Documentation

_____

Creators

- Lucas Gomes Cecchini

- Gabriel Henrique Pereira

Overview

This document will help you use the Assets **Unity MMD Collection.**

With it you have tools that can facilitate the use of **MMD models** in **Unity**, as well as custom **Shaders** for this.

The **Shaders** were made based on studies in the MMD4Mecanim package , being recreated and improved in two tools: The Shader Graph and Amplify Shader Editor due to the capabilities and applications of each resource.

It also has **Scripts** that shape the Unity interface to make it similar to the **MMD tools** and make it more practical and faster to use. In addition to other things that allow you to speed up development.

Instructions

You can get more information in the Playlist on YouTube:

https://youtube.com/playlist?list=PL5hnfx09yM4IWSWveW0NKCfX1Anec4dw7&si=UIqEtAoFcXZH7WFM

Compatibility

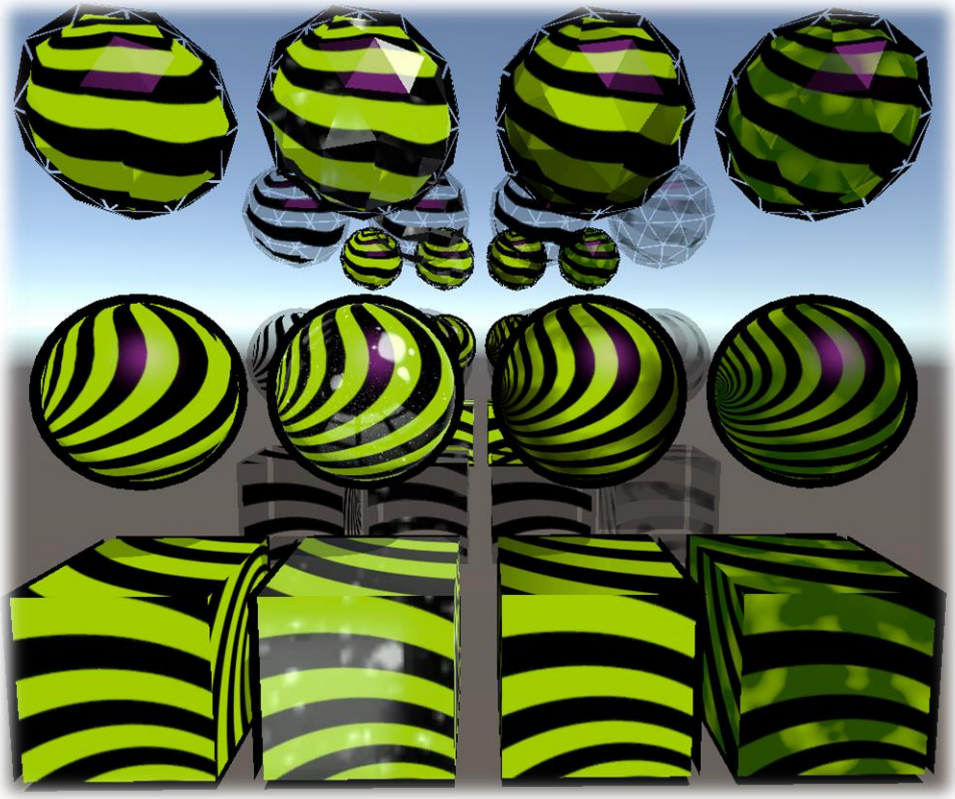
This package is compatible with:

- Unity 6000.0.45f1

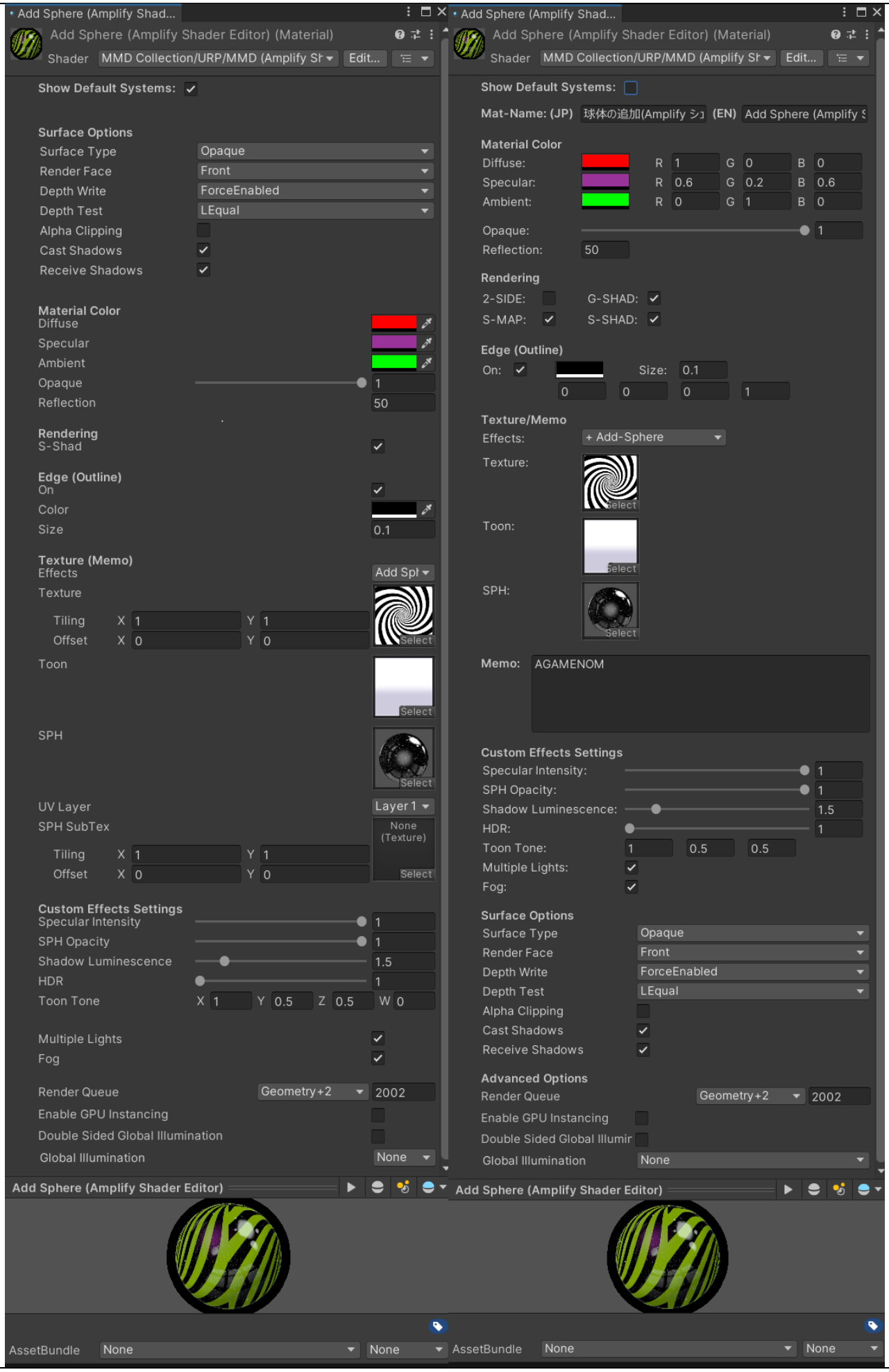
- Amplify Shader Editor v 1.9.9

- MMD4Mecanim 2020-01-05

Shader Explanations



MMD Shader


Add Sphere (Amplify Shad...
Add Sphere (Amplify Shader Editor) (Material)
Shader
MMD Collection/URP/MMD (Amplify Sh
Edit...
Show Default Systems:
Surface Options
Surface Type
Opaque
Render Face
Front
Depth Write
ForceEnabled
Depth Test
LEqual
Alpha Clipping
Cast Shadows
Receive Shadows
Material Color
Diffuse
Specular
Ambient
Opaque
Reflection
50
Rendering
S-Shad
Edge (Outline)
On
Color
Size
0.1
Texture (Memo)
Effects
Add Sph
Texture
Tiling
Offset
Select
Toon
SPH
UV Layer
Layer 1
SPH SubTex
None (Texture)
Custom Effects Settings
Specular Intensity
SPH Opacity
Shadow Luminescence
1.5
HDR
Toon Tone
Multiple Lights
Fog
Render Queue
Geometry+2
2002
Enable GPU Instancing
Double Sided Global Illumination
Global Illumination
None
Add Sphere (Amplify Shader Editor)
Mat-Name: (JP)
球体の追加(Amplify シ
(EN)
Add Sphere (Amplify S
Material Color
Diffuse:
Specular:
Ambient:
R 1 G 0 B 0
R 0.6 G 0.2 B 0.6
R 0 G 1 B 0
Opaque:
Reflection:
Rendering
2-SIDE:
G-SHAD:
S-MAP:
S-SHAD:
Edge (Outline)
On:
Size:
0.1
Texture/Memo
Effects:
+ Add-Sphere
Texture:
Toon:
SPH:
Memo:
AGAMENOM
Custom Effects Settings
Specular Intensity:
SPH Opacity:
Shadow Luminescence:
HDR:
Toon Tone:
Multiple Lights:
Fog:
Surface Options
Surface Type
Opaque
Render Face
Front
Depth Write
ForceEnabled
Depth Test
LEqual
Alpha Clipping
Cast Shadows
Receive Shadows
Advanced Options
Render Queue
Geometry+2
2002
Enable GPU Instancing
Double Sided Global Illumir
Global Illumination
None
AssetBundle
None
None

Shader Properties

==System Properties==

- **Show Default Systems** = Shows default or custom properties.

- **Mat-Name (JP)** = Material name in Japanese.

- **Mat-Name (EN)** = Material name in English.

- **Memo** = Text header for notes.

==Material Color==

- **Diffuse** = Color of the material.
- **_Color ("Diffuse", Color)** : Sets the main color of the material when illuminated directly.

- **Specular** = Color of light reflection/shine.
- **_Specular ("Specular", Color)** : Sets the color of the specular reflections in the material.

