Group 8 - K57CA
Course Project
Computer Graphics
May 20, 2015

Billiards Motion Emulation

Trương Quốc Tuấn
Nguyễn Thạc Thống
Lê Văn Giáp

Table of Contents

1. Introduction

1.1. Requirements

- Loading billiards table, balls and room models.
- Move camera around, closer and further from table.
- Shot the white ball and simulate the physics collisions between ball-ball, ball-table.

1.2. System requirements

- OpenGL
- OpenGL Utility Toolkit (GLUT)
- Programming language: C++

1.3. Third-party libraries

- glm (OpenGL Mathematics): matrix and vector calculation library
http://glm.g-truc.net/0.9.6/index.html
- SOIL (Simple OpenGL Image Library): texture loading library
http://www.lonesock.net/soil.html

2. Implementation

2.1. Resource files

2.1.1. OBJ File

Obj file is an ascii file that stores 3D geometric models. It contains information about vertex, normals, and textures.
Each line of .obj file starts from a special character or word.

File format:

Starting character / word	Meaning
#	Comment line
v	Defines vertex points in 3D space, with coordinates of(x, y,z,w), w is optional
vt	Define texture coordinates (u,v,w) where w is optional
vn	Defines normals (x,y,z)
f	Defines a face, composed of index of vertex/texture/normal. Note that texture and normal is optional. If both texture and normal are absent f 1 2 3 If texture exists, but normal absent f 1/3 2/2 3/4 If texture is absent, but normal exists f 1//3 2//2 3//4 if all them exist f 1/2/3 2/3/2 3/2/4 each face can contains more than 3 elements
mtllib	Materials that describe the visual aspects of the polygons are stored in external .mtl files. The .mtl file may contain one or more named material definitions.
o	Object name
g	group name
usemtl	Defines a material to use, this material will continue to be used until another usemtl line (corresponding material is saved in .mtl file)
s	Smooth shading
mtllib	Point to an external .mtl file that saves the object materials

2.1.2. MTL File

Mtl file is an ascii file, that saves object materials used for obj files.
Each line of .mtl file starts from a special character or word.

File Format:

Starting character / word	Meaning
newmtl	Start a definition of a new material
Ka	ambient color (r,g,b)
Kd	diffuse color (r,g,b)
Ks	specular color (r,g,b)
illum	Define the illumination model: **illum** = 1 a flat material with no specular highlights, **illum** = 2 denotes the presence of specular highlights
Ns	shininess of the material
d or Tr	the transparency of the material
map_Ka	names a file containing a texture map, which should just be an ASCII dump of RGB values
usemtl	Defines a material to use, this material will continue to be used until another usemtl line (corresponding material is saved in .mtl file)
s	Smooth shading
mtllib	Point to an external .mtl file that saves the object materials

Reference: http://en.wikipedia.org/wiki/Wavefront_.obj_file

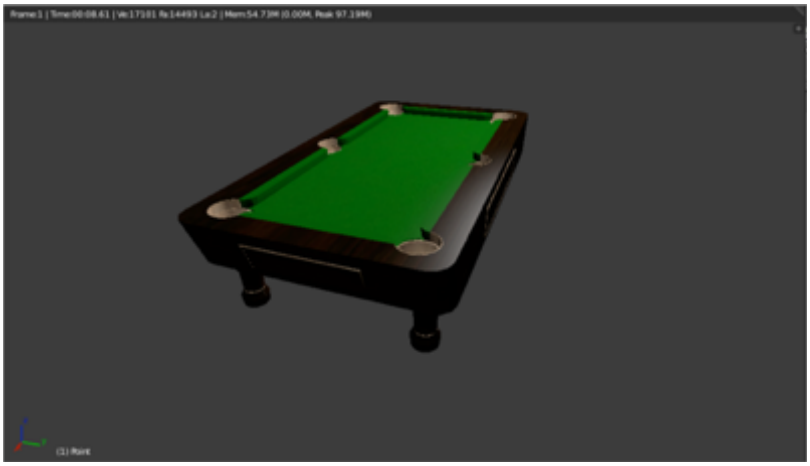
2.1.3. Blender

Blender is a professional free and open-source 3D computer graphics software product used for creating animated films, visual effects, art, 3D printed models, interactive 3D applications and video games.

