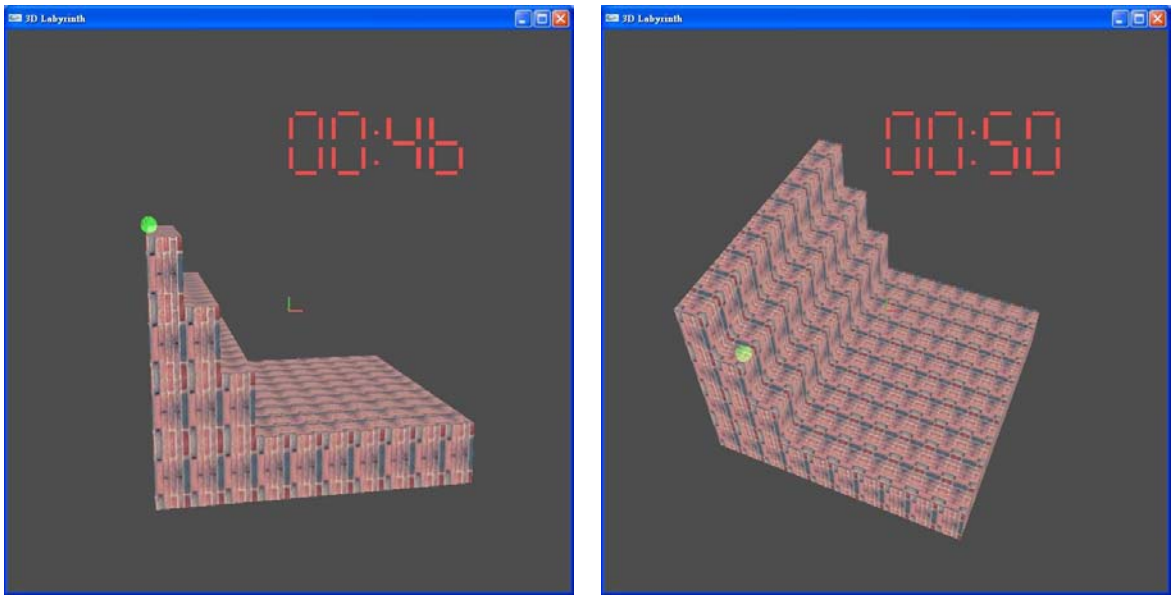

Figure 2.1 Screenshots of 3D Labyrinth

3.0 GAME DESCRIPTION

The 3D Labyrinth game is based on traditional labyrinth game where the user yaw the board and move the ball to the destination. Unlike the traditional one, the 3D Labyrinth is built in 3 dimensions and the cube is composed of multiply layers. The user can yaw, pitch, and roll the cube to move the ball to the destination. The objective of this game is to move the ball to the destination as fast as possible. Through this process, the user has to maneuver the cube to avoid letting the ball fall out of the cube. The user can control the maze's movements by hold down the left or right mouse button and drag the mouse in any direction. The game will also be limited by a countdown clock to increase the "fun" factor.

Game Feature:

- Scripting Language: The levels of the game will be scripted.
- Physical Simulation: The user will be rotating the box around, therefore gravity and perhaps other forces will apply.
- Key Frame Animation: The movement of the balls may need key frame animation.
- Collision Detection: The ball will collide with the walls of the cube.
- Sound Effect: All games must have some sort of sound effect.

