Advanced OOP in C++ — Project

Project: Implementation of a Space Shooter with and ECS

DEADLINE: Sunday 19th November, 2023 (tentative) (6h with teacher)

ESIEE – Paris

Wednesday 25th October, 2023
Supervisor : Michaël ROYNARD
michael.roynard@epita.fr

Setup procedure

The project is designed for group of 2 to 3 people. The groups are communicated and validated to the teacher (via the spreadshit on the drive.)

The student is encouraged use the squeleton project with a working ECS engine found alongside this subject.

The groups of students will submit on Blackboard an archive containing the source code with the following code layout (and without any binary not the `build/` directory):

```
space-shooter/
├── include/
│   ├── ecs/
│   │   ├── entities/
│   │   │   └── ...
│   │   ├── components/
│   │   │   └── ...
│   │   ├── systems/
│   │   │   └── ...
│   │   ├── entity.hpp
│   │   ├── component.hpp
│   │   ├── system.hpp
│   │   └── manager.hpp
│   ├── scenes/
│   │   └── ...
│   ├── game_state.hpp
│   └── ...
├── src/
│   ├── ecs/
│   │   ├── entities/
│   │   │   └── ...
│   │   ├── components/
│   │   │   └── ...
│   │   ├── systems/
│   │   │   └── ...
│   │   ├── entity.cpp
│   │   ├── component.cpp
│   │   ├── system.cpp
│   │   └── manager.cpp
│   ├── scenes/
│   │   └── ...
│   ├── game_state.cpp
│   ├── main.cpp
│   └── ...
└── CMakeLists.txt
```

Please refer to the CMakeLists.txt file in the bundle to setup the project properly.

Late assignment policy : the work will have a 1 point penalty per day late. After 5 days late, the assignment's grade will be set to 0 unless proper justification is provided (written proof from a doctor, etc.)

1 Introduction to ECS

The Entity-Component-System (ECS) is a prevalent architectural pattern in game development, known for its flexibility, modularity, and performance efficiency. In this pattern:

- **Entities** are general-purpose objects identified by a unique ID.
- **Components** are plain data structures that hold the attributes of entities but contain no business logic.
- **Systems** contain the game's business logic and manipulate entities that have specific components.

Game Loop and Ownership

The game loop and ownership in the ECS architecture are illustrated in fig. 1. The main function, being entrypoint, contains the game loop and owns the manager and the game state (which contains various information like path to the assets or the scene the game is at). The manager, in turn, owns the entities and the systems. The scene registers specific systems and entities in the manager but doesn't own them. Each entity owns its components.
During the game loop:

1. The manager launches the initial scene based on the game state (Step 1 in the schematic).
2. The scene registers specific systems and entities in the manager (Step 2).
3. The manager updates all registered systems, passing all entities to each system. The systems filter the entities by their components and process them (Step 3).
4. The manager collects all dead entities and systems and effectively removes them (Step 4).
5. The game state is checked to determine if a scene change is triggered (Step 5).

External Resources

- Entity component system (wikipedia)
- Understanding Component-Entity-Systems
- Entity Systems are the future of MMOG development
- Introduction to Entity Systems in C++
- The Entity-Component-System – An awesome game-design pattern in C++

Assets online

You can find free assets for your game on the following website (non-exhaustive list):

- OpenGameArt.org
- Kenney Game Assets
- Itch.io
- Freesound.org
- Incompetech
- Piskel (Not a library, but a free online editor for animated sprites & pixel art).
- CraftPix.net

