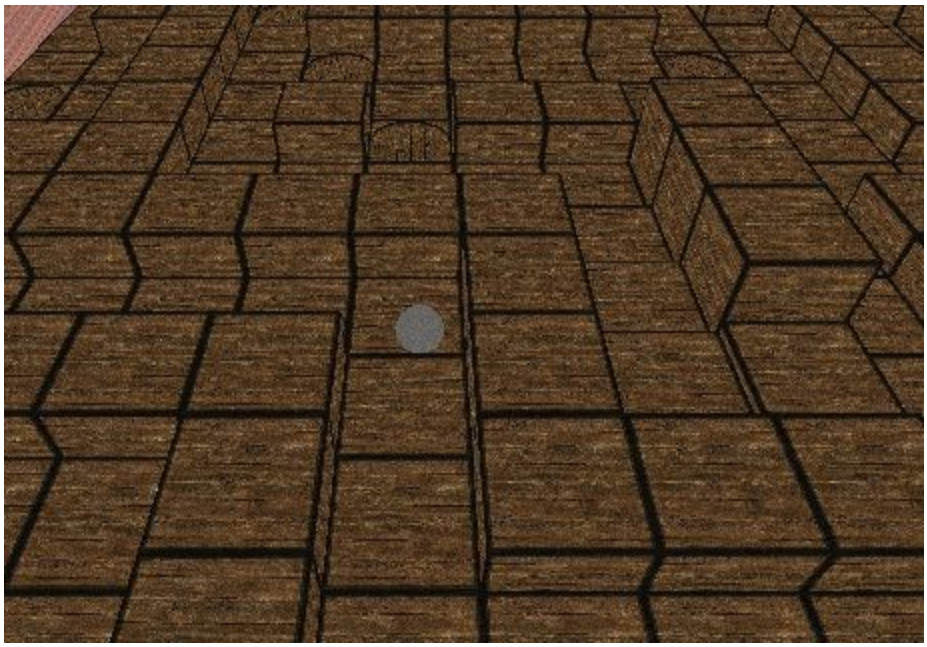
Technical Manual
PA 11 - Labyrinth



Group Members: Zeeshan Sajid, Vance Piscitelli

To compile/run the program from the PA11 bin folder:

```
make -C ../build
```

Then

```
./Lab
```

Extra Credit
For extra credit, we have sound effects,
a top 10 scores history,
multiple balls,
multiple levels,
day/night mode for difficulty,
walls that can change height based on distance from ball,
the ability to load custom levels from text files.

Required Libraries

In order to compile and run this program, some additional programs/libraries must be downloaded and installed. These programs/libraries are:
Magick++
Assimp
OpenGl
Bullet
Irrklang Audio Library

Setting Up Magick++

In your .cpp file, include:
```
#include <Magick++.h>
```

And initialize in main:
```
Magick :: InitializeMagick(*argv);
```

In the makefile, add:
```
LDFLAGS = `Magick++-config --cppflags --cxxflags --ldflags --libs`
```

and then add LDFlAGS to the compilation line like this :
```
$(CC) $(CXXFLAGS) ../src/shader.cpp ../src/main.cpp -o ../bin/Lab $(LDFLAGS)
```

Also, don't forget to add -lMagick++ to the LIBS to use.

Install these libraries with sudo apt-get install:
(These are the libraries installed on the ECC computers)
libgraphicsmagick3
libgraphicsmagick1-dev
libgraphics-magick-perl
libgraphicsmagick++1-dev
libmagickcore5-extra
libmagickwand5

imagemagick
libgraphicsmagick++3
graphicsmagick-libmagick-dev-compat
libmagickcore5
imagemagick-common

Or you can follow the installation instructions from:
http://www.imagemagick.org
To customize the installation.

Setting Up Assimp

To install Assimp, type:
sudo apt-get install libassimp-dev

Then modify your makefile to include `-lassimp` in the `LIBS` section:
```
LIBS= -lglut -lGLEW -lGL -lGLU -lassimp -lMagick++
```
Note: -lMagick++ is also added to use Magick++

In your main.cpp, you also need to add a few lines:
```
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>
#include <assimp/color4.h>
```

Setting Up OpenGL

Most operating systems have some version of OpenGL on them but to run our program, a few additional programs need to be installed. The first two are GLUT and GLEW which can be easily installed with:
```
sudo apt-get install freeglut3-dev freeglut3 libglew1.6-dev
```

If you can type:
```
glxgears
```
into the terminal and some rotating gears appear, then you should have installed everything for OpenGL correctly.

Figure 1: If you can see these gears, then OpenGl should be installed correctly.

