210014557

CS4303 P1 Report

Physics

03.03.24

Overview

"OVER S∞∞N" is a 2D platformer game implementing a custom physics engine that handles forces, collisions, and specialised physics constructs. The player controls a samurai character who must navigate through a vertically-oriented level, defeat enemies, and collect a golden coin to win.

The game has multiple levels of physics implementation: a foundational point-mass physics system that is used as a foundation for a more complex physics, like springs and collision detection. The core gameplay loop revolves around classic platforming mechanics combined with the unique combat and jump+glide physics, which create an engaging gameplay loop that challenges players to combine movement and combat to ‘speedrun’ the level.

Design

Core Game Mechanics

1. **Character Movement and Control System:**
The character movement controls implement a force-based system where player inputs generate forces that are applied to the character's object. This creates a responsive yet physically grounded movement.
Horizontal/Vertical movement with force application is used in:

- Jumping mechanics that use an upward impulse and gravity
- Gliding ability that reduces gravity's effect when the character is in the air
- Melee attack system that has collision detection
- Projectile shooting with its own physics and collision detection

2. Physics-Based Platforming:
The platforming mechanics are built on several key physics systems:

- *Gravity Force Generator:* Constantly applies downward force to objects that are not static
- *Collision Detection*: Handles interaction between the character and platforms/enemies/springs/coins
- *Spring Mechanics:* Specialized platforms that apply a vertical impulse when the character jumps on them
- *Friction:* force applied to horizontal movement that prevents sliding

3. Enemy AI and Combat:
Enemies utilize the same physics system as the player character but with unique behavior patterns:

- Patrol within defined boundaries and changing directions
- Attack mechanic when the player comes within range
- Physics-based knockback when the enemie is hit by the player
- Health system with damage calculation

4. Structured Force System
The game implements a modular force registry that allows:

- Multiple force generators to act on a single entity
- Force accumulation for accurate physics
- Specialized forces like gravity, drag, and buoyancy (not fully implemented)
- Removal of forces during gameplay if needed

Physics Engine Architecture

The physics engine follows a structured architecture with distinct separation of operations:

Physics Objects -> Base class for all entities that interact with the physics system.
The objects have their own position, velocity, acceleration properties, as well as force accumulation, and collision boundaries (that use the radius property)

Force Generators are implemented as an interface system, and are used by the engine to calculate the required forces before applying them. This allows a level of abstraction that enables any changes to be implemented to the core physics mechanics that impact all the non-static objects

Force Registry: Manages relationships between objects and forces, by tracking all the forces that are applied to non-static physics objects and updating them accordingly to the defined physics.

Collision System: Detects and resolves collisions between objects
The collision system uses circles (based in the radius of the object) to detect and resolve any collisions between non static objects. For the character and enemie attacks I have implemented an impulse-based resolution to simulate a more realistic knockback behaviour. Moreover, the system has special cases to resolve collisions with the player and special objects e.g. coins and springs.

Interaction Between Systems

The game loop is created from the interaction between these systems:

Movement + Platforms: The character's movement forces interact with platform collisions to create the vertical platformer experience
Combat + Physics: When attacks land, forces are applied to create a visual representation of it, combined with the HUD that displays all the information to the player.
Springs + Gravity: The spring mechanics temporarily override gravity to create a trampoline effect.
Enemy AI + Physics: Enemies use the physics system for movement while their AI determines when to change direction or attack and when to attack the player. A succeful attack also resolves into not only the visual change in the players health, but also an animation of the player being hit.

This structured approach ensures that each system has a defined responsibility while allowing complex behaviors to form within their interactions. The physics engine's modular design also allows for expansion, which would allow for adding new mechanics or features without major restructuring.

Technical Description

Physics Engine Challenges and Solutions

Force Accumulation System:
The implementation of a robust force accumulation system that could manage multiple simultaneous forces acting on objects was one of the biggest technical challenges in the game's development. The system employs a force accumulator pattern instead of directly altering the acceleration property:

```
void applyForce(PVector force) {
    // Don't apply forces to static objects
    if (isStatic) return;

    // Add force to accumulator instead of directly affecting
acceleration
    PVector f = force.copy();
    forceAccum.add(f);
}
```

In order to avoid residual forces influencing later frames, it was difficult to ensure that forces were properly accumulated, applied, and then cleared in each frame. With this technique, forces from many sources can be applied throughout a frame and then resolved collectively during the integration step:

```
void update() {
    // Calculate acceleration from accumulated forces
    acceleration = PVector.div(forceAccum, mass);

    // Update velocity with acceleration
    velocity.add(acceleration);

    // Apply friction to reduce sliding
    velocity.mult(friction);

    // Update position with velocity
    position.add(velocity);

    // Clear forces for the next update
    clearForces();
}
```

By clearly separating force application from physics integration, this method improved system maintainability and the possibility of adding new features, as well as enabled complex force interactions.

