Object-Orientation Programming

Team 14 Report

TEAM MEMBER

20200009 김동영

20200699 박혜인

20200479 정수용

Contents

1. Introduce and Overview Our Game

We decided through a meeting with the team members what games could be created using example codes. As a result, we decided to create a game to eliminate the bombs falling from above.

An overview of the game is as follows.

1) As executing program, at the top end of the map, the bomb descends to the random location, and the speed descends randomly appears in the currently set speed range.

2) As the game time increases, the minimum and maximum values of the range of marbles and bombs' speeds are increased at regular intervals.

3) The player can shoot the ball through the left click of the mouse, the ball varies in speed in proportion to the distance between the ball and the mouse cursor when firing.

4) The ball shot by the player does not bounce against the wall. When the ball touches the wall, it disappears.

5) When the ball shot by the player comes into contact with the bomb, the ball and bomb disappear.

6) Among the bombs that come down to prevent players from indiscriminately clicking on the screen to continue the game, marbles come down at regular intervals.

7) When the bomb touches the bottom end of the map, the game ends

8) The game is ended, when marble comes into contact with the ball shot by the player.

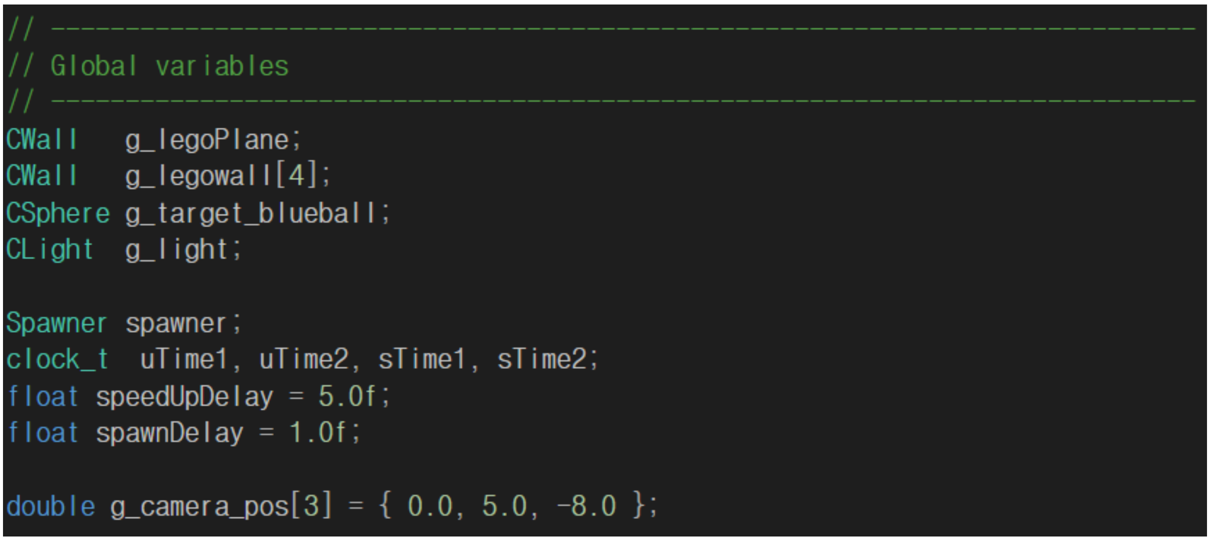
2. Descript Functionally

Our program has several functions to implement the above-described program. In order to proceed with the game, it typically implements the function of firing the ball in that direction by left-clicking the mouse, the function of disappearing the bomb when the ball and bomb come into contact, and the function of ending the game when the marble touches the floor or the ball. Additionally, other additional functions made it a smooth and fun game.

3. How we implemented

We developed this program with a sample source code, three classes, and several global variables and functions.

- Global Variables



```
// -----------------------------------------------------------------------
// Global variables
// -----------------------------------------------------------------------
CWall   g_legoPlane;
CWall   g_legowall[4];
CSphere g_target_blueball;
CLight  g_light;

Spawner spawner;
clock_t  uTime1, uTime2, sTime1, sTime2;
float speedUpDelay = 5.0f;
float spawnDelay = 1.0f;

double g_camera_pos[3] = { 0.0, 5.0, -8.0 };
```

First, create an object spawner of the Spawner class. And we created variables uTime1 and 2 and variables sTime1 and 2. The uTime variables are used to check the play time of the current game and when a certain time passes in each renewal (speedUpDelay), increase the speed range of the ball. And the sTime variables are used to measure time to spawn a new ball every certain time (spawnDelay).

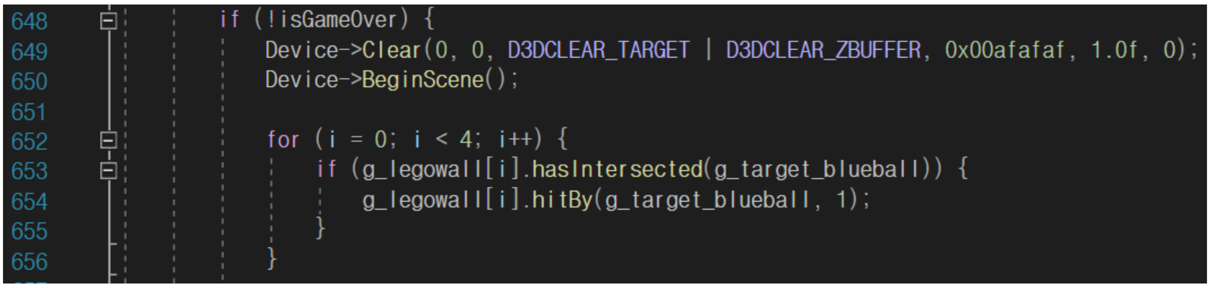
- Functions

- bool Setup()

This function is a function that runs once at the beginning of program execution. In this function, to measure the time while the program is running, the start time stores the time when the program starts. Thereafter, the values of start Time are stored in uTime1 and sTime1 to adjust the speed at which the bombs and the marbles come down as mentioned above, and the ball can be properly spawned. The rest of the codes were originally there to draw a map