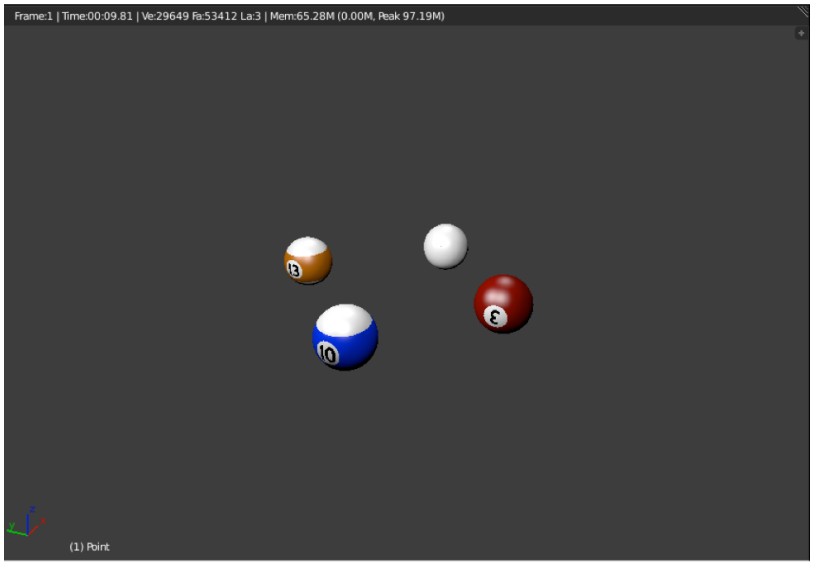
Homepage: https://www.blender.org/

In this project, we use blender to create new models and reshape some downloaded models that we found on the internet.

Pool table:



Balls:



Room:

Put it all together:

2.2. Model loading

Instead of using library, we build ourselves a class Model_OBJ to load 3D model from .obj and .mtl file. In this class, we define some data structures to help us manage and render models efficiently.

Object structure for managing many objects in one model:

```
typedef struct {
    material* material;
    float* faces_triangles;
    float* texts_coords;
    float* norm_vectors;
    long total_triangles_floats;
} object;
```

Material structure for managing many materials in one model:

```
typedef struct {
    string name;
    float Ns;
    float Ka[3];
    float Kd[3];
    float Ks[3];
    float Ni;
    float d;
    int illum;
    bool hasTexture;
    string texture;
} material;
```

Texture structure for loading texture used by SOIL library:

```
typedef struct {
    unsigned char* image;
    int width;
    int height;
} texture;
```

OBJ loading:

```
//  Load material file
if (type.compare("mt") == 0)
{
    string l = "mtllib ";
    string mtlFile = line.substr(l.size());
    mtlFile = "resource/" + mtlFile;
```

```
    const char* cstr = mtlFile.c_str();
    total_materials = loadMTL(cstr);
    cout << "Number of mtl: " << loadMTL(cstr) << endl;
}

// Create new object
else if (type.compare("us") == 0)
{
    obj_textures_floats = 0;
    obj_normal_vectors_floats = 0;
    obj = &objects[total_objects];
    string l = "usemtl ";
    string mtlName = line.substr(l.size());

    for (int i = 0; i < total_materials; i++)
    {
        if (mtlName.compare(string(materials[i].name)) == 0)
        {
            obj->material = &materials[i];
        }
    }

    obj->total_triangles_floats = 0;
    obj->faces_triangles = (float*) malloc (fileSize);
    obj->norm_vectors = (float*) malloc (fileSize);
    if (hasTexture)
    {
        obj->texts_coords = (float*) malloc (fileSize);
    }

    total_objects++;
}

//  Read vertex
else if (type.compare("v ") == 0)
{
    sscanf(line.c_str(), "v %f %f %f",
        &vertexBuffer[total_vertices_floats],
        &vertexBuffer[total_vertices_floats + 1],
        &vertexBuffer[total_vertices_floats + 2]);
    total_vertices_floats += FLOATS_PER_VERTEX;
}

//  Read texture coordinates
else if (type.compare("vt") == 0)
```

```
{
    sscanf(line.c_str(), "vt %f %f",
        &vtBuffer[total_textures_coords_floats],
        &vtBuffer[total_textures_coords_floats + 1]);

    total_textures_coords_floats += FLOATS_PER_TEXTURE_COOR;
}

//  Read normal vectors
else if (type.compare("vn") == 0)
{
    sscanf(line.c_str(), "vn %f %f %f",
        &vnBuffer[total_normal_vectors_floats],
        &vnBuffer[total_normal_vectors_floats + 1],
        &vnBuffer[total_normal_vectors_floats + 2]);

    total_normal_vectors_floats += FLOATS_PER_VERTEX;
}

//  Read faces
else if (type.compare("f ") == 0)
{
    int tCounter = 0;
    unsigned int vertexNumber[3], vtNumber[3], vnNumber[3];

    if (hasTexture)
    {
        sscanf(line.c_str(),"f %d/%d/%d %d/%d/%d %d/%d/%d",
        &vertexNumber[0], &vtNumber[0], &vnNumber[0],
        &vertexNumber[1], &vtNumber[1], &vnNumber[1],
        &vertexNumber[2], &vtNumber[2], &vnNumber[2] );

        vtNumber[0] -= 1;
        vtNumber[1] -= 1;
        vtNumber[2] -= 1;

        tCounter = 0;
        for (int i = 0; i < VERTICES_PER_FACE; i++)
        {
            obj->texts_coords[obj_textures_floats + tCounter] =
                vtBuffer[2*vtNumber[i]];
            obj->texts_coords[obj_textures_floats + tCounter + 1] =
                vtBuffer[2*vtNumber[i] + 1];
            tCounter += 2;
        }
```

```
            obj_textures_floats += FLOATS_PER_TEXTURES_COORS;
        }
        else
        {
            sscanf(line.c_str(),"f %d//%d %d//%d %d//%d",
            &vertexNumber[0], &vnNumber[0],
            &vertexNumber[1], &vnNumber[1],
            &vertexNumber[2], &vnNumber[2]);
        }

        vertexNumber[0] -= 1;
        vertexNumber[1] -= 1;
        vertexNumber[2] -= 1;

        vnNumber[0] -= 1;
        vnNumber[1] -= 1;
        vnNumber[2] -= 1;

        //  object faces triangles
        tCounter = 0;
        for (int i = 0; i < VERTICES_PER_FACE; i++)
        {
            obj->faces_triangles[obj->total_triangles_floats+tCounter] =
                vertexBuffer[3*vertexNumber[i] ];
            obj->faces_triangles[obj->total_triangles_floats+tCounter+1] =
                vertexBuffer[3*vertexNumber[i] + 1];
            obj->faces_triangles[obj->total_triangles_floats+tCounter+2] =
                vertexBuffer[3*vertexNumber[i] + 2];
            tCounter += 3;
        }
        obj->total_triangles_floats += FLOATS_PER_TRIANGLE;

        //  object normal vectors
        tCounter = 0;
        for (int i = 0; i < VERTICES_PER_FACE; i++)
        {
            obj->norm_vectors[obj_normal_vectors_floats + tCounter] =
                vnBuffer[3*vnNumber[i]];
            obj->norm_vectors[obj_normal_vectors_floats + tCounter + 1] =
                vnBuffer[3*vnNumber[i] + 1];
            obj->norm_vectors[obj_normal_vectors_floats + tCounter + 2] =
                vnBuffer[3*vnNumber[i] + 2];
            tCounter += 3;
        }
        obj_normal_vectors_floats += FLOATS_PER_TRIANGLE;
}
```