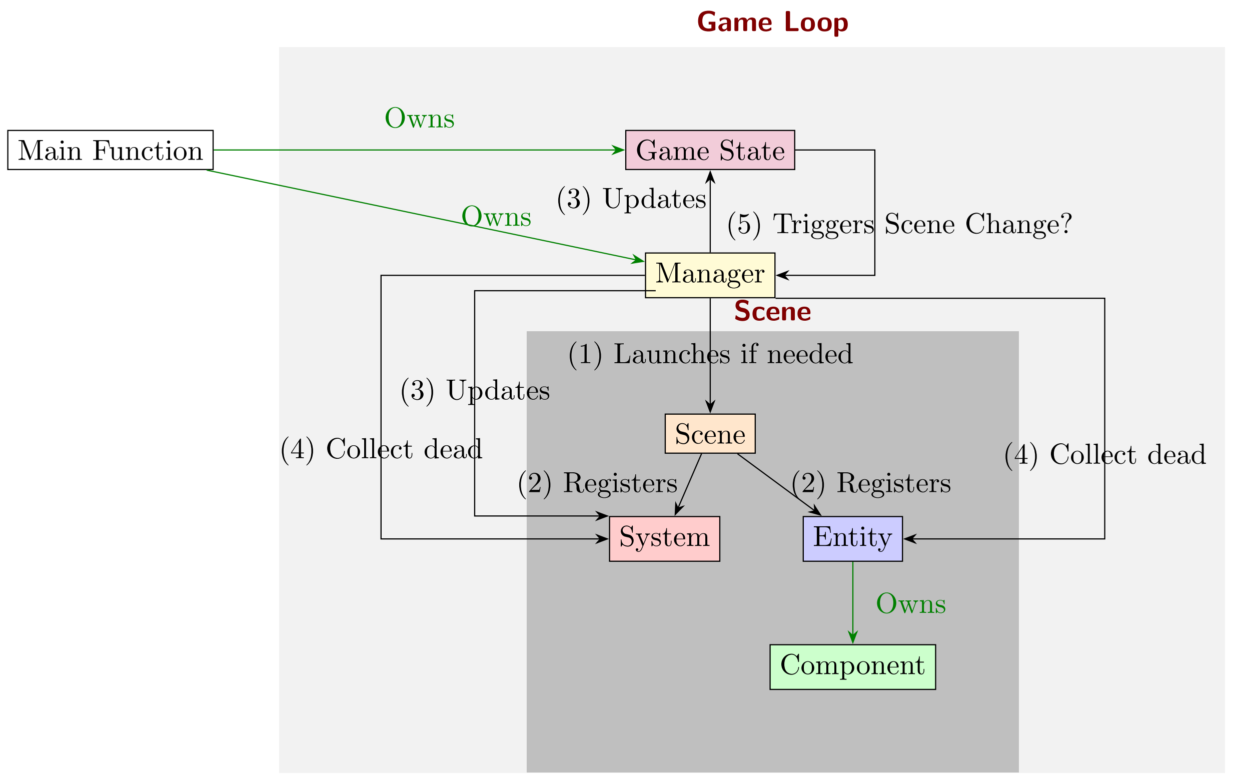

Figure 1: Game Loop and Ownership in ECS

2 Setting up SFML

SFML (Simple and Fast Multimedia Library) is a popular choice for 2D game development and multimedia applications, known for its simplicity and ease of use. It abstracts away many complexities related to OpenGL and window management, making it a great choice for beginners.

Please DO read the turorial: https://www.sfml-dev.org/tutorials/2.6/.

SFML Installation:

Linux: On most distributions, you can install SFML using your package manager:
For Debian-based distributions:

```
sudo apt install libsfml-dev
```

For Fedora:

```
sudo dnf install SFML SFML-devel
```

macOS: Using Homebrew:

```
brew install sfml
```

Windows:

- Download the appropriate version (GCC or Visual C++ and 32 or 64-bit) from SFML's download page.
- Extract the SFML folder to a location on your computer.
- When creating a new project:
  - Add the `include` directory of the SFML folder to your compiler's include paths.
  - Add the `lib` directory of the SFML folder to your compiler's library paths.
  - Link your project with the appropriate SFML libraries. For example, to use the graphics and window modules, link against `sfml-graphics.lib` and `sfml-window.lib`.
  - Copy the DLLs from the SFML `bin` folder to your project's output directory.

Windows WSL: SFML on WSL can be a bit tricky, as WSL doesn't natively support graphical applications. You would need to use an X server for Windows like Xming or VcXsrv. After setting up the X server, you can install SFML as you would on any Linux distribution (e.g., using `apt-get` for WSL with Ubuntu).
However, using SFML on native Windows or Linux is more straightforward and is recommended over WSL for graphical applications.

