Modularity Exercise: Space Shooter Game

Advanced Game Programming Topics

Departamento de Engenharia Informática

2025-10-10

Problem Description

Consider a video game like *Xenon*, where, among other objects, there exists a spaceship that fires shots against enemies. In this assignment, you will model and implement classes corresponding to the **Spaceship**, **Gun**, and **Shot** objects using modern C++ best practices.

This exercise focuses on:

- **Modularity**: Proper separation of interface and implementation
- **RAII**: Resource Acquisition Is Initialization
- **Modern C++**: Using C++17/20 features
- **Class Design**: Concrete classes with proper encapsulation

Object Requirements

1. Position2D Structure

Create a simple structure to represent 2D positions:

```
struct Position2D {
    double x;
    double y;
};
```

2. Vector2D Structure

Create a structure to represent 2D velocity vectors:

```
struct Vector2D {
    double x;
    double y;
};
```

3. Shot Class

A **Shot** represents a projectile fired by the gun.

Attributes:

- Position (x, y coordinates)
- Velocity vector (speed and direction)
- Destruction power (damage value)
- Active status (whether the shot is still active)

Behaviors:

- Move the shot based on its velocity
- Check if the shot is still within bounds
- Get/set shot properties
- Deactivate the shot

Requirements:

- Use proper encapsulation (private data members)
- Provide const-correct member functions
- Implement a method to update position: `void update(double deltaTime)`
- Implement bounds checking: `bool isInBounds(double maxX, double maxY) const`

4. Gun Class

A **Gun** manages shooting mechanics and cooldown.

Attributes:

- Cooldown time between shots (in seconds)
- Current cooldown remaining
- Maximum ammunition capacity
- Current ammunition count
- Shot template (power, speed)

Behaviors:

- Fire a shot (if cooldown allows and ammunition available)
- Update cooldown timer
- Reload ammunition
- Check if ready to fire

Requirements:

- Return `std::optional<Shot>` from `fire()` method (C++17 feature)
- Implement cooldown management: `void update(double deltaTime)`
- Implement `bool canFire() const` to check firing readiness
- Track ammunition: `int getAmmo() const` and `void reload()`

5. SpaceShip Class

A **SpaceShip** represents the player's ship.

Attributes:
- Position (x, y coordinates)
- Velocity vector
- Gun (composition relationship)
- Health points
- Active shots (container of active shots)

Behaviors:
- Move the ship based on velocity
- Fire shots using the gun
- Update all active shots
- Remove inactive/out-of-bounds shots
- Set velocity for movement

Requirements:
- Use `std::vector` to store active shots
- Implement proper copy/move semantics or delete them
- Provide `void update(double deltaTime)` for updating ship and shots
- Implement `void setVelocity(const Vector2D& vel)`
- Implement `void fireShot()`
- Implement `const std::vector<Shot>& getActiveShots() const`

Exercises

Exercise 1: Class Design and Implementation

Design and implement the classes **Shot**, **Gun**, and **SpaceShip** following modern C++ best practices:

Part A: Header Files (.hpp)

Create three header files with proper include guards:

1. **Shot.hpp** - Shot class interface
2. **Gun.hpp** - Gun class interface
3. **SpaceShip.hpp** - SpaceShip class interface

Requirements:

- Use `#pragma once` or traditional include guards
- Declare all public interfaces
- Keep private members private
- Use forward declarations where possible
- Add documentation comments for public methods

Part B: Implementation Files (.cpp)

Create corresponding implementation files:

1. **Shot.cpp** - Shot class implementation
2. **Gun.cpp** - Gun class implementation
3. **SpaceShip.cpp** - SpaceShip class implementation

Requirements:

- Implement all member functions
- Use member initializer lists in constructors
- Implement proper const-correctness
- Handle edge cases (e.g., negative values, null checks)

Part C: Modern C++ Features

Your implementation must use:

- **Uniform initialization** with `{}`
- **`const` correctness** for methods that don't modify the object
- **`std::optional`** for the Gun's `fire()` method (returns shot or nothing)
- **`std::vector`** for managing active shots
- **RAII principles** for resource management
- **Default member initialization** where appropriate

Example class skeleton:

```
// Shot.hpp
#pragma once

struct Position2D {
    double x{0.0};
    double y{0.0};
};

struct Vector2D {
    double x{0.0};
    double y{0.0};
};

class Shot {
public:
    Shot(Position2D pos, Vector2D vel, double power);

    void update(double deltaTime);
    bool isInBounds(double maxX, double maxY) const;
    bool isActive() const;
    void deactivate();

    Position2D getPosition() const;
    double getPower() const;
```

```
private:
    Position2D position;
    Vector2D velocity;
    double destructionPower;
    bool active{true};
};
```

