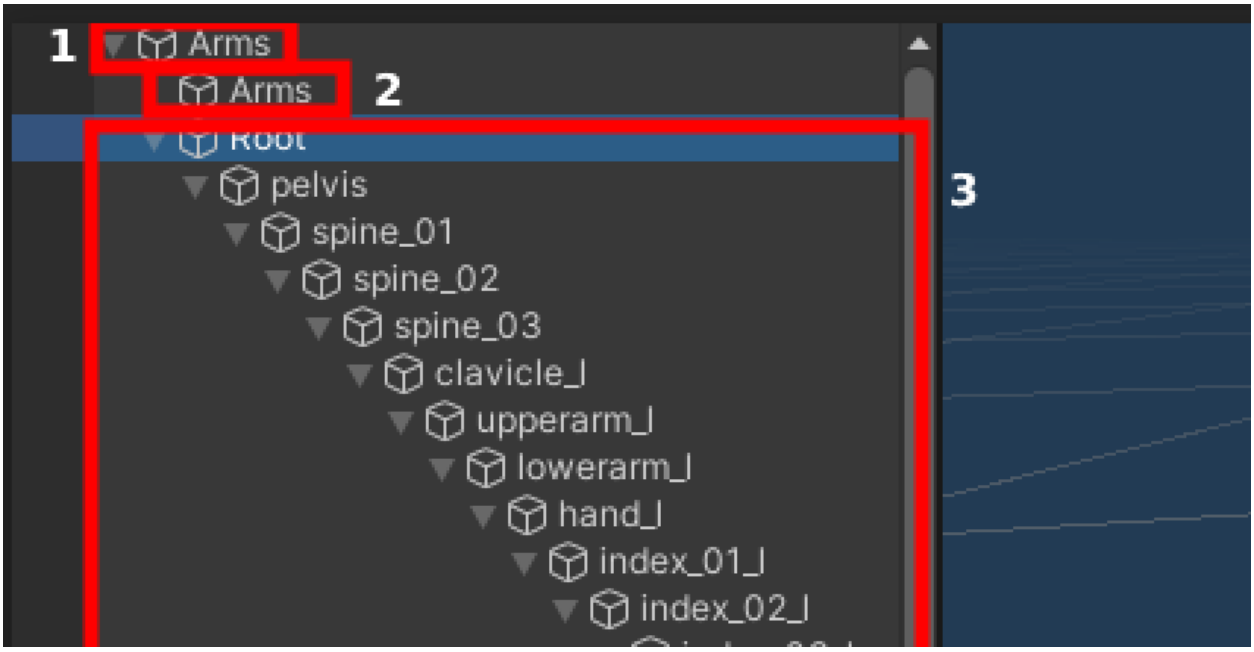
Avatar Saving System Documentation

This asset allows you to save avatars or load and apply them to a game object even when the object is currently animating. It Supports Generic, Humanoid and Legacy.

Requirements

The only requirement for this system is that each part of the avatar (clothes, arms, gloves, hat, etc.) they share the same hierarchy and bone names. Unity automatically renames bones that have the same name as any other object in the imported model, so make sure that no two objects in the model share the same name. This includes meshes. The High Level overview of how the hierarchy looks like is this:

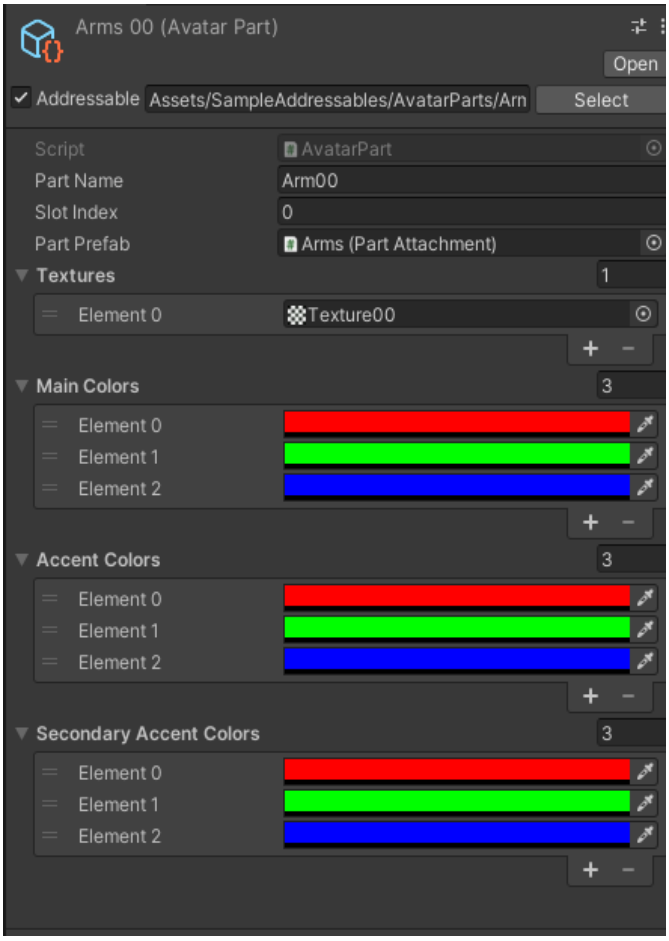


1. The parent is an empty game object that only contains 2 components: Transform and PartAttachment.cs
2. The first child of the parent is the SkinnedMeshComponent that is responsible for the rendering of the part.
3. The second child of the parent is the root bone of all bones in the part.

As long as this format and hierarchy is followed, you can customize basically anything, not just humans! This includes in game props, guns, buildings, etc. as long as you follow this hierarchy and documentation.

AvatarPart Scriptable Object

After setting up the Part Prefab, you can create an AvatarPart Scriptable Object by right clicking the project window > Create > Avatar Part



Part Name = Name of the part of avatar.

Slot Index = Index of the Slot on where to put the part

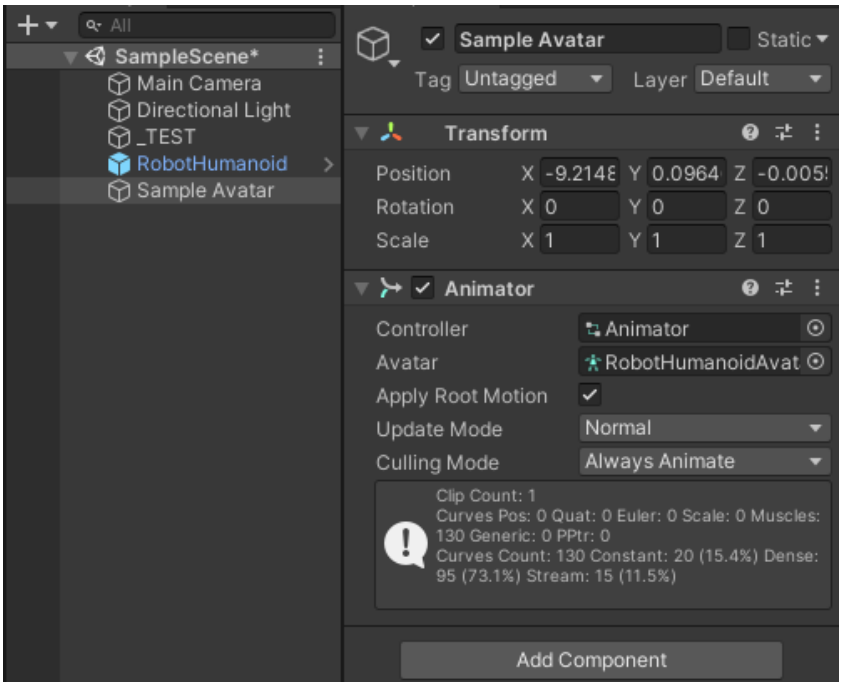
Part Prefab = Prefab of the part that will be attached to the selected slot

