2014

Billiards Motion Emulation

Đinh Trung Anh

Đặng Minh Dũng

Võ Anh Hưng

Ngô Minh Sơn

K56CA - UET, VNU

5/21/2014

Table of Contents

1. Introduction 2
1.1. Problem 2
1.2. Requirements 2
1.3. System Requirements 2
1.4. Third-party libraries 2
2. Implementation 3
2.1. Project Structure 3
2.2. OpenGL 4
2.2.a. Matrices 4
2.2.b. Affine transformation 5
2.2.c. Camera Transform 6
2.2.d. Loading Model 6
2.2.e. Lighting 9
2.3. Ball Movement 10
2.3.a. Velocity 10
2.3.b. Rolling effect 10
2.3.c. Balls collision 11
3. Demo Product 14
4. Tasks division 15

1. Introduction

1.1. Problem

Billiard is a familiar game. It offend exists at the big game all over the world. There are a lot of people play billiard excellent. In this project, we aim to simulate simple pool table using OpenGL.

1.2. Requirements

- Load Billiards table and balls models
- Move camera 360 ° around table
- Shot the ball and Simulate the collision (ball – ball, ball - table)
- Some simple feather:
    - Texture
    - Lighting

1.3. System Requirements

- Programming Language: C++11
- OpenGL 3.3 or later
- GLSL 3.3 or later
- Supporting: The OpenGL Extension Wrangler Library (GLEW)

1.4. Third-party libraries

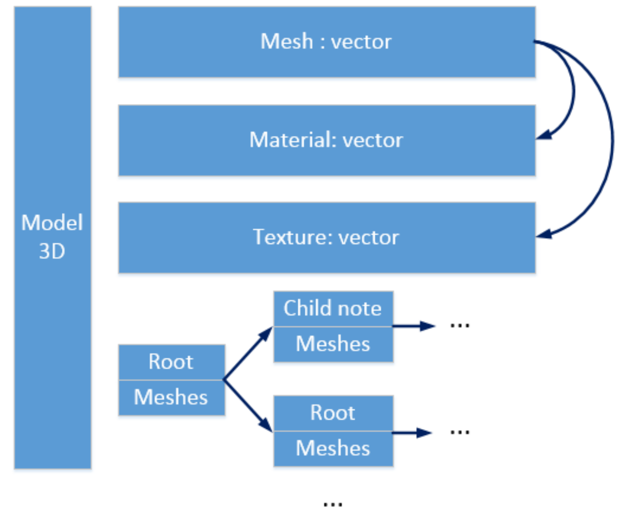
- glm (OpenGL Mathematics): Matrix and Vector library
- DivIL (Developer's Image Library): Loading Texture library
- Assimp (Open Asset Import Library): Loading Model library

Note: Out project and all 3rd-party libraries are portable and suitable for Visual Studio 2013. Any other IDE or other version of Visual Studio must be re-config before compile it.

2. Implementation

2.1. Project Structure

Figure beside illustrate our model3D structure:



1. Model3D: manage all mesh, material, and texture.

Each mesh also have pointer to its materials and textures.

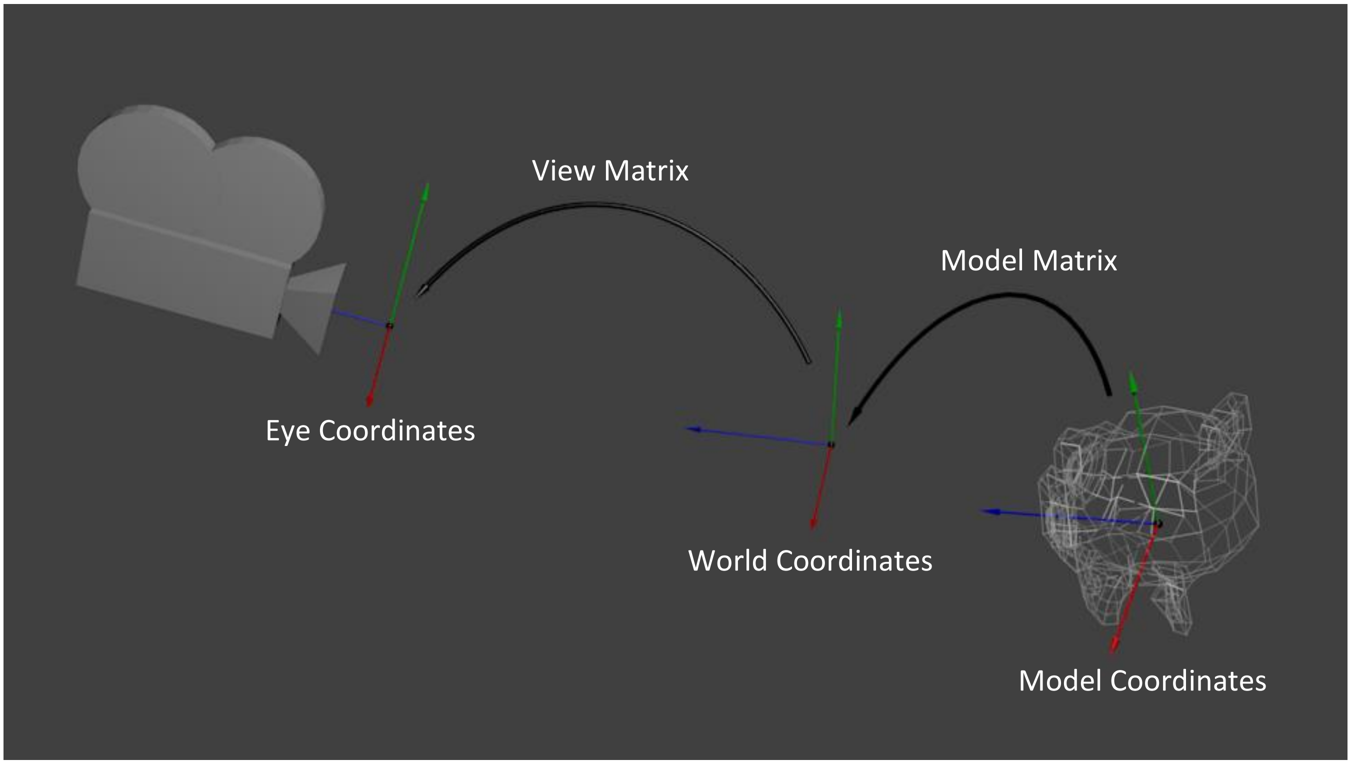
Models have nodes structure it as the tree.