You will also need GLM which can be installed by typing:

```
sudo apt-get install libglm-dev
```

Setting Up Bullet

Bullet is a physics library that can be used in C++ but requires a few steps before it can be used. It can be installed in the command line with:

```
sudo apt-get install libbullet-dev
```

The makefire must be modified to include

```
LIBS = -I/usr/include/bullet - lBulletDynamics -lBulletSoftBody -
lBulletCollision –lLinearMath
```

The Header file needs:

```
#include <btBulletDynamicsCommon.h>
```

Some resource for Bullet are:
http://bulletphysics.org/mediawiki-1.5.8/index.php/Collision_Filtering
http://bulletphysics.org/mediawiki-1.5.8/index. php/Stepping_the_World
http://bulletphysics.org/mediawiki-1.5.8/index.php/Hello_World
http://bulletphysics.org/mediawiki-1.5.8/index.php/Tutorial_Articles

Setting Up Irrklang audio library

Irrklang is a cross platform sound library for C++, C# and all .NET languages, essentially, it is free for non-commercial, while there is a pro version that comes with more features than the free. Since this is not offered on GitHub it is much harder to attain it than by just doing sudo apt-get install lib.

Instructions for usage:

1. To download, go to http://www.ambiera.com/irrklang/, and click download and download the zip file Irrklang has available, and it will come in as a zip file
2. Extract the zip file and run the hello world or any program to see if it works and is not corrupted
3. From there, extract the folder specifically catered to your OS. Since this class is using Linux, grab linux-gcc-64, along with the include folder which will host most of the header files.
4. From there go into your makefile and add this line or equivalent file location:
   OPTS = -I"../src/include" ../src/linux-gcc-64/libIrrKlang.so -pthread
5. From there, add it to compiler flag just like you would with LIBS, something along the lines of this:

```
$(CC) $(CXXFLAGS) ../src/shader.cpp ../src/main.cpp -o
../bin/Lab $(LDFLAGS) $(LIBS) $(OPTS)
```

6. Within your .cpp file or wherever you are running it, #include<irrKlang.h> and if you want to use the namespace you can just by doing using namespace irrKlang
7. From there, the program should be compilable and you can start using irrKlang for a lot of different sound libraries.

Compiling the Program

To compile the program, from the build folder: `make`
Or from the PA11 folder type: `make -C ./build`
Or from the bin folder type: `make -C ../build`

Executing the Program

To run the program, from the bin folder, type: `./Lab`
To change player's name, type the player's name as the first command line argument e.g. `./Lab Vance`

Program Implementation Details

Level Design

Instead of having a board object with holes and another object with the walls for the maze on top of the board, we decided to add many more objects, about 225 of them. We went with a 15x15 array of cubes that could each be positioned at a desired height which was then inclosed by a frame with knobs to still look like the traditional labyrinth game. By having each cube individually set, we did not have to go and design a new level in blender each time we wanted a new level or to just make a simple tweak. Instead, our levels our read in from a text file.

Loading from a Text File

To easily create levels, our program reads in data from a text file specifying each cube's type (regular cube, cube with a hole, start cube, end cube) and height. The first level is read in during the initialize function but as the player completes levels, it reads in the next level from a different text file and repositions all the cubes using a sleek animation.

```
s1.0 s1.0 s1.0 s2.0 s1.0 s1.0 s1.0 s1.0 s2.0 s3.0 h1.0 s1.0 s1.0 s1.0 e1.0
s1.0 h1.0 s1.0 s1.0 s1.0 s1.0 h1.0 s1.0 s1.2 h1.2 s1.0 h1.0 s2.0 h1.0 s2.0
s1.0 s2.0 h1.0 h1.0 s2.0 h1.0 s2.0 s2.4 s1.4 s3.0 h1.0 s1.0 s2.0 s1.0 s1.0
s1.0 s2.0 h1.0 s1.0 s1.0 s1.0 s1.0 h1.0 s1.6 s3.0 s1.0 h1.0 s1.0 h1.0 s1.0
s1.0 s1.0 s1.0 s1.0 s2.0 s1.0 s2.0 s2.0 s1.8 s3.0 h1.0 s1.0 s2.0 s1.0 s1.0
s0.0 s0.0 s0.0 s0.0 s1.8 s0.8 s1.8 s2.0 s3.0 s1.0 s1.0 h1.0 s1.0 h1.0 s1.0
h0.0 s1.0 s1.0 s0.0 s1.6 s0.6 s1.6 s2.0 h2.0 s1.0 h1.0 s1.0 h1.0 s2.0 s2.0
s0.0 s0.0 s0.0 h0.0 s1.4 s0.4 s1.4 s2.0 s3.0 s1.0 h1.0 h1.0 s1.0 h1.0 s1.0
s0.0 s0.0 s1.0 s1.0 h0.0 s0.2 s1.2 s2.0 h2.0 s1.0 s2.0 s2.0 s2.0 s2.0 s1.0
s0.0 s1.0 s0.0 s0.0 s0.0 s0.0 s1.0 s1.8 s2.8 s1.0 s1.0 s1.0 s1.0 s2.0 s1.0
s0.0 s1.0 s0.0 s1.0 h0.0 s1.0 s0.0 s1.6 h1.6 h1.0 h1.0 h1.0 s1.0 s2.0 s1.0
s0.0 s1.0 s0.0 s0.0 s0.0 s0.0 s0.0 s1.4 s2.4 s1.0 s1.0 s1.0 h1.0 h1.0 s1.0
s0.0 s0.0 s1.0 s1.0 s0.0 s1.0 s0.0 s1.2 s1.0 s1.0 h1.0 s1.0 h1.0 s1.0 s1.0
s0.0 s1.0 s0.0 s0.0 s0.0 h0.0 s1.0 s0.0 s2.0 s1.0 s1.0 s1.0 s1.0 s1.0 h1.0
b0.0 s0.0 s0.0 h0.0 s0.0 s0.0 s0.0 s0.0 s2.0 s2.0 s2.0 s2.0 s2.0 s2.0 s2.0
```