4.0 GAME ARCHITECTURE

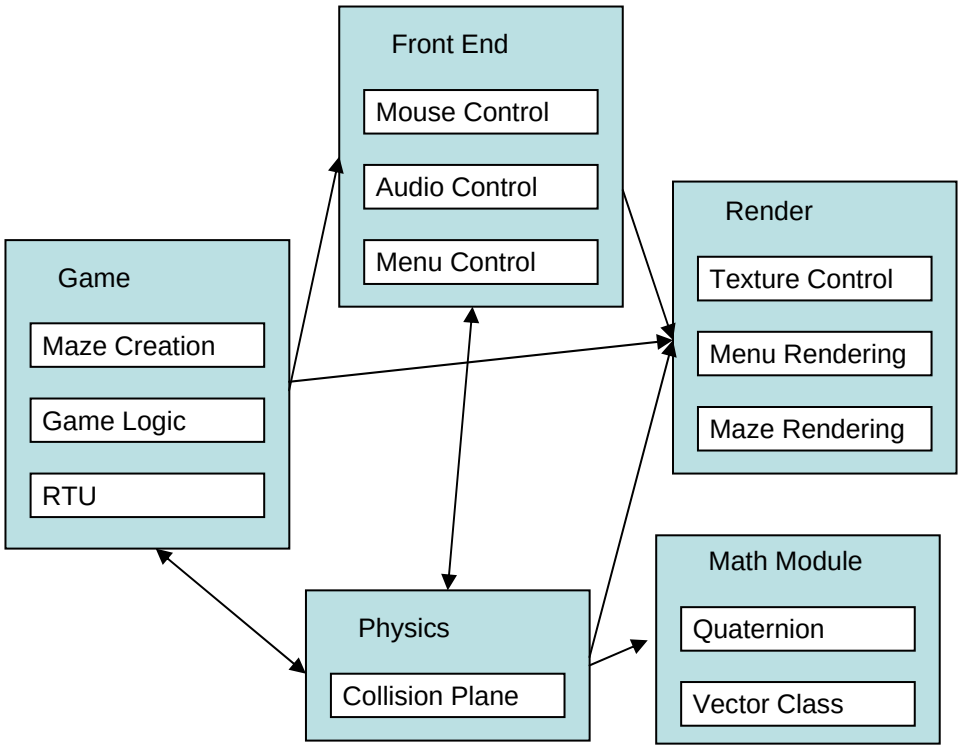

Figure 4.1 Game Architecture

5.0 GAME ENGINE AND GAME LOGIC

The GameData Class object stores data such as the ball's location, the game level, the maze information, the ball's kinematic characteristics, the time elapsed, etc. Level description is defined by a simple text file, which the GameData class is responsible for parsing. The parsed level information is stored in an array that will be used later by the Physics engine and the Render engine.

GameData Class SetMaze()
This function parses the input text file that contains the maze information, and stores it in a three dimensional array. It also initializes and sets up some environment variables for the game.

The Maze in a Text file
Here is an example of a Maze input file:

```
SIZE 5 5 1
LAYER //this is plane layer 1
&BBBBB
&BBBBB
&BBBBB
&BBBBB
&BBBBB

LAYER //this is plane layer 2
&BBBBB
&BWWsB
&BBWWB
&BeWWB
```

&BBBBB

This will create a 5 x 5 x 1 (Width x Depth x Height) maze.

Parsing Symbols Explanation:

&: a token to notify the parser that it's starting to parse a new row

B: a block that is a solid cube with 6 non-passable planes.

W: an empty space where the ball can pass through freely

s: starting location of the ball

e: ending point. When the ball comes into contact with this block, the game ends

6.0 CALCULATION MODULE

6.1 Physics Engine

Physics class handles the movement of the ball inside the maze. Physics class is activated in a set time interval (16ms). Collision handling, force and kinematics calculations of the ball are all done within the Physics class. Parameters of the world are gathered from the GameData class (GDC) object. Newly calculated values are returned back to the GDC object every time Physics Class finishes. The block diagram of the Physics Engine is shown in Figure 6.1, and each function will be briefly discussed below.

