Tail Smash!

- A Technical Report on the Game's Development -

Trevor Black
Johns Hopkins University
Computer Science
May 2024

Tail Smash!

Project Duration
January 13, 2024 –
February 13, 2024
Primary Contributor
Trevor Black
Additional Contributors
Ian Black
Martin Oka
Alexander Strassberg
Final Project Download
`https://TobBot2.itch.io`
Source Code
`https://github.com/TobBot2`
Contact Me
trevoroblack@gmail.com
917-428-8262

Contents

0 Preface

Hello! I'm Trevor Black. I am currently a student at Johns Hopkins University in my second year studying for my Computer Science BS degree. You may have noticed that the formatting of this document does not adhere to convention. Well, maybe it does. That is because I do not know what the convention is. I am merely mocking what I have seen in my textbooks, readings, and peers' writing. With only report writing experience in clubs and small school projects, I will no doubt make several mistakes. Still, I hope that I succeed in my goal of conveying the technical aspects of the journey I have gone through in completing this project.

This report will place an emphasis on the coding of my video game. For example, I will talk about why I chose certain data structures, how I implemented certain aspects, and why I chose certain technologies. It will not go over the decisions I made for game design reasons unless relevant to technical challenges I faced. If the technicalities of this project do not interest you, then you can go straight to the finished product available on my itch.io page at tobbot2.itch.io. If instead you wish to dive into the source code of this project without my guidance in the form of this report, you can visit the GitHub repository at github.com/tobbot2. If technical information is not what you are looking for, you can read a short blog on the ideation of this project if I remember to make that available somewhere (which I haven't).

To the additional contributors Martin Oka and Alexander Strassberg, I thank you for providing feedback and being great motivators to help me finish the project. And a special thanks to Ian Black, who, on top of providing feedback and programming aid throughout development, inspired me to start this project in the first place.

Without further ado, enjoy reading about the development of my video game Tail Smash!

1 Project Overview

1.1 Gameplay

Tail Smash! is the video game I programmed in one month. The gameplay consists of a player-controlled car which can accelerate forward and backward, as well as steer left and right. Attached to the car is a long ball and chain which can be flung around when the car takes sharp turns. Each level is a single screen which consists of targets and obstacles. The player's goal in a level is to hit all the targets as quickly as possible without having the car touch any obstacles. If the car touches an obstacle, it will restart the level. The game is fast paced with levels typically lasting five seconds, though an average player will require

