OOP[Class02]

OBJECT ORIENTED PROGRAMMING 2024

Project #3
4 Balls Billiard Game

[Team 13]

2024.11.23 MON

- [redacted]
- [redacted]

Chapter.

01 Project Introduction

02 Rule of 4 balls

03 Additional Elements

04 Major Functions

05 Execution Video

06 Q&A

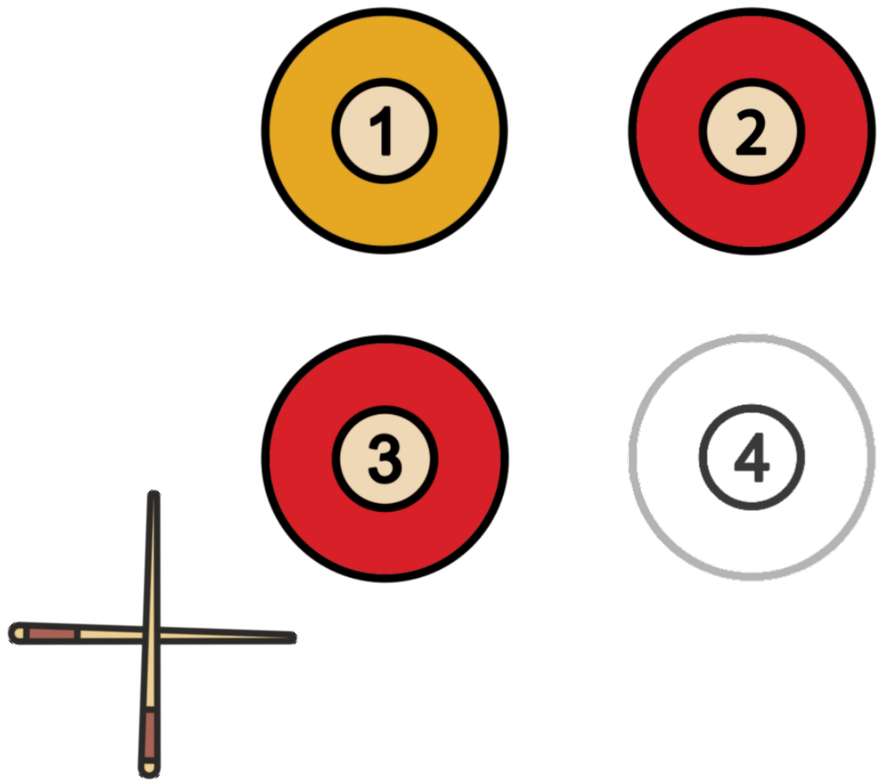
Project Introduction

Goal

This project is a simple 3D crash game simulation developed using DirectX 9. Four spheres and walls interact to simulate physical collisions, and the user controls a specific ball to proceed with the game.

- Collision of Spheres and Walls
- Sperical Control
- World Rotation

Rule of 4 Balls



[Basic Rule]

- With 4 balls (red 2, yellow 1, white 1).
- First person use a white ball, and another use a yellow ball.
- You have to hit the other two red balls with your own ball without touching other player's balls.

Additional Elements of Billards

1

Press the "Start" Button to start the game

2

Use white and yellow balls alternately

3

Press the "Backspace" key to fix the billiard board

1 Press the "Start" Button to start the game

```
bool isGameStarted = false;
ID3DXFont* startg_Font = nullptr;
```

```
bool Setup()
{
HRESULT startHr = D3DXCreateFont(Device, 48, 0, FW_BOLD, 1, FALSE, DEFAULT_CHARSET,
    OUT_DEFAULT_PRECIS, DEFAULT_QUALITY,
    DEFAULT_PITCH | FF_DONTCARE, "Arial", &startg_Font);
if (FAILED(startHr))
{
    ::MessageBox(0, "D3DXCreateFont() - FAILED", 0, 0);
    return false;
}

HRESULT infoHr = D3DXCreateFont(Device, 24, 0, FW_BOLD, 1, FALSE, DEFAULT_CHARSET,
    OUT_DEFAULT_PRECIS, DEFAULT_QUALITY,
    DEFAULT_PITCH | FF_DONTCARE, "Arial", &infog_Font);
if (FAILED(infoHr))
{
    ::MessageBox(0, "D3DXCreateFont() - FAILED", 0, 0);
    return false;
}
}
```

- isGameStarted indicates whether the game has started or not reset to false to show the start button.