SFML "Hello World" Example:

```cpp
#include <SFML/Graphics.hpp>

int main()
{
    // Create a window with the title "Hello World"
    sf::RenderWindow window(sf::VideoMode(800, 600), "Hello World");

    // Main loop
    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }

        // Clear the window
        window.clear();

        // (Any drawing code would go here)

        // Display the frame
        window.display();
    }

    return 0;
}
```

Compile using (assuming you have the necessary paths set up):

```
g++ your_filename.cpp -o output_name -lsfml-graphics -lsfml-window -lsfml-system
```

Remember to link against any SFML modules you use (in this case, `graphics`, `window`, and `system`).

3 Project Presentation

Objective

In this project, you will develop a space shooter game using the Entity Component System (ECS) architectural pattern. The features to be implemented have been divided into mandatory and optional categories. Some features depend on others, and this is detailed in a dependency tree presented at the end of this document.

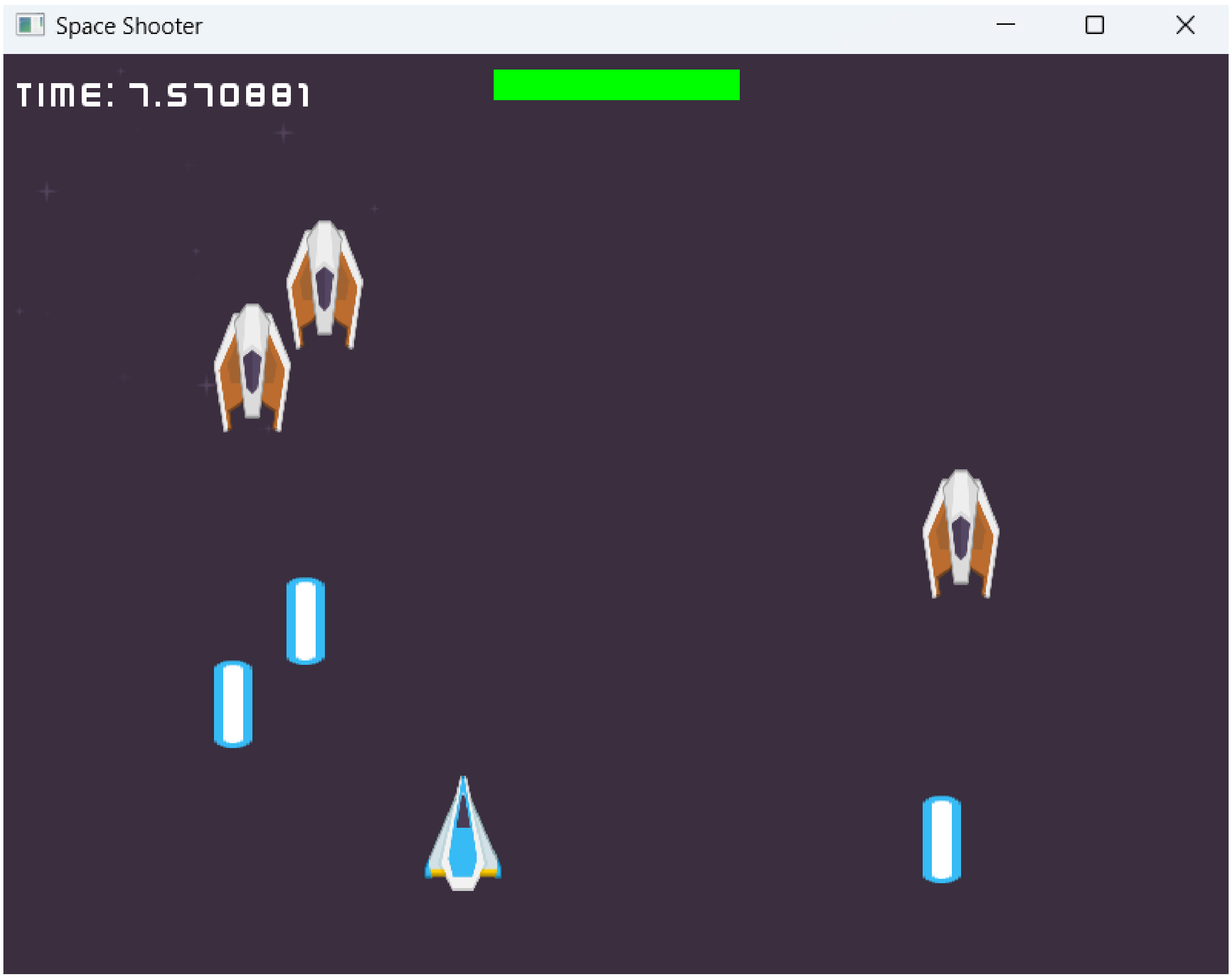

Figure 2: Sample gameplay.

