MASTER CHARACTER CREATOR

If you run into any issues or have any suggestions or requests, please feel free to contact me at : **projectx1901@gmail.com**

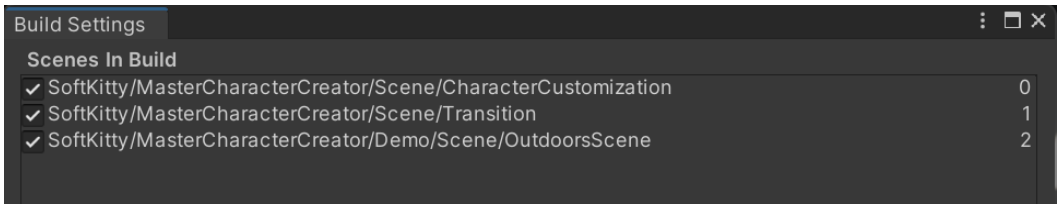
GETTING STARTED

Master Character Creator is designed to integrate seamlessly into your project without the need for scripting. However, it also offers a comprehensive set of API calls for advanced integration. Built with a modular concept, every customization option can be tailored by developers to suit their specific needs.

1. Before we get started, please take a look at the **demo scene** located at :

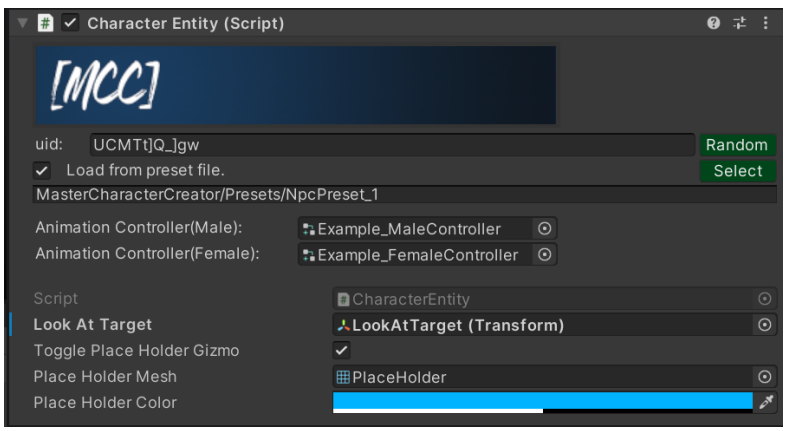
   "*SoftKitty/MasterCharacterCreator/Demo/Scene/OutdoorsScene.unity*"

   To run the demo, make sure to add the following scene to your build settings:



2. In the **demo scene**, you will find a game object called "**CharacterData**," which is the core prefab of the system. You can adjust the settings for the system on this prefab.

3. In the **demo scene**, locate the game object called "**NPC**." It contains a CharacterEntity component, which you'll add to every NPC and the Player in your game. You can also have your character control script inherit from this component. Let's walk through the settings one by one:



UID

The appearance data for all characters is stored in a Dictionary using the UID as the key in the character manager. This allows you to access or modify appearance data without instantiating the character prefab. **Make sure all your characters have different UID .**

LOAD FROM PRESET FILE

The appearance data can be saved as a small **byte file** (typically under 256 bytes). This **byte file** can be placed in your *resources* folder. Click the "**Select**" button to choose the **byte file** from your *resources*, and during runtime, the character will automatically load the data from this file to instantiate the appearance.

PRELOAD CHARACTER

By default, characters are loaded at runtime when the scene starts. However, click the "Preload Character" button allows you to load the character directly within the Unity Editor. This is useful for making in-depth modifications to the character's prefab, such as adding ragdoll components, damage detection colliders, visual effects (VFX), and more, directly in the scene view.

ANIMATION CONTROLLER (MALE/FEMALE)

Assign your desired **animation controller** for the character here.

LOOK AT TARGET

If you want the character to look at a specific transform, you can assign that transform here.

TOGGLE PLACE HOLDER GIZMO

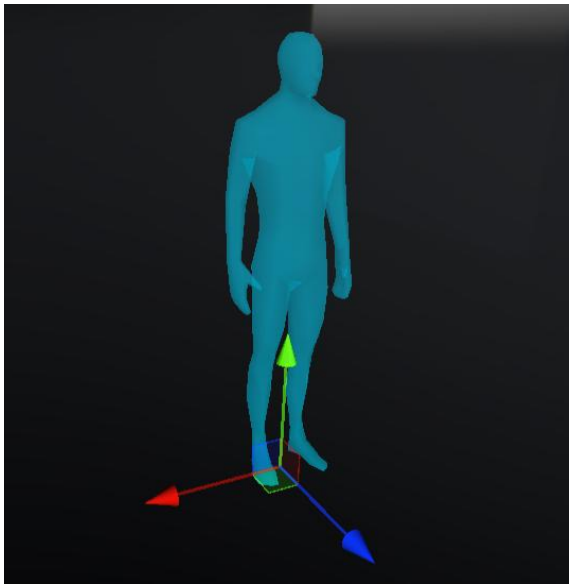
Since the character's appearance will only load during runtime, enabling this option allows you to visualize the character during level design by displaying a placeholder mesh.

PLACE HOLDER MESH

Select the mesh to use as the placeholder.

PLACE HOLDER COLOR

Choose a color for the placeholder mesh.



4. Run the demo scene, and you will notice three buttons on the top left:

CREATE NEW CHARACTER

This calls <CharacterEntity>().CreateCharacter(), which switches to the character creation interface. When a player creates a new character, you can call this function with the character entity you want to create the appearance for.

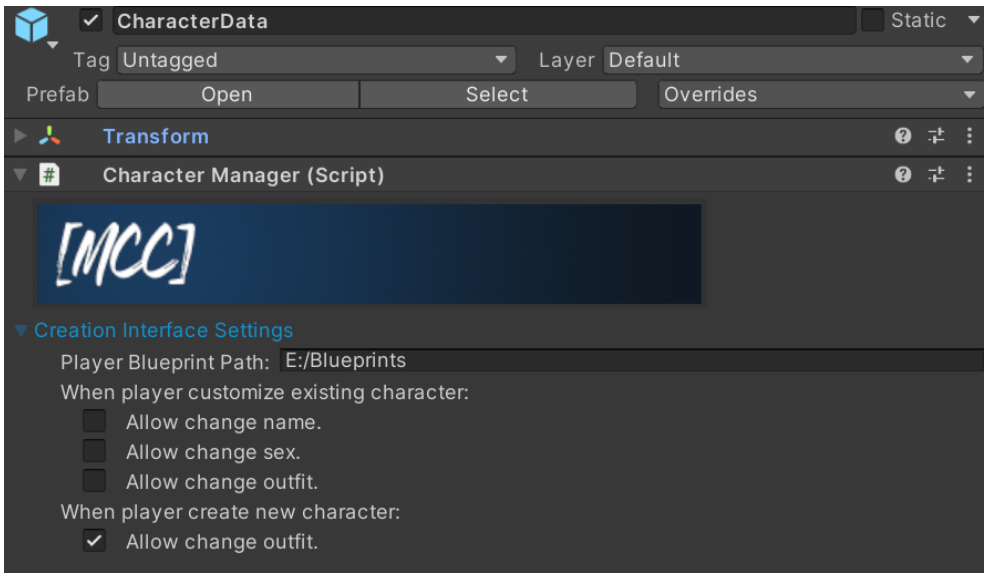
CUSTOMIZE CHARACTER

This calls <CharacterEntity>().CustomizeCharacter(), switching to the character customization interface. When a player wants to customize a character, you can call this function with the character entity you want to customize.

