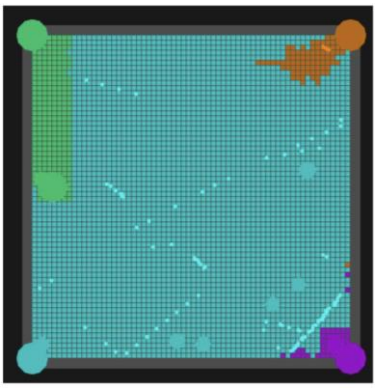
Multiply or Release Report

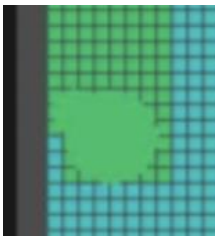
111550176 陳湛宇、111550088 張育維、111550165 吳宗樺

Introduction and Motivation

The "Multiply or Release" project is inspired by an animation observed on YouTube, captivated by its simulation of particle dynamics. This project explores particle manipulation and collision dynamics within a 2D grid referred to as the "playground," with four players each occupying a distinct corner. The core mechanics involve players using control particles on the "control board" to trigger actions such as firing bullets or cannons and reinforcing shields on the playground. Additionally, we've added user interactive features such as mouse clicking on playground to generate "Blackholes" and keyboard inputs to apply force field.

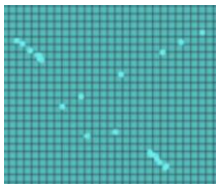
Playground





Shield: The big balls centered at the 4 corners of playground.

Cannon: The bigger painting object released from 4 corners.(First picture)



Bullet: The smallest painting object released from 4 corners.(Second picture)

Squares: Hold one of the 4 colors. The grid is a 64 x 64 square array.

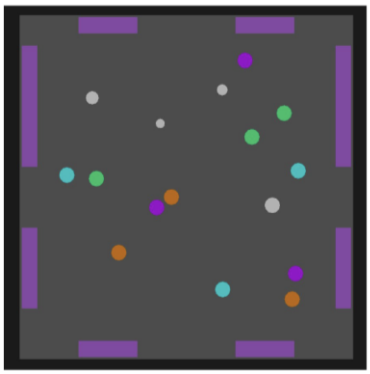
Blackhole: Controlled by the user. Can attract the players' cannons.(Third picture)

Control panel

Trigger the players' behavior on the playground.

1. Multipliers : x2, x4, x8, x16
2. Control balls : at most 3 for each player

Vertical bar	Horizontal bar
1. Reinforce shield	5. add control ball
2. Scatter shot	6. sqrt(control.value)
3. Big shot	7. ghost ball
4. Free sqrt big shot	8. modify gravity constant



Fundamentals

Here is how we implement each physical phenomenon shown in the simulation.

Ball Movement:

$x \mathrel{+}= v * dt$

$v \mathrel{+}= a * dt$

Collision Detection:

Detection method: distance between two balls is small enough and they are heading in each other.

1. ball against ball: $|\text{position difference}|^2 < (r1+r2)^2$
2. ball against wall: given direction and position, determine whether the above two conditions are true

<u>Handle Collision:</u>

1. For dynamic ball against dynamic ball:

swap the two balls' velocity

2. For dynamic ball against a wall:
   reverse normal component
3. For dynamic ball against stationary ball (shield):
   project velocity to the direction of the unit vector in the direction of position difference as vector v, velocity of the dynamic ball -= 2v

Gravity field:

The gravity only exists in the playground.

The formula to compute gravity is $Gm1m2/r^2$, just the same as how we normally compute it.

- G may change when control ball trigger certain zones
- m is the square root of the ball's value
- r is the distance in OpenGL coordinate

Gravity field is applied to:

- Between cannons
- Between cannons and the blackholes
- Outer gravity field: none / up / down / left / right

Newton's second law of motion, F = ma, is used to calculate the acceleration of an object.

