

Project Report

Part 3 - Interim Demo

STATIC

Computer Games Laboratory
Winter Term 2021/2022

Team Well-Meaning Scorpion

Matthias Ellerbeck
Musfira Naqvi
Mert Ülker
Doğa Yüksel

Overview

We choose to implement the game in the Unity game engine. Through university courses we are all familiar with it, in addition it is mature and requires no fees. We decided on version 2021.2.4f1. Long term support is not necessary for our project and we may benefit from the features added since the last LTS version. We also decided to use Unity's new scriptable render pipeline in its URP implementation and the new input system to familiarize ourselves with it.

Current Status

Player

Natural and precise movement is important for the player to feel in control of the player character in a first person shooter. Movement without ramp up feels choppy; at the same time realistic physics, i.e., force based acceleration, feels too weighty and slow. Continuously interpolating between the current and the target velocity avoids both. We also granted the player character the ability to sprint forward. We used the built-in character controller of Unity¹ to keep full control over the player character's movement. To emphasize the ground movement of the player character we introduced slight camera movements to simulate steps; these are synchronized to footstep sound effects.

In addition to player movement, we also integrated the most essential shooter mechanics into our interim demo. These mechanics are heavily based on traditional shooter mechanics and offer nothing particularly innovative. The basic player-weapon interactions include aiming down sight, reloading and shooting, where we additionally utilized multiple crosshair textures to enhance hip firing. Furthermore, we employed a set of audio clips for shooting, reloading, dry firing and added randomness to the fire sound in order to introduce more variance. Finally, we prepared some simple particle effects for the muzzle and hit effects in order to further enhance our shooting mechanics.

Enemies

Art

The team decided to go with the traditional silhouette of the slender man for every enemy type with slight variations regarding the enemy type. We also decided to adopt a non-realistic art style, mainly due to the fact we wanted to design most of the enemy models ourselves and achieving a realistic style requires more effort. All of the modeling, rigging, texture mapping and animating for the current enemy types was done by the team using Blender. During the design process, we also made sure to have an art style which would be compatible with the dither effect we want to incorporate into our game later on.

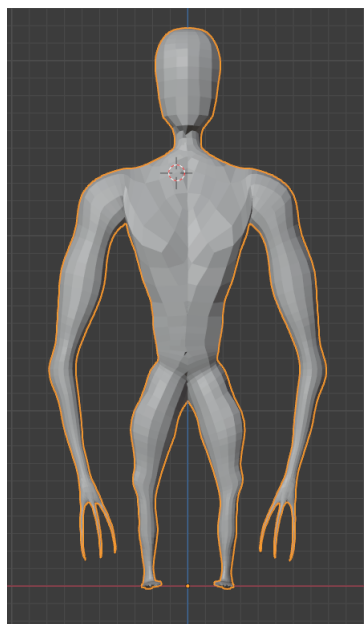
¹ <https://docs.unity3d.com/Manual/class-CharacterController.html>

Base Enemy:

This enemy type's geometric features are based on the original slender man with some additional refinements in order to improve the original design while staying faithful to the theme. While designing this enemy, we made sure to overemphasize some body features such as long arms and claws, giving our model a lanky but scary look. Although this model does not have a texture mapped to it yet, we envision a mostly neutral texture with some details such as scarring in order to maintain the raw and creepy look of the enemy.

Boomer Enemy:

The Boomer enemy (right) has a more broad figure. During the modelling phase it was important to underline its ability to explode any moment - which is how the bulky shape came into place.



Behavior

Base Enemy:

These enemies (left) have the most basic behaviour pattern and only pose a challenge for the player with their numbers, overwhelming the player quickly. These enemies will move around the level randomly until they can sense the player is closeby, and only then will try to follow them. They will close in on the player and start attacking if they are not in the player's vision, and provide the most basic obstacle for the player in getting to the end of the level.

Boomer Enemy:

It will explode when in close proximity to the user. If the player looks at the boomer enemy after the detonation, they can still halt the explosion. With this, the player can get themselves to safety by increasing the distance to the boomer enemy. After being in a secure area, they can move their view away from the enemy to let the explosion continue without the player being hurt. Thus this enemy can become very dangerous if the player let's it get close to them, however it can also become a useful tool in dealing with crowded groups of enemies if the player can avoid the area of explosion.

Physics Interaction

The physics interaction consists of two steps: First, we check if the enemy is visible to the player, i.e. visible to the camera. Then, depending on visibility and the enemy's current state, we freeze or unfreeze the enemy.

We identified three approaches to determining if the enemy is visible to the camera.

- (1) Use the built-in occlusion culling system and check if Unity renders the enemy.
- (2) Determine the enemy's view space position, then cast a ray between the camera and the enemy using the physics engine.
- (3) Check the renderer result.

These approaches differ in accuracy, complexity, performance, and feasibility. We do not know the performance impact of these approaches as we only implemented (2) for the demo. For accuracy we define the following:

	System says visible	System says not visible
Enemy visible	True Positive	False Positive
Enemy not visible	False Negative	True Negative

We want to minimize false positives. The player will most likely not notice when enemies do not move while not visible, but will notice visible enemies moving.

For (1) we expect no false positives and some false negatives as the purpose of occlusion culling is to reduce the render load by culling invisible objects without the player noticing. Implementing this approach would be easy as the script only needs to check if the enemy is being rendered by unity. Unfortunately, Unity's occlusion culling system requires data to be prebaked into the scene². To dynamically load and merge levels we need to do more research on Unity's scene management.

Approach (2) results in no false negatives but many false positives. A thin object may block the ray but not impede visibility enough to hide the enemy. It is slightly more complex to implement than (1) but can still be done in a few lines of code.

Approach (3) is naturally accurate but very complex. It works as follows: First, let Unity render the entire scene without enemies. Then, while rendering the enemies, draw a black and white pixel mask into an empty second render target. Finally, sum up the mask's pixels with a compute shader and transfer the resulting integer to the CPU.

Due to time constraints we currently use (2) but will try to implement (3) for the alpha release for accuracy but also to implement a technical achievement.

After determining visibility we freeze or unfreeze the enemy. To do this we must stop any physics calculations and suspend any scripts while still rendering the enemy. We achieve this by constraining all movement and rotation of affected rigidbodies and manually disabling any behavior component on the object. Before that, we save the enemy's state to restore it when it is unfrozen.

² <https://docs.unity3d.com/Manual/OcclusionCulling.html>

Level

For the demo level, we used assets from Unity's asset store; except for the level geometry itself. The URP only supports a limited number of light sources per object, which is easily visible due to the lack of a main light. We therefore cannot use large geometry which would be influenced by too many lights. To texture the level, we created custom shaders using Unity's shader graph. These shaders compute texture coordinates from the world space position and orientation of the current fragment.