CREATE NEW CHARACTER (DEVELOPER)

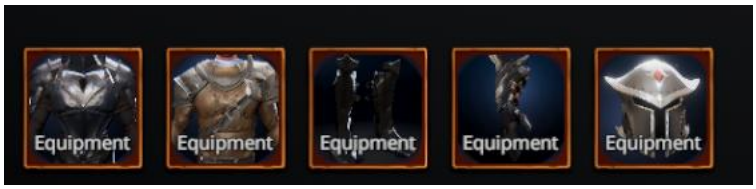
This calls <CharacterEntity>().CreateCharacterByDeveloper(), which switches to the character creation interface with all outfits unlocked. This is a useful tool for creating NPCs or character presets.

5. To modify the settings of the interfaces mentioned above, locate the core prefab "**CharacterData**" and look under the "**Creation Interface Settings**" tab.

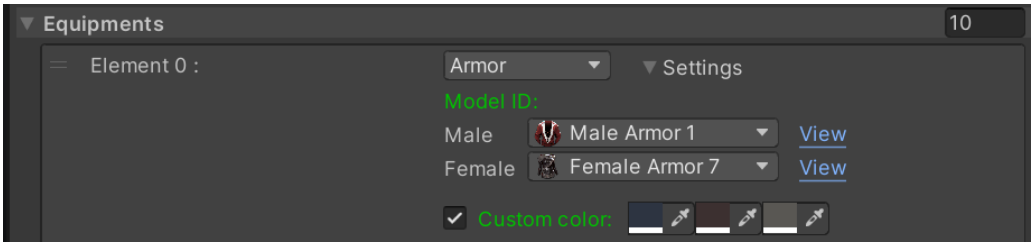


Please note, when a player uses the **save/load** function in the **character customization** UI, the data is saved as a **PNG** image of the character's photo, with the appearance data hidden within the **PNG** file. This file can only be stored in the path defined by the "**Player Blueprint Path**" setting. However, when in the "CreateCharacterByDeveloper" mode, the appearance data will be saved as a **byte file** instead of a **PNG**.

6. On the bottom left of the screen, you'll see some test buttons that demonstrate how equipment in your game can be linked to character appearance.



Open the DemoScript.cs script, and you'll notice it binds the equipment using a class called "EquipmentAppearance." You can easily add this class to your equipment item data. When a player equips an item, simply call <CharacterEntity>().Equip() with the EquipmentAppearance data for that item, and the character's appearance will update accordingly.



7. With the knowledge from the previous steps, you can already use this system to create NPCs and allow players to customize their own characters. If you want to add more models and textures into the system, please refer to the following documentation:

HowToCreateHeadAccessories.pdf

HowToCreateCustomizationTextures.pdf

HowToCreateOutfits.pdf

For more advanced usage of the system, continue reading the following sections of this document.

IMPLEMENTING MCC IN YOUR GAME: EXAMPLE SCENARIOS

First let's go through how to setting Up Your Player and NPC Characters with MCC.

- Duplicate the NPC and Player prefab located in: *Assets/SoftKitty/MasterCharacterCreator/Prefabs/* for your player and NPC.
- You can build your own demo or download the pre-made demo using the link below. This allows for a quicker and more streamlined way to create and save appearance files for your player and NPC characters:
  https://drive.google.com/file/d/19nWtq7E4Ghefq9AulRfroO_oVfYwn5X9/view?usp=drive_link

SETTING UP PLAYER CHARACTER:

1. **CREATE INITIAL APPEARANCE**:

   Pre-designed appearance: Use the Demo scene to design your player's initial appearance. Save this data in either the Resources folder or a designated folder within your game's installation directory (where player save files are stored). Then use the load methods in #5 below to load this appearance data to the player character.

   Default appearance: You can leave the player as default appearance , to do that, use the following code to Initialize your player to the default appearance:

   `GetComponent<CharacterEntity>(). Initialize (Sex _sex)`

   Let player create their character: If you want to begin with character creation interface to let player to create their character before playing, simply call:

   `GetComponent<CharacterEntity>(). CreateCharacter ()`

2. **ADD PLAYER CONTROL SCRIPT**: Attach your custom player control script to the root GameObject of the prefab.
3. **ASSIGN MECANIM CONTROLLERS**: In the `CharacterEntity` component, assign a **Mecanim controller** to both the Male and Female slots to handle animations.
4. **UNCHECK 'LOAD FROM PRESET FILE'**: In the `CharacterEntity` component, uncheck the "**Load from preset file**" option to allow loading from save files.
5. **LOAD APPEARANCE IN SCRIPT**: In your player control script, use `GetComponent<CharacterEntity>().LoadFromResourceFile()` to load the player's appearance from the Resources folder. Alternatively, use `LoadFromByteFileFromDisk` or `LoadFromPngFileFromDisk` to load appearance data from a custom save location within your game's installation path. For example:

   `Application.dataPath+"/../GameSave/PlayerAppearance.bytes"`

   If you want to loads the appearance data with your own save system, you can get the bytes array of player's appearance data with:

   `GetComponent<CharacterEntity>(). LoadFromBytes (byte[] _bytes)`

6. **SAVE APPEARANCE IN SCRIPT**: In your player control script, use `GetComponent<CharacterEntity>().SaveByteFileToDisk ()` or `GetComponent<CharacterEntity>(). SavePngFileToDisk ()`

   to save the player's appearance to custom save location within your game's installation path. For example:

   `Application.dataPath+"/../GameSave/PlayerAppearance.bytes"`

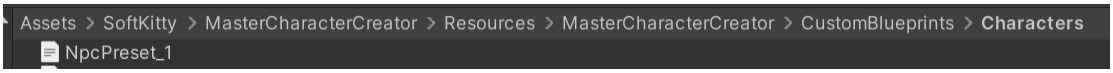
   If you want to save the appearance data with your own save system, you can get the bytes array of player's appearance data with:

   `GetComponent<CharacterEntity>(). GetSaveBytes()`

SETTING UP NPC CHARACTER:

1. **CREATE INITIAL APPEARANCE**: Use the Demo scene to design your NPC's initial appearance. Save this data in the Resources folder or a designated folder within your game's installation directory (where NPC save files are stored).
2. **ADD NPC CONTROL SCRIPT**: Attach your custom NPC control script to the root GameObject of the prefab.
3. **ASSIGN MECANIM CONTROLLERS**: In the CharacterEntity component, assign a **Mecanim controller** to both the Male and Female slots for animations.
4. **ENABLE 'LOAD FROM PRESET FILE'**: Check the "**Load from preset file**" option in the CharacterEntity component, and specify the path to the NPC's appearance file in the Resources folder. For example:

   MasterCharacterCreator/CustomBlueprints/Characters/NpcPreset_1



IN-GAME CUSTOMIZATION OF PLAYER/NPC APPEARANCE:

- To allow players to create their character's appearance with the character customization interface when they first start the game, simply call the following methods from the CharacterEntity component: CharacterEntity().CreateCharacter();
- To enable players to customize the appearance of their own character or any NPC later in the game, use the following method from the CharacterEntity component of the player prefab: CharacterEntity().CustomizeCharacter();
- To save the appearance of the player or NPC, use the following method from the CharacterEntity component of the player or NPC prefab:

  CharacterEntity().SaveByteFileToDisk(string _absolutePath, BlurPrintType _filter)

  CharacterEntity().SavePngFileToDisk(string _absolutePath, Texture2D _photo, BlurPrintType _filter)

  For example:

  CharacterEntity().SaveByteFileToDisk("E:/player.bytes", BlurPrintType.Character);

  You may noticed there is a parameter called "BlurPrintType", it is a **enum** indicate which part of data will be saved:

  BlurPrintType .BodyShape – Only the body shape will be saved, the face/skin color/tattoo/hair/outfits won't be saved.

  BlurPrintType .Character – Only the character itself will be saved, the outfits won't be saved.

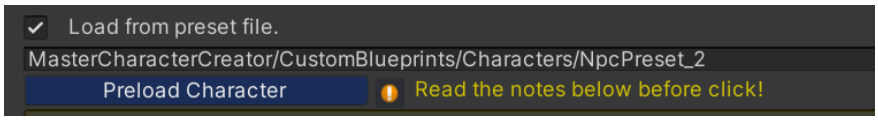
  BlurPrintType .Outfits – Only the outfits will be saved.

  BlurPrintType .AllAppearance – Everything will be saved.

INTEGRATING WITH OTHER CHARACTER CONTROLLERS

The characters in the Master Character Creator (MCC) system function like standard Unity humanoid characters, making integration straightforward. To use MCC characters with your custom character controller, follow these steps:

1. Add Your Character Control Script: Attach your character control script to the player prefab at the same GameObject level as the CharacterEntity component.
2. Assign the **Mecanim** Controller: In the CharacterEntity component, assign the appropriate **Mecanim** controller to both the male and female slots. This will allow the character to animate correctly based on your controller.
3. If the character control system you're using requires reference of the character bone transform or SkinnedMeshRenderer, click on "**Preload Character**" button to load the character in the Editor, then you can reference the bone and mesh GameObjects as needed.

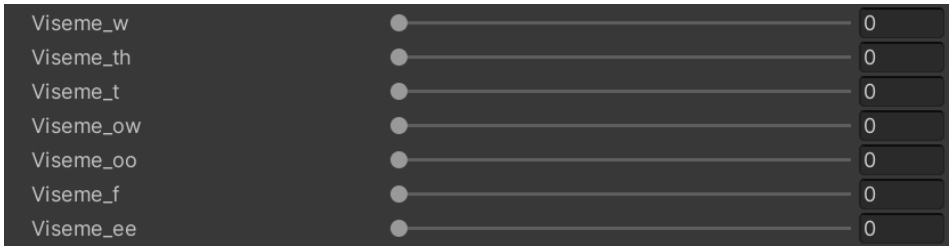


By following these steps, you can seamlessly integrate MCC characters with other character control scripts or systems in your game.

IMPLEMENTING LIP SYNC SYSTEM

MCC is compatible with most **LipSync** system using **BlendShapes**, such as **SALSA LipSync Suite**. Follow the below steps to implement **LipSync** system:

1. Find **CharacterMale** and **CharacterFemale** prefabs in: **Assets\SoftKitty\MasterCharacterCreator\Resources\MasterCharacterCreator\Player**
2. Uncheck "**Manage Blend Shapes**" setting in the CharacterBoneControl component.
3. Find "**Model_Head**" GameObject under each prefab, add your LipSync component to it.
4. Assign the follow **BlendShapes** of "**Model_Head**" as visemes of your **LipSync** system:

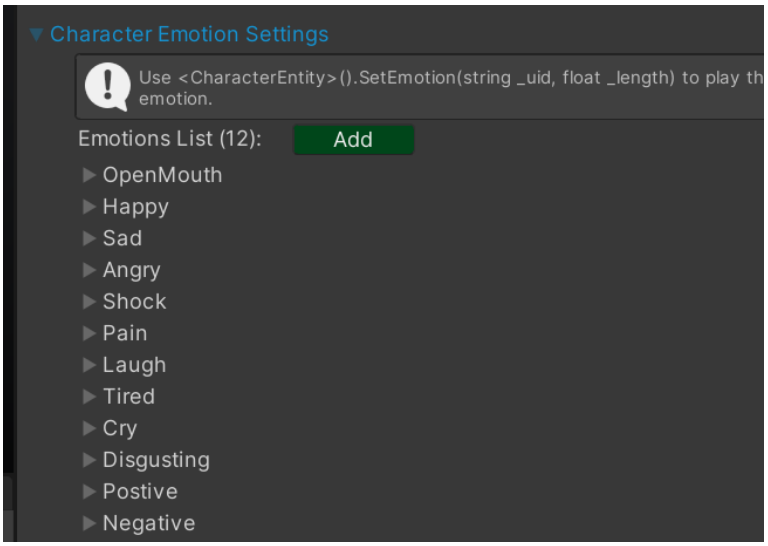


5. Find "**Model_Teeth**" GameObject, make sure include "**Teeth_Open**" **BlendShape** with your **LipSync** system as well.

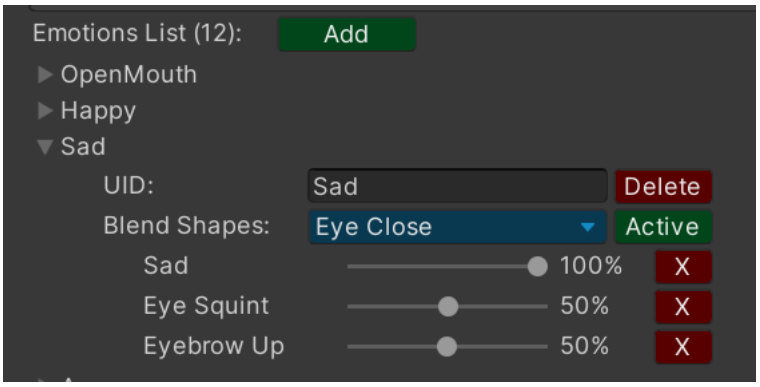
CUSTOMIZE EMOTIONS

Master Character Creator allows you to create and customize unique character emotions, which can then be triggered in your game using their unique identifier (UID).:

1. Locate the **CharacterData** prefab in your project and expand the **Character Emotion Settings** section in its Inspector.



2. Expand an emotion entry in the list. You will see a field labeled UID. This string is the unique key you'll use to play this specific emotion in your game.