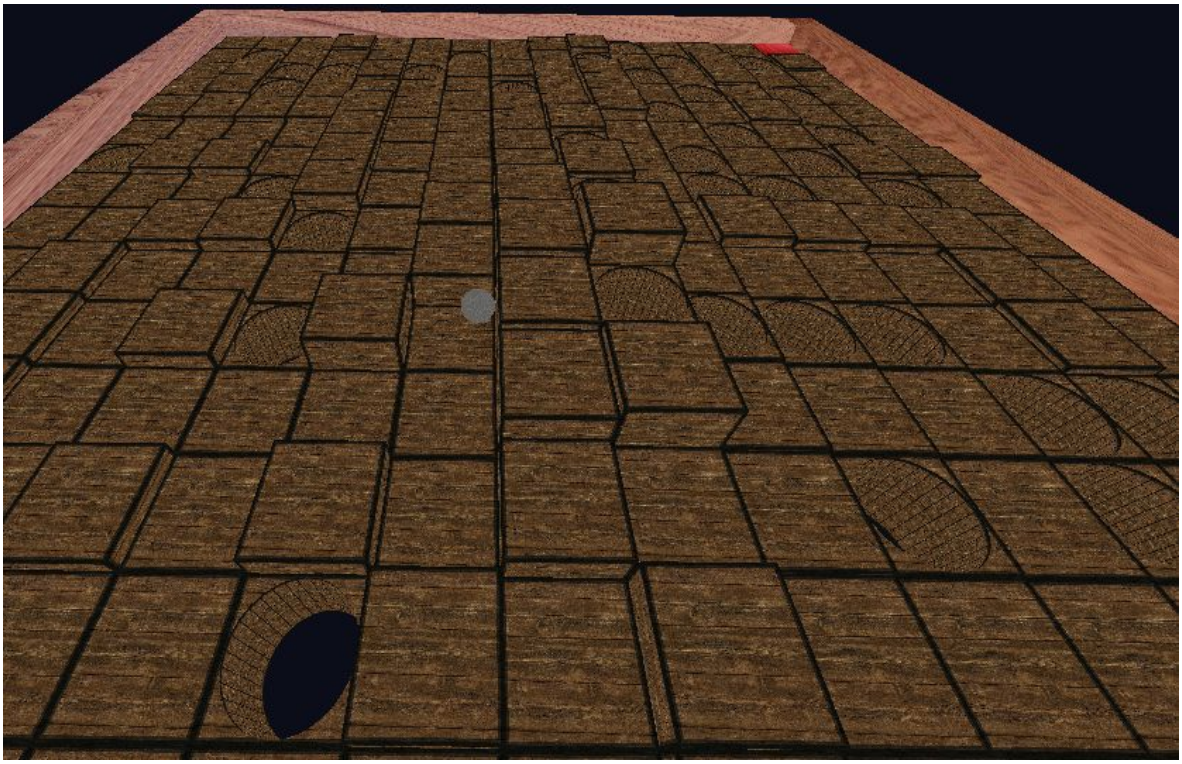
A sample level text file. Each letter represents the type of the cube and the number represents the height the cube should be at.

Moving Walls

This sleek animation is made possible by moving all the cubes between their positions in the current level to their new positions in the next level on step at a time. Each time the update function is called, the board takes one step as it transitions between the levels. Once no cube gets moved during a step, the transition is finished and the game continues. As the cubes change height, the cubes rigid body is updated to be in the proper position. Additionally, if the cube's type is changed between two levels, such as from a regular cube to a cube with a hole in it, the rigid body changes to accurately reflect the new shape. However, we did not only use the moving walls in level transitions.