- Create fonts using D3DXCreateFont()

1 Press the "Start" Button to start the game

```
RECT startButtonRect = { 300, 200, 700, 300 };
RECT infoRect = { startButtonRect.left, startButtonRect.top + 100, startButtonRect.right,
    startButtonRect.bottom + 100 };
```

```
bool Display(float timeDelta)
{
if (!isGameStarted)
{
    D3DRECT fillRect = { startButtonRect.left, startButtonRect.top, startButtonRect.right,
        startButtonRect.bottom };
    Device->Clear(1, &fillRect, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0, 0, 0), 1.0f, 0);

    startg_Font->DrawText(0, "START", -1, &startButtonRect, DT_CENTER | DT_BOTTOM |
        DT_SINGLELINE, d3d::WHITE);

    D3DRECT rect = { infoRect.left, infoRect.top, infoRect.right, infoRect.bottom };
    Device->Clear(1, &rect, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0, 0, 0), 1.0f, 0);
    infog_Font->DrawText(0, "press ESC to exit", -1, &infoRect, DT_CENTER | DT_TOP |
        DT_SINGLELINE, d3d::WHITE);
}
```

- A 'RECT' structure representing the position of buttons and notices.
- If the game has not started (isGameStarted == false), draw the "START" button and the "press ESC to exit" notice on the screen
- The button and notice are drawn in g_Font → DrawText

1 Press the "Start" Button to start the game

```
LRESULT CALLBACK d3d::WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
     if (!isGameStarted &&
         cursorPos.x >= startButtonRect.left && cursorPos.x <= startButtonRect.right &&
         cursorPos.y >= startButtonRect.top && cursorPos.y <= startButtonRect.bottom)
     {
         isGameStarted = true;
         worldMove = true;
     }
 }
 break;
```

- If mouse click occurs inside startButtonRect, set isGameStarted to true to start the game

2 Use white and yellow balls alternately

```
static bool isTargetYellow = false;
LRESULT CALLBACK d3d::WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    int n;

    if (isTargetYellow) {
        n = 2;
    }
    else n = 3;
    isTargetYellow = !isTargetYellow;
    D3DXVECTOR3 targetpos = g_target_blueball.getCenter();
    D3DXVECTOR3 ballpos = g_sphere[n].getCenter();
    double theta = acos(sqrt(pow(targetpos.x - ballpos.x, 2)) / sqrt(pow(targetpos.x -
        ballpos.x, 2) +
        pow(targetpos.z - ballpos.z, 2)));

    if (targetpos.z - ballpos.z <= 0 && targetpos.x - ballpos.x >= 0) { theta = -theta; }

    if (targetpos.z - ballpos.z >= 0 && targetpos.x - ballpos.x <= 0) { theta = PI -
        theta; }
    if (targetpos.z - ballpos.z <= 0 && targetpos.x - ballpos.x <= 0) { theta = PI +
        theta; }
    double distance = sqrt(pow(targetpos.x - ballpos.x, 2) + pow(targetpos.z - ballpos.z,
        2));
    g_sphere[n].setPower(distance * cos(theta), distance * sin(theta));

    break;
}
```

- Distinguish whether the current turn is a white ball or a yellow ball
- White ball if false, and yellow ball if true
- Each time the space key is pressed, reverse the value of isTargetYellow so that the ball moves alternately.