- **Ambient** = Affects the color and lighting of the model, similar to a raycast.
- **_Ambient ("Ambient", Color)** : Sets the color to ambient light, contributing to the overall lighting.

- **Opaque** = Alpha value (opacity)
- **_Opaque ("Opaque", Range(0, 1))** : Controls the opacity of the material.

- **Reflection** = Reflection value
- **_Shininess ("Reflection", Float)** : Sets the intensity of the specular shine (reflection) of the material.

==Rendering==

- **2-SIDE** = Renders both sides of the mesh **[Equivalent to Render Face/_Cull]**
- **_Cull("Render Face", Float)** : Controls which faces of the mesh (front, back or both) should be rendered.

- **G-SHAD** = Shadow on the ground **[Equivalent to Cast Shadows/ShaderPass"SHADOWCASTER"]**
- **_CastShadows ("Cast Shadows", Float)** : Defines whether the material casts shadows on the ground.

- **S-MAP** = Shadow on mesh (including the mesh itself) **[Equivalent to Receive Shadows/_ReceiveShadows/Keyword"_RECEIVE_SHADOWS_OFF"]**
- **_ReceiveShadows ("Receive Shadows", Keyword)** : Controls whether the mesh receives shadows from other sources.

- **S-SHAD** = Receives shadow only from itself
- **_SShad ("S-SHAD", Float)** : Controls the shadow the mesh casts on itself only.

==Edge (Outline)==

- **On** = Activates the contour
- **_On ("On", Float)** : Controls whether the outline (border) is activated.

- **Color** = Outline color, including transparencies
- **_OutlineColor ("Color", Color)** : Sets the color of the outline around the mesh.

- **Size** = Size/distance of the contour
- **_EdgeSize ("Size", Float)** : Controls the thickness of the outline.

==Texture/Memo==

- **Texture** = Texture of the material
- **_MainTex ("Texture", 2D)** : Main texture applied to the material.

- **Toon** = Complements the object's shading
- **_ToonTex ("Toon", 2D)** : Toon texture, used for cartoon-style shading.

- **UV Layer** = UV layer to be used
- **_UVLayer ("UV Layer", Float)** : Selects which UV layer will be applied to the texture.

- **SPH** = Artificial reflection to complement Specular
- **_SphereCube ("SPH", CUBE)** : Cubic texture used for artificial reflections.

==Effects==

- **Disabled** = Does nothing
- **_EFFECTS ("Effects", Float)** : Disables effects.

- **Multi-Sphere** = Multiplies a shiny sphere map (metallic reflection)
- **_EFFECTS ("Effects", Float)** : Activates the effect of multiple glowing spheres.

- **Add-Sphere** = Creates a glowing sphere map (practical reflection)
- **_EFFECTS ("Effects", Float)** : Activates the effect of adding glowing spheres.

- **Sub-Tex** = Adds an extra UV texture layer for more complex effects
- **_EFFECTS ("Effects", Float)** : Switches from using the cubic texture to using a subtexture, applying it to another UV layer.

==Custom Effects Settings==

- **Specular Intensity** = Sets the intensity of the specular gloss
- **SpecularIntensity ("Specular Intensity", Range(0, 1))** : Intensity of the specular brightness.

- **SPH Opacity** = Sets the opacity of the Cubic Texture
- **SPHOpacity ("SPH Opacity", Range(0, 1))** : Opacity of the cubic texture.

- **Shadow Luminescence** = Shadow intensity
- **_ShadowLum ("Shadow Luminescence", Range(0, 10))** : Luminescence of shadows.

- **HDR** = Makes the object glow in the dark
- **_HDR ("HDR", Range(1, 1000))** : Controls HDR mapping.

- **Toon Tone** = Adjusts the tone of the Toon shadow
  - **_ToonTone ("Toon Tone", Vector)** : Defines the tone of the shading toon.

- **Multiple Lights** = Allows the object to receive multiple lights
- **_MultipleLights ("Multiple Lights", Float)** : Enables or disables support for multiple lights.

- **Fog** = Turns fog on and off
  - **_Fog ("Fog", Float)** : Enables or disables fog support.

" **Surface Options"** and **"Advanced Options"** are Unity standards that require prior knowledge of their functionality.

Known bugs

Decal not showing up in Build: To resolve this, add the Decal module to all 'Universal Renderer Data' in your project.
Distorted shadows when using Morph: To resolve this, enable the 'Legacy Blend Shape Normals'

option in the **.FBX** (or 3D model) when importing from Unity.
Light passing through objects: Add a 'Directional Light' with at least 0.001 Intensity to update the Dynamic Lightmap

Script Explanations

Free Camera

This script defines a **`FreeCamera` class** in C# for use in a Unity project. It allows the camera to move freely in 3D space, responding to keyboard and mouse input. Below is a detailed explanation of the script:

Class Declarations and Attributes

1. Dependencies and Initial Configurations :

```
[RequireComponent(typeof(Camera))]
[AddComponentMenu("MMD Collection/Free Camera")]
public class FreeCamera : MonoBehaviour
```

- **`RequireComponent(typeof(Camera))`** : Ensures that the object this script is attached to has a **`Camera` component** .
- **`AddComponentMenu("MMD Collection/Free Camera")`:** Adds this script to the components menu, under the specified path.

2. Camera Configuration Variables:

```
[Header("Camera Settings")]
[SerializeField] private float movementSpeed = 10f;
[SerializeField] private float fastMovementSpeed = 100f;
[SerializeField] private float sensitivity = 3f;
[SerializeField] private float zoomSensitivity = 10f;
[SerializeField] private float fastZoomSensitivity = 50f;
```

- **`movementSpeed`** : Default camera movement speed.
- **`fastMovementSpeed`** : Increased movement speed when fast mode is enabled.
- **` sensitivity`** : Mouse sensitivity for rotation.
- **`zoomSensitivity`** and **`fastZoomSensitivity`** : Sensitivity for zoom (normal and fast).

3. Key Settings:

```
[Header("Key Settings")]
public KeyCode left = KeyCode.A;
public KeyCode right = KeyCode.D;
public KeyCode front = KeyCode.W;
public KeyCode back = KeyCode.S;
[Space(10)]
public KeyCode up = KeyCode.Q;
public KeyCode down = KeyCode.E;
[Space(10)]
public KeyCode upGlobal = KeyCode.R;
public KeyCode downGlobal = KeyCode.F;
[Space(10)]
public KeyCode observe = KeyCode.Mouse1;
[Space(10)]
public KeyCode run = KeyCode.LeftShift;
```

- Set keys to move the camera in different directions and switch modes.

4. Axis Settings:

```
[Header("Axis Settings")]
public string zoom = "Mouse ScrollWheel";
public string axisX = "Mouse X";
public string axisY = "Mouse Y";
```

- Sets the mouse axes for zoom and rotation.

5. Private Variable:

```
private bool looking = false;
```

- `looking` : Indicates whether the camera is in observation mode (free rotation with the mouse).

`Update` Method

6. Camera Movement:

```
var fastMode = Input.GetKey(run);
var currentMovementSpeed = fastMode ? fastMovementSpeed : movementSpeed;

Vector3 moveDirection = Vector3.zero;
if (Input.GetKey(left)) moveDirection += -transform.right;
if (Input.GetKey(right)) moveDirection += transform.right;
if (Input.GetKey(front)) moveDirection += transform.forward;
if (Input.GetKey(back)) moveDirection += -transform.forward;
if (Input.GetKey(up)) moveDirection += transform.up;
if (Input.GetKey(down)) moveDirection += -transform.up;
if (Input.GetKey(upGlobal)) moveDirection += Vector3.up;
if (Input.GetKey(downGlobal)) moveDirection += Vector3.down;

transform.position += currentMovementSpeed * Time.deltaTime * moveDirection;
```

- Checks if fast mode is enabled and adjusts movement speed.
- Calculates the direction of movement based on the keys pressed.
- Updates the camera position.

7. Camera Rotation:

```
if (looking)
{
    float newRotationX = transform.localEulerAngles.y + Input.GetAxis(axisX) * sensitivity;
    float newRotationY = transform.localEulerAngles.x - Input.GetAxis(axisY) * sensitivity;
    transform.localEulerAngles = new Vector3(newRotationY, newRotationX, 0f);
}
```

- Adjusts camera rotation based on mouse movement if observation mode is active.

8. Camera Zoom:

```
float zoomAxis = Input.GetAxis(zoom);
if (zoomAxis != 0)
{
    var currentZoomSensitivity = fastMode ? fastZoomSensitivity : zoomSensitivity;
    transform.position += zoomAxis * currentZoomSensitivity * transform.forward;
}
```

- Adjusts camera zoom based on mouse scrolling.

9. Observation Mode:

```
if (Input.GetKeyDown(observe))
{
    StartLooking();
}
else if (Input.GetKeyUp(observe))
{
    StopLooking();
}
```

Auxiliary Methods

10. `OnDisable` Method:

```
private void OnDisable()
{
    StopLooking();
}
```

- Ensures that watch mode is disabled when scripting is disabled.

11. Methods for Starting and Stopping Observation Mode:

```
public void StartLooking()
{
    looking = true;
    Cursor.visible = false;
    Cursor.lockState = CursorLockMode.Locked;
}

public void StopLooking()
{
    looking = false;
    Cursor.visible = true;
    Cursor.lockState = CursorLockMode.None;
}
```

- `**StartLooking**` : Activates looking mode, hiding and locking the cursor.
- ` **StopLooking**` : Disables looking mode, showing and releasing the cursor.

Summary

This script provides detailed and customizable control over camera movement and rotation in a 3D environment, using keyboard and mouse input. It allows for smooth and intuitive movement, similar to that of an FPS game, and includes functionality for accelerating movement and zooming, as well as switching between observation and normal movement modes.