Collision Detection and Resolution
Implementing collision detection that worked reliably across different object types presented another big challenge. I opted for a circle-based collision system for simplicity and performance:

```
boolean isColliding(PhysicsObject other) {
    float distance = PVector.dist(this.position, other.position);
    return distance < this.radius + other.radius;
}
```

The more complex challenge was implementing proper collision resolution. The solution was an impulse-based approach that handled different scenarios:

1. Collisions between two dynamic objects
2. Collisions between dynamic and static objects
3. Special case handling for character-spring interactions

For dynamic-static collisions, the code determines which object should move and applies forces accordingly:

```
if (other.isStatic) {
    // Push the non-static object away from the static object
    PVector separationVector = PVector.mult(collisionNormal,
this.radius + other.radius - PVector.dist(this.position,
other.position));
    this.position.sub(separationVector);

    // Reflect velocity for the non-static object only
    this.velocity.add(PVector.mult(collisionNormal, -2 *
PVector.dot(this.velocity, collisionNormal)));
    this.velocity.mult(0.7); // damping
    return;
}
```

For dynamic-dynamic collisions, an impulse calculation distributes the collision force based on the masses of the objects:

```
float impulse = newSeparatingVelocity / totalInverseMass;
PVector impulseVector = PVector.mult(collisionNormal, impulse);
// Apply impulse proportionally based on mass
this.velocity.add(PVector.div(impulseVector, this.mass));
other.velocity.sub(PVector.div(impulseVector, other.mass));
```

This method keeps things from interpenetrating while guaranteeing physically realistic collision responses.

Force Generator Architecture

To maintain code organization and flexibility, I implemented a force generator system, which allowed different types of forces to be encapsulated as separate classes implementing a common interface:

```
interface ForceGenerator {
  void updateForce(PhysicsObject object);
}
```

This architecture made it straightforward to implement various force types, described in the design section.

The force registry manages associations between objects and forces:

```
void updateForces() {
    for (ForceRegistration fr : registrations) {
        fr.fg.updateForce(fr.object);
    }
}
```

This approach provided solid separation of forces, making it easy to add new force types without modifying existing code.

Platform Collision Handling

A particularly challenging aspect was implementing one-way platform collision that would allow the character to jump through platforms from below but land on them from above. This required special collision logic outside the already implemented physics system:

```
void handlePlatformCollisions() {
    // Get character's position
    float characterFeetY = character.position.y + character.radius;
    float characterLeftX = character.position.x - character.radius *
0.8;
    float characterRightX = character.position.x + character.radius *
0.8;

    boolean wasOnPlatform = false;

    // Check platforms
    for (PlatformObject platform : platforms) {
        // Calculate platform bounds
        float platformTopY = platform.position.y - platformHeight/2;
        float platformLeftX = platform.position.x - platformWidth/2;
        float platformRightX = platform.position.x + platformWidth/2;

        // Check horizontal overlap
        boolean horizontalOverlap = characterRightX > platformLeftX
&& characterLeftX < platformRightX;

        if (horizontalOverlap) {
            // Check if character is near the top of the platform and
falling
            boolean isFallingOntoTop = character.velocity.y >= 0 &&
                            characterFeetY >= platformTopY &&
                            characterFeetY <= platformTopY + 15;

            if (isFallingOntoTop) {
                // Place character on top of platform
                character.position.y = platformTopY -
character.radius;
                character.velocity.y = 0;
                character.fallingDown = false;
                character.jumpStartY = character.position.y;
                wasOnPlatform = true;
                break;
            }
        }
    }
}
```

Instead of using the conventional circle collision detection, this method employs specialised intersection tests to determine whether the character's feet are moving downward and are close to the top of a platform. As a result, the intended one-way platform behaviour is produced from this.

Spring Physics Implementation

Implementing springs presented a unique challenge in balancing the accuracy of physics with the gameplay feel. My solution was a *semi-realistic* approach that simulates compression and extension:

```
if (isAboveSpring && isTouchingSpring && isFalling) {
    character.position.y = springTopY - character.getRadius();

    if (spring.compress()) {
        // Clear any accumulated forces that might counteract the
bounce
        character.clearForces();

        // Apply upward velocity
        character.velocity.y = -spring.getBounceForce();

        // Add a horizontal boost in the direction the character is
moving
        if (character.velocity.x != 0) {
            character.velocity.x *= 1.3; // Increase horizontal
momentum by 30%
        }

        // Set spring bounce state
        character.setSpringBounce(true);
        character.jumpingUp = true;
        character.fallingDown = false;
        character.jumpStartY = character.position.y;
    }
}
```

The spring class also has visual feedback showing compression and extension states.