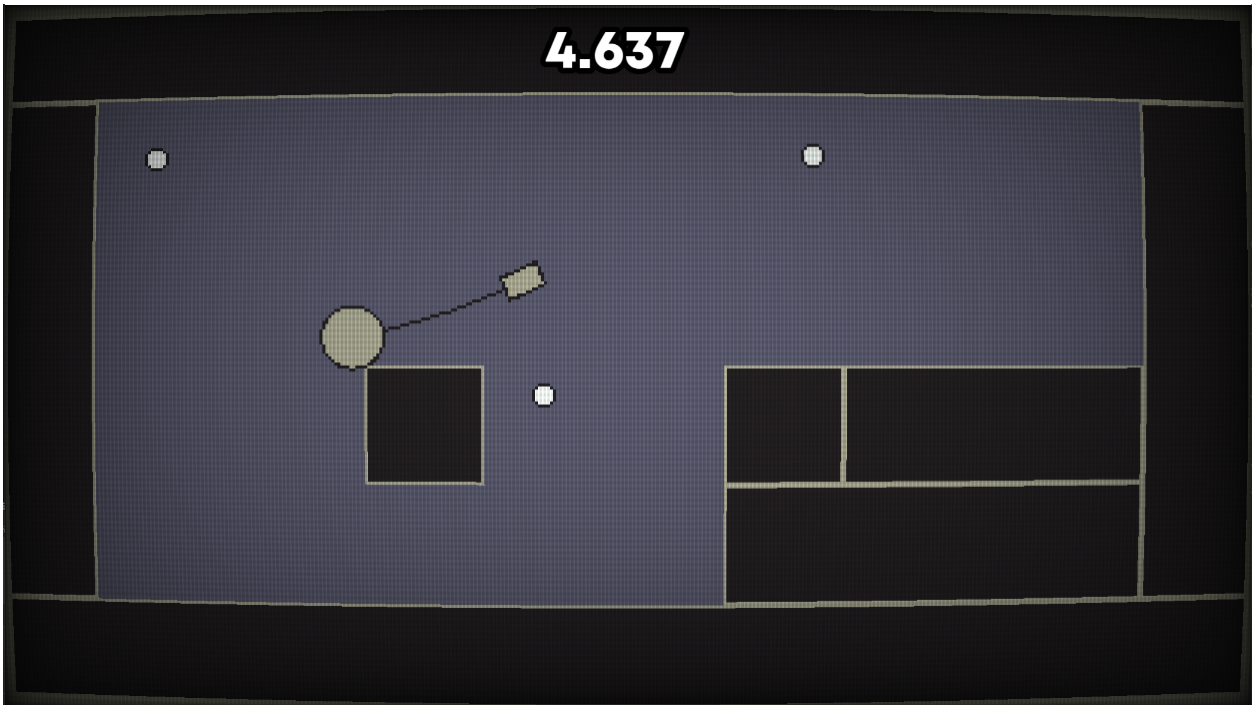

Figure 1: Screenshot from the game.

many attempts to beat a level. Respawning is instantaneous, and once a level is beaten, the next one is automatically loaded. This keeps the player engaged in constant action.

1.2 Technologies

1.2.1 Development Environment

With previous games, I mostly developed in Unity – a free game engine which uses C for scripting. Because I was working on my laptop, however, I switched to writing my own game engine as I found Unity took too long to load. I considered MonoGame, which uses C, but eventually settled on C++ due to my familiarity with it, its widespread usage in the industry, and superior speed.
For configuring the project, I went with Visual Studio 2022. Its out-of-the-box compatibility with C++ projects made it an easy choice over other options like VS Code and Neovim.

1.2.2 SFML

On top of C++, I used the Simple and Fast Multimedia Library v2.6.1 (SFML, `https://www.sfml-dev.org`). It has five modules: System, Window, Graphics, Audio, and Network, all

held in the namespace "sf". For this project, all but the Network module was used. SDL2 was considered, but it is made in C, so SFML seemed more beginner friendly and took advantage of C++'s features better. I am still torn about this decision and will talk more about it in section 5.

1.2.3 Dear ImGui

Dear ImGui v1.90.2 (`https://github.com/ocornut/imgui`) was used to handle the graphical user interface. Dear ImGui is an immediate-mode GUI. This means that the graphics get generated every frame directly from the variables in the code. Conventional restrained-mode GUIs require each widget to manually update the underlying value to properly reflect input. E.g. when a user modifies a volume slider, the restrained-mode widget must update the volume value in the code when it receives the input, rather than being automatically updated with the immediate-mode widget. Dear ImGui was chosen due to its immediate mode and strong community support. ImGui-SFML library (`https://github.com/SFML/imgui-sfml`) was used to incorporate Dear ImGui with SFML.