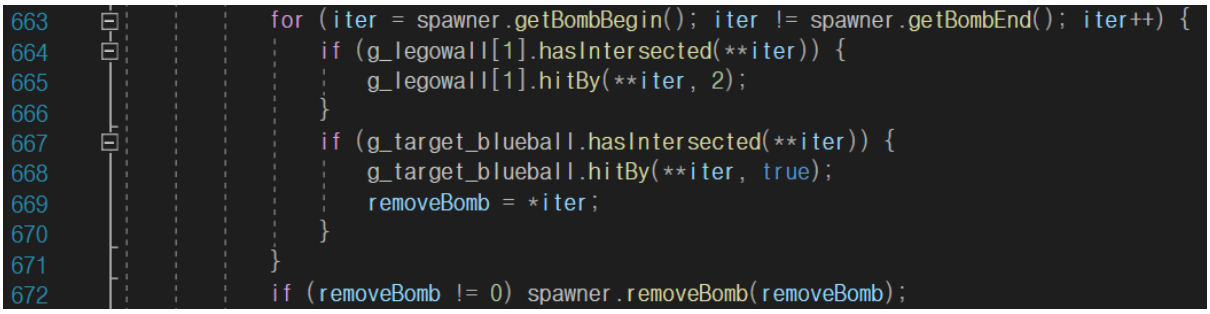
- bool Display(float timeDelta)

This function is executed every frame while the program is running so that the operation is executed without interruption when the program is executed. The first if statement performs whether isGameOver is true or false. If isGameOver is false, that is, the game is not over, the function continues to perform. However, if isGameOver is true, that is, the game is over, it stops performing the function, and updating the screen.



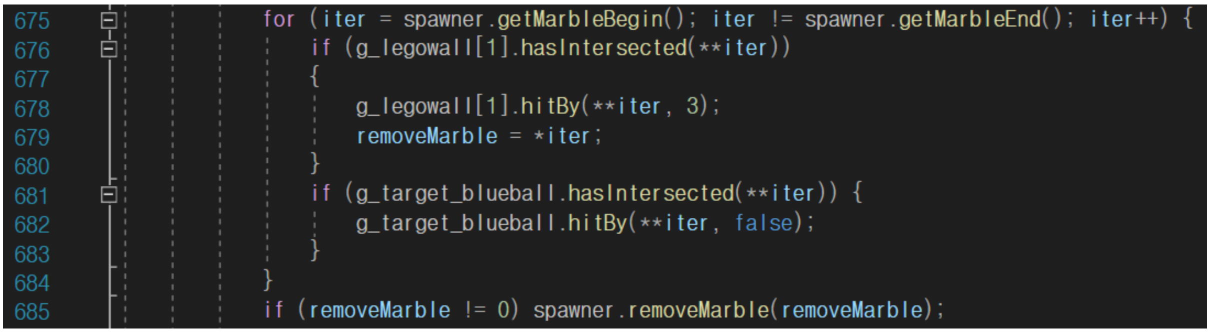
```
if (!isGameOver) {
    Device->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, 0x00afafaf, 1.0f, 0);
    Device->BeginScene();

    for (i = 0; i < 4; i++) {
        if (g_legowall[i].hasIntersected(g_target_blueball)) {
            g_legowall[i].hitBy(g_target_blueball, 1);
        }
    }
```

Anyway, when this function is executed, for statement is used to determine whether the four walls of the map and the ball shot by the player collided. If there is a collision, execute the hitBy() function described below.



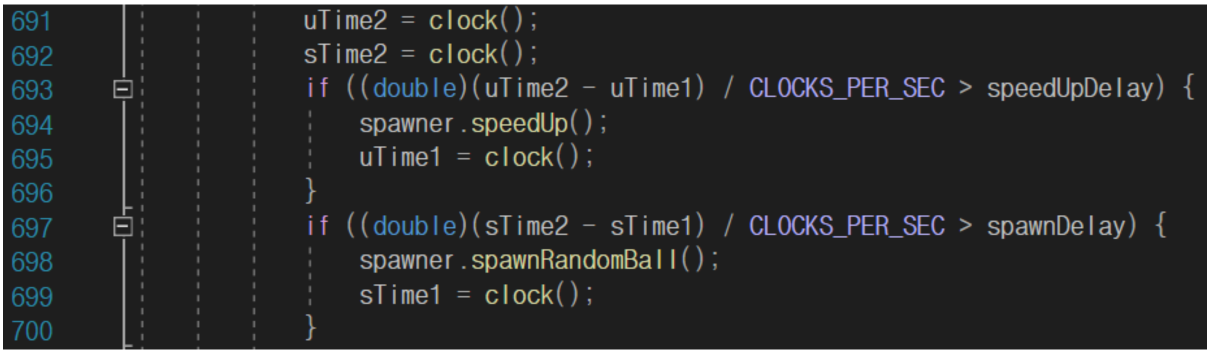
```
    for (iter = spawner.getBombBegin(); iter != spawner.getBombEnd(); iter++) {
        if (g_legowall[1].hasIntersected(**iter)) {
            g_legowall[1].hitBy(**iter, 2);
        }
        if (g_target_blueball.hasIntersected(**iter)) {
            g_target_blueball.hitBy(**iter, true);
            removeBomb = *iter;
        }
    }
    if (removeBomb != 0) spawner.removeBomb(removeBomb);
```

Then, search the bomb list from beginning to end with the for iter as a factor. Therefore, if bomb touches the floor through hasIntersected(), the game is over through hitBy(). In addition, if the player's ball and bomb touch, continue the game and save it (bomb) in the removebomb. And remove the bomb from the list.



```
for (iter = spawner.getMarbleBegin(); iter != spawner.getMarbleEnd(); iter++) {
    if (g_legowall[1].hasIntersected(**iter))
    {
        g_legowall[1].hitBy(**iter, 3);
        removeMarble = *iter;
    }
    if (g_target_blueball.hasIntersected(**iter)) {
        g_target_blueball.hitBy(**iter, false);
    }
}
if (removeMarble != 0) spawner.removeMarble(removeMarble);
```