Copy Animation

This Unity script, called CopyAnimation, is a component responsible for copying rotations from a source bone set to a target bone set in real time. Below is a detailed explanation of each part of the script:

Component and Variable Definitions

1. Imports and Class Attributes

```
using UnityEngine;


[AddComponentMenu("MMD Collection/Copy Animation")]
public class CopyAnimation : MonoBehaviour
```

- `**using UnityEngine;**` : Imports the UnityEngine namespace, required to use Unity classes and methods.
- `**[AddComponentMenu("MMD Collection/Copy Animation")]**` : Adds this script to the Unity components menu under the "MMD Collection" category.

2. Public and Private Variables

```
[Header("Settings")]
public bool update = true;
[Header("Copy and Paste Rotations")]
[SerializeField] private Transform[] copyBone;
[Space(10)]
[SerializeField] private Transform[] pasteBone;
```

- `**public bool update**` : Controls whether rotations should be updated or not.
- `**[SerializeField] private Transform[] copyBone**` : Array of source bones from which rotations will be copied.
- `**[SerializeField] private Transform[] pasteBone**` : Array of target bones where the rotations will be pasted.

`LateUpdate` method

3. `LateUpdate` Method

```
private void LateUpdate()
{
    if (!update) return;
    if (copyBone.Length != pasteBone.Length)
    {
        Debug.LogError("Copy and paste arrays must be of the same length!", this);
        return;
    }

    for (int i = 0; i < copyBone.Length; i++)
    {
        if (copyBone[i] != null && pasteBone[i] != null)
        {
            pasteBone[i].rotation = copyBone[i].rotation;
        }
        else
        {
            Debug.LogWarning("Copy or paste transform is null!", this);
        }
    }
}
```

- `**if (!update) return;**` : If `**update**` is `**false**` , the method returns and does nothing.
`**copyBone**` and `**pasteBone**` arrays have the same length. If not, prints an error to the console and returns.
- Iterates through each pair of bones, copying the rotation from the source bone to the target bone.

Final Considerations

- This script is useful in animations, especially in rigging systems and skeletal animations, where the rotation of certain bones needs to be synchronized or limited.
- Using arrays for `**copyBone**` and `**pasteBone**` allows copying of multiple bones at once, ensuring that the skeletal hierarchy maintains correct rotations.

Draw Mesh Instanced

This script is a Unity component that draws mesh instances using the graphics instancing technique.

Imports and Class Declaration

```
using System.Collections.Generic;
using UnityEngine;

[AddComponentMenu("MMD Collection/Draw Mesh Instanced")]
public class DrawMeshInstanced : MonoBehaviour
{
```

- The script starts by importing necessary namespaces: `**System.Collections.Generic**` for list manipulation and `**UnityEngine**` for Unity API access.
`**DrawMeshInstanced**` class inherits from `**MonoBehaviour**` , allowing it to be attached to a GameObject in Unity.
`**[AddComponentMenu("MMD Collection/Draw Mesh Instanced")]**` attribute adds this class to the component menu in the Unity editor.

Configuration Variables

```
[Header("Settings")]
[SerializeField] private bool OnDrawSelected = false;
[SerializeField] private bool reuseMaterials = true;
[Space(10)]
[SerializeField] private List<DrawMeshInstancedList> drawMeshInstancedLists = new();
```

- `**OnDrawSelected**` : If true, draws mesh instances only when the object is selected in the editor.
- `**reuseMaterials**` : If true, reuse materials for submeshes.
- `**drawMeshInstancedLists**` : A list of parameters to instantiate the meshes.

`Update` Method

```
private void Update()
{
    DrawMesh();
}
```

- The `**Update**` **method** is called every frame. It calls the `**DrawMesh**` **method** to draw the mesh instances.

`DrawMesh` Method

```
private void DrawMesh()
{
    foreach (var drawList in drawMeshInstancedLists)
    {
        if (drawList.mesh != null && drawList.materials != null && drawList.materials.Length > 0)
        {
            for (int i = 0; i < drawList.mesh.subMeshCount; i++)
            {
                Material materialToUse = drawList.materials[i % drawList.materials.Length];

                if (!reuseMaterials && i >= drawList.materials.Length)
                {
                    break;
                }

                Matrix4x4[] matrices = new Matrix4x4[] { drawList.transform.localToWorldMatrix };
                Graphics.DrawMeshInstanced(drawList.mesh, i, materialToUse, matrices);
            }
        }
    }
}
```

- This method iterates through `**drawMeshInstancedLists**` .
- For each item in the list, check if the mesh and materials are valid.
- For each submesh (`**subMesh**`), select the appropriate material.
- If `**reuseMaterials**` is false and all materials have already been used, the iteration stops.
- Creates a transformation matrix and calls `**Graphics.DrawMeshInstanced**` to draw the mesh instance.

`OnDrawGizmosSelected` method

```
    private void OnDrawGizmosSelected()
    {
        if (OnDrawSelected)
        {
            DrawMesh();
        }

        Gizmos.color = Color.green;
        Gizmos.DrawSphere(transform.position, 0.15f);

        if (drawMeshInstancedLists.Count > 0)
        {
            foreach (var drawList in drawMeshInstancedLists)
            {
                if (drawList.transform != null)
                {
                    Gizmos.color = Color.red;
                    Gizmos.DrawSphere(drawList.transform.position, 0.15f);
                }
            }
        }
    }
}
```

- This method is called when the object is selected in the Unity editor.
- If `**OnDrawSelected**` is true, calls `**DrawMesh**` .
- Draws green spheres at the object position and red spheres at the `**Transform**` **positions** of `drawMeshInstancedLists`.

Class `DrawMeshInstancedList`

```
[System.Serializable]
public class DrawMeshInstancedList
{
    public Transform transform;
    [Space(5)]
    public Mesh mesh;
    [Space(5)]
    [Tooltip("Materials must support 'Enable GPU Instancing'")]
    public Material[] materials;
}
```

- This class stores the parameters needed to instantiate a mesh.
- **`transform`** : Defines the position, rotation and scale of the instances.
- **`mesh`** : The mesh to be instantiated.
- **`materials`** : The materials applied to the instances. Materials must support the "Enable GPU Instancing" feature.

Summary

`DrawMeshInstanced` script allows you to draw multiple instances of a mesh with different materials and positions defined by transforms. It also provides the functionality to draw these instances only when the object is selected in the Unity editor. The **`DrawMeshInstancedList` helper class** makes it easy to configure these parameters in the editor.

Custom MMD Data

Overview

It defines a `**ScriptableObject**` called `**CustomMMDData**` to store custom data related to MMD (MikuMikuDance), which is a popular 3D animation software in Japan.

Code Details

Namespace and Libraries

```
using System.Collections.Generic;
using UnityEngine;
```

- `**using System.Collections.Generic;**` : Imports the library to use generic collections, such as lists.
- `**using UnityEngine;**` : Imports the Unity core library that contains essential functionality for game development.

ScriptableObject Definition

```
public class CustomMMDData : ScriptableObject
{
    [Header("MMD Material Settings")]
    public bool showSystemsDefault;
    public List<MMDMaterialInfo> materialInfoList = new();
}
```

- `**public class CustomMMDData : ScriptableObject**` : Defines a class that inherits from `**ScriptableObject**` . `**ScriptableObject**` is a convenient way to store large sets of data that can be easily edited in the Unity editor and used in different parts of a game.
- `**[Header("MMD Material Settings")]**` : Adds a header in the Unity Inspector to better organize and identify fields.
- `**public bool showSystemsDefault;**` : A boolean field indicating whether to show more shader systems (system defaults).
- `**public List<MMDMaterialInfo> materialInfoList = new();**` : A list to store information about MMD materials. The list is initialized as a new empty list.

Serializable Class

```
[System.Serializable]
public class MMDMaterialInfo
{
    public Material mmdMaterial;
    public string materialNameJP;
    public string materialNameEN;
    public string materialMeno;
}
```

- `**[System.Serializable]**` : Allows the class to be serializable, meaning its data can be displayed and

edited in the Unity Inspector.
- `**public class MMDMaterialInfo**` : Defines a class to store information about a specific MMD material.
- `**public Material mmdMaterial;**` : Reference to a Unity material.
- `**public string materialNameJP;**` : Material name in Japanese.
- `**public string materialNameEN;**` : Material name in English.
- `**public string materialMeno;**` : Field for additional notes about the material.

General Operation

1. CustomMMDData:
- It is a `**ScriptableObject**` that stores a list of `**MMDMaterialInfo**` **objects** and a boolean flag (`**showSystemsDefault**`).
- Can be used to organize and manage MMD material data in Unity.

2. MMDMaterialInfo:
- Stores details about a specific material, including references to the material in Unity and its names in Japanese and English, as well as a field for notes.

Usage in Unity

- This `**ScriptableObject**` can be created and edited directly in the Unity editor, allowing easy management and organization of MMD material data.
- Developers can add, remove, and modify entries in the `**materialInfoList**` **list** via the Unity Inspector.
- The `**showSystemsDefault**` **flag** can be used to toggle specific display or behavior settings within the game or tool.

Custom MMD Data Utility Editor

This C# script is used in the Unity Editor to manage assets of type **`CustomMMDData`** .

Overview

This script provides a static utility class called **`CustomMMDDataUtilityEditor`** , which includes methods for finding, creating, and managing **`CustomMMDData` assets** within the Unity Editor.

Code Details

Namespace and Libraries

```
using System.Collections.Generic;
using System.IO;
using UnityEditor;
using UnityEngine;
```

- **`using System.Collections.Generic;`** : Imports the library to use generic collections, such as lists.
- **`using System.IO;`** : Imports input and output functionality, such as file and directory manipulation.
- **`using UnityEditor;`** : Imports Unity editor-specific functionality.
- **`using UnityEngine;`** : Imports the Unity core library.