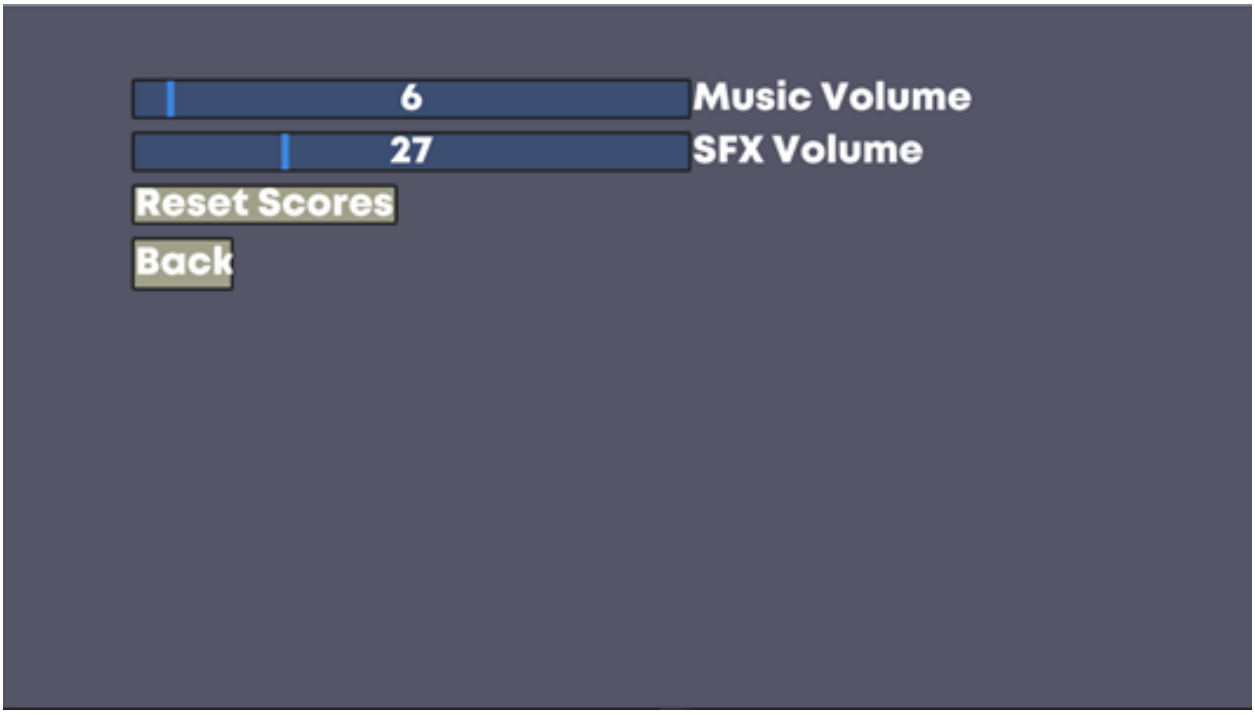

Figure 2: Screenshot of the settings menu with basic sliders and buttons.

2 Project Structure

2.1 Object Oriented Programming

The project structure follows the object-oriented paradigm. In most games, all objects in a scene inherit from the abstract class "Entity," "GameObject," or "Actor." This makes adding new types of objects straight forward, as it ensures all objects can be updated and rendered by default. For any additional functionality, components can be attached to add physics or player input, for example. Because Tail Smash! is so small, the benefits to having more structure with a robust inheritance scheme are diminished, and the overhead for such a system outweighs any benefits. This is best shown in the Level class, seen in figure 2.1. It contains vectors of Target and Wall. In its update function, it loops through all the targets – and it does not even loop through the walls at all. The player has access to the walls via its reference to Level. That is not to say that this method is superior. I will go into the drawbacks of this inheritance scheme in section 5.

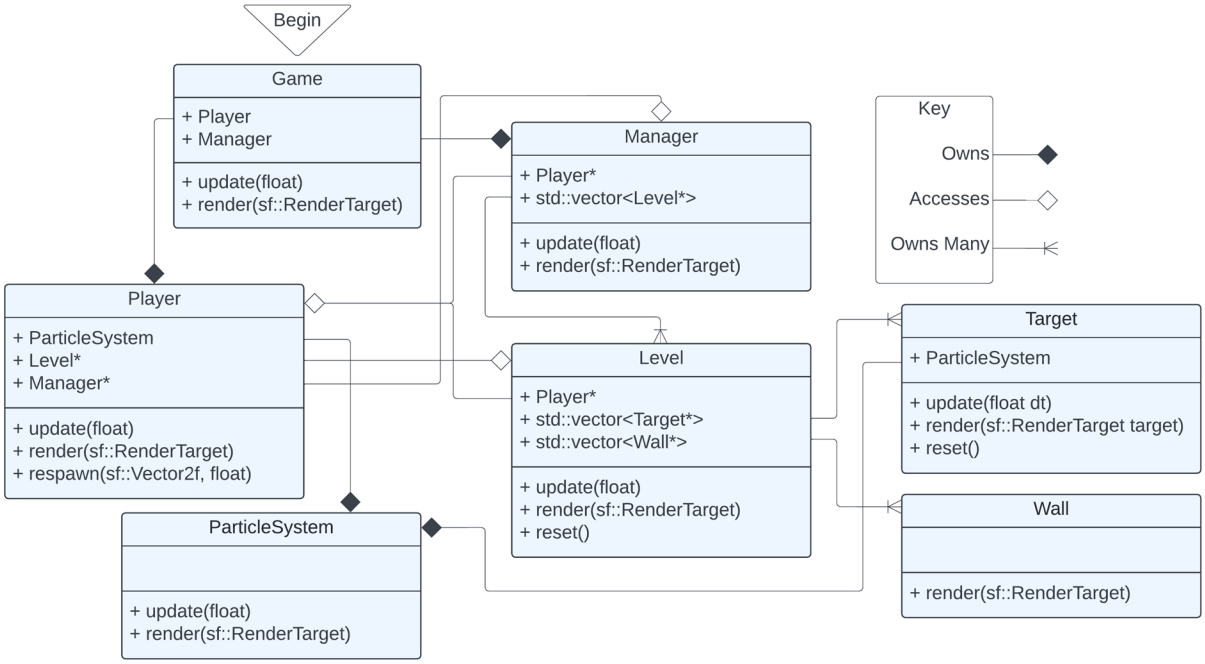

Figure 3: UML Diagram of data structures.

Note that the UML diagram in 3 shows only how each class stores information about other classes, as well as some key functions. As previously mentioned, inheritance is not utilized in this project, so all relationships are either "Owns" (creates the object and is responsible for destroying it) or "Accesses" (stores a pointer to an object that is owned by another class.

