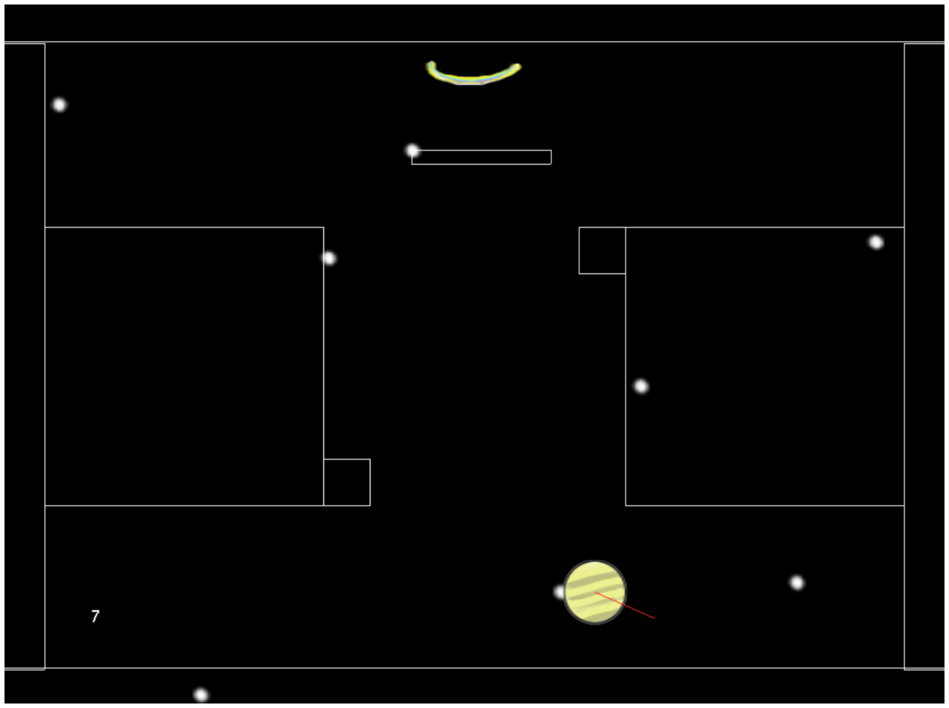
Intergalactic *"Big-Putt"* Documentation

By: Gleb Belyakov



Introduction

Welcome to *Intergalactic "Big-Putt"*! As a player, you command a laser that can apply great amounts of force to an asteroid if used correctly! The goal is to shoot the asteroid into a wormhole with as few shots as possible. But this isn't golf! Because we are in space where there is zero gravity! Shoot the asteroid across space and ricochet it off of obstacles in a convincing physics environment to score victorious!

Now let me explain my thought process and rationale behind implementing my various systems, there are **many** class scripts at play to make this game work correctly. I will briefly explain what executes in run-time within the GameTest.cpp file provided. I will also go into further detail regarding a few foundational systems that this game runs off of.

Game Initialization & Run-flow

As mentioned prior, there are many systems at play that allow this game to function. Throughout development of this project, I've decided to take an **object oriented** approach. As I feel this way the code would be a lot easier to maintain and expand upon if needed, this way it's easier to manage what data is accessible to other classes. Here is a brief rundown of what goes on under the hood.

Class Instances

- Create instance of "StarBackground" (animated background), "PlayerBall" (player character and controller), "SaturnHaloGoal" (worm-hole goal post) and the "GameManager" class (oversees background systems such as the level construction, and collision)

Start / Init()

- Initialize animated background sprites
- Initialize "PlayerBall" constructor (creating sprites and spawn location)
- Initialize "SaturnHaloGoal" constructor (again, creates sprites and spawn location)
- GameManager's BuildWorld method is called, constructing the first level and parsing necessary data to depending scripts (i.e setting the appropriate spawn locations for player, goal and ensuring the correct level is built)
- Play looping track

Game Run-time / Update()

