8 BALL POOL GAME

USING C/C++ AND

OPENGL

INDEX

ABSTRACT

The project, 8 ball pool game, was developed using OpenGL API and its equivalents in C++. The game involves 16 balls and a pool table.

The balls can be hit by mouse by clicking the mouse and releasing the mouse at the point where you want to hit the ball.

The motion of the balls is controlled by friction and collision (Collision between wall and a ball & Collision between 2 balls).

The pocket test is done to make sure that when the ball reaches a pocket, it is not on the pool table anymore.

INTRODUCTION TO OPENGL AND OTHER UTILITIES

Open Graphics Library (**OpenGL**) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering.

Along with OpenGl API, following libraries and APIs have been made use of :

- FreeGLUT : (Free OpenGL Utility Kit), allows the user to create and manage windows containing OpenGL contexts on a wide range of platforms and also read the mouse, keyboard and joystick functions.
- GLEW : (GL Extension Manager), adds support for the newest OpenGL versions. With its help, programmer can access and play with OpenGL's newest functions.
- GLM : (OpenGL Mathematics), is a header only C++ mathematics library for graphics software based on OpenGL Shading Language (GLSL) specifications.
- SOIL : (Simple OpenGL Image Library), is a C library primarily used for uploading textures into OpenGL. With its help, a programmer can provide attractive textures and colors to the game objects.

FreeGLUT

FreeGLUT is an open source alternative to the OpenGL Utility Toolkit (GLUT) library. GLUT (and hence FreeGLUT) allows the user to create and manage windows containing OpenGL contexts on a wide range of platforms and also read the mouse, keyboard and joystick functions. FreeGLUT is intended to be a full replacement for GLUT, and has only a few differences.

Since GLUT has gone into stagnation, FreeGLUT is in development to improve the toolkit.

GLEW

The **OpenGL Extension Wrangler Library** (**GLEW**) is a cross-platform C/C++ library that helps in querying and loading OpenGL extensions. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. All OpenGL extensions are exposed in a single header file, which is machine-generated from the official extension list.

GLM

OpenGL Mathematics (GLM) is a header only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications.

GLM provides classes and functions designed and implemented with the same naming conventions and functionalities than GLSL so that anyone who knows GLSL, can use GLM as well in C++.

Methods Used :

❖ Vector and Matrix Constructors

- If there is a single scalar parameter to a vector constructor, it is used to initialize all components of the constructed vector to that scalar's value:

```
glm::vec4 Position = glm::vec4( glm::vec3( 0.0 ), 1.0 );
```

- If there is a single scalar parameter to a matrix constructor, it is used to initialize all the components on the matrix's diagonal, with the remaining components initialized to *0.0f*

```
glm::mat4 Model = glm::mat4( 1.0 );
```

❖ Matrix transformation

- Matrix transformation is an extension of GLM. Example from GLM manual:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

int foo()
{
    glm::vec4 Position = glm::vec4( glm::vec3( 0.0f ), 1.0f );
    glm::mat4 Model = glm::translate( glm::mat4( 1.0f ), glm::vec3(1.0f
) );
    glm::vec4 Transformed = Model * Position;
    return 0;
}
```

❖ Identity Matrix

- `glm::mat4` constructor that takes only a single value constructs a *diagonal matrix*:

```
glm::mat4 m4( 1.0f );
```

The matrix has all zeros except for *1.0f* set along the diagonal from the upper-left to the lower-right.

- The default constructor `glm::mat4()` creates diagonal matrix with *1.0f* diagonal, that is, the identity matrix:

```
glm::mat4 m4;
```

❖ glm::translate

- When using glm::translate(X, vec3), you are multiplying

```
X * glm::translate( Identity, vec3 )
```

- This means translate first, then X

❖ glm::rotate

- When using `glm::rotate( X, vec3 )`, you are multiplying

```
X * glm::rotate( Identity, vec3 )
```

- This means rotate first, then X

❖ Matrix multiplication is not commutative

- So the conventional Model-View-Projection should be multiplied in reverse:

```
glm::mat4 MVP = Projection * View * Model;
```

- This means that Model transformation happens first, then View, and Projection is last.

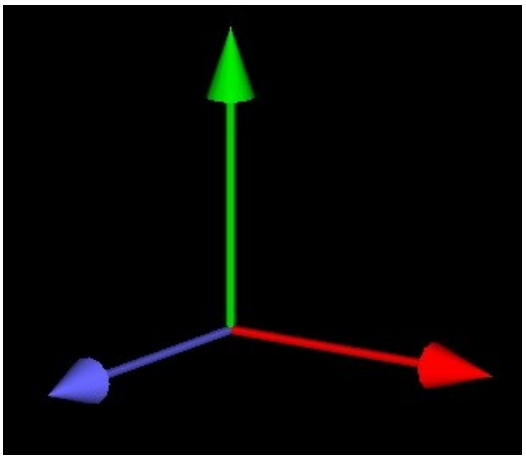
SOIL

The **Simple OpenGL Image Library** or **SOIL** is a public domain image loading library, written in C. The library allows users to load images directly to OpenGL textures.

The Simple OpenGL Image Library can load bitmap, jpeg, png, tga, and dds files.

Vertex

The vertex is the most basic building block in a computer generated space. It's the equivalent of a point in geometry. It lacks size and it occupies a certain spot in 3D space, as described by 3 coordinates.



The red is X-axis, green is Y-axis and blue is Z-axis

Pixel

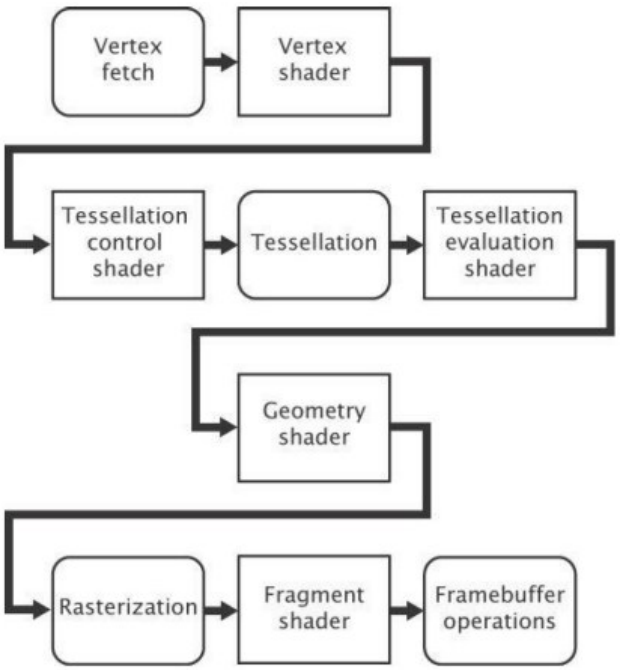
The pixel is more of a physical entity. The screen is composed of a large number of tiny cells, each with its own color. The number of the pixels can be determined by knowing the resolution. So, a screen which can accommodate a maximum resolution of 1080p (or 1920×1080) has about 2,073,600 physical pixels.

It's the smallest component of an image and it has only one color. Of course, depending on the number of bits the color can be described by, the image can have different levels of quality (for example, 1 bpp – bit per pixel – means a black & white image, 8 bpp means greyscale all the way to 24 bpp, with 8 bits per individual color – red, green and blue – leading to a 2^(8*3) = 2^24 = 16,777,216 possible color permutations). This sounds like a lot but we'll see that there are reasons to use a higher bit depth — namely quality to better match what the eye+brain really can see. i.e. 256 shades of primary colors aren't enough.

In a nutshell, the vertex is a 3D point consisting of 3 dimensions: x,y, and z and the pixel is 2D point consisting of 3 (or 4) color channels: red,green,blue, and optionally alpha (we'll deal with this sucker when we reach transparency).

Shaders

Shaders are programs too, the difference being that they run on the GPU (Graphics Processing Unit). This frees up the CPU from the task of drawing hundreds, maybe thousands of polygons each frame, thus giving it the time to listen for pressed keys, running AI scripts, doing physics calculations (although with CUDA you can move all that stuff on the GPU too). Shaders come in all shapes and sizes, so to speak. There are a few types, illustrated in the following image:



Rendering pipeline from OpenGL Superbible 6th Ed.

Vertex and fragment shaders are the most important ones because without them we can't draw anything. Shaders allow us to customize how and when the pixels get colored.

Between shaders, you can pass information through in/out variables. This operation has to follow certain rules for it to be successful:

- The name and type of the input variable from the destination shader must be the same as its corresponding out variable from the sender shader
- An out variable can be sent only to the next existing shader in the pipeline (for ex., you can't send a variable directly from the vertex to the fragment shader if you have a tessellation shader between them)

From the CPU, shaders receive information through buffers, each value corresponding to one or multiple vertices (accessed only by the vertex bufer) and through uniform variables which, can be accessed by any shader directly.

Shaders are written in the GLSL, which is similar to C++.

Before they can be used, we need to read the files where the shaders are kept, compile them and integrate them into your program. To do this, we need to create a C++ class which will deal with these operations. The first class is **Shader_Utils**.

It's necessary to include the **GLEW** and **FreeGLUT** libraries, because we need the variable types **GLuint** and **GLchar**, and important functions like **glLinkProgram** and **glAttachShader**.
The **GLuint** variables will hold the shader and program handles which will be used for integration. OpenGL confusingly uses the name "program" – a program is a container that holds the shaders added to it (vertex, fragment, tessellation, geometry)

ReadShader reads and returns the contents of a file. **CreateShader** method creates and compiles a shader (vertex or fragment). **CreateProgram** method uses **ReadShader** to extract the shader contents and to create both shaders and load them into the program which is returned to be used in rendering loop.

CreateShader encapsulates all relevant operations required to create a shader. The following functions are called:

- **glCreateShader**(shader_type) – it creates an empty shader object (handle) of the wanted type
- **glShaderSource**(shader, count, shader_code, length) – it loads the shader object with the code; count is usually set to 1, because you normally have one character array, shader_code is the array of code and length is normally set to NULL (thus, it will read code from the array until it reaches NULL) , however here I set the actual length.
- **glCompileShader**(shader) – compiles the code
- **glGetShaderiv**(shader, GLenum ,GLint) – Check for errors and output them to the console

You do this operation chain for any shaders you have, to create their objects.