Utility Class

```
// Utility class for managing CustomMMDData assets within the Unity Editor.
public static class CustomMMDDataUtilityEditor
{
    // Retrieves or creates the CustomMMDData asset.
    public static CustomMMDData GetOrCreateCustomMMDData()
    {
        // Attempt to find an existing CustomMMDData asset.
        CustomMMDData customMMDData = FindCustomMMDData();

        // If not found, create a new one.
        #pragma warning disable IDE0270
        if (customMMDData == null)
        {
            customMMDData = CreateCustomMMDData();
        }
        #pragma warning restore IDE0270

        return customMMDData;
    }
```

- **`public static class CustomMMDDataUtilityEditor`** : Defines a static class for utility methods related to **`CustomMMDData`** .
- **`public static CustomMMDData GetOrCreateCustomMMDData()`** : Method that retrieves or creates a **`CustomMMDData` asset** .

existing **`CustomMMDData`** asset by calling **`FindCustomMMDData()`** .

- If it doesn't find one, create a new one by calling **`CreateCustomMMDData()`** .

Method for Finding an Existing Asset

```
// Finds an existing CustomMMDData asset in the project.
private static CustomMMDData FindCustomMMDData()
{
    string[] guids = AssetDatabase.FindAssets("Custom MMD Data t:CustomMMDData");

    // If found, load the first asset.
    if (guids.Length > 0)
    {
        string path = AssetDatabase.GUIDToAssetPath(guids[0]);
        return AssetDatabase.LoadAssetAtPath<CustomMMDData>(path);
    }

    return null; // No existing asset found.
}
```

- **`private static CustomMMDData FindCustomMMDData()`** : Method that finds an existing **`CustomMMDData` asset** in the project.
- Uses **`AssetDatabase.FindAssets`** to search for assets of type **`CustomMMDData`** .
- If found, load the first asset found using **`AssetDatabase.LoadAssetAtPath`** .

Method for Creating a New Asset

```
// Creates a new CustomMMDData asset.
private static CustomMMDData CreateCustomMMDData()
{
    string folderPath = "Assets/Resources"; // Set folder path.

    // Check if the folder exists; if not, create it.
    if (!Directory.Exists(folderPath))
    {
        Directory.CreateDirectory(folderPath);
    }

    string assetPath = AssetDatabase.GenerateUniqueAssetPath(folderPath + "/Custom MMD Data.asset");
    CustomMMDData customMMDData = ScriptableObject.CreateInstance<CustomMMDData>(); // Create a new instance of CustomMMDData.

    // Create the asset and save it.
    AssetDatabase.CreateAsset(customMMDData, assetPath);
    AssetDatabase.SaveAssets();
    AssetDatabase.Refresh();

    return customMMDData; // Return the newly created asset.
}
```

- **`private static CustomMMDData CreateCustomMMDData()`** : Method that creates a new **`CustomMMDData` asset** .
- Defines the path of the folder where the asset will be stored.
- Checks if the folder exists and if it does not exist, creates the folder.
- Generates a unique path to the asset and creates a new instance of **` CustomMMDData`** .
- Creates the asset at the specified path, saves it and updates the asset database.

Method for Removing Invalid Materials

```
// Removes any invalid materials from the CustomMMDData asset.
public static void RemoveInvalidMaterials(CustomMMDData customMMDMaterialData)
{
    // Ensure the asset is not null.
    if (customMMDMaterialData == null)
    {
        return; // Exit if the asset is null.
    }

    List<MMDMaterialInfo> validMaterials = new(); // Create a list to store valid materials.

    // Iterate through each material info in the asset.
    foreach (MMDMaterialInfo materialInfo in customMMDMaterialData.materialInfoList)
    {
        // Check if the material reference is not null.
        if (materialInfo.mmdMaterial != null)
        {
            validMaterials.Add(materialInfo); // Add to the valid materials list.
        }
    }

    customMMDMaterialData.materialInfoList = validMaterials; // Replace the material info list with the list of valid materials.
}
```

- **`public static void RemoveInvalidMaterials(CustomMMDData customMMDMaterialData)`** : Method that removes invalid materials from the **`CustomMMDData` asset** .
- Checks if the asset is null and exits the method if it is.
- Creates a new list to store valid materials.
- Iterates over each **`MMDMaterialInfo` in the `materialInfoList`** list and adds to the list of valid materials only if the material reference is not null.
- Replaces the **`materialInfoList` list** with the list of valid materials.

General Operation

1. Creating or Retrieving `CustomMMDData`:

`GetOrCreateCustomMMDData` method attempts to find an existing **`CustomMMDData`** asset .
- If not found, create a new asset.

2. Find `CustomMMDData`:

`FindCustomMMDData` method searches for assets of type **`CustomMMDData`** in the project.
- If found, load and return the first asset found.

3. Create New `CustomMMDData`:

`CreateCustomMMDData` method creates a new **`CustomMMDData` asset** in a specific folder.
- If the folder does not exist, create the folder and then create the asset.

4. Remove Invalid Materials:

`RemoveInvalidMaterials` method removes entries in the **`materialInfoList` list** that have null material references.

`CustomMMDData` assets are always valid and easily accessible within the Unity Editor, making it easier to manage MMD material data in your project.

Detailed Script Explanation

This script is a custom extension for the Material Editor in Unity, providing advanced functionality for inspecting and modifying materials that use custom shaders. It extends the **`ShaderGUI` class** , allowing fine-grained control over various properties of shaders and materials directly in the editor.

Class `CustomInspectorUtilityEditor`

The **`CustomInspectorUtilityEditor` class** inherits from **`ShaderGUI`** and serves as the basis for creating custom shader inspectors. It provides several static functions and methods for dynamically and interactively loading, saving, and rendering material properties.

Main Variables

- **`private Material currentMaterial`** : Stores a reference to the material currently being inspected.
- **`private CustomMMDMaterialData customMMDMaterialData`** : An object that stores material-specific data, such as names and other metadata.
- **`private bool showSystemsDefault`** : A boolean that determines whether to display the system's default settings.

`LoadData` method

This method loads the existing data for the current material (**`currentMaterial`**) and stores it in the relevant variables. It checks whether the **`customMMDMaterialData` object** exists, and if so, loads information such as the Japanese and English names of the material (**`materialNameJP`** , **`materialNameEN`**), the description (**`materialMemo`**), and the setting about whether to display the system default values (**`showSystemsDefault`**).

`SaveData` method

This method saves the current material data. It uses the **`DetectChanges` method** to check if there have been any changes to the material data. If there are any changes, it updates the information and marks the object as "dirty" (**`SetDirty`**), indicating that there are changes to be saved. Finally, it persists these changes to the Unity asset system.

`DetectChanges` method

This method compares the current values of the material properties with the values previously stored in **`customMMDMaterialData`** . It returns **`true`** if there were changes and **`false`** otherwise, letting the system know when to save the changes.

Rendering Methods

These methods are responsible for rendering the various material properties in the Unity inspector. They allow the developer to view and modify specific shader attributes directly in the Unity interface.

- **`RenderSurfaceOptions`** : Render surface options, such as surface type, blending mode, culling, among others.
- **`RenderLightmapFlags`** : Render lightmap options, including realtime, baked, and emissive.
- **`RenderColorProperty`** : Renders a color property with an associated label.
- **`RenderSliderFloatProperty` : Renders a `float`** type property as a slider.
- **`RenderDoubleSidedToggle`** : Renders a toggle to define whether the rendering will be double-sided.
- **`RenderShaderPassToggle`** : Renders a toggle to enable or disable a specific shader **`pass` .**
- **`RenderKeywordToggle`** : Renders a toggle to enable or disable a specific **`keyword` .**
- **`RenderUIToggle`** : Renders a toggle for UI properties.

- **`RenderUIColorProperty`** : Renders a specific color property to UI.
- **`RenderFloatProperty` : Renders a `float`** type property with a label.
- **`RenderVector4Property`** : Renders a property of type **`Vector4`** .
- **`RenderDropdownProperty`** : Renders a dropdown property, checking if the options and values match. Otherwise, throws an exception.
- **`RenderTextureProperty` : Renders a texture-type property with a label and optionally displays the Tiling** and **Offset** fields .
- **`RenderCubemapProperty`** : Renders a cubemap type property with a label.
- **`RenderVector3Property`** : Renders a property of type **`Vector3`** , adjusting each component separately.
- **`RenderDepthWriteDropdown`** : Renders a dropdown for Depth Write options.
- **`RenderBlendingModeDropdown`** : Renders a dropdown for blending mode options, automatically updating the **`srcBlend`** and **`dstBlend` properties** based on the selection.
- **`RenderSurfaceTypeDropdown`** : Renders a dropdown for surface type options (opaque or transparent), setting the material render tags accordingly.

Auxiliary Methods

- **`IsToggleUIPropertyEnabled`** : Checks if a UI toggle property is enabled, based on the property's float value (1.0 = enabled).
- **`HasFloatPropertyValue`** : Checks whether a **`float` property** has a specific value by directly comparing the property's current value.
- **`CheckBlendingMode`** : Checks and adjusts the blending mode based on the current material properties. This function ensures that the blending properties are set correctly depending on the surface mode.

Conclusion

`CustomInspectorUtilityEditor` script provides a robust and intuitive interface for customizing and managing materials that utilize advanced shaders in Unity. It allows developers and artists to have detailed control over how materials are rendered, providing a rich and flexible editing experience directly in the Unity editor. It allows you to configure everything from simple properties like colors and textures to advanced options like blending and depth write.

Detailed Explanation of Scripts

The two classes, **`MMDMaterialCustomInspector_AmplifyShaderEditor`** and **`MMDMaterialCustomInspector_ShaderGraph`** , are custom inspector scripts for MikuMikuDance (MMD) materials in Unity, using the Amplify Shader Editor. It extends the Unity Editor interface, allowing users to manipulate specific properties of MMD materials with ease, offering both default and custom options.

Main Features:

1. Control Variables:

- Stores material names in Japanese and English, plus a memo field for notes.
- The **`showSystemsDefault`** **variable** controls whether to display the custom inspector or the default Unity inspector.
- **`customMMDMaterialData`** is used to store and manage MMD material specific data.

2. `OnGUI` Method:

- Initializes the material inspector (**`materialInspector`**), material properties (**`materialProperties`**), and the current material being edited (**`currentMaterial`**).
- Provides a toggle option to show or hide Unity's default inspector.
- Loads the custom MMD material data if it is not already loaded.
- If **`showSystemsDefault`** is disabled, render the custom inspector, otherwise render the default Unity inspector.
- Automatically saves changes made to MMD material data.