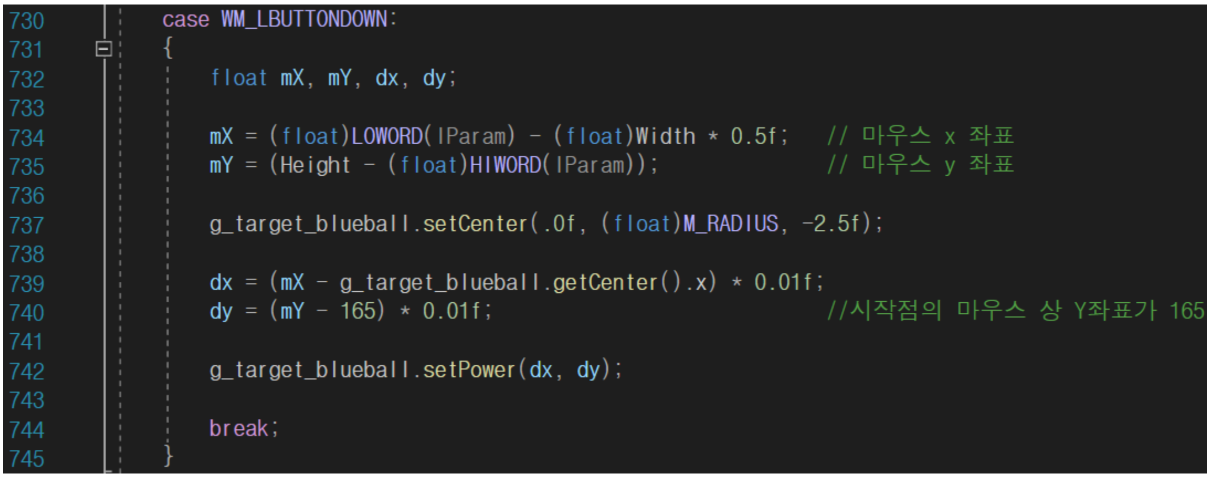
Similar to the above-mentioned method, search for a list of bombs. However, there is a difference in that when the player's ball and marbles touch, the game ends through hitby().



```
uTime2 = clock();
sTime2 = clock();
if ((double)(uTime2 - uTime1) / CLOCKS_PER_SEC > speedUpDelay) {
    spawner.speedUp();
    uTime1 = clock();
}
if ((double)(sTime2 - sTime1) / CLOCKS_PER_SEC > spawnDelay) {
    spawner.spawnRandomBall();
    sTime1 = clock();
}
```

Whenever a program is updated through Display(), time of the moments is stored in uTime2 and sTime2. After that, if the time gap of Time1 and Time2 is greater than speedUpDelay or SpawnDelay, increases the falling speed of the ball or spawns a new ball. In addition, uTime1 and sTime1 are updated again so that this process continues to be repeated.

- LRESULT CALLBACK d3d::WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)



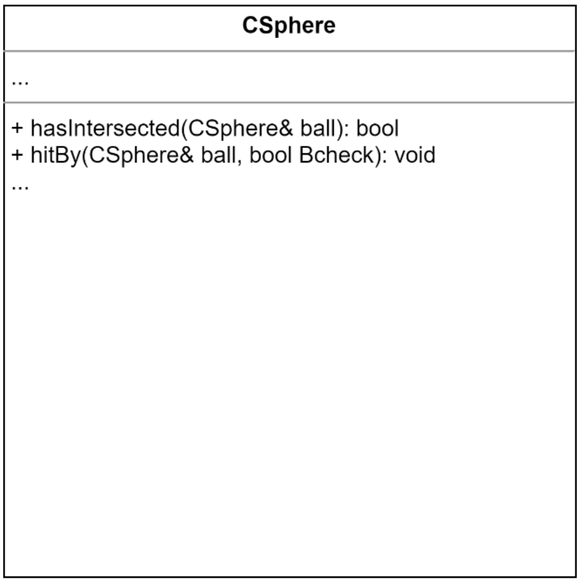
```
case WM_LBUTTONDOWN:
{
    float mX, mY, dx, dy;

    mX = (float)LOWORD(lParam) - (float)Width * 0.5f;   // 마우스 x 좌표
    mY = (Height - (float)HIWORD(lParam));              // 마우스 y 좌표

    g_target_blueball.setCenter(.0f, (float)M_RADIUS, -2.5f);

    dx = (mX - g_target_blueball.getCenter().x) * 0.01f;
    dy = (mY - 165) * 0.01f;                            //시작점의 마우스 상 Y좌표가 165

    g_target_blueball.setPower(dx, dy);

    break;
}
```

If the player clicks left of the mouse, store the clicked coordinates in mX and mY. After that, the difference between the coordinates of the left click of the mouse and the player's ball (initial position before shooting the ball) is stored in dx and dy. Thereafter, this value is transferred to setPower() so that the ball flies in that direction.

- **void game_over()**

When this function is executed, it stores the current time in the endTime variable. And save the difference between endTime and startTime that stored the start time of the program in setUp() in scoreTime to output the survival time with the sentence "Game Over" and the time that has been played so far.

- CSphere class



CSphere

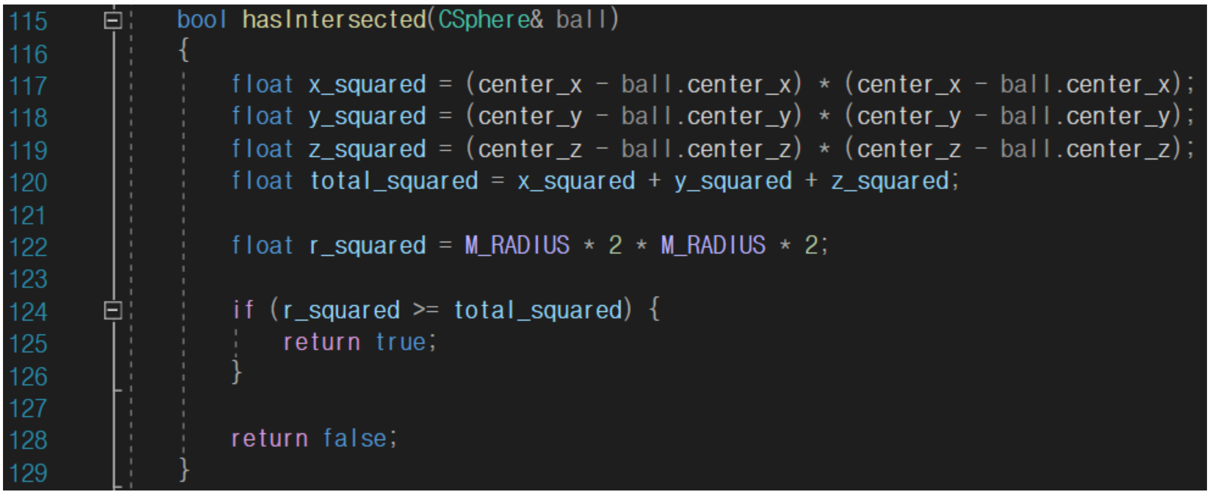
...

+ hasIntersected(CSphere& ball): bool
+ hitBy(CSphere& ball, bool Bcheck): void
...

("…" means the variables and the methods that include in example codes.)

First, we use CSphere class. This class is a class that implements the interaction between the player's ball (blue ball) and bombs and marbles. In this class, hasIntersected() and hitBy() were implemented.