Then in **CreateProgram**, you create the program through **glCreateProgram()** and attach the shaders to it with **glAttatchShader()**. Finally, you finish it off by linking the program with **glLinkProgram()**. We also need to check if the shader was linked properly using **glGetProgramiv()** and output possible errors to the console. Checking and catching shader and linking program errors is extremely important, helping you saving time and many headaches.

To use this program (these shaders) the following method needs to be added to the render loop when we begin to draw.

glUseProgram(program);

The first and most important of all shaders, they have the critical job of processing every vertex and returning its position on-screen through a series of transformations. Along with the vertices, the vertex shader also receives arrays of data associated with each vertex (offset, normal, texture coordinate, etc.) in buffers. The only predefined variable we use for vertex shader is **gl_Position**, in which the final on-screen position of the current vertex is saved (after the chain of transformations).

HARDWARE & SOFTWARE REQUIRED TO CREATE, BUILD, EXECUTE AND TEST OPENGL AND GLUT UTILITIES

Minimum Hardware Requirements for Newest features of OPENGL

- Any CPU (Intel i5/ i7/ Xeon recommended)
- 1.5 GHz or faster processor
- Any GPU that is compatible with OpenGL 3.2. (Preferably NVIDIA GeForce Graphics Card 600 series)
- 4GB RAM or above
- 10 GB HDD Free Space
- Mouse or other pointing device

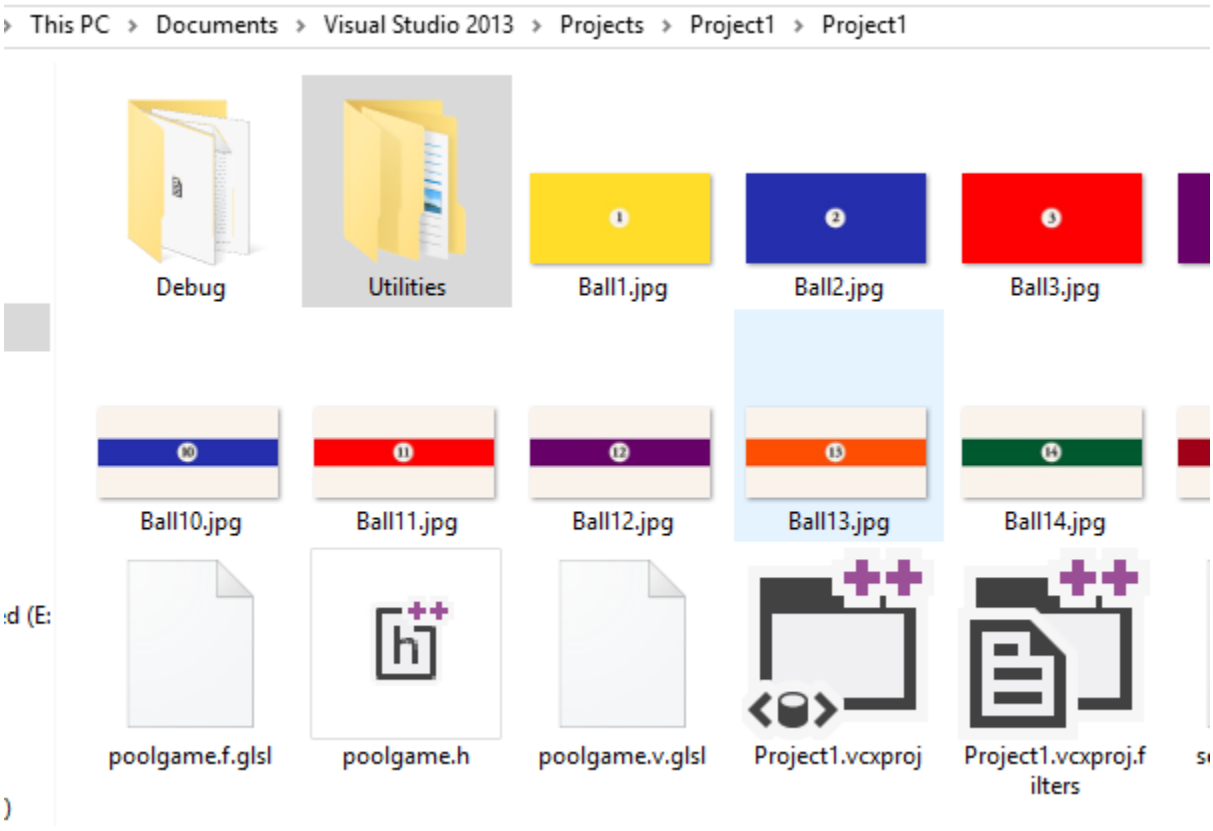
Minimum Software Requirements for Newest features of OPENGL

- Windows 7, 8, 10 – 64 bit (Recommended)
- NVIDIA Graphics Driver version
- OPENGL 2.0
- FreeGLUT 32/64 bit
- GLEW 1.3 – 32/64 bit
- GLM – 32/64 bit
- SOIL – 32/64 bit
- Visual Studio Express 2012 for Windows

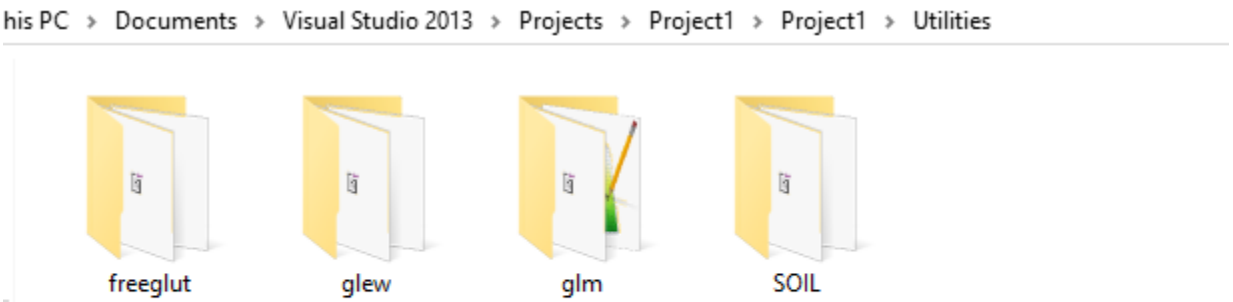
ENVIRONMENT SETUP FOR USING OPENGL UTILITIES

1. Create an Empty Project in Visual Studio
2. Create a main.cpp file with void main() function in it.
3. Build the project
4. Go to Project folder using Windows Explorer (There you will find Debug folder which will only appear once you build the project).

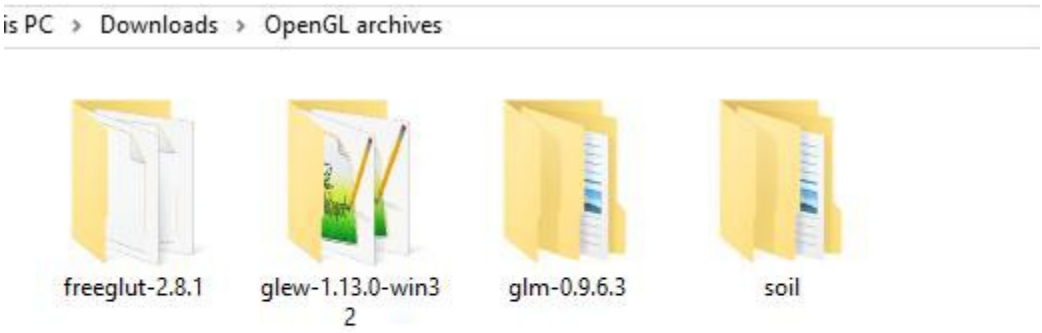
5. Create a new folder named 'Utilities' under Project/Project/



6. Under ‘Utilities’ folder, create 4 folders **: freeglut, glew, glm and SOIL**

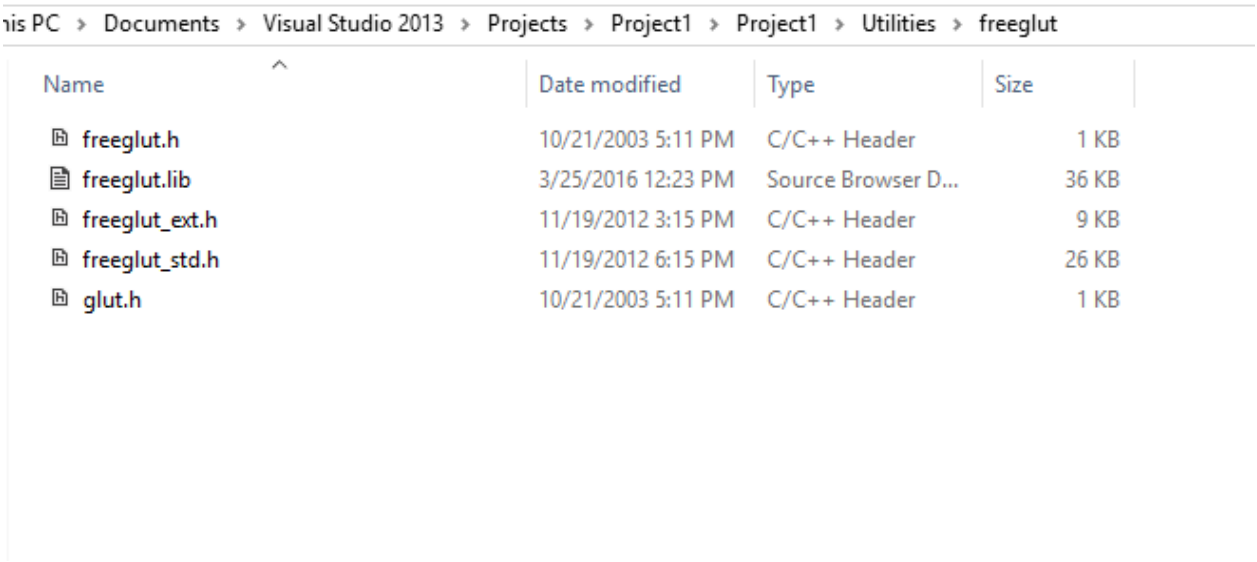


7. Download the following archives of **freeglut, glew, glm** and **SOIL** libraries from the internet.

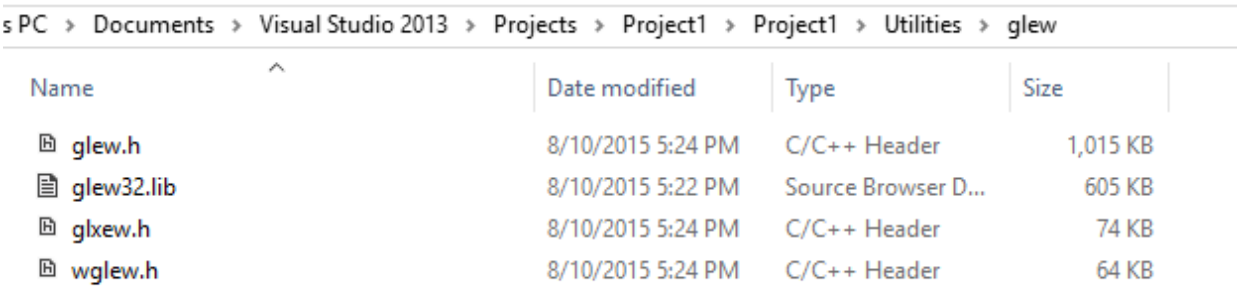


8. Extract **Freeglut** archive, go to Visual Studio/2012 and **load freeglut.sln** into Visual Studio. Build the freeglut as your main project to generate **.dll** and **.lib** files.

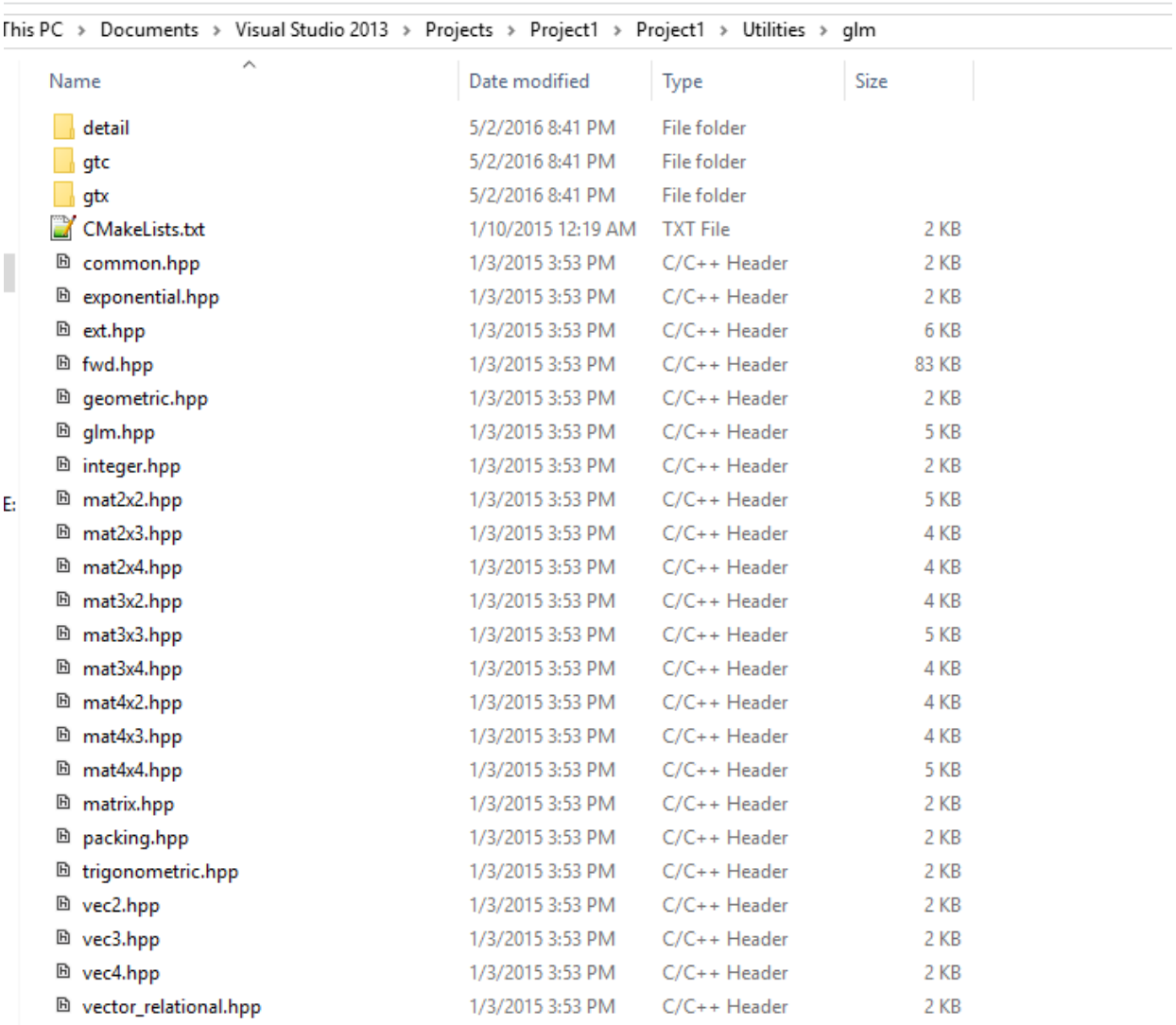
9. To ‘freeglut’ folder, copy the following files :



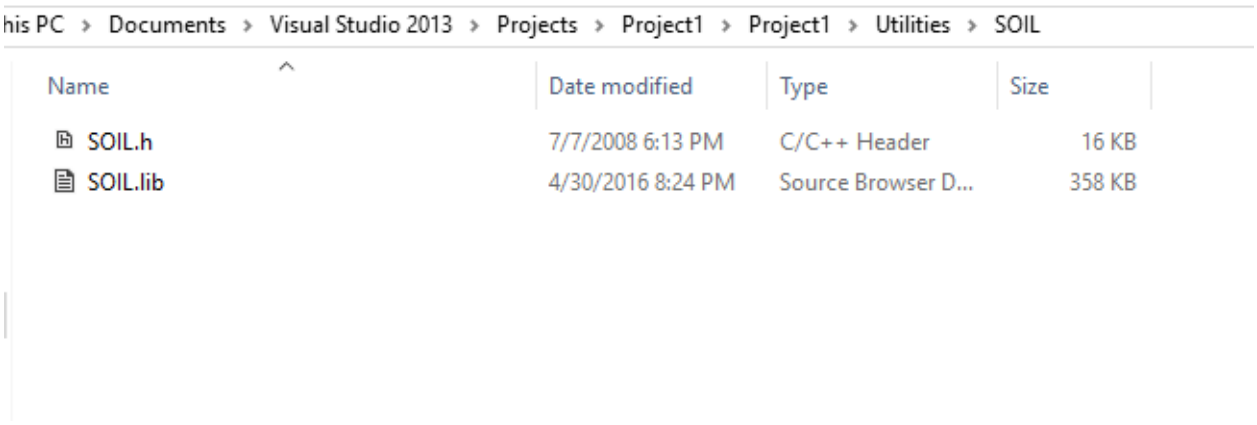
10. To ‘glew’ folder, copy the following files :



11. To ‘glm’ folder copy all the files from GLM/glm directory as it only consists of header and source files. There are no library files.

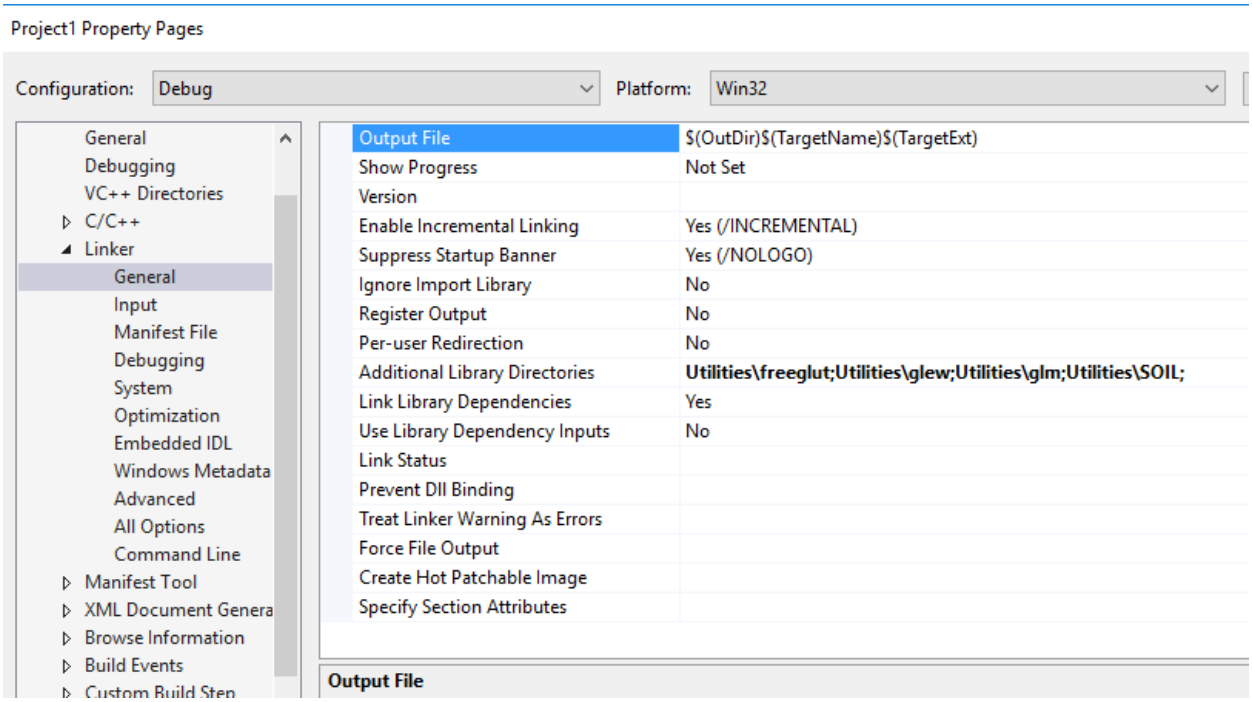


12. Under ‘SOIL’ directory, copy **libSOIL.a** (rename it to **SOIL.lib**) and **SOIL.h**



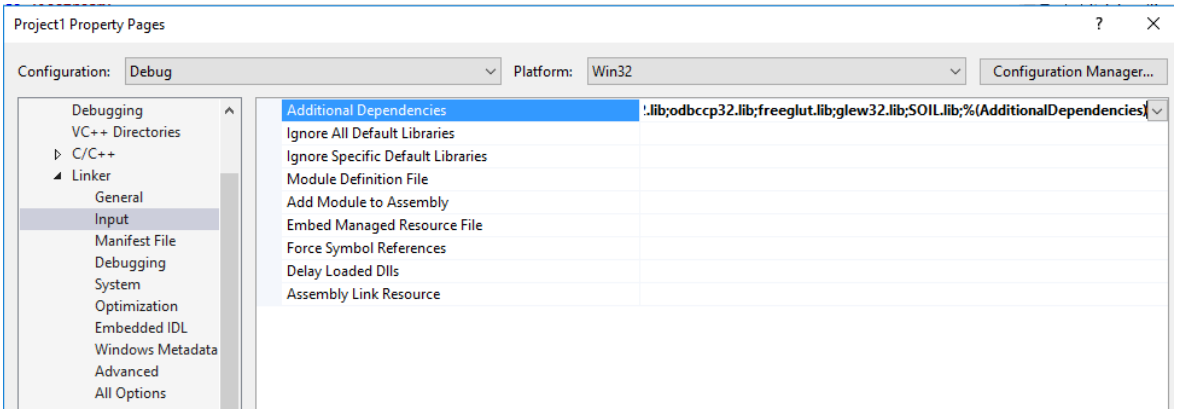
13. Now go to your main project in Visual Studio, open Properties window.
14. Under **Linker -> General**, define **Additional Library Directories**:

    Utilities/freeglut;Utilities/glew;Utilities/glm;Utilities/SOIL

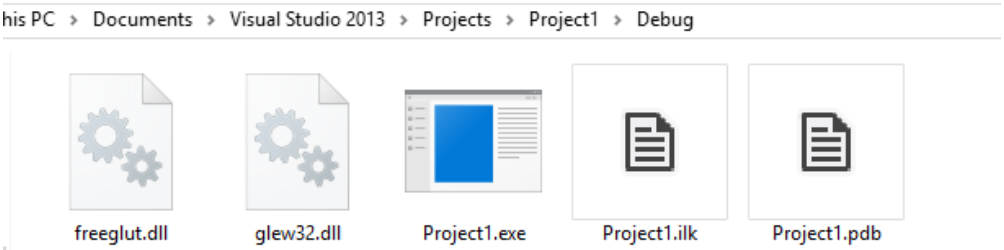


15. Under **Linker -> Input**, add libraries at the end :

    freeglut.lib;SOIL.lib;glew32.lib;



16. Now copy **freeglut.dll** and **glew32.dll** to Project/Debug folder.



17. Now create a new source file, include the required dependencies and create an OPENGL project and include the required functionality using the additional libraries.

8 BALL POOL GAME USING C++ AND OPENGL – CODING & IMPLEMENTATION WITH SCREENSHOTS

1. Setting up structure of Balls and Pool Table

By using method setVBO()

A **Vertex Buffer Object** (VBO) is an OpenGL feature that provides methods for uploading vertex data (position, normal vector, color, etc.) to the video device for non-immediate-mode rendering. VBOs offer substantial performance gains over immediate mode rendering primarily because the data resides in the video device memory rather than the system memory and so it can be rendered directly by the video device.

General VBO functions used in OpenGL

- **GenBuffers**(sizei n, uint *buffers)
  Generates a new VBO and returns its ID number as an unsigned integer. Id 0 is reserved.
- **BindBuffer**(enum target, uint buffer)
  Use a previously created buffer as the active VBO.
- **BufferData**(enum target, sizeiptrARB size, const void *data, enum usage)
  Upload data to the active VBO.
- **DeleteBuffers**(sizei n, const uint *buffers)
  Deletes the specified number of VBOs from the supplied array or VBO id.

2. Load the textures of balls and pool table using SOIL

SOIL is meant to be used as a static library. There are multiple ways to load an image file as texture in OpenGL.

- Load an image file directly as a new OpenGL texture