2.2 Game loop

First, the game opens with a preliminary screen to set the resolution. Once chosen, the program initializes the window, Dear ImGui, and the Game object. Finally, the game loop begins.
Every frame (while the window is open), the Game object's update and render function are called. From here, the Game object checks whether the state is Menu, Settings, Play, etc. For the purposes of this report, we will focus on the Play state. When the game state is Play, the Manager object owned by game is updated. Functionality to switch the state to Pause is also here. Inside manager's update function, the Player object and current Level object are updated sequentially. The player update handles input, collision with walls, and updating the tail, while the level update updates all the targets and checks whether there

```
while (window.isOpen())
{
        sf::Event event;
        while (window.pollEvent(event))
        {
                // ... handle event (omitted)
        }
        sf::Time dt = clock.restart();
        ImGui::SFML::Update(window, dt);

        game.update(dt.asSeconds());

        game.render(&window);

        ImGui::SFML::Render(window);
        window.display();
}
```

Listing 1: Code snippet of the main game loop in main.cpp

```
if (state == GameState::Play) {
        manager.update(elapsedTime);

        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Escape)) {
                state = GameState::Pause;
        }
}
```

Listing 2: Code snippet inside Game::update().

are still any remaining. Once all are gone, the level updates its "complete" flag to true, which is then checked by the manager to reset the level and increment the current level to the next.

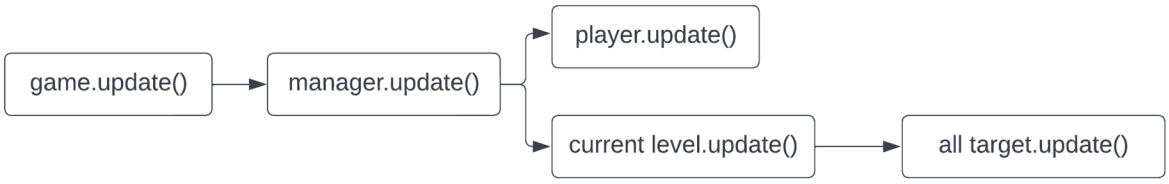

Figure 4: Class-level flow chart of update order.

Not pictured in 4 is the particle system update, which is simply a child process of its parent entity (Player or Target).

3 Implementation

With the general project structure understood, the exact implementation of the more interesting and possibly unintuitive mechanics can be explained in detail. For the sake of brevity, not every detail will be discussed in this section. Topics discussed will include the main mechanics, i.e. the player's car and the ball and chain, as well as the more difficult effects that add to the game's overall polish.

3.1 Player Car

The player's car is contained within the Player class. Recalling the UML diagram in section 2.1, the Player class Following the update function in the Player class, the user input is read first. After processing, the resulting values are then fed into the physics system.
With SFML, reading keyboard input is easily done using the sf::Keyboard class. Within the getInput() function of the Player class, the WASD keys are read, and return their values in a 2D normalized vector float.

```
sf::Vector2f Player::getInput() {
        sf::Vector2f axis(0.f, 0.f);
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::W) axis.y -= 1;
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::A) axis.x -= 1;
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::S) axis.y += 1;
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::D) axis.x += 1;

        float mag = sqrtf(axis.x * axis.x + axis.y * axis.y);
        if (mag <= 1.f) return axis;
        return axis / mag;
}
```

Listing 3: Code snippet handling user input.

The user input is only read if the player is alive. When not alive, the physics still runs, so the car continues rolling on its trajectory until the level is reset. When alive, the car is rotated based on the following line of code: Here, input.x comes from the x value returned from the getInput() method. The steerAcc value is

```
bodyShape.rotate(input.x * steerAcc * sqrt(speedPercent) * dt);
```

Listing 4: Code snippet handling car rotation.