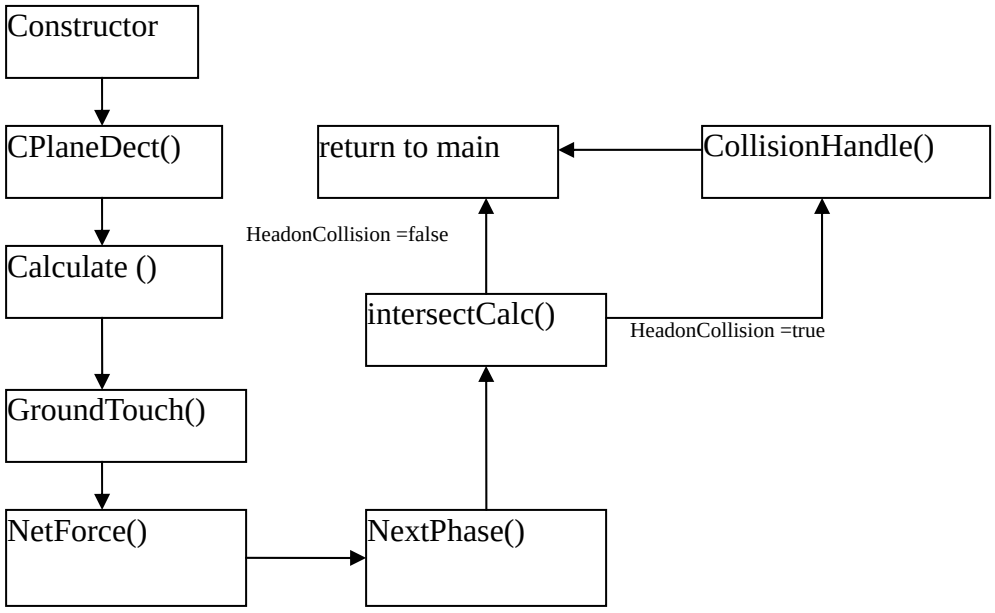

Figure 6.1 Block Diagram of the Physics Engine

CPlaneDect()

This function keeps track of all the planes that are not passable by the ball. These planes are called “Collision Planes” and are stored as CollisionPlaneClass objects.

The CplaneDect function loops through the maze array generated by the GameData class and store all of these collision planes in a local array. The array is unchanged throughout the entire runtime of the game.

Each of these CollisionPlaneClass object consists of a normal vector of the plane, and also the mid-point of the plane.

Calculate ()

The calculate function rotates the gravity vector by the conjugate of the rotation quaternion of the maze.

Instead of having a constant gravity vector with a moving maze, the Physics class treats the game world as having a moving gravity vector with a constant maze. The reason for this is that it would take a lot more resources to keep track of all of the collision plane positions and normal vectors if the maze was moving. By keeping the maze motionless, we only have to generate the collision plane array once for every game session. Although the Physics class treats the maze as stationary, to the player of the game, the maze is moving according to their mouse input. This illusion is done in frontEnd and Render class, and will be described in greater details in their respective sections.

Since the rotation of the maze is done by quaternion matrix transformation of the modelview matrix, the gravity vector is rotated by the maze's rotation quaternion conjugate so that it would always appear to the player that the gravity points downwards (negative Y direction).

GroundTouch()

This function first determines if the ball is currently being pulled against a surface by gravity, if it is, it will then determine the normal force vector of the ball. The

RayToPlane() function is used here to determine if any point on the surface of the ball is touching a collision plane by passing in a vector that has the length of the radius of the ball and is pointing in the gravity's direction. If no touching point is found, the ball is in the air. If a touching plane is found, a normal vector is obtained. This procedure is repeated 3 times for each of the X, Y, Z components. This covers the cases where the ball is concurrently touching multiple planes (possible maximum number of contact planes is 3, but this function will work with any number of contact planes). The XYZ components are added together in the end of the function and magnified to the magnitude of the gravity vector to obtain the normal force vector.

NextPhase()

This function handles all of the kinematic calculations of the ball. It computes the acceleration, the velocity, the resulting displacement and the final location of the ball assuming that no collision will happen. The kinematic formulas we used in this function are listed below:

$$a = \frac{F_{net}}{m}$$

$$V_{next} = V_{previous} + a \cdot dt$$

$$d = \frac{1}{2} a \cdot 2(d_t) + V_{previous} \cdot d_t$$

intersectCalc()

This function is responsible for detecting head on collisions. It takes in the current location of the ball, the after location of the ball computed in "NextPhase" and the collision plane array to determine if a head on collision will happen before the ball gets to its new location. This function loops through all entries of the collision plane array and uses RayToPlane() to detect head on collision. If multiple planes are found to be intersecting with the ball's path, only the one closest to the ball will be considered. We will have a collision.

CollisionHandle()

After a collision has been detected, this function computes the resulting velocity direction using reflected ray principles. Energy conservation is not being fully modeled in our program. Instead, whenever there's a collision, the magnitude of the velocity will be dampened by a preset ratio (~0.3 - 0.6). The results have proven that this method does indeed mimic very well the Law of conservation of Energy observed in reality. This function also notifies Render of a collision, so that a sound effect can be played, and it also determine if the ball hits the exit point, so that the proper actions can be taken accordingly.

RaytoPlane()

This function takes in the current ball location vector, Ball displacement vector and a CollisionPlaneClass object to determine if the Ball's displacement path will intersect with the given plane (i.e. collision). This is modeled by a simple linear algebra mathematical problem that can help determine if a line created by two distinct points are intersecting a given plane. If a collision is determined to be happening, this function also calculates which point on the given plane does the displacement vector of the ball intersects with.