```
GLuint tex_2d = SOIL_load_OGL_texture
        (
                "img.png",
                SOIL_LOAD_AUTO,
                SOIL_CREATE_NEW_ID,

        SOIL_FLAG_MIPMAPS | SOIL_FLAG_INVERT_Y | SOIL_FLAG_NTSC_SAFE_R
GB | SOIL_FLAG_COMPRESS_TO_DXT
        );
```

- Check for an error during the load process

```
if( 0 == tex_2d )
{
        printf( "SOIL loading error: '%s'\n", SOIL_last_result() );
}
```

- Load another image, but into the same texture ID, overwriting the last one

```
tex_2d = SOIL_load_OGL_texture
        (
                "some_other_img.dds",
                SOIL_LOAD_AUTO,
                tex_2d,
                SOIL_FLAG_DDS_LOAD_DIRECT
        );
```

- Load 6 images into a new OpenGL cube map, forcing RGB

```
GLuint tex_cube = SOIL_load_OGL_cubemap
        (
                "xp.jpg",
                "xn.jpg",
                "yp.jpg",
                "yn.jpg",
                "zp.jpg",
                "zn.jpg",
                SOIL_LOAD_RGB,
                SOIL_CREATE_NEW_ID,
                SOIL_FLAG_MIPMAPS
        );
```

- Load and split a single image into a new OpenGL cube map, default format

```
/* face order = East South West North Up Down => "ESWNUD", case
sensitive! */
GLuint single_tex_cube = SOIL_load_OGL_single_cubemap
        (
                "split_cubemap.png",
                "EWUDNS",
                SOIL_LOAD_AUTO,
                SOIL_CREATE_NEW_ID,
                SOIL_FLAG_MIPMAPS
        );
```

- Load an image as a heightmap, forcing greyscale (so channels should be 1)

```
int width, height, channels;
unsigned char *ht_map = SOIL_load_image
        (
                "terrain.tga",
                &width, &height, &channels,
```

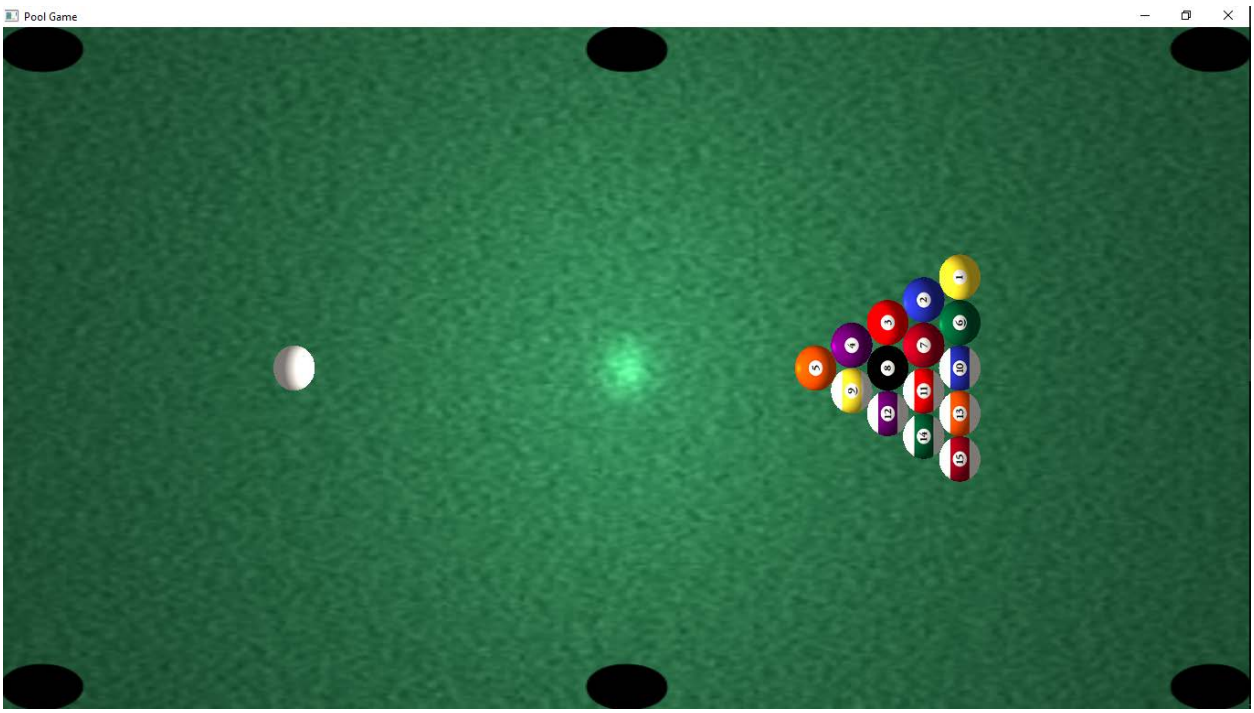
```
                SOIL_LOAD_L
        );
```

- Save a screenshot of your OpenGL game engine, running at 1024x768

```
save_result = SOIL_save_screenshot
        (
                "awesomenessity.bmp",
                SOIL_SAVE_TYPE_BMP,
                0, 0, 1024, 768
        );
```

3. Set Initial positions of the Balls on the table

Position the cue ball and all other balls on the positions as shown :



4. Set the initial velocities of all the balls to 0

Setting the initial linear velocities (ballVel[i]) and Angular velocities (ballAngVel[i]) of all the balls to 0.