Implementation

Basic feature in multiply or release

Balls movement

First, for each ball, compute the movement at the instant with the formula x = v*t. Next, determine whether it is colliding with walls or other balls. If it is, modify its velocity. The following sample shows how the multipliers are managed in the control board.

```
void moveMultiplierBalls()
{
    for (int i = 0; i < 4; i++)
    {
        multipliers[i].move();
        if (multipliers[i].collideLeftWall(-0.885) || multipliers[i].collideRightWall(-0.115)) {
            glm::vec3 vel = multipliers[i].getVelocity();
            multipliers[i].setVelocity(glm::vec3(-vel.x, vel.y, vel.z));
        }
        if (multipliers[i].collideTopWall(0.76) || multipliers[i].collideBottomWall(-0.76)) {
            glm::vec3 vel = multipliers[i].getVelocity();
            multipliers[i].setVelocity(glm::vec3(vel.x, -vel.y, vel.z));
        }

        for (int j = i + 1; j < 4; j++)
            multipliers[i].collideBall(multipliers[j]);
    }
}
```

Collision between balls

First, we should determine whether the collision happens. The basic idea is to check the distance between two balls are short enough. The coefficient in front of the x-component is because I don't know how to set the vertices to draw polygons properly, and I just modify the coefficients or the constants to make it look better. If the collision actually occurs, swap the velocity of the two balls.

```
bool Ball::collideBall(Ball& ball)
{
    glm::vec3 posDiff = this->translation - ball.translation;
    float dist = this->radius + ball.radius;
    if (4.0 * posDiff.x * posDiff.x + posDiff.y * posDiff.y > dist * dist)
        return false;

    glm::vec3 t = ball.velocity;
    ball.velocity = this->velocity;
    this->velocity = t;
    return true;
}
```

Gravity between two balls

For each two balls, first, we compute the position difference and the direction. Next, we calculate the additional force they should have by applying the formula $Gm1m2/r^2$, and apply a = F / m to compute the acceleration. Finally, use the addAcceleration method to modify the ball's acceleration.

```
void Player::computeGravity(Player& p)
{
    for (Ball& cannon1 : this->cannons)
        for (Ball& cannon2 : p.cannons)
        {
            glm::vec3 posDiff = cannon1.getPosition() - cannon2.getPosition();
            glm::vec3 direction = glm::normalize(posDiff);
            float length = posDiff.x / direction.x;
            // float F = G * cannon1.getValue() * cannon2.getValue() / length / length;
            // a1 = direction * (F / cannon1.getValue());
            // a2 = direction * (F / cannon2.getValue());
            float t = gravityConstant / length / length;
            cannon1.addAcceleration(-direction * (float)(t * sqrt(cannon2.getValue())));
            cannon2.addAcceleration( direction * (float)(t * sqrt(cannon1.getValue())));
        }
}
```

Collision with wall

First, determine collision depending on the distance between the ball and the wall, if the distance is small enough, and the direction of the ball's velocity points towards the wall, the collision occurs. Take the left wall as an example, check if the ball's position on x minus its radius smaller than the x-value of the right side of the wall and the ball is moving left, or the x-component of its velocity is < 0. Similar for the other three walls.

```
bool Ball::collideLeftWall(float pos) const
{
    return this->translation.x - this->radius * 0.6 <= pos
        && this->velocity.x < 0.0;
}

bool Ball::collideRightWall(float pos) const
{
    return this->translation.x + this->radius * 0.6 >= pos
        && this->velocity.x > 0.0;
}

bool Ball::collideTopWall(float pos) const
{
    return this->translation.y + this->radius >= pos
        && this->velocity.y > 0.0;
}

bool Ball::collideBottomWall(float pos) const
{
    return this->translation.y - this->radius <= pos
        && this->velocity.y < 0.0;
}
```

When a control particle collides with the left wall, multiply the x-component of the

velocity by -1, making the direction on x opposite. Next, we get the y position and radius of this ball, using them to determine whether one of the two zones on the left wall is triggered. If the upper one is hit, reinforce the corresponding player's shield by adding the control ball's value to it and reset the value of this ball as 2. If the lower one is hit, generate shower shot consisting of a set of bullets and reset the value of the control ball.

```
if (controlBall.collideLeftWall(-0.885))
{
    glm::vec3 vel = controlBall.getVelocity();
    controlBall.setVelocity(glm::vec3(-vel.x, vel.y, vel.z));

    float y = controlBall.getPosition().y;
    float r = value_radius(controlBall.getValue());
    if (0.1 <= y - r && y + r <= 0.7)
    {
        reinforceShield(controlBall.getValue());
        controlBall.setValue(2);
    }
    else if (-0.6 <= y - r && y + r <= -0.2)
    {
        showerShot(controlBall.getValue());
        controlBall.setValue(2);
    }
}
```

