Object Oriented Programing

Project #3 - Report

Course	Object Oriented Programming
Department	Computer Science & Engineering
Professor	Bong-Soo Sohn
Team	#8
Term	2023 - 2
Members	김진호 김혁진 박수빈 박주연 송민혁 신성섭

[Table of Contents]

I. Summary

1) Project title

'3D ARKANOID game'

2) Team members

20201XX4 김진호
20205XX3 김혁진
20226XX9 박수빈
20200XX6 박주연
20203XX0 송민혁
20226XX2 신성섭

3) Presentation Speaker

20203XX0 송민혁

4) Brief Project Description

We made 3D ARKANOID game, which is the game of breaking bricks with a ball and a movable bar. In our case, we replaced bricks with 'red balls'. A player can move a bar by moving mouse from side to side with right click. Once the white ball(player ball) gets a collision with the red ball, red ball disappears and white ball bounces to the opposite direction.
We used C++ for our programming language, and developed program mainly on Visual Studio 2022. Also, we used Microsoft DirectX SDK and file "d3dUtility.h" to make and visualize the 3D ARKANOID game.
In this project, we focused on utilizing object-oriented programming on our code. We tried to divide some roles of each element, such as 'ball', 'life', 'item', and make them be harmonized with each other. Also we tried to make the game more fun and special.

II. How to execute

- **Compiler** : visual studio 2022 recently version (using x32)

- **OS** : Window

- **Source Code**
  1) virtualLego.cpp
  2) CSphere1.cpp
  3) CWall.cpp
  4) CLife.cpp
  5) CLight.cpp
  6) Item.cpp
  7) d3dUtility.cpp

- **Header file**
  1) CSphere.h
  2) CWall.h
  3) CLife.h
  4) CLight.h
  5) Item.h
  6) d3dUtility.h

After placing the files into source files and header files, simply click ‘run’ to execute.

III. Description on functionality

< General >

1) Appearance

- **Ball generation**

Through the create method of Csphere, the ball is generated.

- **Ball disappearance**

When the sphere_exist value of Csphere is set to false, even if the ball object has been created, it will not be visible on the screen. We can also eliminate the ball created by calling the destroy() method of Csphere.

- ***'set_up'* function**

It creates the overall framework of the game board, initializes blocks, user bar, and necessary white balls, and sets the characteristics of the objects.

- ***'Display'* function**

At regular intervals, the program updates the ball's movement, checks for collisions, check the ball is in the correct position, updates the item duration, moves the blocks to new positions, and creates new blocks.

2) Collision

- **between Bar & White ball**

When the white ball collides with the user bar, it bounces off by giving it an angle identical to the angle of impact, along with a random speed variation.

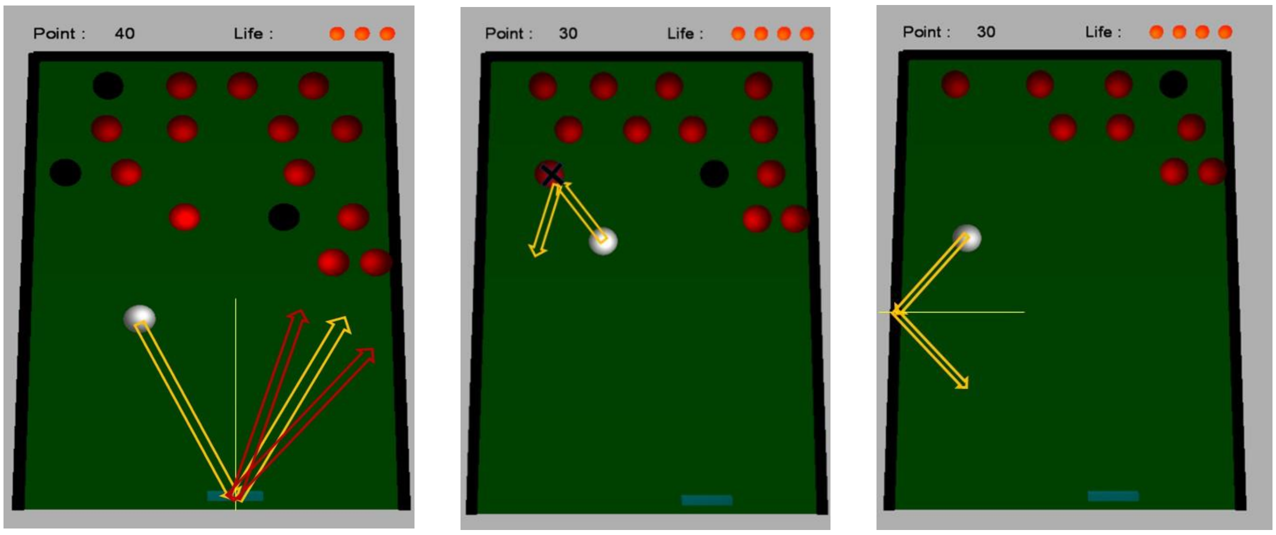
- **between White ball & Red ball(block)**

When the white ball collides with a block, it eliminates the block and randomly varies the speed of the white ball.

- **between White ball & Wall**

When the ball collides with a wall, it bounces off at an angle identical to the angle of impact.

*Collision example



<Left: Collision with the user bar | Middle: Collision with the ball | Right: Collision with the wall>

Bar at an angle identical to the angle of impact. Due to the random speed variation, the ball can move in the direction of the yellow arrows as well as in the direction of the red arrows.

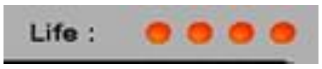
The middle picture shows the movement of the white ball when it collides with a block. When the ball collides, the block disappears, and the white ball's speed varies randomly.

The right picture shows the movement of the ball when it collides with a wall. When the ball collides with the wall, it bounces off at an angle identical to the angle of impact.

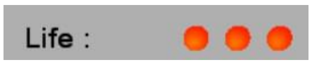
< Additional >

1) Life

- **Life Graphic**

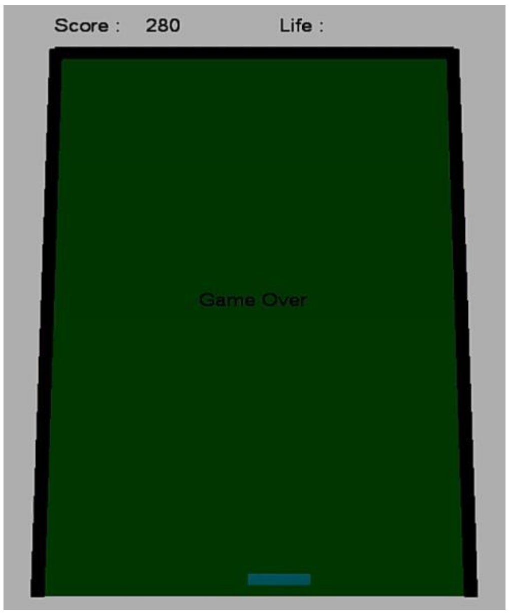


Life points is represented in Graphic UI.



When the user fails to bounce the ball, that is, when the ball falls to the bottom of the table, one life point is deducted.

- **Game over conditions**



The game ends when the remaining number of life points becomes zero.
In addition, if the blocks that were not destroyed reach the bottom line, the game also ends. Life points also becomes zero in that situation.