Features Dependency Diagram

Mandatory Features

3..1 Rendering Feature (4pts)

Objective: Develop a system to render textures and sprites on the screen.
Tasks:

- Understand and implement the usage of textures and sprites in the game.
- Ensure proper asset management and positioning within the game window.

Components Involved: TextureComponent, PositionComponent, SpriteComponent.
Entities Involved: Any entity that need to be visually represented on the screen (PlayerShip?).

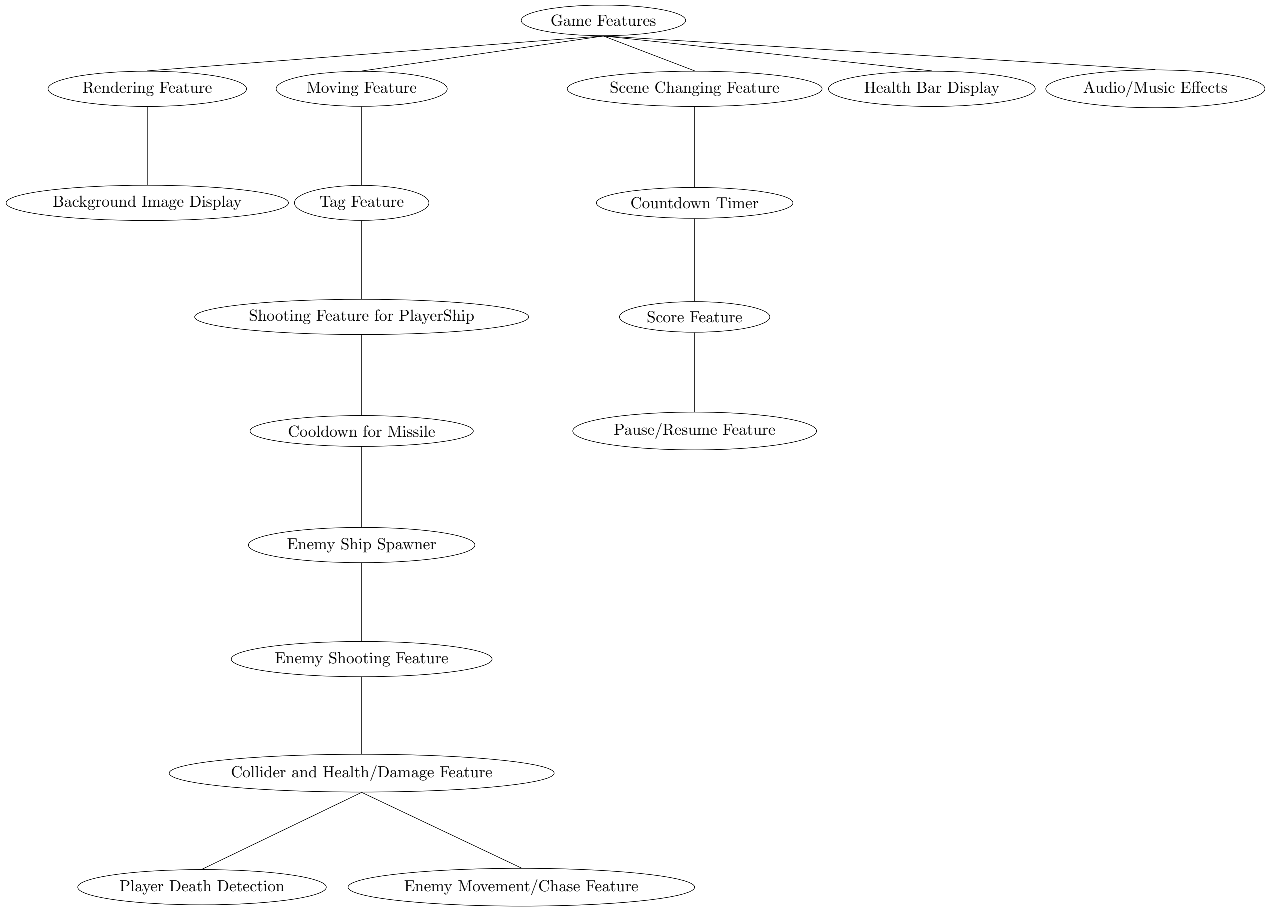

Figure 3: Dependency diagram of game features.