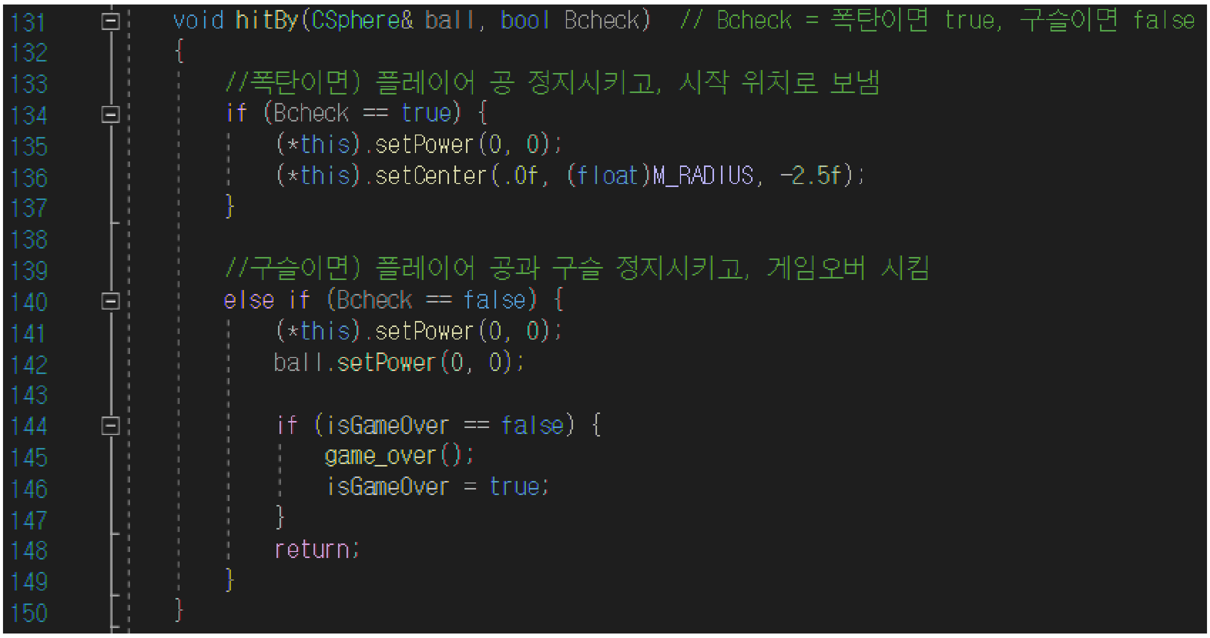
- bool hasIntersected(CSphere& ball)



```
bool hasIntersected(CSphere& ball)
{
    float x_squared = (center_x - ball.center_x) * (center_x - ball.center_x);
    float y_squared = (center_y - ball.center_y) * (center_y - ball.center_y);
    float z_squared = (center_z - ball.center_z) * (center_z - ball.center_z);
    float total_squared = x_squared + y_squared + z_squared;

    float r_squared = M_RADIUS * 2 * M_RADIUS * 2;

    if (r_squared >= total_squared) {
        return true;
    }

    return false;
}
```

This method can know whether the player's ball contacts marbles or bombs. To do this, we made it possible to find the distance between points and points. If the square of the distance is less than the radius of the ball, bomb, and marbles (which are all the same), determine that they collide with each other, and if the distance is not less than the radius, they don’t collide.

- void hitBy(CSphere& ball, bool Bcheck)

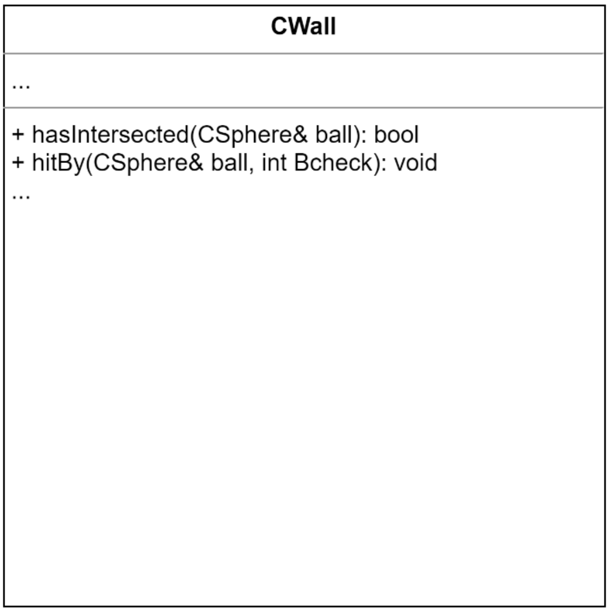


```
void hitBy(CSphere& ball, bool Bcheck)  // Bcheck = 폭탄이면 true, 구슬이면 false
{
    //폭탄이면) 플레이어 공 정지시키고, 시작 위치로 보냄
    if (Bcheck == true) {
        (*this).setPower(0, 0);
        (*this).setCenter(.0f, (float)M_RADIUS, -2.5f);
    }

    //구슬이면) 플레이어 공과 구슬 정지시키고, 게임오버 시킴
    else if (Bcheck == false) {
        (*this).setPower(0, 0);
        ball.setPower(0, 0);

        if (isGameOver == false) {
            game_over();
            isGameOver = true;
        }
        return;
    }
}
```

This method is executed when a ball and a marble, or a ball and a bomb collide. The parameter bool Bcheck of this method stores bomb or marble separately. If a player’s ball hits bomb, store true, and if it hits marble, store false.

Therefore, if Bcheck is false, the game should be stopped. Afterwards, it is determined whether it is true through the if statement (the initial value of the isGameOver is false, so it is determined as true). Therefore, the game_over() function is executed while stopping the player's ball and the program is ended.