2. Camera.
3. Lighting.
4. Resource manager:
   Read "Resource.txt" file (file contain model path, camera, light and its initial state)
   Loading all resource to memory.
5. Scene manager:
   Request resource from Resource manager
   Combine resources to game scene, handling input (keyboard & mouse) and render to screen.

2.2. OpenGL

2.2.a. Matrices

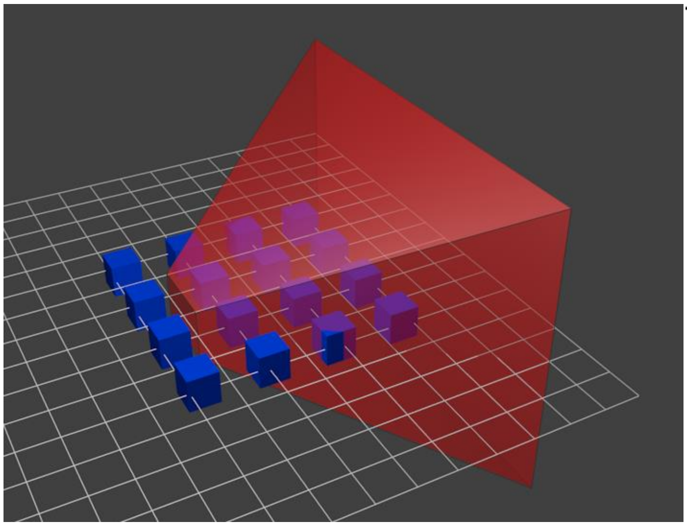
In older openGL, there are 2 transformation matrices: Model-View matrix and Projection matrix. However, openGL recommend make Model-View matrix seperated:

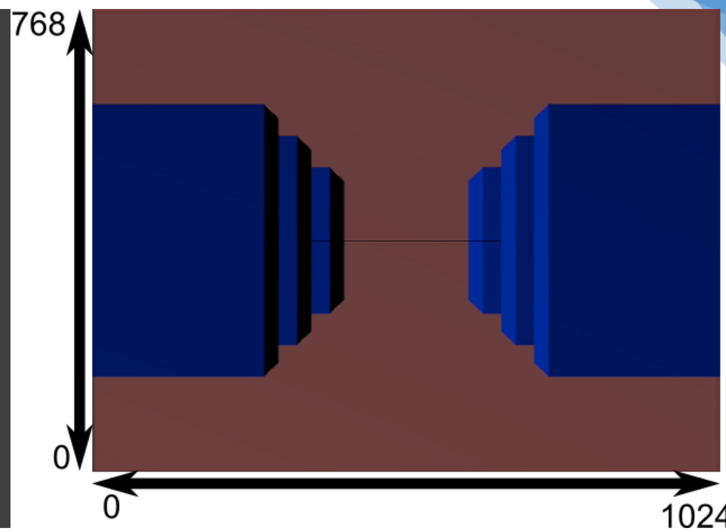


- Model matrix (M): Transform vertex position from Model coordinates to World coordiantes. Each model have at least 1 model-matrix. (Some model have more than 1 nodes must havve more model matrix and model matrix of child node transform form child coordiantes to parent node coordiantes)
- View matrix (V): Transform vertex position form World coordiantes to Eye coordiantes.
- Projection matrix (P): Transform vertex position form eye coordiante to Homogeneous coordiantes
  Each camera have 1 view matrix and 1 projection matrix