Jump and Movement Mechanics

Creating natural-feeling jump mechanics required me to tune the forces multiple times to achieve a ‘smooth’ feel. The solution was a variable-force jump that provides more upward force at the beginning of a jump and decreases as the character rises:

```
// Normal jump
float jumpProgress = (jumpStartY - position.y) / JUMP_HEIGHT;
float currentJumpForce = JUMP_FORCE * (1.0f - jumpProgress * 0.3f);

if (position.y > jumpStartY - JUMP_HEIGHT) {
    applyForce(new PVector(0, -currentJumpForce));
} else {
    jumpingUp = false;
    jumpPauseCounter = JUMP_PAUSE_DURATION;
}
```

This creates a more natural arc to jumps while maintaining physical plausibility. Additionally, a brief pause at the peak of a jump adds a sense of weight:

```
if (jumpPauseCounter > 0) {
    jumpPauseCounter--;
    if (jumpPauseCounter == 0) {
        fallingDown = true;
    }
}
```

The gliding mechanic was implemented as a counterforce to gravity when falling:

```
// Check for gliding
if (fallingDown && gliding) {
    applyForce(new PVector(0, -GLIDE_GRAVITY));
}
```

Despite not being physically precise, these mechanics produce a responsive and enjoyable movement system that is still based on physics.

Optimization Techniques

Several optimization approaches were employed to maintain performance:

1. Object Filtering in Collision Detection: The system skips collision checks between certain object types to avoid unnecessary calculations:

```
  // Skip collisions between Character and Enemy or Platform
   if ((objA instanceof Character && (objB instanceof Enemy || objB
instanceof PlatformObject)) ||
       ((objA instanceof Enemy || objA instanceof PlatformObject) &&
objB instanceof Character)) {
       continue;  // Skip to the next iteration
   }
```

2. Static Object Handling: Static objects bypass most physics calculations, also reducing computational load:

```
 void update() {
       // Static objects don't move
       if (isStatic) return;

       // Normal physics updates...
   }
```

Additionally, the game avoids costly computations for typical interactions by using specialised collision detection (meantioned above) for platforms instead of general-purpose physics collisions.

Even with numerous active physics objects and intricate interactions, the game will continue to run smoothly because of these optimisations.

Conclusion and Critical Review

My project successfully implemented a 2D platformer game with a custom physics engine that demonstrates several levels of physics simulation complexity. The game incorporates a structured force accumulation system, collision detection and resolution, and specialized physics constructs such as springs. Players can traverse platforms, interact with enemies, and use physics-based dynamics in the seamless gameplay.

I believe my game satisfies the requirements by tackling each level of physics complexity one by one. I started with implementing the basic point-mass physics for the character and enemies, which was challenging at first since I had to figure out how forces should accumulate over time. The collision system took several iterations to get right - especially making the one-way platforms work correctly. Originally, I struggled with objects getting stuck in each other, but eventually sorted that out with proper separation logic. For the force system architecture, I'm pretty happy with how I managed to keep things modular with the ForceGenerator interface, even though I went through a few different designs before settling on this approach. Looking back, I probably spent too much time perfecting the spring bouncing mechanic, but it ended up being one of the most satisfying parts of the gameplay.

Optimisation still is one of the areas of the gameplay that could use additional improvements. The current static object filtering works well for the game, but I suspect that with the addition of new features it might not be enough.

Due to the time constraints, I couldn't fully implement the buoyancy system that I was working on into the final project. My implementation for the physics components for it can be found in the 'BuoyancyForce.pde', 'Water.pde', and 'Pond.pde' files.
The buoyancy system was designed to:

- Calculate object submersion depth in water
- Apply upward force proportional to the submerged volume of the object
- Create increased drag when objects move through water
- Generate visual effects like ripples and bubbles

If I had more time I would finalise the integration of the buoyancy system as well enhanced my enemies AI with a coordinate based system and player tracking based in the Manhattan Distance.

In conclusion, while there are areas for improvement and features that could not be implemented within the time constraints, my project successfully demonstrates how a structured physics engine can create engaging gameplay experiences while maintaining code clarity and separation of the game physics elements.

Appendix

Borrowed Implementations (Graphics)

https://hugues-laborde.itch.io/pixelartsamurai?download
https://hugues-laborde.itch.io/pixel-art-character-pack