

GAP 5503 - Assignment 2

Game Development with ECS and SFML

December 15, 2025

Assignment Notes

Submission Guidelines

- Zip and submit the entire `src` folder.
- Do not add extra files outside the provided folder.
- Include at the top of `main.cpp`:
 - All students' full names, usernames, and student numbers.
 - Notes on partial features that didn't fully work.

Program Specification

Player

- Represented by a shape defined in the config file.
- Spawns at the center of the window initially and after death.
- Moves via WASD keys; diagonal speed must be normalized to maintain a total speed S .
- Confined within the window bounds.
- Shoots bullets toward the mouse pointer with properties defined in the config file.

Special Ability

- Triggered with the right mouse button.
- Spawns multiple entities (bullets, etc.) with unique graphics.
- Implement via a new component added to `ComponentTuple`.
- Include a cooldown timer to prevent continuous firing.
- Bonus: use it for unique game mechanics or visual effects.

Enemies

- Spawn randomly on screen every X frames (from config).
- Must not overlap window edges at spawn.
- Random shape vertices between VMIN and VMAX.
- Random color and speed within config bounds.
- Bounce off window edges.
- Large enemies split into N small enemies on collision with player/bullet:
 - N = number of vertices
 - Small enemies inherit color and vertex count
 - Travel outward in angles spaced $360/N$ degrees
 - Speed should be reasonable

Score

- Each enemy has a score = vertices * 100.
- Small enemies get double this value.
- Score increases when a bullet kills an enemy.
- Display score in top-left corner with font from config.

Drawing

- Slow rotation for all entities for visual appeal.
- Lifespan entities: set alpha proportional to remaining life.
- Optional: visual effects for bonus marks (up to 5%).

GUI (ImGui)

- Toggle systems (except rendering/GUI) on/off.
- List all entities: ID, tag, position.
- Destroy entities via the GUI.
- Adjust enemy spawn interval and manually spawn enemies via GUI.

Miscellaneous

- P key pauses the game.
- ESC key closes the game.

Configuration File

Window Configuration

- W: width
- H: height
- FL: frame limit
- FS: fullscreen (1 = yes, 0 = no)

Font Configuration

- F: font file path
- S: font size
- (R, G, B): text color

Player Specification

Parameter	Symbol	Type	Description
Shape Radius	SR	int	Radius of the player shape
Collision Radius	CR	int	Radius used for collision detection
Speed	S	float	Player movement speed
Fill Color	FR, FG, FB	int,int,int	RGB fill color of the player shape
Outline Color	OR, OG, OB	int,int,int	RGB outline color of the player shape
Outline Thickness	OT	int	Thickness of the outline
Shape Vertices	V	int	Number of vertices used to draw the shape

Enemy Specification

Parameter	Symbol	Type	Description
Shape Radius	SR	int	Radius of the enemy shape
Collision Radius	CR	int	Radius used for collision detection
Minimum Speed	SMIN	float	Minimum enemy movement speed
Maximum Speed	SMAX	float	Maximum enemy movement speed
Outline Color	OR, OG, OB	int,int,int	RGB outline color
Outline Thickness	OT	int	Thickness of the outline
Minimum Vertices	VMIN	int	Minimum number of shape vertices
Maximum Vertices	VMAX	int	Maximum number of shape vertices
Small Lifespan	L	int	Lifespan of spawned small enemies
Spawn Interval	SP	int	Time between enemy spawns (in frames)

Bullet Specification

Parameter	Symbol	Type	Description
Shape Radius	SR	int	Radius of the bullet shape
Collision Radius	CR	int	Radius used for collision detection
Speed	S	float	Bullet speed
Fill Color	FR, FG, FB	int,int,int	RGB fill color of the bullet
Outline Color	OR, OG, OB	int,int,int	RGB outline color
Outline Thickness	OT	int	Thickness of the outline
Shape Vertices	V	int	Number of vertices used to draw the bullet shape
Lifespan	L	int	Bullet lifespan (in frames)

Recommended Assignment Approach and Hints

Step-by-Step Implementation Guide

Step 0: Configuration File

Postpone reading the config until entities are implemented. This simplifies initial testing.

Step 1: Vec2 Class

Implement a simple `Vec2` class with addition, subtraction, scaling, normalization, and length methods. This is used in all movement, collisions, and velocity computations.

Step 2: Entity Management

- Implement `EntityManager::removeDeadEntities()` early.
- Ensure entities can have multiple components safely.

Step 3: Game Class Basics

1. Spawn the player using `spawnPlayer()`.
2. Draw entities using `sRender()`.
3. Spawn enemies using `spawnEnemy()`.
4. Spawn bullets with `spawnBullet()`.

Step 4: Player Movement

- Implement in `sUserInput()` and `sMovement()`.
- Normalize diagonal movement to avoid faster speed.