```
ballVel[i] = glm::vec3(0.0f);
```

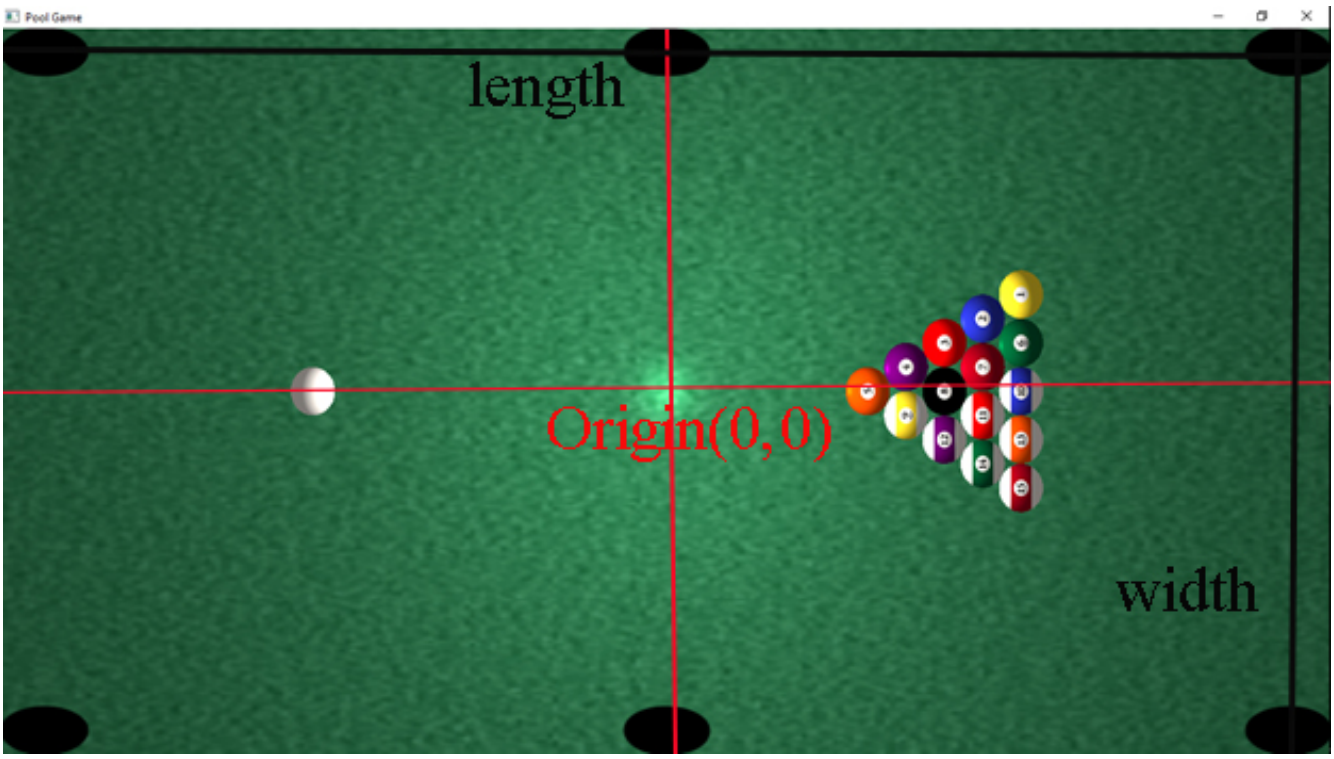
```
ballAngVel[i] = glm::vec3(0.0f);
```

5. Define the positions of the pockets on the table

Take in the width and length of the table

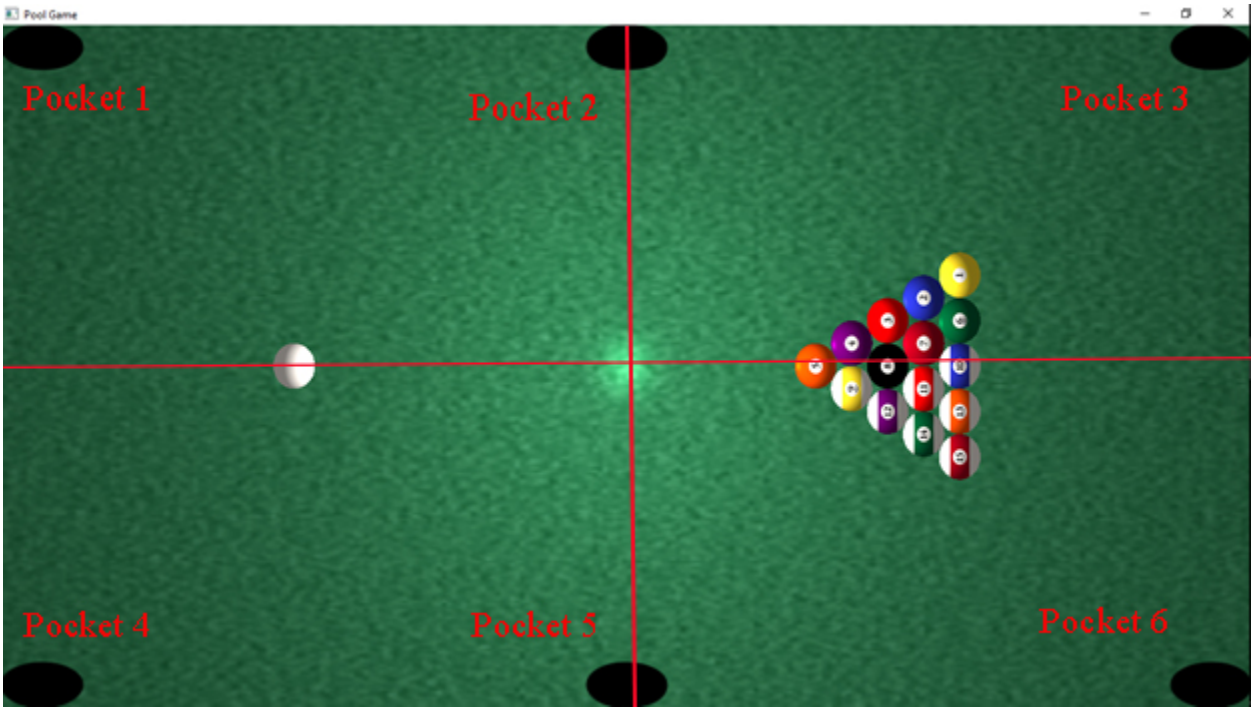
In the project the length and width of the table are 60 and 30 units respectively.

The center of the table is considered the origin and is depicted as following :



So we position the 6 pockets as :

- Pocket1 = (-length/2 , width/2)
- Pocket2 = (0 , width/2)
- Pocket3 = (length/2 , width/2)
- Pocket4 = (-length/2 , -width/2)
- Pocket5 = (0 , -width/2)
- Pocket6 = (length/2 , -width/2)



6. Load shaders and Create Program

CreateShader() method to read the GLSL files and create shaders by making use of the following methods :

- **glCreateShader**(shader_type) – it creates an empty shader object (handle) of the wanted type
- **glShaderSource**(shader, count, shader_code, length) – it loads the shader object with the code; count is usually set to 1, because you normally have one character array, shader_code is the array of code and length is normally set to NULL (thus, it will read code from the array until it reaches NULL) , however here I set the actual length.
- **glCompileShader**(shader) – compiles the code
- **glGetShaderiv**(shader, GLenum ,GLint) – Check for errors and output them to the console

7. GLUT Display Function : glutDisplayFunc()

- **glutDisplayFunc()** sets the display callback for the *current window*. When GLUT determines that the normal plane for the window needs to be redisplayed, the display callback for the window is called. Before the callback, the *current window* is set to the window needing to be redisplayed and (if no overlay display callback is registered) the *layer in use* is set to the normal plane. The display callback is called with no parameters. The entire normal plane region should be redisplayed in response to the callback (this includes ancillary buffers if your program depends on their state).
- GLUT determines when the display callback should be triggered based on the window's redisplay state. The redisplay state for a window can be either set explicitly by calling glutPostRedisplay() or implicitly as the result of window damage reported by the window system. Multiple posted redisplays for a window are coalesced by GLUT to minimize the number of display callbacks called.
- When an overlay is established for a window, but there is no overlay display callback registered, the display callback is used for redisplaying *both* the overlay and normal plane (that is, it will be called if either the redisplay state or overlay redisplay state is set). In this case, the *layer in use* is *not* implicitly changed on entry to the display callback.

8. GLUT Reshape function : glutReshapeFunc()

- `glutReshapeFunc` sets the reshape callback for the *current window*. The reshape callback is triggered when a window is reshaped. A reshape callback is also triggered immediately before a window's first display callback after a window is created or whenever an overlay for the window is established. The `width` and `height` parameters of the callback specify the new window size in pixels. Before the callback, the *current window* is set to the window that has been reshaped.
- If a reshape callback is not registered for a window or `NULL` is passed to `glutReshapeFunc` (to deregister a previously registered callback), the default reshape callback is used. This default

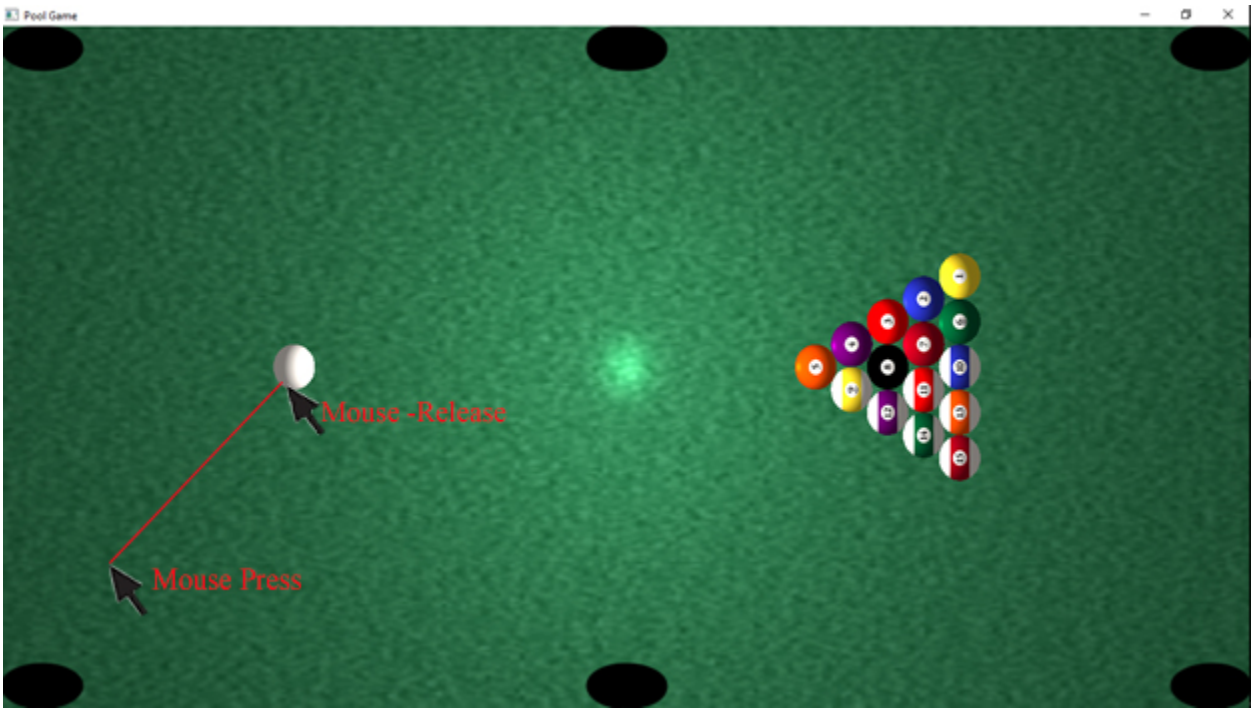
callback will simply call `glViewport(0,0,width,height)` on the normal plane (and on the overlay if one exists).

9. GLUT Mouse Function : glutMouseFunc()

`glutMouseFunc` sets the mouse callback for the *current window*. When a user presses and releases mouse buttons in the window, each press and each release generates a mouse callback. The `button` parameter is one of `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, or `GLUT_RIGHT_BUTTON`. For systems with only two mouse buttons, it may not be possible to generate `GLUT_MIDDLE_BUTTON` callback. For systems with a single mouse button, it may be possible to generate only a `GLUT_LEFT_BUTTON` callback. The `state` parameter is either `GLUT_UP` or `GLUT_DOWN` indicating whether the callback was due to a release or press respectively. The `x` and `y` callback parameters indicate the window relative coordinates when the mouse button state changed.

In the 8 ball Pool Game, to hit the ball, press down the left button of the mouse, mouse the cursor to a ball surface and release the mouse. The end position on the ball is the hitting point, and the vector from which the mouse moved towards the hitting point gives the force on the ball at the hitting point.

The mouse function calculates the last position of the mouse and the release point (current position) of the mouse and passes to the **glutMotionFunc()**



10. GLUT Motion Functions – When the ball is hit – glutMotionFunc()

`glutMotionFunc` sets the motion callback respectively for the *current window*. The motion callback for a window is called when the mouse moves within the window while one or more mouse buttons are pressed.

The implementation after the mouse is released, includes:

- Calculate the force
- Assign an initial velocity to the ball hit

For calculating hitting force we use windows coordinates to decide the size and direction of the force.

Last position of the mouse : (last_x,last_y)

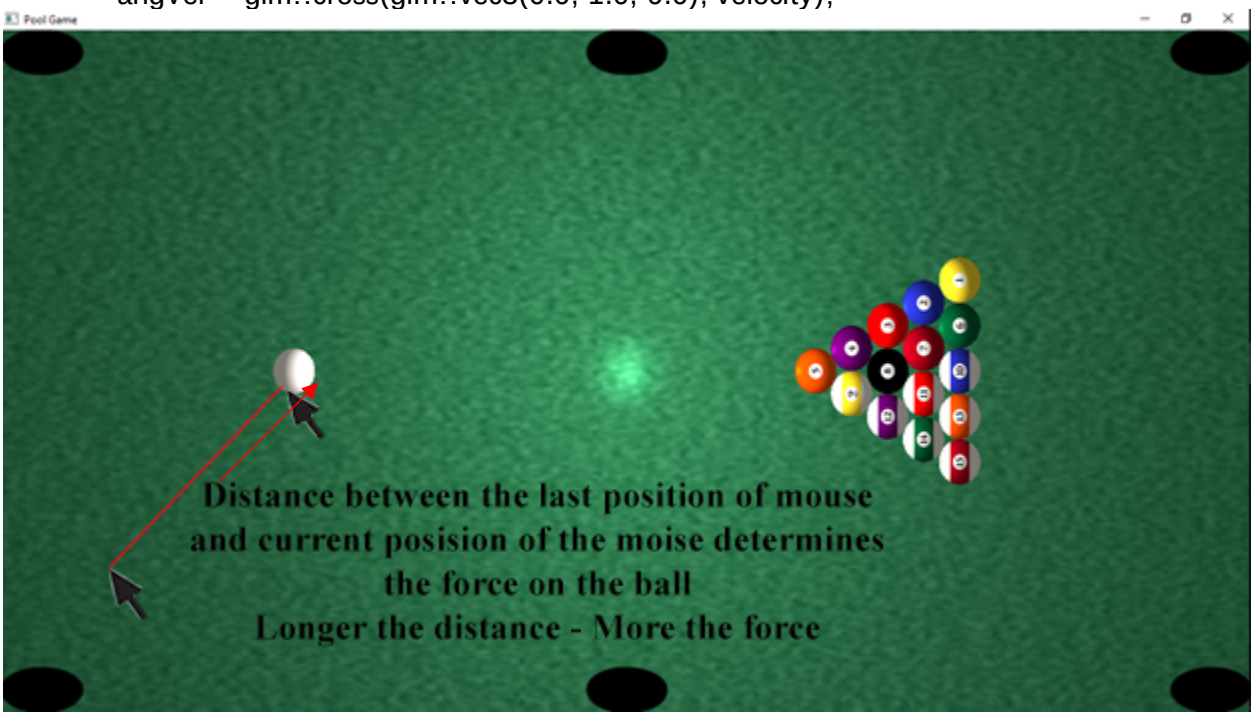
Current position of the mouse : (cur_x,cur_y)

Linear velocity of the ball v is defined in three coordinates :