3. To trigger an emotion in your scripts, call the function <CharacterEntity>().SetEmotion(string _uid, float _length), replacing _uid with the UID of the desired emotion and _length with the duration (in seconds) you want the emotion to play.
4. The blend shapes dropdown allows you to activate multiple facial blend shapes on your character's head simultaneously. Use the percentage sliders next to each blend shape to control the intensity and create nuanced emotional expressions.

MANAGING WEAPONS

MCC provides user-friendly tools and APIs to manage weapons efficiently. Follow the steps below to integrate your weapon models:

1. **Set Up the Weapon GameObject:**

   Create an empty GameObject and add the <WeaponController> component to it.

   Drag your weapon model as a child of this GameObject and adjust its position so the weapon's handle aligns with the pivot of the empty GameObject.

   Ensure the tip of the weapon points along the Z-axis of the empty GameObject.

2. **Weapon Settings in <WeaponController>:**

   Select Weapon Type: Specify how the weapon is held by selecting one of the options: **LeftHand**, **RightHand**, **TwoHanded** or **Custom#**.

   If the weapon is not designed to held by hands, for example, a cross bow equipped on the forearm, you can select the **Custom#** type.

   Set Root Bone for Holding:

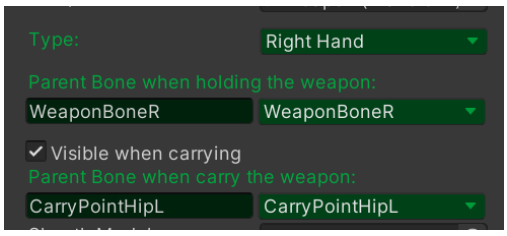
   Assign the bone used for holding the weapon. MCC provides special bones, **WeaponBoneL** and **WeaponBoneR**, which align with the center of the character's hand.

   Alternatively, you can assign any other bone by entering its name.

   Set Root Bone for Carrying:

   Assign the bone used for carrying the weapon. MCC provides pre-defined bones like **CarryPointXXX**, which are correctly positioned based on the character's body shape. These carry points are "soft-connected," meaning they dynamically swing with the character's movement.

   You can also assign any custom bone by entering its name.



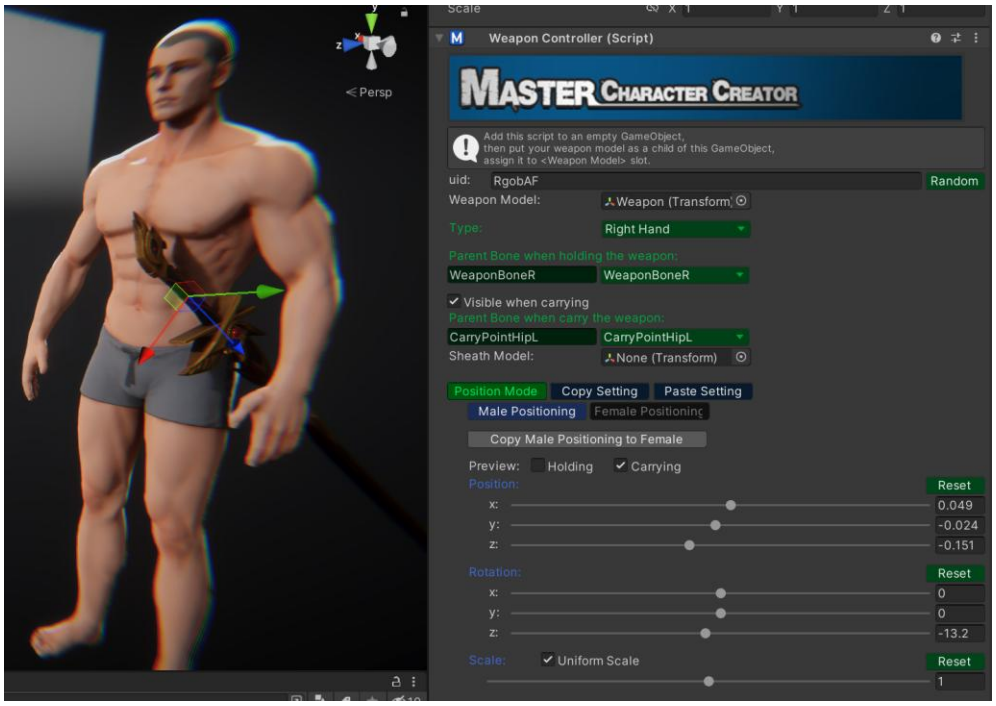
3. **Positioning the Weapon:**

   Click the "**Position Mode**" button in the <WeaponController> inspector to preview how the weapon looks with the character.

   Adjust the weapon's **position**, **rotation**, and **scale** as needed.

   You can switch between **Male** and **Female** characters and set separate positioning for each. For example, you can scale the weapon down for female characters.

Toggle between “**Holding**” and “**Carrying**” states to ensure both are correctly positioned.



4. **Copying Settings:**

   Use the “**Copy XXX Positioning to XXX**” button to copy the current positioning (**Holding** or **Carrying**) between **Male** and **Female** characters.

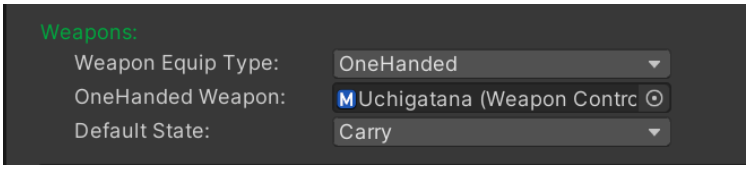
   Click “**Copy Setting**” to save the current settings to the clipboard. This allows you to paste the settings to similar weapons by clicking “**Paste Setting**” in another weapon’s <WeaponController>.

5. **Saving the Weapon:**

   Name your weapon GameObject and save it as a prefab.

6. **Equipping Weapons to Characters:**

   Option 1: Drag the weapon prefab into the “**Weapons**” slots in the <CharacterEntity> component.

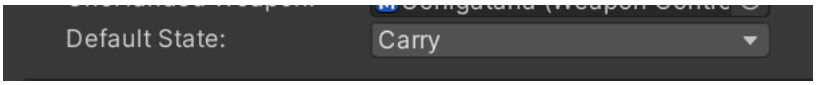


   Option 2: Equip weapons programmatically using the provided API calls. Refer to the <CharacterEntity> API section for details.

7. **Switching Weapon States:**

   Weapons can transition between three states: **Hold**, **Carry**, and **Hide**.

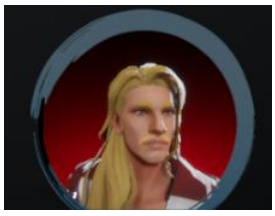
   You can set the state directly in the <CharacterEntity> inspector or change it via API calls. Refer to the <CharacterEntity> API section for details.



MISCELLANEOUS MODULES

- **Taking Character Photos:**

  Capture character portraits at runtime with the GetCharacterPhoto() function.

  ```
  Texture2D photoTexture = characterEntity.GetCharacterPhoto(Vector2 _imageSize, Color _bgColor, float _cameraAngle = 0F, bool _cameraLight = true);
  ```

  _imageSize: A Vector2 specifying the width and height (in pixels) of the generated portrait texture.