- CWall Class



CWall

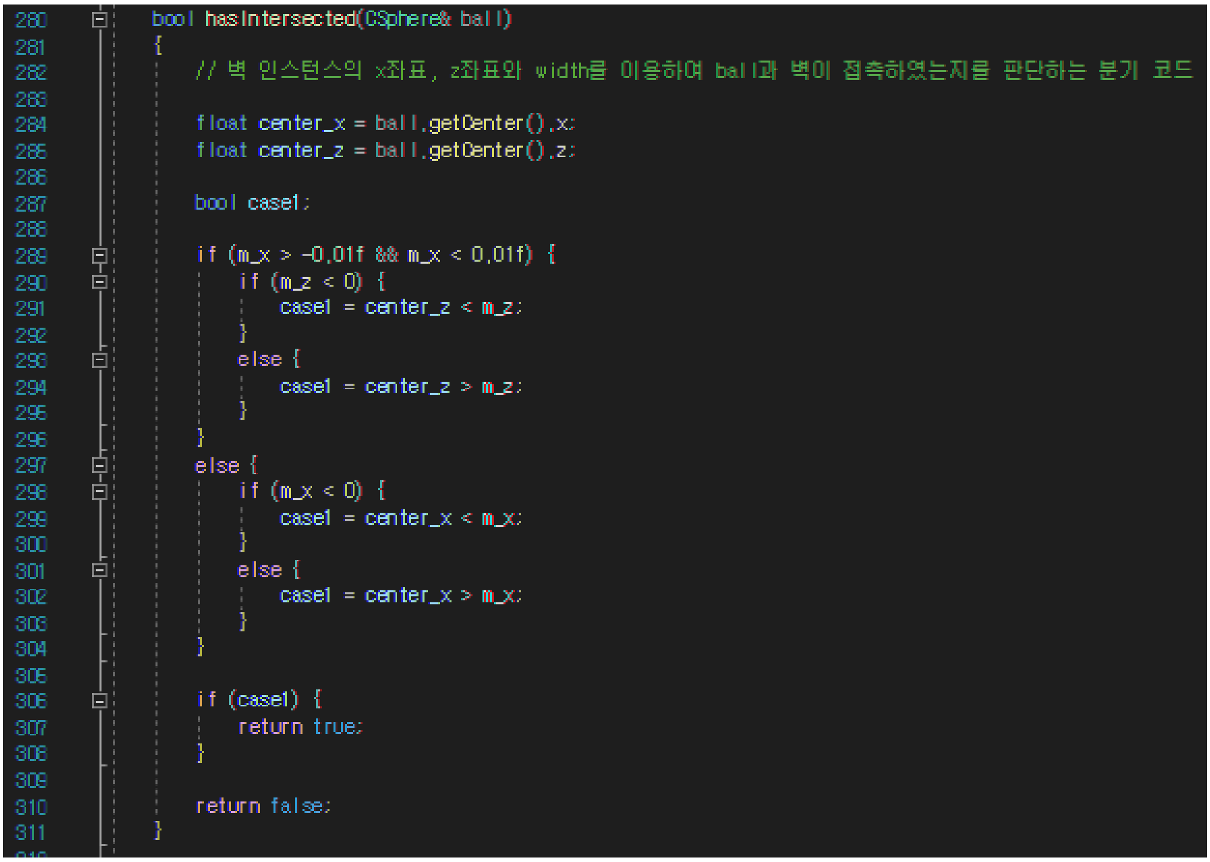
...

+ hasIntersected(CSphere& ball): bool
+ hitBy(CSphere& ball, int Bcheck): void
...

"…" means the variables and the methods that include in example codes.)

And we use CWall class. This class is a class that implements the interaction between all balls (including player's ball, bombs, and marbles) and four walls. In this class, hasIntersected() and hitBy() were implemented like CSphere.

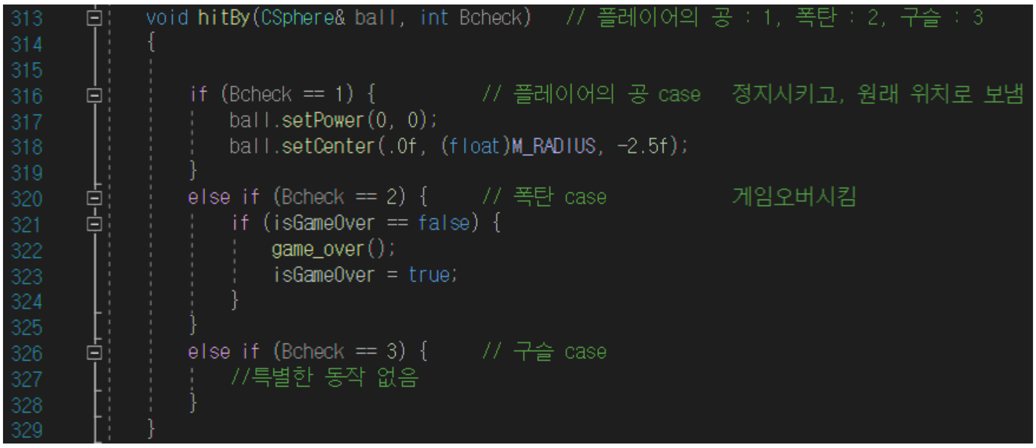
- bool hasIntersected(CSphere& ball)



```
bool hasIntersected(CSphere& ball)
{
    // 벽 인스턴스의 x좌표, z좌표와 width를 이용하여 ball과 벽이 접촉하였는지를 판단하는 분기 코드

    float center_x = ball.getCenter().x;
    float center_z = ball.getCenter().z;

    bool case1;

    if (m_x > -0.01f && m_x < 0.01f) {
        if (m_z < 0) {
            case1 = center_z < m_z;
        }
        else {
            case1 = center_z > m_z;
        }
    }
    else {
        if (m_x < 0) {
            case1 = center_x < m_x;
        }
        else {
            case1 = center_x > m_x;
        }
    }

    if (case1) {
        return true;
    }

    return false;
}
```

In this method, we receive the x-coordinate and z-coordinate of the ball. If the x-coordinate is within a certain range (almost close to zero), check the value of the z-coordinate. If the z-coordinate is less than 0, it means that the ball has touched the floor in the coordinates, and if it is greater than 0, it has touched the top.

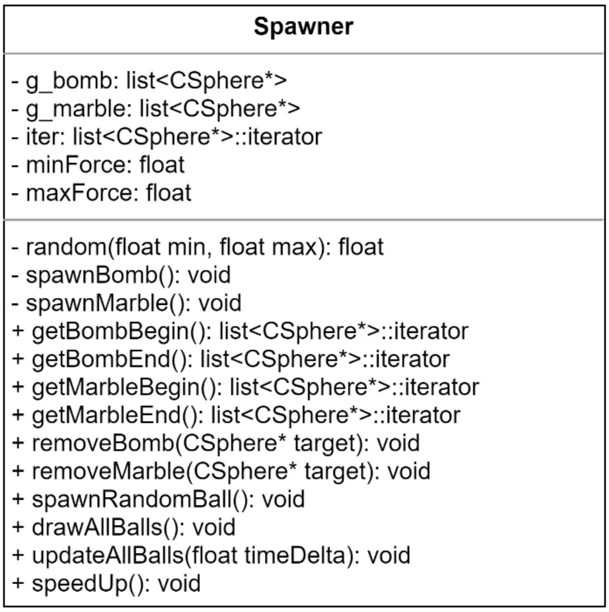
- void hitBy(CSphere& ball, int Bcheck)



```
void hitBy(CSphere& ball, int Bcheck)   // 플레이어의 공 : 1, 폭탄 : 2, 구슬 : 3
{

    if (Bcheck == 1) {          // 플레이어의 공 case   정지시키고, 원래 위치로 보냄
        ball.setPower(0, 0);
        ball.setCenter(.0f, (float)M_RADIUS, -2.5f);
    }
    else if (Bcheck == 2) {     // 폭탄 case            게임오버시킴
        if (isGameOver == false) {
            game_over();
            isGameOver = true;
        }
    }
    else if (Bcheck == 3) {     // 구슬 case
        //특별한 동작 없음
    }
}
```