Textures = List of all valid textures that the Part can change into

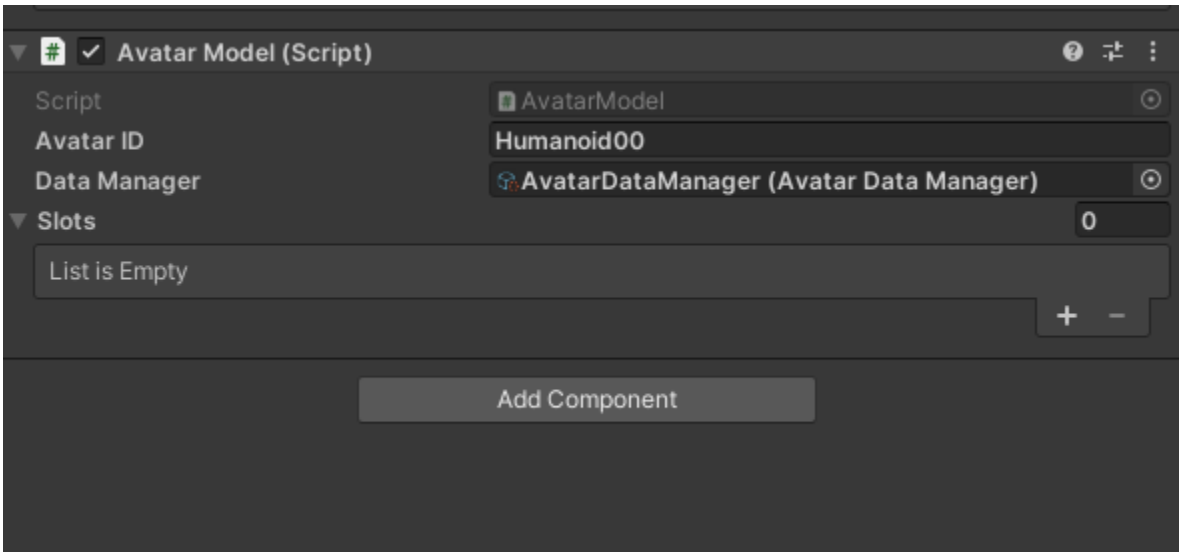
Main Colors, Accent Colors,& Secondary Accent Colors = List of Colors the part can change into (NOTE: There must be a parameter in your material named "MainColor", "AccentColor", and "SecondaryAccentColor" so that each colors can be applied, If there is no parameter, the color will be ignored but still be saved in the json

AvatarModel

AvatarModel is responsible for applying changes to the avatar by attaching and reconfiguring bones to the model to better suit the animation. You only need to attach the script to an empty game object with only two components: transform and Animator (or Animation if you are using Legacy). Note that this game object doesn't need to have a child object since the Avatar Model will be the one assembling the parts to build the final model.

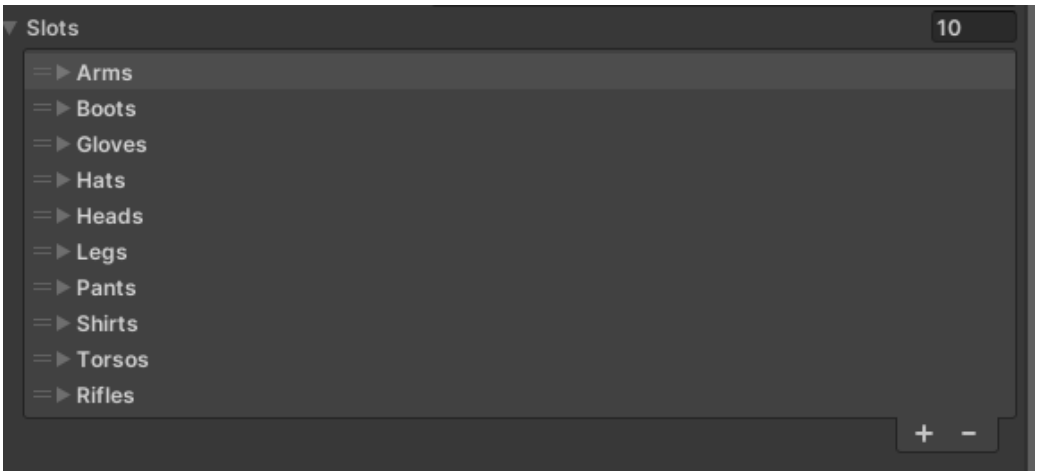


AvatarModel Fields



Avatar ID = The ID of the Avatar that will be saved and Loaded From the json. (Since you can customize anything, the avatar ID is useful for distinguishing what avatar you are working on, eg. Gun_SMG00, HumanoidFemale00, Hospital00, etc)
Data Manager = The data manager handles loading and saving to and from json file.