3 Press the "Backspace" key to fix the billiard board

```
LRESULT CALLBACK d3d::WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
static bool worldMove = false;
case VK_BACK:
    worldMove= !worldMove;
    break;
}
break;

 if (LOWORD(wParam) & MK_LBUTTON) {
if (LOWORD(wParam) & MK_LBUTTON) {
    if (!worldMove) break;
```

- Pressing the 'Backspace' key reverses the worldMove variable
- If worldMove is true, billiard table movement is activated, and if false, it is disabled.

Major Functions

CSphere class

- **hasIntersected()**

: Determine if a collision has occured between the current sphere and another sphere

- **hitBy()**

: When two spheres collide, the role of calculating and reflecting the velocity after the collision

1 CSphere class

- *Private*

: position, radius, velocity

- *Public*

: local transformation matrix, material, mesh

```
class CSphere {
private:
    float center_x, center_y, center_z;
    float m_radius;
    float m_velocity_x;
    float m_velocity_z;

public:
    CSphere(void)
    {
        D3DXMatrixIdentity(&m_mLocal);
        ZeroMemory(&m_mtrl, sizeof(m_mtrl));
        m_radius = 0;
        m_velocity_x = 0;
        m_velocity_z = 0;
        m_pSphereMesh = NULL;
    }
```

1 CSphere :: hasIntersected()

- *Center coordinate calcutation*

: Called to this→getCenter(), ball.getCenter().

- *Distance calculation*

: Calculate the distance between the two points using the Pythagorean theorem.

- *Determining if there is collision*

: Consider the two spheres collided if the distance is less than or equal to the sum of the radii of the two spheres.

```
bool hasIntersected(CSphere& ball)
{
    D3DXVECTOR3 myCenter = this->getCenter();
    D3DXVECTOR3 otherCenter = ball.getCenter();

    float dx = myCenter.x - otherCenter.x;
    float dz = myCenter.z - otherCenter.z;

    float distance = sqrt(dx * dx + dz * dz);

    if (distance <= (this->getRadius() + ball.getRadius())) {
        return true;
    }
    else return false;
}
```

1 CSphere :: hitBy()

- *Calculate center coordinates and velocity*

: Bring the center coordinates of the two spheres and their respective velocities.

- *Calculation of collision direction*

: Calculate the direction vector between the two spheres and store it in collisionNormal

- *Crash detection and handling*

: Call hasIntersected() to see if the collision occurred, normalizes CollisionNormal when the collision and calculate the relative speed of the two spheres.

```
void hitBy(CSphere& ball)
{
    D3DXVECTOR3 myCenter = this->getCenter();
    D3DXVECTOR3 otherCenter = ball.getCenter();

    D3DXVECTOR3 myVelocity = D3DXVECTOR3(this->getVelocity_X(), 0, this->getVelocity_Z());
    D3DXVECTOR3 otherVelocity = D3DXVECTOR3(ball.getVelocity_X(), 0, ball.getVelocity_Z());

    D3DXVECTOR3 collisionNormal = myCenter - otherCenter;
    float distance = D3DXVec3Length(&collisionNormal);

    if (hasIntersected(ball)) {
        D3DXVec3Normalize(&collisionNormal, &collisionNormal);

        D3DXVECTOR3 relativeVelocity = myVelocity - otherVelocity;

        float velocityAlongNormal = D3DXVec3Dot(&relativeVelocity, &collisionNormal);
```

1 CSphere :: hitBy()

- *Calculate post-crash velocity*

: If the speed in the collision direction is greater than 0, no collision processing is performed.

- *Impulse calculation*

: Calculate the velocity after impact using the rebound coefficient(e), calculate the magnitude of the impact amount, and use it to obtain the impact amount vector

- *Update speed*

: Calculate the new speeds of the two spheres and update them using the setPower() respectively.

```
        if (velocityAlongNormal > 0) {
            return;
        }

        float e = 0.9f;
        float impulseMagnitude = -(1 + e) * velocityAlongNormal / 2.0f;

        D3DXVECTOR3 impulse = impulseMagnitude * collisionNormal;

        D3DXVECTOR3 newmyVelocity = myVelocity + impulse;
        D3DXVECTOR3 newotherVelocity = otherVelocity - impulse;

        this->setPower(newmyVelocity.x, newmyVelocity.z);
        ball.setPower(newotherVelocity.x, newotherVelocity.z);
    }
}
```