MTL loading:

```
//  Read new material
if (type.compare("ne") == 0)
{
    nMtl++;
    materials[nMtl].hasTexture = false;

    string l = "newmtl ";
    materials[nMtl].name = line.substr(l.size());
}

//  Read new shininess value
else if (type.compare("Ns") == 0)
{
    sscanf(line.c_str(), "Ns %f", &(materials[nMtl].Ns));
}

//  Read ambients
else if (type.compare("Ka") == 0)
{
    sscanf(line.c_str(), "Ka %f %f %f",
        &(materials[nMtl].Ka[0]),
        &(materials[nMtl].Ka[1]),
        &(materials[nMtl].Ka[2]));
}

//  Read diffuses
else if (type.compare("Kd") == 0)
{
    sscanf(line.c_str(), "Kd %f %f %f",
        &(materials[nMtl].Kd[0]),
        &(materials[nMtl].Kd[1]),
        &(materials[nMtl].Kd[2]));
}

//  Read speculars
else if (type.compare("Ks") == 0)
{
    sscanf(line.c_str(), "Ks %f %f %f",
        &(materials[nMtl].Ks[0]),
        &(materials[nMtl].Ks[1]),
        &(materials[nMtl].Ks[2]));
}

else if (type.compare("Ni") == 0)
```

```
{
    sscanf(line.c_str(), "Ni %f", &(materials[nMtl].Ni));
}

else if (type.compare("d ") == 0)
{
    sscanf(line.c_str(), "d %f", &(materials[nMtl].d));
}

else if (type.compare("il") == 0)
{
    sscanf(line.c_str(), "illum %d", &(materials[nMtl].illum));
}

//  Read texture mapping
else if (type.compare("ma") == 0)
{
    materials[nMtl].hasTexture = true;
    string l = "map_Kd ";
    string textureName = line.substr(l.size());
    materials[nMtl].texture = textureName;

    map<string, texture>::iterator it = textures->find(textureName);
    if (it == textures->end())
    {
        texture tex;
        string texturePath = "resource/" + textureName;
        tex.image = SOIL_load_image(texturePath.c_str(),
                                    &(tex.width), &(tex.height),
                                    NULL,
                                    0);
        textures->insert(it, pair<string, texture>(textureName, tex));
    }
}
```