```
velocity[0] = cur_y - last_y;
velocity[1] = 0.0;                          //Velocity in Z
is 0 velocity[2] = -(cur_x - last_x);
```

And we use the following equation to derive the initial angular velocity for the ball.

```
angVel = glm::cross(glm::vec3(0.0, 1.0, 0.0), velocity);
```



11. Handle Ball Motion

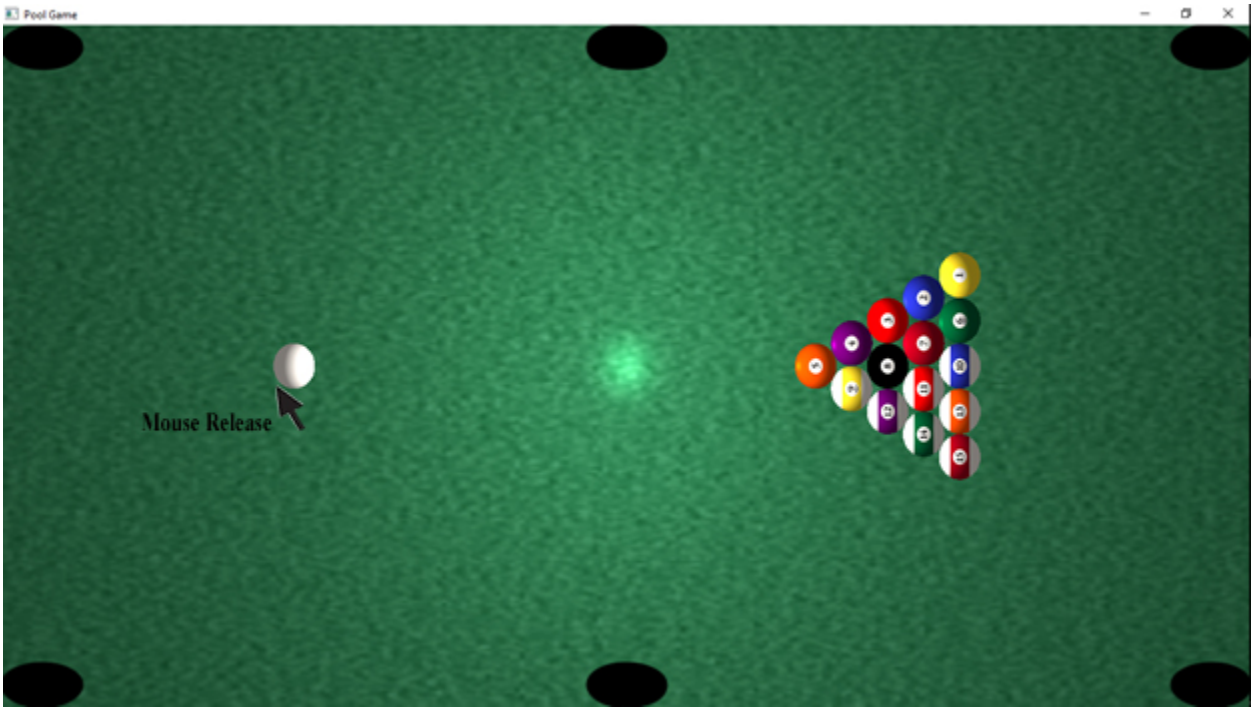
a. Friction

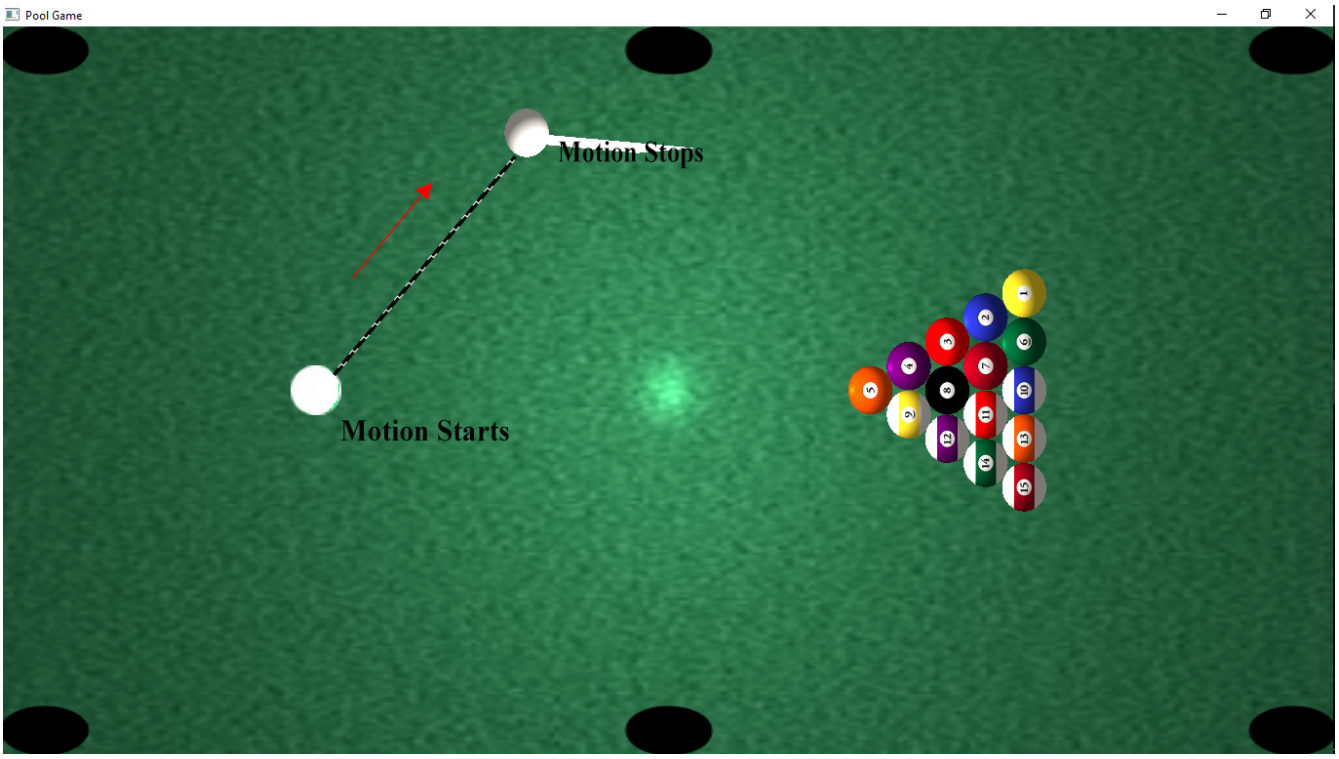
Once the ball has an initial velocity, it begins to move across the table. After the ball is hit and it starts to move, the only force acting on the cue ball is the force of friction from the table.

I am using the following equations to update the linear and angular velocities of the ball when it moves across the table.