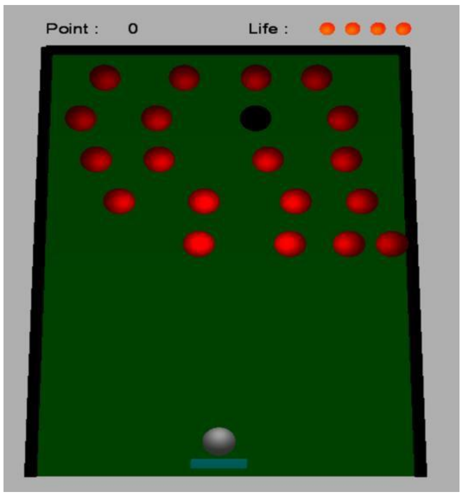
2) Score

After the white ball pops out of the player bar, we added a functionality that increases the score as the number of red balls you match increases until before white ball hit player bar again. The first time the white ball matched the red ball, we were given 10 points, the second time we were given 20 points, the third time we were given 40 points, and so on, giving each player twice the previous points. In order to prevent the score from increasing significantly, the maximum score that can be added has been fixed at 80 points.

3) Text on the run screen(showing score)

A functionality was added to display the total score on the screen each time a score is added.

4) Player Bar

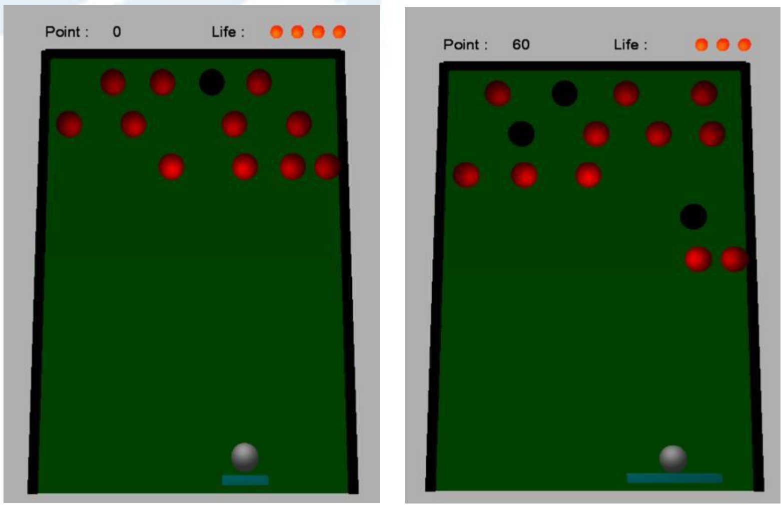


It is the user bar responsible for bouncing the white ball. It can be adjusted left or right with a right-click of the mouse. The user bar can move only within the space between the left wall and the right wall, which share the same x-axis value.

5) Item

* When the black ball is destroyed, the player randomly obtains one of the four items.

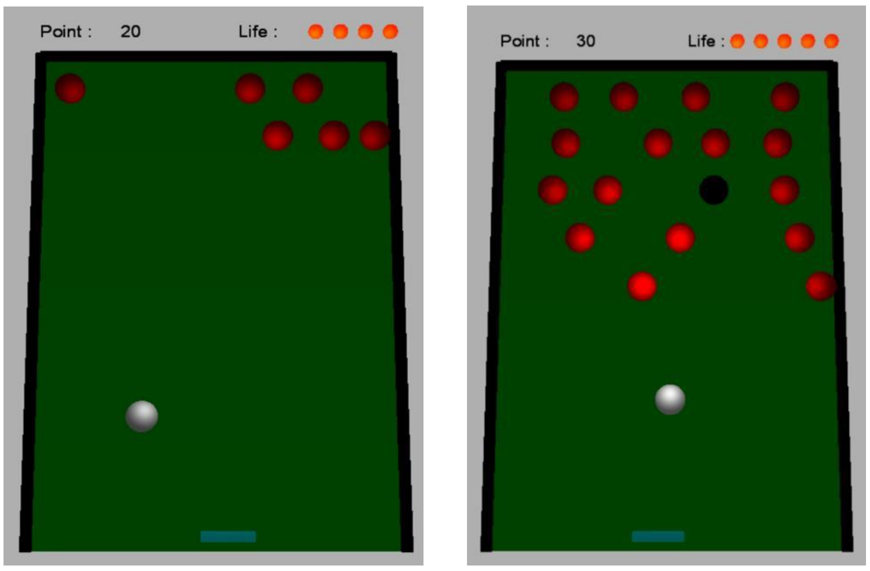
- **Longer player bar**



<Left - Default, Right - User bar length increase item applied>

It increases the length of the item bar to twice its size for a certain duration.

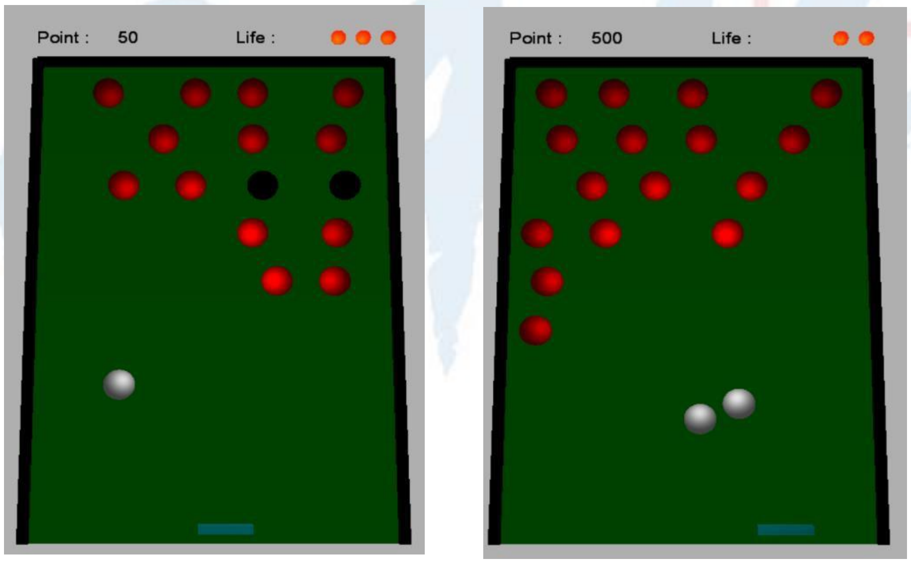
- **Additional life**



<Left - Default, Right – Additional life item applied>

It adds one life. The maximum number of lives that can be held is five.

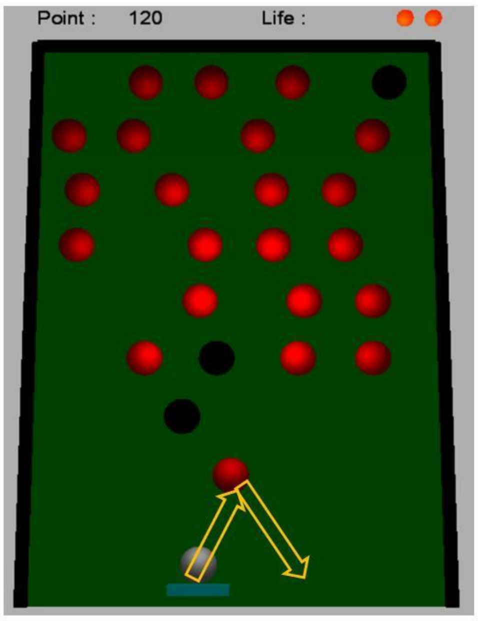
- **Additional white ball**

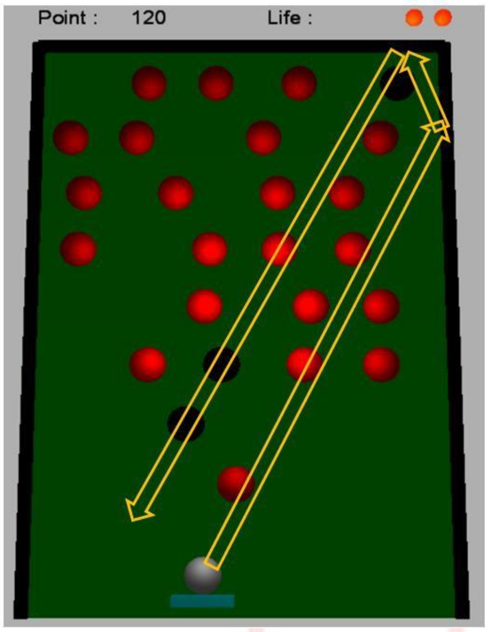


<Left - Default, Right - Additional white ball item applied>

It adds one white ball to the screen. The maximum limit for the number of white balls that can exist on the screen is five.

