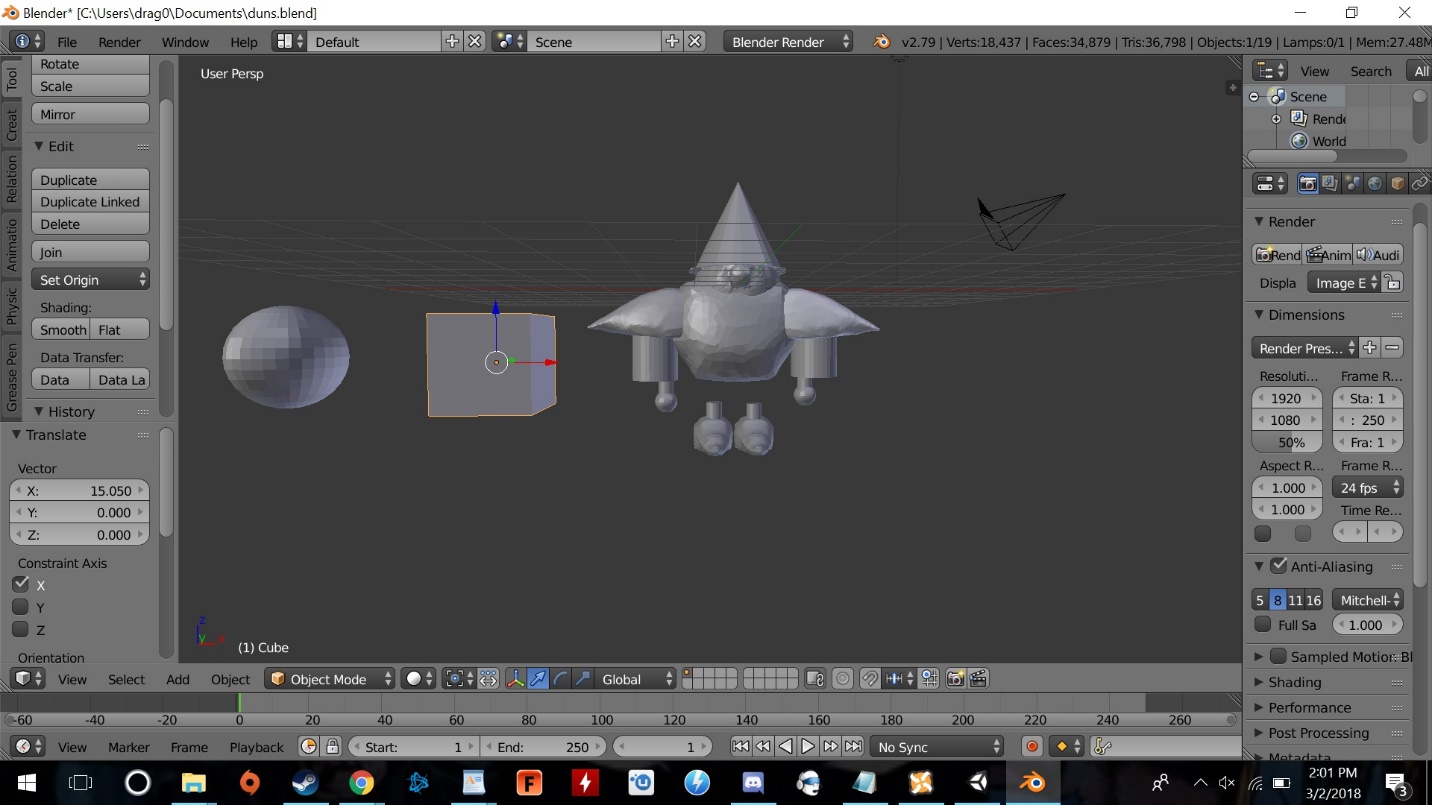
Chapter 4

Implementation

This chapter concentrates on the implementation of many of the game's mechanics, such as general character design, various animation types and functions for each character archetype. An explanation of the incorporated character targeting system will be discussed, as well as the collision and hit-detection system, NavMesh integration, User Interface design, and mini-map creation.