```
glm::vec3 v = BallID.ballVel[i];
glm::vec3 nextVel = v /0.1;

glm::vec3 w = BallID.ballAngVel[i]; glm::vec3 nextAngVel = w/0.1;
```




Pool Game
Motion Stops
Motion Starts

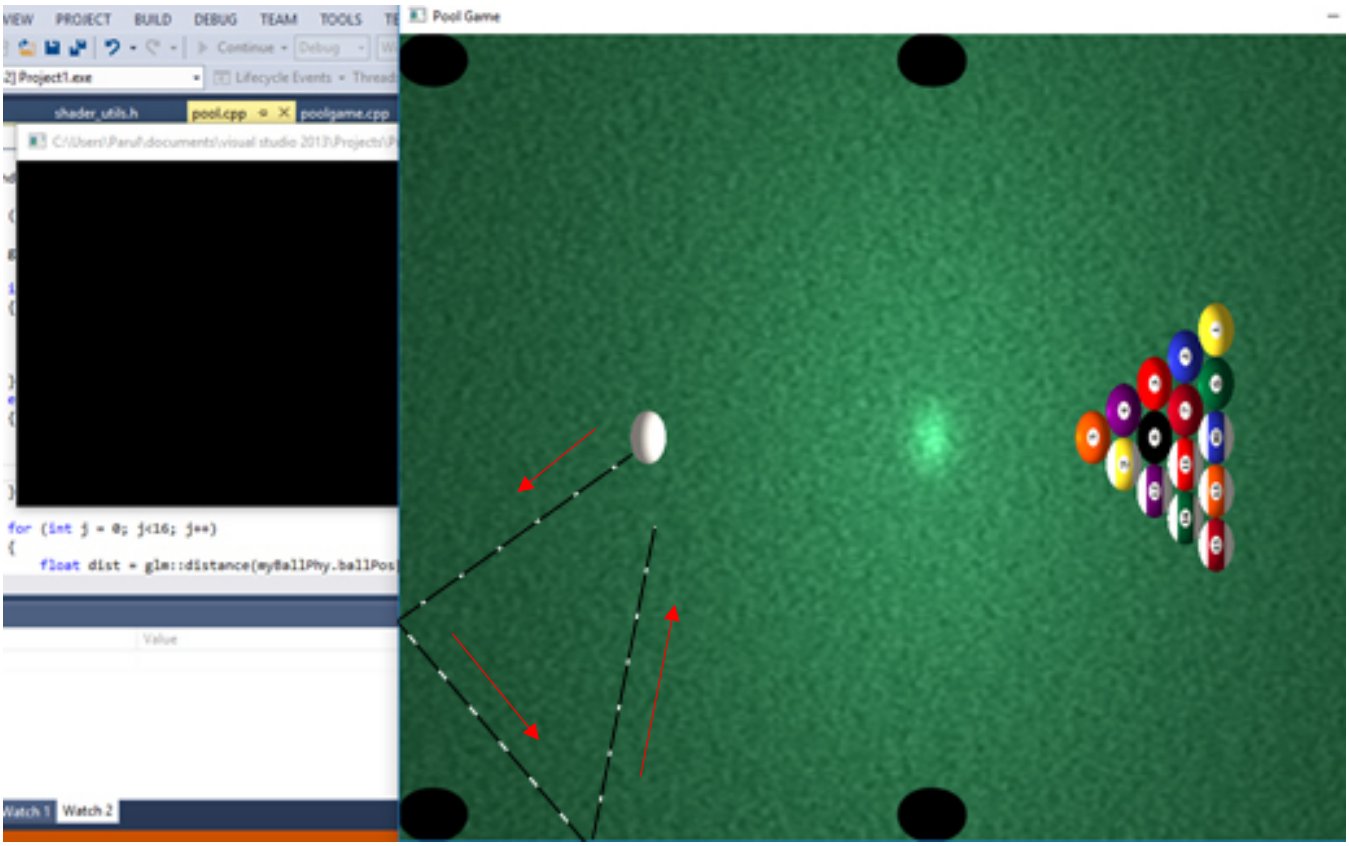
b. Collision

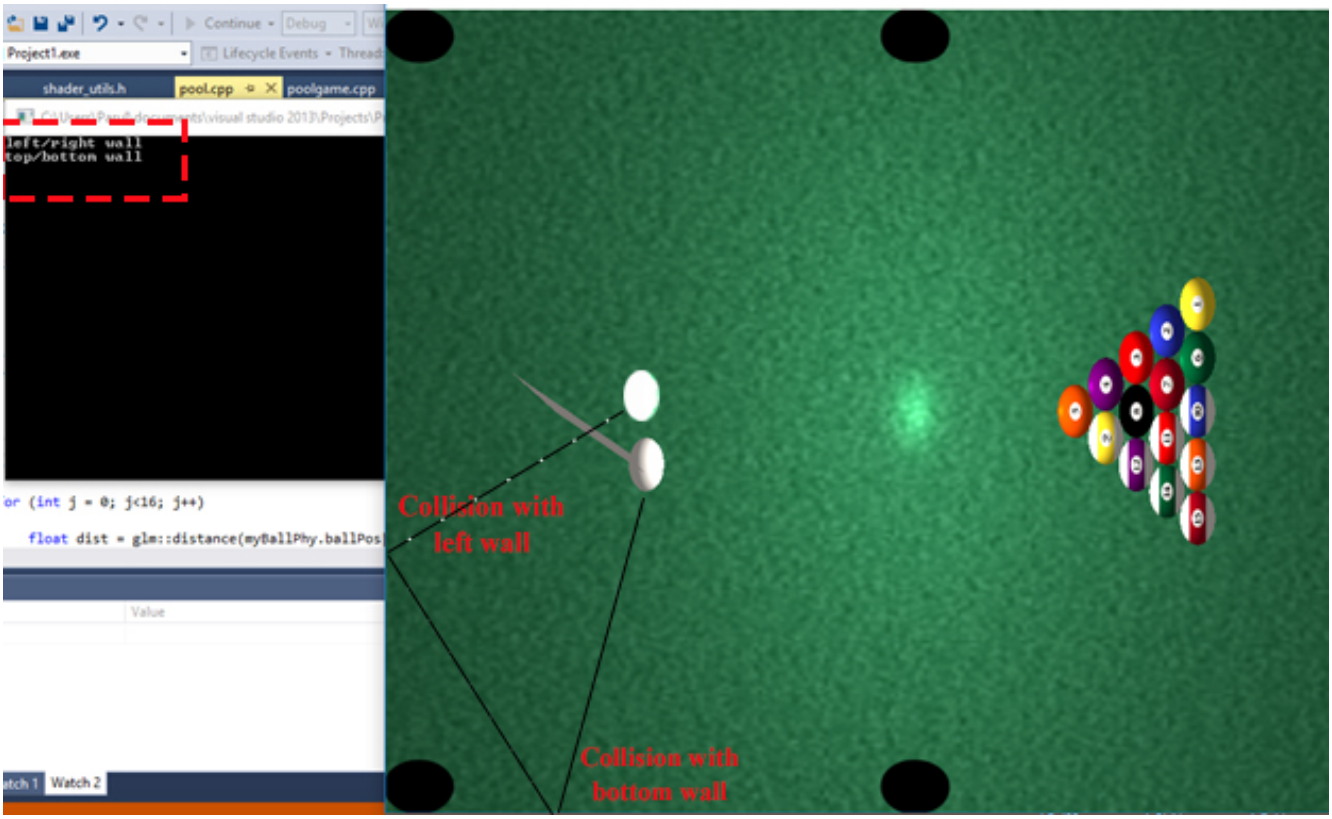
- **Collision between a ball and a wall**

For top/bottom wall *: wall = -1.0;*

For left/right wall *: wall = 1.0;*