When a control particle collides with the right wall, reverse the velocity in x-direction. Get the y position and radius of this ball and use them to determine whether one of the two zones on the right wall is triggered. If the upper one is hit, generate a cannon of value equal to the control ball's value and set the value of this ball as 2. If the lower one is hit, produce a cannon as well but with value equal to the square root of the control ball's value. As a bonus, don't reset the control ball's value this time.

```
if (controlBall.collideRightWall(-0.115))
{
    glm::vec3 vel = controlBall.getVelocity();
    controlBall.setVelocity(glm::vec3(-vel.x, vel.y, vel.z));

    float y = controlBall.getPosition().y;
    float r = value_radius(controlBall.getValue());
    if (0.1 <= y - r && y + r <= 0.7)
    {
        bigShot(controlBall.getValue());
        controlBall.setValue(2);
    }
    else if (-0.6 <= y - r && y + r <= -0.2)
    {
        freeSqrtBigShot(controlBall.getValue());
    }
}
```

When a control particle collides with the top wall, reverse the velocity in y-direction. Then, get the x position and radius of this ball, use them to determine whether one of the two zones on the top wall is triggered. If the left one is hit and the player has fewer than 3 control balls, add one control ball for the player on control panel. If the right one is hit, set the value of the control ball to be the squared root of the original value. To maintain the value to be a power of 2, the modified method is 2 ^ ceil (log2(value)/2).

```
if (controlBall.collideTopWall(0.76))
{
    glm::vec3 vel = controlBall.getVelocity();
    controlBall.setVelocity(glm::vec3(vel.x, -vel.y, vel.z));

    float x = controlBall.getPosition().x;
    if (-0.775 <= x && x <= -0.625)
    {
        // at most 3 control balls
        if (controlBalls.size() < 3) addControlBall();
    }
    else if (-0.375 <= x && x <= -0.225)
    {
        // value = 2 ^ ceil( log2(value)/2 )
        unsigned int power = glm::ceil<int>(log2(controlBall.getValue()) / 2.0);
        controlBall.setValue(1U << power);
    }
}
```

When a control particle collides with the bottom wall, reverse the velocity in y-direction. Then, get the x position and radius of this ball, use them to determine whether one of the two zones

on the bottom wall is triggered. If the left one is hit, set the control ball to be a "ghost ball", so that the ball will collide with only multiplier balls, and thus increase the probability of achieving a larger value. If the right one is hit, increase the gravityConstant by 0.0025. If the gravity constant grows too large, set it as -0.025. Otherwise, the speed of some cannons might become too fast and escape from the playground.

```
if (controlBall.collideBottomWall(-0.76))
{
    glm::vec3 vel = controlBall.getVelocity();
    controlBall.setVelocity(glm::vec3(vel.x, -vel.y, vel.z));

    float x = controlBall.getPosition().x;
    if (-0.775 <= x && x <= -0.625)
    {
        // ghost ball : speed up 1.2x & have no collision with others
        // if the control ball is a ghost ball, set it to be not, and vice versa
        controlBall.swapMayCollide();
    }
    else if (-0.375 <= x && x <= -0.225)
    {
        // increase gravity : -0.025 ~ 0.025
        gravityConstant += 0.0025;
        if (gravityConstant > 0.025)
            gravityConstant = -0.025;
    }
}
```

In the playground, to handle the collision with wall is simpler because there is no other function to trigger except for changing velocity. The way to modify the velocity is the same as in the control panel.

```
void Player::collidePlaygroundWall(Ball& cannon)
{
    if (cannon.collideLeftWall(0.1))
    {
        glm::vec3 vel = cannon.getVelocity();
        cannon.setVelocity(glm::vec3(-vel.x * 1.02, vel.y * 1.02, vel.z));
    }
    if (cannon.collideRightWall(0.9))
    {
        glm::vec3 vel = cannon.getVelocity();
        cannon.setVelocity(glm::vec3(-vel.x * 1.02, vel.y * 1.02, vel.z));
    }
    if (cannon.collideTopWall(0.8))
    {
        glm::vec3 vel = cannon.getVelocity();
        cannon.setVelocity(glm::vec3(vel.x * 1.02, -vel.y * 1.02, vel.z));
    }
    if (cannon.collideBottomWall(-0.8))
    {
        glm::vec3 vel = cannon.getVelocity();
        cannon.setVelocity(glm::vec3(vel.x * 1.02, -vel.y * 1.02, vel.z));
    }
}
```