  _bgColor: The Color of the background of the portrait.

  _cameraAngle: An optional float value (default: 0F) representing the offset angle around the Y-axis for the camera capturing the portrait.

  _cameraLight: An optional bool value (default: true) to toggle an additional light source on during capture, useful if the character is in a dark environment.

  Return Value: A Texture2D containing the character's portrait.

- **Animating Accessories (Back, Tail, Ears):**

  If your character's **back**, **tail**, or **ear** accessories have their own **Animation** components, you can directly access these components using the following functions:

  ```
  Animation backAnimator = characterEntity.GetBackAnimationComponent();
  Animation tailAnimator = characterEntity.GetTailAnimationComponent();
  Animation headAccessoryAnimator = characterEntity.GetHeadAccessoryAnimationComponent();
  ```

  Example (Playing a back accessory animation):

  ```
  if (Player.GetBackAnimationComponent() != null){
      Player.GetBackAnimationComponent().CrossFade("OpenFlap1", 0.25F);
  }
  ```

  Ensure you check if the component exists before attempting to access it to avoid potential null reference errors.

- **Managing Character Eyes:**

  Control your character's gaze and eye expressions using these functions:

  Look At Target:

  ```
  characterEntity.SetLookAt(Transform _target); // Make the character look at the specified Transform.
  characterEntity.SetLookAt(null);    // Stop the character from looking at any target.
  ```

  Blinking:

  ```
  characterEntity.Blink(); // Trigger a blink animation.
  ```

Eye Openness:

```
characterEntity.SetEyeOpen(float _openPercentage); // Set the eye openness percentage (0f - 100f).
```

- **Applying Rim Effect:**

  A rim effect can be used to highlight your character or provide visual feedback, such as when they are hit.

  Set Rim Color and Intensity:

  ```
  characterEntity.SetRimColor(Color _color, float _intensity);
  ```

  _color: The Color of the rim effect.

  _intensity: A float value (typically between 0 and 1, though your original note mentioned 1~10, clarify the intended range).

  Get Current Rim Color:

  ```
  Color currentRimColor = characterEntity.GetRimColor();
  ```

CHARACTER ENTITY

The <CharacterEntity> component is the key element of the Master Character Creator system, allowing you to control and manipulate the character's appearance and interactions. Below are the essential API calls that give you control over character customization, equipment, and more:

- `public CharacterAppearance mCharacterAppearance;`

  Stores the current appearance data of the character.

- `public Sex sex`

  Returns the gender of this character.

- `public void RandomUid()`

  Assigns a random unique ID for this character.

- `public Texture2D GetCharacterPhoto(Vector2 _imageSize, Color _bgColor, float _cameraAngle=0F,bool _cameraLight=true)`

  Take a photo of the character and return the photo as Texture2D.

- `public void Initialize(CharacterAppearance _data)`

  Initializes the character with a specified appearance data.

- `public void Initialize(Sex_sex)`

  Initializes the character with a specified gender.

- `public void LoadFromResourceFile(string _resourcePath)`

  Initializes the character with a "***.bytes**" file in your Resources folder, please use relative path without extension from resources folder. Example:

  "MasterCharacterCreator/CustomBlueprints/Characters/NpcPreset_1"

- `public void LoadFromByteFileFromDisk(string _absolutePath)`

  Initializes the character with a "***.bytes**" file from disk, please use absolute full path and with extension. Example: **"E:/NpcPreset_1.bytes"**

- `public void LoadFromPngFileFromDisk(string _absolutePath)`

  Initializes the character with a "***.png**" file from disk, please use absolute full path and with extension. Example: **"E:/NpcPreset_1.png"**
- `public void LoadFromBytes (byte_[] _bytes)`

  Initializes the character with the bytes array loaded from your own save system.
- `public void SaveByteFileToDisk(string _absolutePath, BlurPrintType _filter)`

  Save the character to a "*.**bytes**" file on disk, please use absolute full path and with extension. Example:"**E:/Player.bytes**"
- `public void SavePngFileToDisk(string _absolutePath, Texture2D _photo, BlurPrintType _filter)`

  Save the character to a "*.**png**" file on disk, please use absolute full path and with extension. Example:**"E:/Player.png"**
- `public void GetSaveBytes (BlurPrintType _filter)`

  Get the save data of the character as bytes array, so you can save it with your own save system.
- `public Transform GetBoneByName(string _name)`

  Get the bone transform of the character by the name of the bone. You can find the name of bones by checking the prefab of characters in : *Assets/SoftKitty/MasterCharacterCreator/Resources/MasterCharacterCreator/Player/Character Male.prefab & CharacterFemale.prefab*
- `public void Equip(EquipmentAppearance _equipment)`

  Equips a gear item and updates the character's appearance accordingly.
- `public void Equip(OutfitSlots _slot, int _id)`

  Equips an item with its slot and id and updates the character's appearance accordingly.
- `public void Equip(OutfitSlots _slot, int _id, Color _color1,Color _color2,Color _color3)`

  Equips an item with its slot, id and custom colors, then updates the character's appearance accordingly.
- `public void Unequip(OutfitSlots _slot)`

  Unequips a specific outfit slot (e.g., helmet, armor).
- `public bool isEquipped(EquipmentAppearance _equipment)`

  Returns whether a specific piece of equipment is equipped.
- `public bool isEquipped(OutfitSlots _slot, int _id)`

  Returns whether a specific equipment with provided slot and id is equipped.
- `public int GetEquippedId(OutfitSlots _slot)`

  Retrieves the mesh ID of the currently equipped item in the specified slot.
- `public void SetEmotion(string _uid, float _length = 3F)`

  Sets the character's emotion by its UID and specifies how long the emotion will last.
- `public void SetEyeOpen(float _openPercentage)`

  Sets the percentage of the character's eye openness (1~100).

- `public void Blink()`

  Forces the character to blink immediately.
- `public void SetLookAt(Transform _target)`

  Makes the character look at a specified transform target.
- `public void SetRimColor(Color _color, float _intensity)`

  Set the rim effect color, this could be useful for highlight the character or when the character gets hit.
- `public Color GetRimColor()`

  Get the rim effect color, this could be useful for highlight the character or when the character gets hit.
- `public void ResetCharacter()`

  Resets the character's appearance back to its default state.
- `public void CustomizeCharacter()`

  Switches to the Character Customization UI with this character (for player use)
- `public void CreateCharacter()`

  Switches to the Character Creation UI (for player use)
- `public void CreateCharacterByDeveloper()`

  Switches to the Character Creation UI (for developer use, with more options)
- `public void LoadDefaultWeapon()`

  Switches to the Character Cre
- `public void EquipWeapon(WeaponController _weapon, WeaponState _state= WeaponState.Carry)`

  Load the weapons set in the inspector
- `public void UnequipWeapon(WeaponType _slot)`

  Equip a weapon and set its default state.
- `public void UnequipAllWeapons()`

  Unequip a weapon with specified slot.
- `public void SwitchWeaponState(WeaponState _state, WeaponType _slot)`

  Unequip all weapons
- `public void SwitchWeaponState(WeaponState _state)`

  Switch the state of the weapon with specified slot.
- `public WeaponState GetWeaponState()`

  Switch the state of all weapons
- `public WeaponState GetWeaponState()`

  Get the current weapon state.
- `public bool isEquippedWeapon(string _uid)`

  Return bool value for whether a weapon with specified uid is equipped.
- `public WeaponController GetEquippedWeaponByType(WeaponType _slot)`

  Get a equipped weapon with specified slot

- `public WeaponController GetEquippedWeaponByUid(string _uid)`
  Get a equipped weapon with specified uid, return null if no match found.
- `public List<WeaponController> GetAllEquippedWeapon()`
  Get a list of all equipped weapon

This breakdown gives you the essential tools to control character customization, equipment, and behavior in your game, offering both player and developer access points for customizing or creating characters.

