Naam: Yassine El Abdellati

Begeleider: Tim Apers

Deze game is een 2D platformer(individual platformer) waar dat je 3 targets moet beschieten om naar de volgende level te kunnen geraken, dit doe je door op slimme manieren door de puzzle te gaan, je moet spikes ontwijken, walljumps gebruiken, springs gebruiken, en de targets schieten om dan naar de finishlijn te gaan.

Taal en technologie: C++, SFML

Ik heb de entity component system gebruikt.

Dit bevat entities, components en systems.

Een entity is gewoon een integer het bevat zelf geen logica.

Een component is in mijn geval een struct die een bepaald aspect van een entity beschrijft zoals ,

```
struct Velocity {
	float dx = 0;
	float dy = 0;
};
```

Systems voeren logica uit op entiteiten die bepaalde logica van components hebben zoals,

```
void ProjectileSystem::update(World& world, float deltaTime) {
	for (auto entity: entities) {
		if (!world.positions.contains(entity) ||
			!world.velocities.contains(entity))
			continue;

		auto& position = world.positions[entity];
		auto& velocity = world.velocities[entity];

		position.x += velocity.dx * deltaTime;
		position.y += velocity.dy * deltaTime;
	}
}
```

Dus hoe werken ze samen, een voorbeeld:
De speler drukt op de up arrow, de inputsysteem detecteert dit, de jumpsysteem kijkt of de speler mag springen en gaat dan de velocity aan passen als die springt, de physicssystem voegt de zwaartekracht toe, collisionsysteem kijkt of hij ergens tegen aan springt en de rendersysteem gaat de speler tekenen op de nieuwe posities.

Systemen:
Ik heb 10 systemen geimplementeert:

De camerasystem volgt de speler zodat de scherm correct is, je kan ook horizontalscrollen via een camera, als die de speler volgt.

De inputsystem registreert keyboard input en gaat die vertalen naar acties zoals bewegen en schieten.

De movementsystem gaat de snelheid en positie van de speler entiteit aanpassen op basis van de inputs.

De collisionsystem detecteert botsingen tussen entiteiten met behulp van de Quadtree.

De jumpsystem bepaalt of een speler kan springen en gaat de speler zijn positie en velocity aanpassen als die springt, die bepaalt nu ook of dat je mag dubbel jumpen.

De physicssystem past de zwaartekracht toe op de speler.

De projectilesystem maakt de projectile aan en ook zijn lifetime en positie.

De shootsystem detecteert wanneer je wilt schieten en gaat dan een projectile aan maken.

De EnemySystem gaat in de verschillende enemystates.

De Rendersystem tekent alle entiteiten op het scherm.

Design patterns:

Ik heb de factory pattern gebruikt om de implementaties aan te maken zoals een wall:

```
Entity ConcreteFactory::createWall(World& world, float x, float y, float width, float height) {
	Entity wall = world.createEntity();

	world.transforms[wall] = Transform{.position = {x, y}};
	world.colliders[wall] = Collider{
			.bounds = {x, y, width, height},
			.isStatic = true
	};
	world.positions[wall] = Position{x, y};
	world.sizes[wall] = Size{width, height};
	world.renderables[wall] = Renderable{{width, height}, "ecsFolder/Resources/Pics/blue.png"};
	world.wallTags.insert({wall, WallTag{}});

	return wall;
}
```

Ik heb dan ook de builder pattern gebruikt om de entiteiten aan te maken:

```
WallBuilder::WallBuilder(World& world, AbstractFactory& factory)
		: world(world), factory(factory) {}

WallBuilder& WallBuilder::makeWall(float x, float y, float width, float height) {
	entity = factory.createWall(world, x, y, width, height);
```

```
    return *this;
}

Entity WallBuilder::build() {
    return entity;
}
```

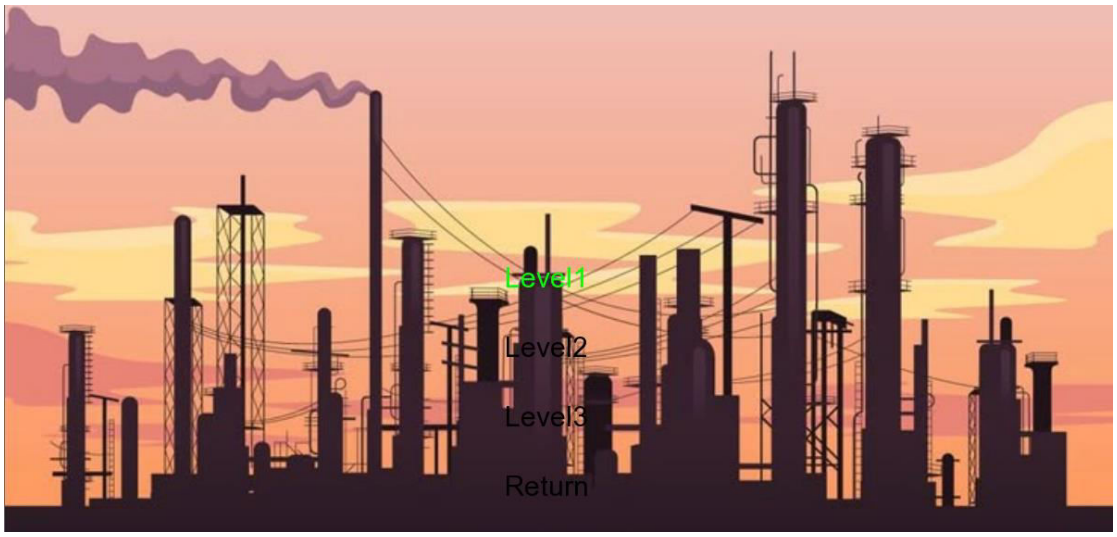
Zo kan je makkelijk een wallbuilden en is het ook duidelijk wanneer iemand naar je code kijkt voorbeeld van implementatie in world:

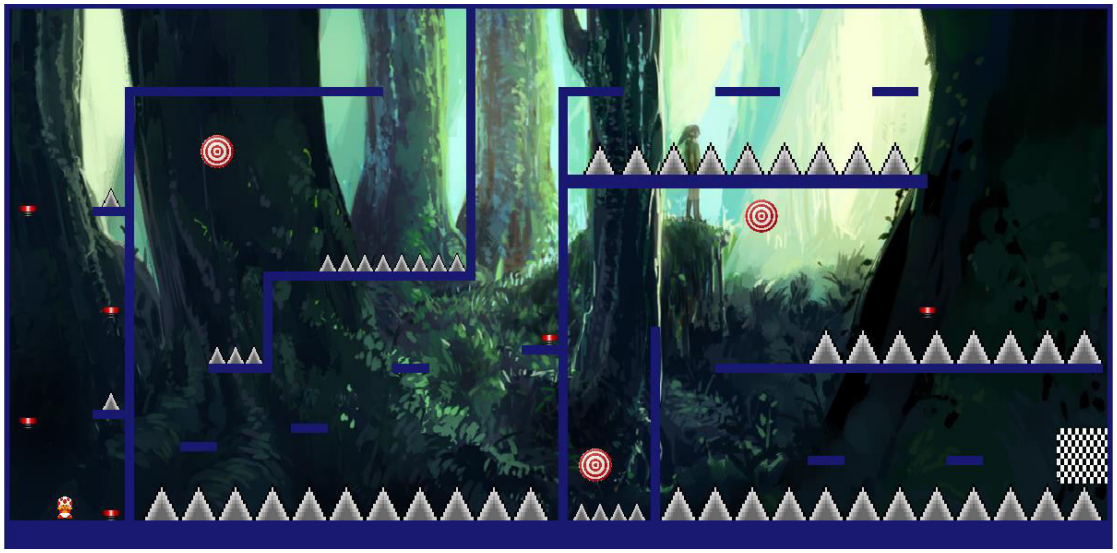
```
Entity World::buildWall(float x, float y, float width, float height) {
    Entity wall = WallBuilder(*this, abstractFactory)
            .makeWall(x, y, width, height)
            .build();

    if (collisionSystem) {
        collisionSystem->AddEntity(wall, *this);
    }

    return wall;
}
```

Alleen maar makewall() en build(), vrij simpel om een entity te maken door dit.

Ik heb de state pattern gebruikt voor de verschillende states van de enemy,

Ik heb ook de state pattern gebruikt om een menu te maken waar dat je op play en quit kan clicken, een levelmenustate waar dat je kan kiezen welke level je wilt doen als je niet van level 1 wilt beginnen, een levelstate, wat je level/world is.





Een stopwatch, singleton pattern, je wilt dat alles gelijkmatig beweegt dus bijvoorbeeld als je beweegt met je speler. zonder de stopwatch zou mijn speler sneller bewegen op verschillende computers. De stopwatch berekent de tijd tussen frames zodat ik de beweging kan schalen op basis van die tijd.

Collision tree, ik gebruik de quadtree om collision detection efficienter te maken, want ik wil niet elke keer naar alle entiteiten op mijn scherm kijken of er een collision is, dit kan ervoor zorgen dat mijn wereld erg traag wordt. De quadtree gaat mijn wereld recursief in 4 delen verdelen, entiteiten worden op basis van hun positie in een quadrant gestoken, bij het controlleren van een collision bekijken wij alleen maar de andere entiteiten binnen de quadrant in plaats van alle entiteiten.

Conclusie en Reflectie:

Ik kreeg een diepe inzicht in de Entity Component System.

Ik leerde meerdere designs patterns implementeren.

Ik optimaliseerde performence door gebruik te maken van een quadtree.

En ik verfijnde mijn vaardigheden in c++ en SFML.

Ik heb geleerd om met sprints te werken.

Ik heb geleerd hoe dat ik een game via ECS van scratch moet maken.

Bijlage:

https://www.umlboard.com/design-patterns/entity-component-system.html