Part 1: 3D Character Controller

Debrief – 08/13/2021

# Overview

I was successful in creating the 3D character controller although testing and iteration led me to some different techniques than what I decided upon in the debrief. For example, I decided that this character controller did not need to obey a NavMesh, and that free movement was more desirable for the style of game I am creating. The result of the last two weeks of work is a fully operational character controller complete with animations that respond to the character’s movement.

# Goals Recap

* Create movement for a 3D character controller that obeys a NavMesh
  + I decided that a NavMesh would be too restrictive on player movement. I will be using a NavMesh for AI later in the project. Here, I used Unity’s built-in Character Controller class for navigation, and controlled its movements using its Move method. I split the movement up into several components, to better follow the Single Responsibility Principle:
    - MoveComponent – Handles the planar movement on the X and Z axes based on player input.
    - Jump Component – Handles the vertical movement on the Y axis based on player input.
    - Gravity Component – Adds negative velocity in the Y direction if the player is not grounded.
    - Crouch Component – Handles the transition to and from the crouch state.

I quickly noticed that with all the possible motions the character already has, limiting the player’s abilities may require a finite state machine. A lot of videos I researched regarding the topic stuck to a simple “if-then” structure to track state. I know this is a dangerous route to follow, especially because I have taken the modular approach of splitting nearly every aspect of movement into its own component script. Instead, I am trying out a new idea – using a scriptable object referenced by nearly all movement components to track state in a global fashion. This way, each component can check within the global state what it is and is not allowed to do before executing its task. It may get messy later, but for this first step of the journey it seems to be an eloquent solution without diving deep into a finite state machine.

* Use the Command Pattern to eliminate player controller’s dependence on player object
  + This worked out just as I had hoped. I created an command infrastructure using a Command Stream ScriptableObject and Commands built as native C# classes. This implementation was based off Robert Nystrom’s explanation of the command pattern on his website, <http://gameprogrammingpatterns.com/>. The player controller accepts input from Unity’s New Input System and translates it to a command, which it then enqueues into a command stream. The player character, which also references the command stream, then dequeues the command and executes it. Since each of the classes are referencing a scriptable object held in the project files, they are not dependent on each other in scene. There are two command streams thus far – one for movement and one for camera control. This is because the player will often be receiving commands for both camera movement and character movement at the same time, and they should be executed simultaneously. I am slightly worried that with this pattern the amount of command streams I will need might explode, but I will be sure to monitor that and devise a better solution should the need arise.
* Use Cinemachine to control the first-person camera
  + The first-person camera was one of the trickier parts of this implementation because it required some intermediate knowledge of Cinemachine’s extension class, and I had only ever used the built-in portions of Cinemachine prior to this project. Thankfully, a YouTube instructor named samyam made an excellent video on how to create a first-person camera. The idea is to inject the mouse delta into a virtual camera’s Post Pipeline Stage Callback, clamp the angles so that the player can do a full 360 when looking up, and move the camera by the resulting angle within its aiming stage. The logic for camera movement is relatively simple but knowing where to put it required some more research. Thank you samyam!
* Use the New Unity Input System to allow seamless transition between mouse & keyboard and gamepad input
  + This was pretty simple on the surface; however the implementation was a little tricky since we are trying to follow the control scheme of *The Elder Scrolls: Oblivion*. In this game, if the player is using a mouse and keyboard, the player must toggle between running and walking using the caps lock. However, if the player is using a gamepad, whether the character runs or walks is based on how far the player pushes the left stick. So on one side movement is discrete, the other continuous. I accounted for this by doing the following:
    - Adding a bool parameter to the MoveCommand class that tracked which input the move command was coming from, keyboard or gamepad.
    - Creating two methods inside the PlayerController class for keyboard movement input and gamepad movement input respectively. I hooked up gamepad movement and keyboard movement separately in the InputActions object from the New Input System.
    - Before enqueuing the command in the command stream, setting the bool parameter in the command to true if it was gamepad input and false if it was keyboard.
    - Treating the move command differently in the MoveComponent class based on which type of input was received.

This worked well and I was able to recreate the behavior from *Oblivion*.

* Allow for rebinding of movement controls  
  + I thought that using the command pattern would make this straight-forward, but I found this was really challenging when using the New Unity Input System. Unity does provide a few classes for rebinding controls but it seems like a prototype and likely to change in future iterations. I think I want to avoid using it for now, but its nice to know that later in the project there may be a pretty simple way of rebinding controls. For now, it has been left out.

# Bugs/Future Enhancements

* The first-person camera can clip when running into a wall. The near-clip plane is pretty much set to as small as it can be, so I will have to do more research on how to fix this.
* There are a bunch of wonderful animations that can’t be seen in first-person view. I took a shot at allowing the player to toggle between first and third-person views, but ultimately fell short. The third-person perspective was not in the original scope of the project, so I decided to move on. It’s something I may like to include in the future.
* I can see that a lot of the parameters for the four movement components will be based on either player stats (strength, speed, etc.) and player settings (look sensitivity, FOV, etc.). Instead of setting these in each class, I will likely use a Scriptable Object for each component to reference to set those parameters instead of in the inspector as it is currently.
* I am also noticing that for NPC’s I will need to create and store the state data ScriptableObject and likely command streams. I don’t think this will force me to change my architecture later on, but its something to be weary of.

# Next Steps

I think from here I want to add more to the character controller by implementing some combat controls. This will be an excellent test of the global state-tracking ScriptableObject architecture I have implemented, and it is a logical next step to creating the character controller I need for this project.