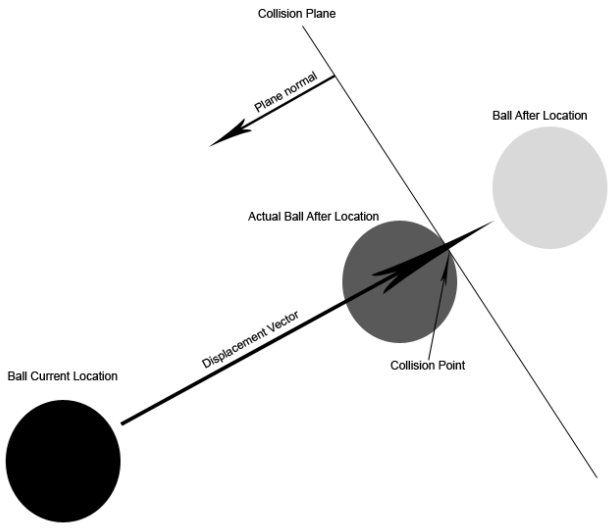

Figure 6.2 Ray to Plane Collision Detection

CollisionPlaneClass()

CollisionPlaneClass is a class object that represents a non-passable finite plane. A plane's normal vector, four vertices, and center point are stored in each of these objects. Its constructor takes in all four vertices of a given plane and uses linear algebra and simple mathematics to calculate its normal and center point. Vertical planes and horizontal planes are treated differently when calculating normal vectors because of the difference of the Z-axis polarity in OpenGL and linear algebra right-hand-rule. This class object is created solely to serve the purpose of collision detection in the Physics engine.

IsOnPlane()

Due to the mathematical model being used in Physics class to determine a ray to plane intersection assumes that a plane is infinite in area, this function is created to help determine if an intersection point calculated in Physics Class is really lying within the bound of a particular collision plane. The method to achieve that is to first connect the calculated collision point with the center of the given plane to form a line. Then using simple linear algebra, we check if there's any intersection between that line and any one of the four bounds of the given plane. If no intersection is found, that means the collision point lies within the bound of the given plane. At such point, the function returns true, otherwise, if an intersection is found, returns false.

6.2 Math Module

6.2.1 Quaternion

The game play of our game requires extensive 3 dimensional control freedom using the mouse. Consistent and intuitive control of the cube is critical for the game. Originally, we translated mouse motion into Euler angle representation and rotated the cube with a simple rotational matrix and modelview matrix multiplication. However, such method

presented a problem generally known as the Gimbal lock, where one of the three rotational axis would be changed when the cube is rotated more than 90 degrees around another rotational axis. Such problem greatly affected the intuitiveness and consistence of the game play. To address this problem, we adopted the quaternion rotation method.

When we're using quaternion to apply rotation to the cube, the rotational direction for each axis is constant regardless of the orientation of the cube. In the Quaternion class, we've defined a quaternion object and a few quaternion specific mathematical operations. With the help of the Vector class, we're able to deliver intuitive cube controls from mouse inputs.

6.2.2 Vector Class

This class defines a 3 dimensional vector with X, Y and Z components. It is greatly used in the Physics engine to describe three-dimensional objects, positions and directions.
This class also contains all of the useful vector mathematical operations such as the cross-product, dot-product and magnitude calculations.

7.0 FRONT END

7.1 Game Display

7.1.1 Menu

The menu option is selected whenever the player presses the key 'm'. The game is then paused and a menu appears on the screen. The options in menu are:

- select background music
- mute
- volume control
- ball style
- restart
- resume
- quit

When the menu option is selected, the mouse is no longer controlling the cube in game play but rather the menu selection. This is done by checking if the menuIsOn variable in frontend is selected. The menu was made click-able with the mouse by using OpenGL's selection render mode. During the creation of each textured menu object, an ID is passed to that object with the use of glLoadName(GLuint ID). This allows us to uniquely identify the rendered item. In the mouse function, the glSelectBuffer(GLsizei size, GLuint* buffer) function is used to create the selectlist buffer stack. Then, whenever a rendered menu object is clicked, it's object ID is passed into the selectlist stack. The integer 'nhits' keeps track of how many clicks have occured. Once 'nhits' are detected, a for loop is used to check which object has been clicked. The object selected's ID during the click 'i' can be found in the selectist at the 4*i+3 position. This ID is then passed to switch cases that determine which follow through with the actions necessary to satisfy the selection chosen. The details of how audio is controlled can be found in the Audio

section of the report. The ball style is changed by changing the material characteristics of the ball object. Resume menu selection is satisfied by changing the menuIsOn variable to not true. Restart menu option is implemented to go back to the initial game sequence. Quit menu selection exits the game program.