-Collision ignoring



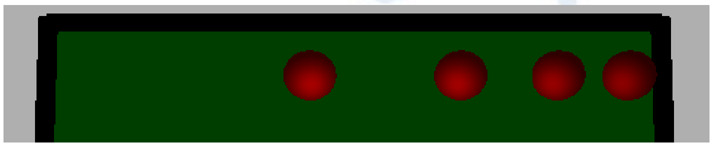


<Left - Default ball movement, Right - Movement after consuming an item>

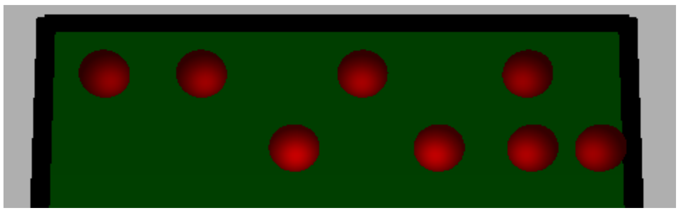
For a certain duration, it moves straight as if there were no collisions between the balls affecting its direction. It's a powerful item that allows the removal of many blocks at once and can lead to earning a high score.

6) Red ball coming down(increasing)

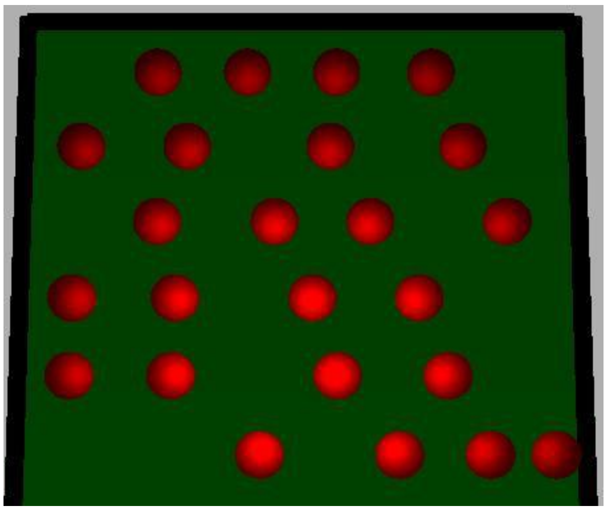
➔ **Set new balls and lower existing balls when 2 sec has elapsed.**



When the game starts, four red balls are forming a single line.



After 2 seconds since starting 'ARKANOID,' the existing line descends one step, and a new line is generated at the top.



In this way, new sets of four red balls are generated at two-second intervals without overlapping.

7) White ball's initial position

When starting the game for the first time or restarting the game after running out of lives, a functionality has been added to place the initial position of the white ball directly above the player bar. Additionally, a functionality was added that allows the white ball to move when the player bar is adjusted with the right cursor when it is directly above the player bar before the game starts.

IV. Implementation

< General >

1) Appearance

- **Ball generation**

Provide the Device and color as parameters in the create method of the *Csphere* object, a ball is created.

- **Ball disappearance**

By using the *set_exist(bool exist)* method of *Csphere* and passing false as a parameter, the *Csphere*'s ball is created but not visible on the screen. The collision detection and drawing processes are designed to execute functions only when the exist value is True.

- ***'set_up'* function**

The *Cwall* class is used to create the background, walls, and the user bar. Each wall and the user bar are assigned unique numbers. The *Csphere* class is used to create white balls and initialize their properties. The *Clife* class is used to create life icons.

- ***'Display'* function**

A *timedelta* is used to trigger the *Display* function for each frame. A loop is employed to check for collisions between all walls and white balls. Additionally, collisions between the user bar and all white balls are checked. Another loop updates the positions of all white balls. Collisions between existing white balls and existing blocks are checked using a loop. If a white ball has an item, it uses the item and sets the item property to false. The item duration is updated by subtracting 1 from its current value. The user bar, white balls, life icons, and blocks are drawn in each iteration of the loop.

2) Collision

To reduce the occurrence of a bug where balls get stuck, We added an attribute called *'no_change_time'* to *Csphere*. If the '*no_change_time'* value is positive, collisions are ignored during that period.

- **between Bar & White ball**

The collision is determined through the *hitBy(CSphere &ball)* method. If the absolute difference between the z-axis values of the ball's center and the user bar is less than the sum of the radii of the user bar and the ball, a collision is detected. For the *UserBar*, which has a *wallnum* of 3, We set the X-axis speed the same and reversed the Y-axis speed, then added a random number to both the X-axis and Y-axis speeds.

- **between White ball & block**

A collision is detected if the distance between the centers of the two balls is less than the sum of their radii. If the attribute of *WhiteBall* is true, the speed of the ball is randomly changed. If it is false, *set_exist(false)* is called to eliminate the ball.

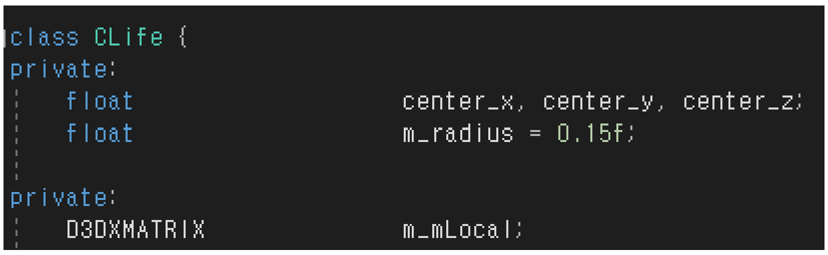
- **between White ball & Wall**

For the left wall, top wall, and right wall with wallnums 1, 2, and 3, respectively, collision detection and speed changes were done based on their wallnums.
For the left and right walls, collision was detected if the absolute difference between the x-axis value of the wall and the x-axis value of the ball is less than the sum of their thickness and the radius of the ball. The X-axis speed was set in the opposite direction, and the Z-axis speed was kept unchanged.
For the top wall, collision was detected if the absolute difference between the z-axis value of the wall and the z-axis value of the ball is less than the sum of their thickness and the radius of the ball. The X-axis speed was kept unchanged, and the Z-axis speed was set in the opposite direction.

<Additional>

1) Life

- **CLife Class**



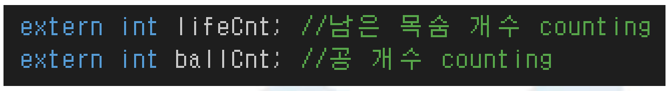
```
class CLife {
private:
    float                   center_x, center_y, center_z;
    float                   m_radius = 0.15f;

private:
    D3DXMATRIX              m_mLocal;
```

The life point UI is implemented as a separate class called *CLife*.

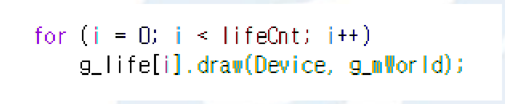
CLife class is created by modifying the existing *CSphere* class. We removed member functions and variable related to collision, and changed the design of the sphere.