First, use the Bcheck parameter to distinguish what type of ball is. And different results come out depending on the type of ball. First, Bcheck is 1, and hitBy() is executed means that the player's ball hits the wall, so place it in its original position and set the speed to zero. And if Bcheck is 2, it means bomb. The only case where the bomb touches the wall is when it touches the floor, and in this case, the game is terminated. Finally, if Bcheck is 3, it proceeds without any special action.

- Spawner Class

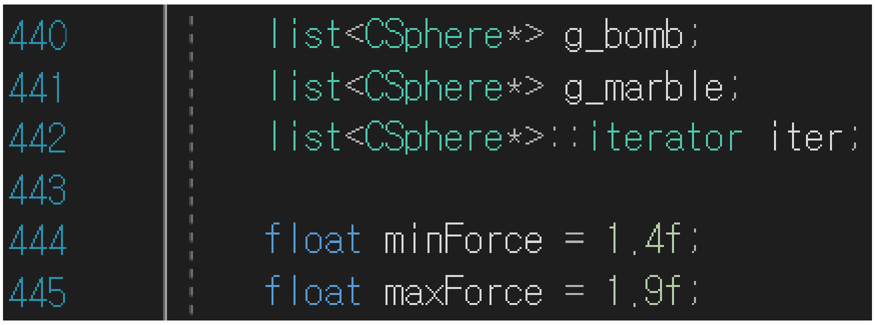


Spawner

- g_bomb: list<CSphere*>
- g_marble: list<CSphere*>
- iter: list<CSphere*>::iterator
- minForce: float
- maxForce: float

```
- random(float min, float max): float
- spawnBomb(): void
- spawnMarble(): void
+ getBombBegin(): list<CSphere*>::iterator
+ getBombEnd(): list<CSphere*>::iterator
+ getMarbleBegin(): list<CSphere*>::iterator
+ getMarbleEnd(): list<CSphere*>::iterator
+ removeBomb(CSphere* target): void
+ removeMarble(CSphere* target): void
+ spawnRandomBall(): void
+ drawAllBalls(): void
+ updateAllBalls(float timeDelta): void
+ speedUp(): void
```

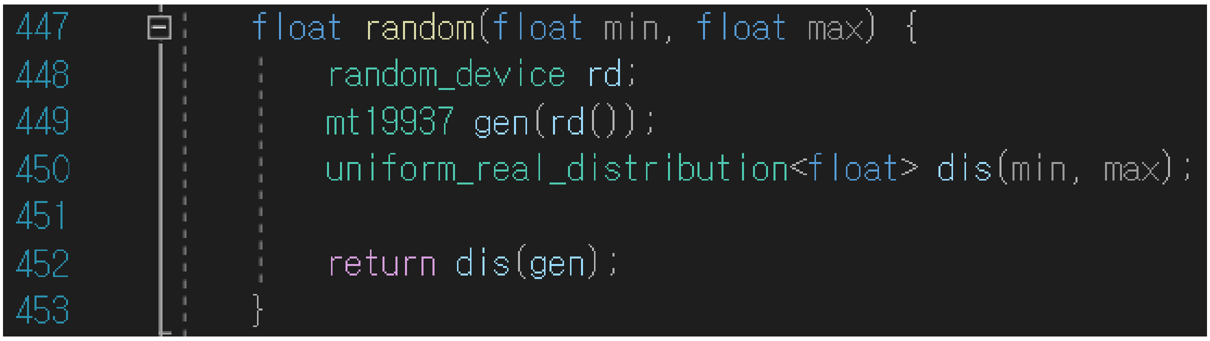
This class is the core of our game. It was created to spawn balls and marbles and also to perform the function of controlling their speed and spawning.



```
    list<CSphere*> g_bomb;
    list<CSphere*> g_marble;
    list<CSphere*>::iterator iter;

    float minForce = 1.4f;
    float maxForce = 1.9f;
```

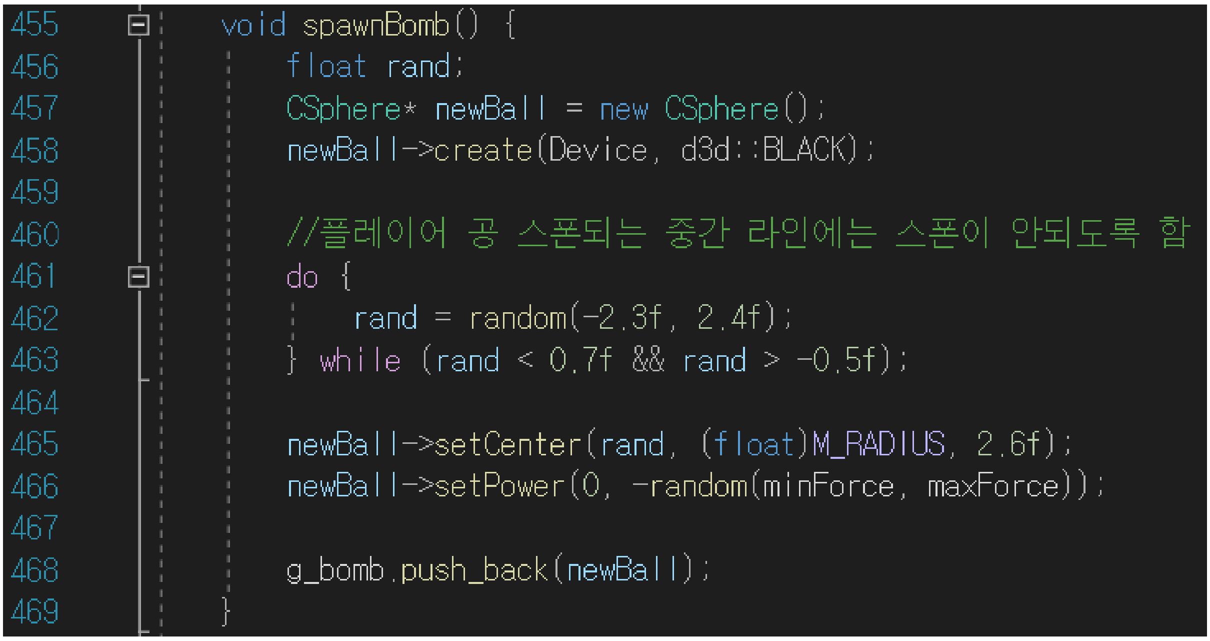
First, we declared a total of three list: for bombs, marble, and searching the list. Initially, it was declared as an arrangement, but as the game progressed, an error may occur if the arrangement is exceeded, so it was declared as a list.