Ball movement on the playground

As the players' cannons and bullets move across the playground over time, they encounter some squares of different color and they set the color of those squares to be the same as themselves and, in exchange, their value and radius will be decreased. As the ball's value becomes 0, it vanishes. The method occupySquares shown below is created to achieve this.

For all cannons belonging to a player, get its center point and radius, calculate the boundaries of the area it covers currently, and translate them into horizontal and vertical indices. For all squares contained in the area, if the color is not the same as the cannon or the bullet, set its color as mentioned above until the cannon or the bullet "runs out of its energy".

```
void Player::occupySquares(Rectangle* grid)
{
    for (int ic = 0; ic < cannons.size();)
    {
        Ball& cannon = cannons[ic];
        glm::vec3 center = cannon.getPosition();
        float radius = cannon.getRadius(),
            left = center.x - radius * 0.5, right = center.x + radius * 0.7,
            top = center.y + radius, bottom = center.y - radius * 1.35;
        unsigned int value = cannon.getValue();

        // h : horizontal index, v : vertical index
        // 0.5 + 8/640.0 * (h-32) > left , h > (left -0.5) * 80 + 32
        // 0.5 + 8/640.0 * (h-32) < right, h < (right-0.5) * 80 + 32
        // 8/320.0 * (32-v) < top        , v < 32 - top    * 40
        // 8/320.0 * (32-v) > bottom     , v > 32 - bottom * 40
        int l = glm::max(glm::ceil<int>((left - 0.5) * 80 + 32), 0.0);
        int r = glm::min(glm::floor<int>((right - 0.5) * 80 + 32), 64.0);
        int t = glm::max(glm::floor<int>(32 - top * 40), 0.0);
        int b = glm::min(glm::ceil<int>(32 - bottom * 40), 64.0);
```

```
for (int i = t; i < b && value > 0; i++)
{
    for (int idx = i * 64 + l; idx < i * 64 + r && value > 0; idx++)
    {
        glm::vec3 posDiff = center - grid[idx].getPosition();
        if (this->color != grid[idx].getColor())
        {
            grid[idx].setColor(this->color);
            value--;
        }
    }
}

if (value <= 0)
    cannons.erase(cannons.begin() + ic);
else
{
    cannon.setRadius(value_radius(value));
    cannon.setValue(value);
    ic++;
}
```

Because the bullet will disappear as soon as it paints on the grid, if the grid color doesn't equal to its color, then we change the grid's color, and make this bullet disappear, or inactive, and number of active bullets – 1. There is no need to compute the radius again.

If the value of a cannon or number of active bullets in a bullet set becomes zero, it is erased instantly. Otherwise, the complexity will grow very fast in the simulation.

```
unsigned int n = bullets[i].size();
for (unsigned int j = 0; j < n; j++)
    if (active[i][j])
    {
        glm::vec3 center = bullets[i][j].getPosition();
        int h = (center.x - 0.5) * 80 + 32;
        int v = 32 - center.y * 40;
        if (h > 63) h = 63;
        else if (h < 0) h = 0;
        if (v > 63) v = 63;
        else if (v < 0) v = 0;
        int idx = v * 64 + h;
        if (grid[idx].getColor() != this->color)
        {
            grid[idx].setColor(this->color);
            active[i][j] = false;
            numActiveBullets[i]--;
        }
    }

if (numActiveBullets[i] == 0)
{
    bullets.erase(bullets.begin() + i);
    active.erase(active.begin() + i);
    numActiveBullets.erase(numActiveBullets.begin() + i);
}
else i++;
```