CHARACTER MANAGER

This script manages all aspects of a character's appearance, storing the appearance data in a **Dictionary** using the character's **UID** as the key. Typically, you don't need to manually update the character's appearance as it is automatically handled by "CharacterEntity.cs". However, if you need to modify the character's appearance independently from "CharacterEntity.cs", you can use this script. Additionally, this script allows save/load a character's appearance from a **byte file** or a **PNG file**.

FUNCTIONS FOR MANAGING APPEARANCE DATA

The following functions provided in CharacterManager.cs allow you to efficiently manage the appearance data of your characters without directly relying on CharacterEntity.cs:

- `public static void UpdateCharacterAppearance(string _uid, CharacterAppearance _data)`
  Updates the appearance data of a character identified by the provided **UID**. Use this function if you need to modify a character's appearance independently of CharacterEntity.cs
- `public static bool isCharacterAppearanceExist(string _uid)`
  Checks if the appearance data for a character with the specified **UID** exists in the internal Dictionary.
- `public static CharacterAppearance GetAppearanceData(string _uid)`
  Retrieves the appearance data of a character using the provided **UID**.

These functions allow for flexible appearance management, enabling you to control or modify the visual data of characters even outside the typical workflow of the

FUNCTIONS FOR SAVING/LOADING APPEARANCE DATA

The following functions allow developers to save and load character appearance data efficiently, whether from disk or resources, in both byte and PNG formats:

- `public static CharacterAppearance LoadCharacterDataFromResources(string _resourcePath, CharacterAppearance _data)`
  Load appearance data from a **byte file** located in the *resources* folder
- `public static CharacterAppearance LoadCharacterDataFromFile(CharacterAppearance _data, string _path)`
  Load appearance data from a **byte file** stored on disk.
- `public static Texture2D LoadCharacterDataWithPhoto(ref CharacterAppearance _data, string _path)`
  Load appearance data from a **PNG file** on disk, with the appearance data hidden inside the image
- `public static void SaveCharacterData(CharacterBoneControl _character, string _path, BlurPrintType _type= BlurPrintType.AllAppearance)`
  Save appearance data to a **byte file**. Use `BlurPrintType` to specify if you want to save the entire appearance data or just specific parts
- `public static void SaveCharacterDataWithPhoto(CharacterBoneControl _character, Texture2D _photo, string _path, BlurPrintType _type = BlurPrintType.AllAppearance)`
  Save appearance data to a **PNG file**. The data is embedded within the image file, with the option to limit saved data to certain parts using `BlurPrintType`
- `public static void SaveCharacterDataToResources(CharacterBoneControl _character, string _resourcePath, BlurPrintType _type = BlurPrintType.AllAppearance)`
  [FOR DEVELOPERS] Save appearance data to a **byte file** in the *resources* folder. Use `BlurPrintType` if you intend to save only specific sections of the data

These functions provide flexibility when handling character appearance data, allowing you to choose between different formats and storage locations. Additionally, developers can manage different parts of the character's appearance data through the `BlurPrintType` *parameter.*

MANAGING PREVIEW CHARACTERS

These functions allow you to manage **preview characters**, *which is useful for displaying a character's current appearance within various UI interfaces (e.g., rendering the player's appearance to a* RenderTexture *for display in an equipment or customization interface).*

- `public CharacterBoneControl CreatePreviewCharacter(string _key, Sex _sex, Vector3 _position, float _angle)`

  Creates a **preview character** at a specified position and angle. The **key** is used to identify this character for further operations
- `public CharacterBoneControl GetPreviewCharacter(string _key)`

  Retrieves a **preview character** using the provided **key**. Once retrieved, you can call `CharacterBoneControl().Initialize(CharacterAppearance _data)` to apply appearance data to the **preview character**
- `public void RemovePreviewCharacter(string _key)`

  Removes the **preview character** identified by the provided **key**

These functions allow for dynamic management of preview characters, making it easy to display and update a character's appearance in interfaces like equipment or character creation screens.

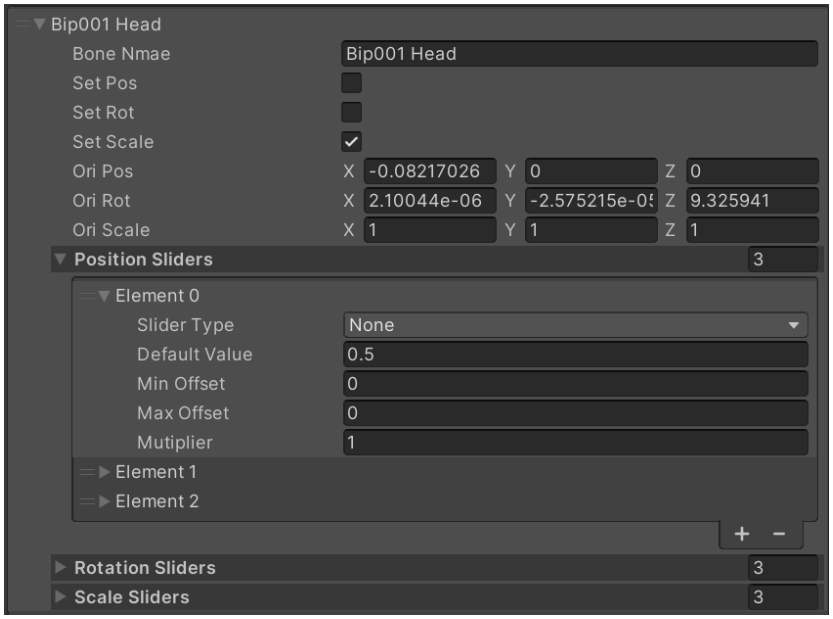
DATA STRUCTURE

The character customization data is organized into the following categories:

BONE SETTINGS | **MATERIAL SETTINGS** | **ACCESSORY SETTINGS** | **OUTFIT SETTINGS**

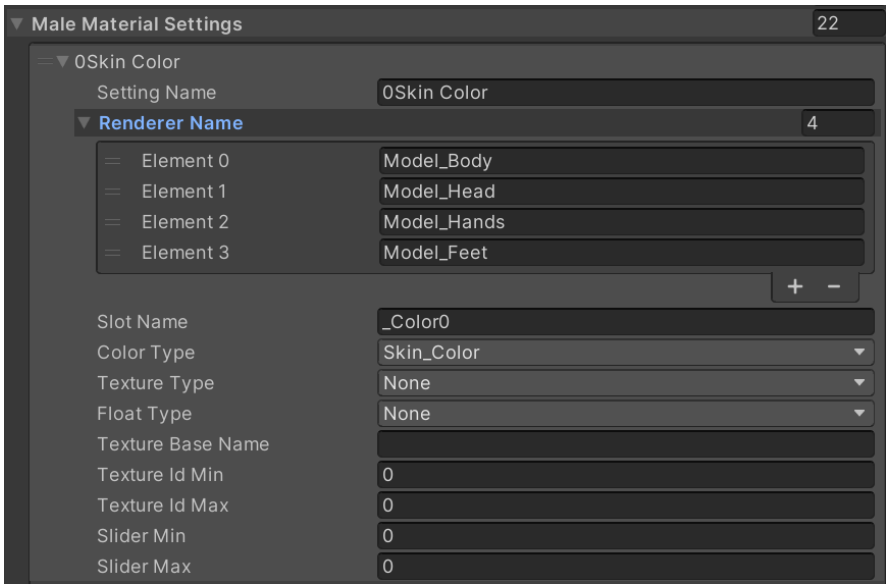
BONE SETTINGS

Defines the character's skeletal structure and proportions. This includes settings for bone scaling, positioning, and rotation to customize the body shape.



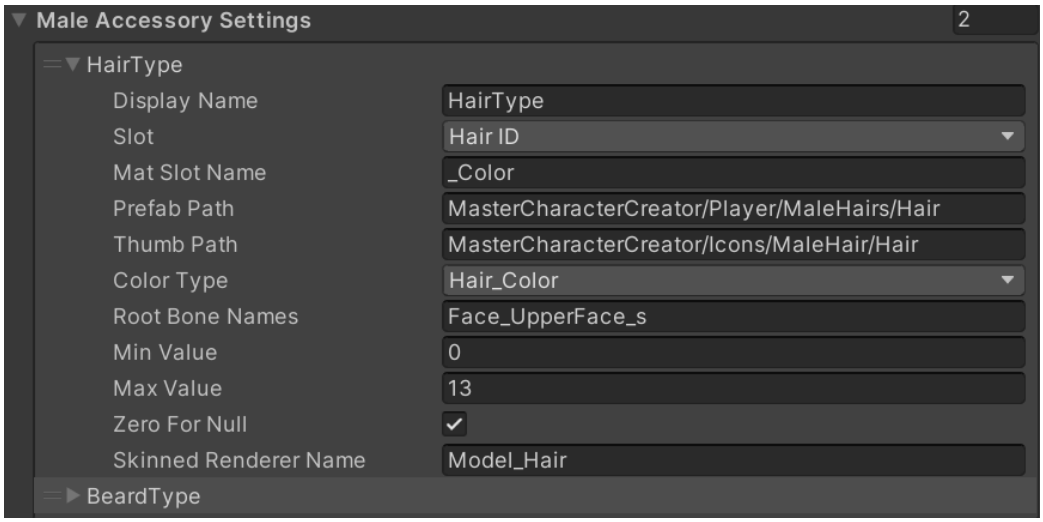
MATERIAL SETTINGS

Controls the visual appearance of the character's skin, hair, and other textures. This includes customization options like skin tone, eye color, hair color, and material attributes like offset and transparency.



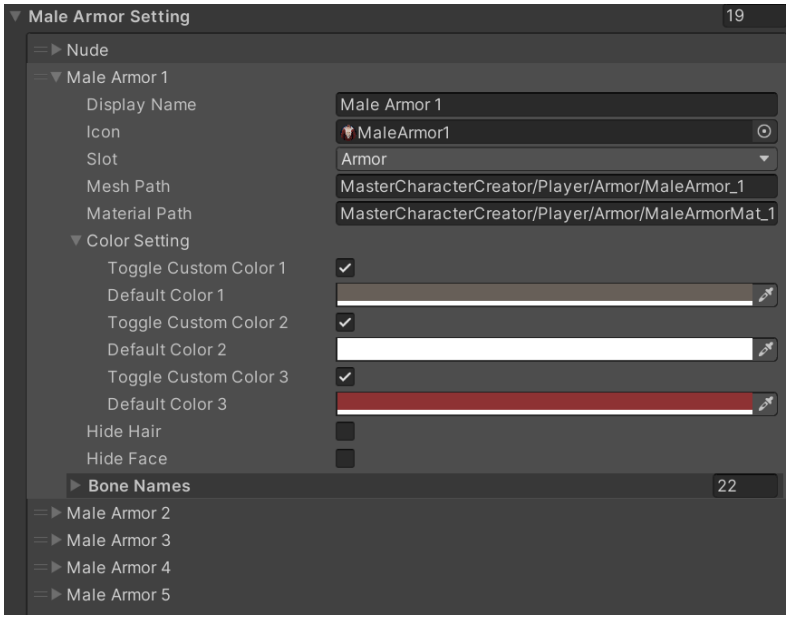
ACCESSORY SETTINGS

Includes customization options for accessories such as beards, hairs, or other character enhancements. These settings allow for adding, modifying, or removing accessories from the character.



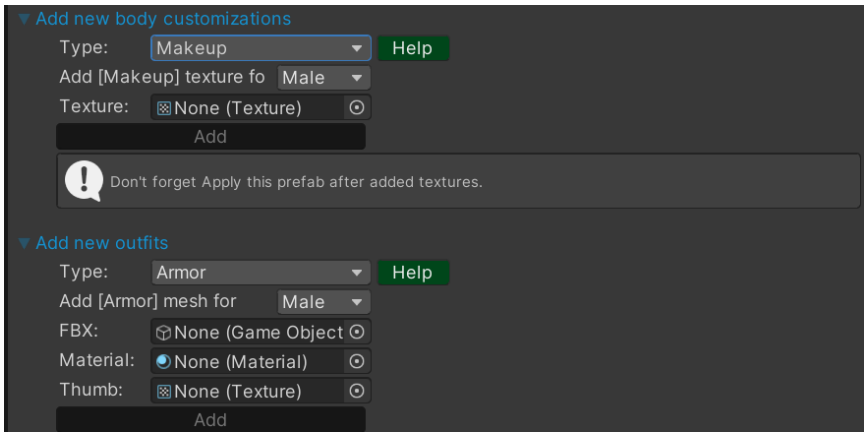
OUTFIT SETTINGS

Manages the character’s clothing and armor, divided into separate parts like **helmet**, **armor**, **gauntlets**, and **boots**. Each slot can be customized or swapped to create unique outfits.