Character Design

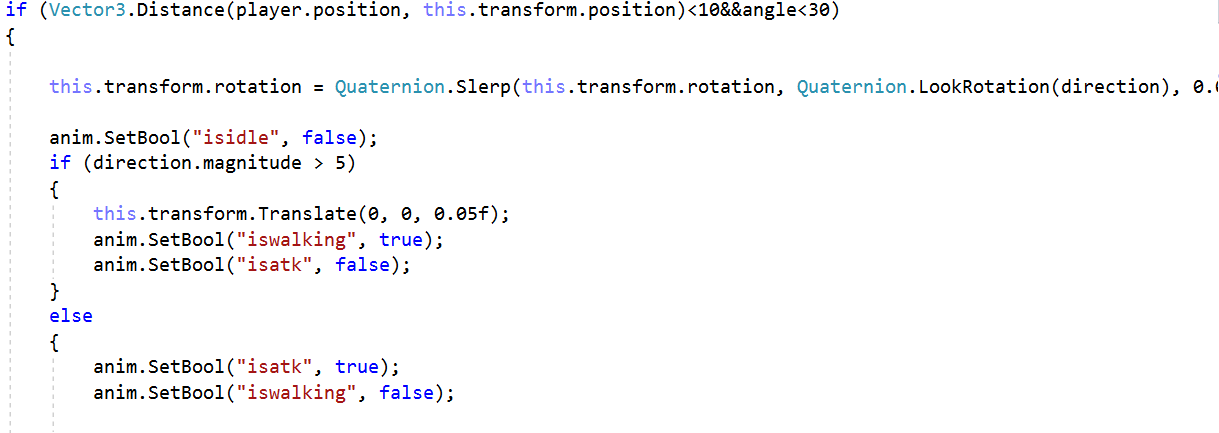
For character design, the software application Blender was used to create each character model. Each model started out as a compilation of basic shapes such as spheres, cones, and squares and were fused together to create complete models. In Blender, sculpt mode was the main mode that was used to mold the basic shapes into different forms using simple tools such as flatten and indent. For the molding of more complex objects such as the main character’s shoulder pads, the dynamic vertices tool was used, allowing the desired objects to be further detailed by adding and removing additional vertices as needed to give the additional shaping.



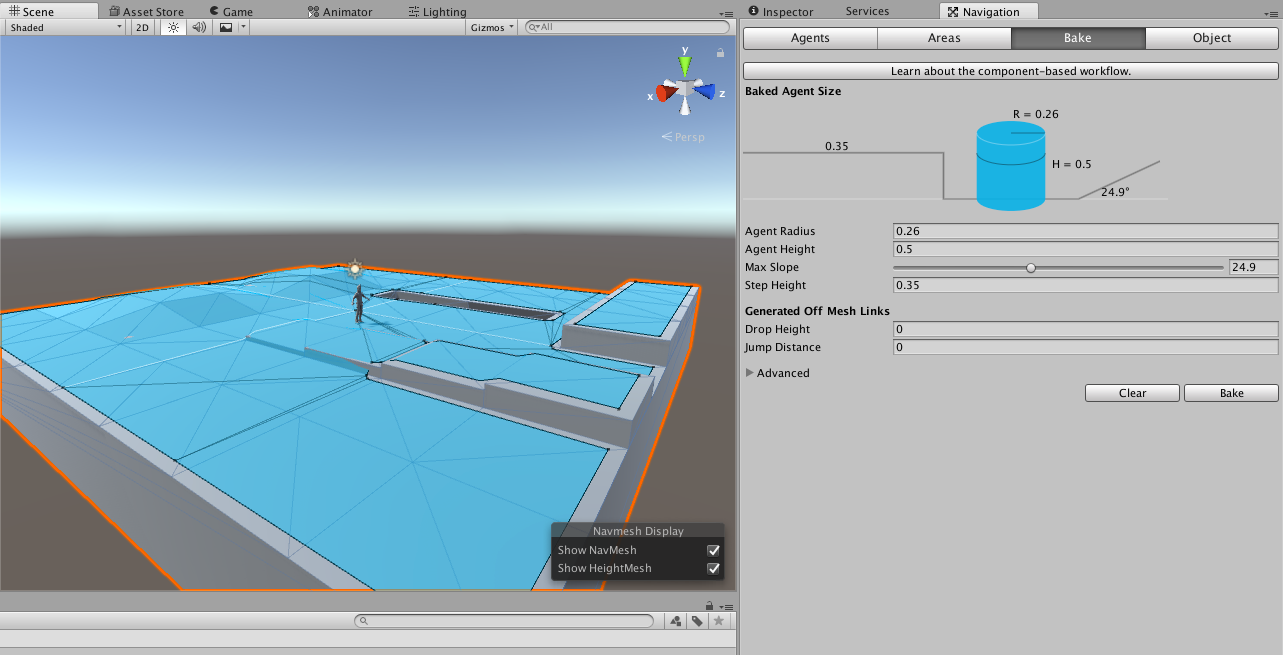
Character/A.I. Movement

Animating characters is done through assigning animations to the character and then going into the animation controller and connecting the animations. Named transitions, these connections are what allows animations to switch into one another. For example, if an enemy that walks towards the player and then attacks him, a transition that goes from the walk animation to the attack animation would occur. Each character aside from player controlled character has 3 animations; walk, attack, and idle, and each one can transition to each of the others.