User interactive

1. Blackhole:

To detect when to add a blackhole, we add a function called add_black_hole, which will get user cursor position. When left mouse button is pressed, get the cursor position, and change it into the OpenGL coordinate. But, if it is out of the range of the playground, no blackhole will be generated. Next, print the message on the terminal to tell user that this position is available for blackhole, and create the blackhole object, change the call back function to remove_black_hole with glfwSetMouseButtonCallback, because what we want to detect becomes when to remove the blackhole.

```
void add_black_hole(GLFWwindow* window, int button, int action, int mods)
{
    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS)
    {
        double xpos, ypos;
        glfwGetCursorPos(window, &xpos, &ypos);
        xpos = (2.0f * xpos) / windowWidth - 1.0f;
        ypos = 1.0f - (2.0f * ypos) / windowHeight;
        if (xpos < 0.075 || xpos > 0.925 || ypos < -0.85 || ypos > 0.85)    //playground range
        {
            return;
        }
        std::cout << "The blackhole is placed at (" << xpos << ", " << ypos << ")\n";
        blackhole = new Ball(16384, glm::vec3(xpos, ypos, -0.002), glm::vec3(0.0), glm::vec3(0.0), glm::vec4(0.0, 0.0, 0.0, 1.0));
        glfwSetMouseButtonCallback(window, remove_black_hole);
    }
}
```

Now that the blackhole can be generated, we still have to implement the function to remove the blackhole, because what we desire is that when user holds left-click on mouse, the blackhole exists, and as long as user release it, the blackhole disappears. What to do here is much simpler than generating a blackhole. Once the left button is released, we change the blackhole to a NULL pointer, print the message for user that blackhole has been removed, and change the call back function to add_black_hole, which is needed for next blackhole detection.

```
void remove_black_hole(GLFWwindow* window, int button, int action, int mods)
{
    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_RELEASE)
    {
        blackhole = NULL;
        std::cout << "The blackhole has been removed.\n";
        glfwSetMouseButtonCallback(window, add_black_hole);
    }
}
```

If a blackhole exists, it will affect the behavior of other balls on the playground such as the players' cannons. For simplicity, get the whole vector of a player's cannons on the playground as reference and change their acceleration. First, compute the direction pointed from the cannon to the blackhole. According to the gravity formula, we compute the acceleration as below, and use addAcceleration method to add this additional acceleration to the cannons.

```
if (blackhole)
{
    std::vector<Ball>& can = players[i].getCannon();
    for (int x = 0; x < can.size(); x++)
    {
        glm::vec3 diff = blackhole->getPosition() - can[x].getPosition();
        glm::vec3 dir = glm::normalize(diff);
        double length = diff.x / dir.x;
        double t = 0.001 / length / length;
        can[x].addAcceleration(dir * (float)(t * sqrt(blackhole->getValue())));
    }
}
```

2.Outer gravity field:

First, we define a function called change_gravity_direction, which will store the number representing direction of force field when specific number button is pressed. As in the following code, we change the variable gravity_dir according to which number button is pressed, and show the message to tell user the direction of current gravity field.

```
void change_gravity_direction(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    if (action == GLFW_PRESS)
    {
        switch (key) {
            case GLFW_KEY_1:
                std::cout << "change gravity from up to down.\n";
                gravity_dir = 1;
                break;
            case GLFW_KEY_2:
                std::cout << "change gravity from down to up.\n";
                gravity_dir = 2;
                break;
            case GLFW_KEY_3:
                std::cout << "change gravity from left to right.\n";
                gravity_dir = 3;
                break;
            case GLFW_KEY_4:
                std::cout << "change gravity from right to left.\n";
                gravity_dir = 4;
                break;
            case GLFW_KEY_0:
                std::cout << "no gravity.\n";
                gravity_dir = 0;
                break;
            default:
                break;
        }
```

Second, similar to what we do in blackhole, we change the acceleration of all cannons. According to the number, we define the unit vector representing force field direction, and apply formula **F = ma** to add additional acceleration to all cannons.