7.1.2 Rendering

In our project, the maze has multiple layers which consist of many blocks. There are four choices for the blocks, the solid block, the empty block, the hole, and the trap. The design of the maze is first stored in a text file, and imported into the game each time when the game is being executed. To translate the block design into three-dimensional maze, we first obtain the pre-set levels of the height, width, and length from the Game Engine, and use for loops to stack different types of blocks according to the maze text file. It is noted that we draw our blocks in a display list to improve our game speed. By putting the code between display list start and end, the code will only be compiled once and can be reused every time we call the list. If display list was not used, we would have needed to configure the pipeline, push the data, and obtain the binary value for draw each block.

For the ball to be visible to the player while moving the ball, the layers that are one level higher than the ball position or higher are set to be transparent. The transparency is set by using alpha bending, which is setting the alpha value to be 0.05. Also, because our cube can be rotated in any direction in any degree, we use the normal vector of the top plane to detect whether the cube is up-side-down. When the cube's top plane is facing downward, the layers that were originally one level lower than the ball position or lower are set to be transparent.

7.1.3 Texture

The LoadTextureRaw(const char * filename, int wrap) from http://www.nullterminator.net/gltexture.html was used. This function takes in a .raw file that is 256x256 and loads it into a buffer, and sets the attributes of the texture. By binding a texture, we can choose which texture is used. When the game is ended, the textures are deleted.

In this game, textures are used for the background, menu screen, and cube. The background is loaded and not changed. For the menu screen, each button has a unique texture mapped to it. Since the cube is rendered block by block, it was simpler to texture each block individually. To keep the appearance of the cube's surface continuous, the texture of the blocks were created so that the opposite edges of the texture would match (Figure 7.1) when tiled.

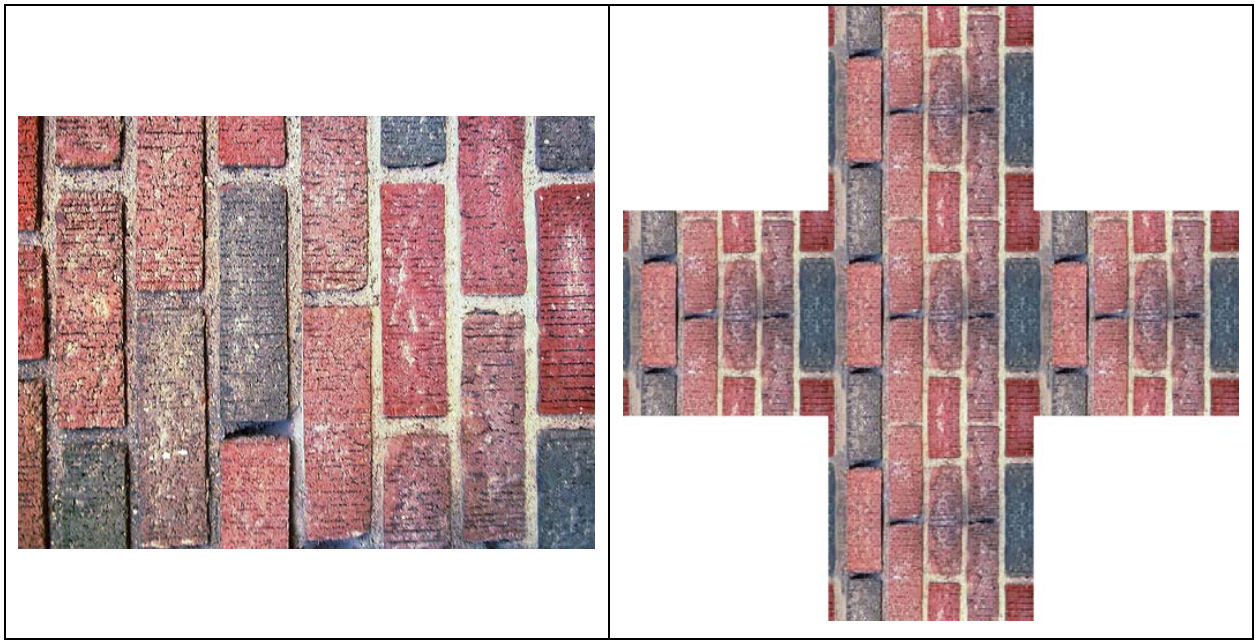

Figure 7.1 Textures of the Blocks vs. Seamlessly Tiled Version of the Texture File

The textures used in the game are from images online that have been edited and converted to .raw format using Photoshop.