3. `RenderCustomMaterialInspector` method:

- Provides detailed controls for adjusting MMD material properties, including:
- Material names (in Japanese and English).
- Color properties like **`Diffuse`** , **`Specular`** and **`Ambient`** .
- Opacity and reflection settings.
- Advanced rendering options (like dual faces, shadow shader, shadow reception).
- Border (outline) settings, including size and color.
- Texture settings (main, toon and spherical effects like **`SPH`**).
- A memo field for notes or additional information.
- Custom effects adjustments, such as specular intensity, shadow opacity, HDR, among others.
- Advanced options including render queue, GPU instancing, and dual-face global illumination.

Both scripts use the custom utility **`CustomMMDDataUtilityEditor`** to load, save, and render custom MMD material properties and data. This utility is essential for keeping the code organized and reusable, and for making it easier to add new functionality in the future.
There is also a variant of the script called **`MMDTessellationMaterialCustomInspector`** to add **`Edge Length`** , **`Phong Tess Strength`** and **`Extrusion Amount`** which are fusions of Tessellation which is a technique used in shaders to subdivide polygons of a mesh into smaller triangles at runtime. This allows you to add more geometric detail to objects.

Create Prefab From Model

This script is a custom tool for the Unity Editor, designed to create prefabs from selected models. Let's break down each part of the script:

Imports and Initial Settings

```
using UnityEngine;
using UnityEditor;
using System.Collections.Generic;
using System.IO;
```

These lines import the necessary namespaces:
- `**UnityEngine**` : To access the main functionalities of Unity.
- `**UnityEditor**` : To create custom tools within the Unity Editor.
- `**System.Collections.Generic**` : To use generic lists.
- `**System.IO**` : For manipulating file and directory paths.

Main Class

```
public class CreatePrefabFromModel : EditorWindow
```

Defines a class called `**CreatePrefabFromModel**` , which inherits from `**EditorWindow**` , allowing you to create custom windows in the Unity Editor.

Static Method for Creating Prefabs

```
[MenuItem("Assets/MMD Collection/Create Prefabs From Selected Model")]
private static void CreatePrefabsFromModel()
```

This method creates a menu entry in the Unity Editor under "Assets/MMD Collection/Create Prefabs From Selected Model". When this entry is clicked, the `**CreatePrefabsFromModel**` **method** is executed.

Object Selection and Validation

```
Object[] selectedObjects = Selection.objects;
List<GameObject> models = new();

foreach (Object obj in selectedObjects)
{
    if (obj is GameObject selectedObject && PrefabUtility.GetPrefabAssetType(selectedObject) == PrefabAssetType.Model)
    {
        string modelPath = AssetDatabase.GetAssetPath(selectedObject);
        GameObject model = AssetDatabase.LoadAssetAtPath<GameObject>(modelPath);

        if (model != null)
        {
            models.Add(model);
        }
        else
        {
            Debug.LogError($"Failed to load model from path: {modelPath}");
        }
    }
}

if (models.Count == 0)
{
    Debug.LogWarning("Select one or more valid 3D model files to create Prefabs.");
}
```

- `**Selection.objects**` : Gets the selected objects in the Unity Editor.
- `**List<GameObject> models**` : List to store valid models.
- Iterates through the selected objects:

- Checks if the object is a **`GameObject`** and if it is a model (**`PrefabAssetType.Model`**).
- Gets the asset path and loads the model.
- Adds the model to the list of valid models if loaded successfully.
- Display a warning if no valid model is found.

Converting Models to Prefabs

```
foreach (GameObject model in models)
{
    ModelConverter(model);
}
```

`ModelConverter` method .

`ModelConverter` method

```
private static void ModelConverter(GameObject model)
{
    List<GameObject> createdObjects = new();
    SkinnedMeshRenderer[] skinnedMeshRenderers = model.GetComponentsInChildren<SkinnedMeshRenderer>();
    MeshFilter[] meshFilters = model.GetComponentsInChildren<MeshFilter>();

    if (skinnedMeshRenderers.Length == 0 && meshFilters.Length == 0)
    {
        Debug.LogError("The selected model does not contain any SkinnedMeshRenderer or MeshFilter components.");
        return;
    }

    SkinnedMeshRendererConverter(skinnedMeshRenderers, createdObjects);
    MeshFilterConverter(meshFilters, createdObjects);

    if (createdObjects.Count > 0)
    {
        CreatePrefab(model, createdObjects);
    }
}
```

- Creates a list to store created objects.
SkinnedMeshRenderer`** and **`MeshFilter` components from the model.
- If there are no valid components, display an error and return.
- Converts the **`SkinnedMeshRenderer`** and **`MeshFilter`** into separate objects.
- If valid objects are created, call the **`CreatePrefab` method** .

Converting `SkinnedMeshRenderer` and `MeshFilter`

```
private static void SkinnedMeshRendererConverter(SkinnedMeshRenderer[] skinnedMeshRenderer[] skinnedMeshRenderers, List<GameObject> createdObjects)
{
    foreach (SkinnedMeshRenderer skinnedMeshRenderer in skinnedMeshRenderers)        leshRenderers)
    {
        GameObject emptyObject = CreateGameObjectWithMesh(skinnedMeshRenderer.sharedMedMeshRenderer.sharedMesh, skinnedMeshRenderer.sharedMaterials);
        createdObjects.Add(emptyObject);
    }
}

private static void MeshFilterConverter(MeshFilter[] meshFilters, List<GameObject> cre, List<GameObject> createdObjects)
{
    foreach (MeshFilter meshFilter in meshFilters)
    {
        if (meshFilter.TryGetComponent<MeshRenderer>(out var meshRenderer))          hRenderer))
        {
            GameObject emptyObject = CreateGameObjectWithMesh(meshFilter.sharedMesh, eshFilter.sharedMesh, meshRenderer.sharedMaterials);
            createdObjects.Add(emptyObject);
        }
        else
        {
            Debug.LogError("MeshFilter does not have a MeshRenderer.");
        }
    }
}
```

- **`SkinnedMeshRendererConverter` : Converts `SkinnedMeshRenderer`** components into new objects.
- **`MeshFilterConverter`** : Converts **`MeshFilter` components** into new objects, checking if they have a **`MeshRenderer`** .

GameObject Creation with Mesh and Materials

```
private static GameObject CreateGameObjectWithMesh(Mesh mesh, Material[] materials)
{
    GameObject emptyObject = new(mesh.name);
    MeshFilter meshFilter = emptyObject.AddComponent<MeshFilter>();
    MeshRenderer meshRenderer = emptyObject.AddComponent<MeshRenderer>();

    meshFilter.sharedMesh = mesh;
    meshRenderer.sharedMaterials = materials;

    return emptyObject;
}
```

Creates a new **`GameObject`** with **`MeshFilter`** and **`MeshRenderer` components** , assigning it the provided mesh and materials.

Prefab Creation

```
private static void CreatePrefab(GameObject model, List<GameObject> createdObjects)
{
    if (createdObjects == null || createdObjects.Count == 0)
    {
        Debug.LogError("No valid objects created to make a prefab.");
        return;
    }
    ...
```

- Checks if there are valid objects to create the prefab.
- If there is only one object, use it directly; otherwise, create a new **`GameObject`** to group all created objects.
- Gets the model directory path and builds the prefab path.

- Try to save the created object as a prefab.
- Displays log messages for success or error in prefab creation.
- Destroys the created object after saving the prefab to avoid residue in the scene.

Summary

This script automates the process of creating prefabs from selected 3D models in the Unity Editor. It validates the models, converts mesh components into separate objects, and saves these objects as prefabs.

Paste As Child Multiple

This script is a custom editor for Unity that adds the functionality to paste multiple instances of a prefab or selected object as children of selected objects in the hierarchy. I will explain each part of the script in detail:

Imports and Declarations

```
using UnityEngine;
using UnityEditor;
using System.Collections.Generic;
using System;
```

These are the required namespaces:
- `**UnityEngine**` and `**UnityEditor**` are essential for any script that interacts with Unity and its editor.
- `**System.Collections.Generic**` and `**System**` provide basic data structures and functions.

Main Class

```
public class PasteAsChildMultiple : EditorWindow
{
    private GameObject objectToCopy;
    private bool enumerate;
    #pragma warning disable IDE0044
    private List<GameObject> newObjects = new();
    #pragma warning restore IDE0044
```

- `**PasteAsChildMultiple**` inherits from `**EditorWindow**` , allowing you to create a custom window in the Unity editor.
- `**objectToCopy**` stores the object to be copied.
- `**enumerate**` defines whether new objects will be given enumerated names.
- `**newObjects**` maintains a list of newly created objects.

Window Initialization

```
[MenuItem("GameObject/MMD Collection/Paste as Multiple Children")]
private static void Init()
{
    PasteAsChildMultiple window = (PasteAsChildMultiple)GetWindow(typeof(PasteAsChildMultiple));
    window.titleContent = new GUIContent("Paste as Multiple Children");
    window.minSize = new Vector2(350, 200);
    window.maxSize = new Vector2(350, 200);
    window.Show();
}
```

- `**MenuItem**` defines where the new menu option will be displayed in the Unity editor.
- `**Init**` creates and displays the custom window with fixed title and dimensions.