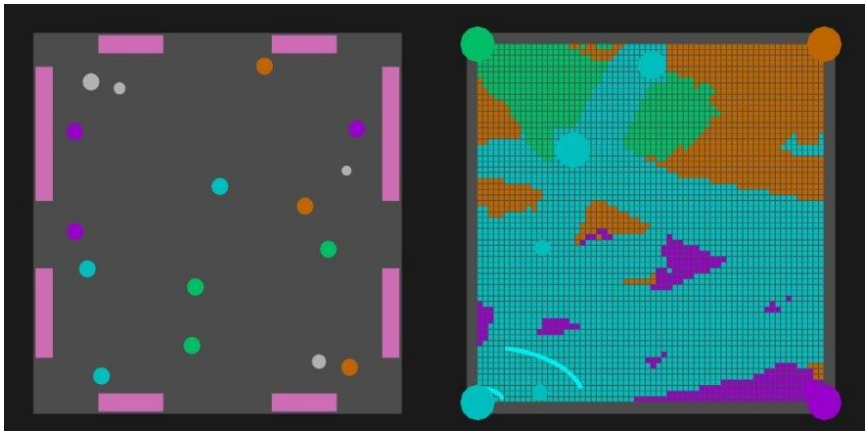
```
//additional gravity effect
if (gravity_dir == 1 || gravity_dir == 2 || gravity_dir == 3 || gravity_dir == 4)
{
    std::vector<Ball>& can = players[i].getCannon();
    if (gravity_dir == 1)
    {
        gravity = { 0.0, -1.0, 0.0 };
    }
    else if (gravity_dir == 2)
    {
        gravity = { 0.0, 1.0, 0.0 };
    }
    else if (gravity_dir == 3)
    {
        gravity = { 1.0, 0.0, 0.0 };
    }
    else if (gravity_dir == 4) {
        gravity = { -1.0, 0.0, 0.0 };
    }
    else
    {
        gravity = { 0.0, 0.0, 0.0 };
    }
    for (int x = 0; x < can.size(); x++)
    {
        can[x].addAcceleration(gravity / (float)(can[x].getValue()));
    }
}
```

Results and Analysis

Demo video link:

https://drive.google.com/file/d/1EH0DmTgYYlzMc93eAPnaYZOCJNFCUkeT/view?usp=sharing

(There is no reset, I just clip up the segment I like)



The result of our project is shown in this picture. The left is the control panel, and the right is the playground. They move fluently and all collisions are reasonable, and the way about painting, blackhole and gravity field all work properly.

The delta time is set to be 0.005. And the gravity constant is restricted in the range [–0.025, 0.025]. These restrictions are set to prevent the cannons from moving too fast and escaping the playground.

Discussion

This project aims to provide insights into the practical applications of OpenGL for simulation purposes, as well as deepening understanding of object-oriented programming and physics rules such as collision handling, etc. (mentioned above). The project's success will be measured by its ability to simulate realistic particle dynamics and provide an engaging user experience. Challenges may include optimizing the simulation for performance and ensuring accurate physics calculations.

We observed that the simulation's accuracy and realism significantly improved with an increase in delta time (dt). A smaller dt allowed for more precise calculations of particle dynamics and collision handling, leading to a more accurate representation of physical interactions. However, this increase in accuracy came at the cost of computational efficiency. With a smaller dt, the simulation required more computational power, resulting in lower frames per second (FPS) and a less smooth animation. This trade-off between accuracy and performance was a key challenge in optimizing the simulation.

Future Works

Given that this is our first experience using OpenGL, we were unable to explore every aspect as thoroughly as we would have liked. If given the opportunity, we hope to expand this project into a three-dimensional space in the future and incorporate more interactive features for users.

Additionally, we also hope to optimize the simulation's performance, providing a realistic depiction of particle dynamics, while not significantly compromising the smoothness of the animation. We could explore various strategies, such as adaptive time-stepping and other optimization techniques to improve the performance while maintaining a high level of accuracy.

Conclusion

The "Multiply or Release" project successfully implemented a 2D grid simulation, visualizing particle dynamics and collisions, while incorporating user interactions through mouse and keyboard inputs to generate "Blackholes" and force fields. This endeavor provided practical experience with OpenGL, deepened our understanding of object-oriented programming, and enhanced our knowledge of physics rules such as collision handling. Despite challenges in optimizing performance and ensuring accurate physics calculations, the project achieved its primary goals. As this was our first experience using OpenGL, we could not explore every aspect as thoroughly as desired. In the future, we aim to expand this project into a three-dimensional space and integrate more interactive features, thereby enriching the user experience further. Inspired by animations and driven by our learning objectives, this project laid a solid foundation for future advancements.