Systems Involved: RenderingSystem.

3..2 Background Image Display (2pts)

Objective: Set the mood of the game by rendering a background image.
Tasks:

- Load and display a full-screen background image.

Components Involved: TextureComponent.
Entities Involved: BackgroundEntity.
Systems Involved: RenderingSystem.

3..3 Moving Feature (4pts)

Objective: Enable player-controlled movement of the spaceship using keyboard inputs.
Tasks:

- Detect and process keyboard inputs.
- Implement a velocity-based movement system.
- Ensure the spaceship does not move outside the boundaries of the game window.

Components Involved: PositionComponent, VelocityComponent, InputComponent.
Entities Involved: PlayerShip.
Systems Involved: MovementSystem, InputSystem.

3..4 Scene Changing Feature (2-3pts)

Objective: Implement a system to transition between different game scenes.
Tasks:

- Use the game state to determine the active scene.
- Notice that the scene changing mechanism behaves like a state machine.
- Handle scene transitions based on game events in the Manager.
- Implement a menu scene before diving into the game (1pts).

Core Code Involved: Game loop, scene manager, game state.

4 Optional (additional) features

4..1 Tag Feature for Entities (2pts)

Objective: Classify entities using relevant tags for better handling in various systems.
Tasks:

- Tag entities appropriately (e.g., "PlayerShip", "EnemyShip", "PlayerMissile", "EnemyMissile", . . .).
- Ensure that any system (requirering the TagComponent) can filter and process entities based on their tags.

Components Involved: TagComponent.
Entities Involved: All game entities that need classification.
Systems Involved: All systems that will need this tag (mainly the ColliderSystem).

4..2 Shooting Feature for PlayerShip (2pts)

Objective: Allow the player's spaceship to shoot missiles.
Tasks:

- Augment input handling to detect shooting commands.
- Create missile entities moving upwards (or in the shooting direction).
- The missile entities need a special process in the MovementSystem (tag) to move without requirering an input.
- Ensure missiles get destroyed once they leave the screen boundaries.

Components Involved: PositionComponent, VelocityComponent, TextureComponent, TagComponent.
Entities Involved: PlayerShip, PlayerMissile.
Systems Involved: ShootingSystem, MovementSystem, InputSystem.

4..3 Cooldown Between Missile Shots (2pts)

Objective: Introduce a delay between consecutive missile shots to avoid rapid-fire.
Tasks:

- Use a timer or clock system to track the time since the last missile was shot.

- Introduce a system that forward the change of delta_time (from the main function) to all active clocks (components) in the game.
- Allow shooting only after a specified cooldown period has passed.

Components Involved: ClockComponent, CooldownComponent.
Entities Involved: PlayerShip, Missile.
Systems Involved: ClockSystem, ShootingSystem.

4.a Enemy Ship Spawner (2pts)

Objective: Introduce enemy ships into the game at regular intervals.
Tasks:

- Create a mechanism to spawn enemy ships from the top of the screen.
- Implement a cooldown mechanism to control the rate at which enemy ships are spawned.
- Introduce an EnemySpawner entity that will own the components related to the clock & cooldown of enemy ship spawning (use a tag)

Components Involved: PositionComponent, CooldownComponent, ClockComponent, TextureComponent, TagComponent.
Entities Involved: EnemyShip, EnemySpawner.
Systems Involved: EnemySpawnSystem.

4.b Enemy Shooting System (2pts)

Objective: Equip enemy ships with the ability to shoot missiles at the player.
Tasks:

- Use a cooldown mechanism to control the rate at which enemy ships shoot missiles.
- Ensure that missiles move downwards towards the player's ship.