- **Life counter**



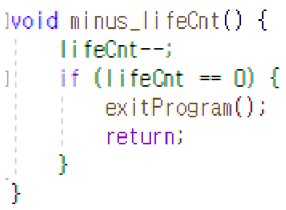
```
extern int lifeCnt; //남은 목숨 개수 counting
extern int ballCnt; //공 개수 counting
```

The global variable *lifeCnt* manages the number of remaining life points.



```
for (i = 0; i < lifeCnt; i++)
    g_life[i].draw(Device, g_mWorld);
```

The program draws life points corresponding to the number of life counter at the *Display()* function part.



```
void minus_lifeCnt() {
    lifeCnt--;
    if (lifeCnt == 0) {
        exitProgram();
        return;
    }
}
```

When the mentioned condition is detected, the program executes the *minus_lifeCnt()*function to deduct life point.

When the number of life points becomes zero, *exitProgram()* function is executed toterminate the game.

The *exitProgram()* function simply clears all the blocks on the screen.

2) Score

We added the *plusPoint* function to the *CSphere* class and placed it in *hitby*, which is a function in the *CSphere* class. So, it was implemented to increase the score every time the ball collides. In addition, doubling the score or fixing the additional score to a maximum of 80 points was implemented within the *plusPoint* function. Since the score added each time when the white ball hits the player bar needs to be initialized to zero, I added a bool member variable *first_hit* to the *CSphere* class that checks that the white ball hits the player bar again.

3) Text on the run screen(showing score)

Font was set to ID3DXFont using the D3dx9core header file, and text was printed on the execution screen using the *DrawText* function along with the set font. Text was also added to the display function so that it continues to be displayed in each frame.

4) Player Bar

The *Player Bar* is implemented using the *CWall* class. We assigned a unique *Wallnum* of 3 to the *Player Bar* and handled collisions with the ball in a similar way as the walls collide with the ball.

One difference in the collision handling compared to wall is that we introduced a slight random variable to the speed after the collision, adding variability to the speed.

In the existing sample code, we replaced the blue ball object with the *PlayerBar* in the *WndProc()* function to allow the Player Bar to move with right-click. We restricted the x-value range of the *Player Bar* within a certain limit and fixed the Z-value as a constant with a change amount of 0 to keep the Z-value constant.

5) Item

* We implemented the *Item* class, providing attributes that show the remaining duration of items and whether the item is currently in use. When the *has_item* attribute of *CSphere* is True, the item object called the *Use_item()* method and then activated the item corresponding to the randomly obtained item number.

- **Longer player bar**

Set the Use_item1 attribute to True and added a specific constant to the value of *item_time[0]. Item_time[0]* represents the duration of the item, and we decreased its value by 1 each frame.
While item_time[0] is positive, we increased the length of the board. When *item_time[0]* becomes 0 and the *Use_item1* attribute is true, we implemented it to return to the original board length.

- **Additional white ball**

In the *Setup* function, we created five white balls in advance, setting the exist attribute to False except only one white ball. Then, when obtaining the additional white ball item, we changed the exist attribute of one of the white balls, which was previously set to False. This makes the white ball appear on the screen.

- **Additional life**

If the *LifeCnt* value is less than the maximum *Life*, We added 1 to the *LifeCnt*.

- **Collision ignoring**

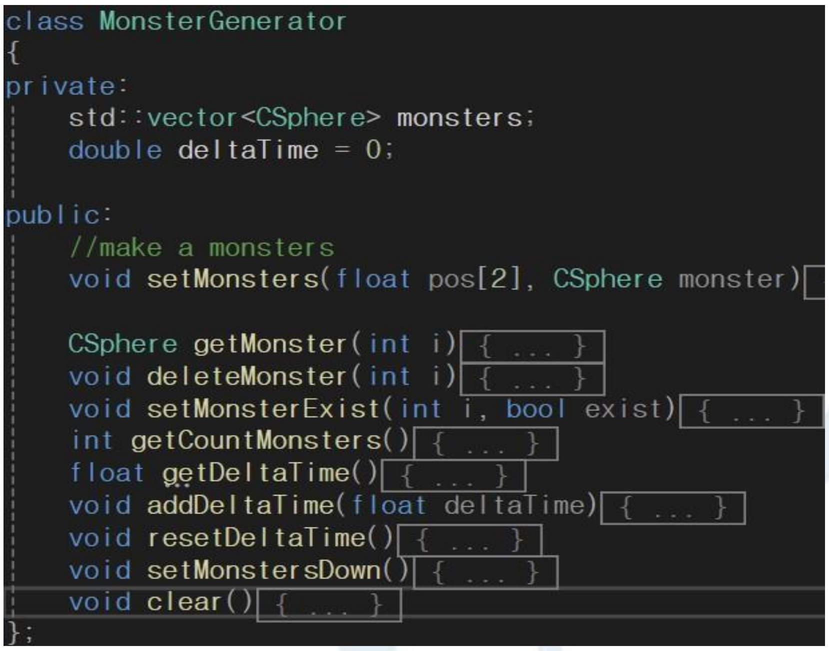
When we obtain item 4, we set the Use_item4 attribute to True and added a specific constant to the value of *item_time*[3]. Then, we set the *ignore_collision* attribute of *Csphere* to True. When *ignore_collision* is True, we implemented the *hitBy*() function of *Csphere* in a way that the speed does not change. We decreased the value of *Use_item4*, which represents the duration of the item, by 1 each frame. When the value of *Use_item4* becomes 0 and *Use_item4* is true, We set the *ignore_collision* attribute of *Csphere* to false.

6) Red ball coming down(increasing)

Todo is the same as below.

1. check the deltatime
2. if deltatime>2, set lower existing balls
3. randomly pick 4 x_values.
4. Set new balls on x_values.

So, we felt the need to separate the class that manages the ball.



```
class MonsterGenerator
{
private:
    std::vector<CSphere> monsters;
    double deltaTime = 0;

public:
    //make a monsters
    void setMonsters(float pos[2], CSphere monster)

    CSphere getMonster(int i){ ... }
    void deleteMonster(int i){ ... }
    void setMonsterExist(int i, bool exist){ ... }
    int getCountMonsters(){ ... }
    float getDeltaTime(){ ... }
    void addDeltaTime(float deltaTime){ ... }
    void resetDeltaTime(){ ... }
    void setMonstersDown(){ ... }
    void clear(){ ... }
};
```

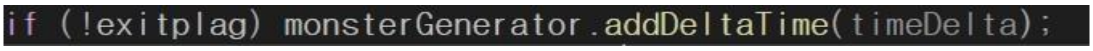
A variable *monsters* save the generated red balls.

A variable *deltatime* save the time for generating new balls.

1.check the deltatime.

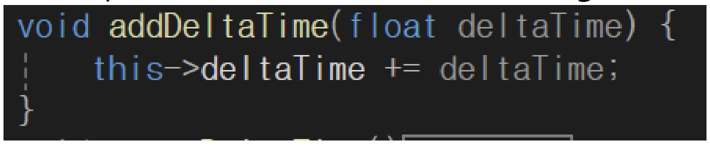
For checking the deltatime. we use the value of *'timeDelta'* of function display. it means the time between the current image frame and the last image frame.

we call the method *'addDeltatime'* when the function *'display'* is called.



```
if (!exitplag) monsterGenerator.addDeltaTime(timeDelta);
```

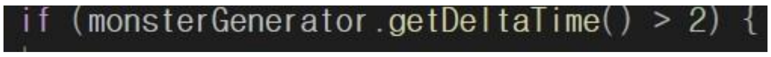
It is implemented in the class '*Monstergenerator'*.



```
void addDeltaTime(float deltaTime) {
    this->deltaTime += deltaTime;
}
```

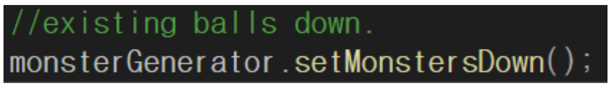
2.if deltatime >2 , set lower existing balls.