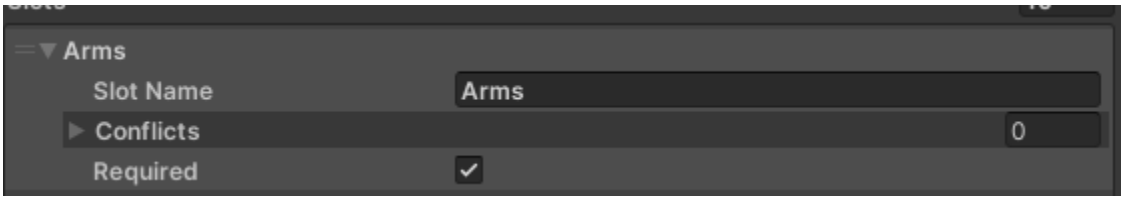
Slots = Slots is the list of all the slots in a certain game object. You need to initialize slots by adding a new item to the list, Naming the slot, and remembering the index.



The order of the slots is important since the slot index will be dependent on the order of the slots (if you put in slot index 4 in the Avatar Part SO, it will be inserted on the 4th index. (In this case, the Head Slot)

The reason why it is an index-based system as opposed to enum-named slots because the object that will be customized varies. Its not only limited to humanoid creatures.

AvatarPartSlot



Slot Name = Name of the slot to be inserted to. NOTE: the slot name must be the same name as the folder name where you will store all the parts for that slot. In this case, the folder where you will be storing all arms attachment will be Named "Arms". Where you store that arms folder is anywhere you want as long as the folder name matches the Slot name.

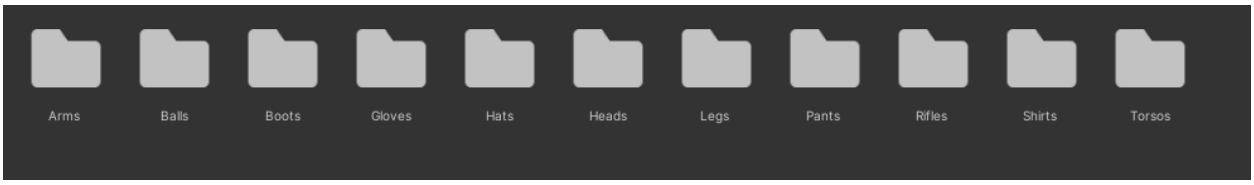
Conflicts = List of Slot Index the will have conflict to (For example: if the Rifle Slot has the numbers 2 (glove) and 7(shirts), this means that if you equip a rifle, glove and shirts will conflict, thus removing the attachment on slots 2 and 7. If there is no conflict, you can leave it empty.

Required = Boolean indicating whether or not this slot is required. Required means that the slot must never be empty at all times. If false, the slot can be empty at any point.

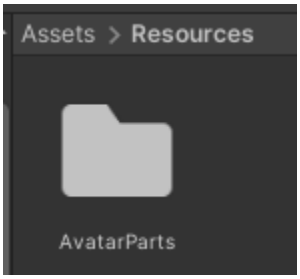
Storing Parts

As mentioned earlier, folder name of parts will be dependent on Slot name found on each AvartPart Scriptable Object.

Your folder naming will look like this if we base on slot names



After that, you need to store all these folders in wherever parent folder you want in the resources folder. I stored them in Resources/AvatarParts



When that's done, Just change the `AVATAR_PARTS_PATH` in the AvatarModel Script to match the parent folder.

The whole path to a specific part will look like this:
AVATAR_PARTS_PATH + AvatarPartSlot.SlotName + "/" + PartSlotData.PartID

AvatarDataManager

The manager that is responsible for storing the list of all avatars you have in a single string – AvatarData Dictionary. You can access a certain AvatarData by referencing `AvatarDatas[AvatarID]` . The manager is also responsible for exporting the dictionary into different json files. One for each key. And also importing from json.

Retargeting

If you have trouble with retargeting animations in cases where the animation bones and the mesh bones are different, instead of creating an empty Gameobject with AvatarModel attached and with no rig, you can provide an existing rig from the animation model before any attachments attach to it.

TestScene

You can test out the system by going to the test scene.

There will be a gameobject in the test scene named _TEST. You can plug in any part you want to the correct slot and plug in any texture and colors in the slot to test it out. If you run the game, it will reflect the changes
To Save the current customization in the test scene – press Q
to load in the test scene – Press E

You can also change the parts during runtime in the test scene. All you need to do is to change the inspector values during runtime and then press R to update the changes.