2.3. Model rendering

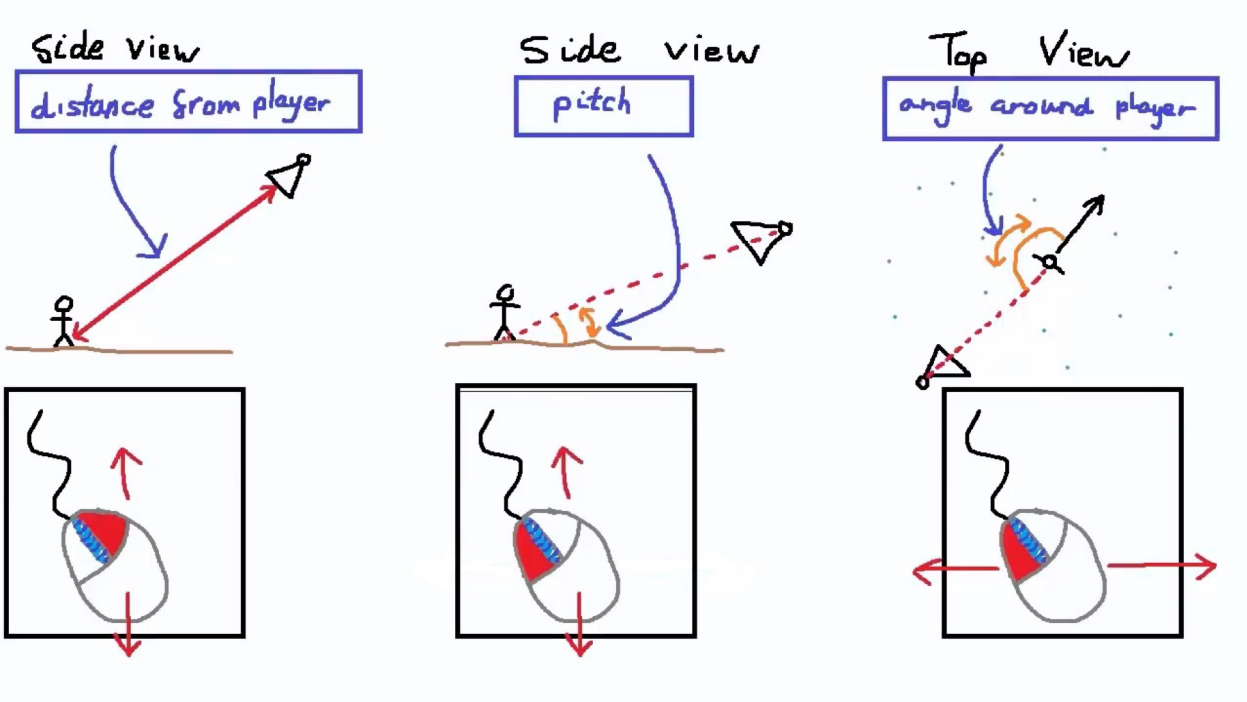
After reading models, we can render models into scene by OpenGL.
Within Model_OBJ class, we also write a piece of code for rendering model in draw function.

```
void Model_OBJ::draw()
{
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_NORMAL_ARRAY);
    glEnableClientState (GL_TEXTURE_COORD_ARRAY);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);

    if (name.compare("resource/Room.obj") == 0)
    {
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    }
    else
    {
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    }

    for (int i = 0; i < total_objects; i++)
    {
        material* mtl = objects[i].material;

        if (mtl->hasTexture)
        {
            glTexImage2D(GL_TEXTURE_2D,
                         0,
                         GL_RGB,
                         (textures->find(mtl->texture)->second).width,
                         (textures->find(mtl->texture)->second).height,
                         0,
                         GL_RGB, GL_UNSIGNED_BYTE,
                         (textures->find(mtl->texture)->second).image);

            glEnable(GL_TEXTURE_2D);
        }
```

```
        else
        {
            glDisable(GL_TEXTURE_2D);
        }

        float ambient[] = {mtl->Ka[0], mtl->Ka[1], mtl->Ka[2], 1.0};
        float diffuse[] = {mtl->Kd[0], mtl->Kd[1], mtl->Kd[2], 1.0};
        float specular[] = {mtl->Ks[0], mtl->Ks[1], mtl->Ks[2], 1.0};
        float shininess = mtl->Ns;

        glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);
        glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse);
        glMaterialfv(GL_FRONT, GL_SPECULAR, specular);
        glMaterialf(GL_FRONT, GL_SHININESS, shininess);

        glVertexPointer(3, GL_FLOAT, 0, objects[i].faces_triangles);
        glTexCoordPointer(2, GL_FLOAT, 0, objects[i].texts_coords);
        glNormalPointer(GL_FLOAT, 0, objects[i].norm_vectors);

        glDrawArrays(GL_TRIANGLES,
                     0,
                     objects[i].total_triangles_floats);
    }

    glDisableClientState(GL_VERTEX_ARRAY);
    glDisableClientState(GL_NORMAL_ARRAY);
    glDisableClientState (GL_TEXTURE_COORD_ARRAY);
}
```

2.4. Camera transformation

We implement a pure 3rd person camera which the white ball is the main character.



Calculate the distance from the white ball based on mouse motion and mouse clicking button:

```
void calculateZoom(float command)
{
    distanceFromObject += command * 0.04f;
}
```