check the deltatime by getter and check its value is over 2,



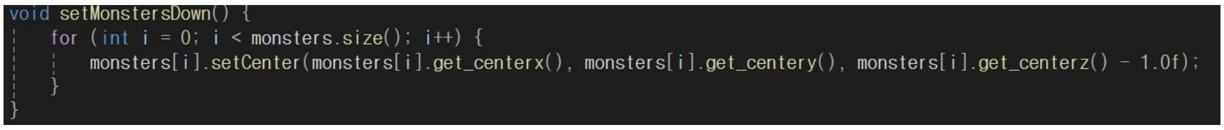
```
 if (monsterGenerator.getDeltaTime() > 2) {
```

if it is true, execute below code.



```
//existing balls down.
monsterGenerator.setMonstersDown();
```

it is implemented code in class *Monstergenerator*

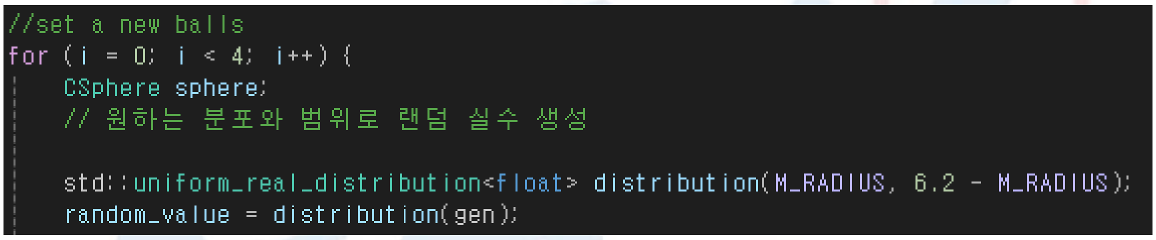


```
void setMonstersDown() {
    for (int i = 0; i < monsters.size(); i++) {
        monsters[i].setCenter(monsters[i].get_centerx(), monsters[i].get_centery(), monsters[i].get_centerz() - 1.0f);
    }
}
```

It sets lower existing balls *center_z* values 1.0f by using the function '*setCenter*' in cass *CSphere*.

3.randomly pick 4 x_values.

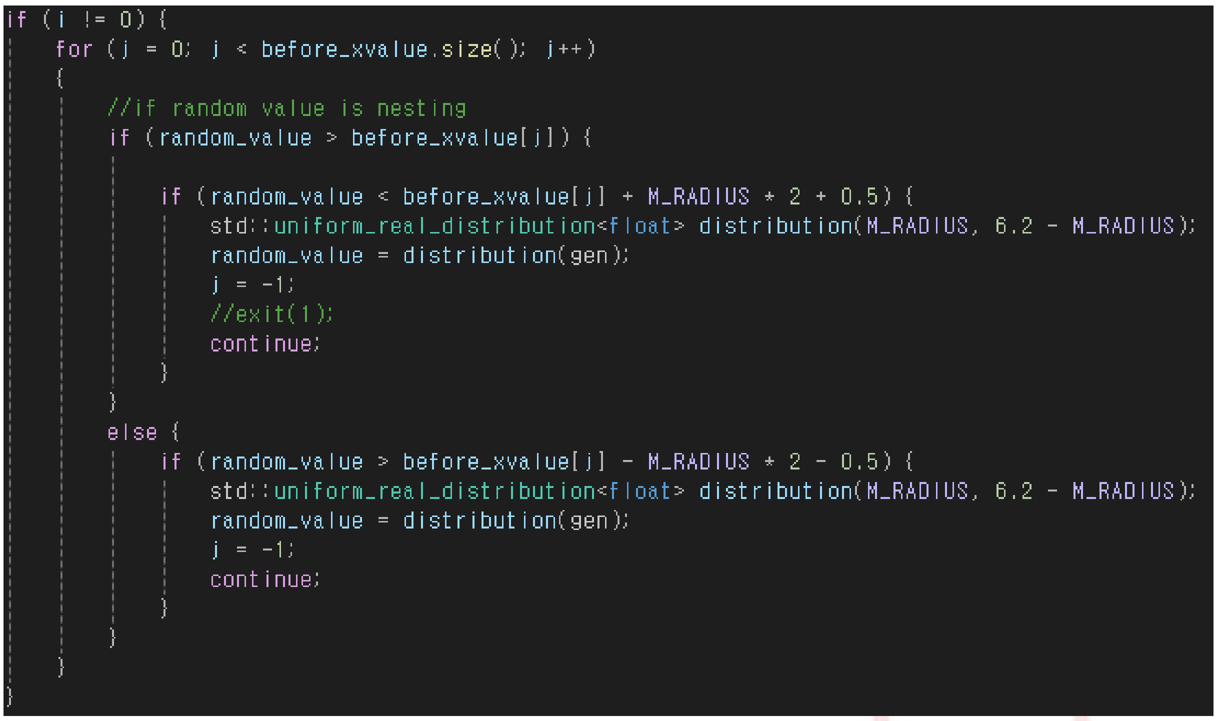
After that, prepare to set new red balls.



```
//set a new balls
for (i = 0; i < 4; i++) {
    CSphere sphere;
    // 원하는 분포와 범위로 랜덤 실수 생성

    std::uniform_real_distribution<float> distribution(M_RADIUS, 6.2 - M_RADIUS);
    random_value = distribution(gen);
```

First, write a for loop to create a total of four balls on one line. Generate a randomvalue within the range of greater than *M_RADIUS* and less than (6.2 - *M_RADIUS*).



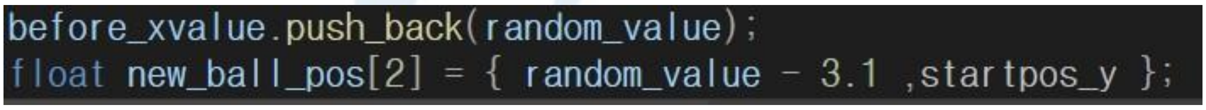
```
if (i != 0) {
    for (j = 0; j < before_xvalue.size(); j++)
    {
        //if random value is nesting
        if (random_value > before_xvalue[j]) {

            if (random_value < before_xvalue[j] + M_RADIUS * 2 + 0.5) {
                std::uniform_real_distribution<float> distribution(M_RADIUS, 6.2 - M_RADIUS);
                random_value = distribution(gen);
                j = -1;
                //exit(1);
                continue;
            }
        }
        else {
            if (random_value > before_xvalue[j] - M_RADIUS * 2 - 0.5) {
                std::uniform_real_distribution<float> distribution(M_RADIUS, 6.2 - M_RADIUS);
                random_value = distribution(gen);
                j = -1;
                continue;
            }
        }
    }
}
```

To prevent overlapping of red balls, we calculated the distance between the current random value and the previously generated random values. Implemented a conditional statement with an if-else structure that checks if two red balls overlap. If the condition is met, the random value is regenerated, and the comparison is performed again.

4. set new balls on x_values.

After section 3, we can get the random float value. Then, We will create new balls.