```
ballVel[i] *= glm::vec3(wall, 0.0, -wall);
ballAngVel[i] = 2.5f*glm::cross(glm::vec3(0.0, 1.0, 0.0), ballVel[i]);
```




Project1.exe
shader_utils.h
pool.cpp
poolgame.cpp
left/right wall
top/botton wall
Collision with
left wall
Collision with
bottom wall

- **Collision between two balls**

If a collision is detected, we need to update the velocity vectors of the ball involved. And we use conservation of energy and conservation of momentum principles to derive the updated velocities. The following equations show the final results.

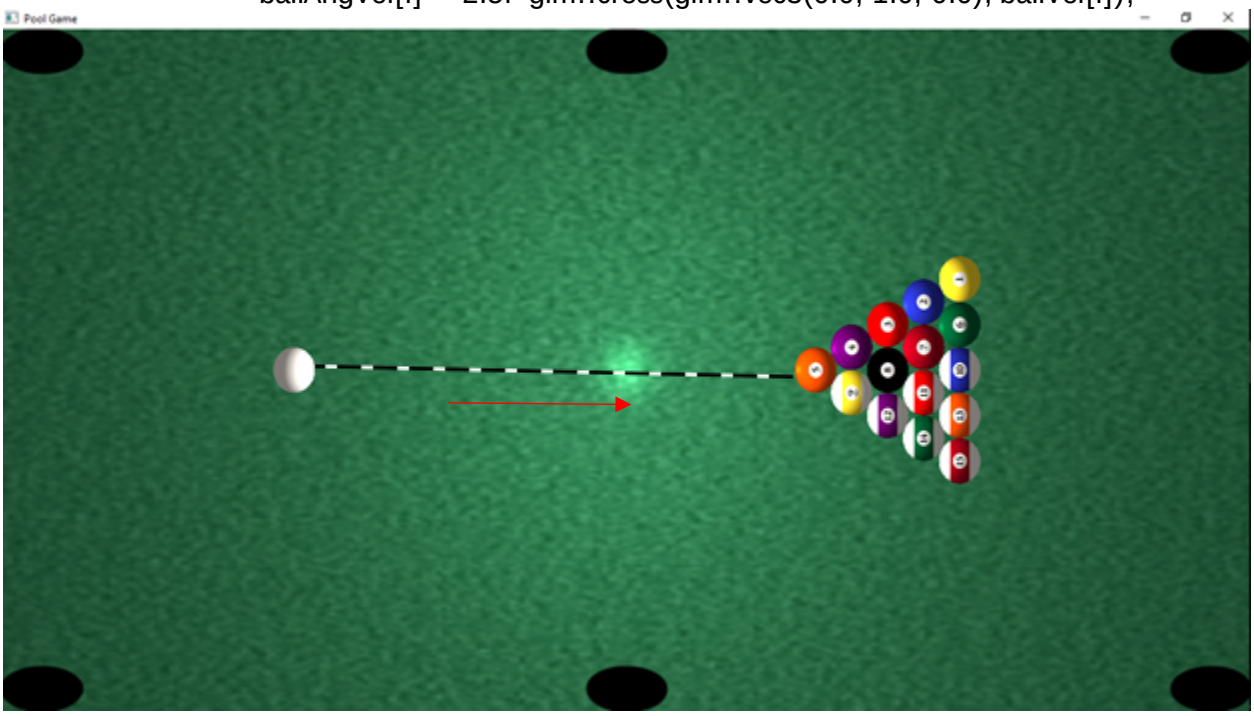
For linear velocity:

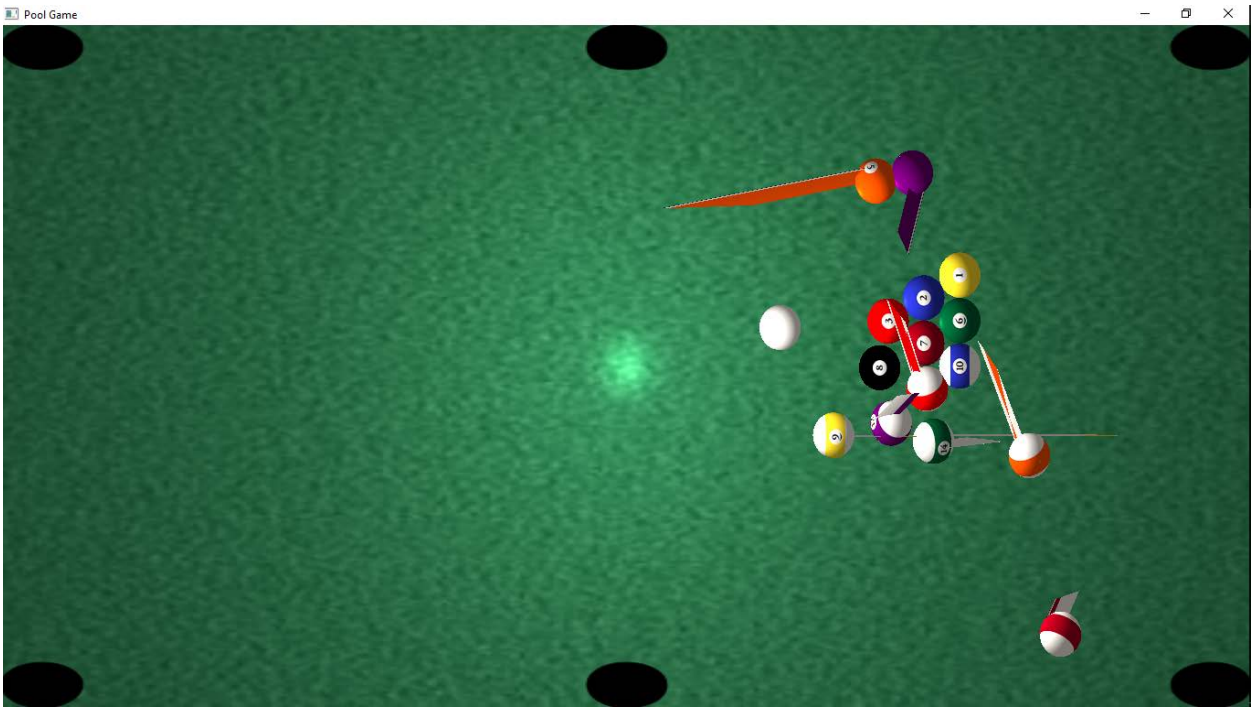
```
dist = glm::distance(myBallPhy.ballPos[j], p);
if ((dist>2) ||i!=j)            //i and j are two ballIDs that are going
through collsion
{
      glm::vec3 niVel = glm::dot(ballVel[i], n)*n;
      glm::vec3 njVel = glm::dot(ballVel[j], -n)*(-
      n); glm::vec3 tiVel = ballVel[i] - niVel;
      glm::vec3 tjVel = ballVel[j] -
      njVel; ballVel[i] = njVel + tiVel;
      ballVel[j] = niVel + tjVel;
}
```

The vectors nVel and tVel are the normal and tangential components of the velocity of ball 1. And the vn2 and vt2 are for ball 2.

For angular velocity:

```
ballAngVel[i] = 2.5f*glm::cross(glm::vec3(0.0, 1.0, 0.0), ballVel[i]);
```




Pool Game

SYSTEM SECURITY MEASURES

1. What important things to keep in mind while dealing with OpenGL?

The important thing to know is that opengl32.dll belongs to Microsoft. No one can modify it. You must not replace it. You must not ship your application with this file.

2. To check what version of freeGLUT and GLEW, you are using?

a. To check the version of OpenGL

Use the function glGetString, with `GL_VERSION` passed as argument. This will return a null-terminated string.

```
versionGL = (char *)(glGetString(GL_VERSION));
cout << "OpenGL version: " << versionGL << endl << endl;
```

b. To check the version of FreeGLUT

```
versionFreeGlutInt = (glutGet(GLUT_VERSION));
cout << "FreeGLUT version: " << versionFreeGlutString << endl << endl;
```

c. To check the version of GLEW

```
cout << "GLEW version: " << GLEW_VERSION << "." << GLEW_VERSION_MAJOR << "."
<< GLEW_VERSION_MINOR << "." << GLEW_VERSION_MICRO << endl;
```

3. To make sure GLEW and SOIL are initialized properly.

a. To check whether the GLEW is initialized properly and whether the version of GLEW is what you require.

```
if (glewInit()) { // checks if glewInit() is activated
        cerr << "Unable to initialize GLEW." << endl;
       exit(EXIT_FAILURE);
}
if (glewIsSupported("GL_VERSION_4_5"))
{
  std::cout << " GLEW Version is 4.5\n ";
```

```
}
else
{
   std::cout << "GLEW 4.5 not supported\n ";
}
```

b. To check whether the SOIL is able to load texture properly

```
mytexture_id[i] = SOIL_load_OGL_texture
                  (
                  texture_files[i].c_str(),
                  SOIL_LOAD_AUTO,
                  SOIL_CREATE_NEW_ID,
                  SOIL_FLAG_INVERT_Y | SOIL_FLAG_TEXTURE_REPEATS
                  );
if (mytexture_id[i] == 0)
                  cerr << "SOIL loading error: '" << SOIL_last_result() <<
"' (" << texture_files[i] << ")" << endl;
```

4. Common Mistakes

- Adding **Utilities** folder in Solution Explorer.
- Removing other libs from Additional Dependencies
- Invalid paths provided in Additional Liba
- Copying wrong libs. You will get LNK errors.
- Using FreeGLUT for x64 and GLEW for x86 or vice versa. LNK errors

5. Make sure you delete the program and buffers and free the resources acquired by it whenever you terminate the execution of the game.

```
void endPoolGame()
{
       glDeleteProgram(program);
       glDeleteBuffers(1, &ball_vbo);
       glDeleteBuffers(1, &table_vbo);
}
```

Even though you terminate the execution in Visual Studio, the shader object will not actually be deleted until it us unattached from all programs. When you delete the program, all attached shaders become unattached, so it is only *then* that they will be deleted.

FUTURE SCOPE OF THE PROJECT

The 8 ball pool project could also be built using Java Swing and AWT. But using OpenGL and another utilities in C/C++, provides many advantages and thus enables the future use of OpenGL.

First, you need not set up any particular mouse/keyboard/joystick controls in OpenGL. These functionalities are provided by FreeGLUT and you just need to define what will actually happen when the mouse/keyboard event takes place.

Second, OpenGL provides smooth graphics. It is so because of Shaders. Shaders are the responsibility of the Graphical Processing Unit (GPU), so that CPU spends its time setting up other functionalities while GPU takes care of drawing the vertices and setting up textures.

Third, Building a 3-D view in OpenGL is actually easy as the screen is already divided into 3 dimensions (X, Y, Z).

Fourth, OpenGL is handled using GLM i.e. all the mathematics we require to implement our program can be done using GLM source files. One does not need to know all the STL libraries or other data structures.

The only disadvantage due to which a beginner programmer would avoid the use of OpenGL is due to

First, Very complicated functions and methods to implement OpenGL functionalities. But there are several tutorials that can guide a coder about various methods to be used.

Second, it is sometimes difficult to write shaders for complicated programs as one needs to define the position of vertex and textures for proper implementation of graphics.

BIBLIOGRAPHY

- https://www.opengl.org/
- https://en.wikipedia.org/wiki/OpenGL
- https://en.wikipedia.org/wiki/FreeGLUT
- http://glew.sourceforge.net/
- https://en.wikipedia.org/wiki/Glew
- http://www.lonesock.net/soil.html
- https://www.opengl.org/wiki/Image_Libraries
- http://www.opengl-tutorial.org/
- http://learnopengl.com/
- http://in2gpu.com/2014/10/15/setting-up-opengl-with-visual-studio/
- https://www.opengl.org/wiki/Shader
- https://en.wikipedia.org/wiki/OpenGL_Shading_Language
- https://github.com/zhongyn/opengl-8-ball-pool-game
- http://www.dreamincode.net/forums/topic/123107-8ball-game-c/
- http://glm.g-truc.net/0.9.7/index.html
- http://glm.g-truc.net/0.9.2/api/a00001.html
- http://www.c-jump.com/bcc/common/Talk3/Math/GLM/GLM.html