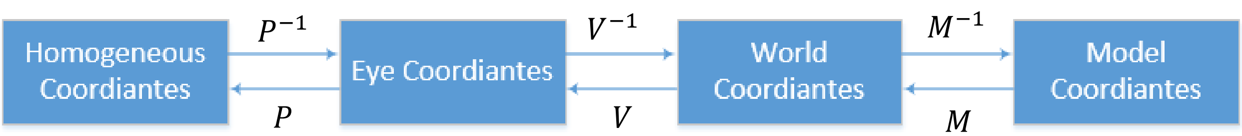
Other matrix:

- Normal matrix (N): Transform normal vector form model coordiante to world coordiante
  $$N = transpose\left[inverse\left(mat3\text{x}3(M)\right)\right]$$
  - N : normal matrix (size of 3×3)
  - M : model matrix (size of 4×4)
  - $mat3\text{x}3(M)$: upper left matrix 3x3 of M





Spaces transforms:



2.2.b. Affine transformation

All most matrix and vector operation, glm has been supported, so what we only care is which coordiantes, object transforms. for example:

In the figure beside, you can se the difference of rotation in local space and global space

Call

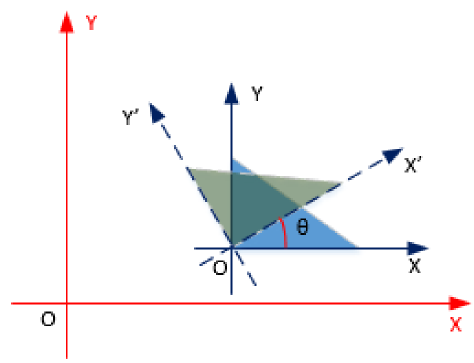
- H is transformation matrix.
- $R(\theta)$ is rotation matrix an angle θ

⇨ Rotate in local space:

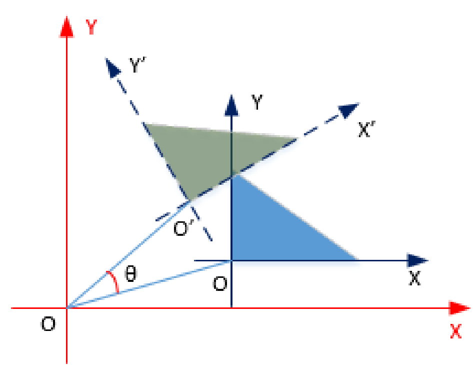
$$H = H * R(\theta)$$

⇨ Rotate in global space:

$$H = R(\theta) * H$$

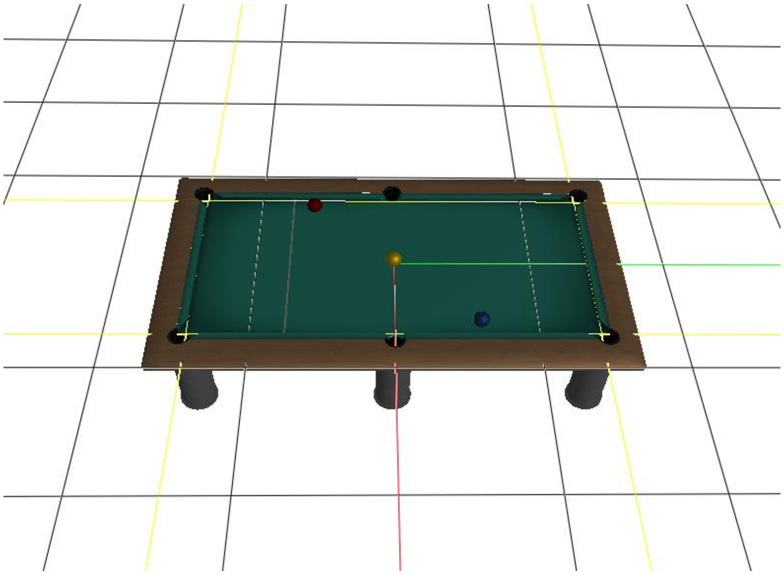


Rotate in Local Space



Rotate in Global Space

2.2.c. Camera Transform



- Use arrow key (Up, Right, Left, Down) and Ctrl + Up, Ctril + Right to translate camera (In View Space)
- Drag left mouse to rotate mouse around origin of World Space.

```
static const float coef = 90.0f;
glm::ivec2 delta = dragStartPos - curPos;

glm::mat3 viewInverse = glm::inverse(glm::mat3(cam->getViewMatrix()));
glm::vec3 vAxis = glm::normalize(viewInverse * yAxis);
cam->rotate(coef * delta.x / float(windowWidth), vAxis, WORLD_COORDINATES);

viewInverse = glm::inverse(glm::mat3(cam->getViewMatrix()));
glm::vec3 uAxis = glm::normalize(viewInverse * xAxis);
cam->rotate(coef * delta.y / float(windowHeight), uAxis, WORLD_COORDINATES);
```

2.2.d. Loading Model

Assimp (Open Asset Import Library) is a portable Open Source library to import various well-known 3D model formats in a uniform manner.