7.2 Mouse Control

The maze's movement is fully under the control of the mouse. The maze can be rotated in 3 degree of freedom. Combined with mouse clicks, the 2D motion of the mouse can be translated into 3-dimensional motion of the cube.

Left Mouse Button held down:
Forward and Backward motion of mouse is translated into pitching motion of the cube.
Left and right motion of the mouse is translated into rolling motion of the cube.

Right Mouse Button held down:
Forward and Backward motion of the mouse is also translated into pitching motion, while left and right motion of mouse is now translated into yawing motion of the cube.

glutMouseFunc, glutMotionFunc and glutPassiveMotionFunc are used to achieve the mouse control scheme. glutMouseFunc is used to determine which of the two mouse button is being pressed. glutMotionFunc is used to track the mouse's position while a mouse button is pressed, and the displacement of the mouse cursor is translated into rotation degree. glutPassiveMotionFunc tracks the position of the mouse while NO BUTTON is pressed. The purpose of this is so that while not attempting to rotate the cube (i.e. no mouse button pressed), the user can still freely move the cursor around the screen.

7.3 Audio

The audio used in the game was loaded using the OpenAL API. OpenAL objects consist of a Listener, a Source, and a Buffer. There can be multiple buffers and sources. The buffers contain the audio data (.wav format) and are attached to a source. The source is

the object that actually emits the sound. The listener represents the position where the sounds are heard.

As openAL has the capabilities to create multiple sources that hold multiple buffers, the listener is able to hear more than one audio clip at one time. The background music and sound effects files were all loaded as buffers attached to different sources so that we could control the behaviour of each separately. This allows us to create a game that has many different audio situations; for example, the background music continues playing when an event driven sound effect, such as the ball colliding with the wall, is activated. For this game, 8 sources were used for the following sounds: rolling, collision, game over, game win, menu click, menu open, background music normal tempo, and background music fast tempo.

Some of OpenAL's features are that it allows the programmer to set the location, velocity and orientation of the listener as well as set the gain, looping option, pitch, location and velocity of the source. The listener's location, velocity and orientations and the source's pitch, location and velocity are all initialized when the game is started. Depending on the current situation, different sources were activated to play. During normal game play, the background music normal tempo source was set to loop, and rolling and collision sources were triggered by the balls current action. When the timer reaches only 10 seconds left, the background music fast tempo source is activated instead of the normal tempo. When game end occurs, either the game over source or game win source is activated. For the duration of the entire game, menu can be opened. When it's open, menu open source is played once, then for each time a menu option is clicked, menu click source is played.

Inside menu, extra features were added such as the option to change the volume or music being played. The volume of the background music could be changed by changing that particular sources gain value lower than it's current value. Also, different background

music selections were made possible by allowing the buffer associated with the background music to be changed.

In our game, OpenAL worked well in playing audio files for music and background music. However, a few issues arised during implementation. One issue that came up was that only .wav files could be used and as such only files of relatively low quality could be used. If .wav files of high quality or long length were used, the size of the .wav would be high. This was acceptible as there were only a few lengthy files. The other issue was that the installation of OpenAL was not very many well documented on the sources that were found online. In the future of the game however, this is not an issue as eventually we were able to install OpenAL on Windows OS and it is already installed on Mac OS.

8.0 RTU

The Real-time Tweaking Unit (RTU) is a function that provides developers a more convenient way to change the values for all adjustable parameters, such as camera position, light parameters, ball positions, gravity, and etc. The developers can key in the variable names they would like to monitor into the code, and use arrow keys to adjust the values while the game is being executed. The real-time response from the RTU enables the developers to find the best solution for the game setting without repeating the tedious trial-and-error work, such as going back to the code, increasing the value by 1, pressing save, re-compiling and re-running the program to monitor the effect. It is a very helpful tool for game development at the finishing stage. After using the RTU for tweaking, the developers will remove it from the final design for demo purpose. Figure 8.1 shows the look of the RTU.