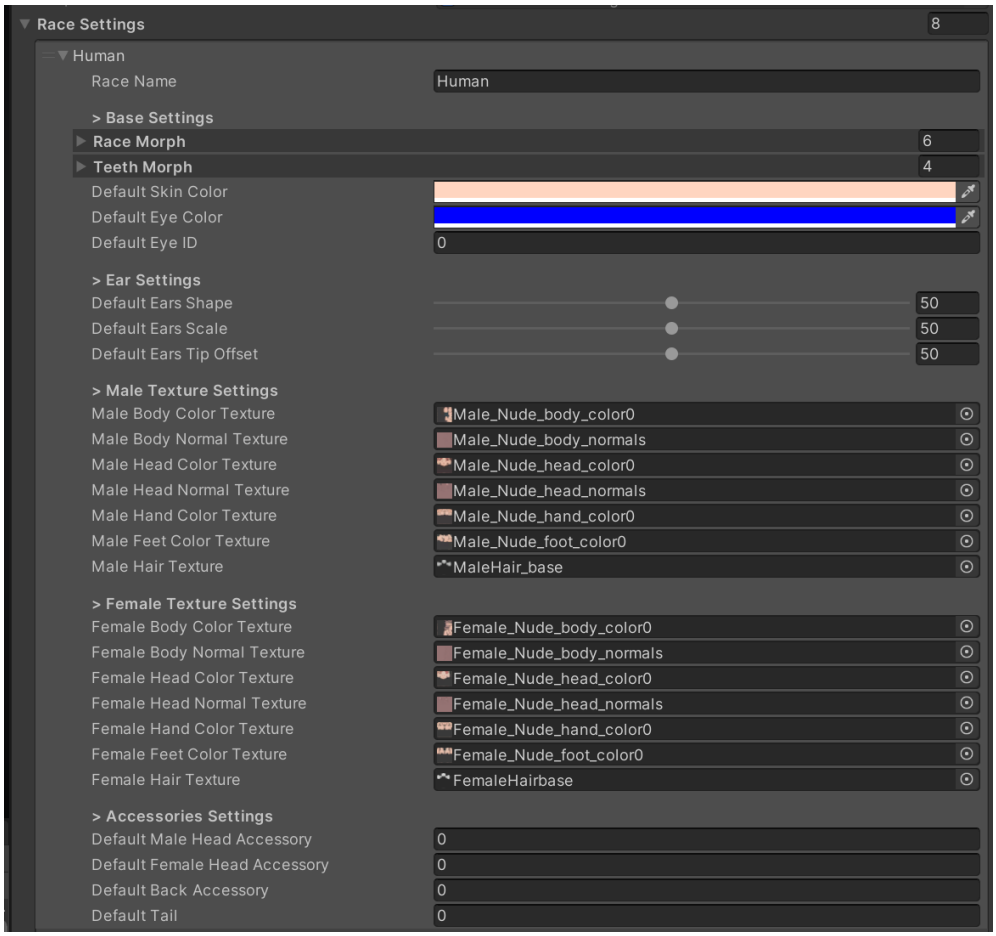
These categories provide a modular structure for fully customizing your character's appearance, allowing developers to tweak everything from body proportions to detailed accessories and

outfits.You can easily modify the above settings on the **CharacterData** prefab manually. Additionally, we provide tools on the **CharacterData** prefab that allow you to set up those customization options with a single click when adding new customization items. For more detailed instructions, please refer to the other documentation provided.



RACE SETTINGS

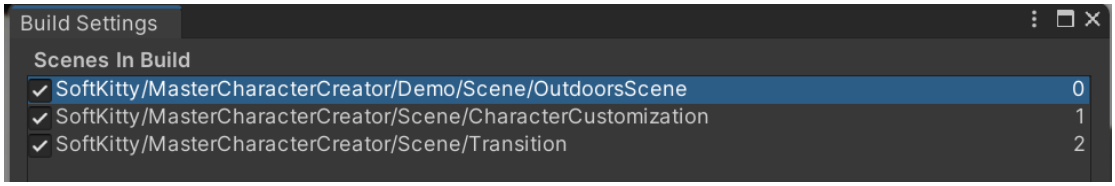
You can easily customize or create new race with the race settings:



You can blending the head morphs and assigning textures for each race, you can also adjust the default skin color, eye color, ear shape, head accessory id, back accessory id and tail id.

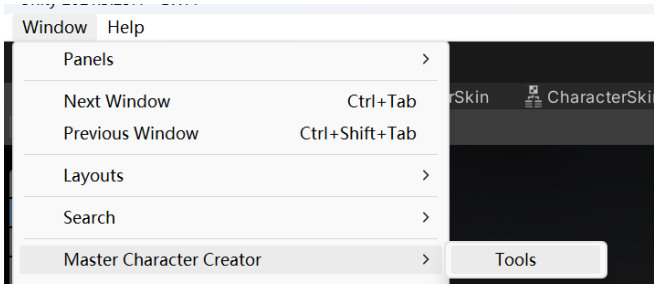
WORK FLOW

As mentioned earlier, you can use the runtime interface to create characters for NPCs or player presets. It is recommended to build the demo scene as an external tool, allowing you to run it while you develop your game in Unity. When you build the demo scene, please make sure the "**Scene In Build**" of **Build Settings** looks like this:

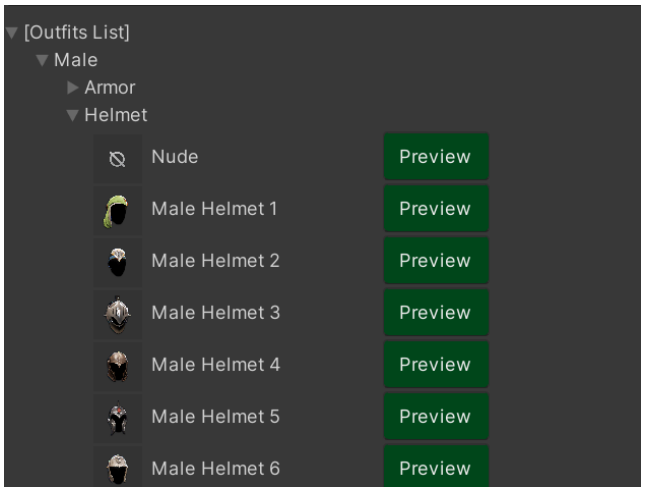


We provide a handy tool that allows you to browse through the outfits and preview them quickly.

1. Access the tool though **Window > Master Character Creator > Tools**



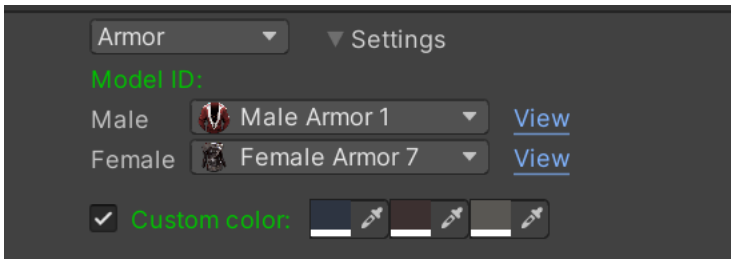
2. You can view the outfits list in the "[Outfits List]" section:



Click on the “Preview” button to preview the mesh in the pop up window:

This way, you can quickly find the suitable outfit to bind with the equipment in your game.

When you add the EquipmentAppearance class to your equipment data, the inspector will look like this:



You can select and preview the **mesh** for both Male and Female characters. Additionally, you can toggle the "**Custom Color**" option to override the default color, allowing you to have multiple pieces of equipment with the same mesh ID but different colors.