Each character type has their own movement archetype. The player-controlled character, Too, also has commands that allows it to interact with the knight using one of the get-key-down prompts. Right clicking toggles a spotlight that is a child of the light. Toggling the light initiates the trust code in the knight. This in turn generates a random variable and if it is less than the trust variable, the knight will move to where the light is shining. For example, if an item or object is near the location of the light and the knight sees it, he will then attempt to interact with it without activating the random variable chance. If the object is beneficial, it will increase the chances he will come to the light and interact with those objects in the future. However, if the object has negative repercussions such as opening a trap chest, his chances of coming to the light and opening any chest will lower.

The main character, Light Knight, has his programming limited in aspects that only makes him active when objects are within his cone of vision, with the exceptions being the player-controlled character, Too, and the main character’s goal. Enemy character types will also have a cone of vision, and will act when the main character enters, giving chase and attacking. The recognition of the objects within the cone is due to the implementation of tags and targets, which cues the codes to run. Once the tag or target variable is declared in the code you set its "value" in Unity by dragging whatever you want to be the target into the variable space in the script component window. For example, with “targets”, public object variables are set within the code, which lets enemies recognize and pursue the knight, and the same can be done to let the main character know where the goal is. This target variable is what lets enemies recognize and follow the knight and what lets the knight know where its goal is. Once the target variable is declared in the code you set its "value" in unity itself by dragging whatever you want to be the target into the variable space in the script component window. For multiple objects “tag” must be implemented to differentiate between them. For instance, for all non-trapped chests are tagged "goodchest", the A.I. is programmed to generate a random number that decides whether or not to approach them if they are in the cone of vision and tagged with the tag that its programmed to operate by. As a result, the “goodchest” code only plays when an object tagged “goodchest” enters the knight’s cone of vision. 

NavMesh

Navigation meshes, or NavMesh, is a process that involves collecting terrain and other environmental meshing that have been marked as navigation static, then generates approximate walkable surfaces within the stage to more seamlessly move through. This essentially helps the A.I. character navigate through the more complex collision detection checks with objects that stand out in particular parts of the environment. The process starts by selecting a navigation area with walkable surfaces that would need altering. The next step is to turn navigation static on to include any environmental objects in the baking process. Adjustments can also be made to the bake settings to fit your character size more fittingly such as agent radius, Agent height, Max slope, and Step height. Agent Radius allows adjustments on how close the character’s center can get to a wall. Agent height defines how low the spaces are that the character can reach. Max slope lets you alter how steep ramps and hills can be defined that the character can walk up. Step height lets you select the max height of objects and obstructions in the environment that can be stepped on by the character. Once all of the adjustments have been made, the bake button is clicked to build the NavMesh, resulting in the environment being covered in a blue overlay that represents the areas that the character is allowed to traverse.

Collision/Hit Detection

The knight interacts with these objects through the use of colliders. Every object has a collider and the knight has his own collider. When the knight’s variable determines whether he wants to activate the object is true and his collider hits the objects collider the object is activated. These colliders determine hitboxes and hurtboxes. The enemies have small colliders on their weaponry and whenever that collider enters the knights collider the knight loses health which is a variable within the game rules, which is how traps damage the knight and how he changes between scenes as well.

User Interface/title screen

Within unity, there a user interface controller that can be applied which keeps all UI elements in front of the camera. An image or text can also be added to the UI, which will always be in front of the camera. This is how the health bar, which is a number represented as a bad graph, works. The title screen works this way and has buttons which, when a button is clicked, activates the set code. This function can be anything from starting the game to switching to the options screen which operates as its own scene like the title screen.

Minimap

The game’s mini-map was integrated by essentially creating a new camera and placing it high atop the main character. This was done by creating a render texture and placing the texture inside camera target texture to show the location of the map.



A raw image was created and set with a width of 300 and height of 250, and the mini-map was anchored to the bottom-left of the game screen.