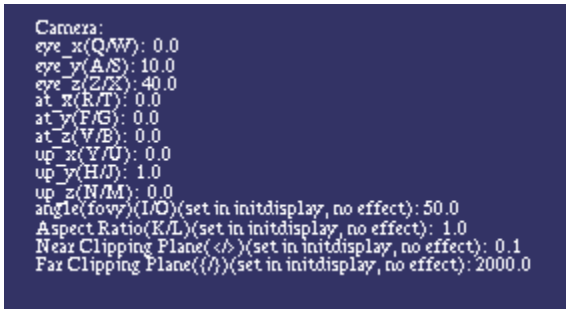

Figure 8.1 Sample of the Real-time Tweaking Unit Display

In our game, we implement the RTU by using function renderBitmapString to show the variable names and the pointers that point to the desired variables to be changed. Function glViewport is used to separate the RTU from the real game display. Special keys, the arrow keys, were set to point to different variables to adjust them. For example, pressing **DOWN** would change the variable to be changed from “eye_x” to “eye_y” in Figure 8.1, and pressing **LEFT** would decrease the value of “eye_y” by 1 while pressing “right” would increase the value by 1.

9.0 FUTURE IMPROVEMENT

The following components were discussed and implemented to add to overall game processing efficiency. However, due to unforeseen circumstances, it is deprecated. These components are included within this report for completeness.

9.1 Multi-threading

To improve performance of the game, each component of the game is executed on different threads. Posix Pthreads API was used because of its simple interface and its portability. Using threading means that each of the components of the game has to internally take care of multithreading and resource sharing. The components also has to be designed around the concept of multithreading. Each of the components are already quite difficult to design and implement without the extra burden of multithreading. A solution to this is to encapsulate the logic of multithreading within one component, and all other systems only needs to implement the interfaces to the multithreading component to be able to run on different threads.

ThreadClass contains all the logic of threading such as creating threads, suspending threads, resuming threads as well as terminating threads. ActiveClass inherits from ThreadClass to act as the interface for other components. ActiveClass also provides an abstract function that each component must override to execute as a thread.

9.1.1 ThreadClass

ThreadClass was designed to act as an all in one package that allows any class to run as multithreaded application while providing the ability to suspend, resume and terminate it. Unfortunately pthread does not provide a means of suspending and resume threads other then using conditional wait variables. To solve this, a special function has to be written so

that it runs on its own thread, and it will continuously call user's intended threading function. This design allows external threads to suspend the currently executing thread by the use of conditional wait variables. In our ThreadClass, this special function is called ThreadRunner. As the name suggests, its purpose is to run the user's intended thread function.

ThreadRunner contains an infinite loop that continuously runs user's thread function until user chooses to quit via return value or some other means. Within the loop, the function will check whether or not to suspend the thread.

9.1.2 ActiveClass

ActiveClass acts as an interface between other components to ThreadClass. ActiveClass contains an abstract function threadFunction that each of the inherited component must implement. The inherited components will put all of their logic within this function. The function pointer of threadFunction is then passed into ThreadClass to execute as a thread. Because the function pointer of threadFunction is passed into ThreadClass through the constructor, a wrapper around threadFunction has to be created because it is pure virtual. The wrapper's purpose is to act as a non pure virtual function so that the pointer of the function can be passed into base class. Within the wrapper, it just calls the original threadFunction.

9.2 Multi-threading Communication

In a multithreading environment, a particular thread may not be aware of other sister threads that are created by the parent thread. In order for the sibling threads to know of each other and to communicate with each other, a dispatcher should be used. The dispatcher's purpose is to deliver messages that it receives from other systems to the destination systems.

9.2.1 Design

As stated above, sibling threads are not aware of each other when it is running. To solve this, each of the threads only needs to know of the dispatcher, and let the dispatcher take care of the actual message delivery. Some messages may have multiple recipients as well as different priorities. The approach we chose to take to solve this problem is to have one single dispatcher datapool that receives messages from all external systems. The dispatcher would sort and prioritize the messages for each external system concurrently, and then finally send the message out.

9.2.2 Dispatcher

The dispatcher itself is a thread running concurrently with other systems in the game. However dispatcher also contains multiple threads within it to achieve maximum efficiency. There is one single thread called Parent, and multiple Child threads for each of the external systems.