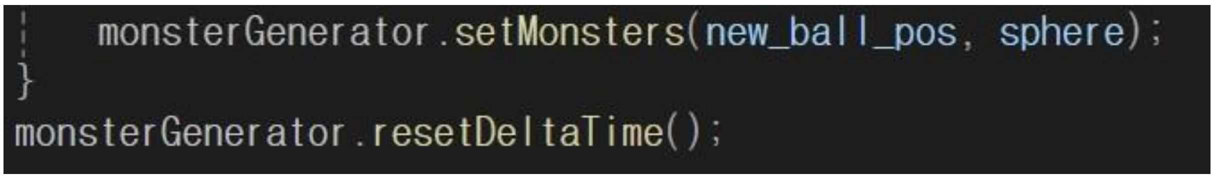


```
before_xvalue.push_back(random_value);
float new_ball_pos[2] = { random_value - 3.1 ,startpos_y };
```

First, we save the *random_value* for avoiding overlap by push the vector '*before_xvalue*'.

and set the pos x is (*random_value*-3.2) and the pos y is variable *'startpos_y'*

we pick the number of the range(0~6.2). but real board's x range is -3.1~3.1. so we subtract 3.1 from the random value.



```
    monsterGenerator.setMonsters(new_ball_pos, sphere);
}
monsterGenerator.resetDeltaTime();
```

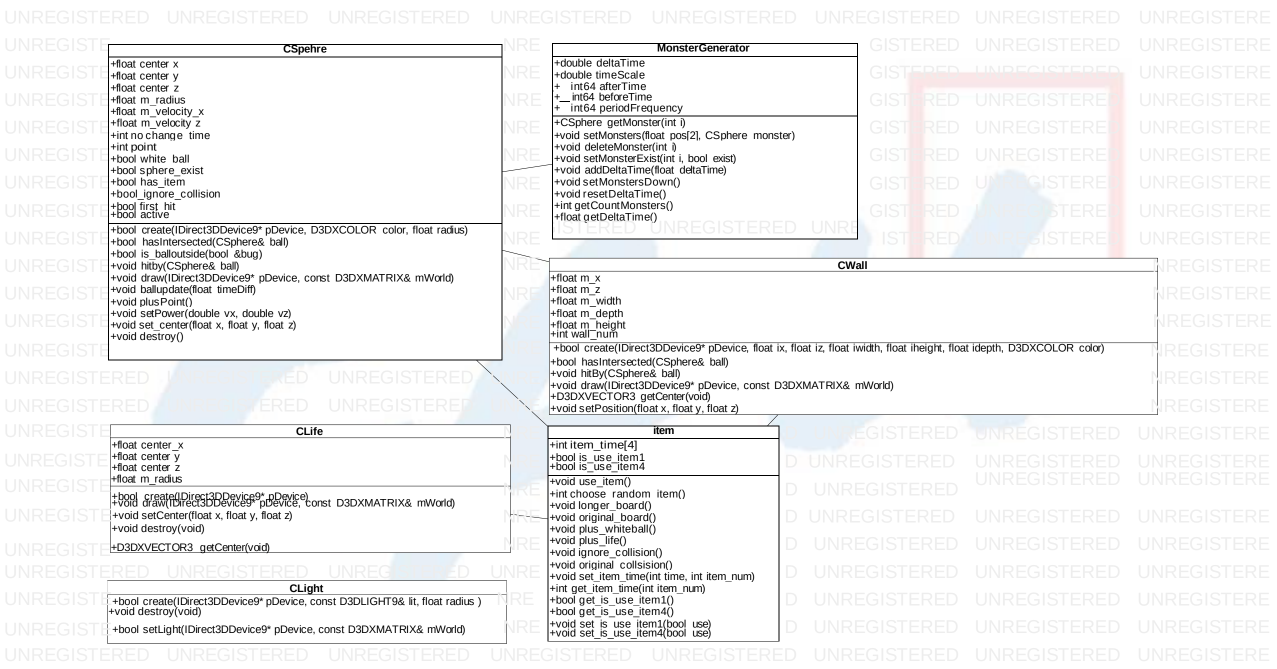
and create new monsters by *new_ball_pos*, and sphere.

Lastly, reset the *deltatime* because we should check the time newly.

7) White ball's initial position

Code was added to the *CALLBACK* class so that the white ball moves with the player bar when the right cursor is pressed. However, when the white ball is launched from the player bar, it should not move under the influence of the right cursor. Therefore, we added active of bool type as a member variable to the *CSphere* class and set active to false when the white ball is launched by pressing the space bar, so that it is not affected by the right cursor.

V. SW design result(diagram)

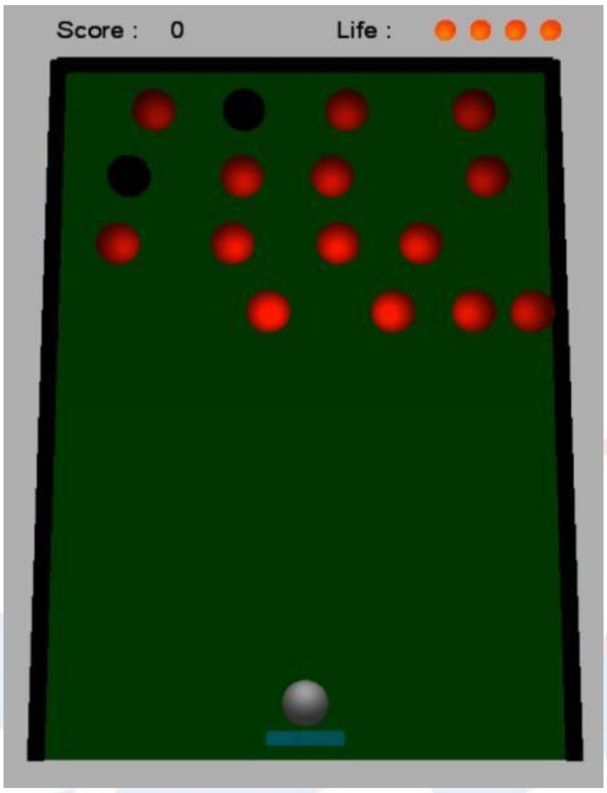

CSpehre
+float center x
+float center y
+float center z
+float m_radius
+float m_velocity_x
+float m_velocity z
+int no change time
+int point
+bool white ball
+bool sphere_exist
+bool has_item
+bool_ignore_collision
+bool first hit
+bool active
+bool create(IDirect3DDevice9* pDevice, D3DXCOLOR color, float radius)
+bool hasIntersected(CSphere& ball)
+bool is_balloutside(bool &bug)
+void hitby(CSphere& ball)
+void draw(IDirect3DDevice9* pDevice, const D3DXMATRIX& mWorld)
+void ballupdate(float timeDiff)
+void plusPoint()
+void setPower(double vx, double vz)
+void set_center(float x, float y, float z)
+void destroy()
MonsterGenerator
+double deltaTime
+double timeScale
+ int64 afterTime
+__int64 beforeTime
+ int64 periodFrequency
+CSphere getMonster(int i)
+void setMonsters(float pos[2], CSphere monster)
+void deleteMonster(int i)
+void setMonsterExist(int i, bool exist)
+void addDeltaTime(float deltaTime)
+void setMonstersDown()
+void resetDeltaTime()
+int getCountMonsters()
+float getDeltaTime()
CWall
+float m_x
+float m_z
+float m_width
+float m_depth
+float m_height
+int wall_num
+bool create(IDirect3DDevice9* pDevice, float ix, float iz, float iwidth, float iheight, float idepth, D3DXCOLOR color)
+bool hasIntersected(CSphere& ball)
+void hitBy(CSphere& ball)
+void draw(IDirect3DDevice9* pDevice, const D3DXMATRIX& mWorld)
+D3DXVECTOR3 getCenter(void)
+void setPosition(float x, float y, float z)
CLife
+float center_x
+float center y
+float center z
+float m_radius
+bool create(IDirect3DDevice9* pDevice)
+void draw(IDirect3DDevice9* pDevice, const D3DXMATRIX& mWorld)
+void setCenter(float x, float y, float z)
+void destroy(void)
+D3DXVECTOR3 getCenter(void)
CLight
+bool create(IDirect3DDevice9* pDevice, const D3DLIGHT9& lit, float radius)
+void destroy(void)
+bool setLight(IDirect3DDevice9* pDevice, const D3DXMATRIX& mWorld)
item
+int item_time[4]
+bool is_use_item1
+bool is_use_item4
+void use_item()
+int choose random item()
+void longer_board()
+void original_board()
+void plus_whiteball()
+void plus_life()
+void ignore_collision()
+void original collsision()
+void set_item_time(int time, int item_num)
+int get_item_time(int item_num)
+bool get_is_use_item1()
+bool get_is_use_item4()
+void set is use item1(bool use)
+void set_is_use_item4(bool use)

(DirectX-related properties were omitted from the class diagram, and setters and getters to change properties were omitted from the method.)