2.2.d.i. Code for loading model:

```
// model3D.cpp
static Assimp::Importer imp;
const aiScene* scene = imp.ReadFile(path, aiProcessPreset_TargetRealtime_Quality);
if (scene == NULL){
    fprintf(stderr, "Error : '%s'\n", imp.GetErrorString());
    exit(EXIT_FAILURE);
}
```

after that, for each mesh we:

2.2.d.ii. Bind all vertices to Vertex Array Object (VAO)

```
//mesh.cpp
glGenVertexArrays(1, &vao);
```

```
glBindVertexArray(vao);

// buffer for faces
unsigned int *faceArray;
this->numIndices = mesh->mNumFaces * 3;
faceArray = new unsigned int[this->numIndices];
for (unsigned int i = 0; i < mesh->mNumFaces; i++) {
    const aiFace* face = &mesh->mFaces[i];
    assert(face->mNumIndices == 3); //If mNumIndices != 3, program rase error
    memcpy(&faceArray[3 * i], face->mIndices, 3 * sizeof(unsigned int));
}
glGenBuffers(1, &buffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(unsigned int)* mesh->mNumFaces * 3, faceArray, GL_STATIC_DRAW);

// buffer for vertex positions
if (mesh->HasPositions()) {
    glGenBuffers(1, &buffer);
    glBindBuffer(GL_ARRAY_BUFFER, buffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(aiVector3D)* mesh->mNumVertices, mesh->mVertices, GL_STATIC_DRAW);
    glEnableVertexAttribArray(aPositionLoc);
    glVertexAttribPointer(aPositionLoc, 3, GL_FLOAT, GL_FALSE, 0, 0);
}

// buffer for vertex normals
if (mesh->HasNormals()) {
    glGenBuffers(1, &buffer);
    glBindBuffer(GL_ARRAY_BUFFER, buffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(aiVector3D)* mesh->mNumVertices, mesh->mNormals, GL_STATIC_DRAW);
    glEnableVertexAttribArray(aNormalLoc);
    glVertexAttribPointer(aNormalLoc, 3, GL_FLOAT, GL_FALSE, 0, 0);
}

// buffer for vertex texture coordinates
// Now out project only support single texture.
if (mesh->HasTextureCoords(0)) {
    glGenBuffers(1, &buffer);
    glBindBuffer(GL_ARRAY_BUFFER, buffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(aiVector3D)* mesh->mNumVertices, mesh->mTextureCoords[0], GL_STATIC_DRAW);
    glEnableVertexAttribArray(aTexCoordLoc);
    glVertexAttribPointer(aTexCoordLoc, 2, GL_FLOAT, GL_FALSE, sizeof(aiVector3D), 0);
}

// unbind buffers
glBindVertexArray(NULL);
glBindBuffer(GL_ARRAY_BUFFER, NULL);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, NULL);
```

2.2.d.iii. Reading Material:

```
//material.cpp
aiColor4D color;

if (material->Get(AI_MATKEY_COLOR_AMBIENT, color) == AI_SUCCESS){
    uMaterial->ambient = glm::vec4(color.r, color.g, color.b, color.a);
}

if (material->Get(AI_MATKEY_COLOR_DIFFUSE, color) == AI_SUCCESS){
    uMaterial->diffuse = glm::vec4(color.r, color.g, color.b, color.a);
```

```
}

if (material->Get(AI_MATKEY_COLOR_SPECULAR, color) == AI_SUCCESS){
    uMaterial->specular = glm::vec4(color.r, color.g, color.b, color.a);
}

if (material->Get(AI_MATKEY_COLOR_EMISSIVE, color) == AI_SUCCESS){
    uMaterial->emissive = glm::vec4(color.r, color.g, color.b, color.a);
}

uMaterial->shininess = 0.0;
unsigned int max;
aiGetMaterialFloatArray(material, AI_MATKEY_SHININESS, &(uMaterial->shininess), &max);

glGenBuffers(1, &uboMaterial);
glBindBuffer(GL_UNIFORM_BUFFER, uboMaterial);
glBufferData(GL_UNIFORM_BUFFER, sizeof(LightMaterial), (void*) uMaterial, GL_STATIC_DRAW);

glBindBuffer(GL_UNIFORM_BUFFER, NULL);
```