Window Layout

```
private void OnGUI()
{
    GUIStyle boldLargeStyle = new(GUI.skin.label)
    {
        fontSize = 15,
        fontStyle = FontStyle.Bold
    };


    GUILayout.Label("Select Prefab to Copy", boldLargeStyle);
    GUILayout.Space(10f);


    EditorGUILayout.BeginHorizontal();
    GUILayout.Label("Object to Copy:", GUILayout.Width(100f));
    objectToCopy = EditorGUILayout.ObjectField(objectToCopy, typeof(GameObject), true) as GameObject;
    EditorGUILayout.EndHorizontal();
    GUILayout.Space(10f);


    EditorGUILayout.BeginHorizontal();
    GUILayout.Label("Enumerate:", GUILayout.Width(100f));
    enumerate = EditorGUILayout.Toggle(enumerate);
    EditorGUILayout.EndHorizontal();
    GUILayout.Space(35f);


    EditorGUI.BeginDisabledGroup(objectToCopy == null);
    GUILayout.BeginHorizontal();
    GUILayout.FlexibleSpace();
    if (GUILayout.Button("Paste As Child", GUILayout.Width(150), GUILayout.Height(40))
    {
```

- `**OnGUI**` defines the graphical interface of the window.
- Creates a title with bold style.
- Field to select the object to be copied.
- Toggle to define whether new objects will have enumerated names.
- "Paste As Child" button, which calls the `**PasteAsChild**` **method** when pressed.

Method to Paste the Object

```
private void PasteAsChild()
{
    GameObject[] selectedObjects = Selection.gameObjects;

    if (selectedObjects.Length == 0)
    {
        Debug.LogWarning("No objects selected.");
        return;
    }

    if (objectToCopy == null)
    {
        Debug.LogWarning("No object selected to copy.");
        return;
    }
    ...
```

- `**PasteAsChild**` is the main method that deals with pasting objects.
- Checks if there are selected objects and an object to be copied.
- Iterates over the selected objects, instantiating the prefab and setting it as a child of each selected object.
- Sets the name of the new object if `**enumerate**` is enabled.
- Records the undo operation for each new object created.
- Expands the hierarchy to show new children.
- Select the new objects and close the window.

Method for Expanding the Hierarchy

```
private void ExpandHierarchy(GameObject obj)
{
    try
    {
        var type = typeof(EditorWindow).Assembly.GetType("UnityEditor.SceneHierarchyWindow");
        var hierarchyWindow = GetWindow(type);
        var expandMethod = type.GetMethod("SetExpandedRecursive");

        if (hierarchyWindow != null && expandMethod != null)
        {
            expandMethod.Invoke(hierarchyWindow, new object[] { obj.GetInstanceID(), true });
        }
    }
    catch (Exception e)
    {
        Debug.LogError($"Error expanding hierarchy: {e.Message}");
    }
}
```

- `**ExpandHierarchy**` uses reflection to access and invoke methods of `**SceneHierarchyWindow**` , expanding the hierarchy to show the new objects.

Final Considerations

This script provides a useful tool for duplicating and organizing objects in the Unity hierarchy, making the development and prototyping process easier. The interface is simple yet functional, allowing developers to quickly instantiate multiple objects and configure their properties efficiently.

Material Property Cleaner

This script is a Unity editor that cleans up invalid properties of selected materials. I will explain each part in detail.

Imports

```
using UnityEngine;
using UnityEditor;
using System.Collections.Generic;
```

These are the imported namespaces:
- `**UnityEngine**` : The main Unity library.
- `**UnityEditor**` : Provides Unity-specific editor functionality.
- `**System.Collections.Generic**` : Allows the use of generic collections such as `**HashSet**` and `**List**` .

Class Declaration

```
public class MaterialPropertyCleaner : MonoBehaviour
```

`**MaterialPropertyCleaner**` is a public class that inherits from `**MonoBehaviour**` , but in this context, it is primarily used to access editor functionality.

Menu Item and Main Method

```
[MenuItem("Assets/MMD Collection/Clean Invalid Material Properties")]
public static void CleanMaterialProperties()
```

This attribute adds a menu item in the Unity editor. When clicked, it calls the `**CleanMaterialProperties**` **method** .

Confirmation Window

```
if (!EditorUtility.DisplayDialog(
    "Confirm Material Clean",
    "Are you sure you want to clear invalid properties from the selected material?\nThis operation cannot be undone.",
    "Yes",
    "No"))
{
    return;
}
```

Shows a confirmation dialog to the user. If the user clicks "No", the method exits.

Loop for Selected Objects

```
foreach (Object selectedObject in Selection.objects)
{
    if (selectedObject is Material material)
    {
        Shader shader = material.shader;

        if (shader == null)
        {
            Debug.LogError($"The material '{material.name}' does not have a shader.");
            continue;
        }
```

For each object selected in the editor:
- Checks if the object is a `**Material**` .
- Gets the material's shader, and if there is no shader, logs an error and continues to the next object.

Valid Shader Properties

```
var validProperties = new HashSet<string>();

for (int i = 0; i < ShaderUtil.GetPropertyCount(shader); i++)
{
    string propertyName = ShaderUtil.GetPropertyName(shader, i);
    validProperties.Add(propertyName);
}
```

Creates a set of valid properties by iterating through the shader's properties and adding their names to the **`validProperties` set** .

Saved Material Properties

```
var materialSerializedObject = new SerializedObject(material);
var savedProperties = materialSerializedObject.FindProperty("m_SavedProperties");
```

Gets a serialized material object and accesses the saved properties.

Removing Invalid Properties

```
RemoveInvalidProperties(savedProperties.FindPropertyRelative("m_TexEnvs"), validProperties);
RemoveInvalidProperties(savedProperties.FindPropertyRelative("m_Ints"), validProperties);
RemoveInvalidProperties(savedProperties.FindPropertyRelative("m_Floats"), validProperties);
RemoveInvalidProperties(savedProperties.FindPropertyRelative("m_Colors"), validProperties);

materialSerializedObject.ApplyModifiedProperties();
```

`RemoveInvalidProperties` method for different types of properties (**`m_TexEnvs`** , **`m_Ints`** , **`m_Floats`** , **`m_Colors`**) and applies the modifications.

Verification of Non-Material Objects

```
else
{
    Debug.LogWarning($"Selected object '{selectedObject.name}' is not a material.");
}
```

If the selected object is not a material, log a warning.

Invalid Property Removal Method

```
private static void RemoveInvalidProperties(SerializedProperty properties, HashSet<string> validProperties)
{
    for (int i = properties.arraySize - 1; i >= 0; i--)
    {
        var property = properties.GetArrayElementAtIndex(i);
        string propertyName = property.FindPropertyRelative("first").stringValue;

        if (!validProperties.Contains(propertyName))
        {
            properties.DeleteArrayElementAtIndex(i);
        }
    }
}
```

This method removes invalid properties:
- Iterate through the properties from back to front.

- If the property is not in **`validProperties`** , it is removed.

Summary

This script clears invalid properties from selected materials in the Unity Editor, ensuring that only valid properties set by the material's shader are retained. This helps avoid redundant or invalid data in materials, potentially improving performance and organization of Unity projects.

Script Purpose

This script simplifies the conversion of material shaders for use in Unity's Universal Render Pipeline (URP). It's designed for MikuMikuDance (MMD) shaders, enabling a smoother transition of shader properties and ensuring compatibility with URP.

Variables and Methods Overview

1. `ShaderModel` Enum

Defines types of shaders the script can convert to:
- **`Default`**: Basic shader.
- **`Tessellation`**: Shader with finer surface details.
- **`Empty`**: Minimal shader without additional effects.
- **`FourLayers`** & **`EightLayers`**: Shaders supporting multiple outline layers.
- **`NoShadow`** & **`NoShadowAndTessellation`**: Disables shadows, with an option for tessellation.

2. ConvertShader() Method

- This is the main entry point, allowing a user to select materials in the Unity editor to be converted.
- It iterates over each selected material and, based on the shader's name, selects the appropriate conversion by calling **`ChangeShader()`** with specific parameters.

3. ChangeShader() Method

- Adjusts shader properties for the selected material, recording it for potential undo actions.
- Sets basic rendering parameters (**`instancing`**, **`GI`** settings).
- Calls helper methods (**`ApplyStandard`**, **`ApplyEmpty`**, **`ApplyMultiplePass`**) based on the shader type.
- Cleans up incompatible properties using **`CleanMaterialProperties()`**.

4. ApplyStandard() Method

- Configures a material's primary color, transparency, and edge size.
- Adjusts shadow and outline properties based on the provided **`ShaderModel`**.
- Handles tessellation settings for shaders needing finer surface details.

5. ApplyEmpty() Method

- Applies a minimal shader, clearing extra properties and setting transparency.
- Disables shadow casting and adjusts rendering queue for transparent display.

6. ApplyMultiplePass() Method

- Configures shaders that require multi-pass rendering (e.g., multiple outlines).
- Retains outline color and edge size settings for continuity.

7. RenderQueueToTransparent() Method

- Adjusts the material's render queue if transparency is enabled, placing it in the correct range for transparent rendering.

8. CleanMaterialProperties() Method

- Removes properties from the material that aren't needed by the new shader.
- Uses **`RemoveInvalidProperties`** and **`CleanInvalidKeywords`** for precise cleanup.

9. RemoveInvalidProperties() Method

- Removes invalid properties from the material based on those supported by the new shader, which ensures clean transitions between shaders.

10. CleanInvalidKeywords() Method

- Disables any shader keywords not supported by the new shader, ensuring compatibility.

Usage

1. Selecting Materials in Unity Editor

Select materials you want to convert in the Unity editor, then run **`ConvertShader()`** from the menu under **`Assets > MMD Collection > Convert Material Shader (MMD4Mecanim)`**.

2. Shader Conversion

The script will detect each material's shader name, select an appropriate URP-compatible shader, and apply settings based on the shader type.

3. Undo Capability

Since changes are recorded, users can undo the shader conversion in the editor.

4. Rendering Optimization

Through the helper functions, the script ensures only necessary properties remain in the material, which optimizes rendering performance.

This setup automates URP shader conversion, ensuring MMD shaders remain visually consistent in URP projects.

Manage Objects

This script is a custom editor for Unity. The script aims to manage the visibility of a list of GameObjects in the Unity editor. I will divide the explanation into two parts: the main script (**`ManageObjects`**) and the custom editor (**`ManageObjectsEditor`**).

Main Script: `ManageObjects`

`ManageObjects` script is a MonoBehaviour component that can be added to a GameObject in the scene. It provides functionality for managing the visibility of a list of GameObjects.