- Call PlayerBall's local update method (handles real-time physics calculations, collision detection, and animations parsing deltaTime)
- Call SaturnHaloGoal local update method (handles animations and collision checks)
- Check if SaturnHaloGoal's BallCollisionCheck method returns true.
    - If so, call Gamamanger's NextLevel method and construct the next level
    - Reset the players score
    - Respawn Wormhole/SaturnHalo and Player to appropriate spawn locations (Handled by GameManager reading LevelData class)
- Check if GameManager returns true for player collision
    - If so, call the PlayerBall's Bounce method
- Call StarBackground's animate method

Visual Calls / Render()

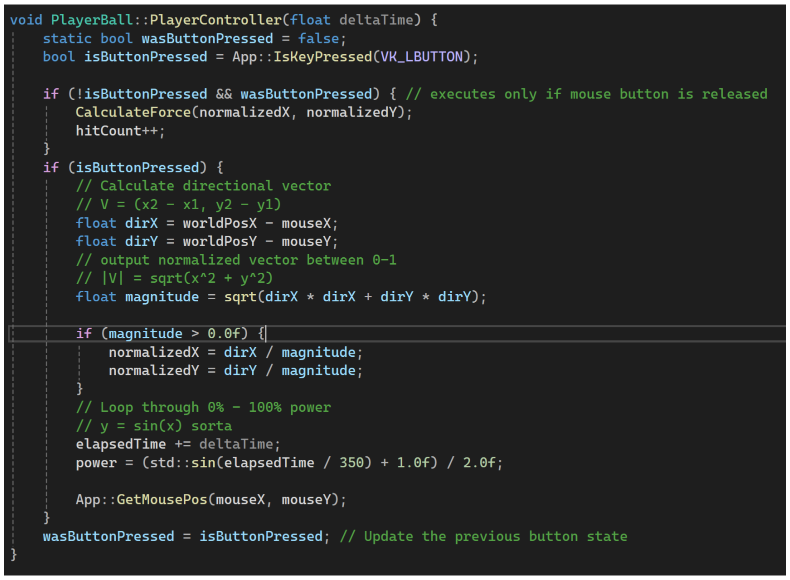
- StarBackground and SaturnHaloGoal render method
- PlayerBall's render method, DrawMouseLine method (for shooting mechanic explained later) and display player score
- GameManager draw walls, and "Thanks for Playing!" message if condition is met

“PlayerBall” Class

Overview:

For the player controller, I wanted to convince the player that they were in outer space. To accomplish this I’ve created a simple 2D physics simulation of “zero gravity-like” movement (displayed in Figure 2’s RigidBody method). Where I wanted to create an oscillating range of power at the player's disposal. Where if the mouse button is held the laser-beam generates a light ranging between white and red. Red is the most powerful force the player can apply to the ball or “asteroid”. I wanted to give the player full freedom in how they can manipulate the ball's movement, hence I allowed players to apply a force whilst the ball was still moving. Allowing players to strategize how they can manipulate the ball’s momentum to their advantage.
To accomplish this I’ve used the API’s ability to detect player input to oscillate incoming power to the ‘ApplyForce’ method. Also using basic **projectile motion** and **vector mathematics**, I am able to easily calculate the directional vector in which the ball will be pushed towards.

Small Code Snippets:



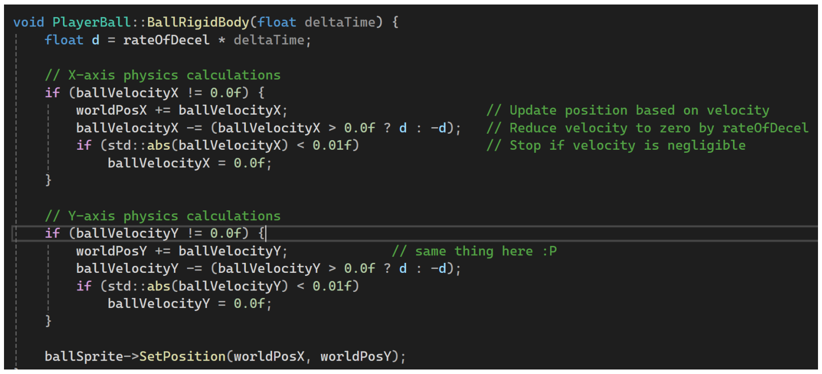
```cpp
void PlayerBall::PlayerController(float deltaTime) {
    static bool wasButtonPressed = false;
    bool isButtonPressed = App::IsKeyPressed(VK_LBUTTON);

    if (!isButtonPressed && wasButtonPressed) { // executes only if mouse button is released
        CalculateForce(normalizedX, normalizedY);
        hitCount++;
    }
    if (isButtonPressed) {
        // Calculate directional vector
        // V = (x2 - x1, y2 - y1)
        float dirX = worldPosX - mouseX;
        float dirY = worldPosY - mouseY;
        // output normalized vector between 0-1
        // |V| = sqrt(x^2 + y^2)
        float magnitude = sqrt(dirX * dirX + dirY * dirY);

        if (magnitude > 0.0f) {
            normalizedX = dirX / magnitude;
            normalizedY = dirY / magnitude;
        }
        // Loop through 0% - 100% power
        // y = sin(x) sorta
        elapsedTime += deltaTime;
        power = (std::sin(elapsedTime / 350) + 1.0f) / 2.0f;

        App::GetMousePos(mouseX, mouseY);
    }
    wasButtonPressed = isButtonPressed; // Update the previous button state
}
```

Figure 1. (Snippet of PlayerBall.cpp file, highlighting a major section to the player controller logic)



```cpp
void PlayerBall::BallRigidBody(float deltaTime) {
    float d = rateOfDecel * deltaTime;

    // X-axis physics calculations
    if (ballVelocityX != 0.0f) {
        worldPosX += ballVelocityX;                        // Update position based on velocity
        ballVelocityX -= (ballVelocityX > 0.0f ? d : -d);   // Reduce velocity to zero by rateOfDecel
        if (std::abs(ballVelocityX) < 0.01f)                // Stop if velocity is negligible
            ballVelocityX = 0.0f;
    }

    // Y-axis physics calculations
    if (ballVelocityY != 0.0f) {
        worldPosY += ballVelocityY;             // same thing here :P
        ballVelocityY -= (ballVelocityY > 0.0f ? d : -d);
        if (std::abs(ballVelocityY) < 0.01f)
            ballVelocityY = 0.0f;
    }

    ballSprite->SetPosition(worldPosX, worldPosY);
```

Figure 2. (Section of PlayerBall.cpp, displaying the ball’s rigid body physics method)

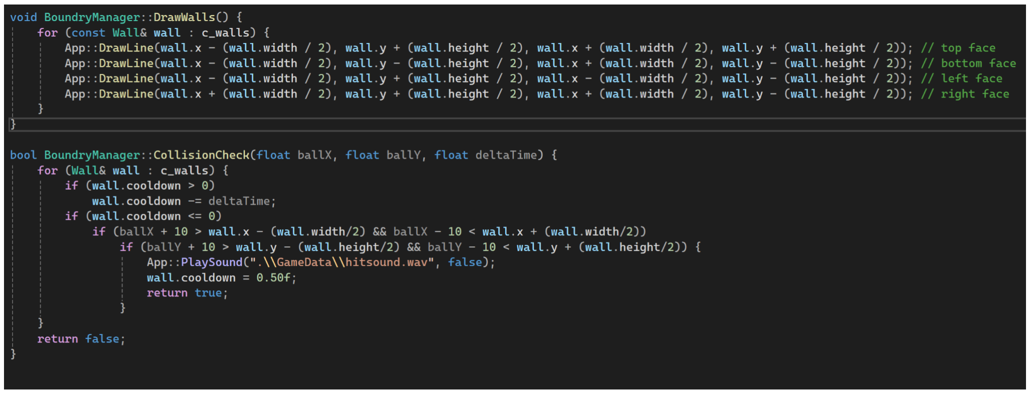
“BoundryManager” & “LevelData” Class

I’m aware I spelt ‘boundary’ wrong 😭

Overview:

An important goal of mine was to design the game's codebase to be as modular and expandable as possible (to my ability). So instead of hard coding every level, I instead created a modular level builder that allows developers to design and customize *Intergalactic “Big-Putt”* levels by simply making additions to a “LevelData” Script.
To accomplish this, I first needed to define a wall object within the BoundryManager’s header file by creating a constructor called “wall” (this struct defines the walls x and y coordinate, and also the walls height and width in world-space). This way we can easily store a collection of walls and iterate them in order to make simple render calls or collision checks as shown below.