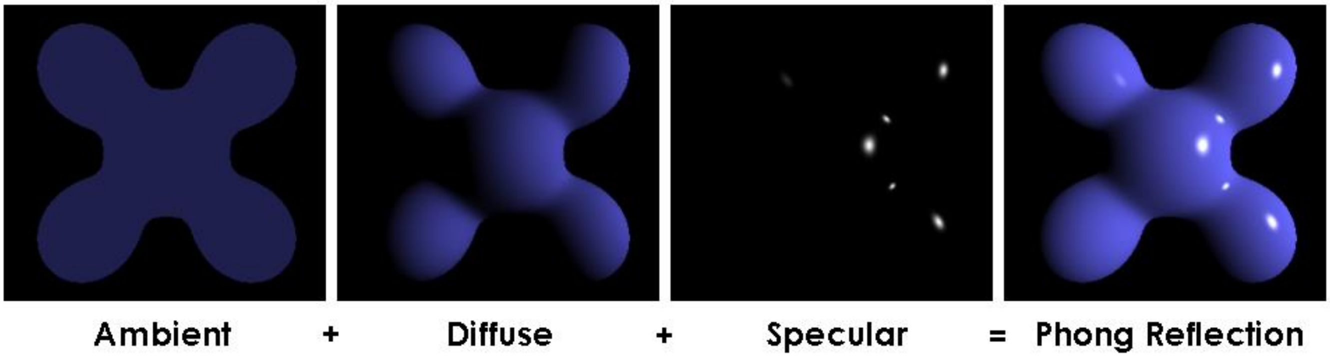
2.2.d.iv. Loading Texture:

We use DevIL libarary to loading Texture:

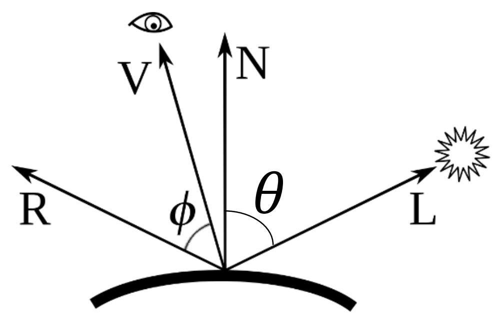
```
// texture.cpp
ilInit();
ilEnable(IL_ORIGIN_SET);
ilOriginFunc(IL_ORIGIN_LOWER_LEFT);

unsigned int imageID;
ilGenImages(1, &imageID);
ilBindImage(imageID);
if (!ilLoadImage(path.data())){
    ilDeleteImages(1, &imageID);
    fprintf(stderr, "Error while read texture.\n");
    exit(EXIT_FAILURE);
}

if (!ilConvertImage(IL_RGBA, IL_UNSIGNED_BYTE)){
    ilDeleteImages(1, &imageID);
    fprintf(stderr, "Error while convert texture");
    exit(EXIT_FAILURE);
}

glGenTextures(1, &texUnit);
glBindTexture(GL_TEXTURE_2D, texUnit);
glTexImage2D(GL_TEXTURE_2D, 0,
    ilGetInteger(IL_IMAGE_FORMAT),
    ilGetInteger(IL_IMAGE_WIDTH),
    ilGetInteger(IL_IMAGE_HEIGHT),
    0, GL_RGBA, GL_UNSIGNED_BYTE,
    ilGetData());

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glBindTexture(GL_TEXTURE_2D, NULL);

ilDeleteImage(imageID);
```

2.2.e. Lighting

Following Phong illumination model:



Formula: $I = I_a + I_p + I_s$



- $I_a = k_a * i_a$
- $I_p = k_d * i_d * \cos(\theta)$
- $I_s = k_s * i_s * \cos(\phi)^{\alpha}$

Anh here is GLSL (Fragment Shader) code for this one:

```
// basic_light.frag
vec3 lightDir = normalize(light.position - Position);
vec3 viewDir = normalize(viewPos - Position);
vec3 normalDir = normalize(Normal);

//Ambient
vec4 La = light.ambientIntensity * material.ambient;

//Diffuse reflection - Lambertian illumination model
float diffuseCoef = max(dot(normalDir, lightDir), 0.0);
vec4 Ld = light.diffuseIntensity * material.diffuse * diffuseCoef;

//Specular reflection. - Phong illumination model
vec3 reflectDir = normalize(reflect(-lightDir, normalDir));
float specularCoef = max(dot(reflectDir, viewDir), 0.0);
specularCoef = pow(specularCoef, material.shininess);
vec4 Ls = light.specularIntensity * material.specular * specularCoef;

//attenuation
float distanceToLight = length(light.position - Position);
float attenuation = light.attenuationConstant;
attenuation += light.attenuationLinear * distanceToLight;
attenuation += light.attenuationQuadratic * pow(distanceToLight, 2);
attenuation = 1.0f / attenuation;

//vec4 linearColor = material.emissive + attenuation * (La + Ld + Ls);
vec4 linearColor = La + Ld + Ls;
vec4 texColor = vec4(1.0, 1.0, 1.0, 1.0);
if (texCount != 0){
    texColor = texture(uSampler, TexCoord);
}

//after gamma correction
vec4 gamma = vec4(1.0f / 2.2f);
gl_FragColor = pow(linearColor, gamma) * texColor;
```

2.3. Ball Movement

2.3.a. Velocity

Poll ball movement can be considered as a regular 2D motion with constant acceleration or deceleration. Since the friction affects to pool balls are "rolling friction", the friction is not constant. In my program, I simply subtract 1/10 of the velocity, and set the velocity to 0 when the magnitude of velocity is very close to 0.