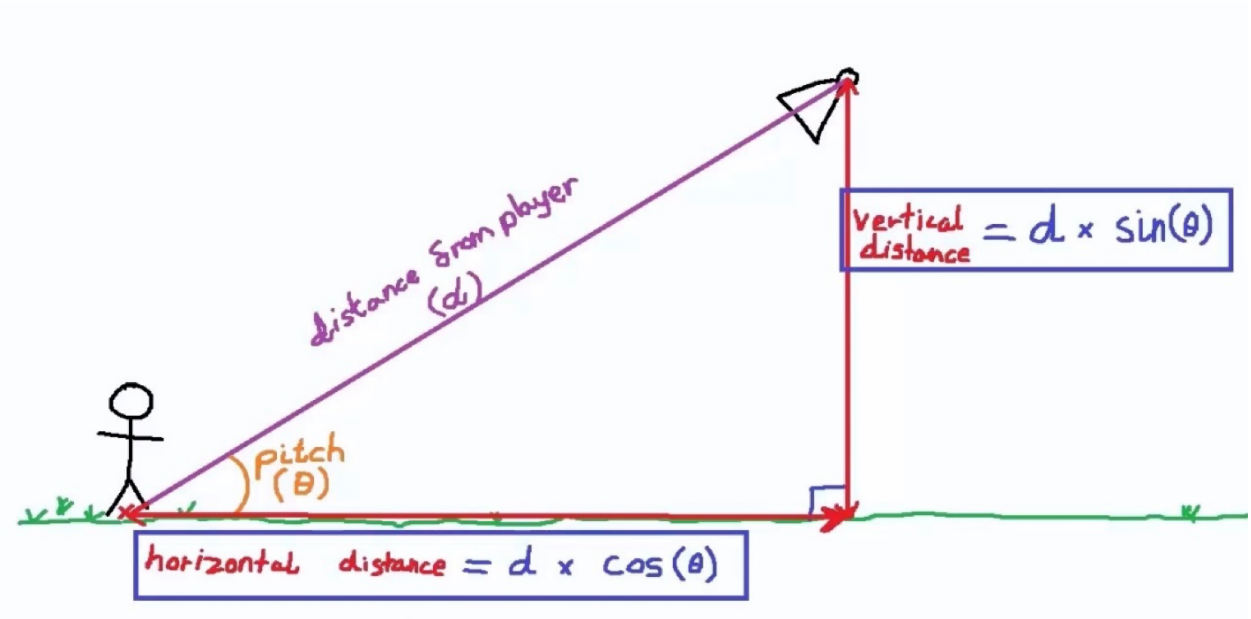
Calculate the Pitch angle based on mouse motion and mouse clicking button:

```
void calculatePitch(float MouseDY)
{
    originalPitch = pitch;
    pitch -= MouseDY * 0.08f;
    if (!(abs(pitch)>1 && abs(pitch)<40) | pitch>=0)
    {
        pitch = originalPitch;
    }
}
```

Calculate the angle around the white ball based on mouse motion and mouse clicking button:

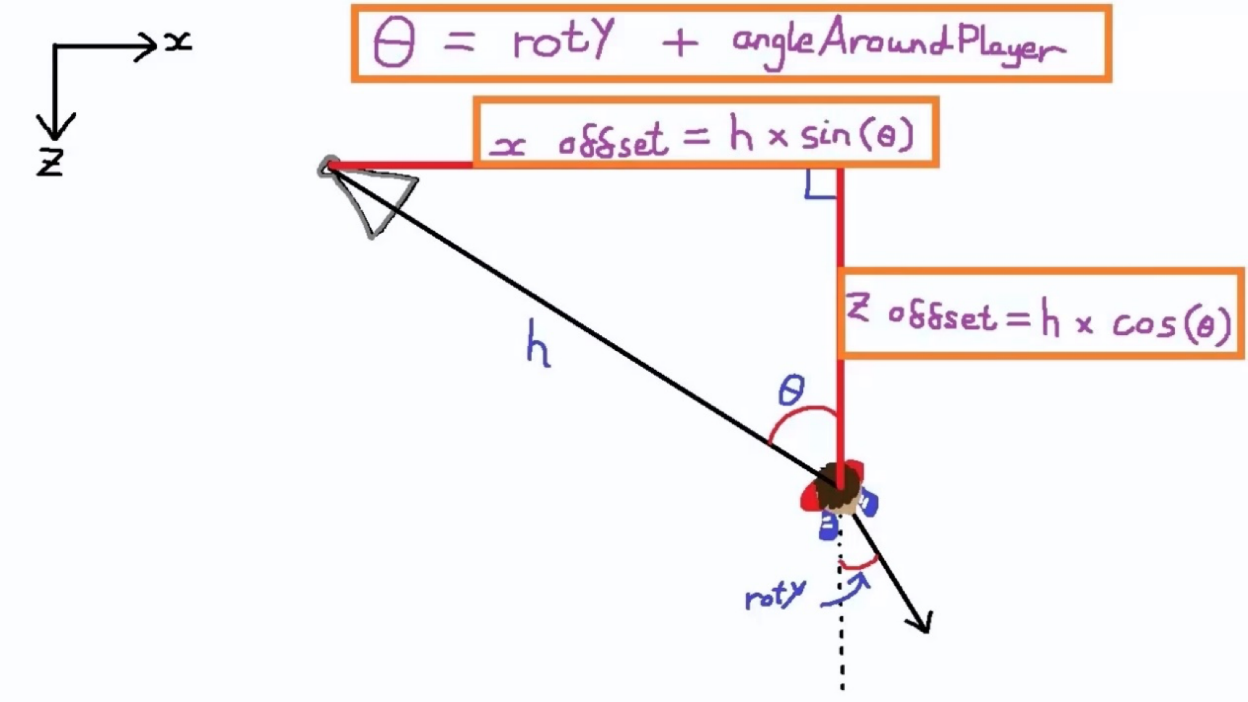
```
void calculateAngleAroundObject(float MouseDX)
{
    float angleChange = MouseDX * 0.08f;
    angleAroundObject -= angleChange;
}
```

