# Developing a Switch-Accessible Tower Defense Game in Unity 3D

For a game that primarily relies on mouse accessibility, creating a keyboard -only version is no trivial task, especially if you reduce the number of possible inputs down to two switches/keys. In my implementation, every game object is loosely tied to a grid system, mainly to facilitate user interaction with only two inputs. Every element in the scene is divided up into layers, so the GUI is one layer, the towers and enemies are another, and so forth to prevent unwanted activity between physics colliders (more on that later) and to allow for layered rendering, or the ability to render any type of artistic graphics over another layer. Enemy pathfinding relies on a system of waypoints, a simple routing mechanism where enemies progress by simply following one waypoint after another. And finally, input is handled by identifying two different inputs: a mover, where the user can cycle through menu elements, and a chooser, which allows a user to activate a particular functionality.

In designing the tower defense game, I chose to stick with a 2D birds-eye perspective. In Unity, the game environment is solely 3D with a camera that is by default set to a 3D perspective. Given that there is no native 2D support in Unity as it is a 3D game engine, I had to emulate a 2D perspective using 3D objects. First, the camera had to be changed so it wouldn't render in 3D, but in 2D. The default camera (and any camera) can actually render the viewable scene in two ways: perspective, which is the default 3D view, and orthographic, which views the scene in 2D and is the view mode of the tower defense game camera. Next, 2D art and textures had to be somehow rendered in a 3D world. To do so, I utilized a CreatePlane script (under the "Editor" folder) to create a 2D plane mesh, similar to Unity's built-in Plane primitive, but with only two polygonal triangles to reduce rendering resources. Art is applied to these planes via textures, Unity's jargon for imported graphics, and is applied as easily as dragging and dropping the textures onto the appropriate object in the scene. One issue I had with Unity importing graphics is that Unity automatically scales the graphics to the nearest power of 2. To disable that unneeded functionality, every affected texture must set the setting "Non Power of 2" to "None". Otherwise, rendered textures will not reflect the proper scaling of the original.

As a digression on textures, I found it troublesome to introduce pixel-perfect graphics as pixel-perfect textures. As it turns out, achieving pixel-perfect graphics involves a bit of mathematical manipulation and an understanding of how cameras work in Unity. To avoid unneeded effort in disabling Unity's automatic texture scaling, I opted to set one world unit, the length of any side of a Unity Cube primitive, to 32 pixels. The resolution of my game is 704x608 (Edit > Project Settings > Player), so my game scene in 32x32 tiles/world units is 22x19 tiles/world units. When the camera is set in orthographic mode, it has a viewport "size" associated with it. For reasons unknown to me, the size is best set at ½ of the scene height in world units, so my camera size is 9.5.

Moving on from understanding the basics of Unity graphics, I began working with the enemy AI and towers. Researching how other people implemented enemy movement in their

tower defense implementations, I noticed that most if not all relied on an implementation of the A* search algorithm. Being a versatile algorithm as it is, I discovered a different means of implementing a basic not-so-costly AI system: waypoints. The concept of waypoints is simple: given an ordered list of points, the enemy can follow a predetermined path by advancing to the first waypoint, then the next, and so forth. In Unity, that was easily done by placing empty GameObjects into the scene and making an array of the waypoints in the order to be followed. This array, managed by the EnemySpawner object, was passed on to the WaypointTraveler component of each Enemy, thus allowing the enemy to travel in the predetermined path.

I found the tower logic to be relatively easy to implement, as tracking enemy movement and locking on to nearby enemies involved simple applications of Physics colliders. However, there was an issue involving collider triggers when a bullet collided with an enemy. In Unity, colliders can either act as a trigger (i.e. collisions are only reported) or as an actual physical collision that results in two objects interacting with one another. In this case, I want the triggers to simply report the collision. Unfortunately, adding a Collider component to both the bullet and the enemy objects does not trigger a collision report. To do so, I had to add the Rigidbody component, the component that allows an object to respond to physics, to the enemy, thus triggering the appropriate function of reducing the targeted enemy's health.

As I worked on more and more of the gameplay, I began to notice that script execution management was becoming more and more difficult. Every activated script on every active GameObject is constantly being executed (assuming they have an Update() function), leading to terrible backend logic when dealing with scripts that shouldn't be active when another one was. To remedy this problem, I introduced a static global GameManager script that also implemented the Observer design paradigm. Using the Observer paradigm allowed me to introduce the states of the game (i.e. main menu state, wave start state, etc.) into the backend code. Whenever a script switched the game state, it would awaken all the other scripts that were listening for that specific game state. This allowed for much more flexibility when dealing with game state transitions, as only the scripts for a certain game state will only activate when their corresponding state was triggered.

Creating the GameManager class and introducing the Observer pattern also simplified the effort needed to set up the GUI. In Unity, there are two ways of creating a GUI: the first way is to utilize GUIText and GUITextures, both of which can be manipulated in the scene, and GUI scripting, where GUI elements are created and managed in the code. I soon found out that Unity GUI controls do not have native keyboard interaction; that is, it is nearly impossible to implement a keyboard-driven menu using basic GUI controls. Unity's GUI controls are, by default, reliant on mouse input to generate any response, so in order to implement keyboard interaction, I had to utilize Unity's Input class. Through a combination of the Input class and Unity's SelectionGrid GUI control, I was able to create a keyboard-driven menu system that used only two keys, a mover to cycle through menu options, and a chooser to select the chosen menu option. This way, I could implement the majority of all aspects of the game that required user

interaction while ensuring a uniform means of access, so users wouldn't get confused as to the controls at any point in the game.

Finally, I went forth and implemented a rudimentary main menu that would serve as the transition point between the game loading and actually playing the game. As I created the main menu in a different scene from the actual game, I had to tweak my build settings so that the main menu will start first. To load up the actual game, I used Unity's Application.LoadLevel function. However, the catch is for Application.LoadLevel to work, I had to add all my scenes into the build settings (File > Build Settings), which then allows me to load any scene at any point in my code.

In the making of this game, I tried to be as thorough as I possibly could, especially on user interactivity. While there are still some issues to be covered, such as being able to back out when placing towers and more flexibility with different types of input, I believe that I have created a stable game that is easily extensible and that provides a good foundation for anyone who would like to further the development of a switch-accessible tower defense game.