Space Impact

Introduction

The video game that we decided to develop for this project is a simplified version of the old *Nokia*'s video game called *Space Impact*:

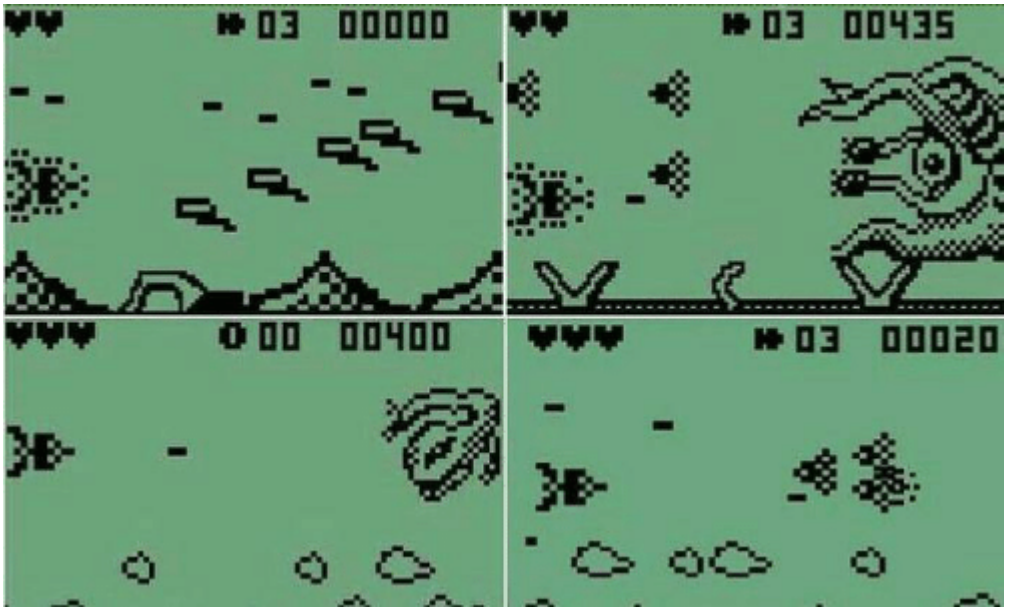

Image 1: Different snips of the *Nokia*'s *Space Impact*

In this game, the player is a spaceship that navigates through space while enemy hordes appear, which they must either eliminate or dodge. The goal is to complete several levels at the end of which there is a boss battle. In our version of the game, the objective will be to survive as long as possible while destroying as many enemies as possible.

Game Stages

Main Menu

This is the game's starting stage. It presents the game and it gives the player options to start playing or to quit the game. This is how this stage will look like:

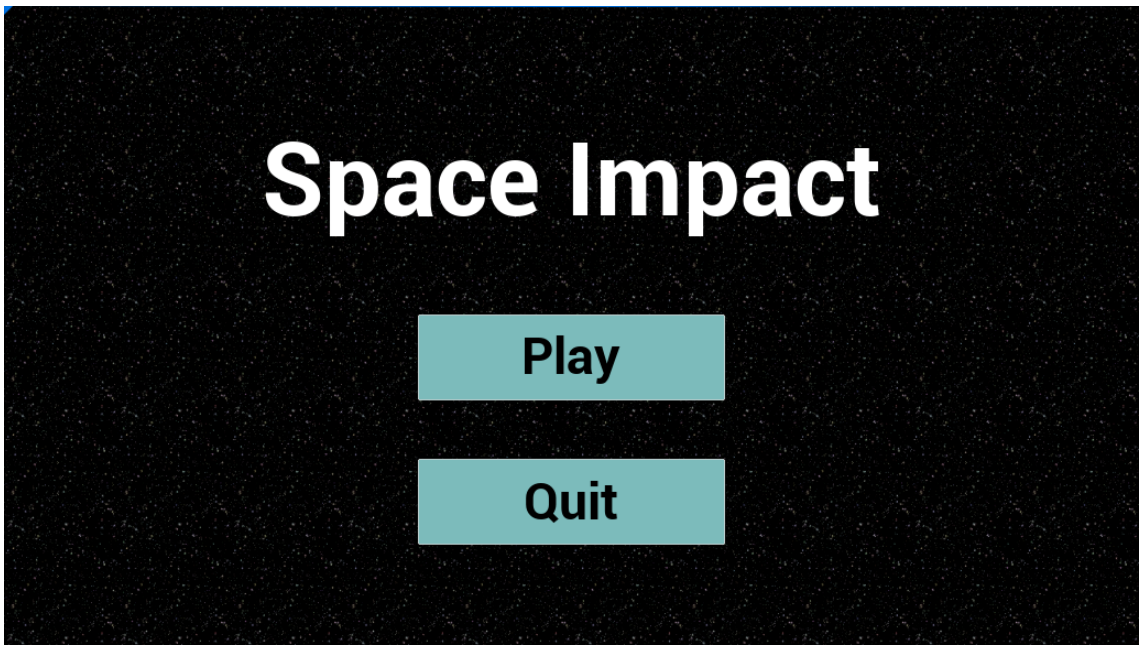

Image 2: Appearance of the game's stage *Main Menu*

It is made of an empty scene with a background that has a space texture and a widget with all the UI.

Game

This is the stage where the player will play. It shows the world where the action is happening and a HUD that displays the game information useful for the player: their health and score. This is how this stage will look like:

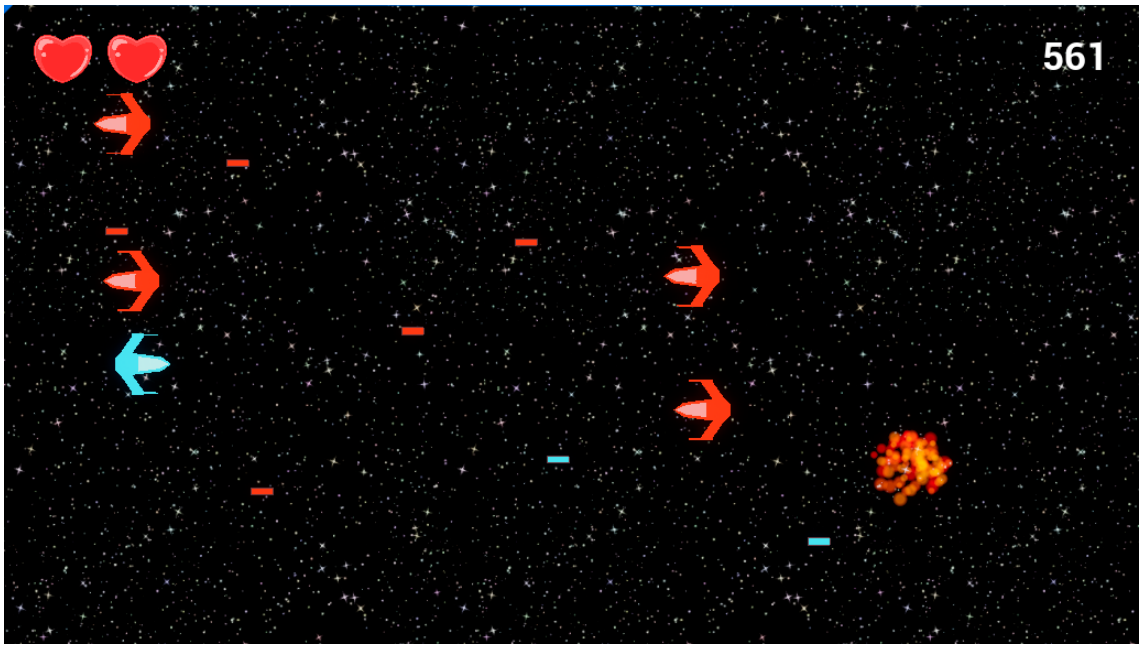

Image 3: Appearance of the game's stage *Game*

End of Game

This is the stage shown when the game is over, i.e. when the player has run out of lives. It shows the player's final score and gives the player options to start playing again or to quit the game. This is how this stage will look like:

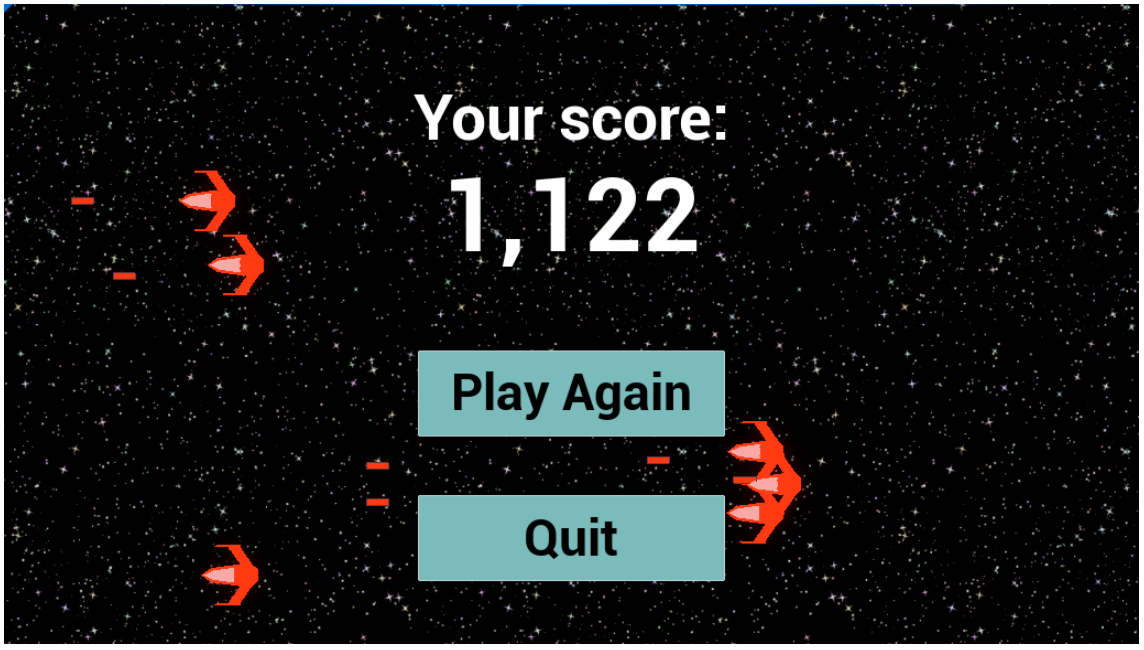

Image 4: Appearance of the game's stage *End of Game*

How the game works

Game Actors

There are two types of game actor, and each will have a collision profile associated: *Player*, represented as blue; and *Enemy*, represented as red. There are two types of game actors: pawn and bullet. All pawns use the same spaceship static mesh with different materials for the ship's metal and glass parts. Pawns shoot bullets. All actors with the same collision profile will not detect overlaps between each other. When an overlap is detected between game actors with different collision profiles, both are subtracted a value of 1 to their health. If any game actor's health gets to 0, it explodes and, if that's the case of the player pawn, the game ends. The explosion of a game actor will trigger a burst PFX (common to all game actors) and a SFX (different for pawns and bullets).

Player

The player pawn will move with WASD or the arrow keys and shoot with Space. It will be surrounded by a hollowed blocking volume that defines its movement range. This volume will only detect collisions with actors of collision profile *Player*. As the

player pawn and its bullets share this collision profile, the pawn will detect collisions with WorldStatic while the bullets won't. This will make the bullets able to not collide with the blocking volume. The player will gain score passively through time and actively by destroying enemy pawns or bullets. Every time that happens or the player loses health, the UI will immediately update.

Enemy/AI

At the right of the screen there will be an actor spawner that regularly spawns 3 pawns of type enemy oriented to the left. They will regularly fire bullets that the player will have to dodge. When they are created, they calculate a target Z position that they will approach during their life cycle, which will make their movement non-linear. This is the formula used for this calculation:

```
TargetHeight = OscillationHeight + FMath::RandRange
(-OscillationAmplitude, OscillationAmplitude);
```

And this is how they will move at each frame:

```
      MoveForward(1.f);
      const float HeightDifference = GetActorLocation().Z -
TargetHeight;
      const float AbsHeightDifference =
FMath::Abs(HeightDifference);
      const float CorrectionFactor = -OscillationStrength *
(AbsHeightDifference / HeightDifference) *
FMath::Sqrt(AbsHeightDifference / OscillationAmplitude);
      MoveRight(CorrectionFactor);
```

