UViC CSC 205 - “Introduction to Game Development” Student Game Project - *Gate Defender*

Mason Gordon, Brenden Ward, Robert Craig

ABSTRACT

“Gate Defender” is a student project for CSC 205 at the University of Victoria. The project is a tower defense style game that explores AI systems; 3D modeling, skinning, and animation techniques; 2D asset creation and implementation in level and UI design; and the integration of audio and visual effects.

1 INTRODUCTION

Tower defense is a subgenre of real-time strategy video games. The goal is to defend a particular point against waves of increasingly difficult enemies by constructing towers to impede and destroy enemies. Currency is earned from destroying enemies, and allows the player to purchase more advanced towers or upgrade existing ones. The game is lost when too many enemies make it through the player’s defenses, and is won when the player successfully defends against a predetermined number of waves.

2 GAMEPLAY

The goal of the game is to prevent enemies from travelling from the top of the game level to the bottom for 20 waves. Three types of turrets are available to hinder the enemy’s advance, and each type has 3 tiers of upgrades. Each turret type is unique in its functionality and effects, and has particular strengths and weaknesses against certain enemies, encouraging the player to diversify their defenses. Destroying enemies will earn the player money, which can be used to buy new turrets or upgrade existing ones, or the player can sell existing turrets to recoup some of the cost.

3 FEATURES

3 maps of varying difficulty including custom textures with displacement and normal maps.
3 unique tower types, each with 3 tiers of upgrades. Include custom models, textures, sounds, and effects, as well as animations for firing, deploying, and undeploying.
4 types of enemies, each with 4 subtypes. Each type has its own strengths and weaknesses, and unique stats.
Procedural wave-spawning algorithm that ensures that waves increase progressively in difficulty, but allows for randomness in enemy spawning and consistent spawn placement of enemies.
Detailed, dynamic UI with custom assets.
4 game speeds, to enable fast forwarding and pausing of the game; accessible from the UI and keyboard shortcuts.

4 TECHNICAL BREADTH

The focus of this project was placed on AI, rendering, audio, animation, and collisions. Dynamics and networking were not emphasised as the game genre does not provide many opportunities for having complex dynamic systems, and implementing networking functionality was deemed too complex and time consuming for the scope of the project.

4.1 AI

We believe we have exceeded the requirements.
AI was required for the functionality of enemies, wave spawning, and tower behaviours. Enemy AI consists of linear navigation of a waypoint network. Wave spawning utilizes probabilities for each enemy (updated each wave) to determine which enemies to spawn. The algorithm then ensures that enemies are spawned gradually without overlapping or clumping. Towers are entirely dependant on automatic systems, as players only construct or deconstruct them. As such, enemy acquisition, targeting, attacking, sounds, and visual effects are managed via script.

4.2 Collisions

We believe we have met the requirements.
Collisions are not commonly required, but the instances in which they are include: tower acquisition of enemies within range, tower projectile damage distribution, and enemy waypoint arrival. Unity OnTrigger functions are used to determine when collisions occur and invoke subsequent logic, except in the case of waypoint arrival which instead uses a distance comparison.

4.3 Audio

We believe we have exceeded the requirements.
Unique sound effects include: tower building, upgrading, and selling; enemy death; tower firing (unique to tower type); UI interaction; and victory/defeat sounds.
The audio management system uses a request confirmation/denial scheme to regulate the number of

sounds playing, and ensures that there is an even distribution of sound types, without playing too many sounds and overwhelming the player. With the ability for the player to change game speed, audio effects must be able to modify their pitch depending on the speed of the game, or be muted in the case that the game is paused. This is achieved through scripting.

4.4 Rendering

We believe we have exceeded the requirements.
3D Models were modeled in low-poly and UV mapped. Extra details were added to a high-poly mesh. The mesh was used to bake various texture files for the lower-poly model. Textures used on most models were diffuse and emission maps. Other textures include normal and displacement maps which were used where appropriate such as on game level textures. A custom shader was developed based on the Unity transparency shader for the tower spawning blueprint effect, as the default transparency shader rendered transparent faces additively, which was not desirable.

4.5 Animations

We believe we have exceeded the requirements.
Non-procedural animations for articulated models managed with Unity's Mecanim system. Articulated models use Bone Assignment for skeletal animations. These include animations for tower firing, deploying, and undeploying.
Procedural animations run via scripts for movements of rigid models (enemies), specific bones of articulated models (towers), pulsation effects on emissive materials, and UI animations.

5 TECHNICAL DEPTH

While we believe we provided depth beyond the scope of the course in all aspects apart from collisions, the greatest depth is achieved in the design and implementation of our AI systems, most notably in the wave generation algorithm.

Wave Generation
Wave generation relies on the use of coroutines to control timing and distribute tasks over multiple frames. An array of probabilities is maintained with an entry for each enemy type. Initially, the array indicates a 100 percent chance for the simplest enemy, and each wave adjusts the table to introduce more difficult enemies as the game progresses, and also to remove obsolete enemies. Two queues are maintained, one containing references to the the next wave's enemies (hereby *NWQ*), and the other containing references to the current wave's enemies (hereby *CWQ*). Upon wave start, the *NWQ* is loaded into the *CWQ*, and while the current wave is deploying, the new *NWQ* is constructed over the course of multiple frames to distribute the workload.
As enemies (of varied sizes) are dequeued from the *CWQ*, they are fit onto a line along the start point of the path. Once the line is saturated, enemies are spaced evenly along it to provide symmetry, and the line is deployed. This system allows for random line composition, and can adapt to any permutation of enemies in the wave queue. A brief wait occurs to provide spacing and subsequent lines are deployed until the wave queue is empty, at which point a timer begins to delay the next wave (skippable), after which the process repeats with the pre-generated *NWQ* now utilized as the *CWQ*.

6 CONCLUSION

We believe our game exceeds requirements in all areas with the exception of collisions, and in doing so provides an engaging and fun experience. Notable mentions that were not required aspects include UI design and implementation, and the creation of assets for UI elements, towers. enemies, game levels, and visual effects. All assets were developed by the team with the exception of sounds.

6.1 WORKLOAD

Mason Gordon - V00835221
50%

Brenden Ward - V00234857
30%

Robert Craig - V00832582
20%