Calculate the camera horizontal distance and vertical distance based on distance from the white ball and the pitch angle:




```
float calculateHorizontalDistance()
{
    return distanceFromObject*cos(pitch*PI/180);
}

float calculateVerticalDistance()
{
    return distanceFromObject*sin(pitch*PI/180);
}
```

Put it all together, we have the function to calculate camera position:




```
void calculateCameraPosition(float horizDistance, float verticDistance)
{
    float theta = balls[0]->angle + angleAroundObject;
    float offsetX = horizDistance * sin(theta*PI/180);
    float offsetZ = horizDistance * cos(theta*PI/180);
    position.x = balls[0]->pos[0] - offsetX;
    position.z = balls[0]->pos[1] - offsetZ;
    position.y = balls[0]->pos[1] - verticDistance;
}
```

Reference: https://www.youtube.com/watch?v=PoxDDZmctnU

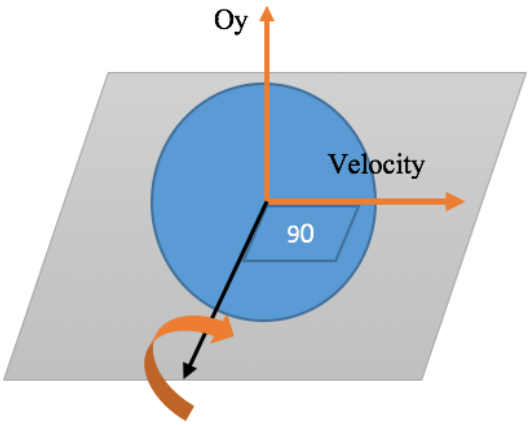
2.5. Ball movement

2.5.1. Ball rolling effect

a. Rolling angle

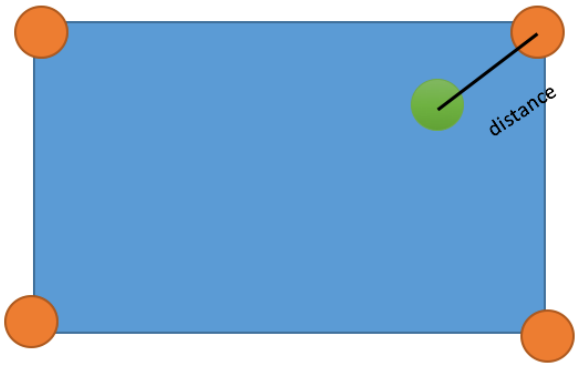
$$angle = \frac{AB * 360}{radius * 2 * \pi} = \frac{AB * 180}{radius * \pi}$$

b. Rolling vector



Balls are rolling in the surface of the table and the rolling vector is perpendicular with the velocity vector.

2.5.2. Ball in holes

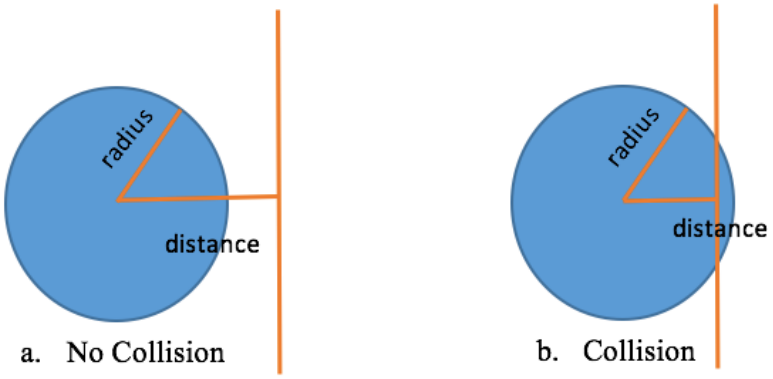


Holes in this game were simulated as a semi –spheres with a pre-specified radius.
To check whether a ball is in a hole, we calculated the distance between the ball and the hole. If the distance is less than the sum of ball radius and hole radius, the ball is in the hole.

2.5.3. Ball - Table collisions

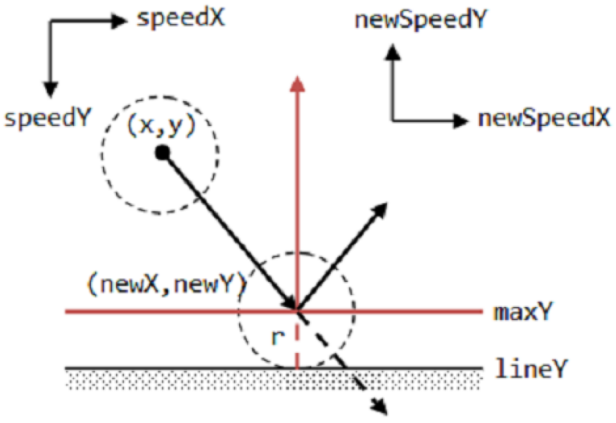
a. Collision detection

We calculate the distance between the ball and four borders (top, bottom, left, right) of the table. If the distance the less than the radius of the ball, the ball-table collision occur.