Components Involved: CooldownComponent, ClockComponent, PositionComponent, VelocityComponent, TextureComponent, TagComponent.
Entities Involved: EnemyShip, Missileissile.
Systems Involved: EnemyShootingSystem.

4.c Collider and Health/Damage System (4pts)

Objective: Introduce a mechanism where missiles and ships collide, causing damage.
Tasks:

- Implement the AABB collision detection mechanism (the simplified version with the bounding box of the sprite).
- On collision, reduce the health of the involved entities.
- Destroy entities whose health drops to 0.

Components Involved: HealthComponent (ships), DamageComponent (missile), TagComponent.
Entities Involved: PlayerShip, EnemyShip, Missile.
Systems Involved: CollisionSystem, CleanKilledShipsSystem.

4.d Enemy Movement/Chase Feature (2pts)

Objective: Make enemy ships follow or move towards the player's ship.
Tasks:

- Use the player's position to determine the direction in which the enemy ships should move.
- Implement a mechanism that adjusts the enemy ship's velocity based on the player's current position.

Components Involved: PositionComponent, VelocityComponent.
Entities Involved: EnemyShip.
Systems Involved: EnemyChasingSystem.

4.e Player Death Detection (2pts)

Objective: Display a game over scene when the player's health reaches zero.

- Detect when the player's health is zero.
- Trigger a scene change to the game over screen.

Components Involved: HealthComponent
Entities Involved: PlayerShip
Systems Involved: Collider
Core Code Involved: GameState (to handle scene changes).

4.f Health Bar Display (2pts)

Objective: Display the player's health on the screen.

- Create a health bar entity that updates based on the player's health.
- Position the health bar at the top center of the screen.
- You can get the information about remaining health point from another entity via `manager.getFromEntity<E`

Components Involved: HealthComponent, PositionComponent, TextureComponent, SpriteComponent
Entities Involved: PlayerShipHealthBar
Systems Involved: HealthBarDisplaySystem.

4.g Countdown Timer (2pts)

Objective: Display a timer counting down the level's remaining time.

- Start the timer at the beginning of the game level.
- Update and display the timer in real-time.
- Trigger the end of the level when the timer reaches zero.

Components Involved: TimerComponent
Entities Involved: LevelTimer
Systems Involved: TimerSystem, TextRenderingSystem.

4.h Scoring Mechanism (2pts)

Objective: Keep track of and display the player's score.

- Increase the score for various game events (e.g., destroying enemy ships).
- Display the score on the screen.
- Implement a scene change to game over screen when countdown timer reaches 0.

Components Involved: ScoreComponent
Entities Involved: ScoreDisplay
Systems Involved: ScoringSystem, SwitchToScoreSceneSystem.

4.i Pause/Resume Feature (2pts)

Objective: Allow players to pause and resume the game.

- Detect the pause command (e.g., pressing the "Escape" key).
- Freeze all game activity during the pause (use manager.disable/enable systems/entities tools to freeze the game).
- Display a pause menu with options to resume or quit.

Components Involved: InputComponent
Entities Involved: PauseMenu
Systems Involved: InputSystem
Core Code Involved: GameState, SwitchToPauseMenuSystem.

4.j Audio Integration (2–4pts)

Objective: Add audio effects and background music to enhance gameplay.

- Play specific sound effects for game events (e.g., shooting, explosions).
- Play background music suitable for each game scene.

Components Involved: AudioComponent, MusicComponent
Entities Involved: BackgroundMusic, SoundEffects
Systems Involved: AudioSystem, MusicSystem.

Conclusion

Once you have implemented all the mandatory and chosen optional features, test your game thoroughly to ensure smooth gameplay and a bug-free experience. Remember that game development is iterative; feel free to go back, refine features, and even add some of your own to make your space shooter game unique and engaging.

Submission

Compile your code, ensure that all assets are correctly linked, and submit the entire game package in the format specified at the beginning of the subject. Don't forget to include a brief document detailing the features you've implemented and any special instructions for playing or testing your game. The document shall include:

- A brief introduction to your game.

- List of implemented features.
- Any known bugs or issues.
- Special gameplay instructions or controls if any.

Remember, creativity and functionality go hand in hand. Your goal is to create a functional game that is also enjoyable. All the best and happy coding!