The dispatcher thread is responsible for creating and killing off the parent and child threads. The parent thread is responsible for testing if there are any messages sent from external systems to the dispatcher. If there are messages, it would copy the message to the appropriate child threads for sorting and prioritizing.

The children threads would sort the messages according to priority. However there are potential cases where high priority messages would keep on interrupting previous messages resulting in those messages with less priority not getting executed. To combat this issue, we also added a timestamp to each message.

The messages are first sorted by priority with higher priority at the front of the queue. Within messages with same priority, the messages that are older are moved closer to the front of the queue. This algorithm ensures higher priority messages will be executed first and message order will not keep on being interrupted by higher priority messages.

9.2.3 Messaging System

As mentioned above, Datapools are used as container to store messages in a queue. Datapools has the ability to provide locking mechanisms for concurrent access between multiple threads to shared resources. The datapools are declared “global” to the threads accessing it because threads share same memory space.

Datapools contain a queue of messages. From the description before, each of the messages has to contain a priority as well as a timestamp for sorting purposes. Below is the schema of the messages.

```
Class
{
  Sender,
  List of Recipeints;
  Priority,
  TimeStamp;
  Message
}
```

The message class is declared as a template class because many different types of messages may need to be passed from one system to the other.

10.0 CONCLUSIONS

Overall, the game achieved what our group envisioned to do from the beginning. The game included all the playable features such as game winning and losing scenarios as well as mouse control of the maze and real world physical simulation. The success of the project is primarily a result of our ability to follow through with an iterative software development cycle. Another major factor in our success is the result of our awesome teamwork. If more time were given, a lot more visual effects could be added to improve the overall effectiveness of the game. The improvements mentioned in previous section could also be fully realized.

11.0 REFERENCES

[1] Hodges, Brian. "OpenGL Texture Tutorial." [Online]
http://www.nullterminator.net/gltexture.html, August 25, 2001.

[2] Olson, Curtis. "Flight Gear Installation Guide." [Online]
http://www.flightgear.org/Docs/Tutorials/fg_cygwin/fgfs_cygwin.htm#_Toc117306445,
October 2005.

[3] Vine, Norman. "Guide to cyg_openAL."
http://www.vso.cape.com/~nhv/files/cygwin/cyg_openAL.tgz, June 14, 2005.

[4] Winder, Lee. "A Guide To Starting With OpenAL."
http://www.gamedev.net/reference/articles/article2008.asp, October 23, 2003.

[5] "OpenAL Tutorial." http://www.edenwaith.com/products/pige/tutorials/openal.php,
November 10, 2005.

[6] "OpenAL." http://www.openal.org/, December 12, 2006.
Sherrod, Allen. "Ultimate Game Programming."
http://www.ultimategameprogramming.com/zips/Gl_Selection.ZIP, October 2004.

[7] Void, Sobiet. “Quaternion Powers.”
http://www.gamedev.net/reference/articles/article1095.asp,
February, 2003

[8] “The Matrix and Quaternion FAQ, Version 1.4”
http://www.flipcode.com/documents/matrfaq.html#Q54, December, 1998

[9] “OpenGL:Tutorials:Using Quaternions to Represent Rotation”
http://gpwiki.org/index.php/OpenGL:Tutorials:Using_Quaternions_to_represent_rotation

[10] Bobick, Nick. “Rotating Objects Using Quaternions”
http://www.gamasutra.com/features/19980703/quaternions_01.htm, July, 1998

[11] “Little Quaternion Example”
http://www.idevgames.com/forum/showthread.php?p=134470, September 25, 2007

[12] Funkhouser, Thomas. "Ray Casting – Princeton University CS 426 Lecture 17”
http://www.cs.princeton.edu/courses/archive/fall00/cs426/lectures/raycast/sld017.htm, 2000

[13] “Sphere/Plane Intersection: Collision Detection Tutorial”
http://www.gamespp.com/algorithms/collisionDetectionTutorial01.html

[14] Owen, Scott. “Ray-Plane Intersection”
http://www.siggraph.org/education/materials/HyperGraph/raytrace/rayplane_intersection.htm, June 2, 1999

[15] “Collision Detection”
http://www.edenwaith.com/products/pige/tutorials/collision.php, October, 2003

[16] “Posix Threads Programming”
https://computing.llnl.gov/tutorials/pthreads/, April 2, 2008