Major Functions

CWall class

- **hasIntersected()**

: Determine if a collision has occured between the wall and the ball

- **hitBy()**

: When a sphere collides with a wall, the role of calculating and reflecting the velocity after impact

2 CWall class

- *Private*

: x, z-axis coordinates, width, depth, height

- *Public*

: local transformation matrix, material, mesh

```
class CWall {
private:

    float m_x;
    float m_z;
    float m_width;
    float m_depth;
    float m_height;

public:
    CWall(void)
    {
        D3DXMatrixIdentity(&m_mLocal);
        ZeroMemory(&m_mtrl, sizeof(m_mtrl));
        m_width = 0;
        m_depth = 0;
        m_pBoundMesh = NULL;
    }
```

2 CWall :: hasIntersected()

- *Get center coordinates, radius of sphere*

: Called to ball.getCenter(), ball.getRadius().

- *Calculate the boundary of the wall*

- *Determining if there is a collision*

: Considering the radius of a sphere, determine whether the sphere overlaps with the wall on the x and z axes to determine whether it collides.

```
bool hasIntersected(CSphere& ball)
{
    D3DXVECTOR3 ballCenter = ball.getCenter();
    float ballRadius = ball.getRadius();

    float wallLeft = m_x - m_width / 2.0f;
    float wallRight = m_x + m_width / 2.0f;
    float wallTop = m_z + m_depth / 2.0f;
    float wallBottom = m_z - m_depth / 2.0f;

    bool intersectsX = (ballCenter.x + ballRadius >= wallLeft) && (ballCenter.x - ballRadius
        <= wallRight);
    bool intersectsZ = (ballCenter.z + ballRadius >= wallBottom) && (ballCenter.z - ballRadius
        <= wallTop);

    return intersectsX && intersectsZ;
}
```

2 CWall :: hitBy()

- *Get center coordinate, radius of sphere and calculate velocity*

: Called to ball.getCenter(), ball.getRadius() and calculate the velocity.

- *Calculate the boundary of the wall*

- *Calculate the direction of the collision*

: If a sphere collides with a wall, set the collision direction vector according to which direction the collision occured.

```
void hitBy(CSphere& ball)
{
    D3DXVECTOR3 ballCenter = ball.getCenter();
    float ballRadius = ball.getRadius();

    D3DXVECTOR3 ballVelocity = D3DXVECTOR3(ball.getVelocity_X(), 0, ball.getVelocity_Z());

    float wallLeft = m_x - m_width / 2.0f;
    float wallRight = m_x + m_width / 2.0f;
    float wallTop = m_z + m_depth / 2.0f;
    float wallBottom = m_z - m_depth / 2.0f;

    if (hasIntersected(ball)) {
        D3DXVECTOR3 collisionNormal(0, 0, 0);

        if (ballCenter.x - ballRadius < wallLeft) {
            collisionNormal = D3DXVECTOR3(-1, 0, 0);
        }
        else if (ballCenter.x + ballRadius > wallRight) {
            collisionNormal = D3DXVECTOR3(1, 0, 0);
        }
        else if (ballCenter.z - ballRadius < wallBottom) {
            collisionNormal = D3DXVECTOR3(0, 0, -1);
        }
        else if (ballCenter.z + ballRadius > wallTop) {
            collisionNormal = D3DXVECTOR3(0, 0, 1);
        }
```