Step 5: Collision Detection

- Implement `sCollision()`.
- Call `entity.destroy()` for destroyed objects.
- Use circle-circle collision for bullets and enemies.

Step 6: Special Weapon

- Add `CSpecialAttack` component.
- Configure bullets' speed, color, lifespan, and number.
- Ensure bullets spread evenly in a circle around the player.

Step 7: Enemy Splitting

- When a large enemy is destroyed:
 - Spawn N small enemies ($N = \text{vertices}$).
 - Set position = original enemy's center.
 - Compute angle = $i * 360 / N$.
 - Velocity = angle vector * reasonable speed.
 - Half-size, same color, double points.

Step 8: GUI Implementation

- List entities by ID, tag, position.
- Buttons to delete entities.
- Slider or input to change spawn interval.
- Optional: spawn enemies manually.

Step 9: Score Display

- Update `sf::Text` each frame:

```
m_text.setString("Score: " + std::to_string(m_score));
```

- Ensure correct font, size, and color.

Step 10: Lifespan Effects

- Entities with lifespan should have $\alpha = \text{remaining} / \text{total} * 255$.
- Update each frame in `sRender()`.

Extra Hints

- Use SFML's `sf::degrees()` and `sf::asRadians()` for angle conversions.
- When spawning multiple small enemies, loop over vertex count and compute evenly spaced angles.
- For GUI buttons, use `ImGui::PushID()` to avoid ID collisions.
- Always remove dead entities after the main loop to avoid iterator invalidation.
- Normalize diagonal player velocity using:

$$\vec{v} = \frac{\vec{v}}{||\vec{v}||} \times S$$

Tips for Debugging

- Start with a single entity and draw it before adding movement.
- Test bullets and collisions before adding special abilities.
- Use ImGui to visualize entity positions and IDs to help debug.
- Gradually build functionality rather than all at once.

IMGUI Documentation for Assignment 2

Dear student, the following section explains how you can leverage ImGui for creating an interactive GUI in your game, without giving away direct assignment solutions. These guidelines focus on usage patterns, best practices, and relevant API functions.

Basic ImGui Window Setup

- Use `ImGui::Begin()` and `ImGui::End()` to create a window.
- Window title is customizable, e.g.,

```
ImGui::Begin("Geometry Wars");  
...  
ImGui::End();
```

- Everything you want displayed inside the window must be called between `Begin` and `End`.

Tab Bars and Tabs

- `ImGui::BeginTabBar()` and `ImGui::EndTabBar()` create tab containers.
- Individual tabs are created using `ImGui::BeginTabItem()` and `ImGui::EndTabItem()`.
- Useful for separating functionality such as system toggles, entity management, and settings.

Checkboxes and Sliders

- Checkboxes can be used to enable or disable game systems:

```
ImGui::Checkbox("Movement", &m_show_sMovement);  
ImGui::Checkbox("Collision", &m_show_sCollision);
```

- Sliders allow controlling numeric variables like spawn intervals:

```
ImGui::SliderFloat("Spawn Interval", &m_enemyConfig.SI, 0.0f, 160.0f);
```

Buttons and Actions

- Use `ImGui::Button()` or `ImGui::SmallButton()` for actions.
- When the button is pressed, you can trigger game events like spawning an enemy:

```
if (ImGui::Button("Manual Spawn")) {  
    m_manualSpawn = true;  
}
```

- Ensure unique identifiers for buttons in lists using `ImGui::PushID()` and `ImGui::PopID()` to avoid conflicts.

Collapsing Headers and Indentation

- Use `ImGui::CollapsingHeader()` to group related GUI elements, such as entities by tag.
- Indent child elements for clarity with `ImGui::Indent()` and `ImGui::Unindent()`.

Displaying Entity Information

- Text elements can display entity data like ID, tag, or position:

```
ImGui::Text("%d", entity->id());  
ImGui::Text("%s", entity->tag().c_str());  
ImGui::Text("(%.1f, %.1f)", pos.x, pos.y);
```

- Use `ImGui::SameLine()` to display multiple columns on the same row.

Color Customization for Buttons

- You can use SFML colors to customize button appearance:

```
sf::Color c = entity->get<CShape>().circle.getFillColor();
ImVec4 imguiColor(c.r/255.f, c.g/255.f, c.b/255.f, c.a/255.f);
ImGui::PushStyleColor(ImGuiCol_Button, imguiColor);
ImGui::PopStyleColor();
```

- Optionally, set hover and active colors for better feedback using `ImGui::PushStyleColor(ImGuiCol_ButtonHovered, ...)` and `ImGuiCol_ButtonActive`.