simply a multiplier to change the sensitivity of the steering. The sqrt(speedPercent) is the current speed divided by the maximum speed. This element makes it so that the car turns more quickly when going slow. The reason for the square root function is because it gives a good curve as a linear function did not feel right. Finally, dt is the length of the frame, so the speed doesn't change with a different framerate.
After rotating the car, the forward acceleration is applied to the velocity, which is stored as a simple float because the car only moves forwards and backwards. Friction is also applied to the same velocity variable by adding a friction value multiplied by deltaTime in the direction opposite of the velocity. Finally, the velocity is applied to the car's position by simply taking the sin and cos of the car's rotation and scaling it by the velocity value and deltaTime.
After all movement calculations are made, the ball and chain physics are then calculated, but this will be discussed in a later chapter due to its complexity. Finally, collision detection is implemented. Because there is no collision resolution (i.e. the car does not need to be stopped from hitting a wall), very little information is calculated. Recalling the UML diagram in figure 3, The Player class has access to the current level, which in turn has a list of all walls in the level. Collision detection is simply implemented by looping over each wall and checking if each corner of the car is inside the wall or not.
In listing 5, first the function utilizes two guard statements so it does not unnecessarily compute a collision if a level is somehow not loaded or if the player is already dead. Next, looping through all the walls, it

```
if (level == nullptr) return;
if (!alive) return;

for (Wall* w : level->getWalls()) {
        sf::FloatRect wallBounds = w->getShape().getGlobalBounds();
        float rads = bodyShape.getRotation() * 3.141f / 180.f;
        sf::Vector2f sides(bodyShape.getSize().x * cos(rads),
            bodyShape.getSize().y * sin(rads));

        if (wallBounds.contains(bodyShape.getPosition() - sides / 2.f) ||
            wallBounds.contains(bodyShape.getPosition() + sides / 2.f) ||
            wallBounds.contains(bodyShape.getPosition()
            + sf::Vector2f(sides.x / 2.f, -sides.y / 2.f)) ||
            wallBounds.contains(bodyShape.getPosition()
            + sf::Vector2f(-sides.x / 2.f, sides.y / 2.f))) {
                // HIT WALL
                bodyShape.setFillColor(sf::Color(0x902e29ff));
                vel *= -.4f;
                alive = false;
                level->playerDie();
                break;
        }
}
```

Listing 5: Code snippet in Player::checkCollision()

retrieves the bounds for each wall and calculates the corners of the car. To calculate the corners, it stores a variable (called sides) which represents the vector pointing from the bottom left corner of the car to the top right, using the car's rotation, size, and sin/cos functions. Dividing this vector by two lets it be added and subtracted from the car's center to get each of the corners.
If any of the corners are found to be within the wall's bounds, the car is put in a 'dead' state. This includes changing its color to red and 'bouncing' backwards by reversing (and scaling down) its velocity. It then simply exits the loop as it doesn't need to check the later walls because it already knows it has been hit by a wall. Note that no information on the collision is saved (i.e. collision point, wall object, overlapping area, etc.). I found that calculating collision resolution for the car was too complex for the results, as the level is reset anyway shortly after the car hits a wall. Still, giving the player some feedback is important, so instantly reversing the velocity was a simple way to achieve this, in addition to changing the color of the car to red.

3.2 Ball and Chain (Inverse Kinematics)

Though the ball (also referred to as tail) and chain are tied to the player car and even updated within the player update function, the simulation is more involved than the car controller and so warrants its own section. The simulation uses a simple form of inverse kinematics to have each link in the chain properly have physical forces applied to it while maintaining their lengths. To apply these forces and constraints to the ball and chain, The chain is stored as a list of segments represented in a simple struct, with the ball being attached to the position of the last segment. With the struct, it becomes trivial to loop through each segment and calculate the simulation based on the state of each node. Notice how the parentPos is stored as a pointer to a vector. When a new ChainNode is created, the vector holding its 'anchor' position must be passed as well so that it can be dragged around. For most segments, this parentPos is the previous segment's position, but for the first segment it is nullptr, which represents the car's position (as seen in the next code snippet in listing 7). Additionally, the friction value is set to an arbitrary value found by trial and error to stop the chain from sliding and spinning around indefinitely.

```
struct ChainNode {
        ChainNode(const sf::Vector2f* parentPos, sf::Vector2f pos)
                : parentPos(parentPos), pos(pos) { };
        const sf::Vector2f* parentPos;

        sf::Vector2f pos;
        sf::Vector2f vel;

        float friction = 120.f;
};
```

Listing 6: ChainNode struct, responsible for storing information about each segment.

The above code snippet runs once every frame in the player's update function just after its own physics has been simulated. The ball and chain then reacts to this change by iterating down the chain and simulating the new forces and constraints. First, it moves based on its velocity and applies friction. After, it applies the length constraint by calculating the distance from its parent position and modifying the velocity and position to bring it back into range. Note that if the parentPos is nullptr, it is assumed that it is the first segment, whose parent is the car itself. After looping through all of the segments, the ball's position is set to the last segment's position, which is stored in a pointer called tailPos. After updating, the ball can then calculate if it has collided with any targets.
Again, looking back at the UML diagram in figure 3, the Player class has access to the active level, which in turn has access to the targets. The Target class is trivially simple, containing only information on its position, radius, and if it is active. In the loop, it first passes a guard statement making sure the ball cannot collide with a deactivated target. Once passing that, it does a distance check to see if the ball and target collide (if the distance is less than the sum of the objects' radii). This implementation of inverse kinematics does not account for the ball having any weight, so the momentum will not affect the car. A notable point of contention when writing this code was in line 12 in the above figure. In this line, I increment the pointer storing the tail's position by one, and then dereference that value and call it the tail velocity. This works because, as the comment suggests, the velocity of a chain node is stored directly after the position in memory as seen in listing 6. I knew at the time how bad this was, as any modification to the ChainNode struct could lead to chaos, but in the interest of time and due to the small scope of the project, I decided to let the hacky solution persist. This velocity value is then passed to the Target object so it can calculate a trajectory for the debris which uses the particle system discussed in the next section. Additionally, it also activates a shockwave effect originating from the target's position and dependent on the velocity of the tail on impact. This effect is also discussed in a later section on shaders.