VI. Execution Result

< General >

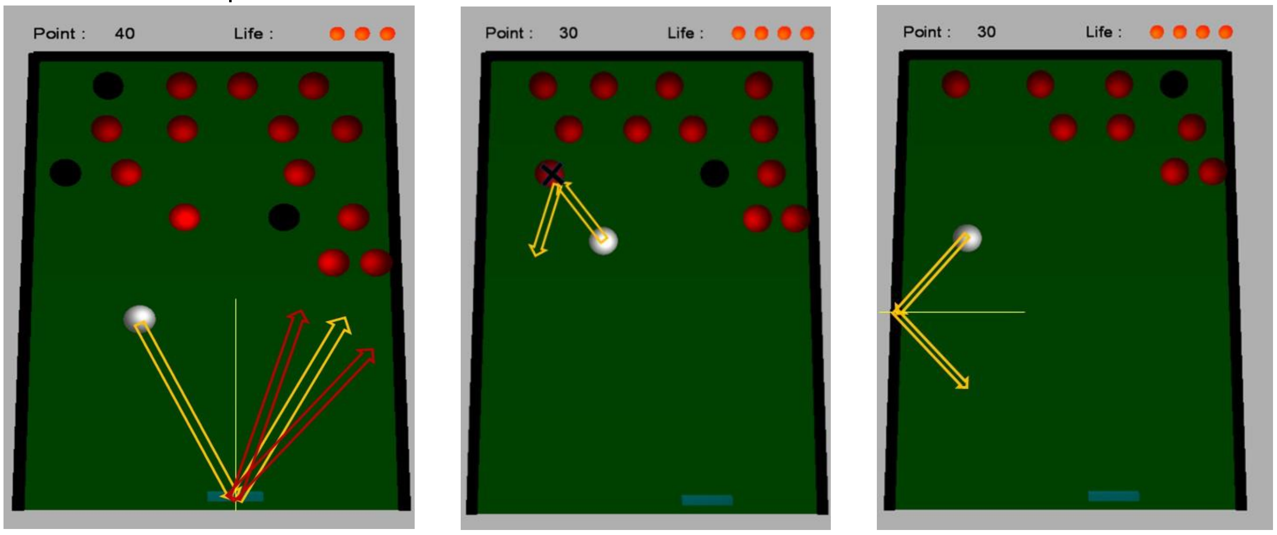
1) Appearance



2) Collision

- **between Bar & White ball**
- **between White ball & Red ball(block)**
- **between White ball & Wall**

*Collision example

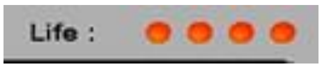


<Left: Collision with the user bar | Middle: Collision with the ball | Right: Collision with the wall>

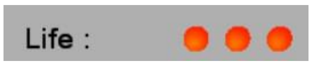
< Additional >

1) Life

■ **Life Graphic**

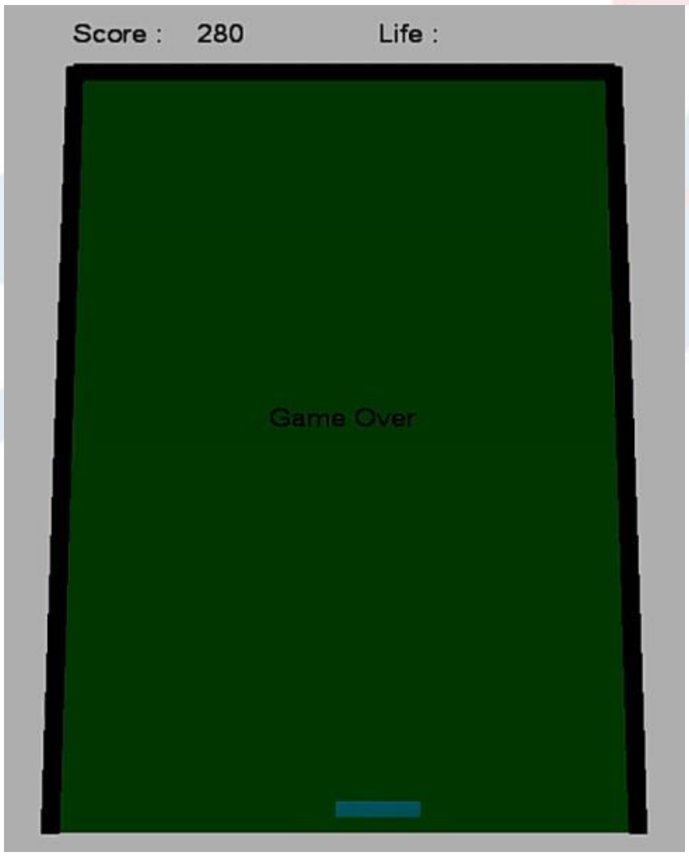


Life points is represented in Graphic UI.

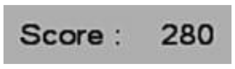


When the user fails to bounce the ball, that is, when the ball falls to the bottom of the table, one life point is deducted.

■ **Game over conditions**

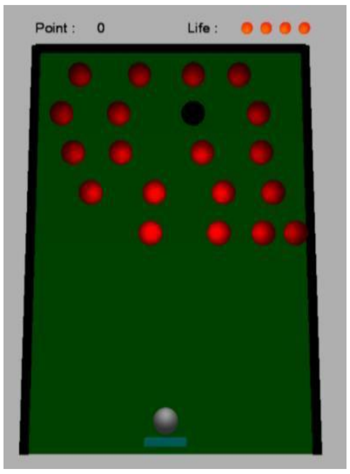


2) Score



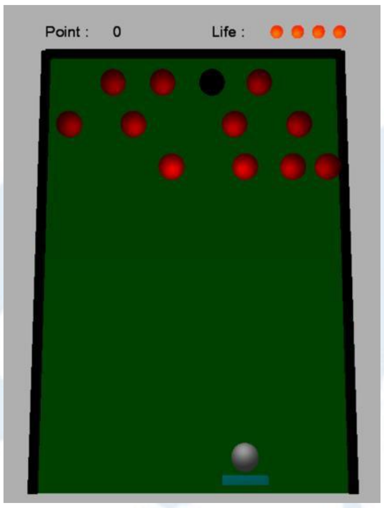
The score was located in the upper left corner of the execution screen.