The acceleration, velocity, and time relationship for 2D motion is:

$$velocity(t) = velocity(0) + acceleration \times time$$

And based on my ideas: $velocity(n) = velocity(n-1) * 0.95$ for every 10ms

2.3.b. Rolling effect

2.3.b.i. Rolling Angle

Supposing that, ball rolling from A to B. So the rolling angle is:

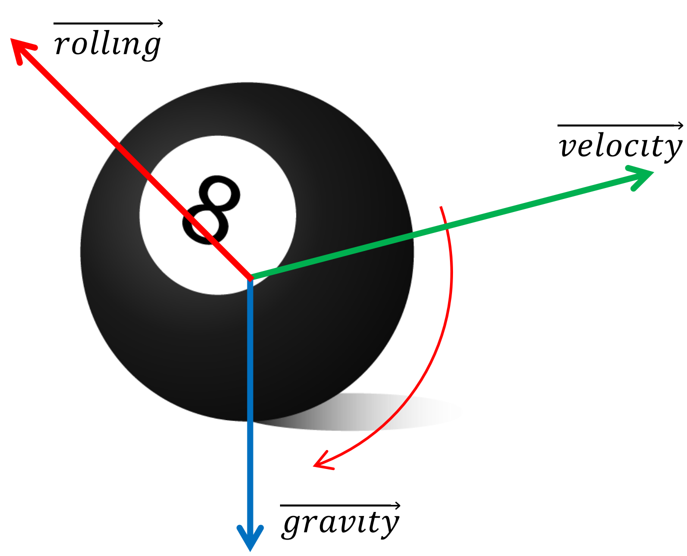
$$angle = \frac{AB \times 2\pi}{perimeter} = \frac{AB}{radius} \times \pi$$

2.3.b.ii. Rolling orientation

We have formula:

$\overrightarrow{rolling} = \overrightarrow{velocity} \times \overrightarrow{gravity}$

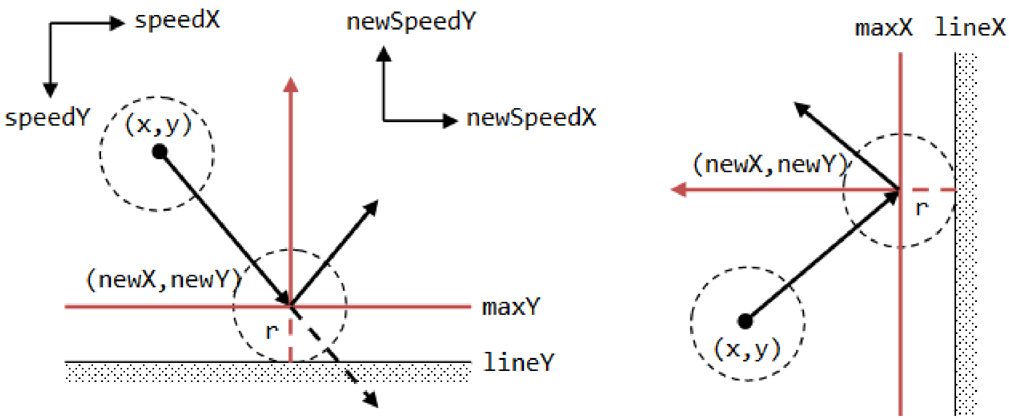
(It's cross product)