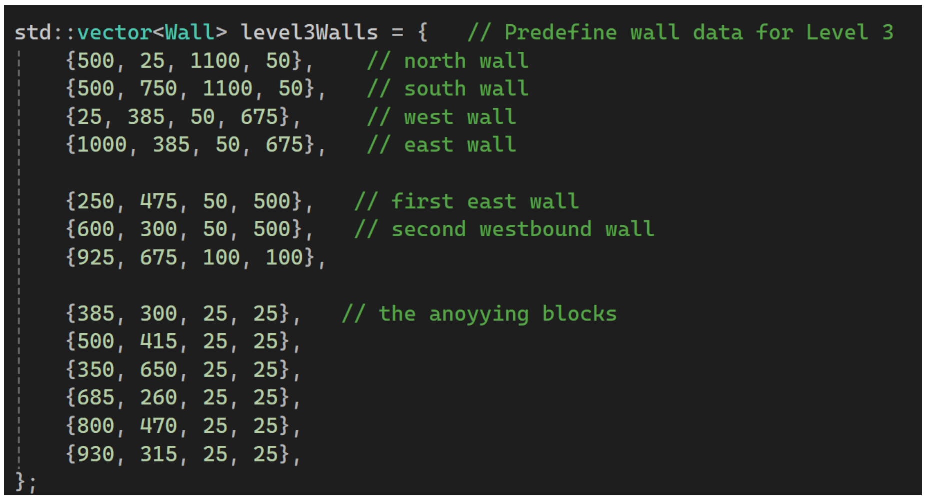
Small Code Snippets:



```cpp
void BoundryManager::DrawWalls() {
    for (const Wall& wall : c_walls) {
        App::DrawLine(wall.x - (wall.width / 2), wall.y + (wall.height / 2), wall.x + (wall.width / 2), wall.y + (wall.height / 2)); // top face
        App::DrawLine(wall.x - (wall.width / 2), wall.y - (wall.height / 2), wall.x + (wall.width / 2), wall.y - (wall.height / 2)); // bottom face
        App::DrawLine(wall.x - (wall.width / 2), wall.y + (wall.height / 2), wall.x - (wall.width / 2), wall.y - (wall.height / 2)); // left face
        App::DrawLine(wall.x + (wall.width / 2), wall.y + (wall.height / 2), wall.x + (wall.width / 2), wall.y - (wall.height / 2)); // right face
    }
}

bool BoundryManager::CollisionCheck(float ballX, float ballY, float deltaTime) {
    for (Wall& wall : c_walls) {
        if (wall.cooldown > 0)
            wall.cooldown -= deltaTime;
        if (wall.cooldown <= 0)
            if (ballX + 10 > wall.x - (wall.width/2) && ballX - 10 < wall.x + (wall.width/2))
                if (ballY + 10 > wall.y - (wall.height/2) && ballY - 10 < wall.y + (wall.height/2)) {
                    App::PlaySound(".\\GameData\\hitsound.wav", false);
                    wall.cooldown = 0.50f;
                    return true;
                }
    }
    return false;
}
```

Figure 3. (Image of BoundryManager.cpp file, showing an example of how the wall collection is utilized in run-time)



```cpp
std::vector<Wall> level3Walls = {   // Predefine wall data for Level 3
    {500, 25, 1100, 50},    // north wall
    {500, 750, 1100, 50},   // south wall
    {25, 385, 50, 675},     // west wall
    {1000, 385, 50, 675},   // east wall

    {250, 475, 50, 500},   // first east wall
    {600, 300, 50, 500},   // second westbound wall
    {925, 675, 100, 100},

    {385, 300, 25, 25},   // the anoyying blocks
    {500, 415, 25, 25},
    {350, 650, 25, 25},
    {685, 260, 25, 25},
    {800, 470, 25, 25},
    {930, 315, 25, 25},
};
```

Figure 4. (Example of LevelData.cpp file, representing how levels are built and parsed through the ‘BoundryManager’ as a vector collection)