And minForce and maxForce are variables that store the minimum and maximum boundaries to control the rate at which the marbles fall.



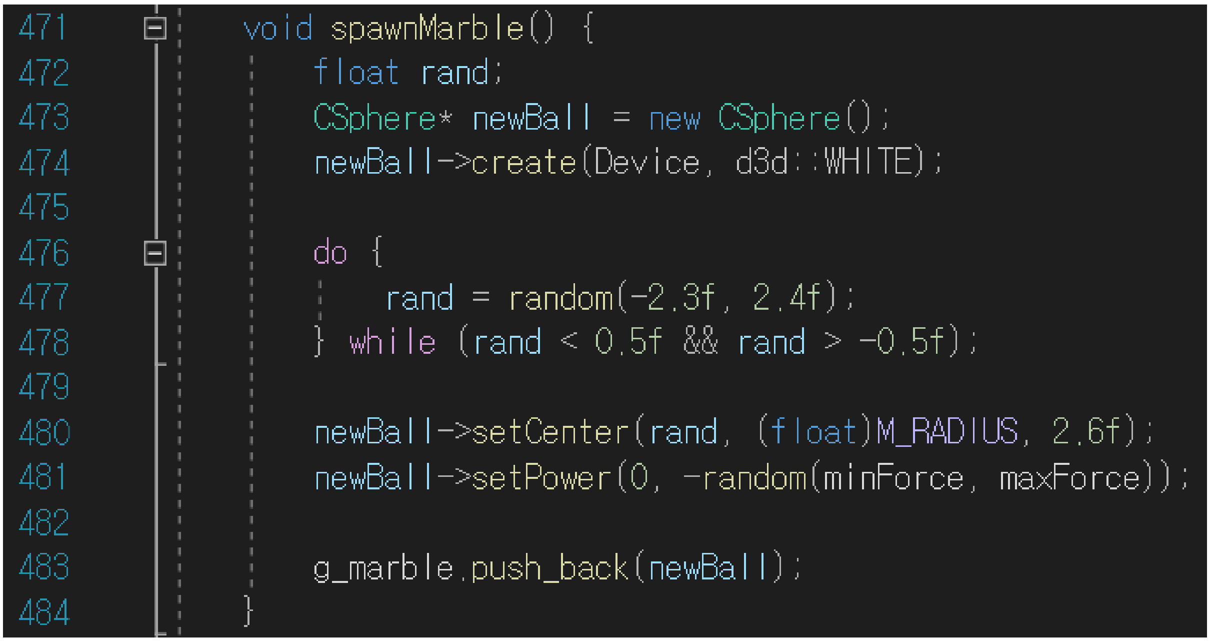
```
float random(float min, float max) {
    random_device rd;
    mt19937 gen(rd());
    uniform_real_distribution<float> dis(min, max);

    return dis(gen);
}
```

Pick one random number between the minimum and maximum boundaries of speed and apply it to the ball.



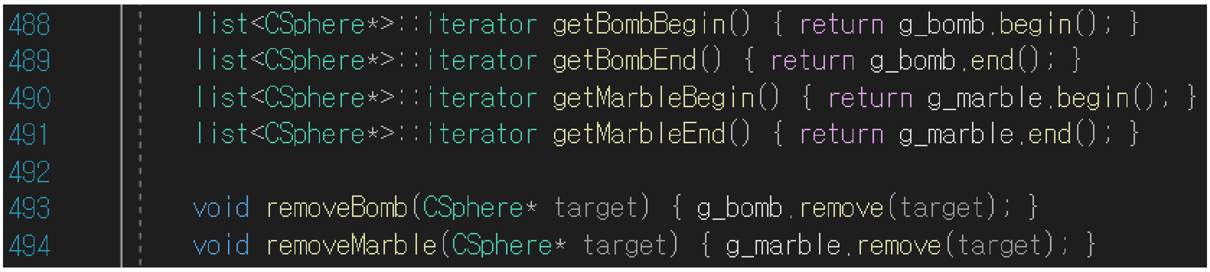
```
void spawnBomb() {
    float rand;
    CSphere* newBall = new CSphere();
    newBall->create(Device, d3d::BLACK);

    //플레이어 공 스폰되는 중간 라인에는 스폰이 안되도록 함
    do {
        rand = random(-2.3f, 2.4f);
    } while (rand < 0.7f && rand > -0.5f);

    newBall->setCenter(rand, (float)M_RADIUS, 2.6f);
    newBall->setPower(0, -random(minForce, maxForce));

    g_bomb.push_back(newBall);
}
```

The method is to spawn bomb on random x-coordinates and send it down. Through the do while door, the player's ball was not spawned at the center of the map.



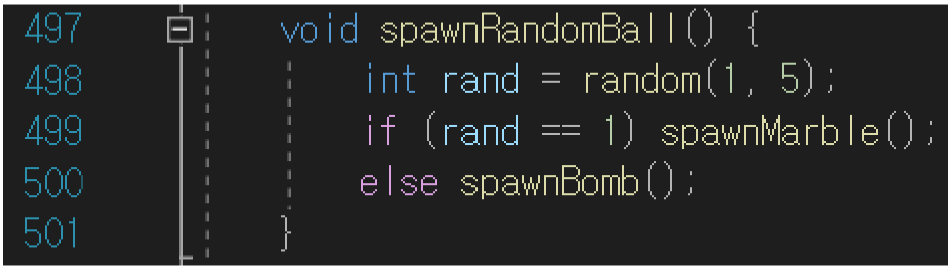
```
    void spawnMarble() {
        float rand;
        CSphere* newBall = new CSphere();
        newBall->create(Device, d3d::WHITE);

        do {
            rand = random(-2.3f, 2.4f);
        } while (rand < 0.5f && rand > -0.5f);

        newBall->setCenter(rand, (float)M_RADIUS, 2.6f);
        newBall->setPower(0, -random(minForce, maxForce));

        g_marble.push_back(newBall);
    }
```

This is very similar to the method above. There is a difference in that it spawns marbles, not bombs.