Rising Walls

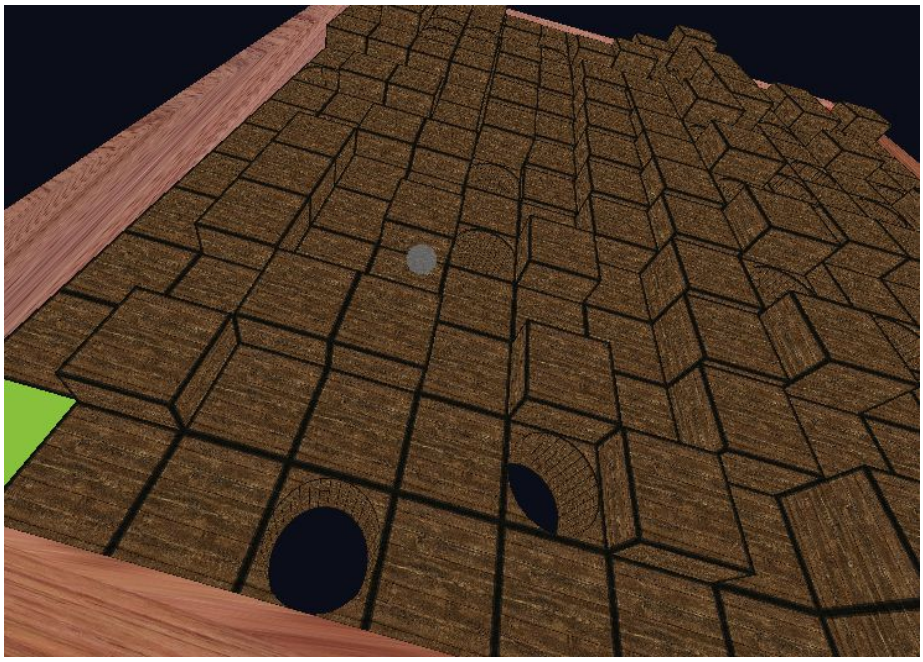
If the player hits ‘2’ while playing the game, the game toggles to use moving cubes that change height based on distance from the ball. As the ball is closer to a given cube, the cube gets closer to its correct height. Once the cube is within 3 units of the ball, it is at its proper height. Since the cubes that are far away from the ball are close to 0, it makes it harder for the player to figure out where to go and what the rest of the maze looks like.



The cubes get closer to their correct height the closer they are to the ball.

Tilting the Board

In the original labyrinth game, the ball is controlled by tilting the board. Since we have about 225 cubes on the screen, it would be a bit difficult to correctly tilt each cube such that it remains in the correct position. To avoid this problem, we simply tilt the camera. To simulate the left and right tilting of the board, we tilt the camera left and right respectively. To simulate tilting the board forward and backward, we adjust what the camera is looking at by either looking above or below. At the same time as we tilt the camera, we adjust gravity to make the ball appear to roll in the direction of the tilt. While the gravity change instantly changes based on current input from the arrow keys, the camera slowly changes from its current tilt to the proper tilt a small degree each time the update function is called, similar to the moving walls. By not rushing between tilting the board in different directions, the game controls feel smoother and are more pleasant to look at.



An example of the board “tilting” to the right. The camera is actually doing the real tilting.

Challenges We Faced

Number of Cubes

The biggest challenge in completing this project was the large amount of objects that we had on the screen at the same time. In Air Hockey, we only had 5 objects on the screen at the same time. In Labyrinth however, we have a little over 225 objects on the screen. Therefore, when designing our program, we had to make it efficient or else the slightest delay could propagate into a large amount of lag for the player as each cube struggles to operate correctly. Solutions to this problem included using only two models for the cubes, the basic cube and a cube with a hole, and not calling unnecessary operations.

Lighting

While we had basic lighting working for our labyrinth early on, it did not look as good as it did for PA10. This was largely due to the increased number of cubes. Therefore, once we got towards the deadline, we switched back to focusing on the lighting. However, as we adjusted how the lighting would work, we would frequently run into issues where the code would run on one computer perfectly but would simply crash on another. After a few days of tinkering, we managed to get the lighting to work by adjusting the lighting calculations in the shaders.

Future Improvements

Given more time, we would likely try to improve the performance. While testing out our idea for having the a board made up of a grid of cubes, the program could handle larger numbers of cubes, easily over a thousand, but once the lighting was added, the framerate started suffering. During one test, we tried a 1,000 x 1,000 board which managed to run, but under 1fps. However, we could actually improve this by deactivating cubes that are off the screen and not worrying about their physics. Additionally, with larger boards, we would need better level editing besides simply modifying a text file.