3.3 Particle System (Object Pooling)

Special effects are crucial for giving feedback to the player on the impact they are having within the game world. Particle systems are an easy way to do this as they add visual variation in a large area, while still centering around the source – whatever caused the particle system to emit particles. Due to the nature of particle systems constantly creating and destroying particles, or creating thousands of particles in bursts, handling the initialization, storage, rendering and destruction of the particles is important for optimizing performance. The key technique used is called object pooling. Instead of destroying objects once they become used, it simply deactivates and hides them, so that when later objects need to be created, the deactivated ones can then be reset and reactivated so the game does not need to constantly perform expensive create and destroy routines for every particle. This concept can be implemented for many systems, but it is especially crucial for particles due to the outstanding number of objects – typically orders of magnitude greater than any other system. The particle system can then keep a list of all the particles and only updates the ones that are active. Each particle, naturally has a position, velocity, and color. The lifetime dictates how long until it is to be destroyed, and percent stores how far along its lifetime it is and is updated each frame when it is awake.

```
for (ChainNode* n : chainNodes) {
        n->pos += n->vel;

        if (abs(n->vel.x) > 0.f || abs(n->vel.y) > 0.f) {
                float nVelLen = sqrt(n->vel.x * n->vel.x + n->vel.y * n->vel.y);
                if (n->friction * dt > nVelLen) {
                        n->vel = {};
                }
                else {
                        n->vel -= n->vel / nVelLen * n->friction * dt;
                }
        }

        sf::Vector2f parentPos;
        if (n->parentPos == nullptr) {
                parentPos = bodyShape.getPosition();
        }
        else {
                parentPos = *n->parentPos;
        }

        float a = n->pos.x - parentPos.x;
        float b = n->pos.y - parentPos.y;
        float len = sqrt(a * a + b * b);

        if (len > segLen) {
                sf::Vector2f diff(a * segLen / len, b * segLen / len);
                n->vel += parentPos + diff - n->pos;
                n->pos = parentPos + diff;
        }
}

tailShape.setPosition(*tailPos);
```

Listing 7: Code snippet from Player::updateChains().

In the update function in listing 10, each particle that is active is looped through. It does this by simply storing all active particles in a vector (as well as all sleeping particles in a different vector). In the update loop, each particle updates its position and physics, along with its lifetime percent. Then the particle updates its color based on a decayAt variable. This simply makes the particle linearly lower its alpha towards the end of its lifetime so that it fanishes smoothly rather than suddenly disappears. Finally, if the particle has exceeded its lifetime, it then gets moved to the vector of sleeping particles so it isn't unnecessarily updated. This function covers updating and destroying particles, but creating particles is covered in the emit function as seen in listing 11.
When a new particle is emitted, it first calculates all the properties including position, speed, and direction. This becomes slightly more complex because particles are emitted with randomly varying directions and positions based on value for velVariability and spawnRange. After all of its properties are calculated, it then checks whether it should create a new particle or if a particle already exists that is not active and therefore can be reused. If a particle is reused, all of its values are overridden and it is moved to the vector of awake particles.

```
for (Target* t : level->getTargets()) {
        if (!t->isActive()) continue;

        sf::Vector2f targetPos = t->getShape().getPosition();
        float targetRadius = t->getShape().getRadius();

        sf::Vector2f diff = targetPos - *tailPos;
        float distanceSq = diff.x * diff.x + diff.y * diff.y;

        if (distanceSq < (targetRadius + tailRadius) * (targetRadius + tailRadius)) {
                // HIT TARGET
                sf::Vector2f tailVel = *(tailPos + 1); // disguisting. vel is stored after pos
                t->onHit(tailVel);
                manager->shockwave(t->getShape().getPosition(), tailVel.x * tailVel.x + tailVel
        }
}
```

Listing 8: Code snippet from Player::checkCollision() for the ball and targets.

```
struct Particle {
        Particle(sf::Vector2f pos, sf::Vector2f vel, sf::Color color, float lifetime)
                : pos(pos), vel(vel), color(color), lifetime(lifetime) { };
        sf::Vector2f pos;
        sf::Vector2f vel;

        sf::Color color;

        float lifetime;
        float percent = 0.f;
};
```

Listing 9: Code snippet in ParticleSystem.h showing Particle struct.