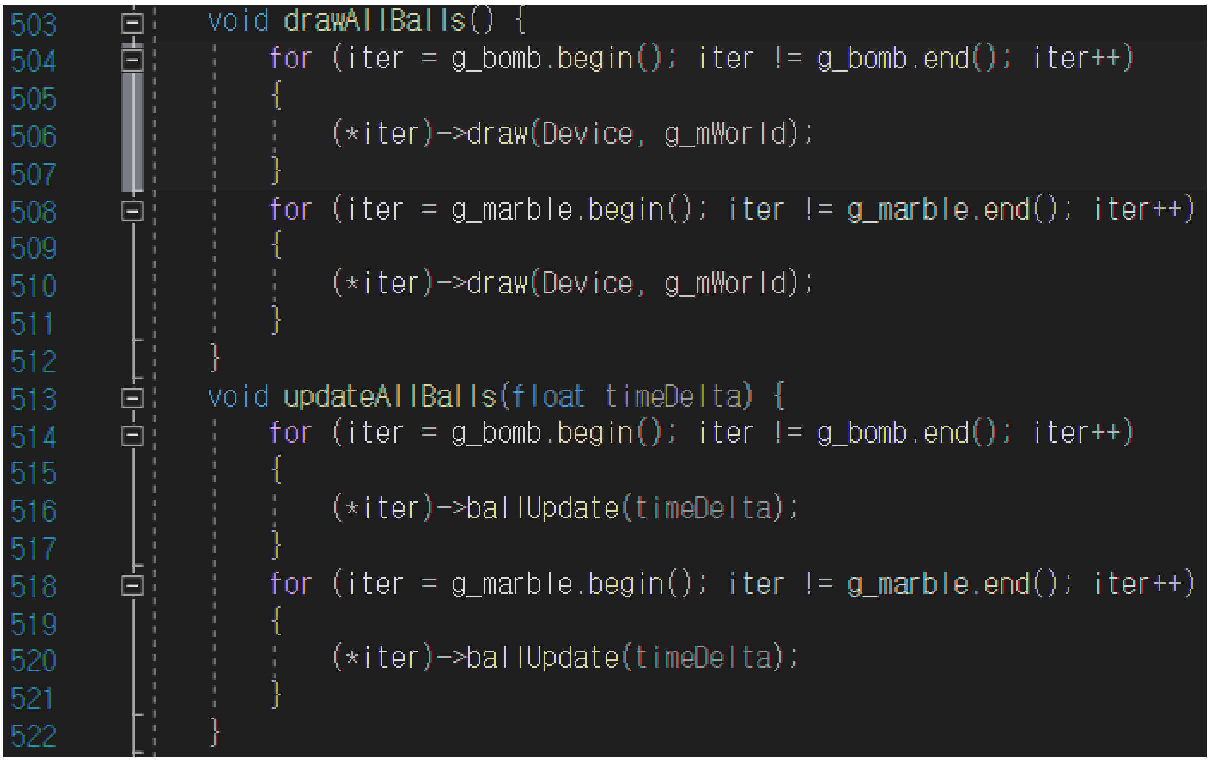
```
    list<CSphere*>::iterator getBombBegin() { return g_bomb.begin(); }
    list<CSphere*>::iterator getBombEnd() { return g_bomb.end(); }
    list<CSphere*>::iterator getMarbleBegin() { return g_marble.begin(); }
    list<CSphere*>::iterator getMarbleEnd() { return g_marble.end(); }

    void removeBomb(CSphere* target) { g_bomb.remove(target); }
    void removeMarble(CSphere* target) { g_marble.remove(target); }
```

First, the four iterators are those used for list search. And removeBomb and removeMarble are methods to remove bomb and marble from the list when they disappear according to each condition.



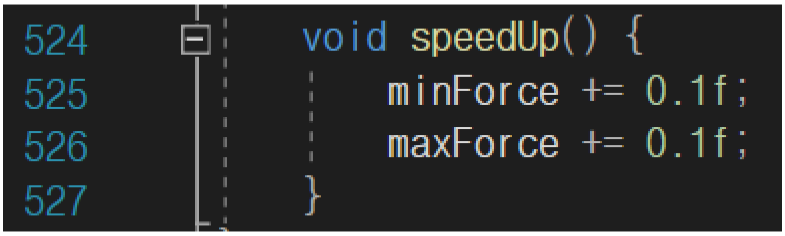
```
    void spawnRandomBall() {
        int rand = random(1, 5);
        if (rand == 1) spawnMarble();
        else spawnBomb();
    }
```

Through this method, the marble is spawn with a 1/4 probability. Marble is spawned only when it is 1 of the numbers 1 to 4, and bomb is spawned in the remaining cases.



```
void drawAllBalls() {
    for (iter = g_bomb.begin(); iter != g_bomb.end(); iter++)
    {
        (*iter)->draw(Device, g_mWorld);
    }
    for (iter = g_marble.begin(); iter != g_marble.end(); iter++)
    {
        (*iter)->draw(Device, g_mWorld);
    }
}
void updateAllBalls(float timeDelta) {
    for (iter = g_bomb.begin(); iter != g_bomb.end(); iter++)
    {
        (*iter)->ballUpdate(timeDelta);
    }
    for (iter = g_marble.begin(); iter != g_marble.end(); iter++)
    {
        (*iter)->ballUpdate(timeDelta);
    }
}
```

DrawAllBalls() and updateAllBalls() are executed through display(). It draws and updates the map one by one while exploring the list from beginning to end. Through this, the balls on the list are displayed on the screen.



```
void speedUp() {
    minForce += 0.1f;
    maxForce += 0.1f;
}
```

This method increases the speed of the ball falling as the game play time increases by increasing the minimum and maximum boundaries of the speed every certain period of time.

4. Executing Result

It is recorded as a video and cannot be attached to the word file. I'll submit it separately.

5. How to Compile and Execute

We compiled this program in Visual Studio 2019. And it is executed when clicking the “F5” or “Debug” -> “Start Debugging”.

6. Summary and Conclusion

We were able to apply various concepts while implementing this program. We can study encapsulation. For example, in CSphere class and Spawner class, we declared private members and manipulated them using methods rather than directly dealing with them to prevent misuse of objects. So, we can study encapsulation, one of the very important concepts in object-oriented programming, more detail. And we can study unfamiliar contents by using the list library for marbles and Bombs, random library to use random numbers, and time library to measure time.

Above all, however, it was very complicated and difficult at first by implementing it using an unfamiliar library, direct x, but it seems to have been very beneficial to learn new things.