Entity Deletion in ImGui Loops

- In this engine, calling `e->destroy()` inside an iteration loop is permitted because entities are not removed immediately. Actual removal is deferred by the Entity Manager until a later phase.
- As a result, it is safe to destroy entities directly from ImGui callbacks without using a temporary container or post-loop cleanup pass.
- The primary concern when deleting entities inside ImGui loops is **ImGui stack correctness**, not container invalidation.
- Any ImGui state pushed before a deletion (e.g. `PushID`, `PushStyleColor`) must be popped before using `continue` or exiting the loop body.
- A safe pattern ensures all ImGui stacks are balanced regardless of whether deletion occurs:

```
ImGui::PushID(e->id());

bool deletePressed = ImGui::SmallButton("D");

ImGui::PopID();

if (deletePressed)
{
    e->destroy();
    continue;
}
```

- This avoids ImGui assertion failures related to mismatched ID or style stacks, such as `IDStack.Size > 1`.

Organizing Large Lists

- Use multiple `CollapsingHeader` sections for each entity type (player, enemy, bullet, small).

- Within each header, display columns consistently using `SameLine()` and properly align IDs, tags, and positions.
- This creates a clear, navigable GUI without clutter.

Best Practices

- Always match `PushID` with `PopID` and `PushStyleColor` with `PopStyleColor`.
- Avoid expensive operations in the GUI loop (like complex vector math); compute values beforehand if needed.
- Keep GUI rendering separate from game logic to maintain clean code structure.

1 Gameplay Timing and Radial Spawning Logic (Tutorial Notes)

This section explains two important gameplay mechanisms used in the assignment:

- Frame-based enemy spawning
- Radial spawning of smaller enemies using geometry and trigonometry

The goal of this section is to help you understand **why** these systems work, not to provide a full implementation.

1.1 Frame-Based Enemy Spawning

Most real-time games do not spawn enemies every frame. Instead, they use a **time-based or frame-based interval** to control pacing.

Core Idea

The game keeps track of:

- The current frame number
- The frame number when the last enemy was spawned
- A configurable spawn interval

Each frame, the game checks how much time (in frames) has passed since the last spawn. If enough frames have elapsed, a new enemy is created.

Conceptual Explanation

Think of the frame counter as a clock that increments once per update. By subtracting the frame number of the last spawn from the current frame number, you compute how long it has been since the previous enemy appeared.

If this difference exceeds a configured threshold, spawning is allowed again.

Why This Works

- This approach avoids using real-world timers, which simplifies debugging.
- It ensures consistent behavior across machines when frame limits are enforced.
- It allows spawn rates to be modified easily through configuration or GUI controls.

Debugging Hint

Printing the current frame to the console is a common debugging strategy. It helps verify:

- That the game loop is running
- That the spawn condition is evaluated correctly

1.2 Radial Spawning of Smaller Enemies

When a large enemy is destroyed, it breaks apart into several smaller enemies that move outward in evenly spaced directions.

This effect relies on **basic geometry and trigonometry**.

Using Shape Geometry

The number of smaller enemies spawned is derived from the number of vertices in the original enemy's shape.

This creates a strong visual and mechanical link:

- A triangle splits into three pieces
- A pentagon splits into five pieces

Dividing the Circle

A full circle contains 360° . To distribute entities evenly around a circle:

$$\text{Angle Step} = \frac{360^\circ}{N}$$

Where N is the number of directions (or children).
Each child is assigned a unique angle:

$$\theta_i = i \times \text{Angle Step}$$

This guarantees rotational symmetry.

Degrees vs Radians

C++ trigonometric functions such as `sin()` and `cos()` expect angles in **radians**, not degrees.

The standard conversion is:

$$\text{Radians} = \text{Degrees} \times \frac{\pi}{180}$$

This conversion is a requirement of the math library, not the game logic.

Direction Vectors

Using cosine and sine together produces a **unit direction vector**:

$$\vec{d} = (\cos(\theta), \sin(\theta))$$

This vector:

- Always has length 1
- Points outward from the center at angle θ

Scaling this vector by a speed value converts it into a velocity.

Design Decisions (Not Math)

Some values are intentionally adjusted for gameplay reasons:

- Smaller enemies often move slower or faster than their parent
- Collision radius is reduced to match visual size
- Lifespan is added so fragments do not persist forever

These are **design choices**, not mathematical requirements.

Visual Consistency

Smaller enemies inherit visual properties from their parent:

- Fill color
- Outline color
- Number of vertices

This helps the player instantly recognize that these entities came from the same source.

Score Scaling

Fragments are often worth more points than their parent. This introduces:

- Risk–reward tradeoffs
- Incentives for skilled play

The math here is trivial; the design impact is not.

Key Takeaways

- Frame counters can act as reliable timers
- Even angular spacing comes from dividing the circle
- Trigonometry converts angles into movement directions
- Many numeric values exist for game feel, not correctness

Understanding these principles allows you to adapt the system to new mechanics without rewriting everything from scratch.