1. Definitions and Variables

- **`public ManageObjectsList[] manageObjects`** : An array of **`ManageObjectsList`** , which is a serializable class (defined at the end of the script). Each item in the array contains a reference to a GameObject and its visibility state.

- **`[HideInInspector] public bool state`** : Variable that controls the global visibility state for all objects in the list. If **`true`** , all objects are visible; if **`false`** , all are invisible.

- **`[HideInInspector] public bool hide`** : Variable that controls whether objects will be hidden in the inspector.

- **`[HideInInspector] public bool hideInspector`** : Variable that controls whether the component's default inspector will be hidden.

2. Methods

- **`public void Toggle(int i)`** : Toggles the visibility of the GameObject at position **`i` in the `manageObjects`** array . The method registers the action to enable undo and changes the activation state of the GameObject.

- **`public void ToggleAll()`** : Toggles the visibility of all GameObjects in the list, according to the global state **`state`** . Also registers the undo action and applies the state to all GameObjects.

- **`public void RemoveItem(int i)`** : Removes the GameObject at position **`i` from the `manageObjects`** array , recording the action to undo.

- ' **Conditional Destruction'** : Outside of development builds, the ' **ManageObjects** ' component is automatically destroyed in the Start() method to optimize performance in production builds.

Custom Editor: `ManageObjectsEditor`

` **ManageObjectsEditor`** is a class that extends **`Editor`** and is responsible for creating a custom interface in the Unity inspector for the **`ManageObjects` component** .

1. `OnInspectorGUI()` method

This method is called to draw the custom inspector interface:

- **"Toggle All" button** : Toggles the visibility of all objects in the list and displays the global state.

- **`DrawObjectBox(script)`** : Draws an area where you can drag and drop GameObjects to add them to the list. The method handles drag and drop events and adds the objects to the array if they are not already present.

- **`EditorGUILayout.Toggle()`** : Adds toggles to hide objects and the default inspector.

- **`DrawDisplayButtons(script)`** : Draws buttons to toggle visibility and remove each GameObject from the list, if the objects are not hidden.

- **`DrawDefaultInspector()`** : Draws the component's default inspector if it is not hidden.

2. Auxiliary Methods

- **`DrawObjectBox(ManageObjects script)`** : Draws the drag and drop area for adding new GameObjects to the list. Checks for drag and drop events and adds the GameObjects to the **`manageObjects` array** if they are not already present.

- **`IsObjectInList(ManageObjects script, GameObject gameObject)`** : Checks if a GameObject is already in the **`manageObjects` list** .

- **` DrawDisplayButtons(ManageObjects script)`** : Draws buttons to toggle visibility and remove objects from the list, and displays warning messages if there are no objects.

Class `ManageObjectsList`

- **`public GameObject gameObjects`** : Reference to the GameObject that will be managed.

- **`public bool objectState`** : Visibility state of the GameObject.

Summary

The script provides a handy way to manage the visibility of multiple GameObjects from a custom editor in Unity. The **`ManageObjects` component** manages the list of GameObjects and their visibility state, while the **`ManageObjectsEditor`** provides a user interface in the inspector to add, remove, and toggle the visibility of objects. In production builds, the component is automatically removed to avoid performance impact.

Find Missing Scripts

General Description

The goal of this script is to find all GameObjects in the scene that have missing scripts and select them in the Unity editor. It adds a menu item to make it easier to perform this check.

Script Structure

1. Namespace and Library Declarations

```
using UnityEngine;
using UnityEditor;
using System.Collections.Generic;
using System.Linq;
```

- `**UnityEngine**` : Contains the Unity API for manipulating game objects.
- `**UnityEditor**` : Provides classes and functions for creating custom tools in the Unity Editor.
- `**System.Collections.Generic**` : Includes generic collection classes like `**List**` .
- `**System.Linq**` : Provides query functions for collections, such as `**Any**` .

2. Class `FindMissingScripts`

```
public class FindMissingScripts : MonoBehaviour
{
```

- ` **public class FindMissingScripts : MonoBehaviour**` : Defines a public class that inherits from `**MonoBehaviour**` . This class will be attached to a GameObject as a component.

3. `FindAllMissingScripts` Method

```
[MenuItem("GameObject/MMD Collection/Find Missing Scripts")]
private static void FindAllMissingScripts()
```

- `**[MenuItem("GameObject/MMD Collection/Find Missing Scripts")]**` : Adds a menu item in the Unity Editor under `**GameObject -> MMD Collection -> Find Missing Scripts**` . This attribute causes the `**FindAllMissingScripts` method** to be executed when the menu item is selected.
- `**private static void FindAllMissingScripts()**` : Defines a private static method that finds all GameObjects with missing scripts.

4. List of GameObjects with Missing Scripts

```
List<GameObject> objectsWithMissingScripts = new();
```

- `**List<GameObject> objectsWithMissingScripts = new();**` : Creates a list to store the GameObjects that have missing scripts.

5. Find All GameObjects

```
GameObject[] allObjects = Resources.FindObjectsOfTypeAll<GameObject>();
```

- ` **GameObject[] allObjects = Resources.FindObjectsOfTypeAll<GameObject>();**` : Gets all GameObjects in the project, including those that are not loaded in the current scene.

6. Iterate Over All GameObjects

```
foreach (GameObject obj in allObjects)
{
    if (!obj.scene.IsValid() || !obj.scene.isLoaded || (obj.hideFlags & (HideFlags.NotEditable | HideFlags.HideAndDontSave)) != 0)
    {
        continue;
    }

    Component[] components = obj.GetComponents<Component>();
    if (components == null) continue;

    if (components.Any(component => component == null))
    {
        objectsWithMissingScripts.Add(obj);
    }
}
```

- `**foreach (GameObject obj in allObjects)**` : Iterates over all GameObjects found.
- `**if (!obj.scene.IsValid() || !obj.scene.isLoaded || (obj.hideFlags & (HideFlags.NotEditable | HideFlags.HideAndDontSave)) != 0)**` : Checks if the object belongs to a valid scene, if the scene is loaded, and if the object is not marked as uneditable or unsaveable. If any of these conditions are true, the object is ignored (`**continue**`).
- `**Component[] components = obj.GetComponents<Component>();**` : Gets all components attached to the current GameObject.
- `**if (components == null) continue;**` : Skip to the next object if there are no components (which shouldn't happen, but is a safety check).
- `**if (components.Any(component => component == null))**` : Checks if any of the components is null (indicating a missing script).
- `**objectsWithMissingScripts.Add(obj);**` : Adds the GameObject to the list if it has any missing scripts.

7. Logging and Selecting Objects

```
Debug.Log($"Found {objectsWithMissingScripts.Count} objects with missing scripts.");
Selection.objects = objectsWithMissingScripts.ToArray();
```

- `**Debug.Log($"Found {objectsWithMissingScripts.Count} objects with missing scripts.");**` : Displays a message in the Unity console with the number of objects found with missing scripts.
- `**Selection.objects = objectsWithMissingScripts.ToArray();**` : Selects objects with missing scripts in the Unity Editor, making them easier to find and fix.

Conclusion

This script is useful for developers who need to ensure that there are no missing scripts on GameObjects in the scene, which could cause errors or unexpected behavior. By adding a menu item in the editor, it makes the checking process quick and accessible.

Shader Keyword Checker

This script was created to check global shader keywords on selected materials in the Unity Editor. It adds a new option in the Unity Editor Asset context menu, allowing users to trigger the check on selected materials.

Variables and methods

1. **MenuItem Attribute**
This attribute is used to create a new item in the Unity Editor menu. The menu path is **`"Assets/MMD Collection/Check Shader Keywords"`** , and it links to the **`CheckShaderKeywords` method** . When selected in the menu, the method is executed.

2. CheckShaderKeywords Method
This is the main method that is triggered when the user selects the menu item. Its main purpose is to iterate through all selected objects in the Unity Editor and check if each one is a material.

- **Selection.objects** : This is a Unity property that contains the list of objects selected in the editor. The method uses it to access these objects.

- **Material material** : The script checks if each selected object is a **`Material`** . If it is, the **`Shader`** associated with the material is retrieved.

- **shader** : This variable represents the shader associated with the material. If no shader is assigned to the material, an error message will be logged.

`CheckGlobalShaderKeywords` method to inspect the global shader keywords.

3. CheckGlobalShaderKeywords Method
This method takes a **`Shader` object** and checks its global shader keywords.

- **ShaderUtil** : Unity has an internal class called **`ShaderUtil`** that contains utilities for shader-related operations. However, this class is not publicly accessible.

- **Reflection (GetMethod)** : Since **`ShaderUtil`** is internal, the script uses reflection to access its hidden **`GetShaderGlobalKeywords` method** . Reflection allows you to access private or internal methods and properties in C#.

- **getKeywordsMethod** : This variable stores the result of calling the **`GetMethod` function** , which attempts to find the **`GetShaderGlobalKeywords` method** using its name and special binding flags (**`Static`** and **`NonPublic`**).

- If the method is found, it will be invoked to retrieve the global shader keywords, which are logged to the Unity console. If the method is not found, an error will be logged indicating the failure.

Use

1. Select materials in the Unity Editor.
2. Right-click on the selected assets and navigate to the **`"MMD Collection"` option under the `"Assets"`** menu .
3. Choose **`"Check Shader Keywords"`** .
4. The script will check if the selected assets are materials and retrieve the shader's global keywords. These keywords will be logged to the Unity console.

MMD Bone Renderer

Overview (Purpose)

This script is a utility component made to help developers visualize bones of a character or rig in the Unity Editor. It works while you're editing the scene (not just in play mode) and allows you to see lines between bones, the end tips of bone chains, and set specific display options for each bone.
It's especially useful when dealing with complex 3D characters like those used in MikuMikuDance (MMD) models, where understanding bone structure visually can be a big help.

Main Variables

- **drawBones**: A switch that controls whether bones should be drawn in the scene or not.
- **jointSize**: A number that controls how big the joint points look when visualized in the scene.
- **boneRendererList**: A list of groups, where each group has:
  - a specific **color** for the bones.
  - a list of **individual bones** and their settings (like if they're rotatable, movable, or visible).