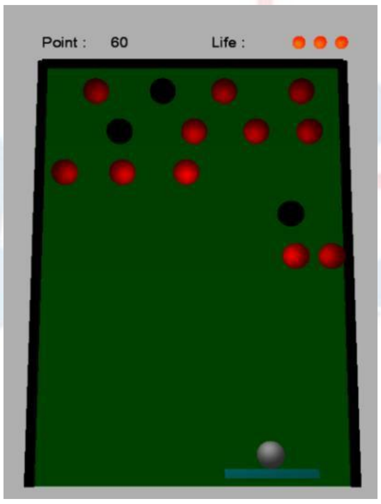
3) Player Bar



4) Item

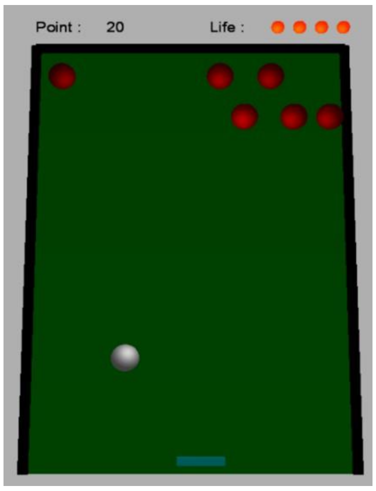
■ **Longer player bar**

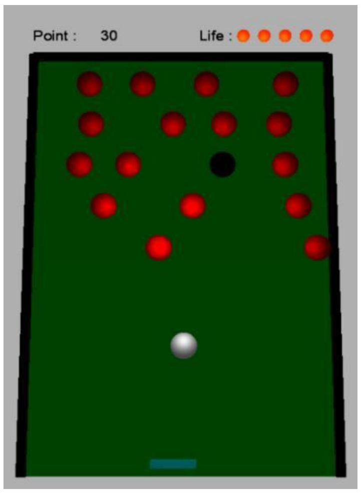




<Left - Default, Right - User bar length increase item applied>

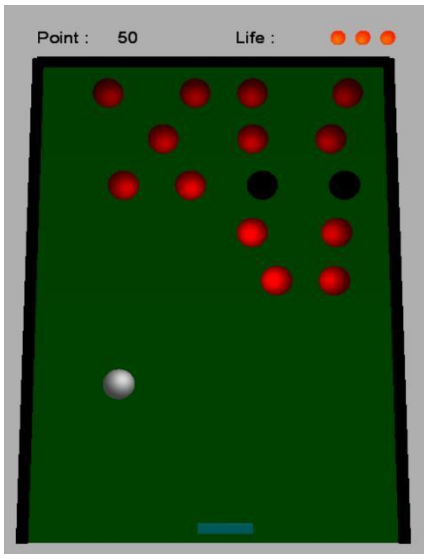
■ **Additional life**

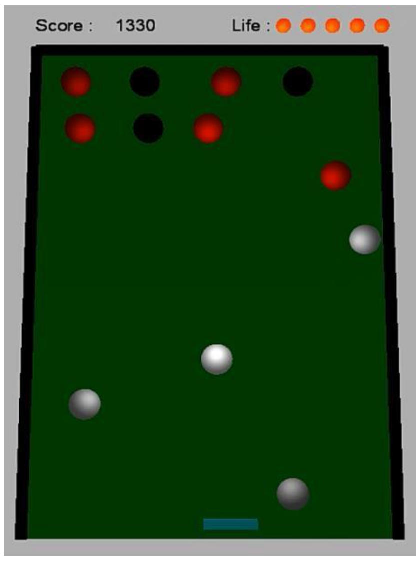




<Left - Default, Right - - Additional life increase item applied>

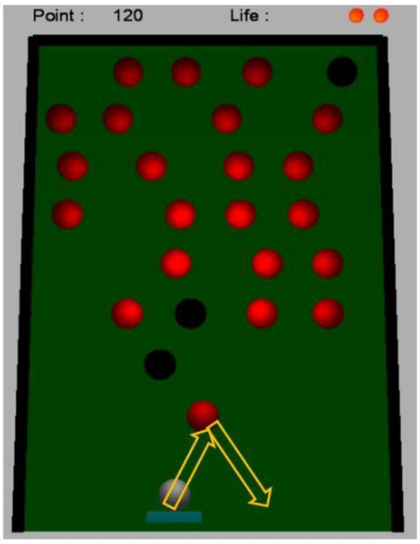
- **Additional white ball**

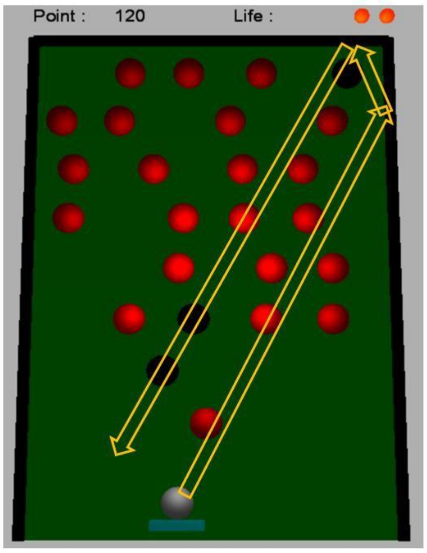




<Left - Default, Right - Additional white ball item applied>

- **Collision ignoring**



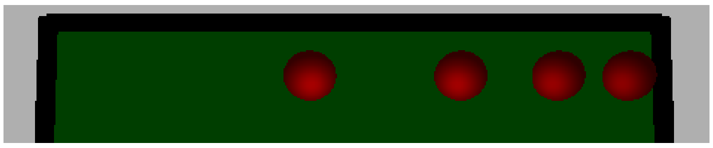


<Left - Default ball movement, Right - Movement after consuming an item>

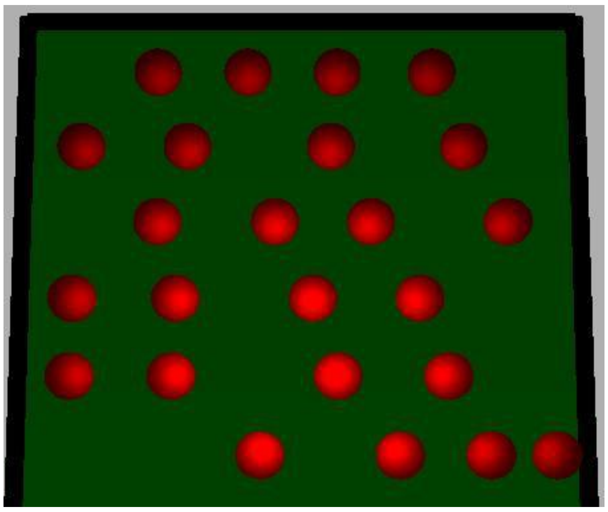
-

5) Red ball coming down(increasing)

➔ **Set new balls and lower existing balls when 2 sec has elapsed.**



After 2 seconds since starting 'ARKANOID,' the existing line descends one step, and a new line is generated at the top.



6) White ball's initial position

VII. Object Oriented Programming

We devided the program components into various objects which are *CSPhere* object, *CWall* object, *CLight* object, *CLife* object, *Item* object, and *MonsterGenerator* object. Each object works by interacting and communicating with other objects.
Also we implemented additional features by using the extension of methods and properties in the class. Since it takes a lot of time for many people to write and integrate their own code, we divided the code into as many classes as possible. Therefore, if each object only understands what role it plays and when and what functions it uses, all team members can understand the flow, making development easier.

- It was a good chance to make a program or game that is visible.
- It was certainly convenient in terms of usability and organicity to incorporate OOP, and each role was clearly separated, making it easy to read, interpret, and modify the code.
- There are so many different bugs in the game, and We found it very difficult to catch them.
- When developing a program for multiple people at the same time, we felt it was important to predetermine and work on code styles such as global variable usage rules, class writing rules, and variable name declaration rules before writing the code. There are a wide variety of conflicts in the process of merging codes written by multiple people, and the time it takes to resolve these conflicts is enormous. The rules set before writing the code will reduce the cost of writing these codes, so the next time we collaborate, we should set the rules before developing the code.

VIII. Conclusion

In this project, we made an ARKANAOID game using the directX program. Our six team members were divided into three teams of two each, and each wrote the code for their class. The object-oriented programming concept was used to allow the classes created by each team to operate organically. In addition to the functions presented as conditions, various functions such as items, ball shape changes, and additional balls were added to make the game more fun. Although it was difficult to merge because many people had to write large amounts of code, object-oriented programming eventually made the game complete.