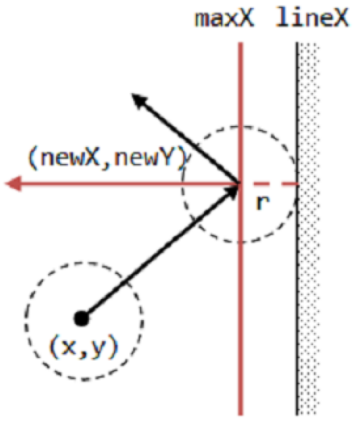


b. Reflection

Reflection happens when a pool ball hit the side edge cushions. Reflection ideally doesn't change the magnitude of velocity, but only reverses one of its velocity component. For example, if a ball hit the top cushion as the figure on the right shows, the X component of velocity remains and reverse the Y component. The resulting velocity is the velocity when pool ball bounces back. So, for the reflection happens on side edges, flip the x component; for reflection happens on top or bottom edges, flip the z component.




```
maxY = lineY - r
t = (maxY - y)
newY = maxY
newX = x + speedX * t
newSpeedX = speedX
newSpeedY = -speedY
```




```
maxX = lineX - r
t = (maxX - x)
newX = maxX
newY = y + speedY * t
newSpeedY = speedY
newSpeedX = -speedX
```

2.5.4. Ball - Ball collisions

a. Collision detection

To determine if two balls are colliding, we take the sum of the radiuses and compare it with the length from the centers of the spheres. If the length is smaller than the sum of the radiuses, we have a collision.

Difference vector (the length is the distance between those two balls):

$$\vec{d} = ball1.pos - ball2.pos$$

Then the length is computed:

$$distance = \vec{d}.length = \sqrt{\vec{d}.x^2 + \vec{d}.y^2 + \vec{d}.z^2}$$

Sum of the radiuses:

$$sumradius = ball1.radius + ball2.radius$$

If distance < sumradius ,we have a collision to take care of.

b. Collision response

If a collision is detected, I must recalculate the velocity vectors of the balls involved. This stage of the simulation is the most complex, and requires concepts such as *conservation of energy, conservation of linear momentum.*

For collisions between balls, the first step is to calculate the normal to the collision plane. This is along the line between the centers of the two balls, so the easiest way to calculate it is to normalize the difference between the position vectors of the two balls.

$$\vec{n} = \frac{\vec{r}_1 - \vec{r}_2}{\left|\vec{r}_1 - \vec{r}_2\right|}$$

The next step is to divide each velocity vector into a normal component and a tangential component. The normal component for ball 2 will be in the direction of the normal vector, and the normal component for ball 1 will be in the direction opposite the

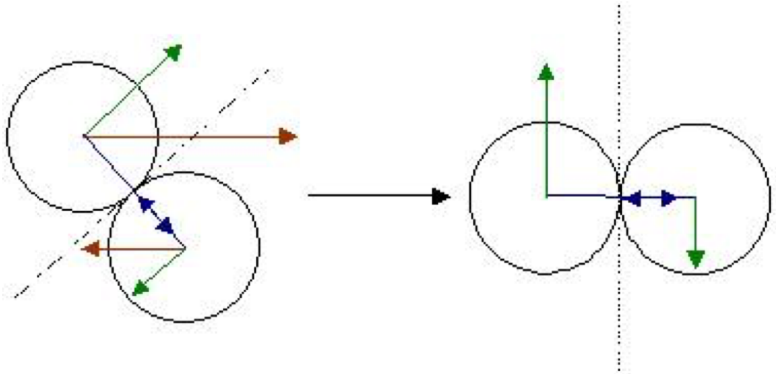
normal vector. The magnitude of these normal vectors can be calculated using the dot product, as shown below.

$$\vec{v}_{n_1} = \left[\vec{v}_1 \cdot \left(-\vec{n}\right)\right]\left(-\vec{n}\right)$$
$$\vec{v}_{n_2} = \left[\vec{v}_2 \cdot \vec{n}\right]\vec{n}$$

Once I've calculated the normal components of their velocity vectors, I can calculate the tangential components using vector subtraction:

$$\vec{v}_{t_1} = \vec{v}_{n_1} - \vec{v}_1$$
$$\vec{v}_{t_2} = \vec{v}_{n_2} - \vec{v}_2$$

During the collision, the tangential velocity components do not change. Using the normal components, I can treat the collision as 1-dimensional. This is depicted in the example below. The red arrows represent the velocity vectors immediately before the collision. The blue arrows represent the normal components of those vectors and the green arrows represent the tangential components of those vectors. The black dotted line is the collision plane.