3.4 Levels

Due to the static nature of levels, the update code is uninspired and thus is omitted from the report. It simply updates a timer, the targets, and if the targets are all gone, the level is won. This only changes a boolean variable which can be accessed via a public function called isComplete(). The manager class then checks this function each frame and moves to the next level, resetting it in the process.
Levels are stored as code in Levels.h. Each level is stored in a pointer to a Level object, and then the level's public functions are utilized to set the name, spawn point (along with spawn rotation), the walls, and the targets. This file is then included with the #include directive inside Manager.cpp so the variables can pushed onto the vector containing all levels. An example level is seen in listing 12.
The reason for putting the level generation code in a different file is because for levels with lots of targets or especially walls, the number of lines of code rapidly increased. Putting them in a separate file and including it was easier to navigate and worked because it effectively copies and pastes the file to wherever the #include directive is. Of course, this is not the intended usage (or at least common usage) of the #include directive, but regardless, this is a small project and it worked quickly and easily. When making levels, it was nearing the end of the project and I was running out of steam, so I embraced the mentality of "There are two types of code: Good code, and production code." It was more important to me that it works rather than is maintainable due to the small nature of the project and the fact that a lot more would have to be rewritten if I wanted to take the project further anyway.
Creating the levels was done through a custom level editor written in HTML/JavaScript. I made it a

```
for (Particle* p : awakeParticles) {
        p->pos += (p->vel + forceField) * dt;
        p->vel *= friction;

        p->percent += dt / p->lifetime;

        if (p->percent > decayAt) {
                int alpha = 255 * (decayAt - p->percent) / (1.f - decayAt);
                p->color = sf::Color(p->color.r, p->color.g, p->color.b, alpha);
        }

        if (p->percent > 1.f) {
                sleepingParticles.push_back(p);
                // gets removed from awakeParticles just after loop ends...
        }
}
awakeParticles.erase(std::remove_if(awakeParticles.begin(), awakeParticles.end(),
        [](Particle* p) { return p->percent > 1.f; }), awakeParticles.end());
```

Listing 10: Code snippet in ParticleSystem::update()

simple web app because it really just needed to be something super bare-bones as the levels are not big or complex. JavaScript was a natural choice as I am familiar with it and didn't have to deal with a lot of boiler plate code necessary (though handy) for C++. The level editor can be run by opening editor.html in a web browser from the resources directory.

3.5 Polish (Shader Effects)

Similarly to the particle system, shaders were also heavily utilized to increase the environmental feedback from the player's actions. When targets are hit at faster speeds, varying levels of explosions occur to emphasize the hit. A combination of a flash of white washing out everything except the outlines of objects along with a fiery explosion and even a brief stutter – pausing the game for an instant – to prolong the impact. Additionally, the whole screen is covered in a CRT screen effect to give it a more arcade-like feel. The shaders were written in GLSL and simply applied to the entire screen. This was my first time working with shaders, and I only had a few effects, so I felt having all of the effects in the same file (globaleffects.frag) was reasonable. Performance sometimes drops when hitting a target which I suspect is due to the shader effects, but I did not have the time nor energy to resolve or even pinpoint the source as college work picked up in pace as I was wrapping up the project. Additionally, my computer is on the lower end of performance (but still modern), so that may play a minor role.

```
sf::Vector2f dir = vel;
float mag = sqrt(dir.x * dir.x + dir.y * dir.y);
float angle = atan2(dir.y, dir.x);
int randAngle = gaussianAngle
        ? rand() % 20 + rand() % 20 + rand() % 20 + rand() % 20 + rand() % 20
        : rand() % 100;
angle += angleVariability * ((float)(randAngle) / 100.f - .5f);
float power = rand() % velVariability;
dir.x = cos(angle) * power * mag;
dir.y = sin(angle) * power * mag;

sf::Vector2f pos = *position;
float randPosAngle = 6.282f * (float)(rand() % 100) / 100.f;
pos += sf::Vector2f(cos(randPosAngle) * spawnRange, sin(randPosAngle) * spawnRange);

if (sleepingParticles.empty()) {
        // create new particle
        Particle* p = new Particle(pos, dir, color, lifetime);
        awakeParticles.push_back(p);
}
else {
        Particle* p = sleepingParticles.back();
        sleepingParticles.pop_back();
        // reset p properties (omitted from report for brevity)
        awakeParticles.push_back(p);
}
```

Listing 11: Code snippet from ParticleSystem::emit().

```
Level* dots = new Level();
dots->setName("Dots");
dots->setSpawn(sf::Vector2f(550.f, 750.f), 90.f);
dots->addWall(sf::Vector2f(2, 4), sf::Vector2f(2072, 142));
dots->addWall(sf::Vector2f(2, 1004), sf::Vector2f(2072, 162));
dots->addWall(sf::Vector2f(1952, 154), sf::Vector2f(122, 842));
dots->addWall(sf::Vector2f(2, 154), sf::Vector2f(142, 842));
dots->addTarget(sf::Vector2f(550, 550));
dots->addTarget(sf::Vector2f(1050, 550));
dots->addTarget(sf::Vector2f(1550, 550));
```

Listing 12: Code for generating a level.