2.3.c. Balls collision

2.3.c.i. 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
```

2.3.c.ii. Ball Collision

- Time to collision:

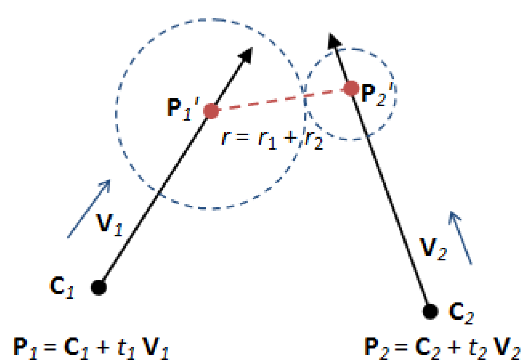
Collision occurs if the distance between the two balls is equal to the sum of their radiuses.

The parametric equations for the 2 moving balls are:

$$\begin{cases} P_1 = C_1 + t_1 V_1 \\ P_2 = C_2 + t_2 V_2 \end{cases}$$

Where :

- $t_1, t_2 \in (0,1)$: time-step
- C_1, C_2 : starting point
- V_1, V_2 : velocities of the 2 moving balls



Collision occurs at :

- $t = t_1 = t_2$
- $P_1' P_2' = r = r_1 + r_2$

Lets $C = C_2 - C_1$ & $V = V_2 - V_1$

$$|P_2' - P_1'|^2 = r^2$$

$$|(C_2 + tV_2) - (C_1 + tV_1)|^2 = r^2$$

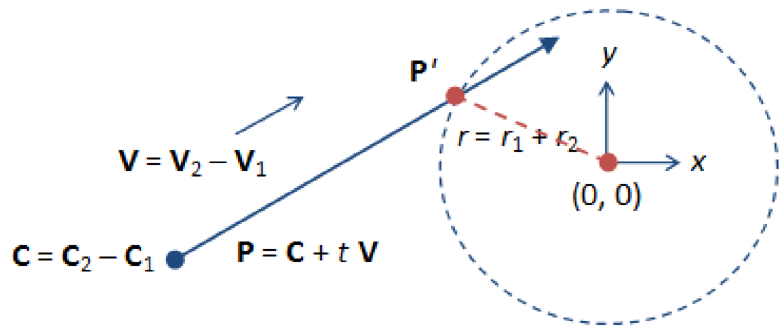
$$|(C_2 - C_1) + t(V_2 - V_1)|^2 = r^2$$

$$|C + tV|^2 = r^2$$

In other word, we can treat one ball as statinary at the origin, and move the other with the relative velocity and from the starting point, as illustrated:

2D, Let $C = \begin{bmatrix} C_x \\ C_y \end{bmatrix}$ and $V = \begin{bmatrix} V_x \\ V_y \end{bmatrix}$

$\Rightarrow \left\| \begin{bmatrix} C_x \\ C_y \end{bmatrix} + t \begin{bmatrix} V_x \\ V_y \end{bmatrix} \right\|^2 = r^2$

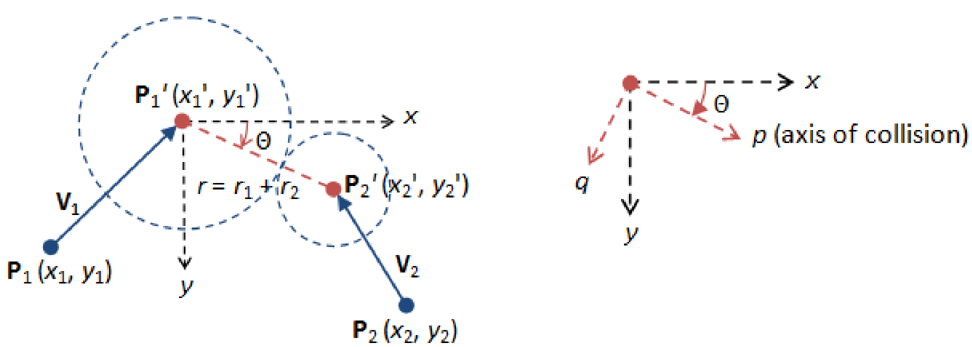


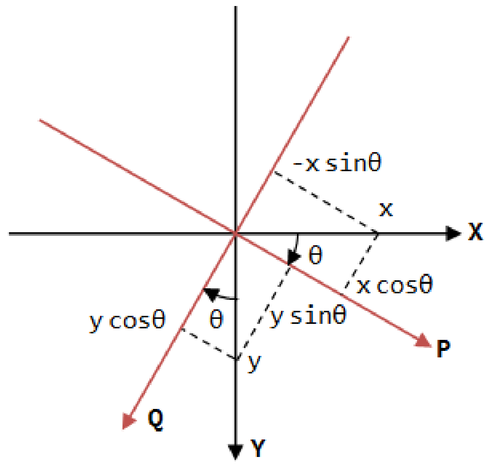
Solve the quadratic equation:

$$t = \frac{-(c_x v_x + c_y v_y) \pm \sqrt{r^2(v_x^2 + v_y^2) - (c_x v_y - c_y v_x)^2}}{v_x^2 + v_y^2}$$

- Collision Response:

We first dissolve the velocities (V_1 and V_2) along the axes of collision, p and q (as illustrated). We then apply the laws of conservation of momentum and energy to compute the velocities after collision, along the axis of collision p. The velocities perpendicular to the axis of collision q remains unchanged.





p = x cosθ + y sinθ
q = -x sinθ + y cosθ
y inverted, θ clockwise

Along axis of collision:

- Conservation of momentum:

$$m_1 v_1 + m_2 v_2 = m_3 v_3 + m_4 v_4$$

- Conservation of energy:

$$\frac{1}{2} m_1 v_1 + \frac{1}{2} m_2 v_2 = \frac{1}{2} m_3 v_3 + \frac{1}{2} m_4 v_4$$

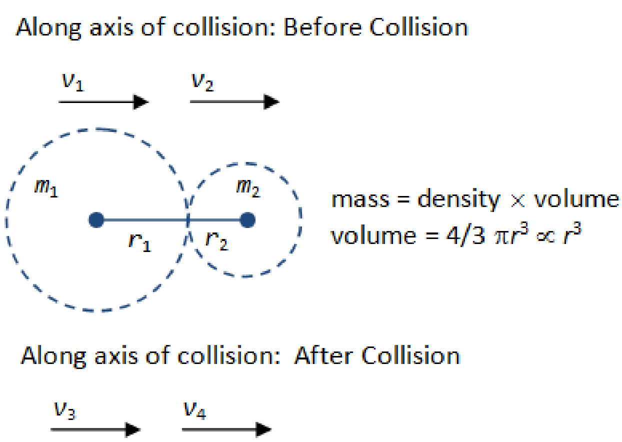
<u>Slove ablove equation, we have:</u>

- $v_4 =$ $[2m_1 v_1 + (m_2 - m_1) v_2]/(m_1 + m_2)$
- $v_3 =$ $[2m_2 v_2 + (m_1 - m_2) v_2]/(m_1 + m_2)$

In our problem, all balls have the same mass

So:

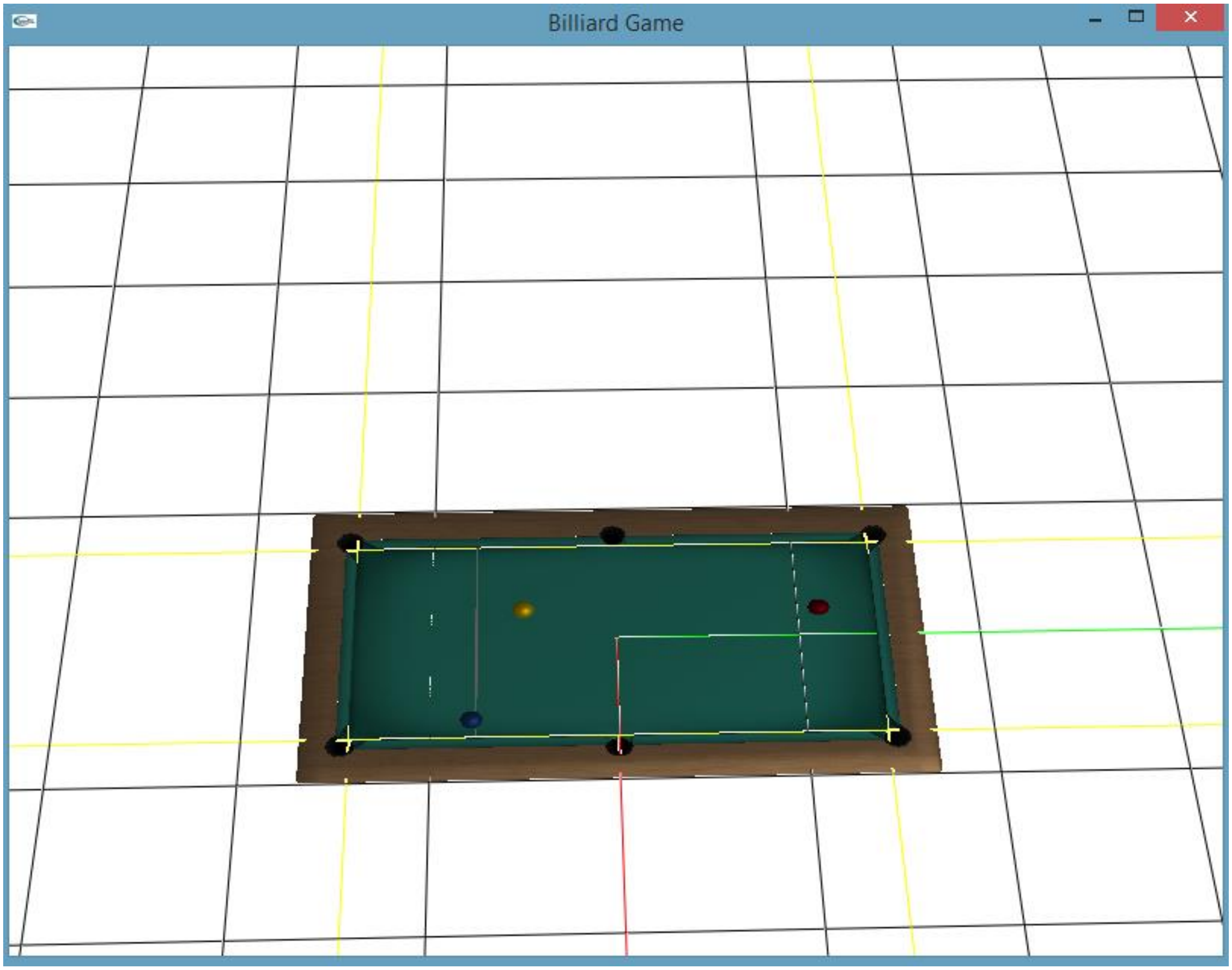
- $v_4 = v_1$
- $v_3 = v_2$



3. Demo Product

Github: https://github.com/anhhung1303/CG_BilliardsMotion

Some screenshots:



4. Tasks division

- Each member of team have individual task.
- We divide base on ability of each one
- Particular tasks:
  - Đinh Trung Anh:
    - Lighting effect
    - Implement physic simulation.
    - Handle mouse, keyboard events.
  - Đặng Minh Dũng:
    - Learn about library and reteach for other members.
    - Design structure of project
    - Code based project.
    - Load models, camera, shader and render
  - Võ Anh Hưng
    - Render models using 3ds max
    - Handle mouse, keyboard events
    - Documenation
  - Ngô Minh Sơn:
    - Physical simulation.
    - Keyboard event handler