This way of moving will make the pawn move faster the further it is from the target height. This curve shows the value of the *CorrectionFactor* depending on the distance between the current and the target height:

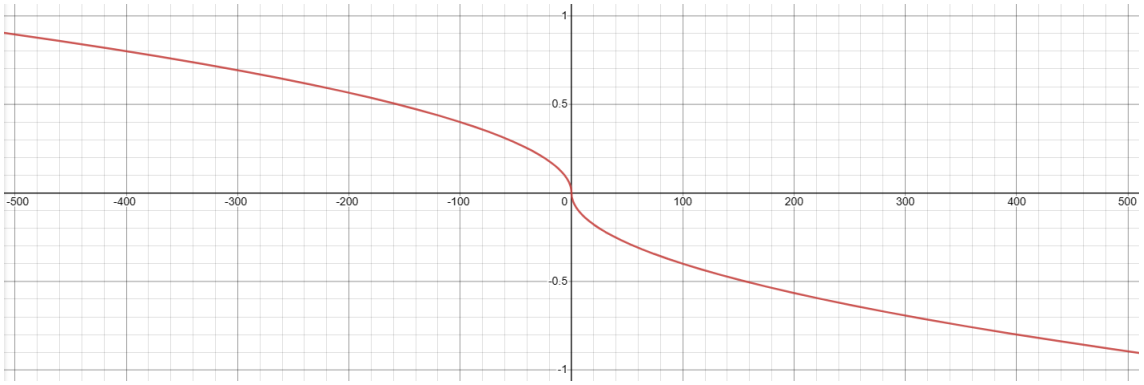

Image 5: Curve showing the relation between the current and target height difference and the CorrectionFactor in an enemy pawn

Bullet

Bullets will move linearly in their forward direction until they overlap with either a game actor of different type or an actor destroyer of the same type.

Actor Management

Actor Destroyer

There is a volume called actor destroyer at the left and right of the screen, out of the game's vision range, that prevents all actors from navigating indefinitely through the space because, once they exit the game screen, they will no longer be part of the game's action. As the player actors only move right and the enemy actors only move left, the actor destroyer at the left of the screen will only detect collisions with *Enemy* actors and the one at the right of the screen will only detect collisions with *Player* actors. As soon as this actor detects an overlap with any actor, it will destroy that actor.

Spawn Actor Scene Component

This Scene Component is used by the Actor Spawner and will define the locations where it can spawn actors. The Actor Spawner can have as many instances of this component as we want to and locate them wherever we want.

Actor Spawner

This actor will regularly spawn as many actors as it is configured to spawn and will choose randomly as many Spawn Actor Scene Components as necessary to be the new locations for the newly spawned actors. The class of the spawned actors will be configured as well, though in our case we will just need an Actor Spawner that spawns enemy pawns.

Materials

We created two materials: an unlit emissive for the ships and bullets and an unlit emissive with a space texture for the main menu and the game's backgrounds.
The first material has four instances: for the player's metal parts, player's glass parts, enemy's metal parts and enemy's glass parts. The player MI's are blue and the enemy MI's are red, while the glass parts MI's have a lighter color than the corresponding metal parts.
The second material has two instances: a dynamic MI that multiplies the horizontal UV of the texture with a time scale factor which in our case is 0.07 and a static MI, whose time scale factor is 0. The *Main Menu* stage uses the static MI while the *Game* stage uses the dynamic MI.

References

In this document

Image 1: https://pics.me.me/i-used-to-love-playing-space-impact-on-the-old-71722816.png
Image 5: Graph built using Desmos https://www.desmos.com/calculator/

Game assets

SM_SpaceShip: https://free3d.com/3d-model/low-poly-spaceship-37605.html
SW_ExplosionBullet: https://opengameart.org/content/explosion-8
SW_ExplosionShip: https://opengameart.org/content/explosion-0
SW_LaserFire: https://opengameart.org/content/sci-fi-laser-fire-sfx
T_Heart: https://opengameart.org/content/heart-7
T_Space: https://opengameart.org/content/seamless-space-stars