Exercise 2: Simulation Application

Create a **main.cpp** file that demonstrates your classes:

Requirements:

1. **Create a SpaceShip** object at position (100, 100)
2. **Simulate movement** over 10 seconds:
   - Update at 60 FPS (deltaTime = 1/60 seconds)
   - Set different velocities for the ship
   - Fire shots at regular intervals
3. **Display information** at each second:
   - Ship position
   - Number of active shots
   - Gun ammunition count
   - Gun cooldown status
4. **Remove inactive shots** that go out of bounds

Example simulation structure:

```
int main() {
    // Create spaceship
    SpaceShip ship{100.0, 100.0}, Gun{0.5, 20};

    // Simulation parameters
    constexpr double FPS = 60.0;
    constexpr double deltaTime = 1.0 / FPS;
    constexpr double simulationTime = 10.0;
    constexpr double worldWidth = 800.0;
    constexpr double worldHeight = 600.0;

    // Set ship velocity
    ship.setVelocity({50.0, 0.0});  // Move right at 50 units/sec

    // Simulation loop
    double elapsedTime = 0.0;
    int frameCount = 0;

    while (elapsedTime < simulationTime) {
        // Update ship
        ship.update(deltaTime);
```

```
        // Try to fire every 30 frames (0.5 seconds)
        if (frameCount % 30 == 0) {
            ship.fireShot();
        }

        // Print status every second
        if (frameCount % 60 == 0) {
            std::cout << "Time: " << elapsedTime << "s\n";
            // Print ship info...
        }

        elapsedTime += deltaTime;
        ++frameCount;
    }

    return 0;
}
```

Exercise 3: Extended Features (Optional)

Implement additional features to enhance your design:

A. Shot Pooling

Instead of creating/destroying shots continuously, implement an object pool:

- Create a fixed pool of Shot objects
- Reuse inactive shots instead of creating new ones
- This improves performance by reducing allocations

B. Different Gun Types

Create derived classes for different gun types:

- **RapidFireGun**: Lower cooldown, lower power
- **HeavyGun**: Higher cooldown, higher power
- **BurstGun**: Fires multiple shots at once

Use inheritance and virtual functions appropriately.

C. Ship Upgrade System

Add an upgrade system to SpaceShip:

- Health upgrades
- Speed upgrades
- Gun upgrades (swap between different gun types)

D. Statistics Tracking

Add statistics tracking:

- Total shots fired
- Shots currently active
- Maximum shots on screen simultaneously
- Total distance traveled

Specific Requirements:

✓ All classes compile without warnings (`-Wall -Wextra`)
✓ Proper const-correctness throughout
✓ No memory leaks (use smart pointers if dynamic allocation needed)
✓ Proper RAII principles
✓ Clear separation of .hpp and .cpp files
✓ Meaningful variable and function names
✓ Simulation produces reasonable output

Tips and Best Practices

Design Tips:

1. **Start simple**: Get basic functionality working before adding features
2. **Think about ownership**: Who owns the shots? The gun or the ship?
3. **Consider the update loop**: How do objects update each frame?
4. **Bounds checking**: When should shots be deactivated?

C++ Best Practices:

```
// Good: Use const references for parameters
void setVelocity(const Vector2D& vel);

// Good: Mark non-modifying functions const
Position2D getPosition() const;

// Good: Use member initializer lists
Shot::Shot(Position2D pos, Vector2D vel, double power)
    : position{pos}, velocity{vel}, destructionPower{power}, active{true} {}

// Good: Use std::optional for operations that may fail
std::optional<Shot> Gun::fire() {
    if (!canFire()) {
        return std::nullopt;
    }
    // ... create and return shot
}

// Good: Use range-based for loops
for (auto& shot : activeShots) {
    shot.update(deltaTime);
}
```

Common Pitfalls to Avoid:

❌ Forgetting to update cooldown timers
❌ Not removing out-of-bounds shots (memory leak)
❌ Mixing up position and velocity
❌ Not handling edge cases (negative values, division by zero)
❌ Using raw `new`/`delete` instead of smart pointers or containers
❌ Making everything public

Additional Resources

- **C++ Reference**: cppreference.com
- **C++ Core Guidelines**: isocpp.github.io/CppCoreGuidelines
- **std::optional**: en.cppreference.com/w/cpp/utility/optional
- **RAII**: en.cppreference.com/w/cpp/language/raii

Good luck and happy coding! 🚀

Remember: Write code that you would want to maintain in 6 months. Your future self will thank you!