4 Mistakes

Throughout this project, I have learned a great deal. Previously, I had only fully finished coding a game in Unity with the help of my brother. Finishing this game was a great way for me to learn about the more tedious parts of coding a game. Elements like polish, menus and level creation, were ignored for previous projects as I would typically abandon them before they got to the point where these elements were necessary. Because it was my first time with these elements, I certainly made errors. Even with parts I am more familiar with, like project structure, I added unnecessary functionality and made the scheme of which objects control what unclear.

Starting with the project structure, I am glad I ditched inheritance for this project. The small nature and uniqueness of each object would have led to unnecessary complexity. Typically I have all objects inherit from an abstract 'Entity' class, but foregoing that allowed me to freely ignore some boiler plate code and project clutter. Instead, I had to rewrite the rendering function a couple times more than before, but for a simple 2D game like this it is very short anyway. My error in the project structure comes from the Manager class. It handles the levels and player, but overall is an unnecessary middle-man as the Game class can just as easily take care of it. This led to a lot of confusion about where certain bits of logic should be placed as I often forgot whether the Level, Manager, or Game class was supposed to handle a certain event.

When coding the menus, I had to create UI components, which I had never done before. I picked a popular UI library (Dear ImGui) due to its variety of resources and ease of use. In the past, I have only made a UI with Unity, so creating and placing all of the elements in code took a moment to learn, but was fairly easy. However, due to Dear ImGui being an immediate-mode UI, the logic for interacting with it as well as the rendering positions of the elements were combined into one function call which I was unfamiliar with. I mistakenly put much of this logic in my rendering functions as I figured I was using these components to render them on the screen, but this was a mistake as I had to deal with the logic in the update function, causing my menu code to be split in two places. In the future, with immediate-mode UI, I will put all of the code in the update function where I can safely handle the logic. The rendering of the UI has its own function that I call later, which I inconveniently forgot about.

I already mentioned my mistakes with shaders and the level storing system in sections sections 3.5 and 3.4, but I will mention them again just to reiterate and propose a future solution. With shaders, I still have much reading up to do on best practices and common optimization pitfalls especially, but for now I would guess (based on intuition alone) that my biggest pitfall was in having all the effects interact with the entire screen rather than a smaller texture. For the fiery explosion, for example, I would simply offset the animation's position to wherever the explosion occurred, but this meant I could only have one explosion at a time. To improve the level system, I will store them in custom binary files (or json if I get lazy enough). This will allow me to dynamically load the levels rather than loading them all at once when I start the game. For such small levels, it was never an issue, but if I wanted to add scaling levels, or edit existing levels, it would be much easier to overwrite them via code rather than having to edit C++ code in a header file where my IDE doesn't recognize the variable names due to my unorthodox usage of the #include directive.

Finally, and this isn't really a misstep, but next project I will attempt to use SDL2. Not only is it more common in the industry, but it gives much more flexibility when it comes to rendering graphics especially. There is still an optional 2D graphics library that can be used with SDL2 called SDL2_image, but without it I will be able to either write my own or use a 3D graphics library like glfw. I have only coded 3D games with Unity because it handles the most difficult parts (graphics and physics) for me, but using open source libraries looks to be a surprisingly simple – if still difficult – alternative.

5 Conclusion

This project started as a short in-flight game jam, but extended to a month long endeavor leading to my first fully-finished game. I knew the dangers of scope-creep before going in, so I cut ideas left and right to just focus on finishing the game as quickly as possible. Still, it took a month to complete (granted, I wasn't working on it full-time). After the proof-of-concept was finished in the plane, I had to finish it. I scoped it out to finish within a week, though I knew this was a lie. Pushing through new topics and really embracing the mantra of "There is only good code and production code," I powered through and was able to finish this minimum viable product.
In the future I will come back to this idea. Not the code, though – I'll definitely want to rewrite everything from scratch. But I really think this game idea has a lot of potential. With a cleaner code-base, better visuals, and endless possibilities for new mechanics, this project could one day become even more fun and possibly even profitable. For now though, it will collect some dust, waiting for a future, more experienced me to take the project on and deliver on my current dreams.