2 CWall :: hitBy()

- *Calculate speed*

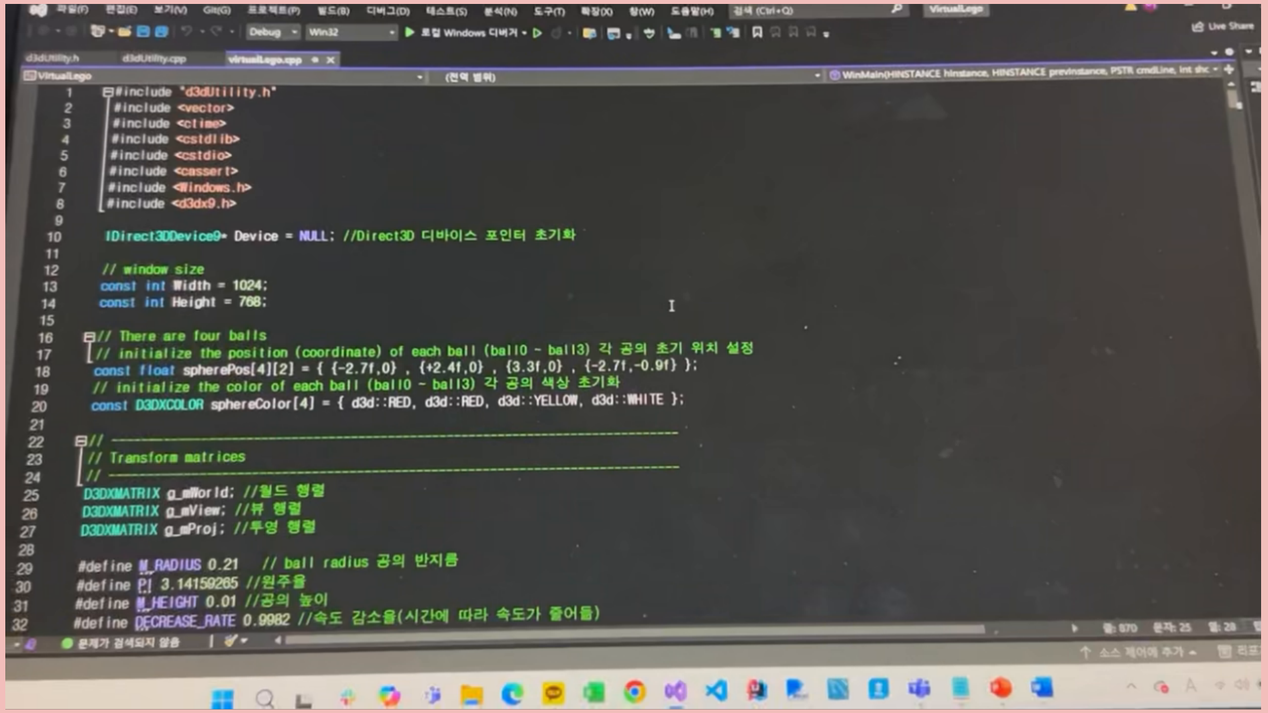
: Calculate the velocity in the direction fo collision, calculate the amount of impact using the rebound coefficient(e), and update the new velocity of the sphere.

- *Modify the position of the sphere*

: When a sphere collides with a wall, adjust the position of the sphere so that it does not across the wall.

```
        float velocityAlongNormal = D3DXVec3Dot(&ballVelocity, &collisionNormal);

        float e = 0.8f;
        float impulseMagnitude = -(1 + e) * velocityAlongNormal;

        D3DXVECTOR3 impulse = impulseMagnitude * collisionNormal;
        D3DXVECTOR3 newBallVelocity = ballVelocity + impulse;
        ball.setPower(newBallVelocity.x, newBallVelocity.z);

    D3DXVECTOR3 correctedPosition = ballCenter;
    if (collisionNormal.x != 0) {
        correctedPosition.x = collisionNormal.x < 0 ? wallLeft - ballRadius
            : wallRight + ballRadius;
    }
    if (collisionNormal.z != 0) {
        correctedPosition.z = collisionNormal.z < 0 ? wallBottom -
            ballRadius : wallTop + ballRadius;
    }

    ball.setCenter(correctedPosition.x, ballCenter.y, correctedPosition.z);
}
```

Execution Video



Q&A

Thank You