To deal with the one-dimensional collision, I once again use *Newton's Second Law* in yet a different form, derived below:

$$\vec{F} = m\vec{a} = \frac{m\Delta\vec{v}}{\Delta t}$$
$$\vec{F} = \frac{\Delta\vec{p}}{\Delta t}$$

The quantity *mv* is the linear momentum of the ball, denoted by the letter p. Since there is no net linear force on the two balls during the collision, Δp is 0. This is the concept known as *conservation of linear momentum*, which states that the total momentum of a system will remain constant in the absence of an outside force. From this, I obtain the following equation:

$$\vec{p}_i = \vec{p}_f$$
$$m_1\vec{v}_1 + m_2\vec{v}_2 = m_1\vec{v}_1{}' + m_2\vec{v}_2{}'$$
$$\vec{v}_1 + \vec{v}_2 = \vec{v}_1{}' + \vec{v}_2{}'$$

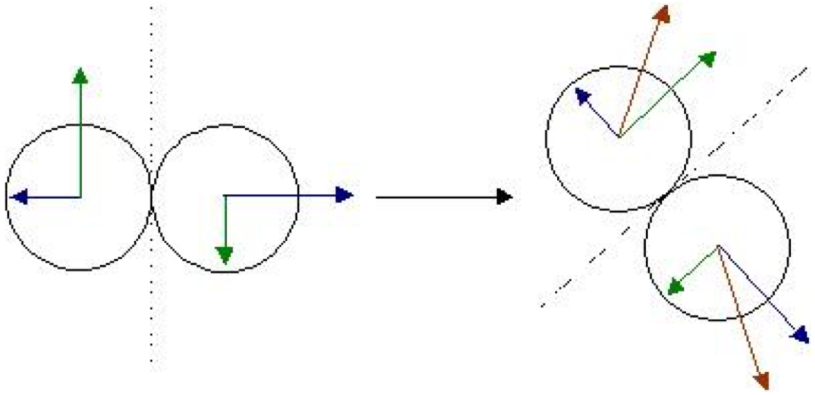
Since the masses of all the balls are identical, we can cancel out the mass terms, yielding the equation above. Using the conservation of kinetic energy, we can solve for v1' and v2'.

$$\tfrac{1}{2}m_1\vec{v}_1{}^2 + \tfrac{1}{2}m_2\vec{v}_2{}^2 = \tfrac{1}{2}m_1\vec{v}_1{}'^2 + \tfrac{1}{2}m_2\vec{v}_2{}'^2$$
$$\vec{v}_1{}^2 + \vec{v}_2{}^2 = (\vec{v}_1 + \vec{v}_2 - \vec{v}_2{}')^2 + \vec{v}_2{}'^2$$
$$\vec{v}_1{}^2 + \vec{v}_2{}^2 = \vec{v}_1{}^2 + \vec{v}_2{}^2 + \vec{v}_2{}'^2 + 2\vec{v}_1\vec{v}_2 - 2\vec{v}_1\vec{v}_2{}' - 2\vec{v}_2\vec{v}_2{}' + \vec{v}_2{}'^2$$
$$\vec{v}_2{}'^2 - \vec{v}_1\vec{v}_2{}' - \vec{v}_2\vec{v}_2{}' + \vec{v}_1\vec{v}_2 = 0$$
$$(\vec{v}_2{}' - \vec{v}_1)(\vec{v}_2{}' - \vec{v}_2) = 0$$

This equation yields two possible solutions, only one of which is practical. Therefore, assuming total elasticity in the balls,

$$\vec{v}_2{}' = \vec{v}_1, \text{ and } \vec{v}_1{}' = \vec{v}_2$$

Now I return to the 2-dimensional collision problem above. Since the normal velocity vectors can simply be exchanged, the velocity of each ball immediately after the collision can be obtained by adding the tangential velocity to the normal velocity of the ball it collides with. This is shown below (the red arrows now represent the velocity vectors immediately after the collision).



The red vectors in the diagram on the right can be calculated using the equations below:

$$\vec{v}_1' = \vec{v}_{t_1} + \vec{v}_{x_2}$$

$$\vec{v}_2' = \vec{v}_{t_2} + \vec{v}_{x_1}$$

Reference: http://archive.ncsa.illinois.edu/Classes/MATH198/townsend/math.html

3. Product

Github: https://github.com/tuantq57/3D-Billiards-Game-OpenGL

Recommended system:

CPU	Intel Core i7 3610 @ 2.1GHZ (4 CPUs) / AMD X8 FX-8350 @ 4GHZ (8 CPUs)
RAM	8 GB
OS	Windows 8.1 64 Bit, Windows 8 64 Bit, Windows 7 64 Bit Service Pack 1
Video Card	NVIDIA GeForce GT 630M 1GB / AMD Radeon HD 7870 2GB
Free Disk Space	1 GB - Intel 520 Series Solid-State Drive

Screenshot:

4. Contributions

Trương Quốc Tuấn:
- Model loading
- Model rendering

Nguyễn Thạc Thống:
- Ball movement
- Keyboard event handling

Lê Văn Giáp:
- Model making (using blender)
- Camera transformation
- Mouse event handling