Data Classes

- **BoneRendererData**:
  - This defines a group of bones with a shared color.
  - Each group contains multiple bones and their individual settings.
- **BonesTransform**:
  - This represents a single bone and its visual behavior:
    - Should it rotate?
    - Should it move?
    - Should it be visible in the scene?
- **TransformPair**:
  - Just a pair of bones: one is the parent, and the other is the child.
  - Used for drawing lines between connected bones.

Main Features and Methods

OnEnable / OnDisable

- When this component is activated or deactivated, it will:
  - Rebuild the list of bones.
  - Notify other parts of the editor that this renderer was added or removed (through static events).

AltoGetBones

- This is a context menu option you can trigger manually.
- It scans **all children of the current object** and adds them to a new bone group.
- This is a fast way to auto-fill the bone list, though a warning makes it clear this might not always be the best way to use the tool.

Transforms (Property)

- This allows you to **get** a list of all the bone transforms currently stored in the renderer.
- You can also **set** a new list of bone transforms, which will replace the existing groups and rebuild the internal data.

Reset / ClearBones

- These methods remove all existing bone references and clear the visual data.
- It's like hitting "reset" on your visual bone setup.

Invalidate / ExtractBones

- These methods refresh or rebuild the visual structure based on the bones currently stored.
- The script checks:
  - Which bones are connected.
  - Which bones have no children and are considered tips.
- It stores this information in arrays to be used for drawing.
- It also ignores bones that are hidden in the editor or not visible in the current layer view.

Events
- The script supports events that trigger when a bone renderer is added or removed.
- Other systems (like custom editors or managers) can listen for these events and update accordingly.

Editor-Only Execution
Everything is wrapped to run **only in the Unity Editor**:
- The visualization, bone selection, and extraction don't affect gameplay.
- They are meant only to help while designing or editing the scene.

How to Use It
1. **Attach it to a GameObject**: You can do this through the Unity editor's "Add Component" menu, under "MMD Collection/Bone Renderer".
2. **Manually add bones**: Through the Inspector, assign bones to the list, grouped by color.
3. **Use AltoGetBones**: Optionally, right-click the component and choose "Alto Get Bones" to auto-fill the list with all child bones.
4. **Customize each bone**: Toggle movement, rotation, and visibility per bone.
5. **View in Scene View**: If enabled, bones will appear as lines with joints and colored segments, based on your settings.

MMD Bone Renderer Inspector

Overview (Purpose)
This script is a **custom editor** for the MMDBoneRenderer component. It replaces Unity's default inspector UI with a custom one that:
- Allows **live updates** when values change.
- Handles **Undo** and **Redo** operations correctly.
- Ensures the internal bone data stays up to date without the developer having to press a button manually.

This is part of what makes the visual bone system feel responsive and user-friendly in the Unity Editor.

Key Components and What They Do
- **Custom Editor for MMDBoneRenderer**:
  - This means whenever you select a GameObject that has the MMDBoneRenderer component, Unity will use this custom interface instead of its built-in one.
- **Can Edit Multiple Objects**:
  - This lets you select multiple GameObjects with MMDBoneRenderer components and edit them at the same time.

Main Method: OnInspectorGUI
This method controls what happens when Unity draws the inspector in the Editor window. Here's a breakdown of the process it follows:
1. **Update Serialized Data**:
   - It grabs the latest info from the selected MMDBoneRenderer component(s) and syncs it with the inspector.
2. **Detect Changes**:
   - It starts tracking if the user changes anything in the Inspector window — such as toggling visibility, adjusting joint size, or modifying bone lists.
3. **Draw the Inspector UI**:
   - It uses Unity's built-in method to draw all fields as usual, so the user still sees and edits the expected controls.
4. **Check If Anything Changed**:
   - After drawing, it checks whether any property actually changed.
5. **Handle Undo/Redo Events**:
   - It also checks if the user pressed **Undo** or **Redo**, because those can change properties without any direct user input in the inspector.
6. **Apply the Changes**:
   - If any changes were made, it commits them to the component.

7. **Trigger a Bone Update**:
    - If anything did change (either by edit or undo), it loops through every selected target, finds the associated MMDBoneRenderer, and tells it to **recalculate bones** using its internal method.
    - This ensures the visuals and internal data always reflect the latest state in the inspector.

🧪 Usage

This script runs **automatically**. You don't have to call or attach it manually. Here's what happens when it's in your project:

1. **Select an object with MMDBoneRenderer**: You'll see the usual inspector fields (like drawBones, jointSize, and the bone list).
2. **Make changes**: Adjust joint size, change visibility, add bones, etc.
3. **Live update**: As soon as you change something, the bone renderer updates internally, so your changes show up in the scene view without delay.
4. **Undo or Redo**: Press Ctrl+Z or Ctrl+Y (or use the menu) to undo or redo a change. The system updates the bones automatically without breaking the visual state.

💡 Summary

This script doesn't add flashy new controls — it enhances the usability and stability of the MMDBoneRenderer component by:

- Ensuring inspector edits are tracked properly.
- Reacting to Undo/Redo like a pro.
- Keeping the visual bone data fresh at all times.

It's a behind-the-scenes quality-of-life booster that makes the editor experience smoother and more predictable for developers and artists.

MMD Bone Renderer Utils

This script is a Unity Editor utility for rendering bone structures, specifically for MikuMikuDance (MMD) models, using GPU instancing to improve performance. It's designed to visualize bone hierarchies in the scene view and support interactions like selection and transformation. Here's how it works, broken down by section:

🔧 Core Class: MMDBoneRendererUtils

Marked with [InitializeOnLoad], meaning it runs its static constructor when Unity loads the editor.

📦 Inner Class: BatchRenderer

Handles batched rendering of many bone meshes using GPU instancing.

- **Constants:**
    - kMaxDrawMeshInstanceCount: Maximum number of mesh instances per draw call.
    - SubMeshType: Enum distinguishing solid faces and wireframe for bones.
- **Fields:**
    - mesh: The mesh to be drawn (e.g., a pyramid for a bone).
    - material: The shared material for drawing.
    - m_Matrices: Transform matrices for each instance.
    - m_Colors: Base color per instance.
    - m_Highlights: Highlight color per instance (hover/selection).
- **Key Methods:**
    - AddInstance(...): Queues a mesh for rendering with transform and colors.
    - Clear(): Resets the instance queues.
    - Render(): Draws all queued instances using GPU instancing and command buffers.

📋 Global Fields

- s_BoneRendererComponents: Active MMDBoneRenderer instances in the scene.
- s_PyramidMeshRenderer: A lazily-created instance of BatchRenderer using a custom pyramid mesh.
- s_Material: A shared material loaded via EditorGUIUtility.LoadRequired(...).
- s_ButtonHash: Unique ID for GUI controls per bone.

- s_VisibleLayersCache: Stores visible layers to detect changes.

🏗️ **Static Constructor**
Sets up event hooks:
- Adds/removes renderers.
- Reacts to visibility changes and hierarchy modifications.
- Hooks into SceneView.duringSceneGui to draw bones.

📦 **Mesh and Material Setup**
- **Material Property (Material)**: Loads and caches a shader used for bone rendering with GPU instancing.
- **Pyramid Mesh (PyramidMeshRenderer)**: Creates a custom mesh shaped like a pyramid to represent a bone.
  - **Faces (SubMesh 0)**: Triangles forming the pyramid.
  - **Wire (SubMesh 1)**: Lines outlining the mesh for wireframe.

🧮 **ComputeBoneMatrix(...)**
Calculates the transform matrix to draw a bone between two 3D points, aligning a pyramid in space to visually represent it.

🦴 **DrawSkeletons(SceneView)**
The heart of the script. Runs every frame in the scene view.
- Skips invisible or irrelevant objects (like those in a different prefab stage).
- Iterates over all visible bones:
  - Draws bone lines between parent/child.
  - Draws "tips" for bones without children.
  - Renders handles (cubes/spheres) for interaction (select/move/rotate).
- Calls PyramidMeshRenderer.Render() to draw all batched instances.

🔎 **AreBonesInSameGroup(...)**
Checks if two bones belong to the same BoneRendererData group — used to decide whether a bone should be rendered.

🖱️ **DoBoneRender(...)**
Handles all interactions for a single bone:
- Selection with mouse.
- Sets the correct Unity tool (move/rotate/transform).
- Renders bone using PyramidMeshRenderer.AddInstance(...).

➕/➖ **Add/Remove Bone Renderer**
- OnAddBoneRenderer(...) and OnRemoveBoneRenderer(...): Maintain the list of active renderers.

👀 **Visibility and Hierarchy Changes**
- OnVisibilityChanged() and OnHierarchyChanged(): Mark renderers as invalidated and refresh the scene view.

Bones Transform Drawer

This script defines a **custom property drawer** in the Unity Editor for a class called BonesTransform. Its purpose is to improve how Unity displays BonesTransform objects in the Inspector. Instead of showing a plain foldout with subfields, this script lays them out cleanly and compactly for better usability when working with many bones in MMD models.
Let's go through it piece by piece:

🔧 Class Declaration: BonesTransformDrawer

This class is tagged with [CustomPropertyDrawer(typeof(BonesTransform))], meaning:

- Unity will automatically use this drawer whenever it encounters a BonesTransform field in the Inspector.
- It's meant to visually customize how the data in BonesTransform is displayed.

It inherits from PropertyDrawer, which gives access to two key methods:

- OnGUI(...): Renders the custom GUI.
- GetPropertyHeight(...): Calculates how much vertical space the property takes.

🧩 Method: OnGUI(position, property, label)

This method is called every time Unity needs to draw a BonesTransform field in the Inspector.

➕ EditorGUI.BeginProperty(...) and EndProperty(...)

These functions wrap the drawing process, enabling prefab overrides and undo tracking for the fields.

📐 lineHeight and spacing

These store values for UI layout:

- lineHeight: Standard line height for one property line.
- spacing: Vertical space between lines.

🏷️ Label Row

'Rect labelRect = new(position.x, position.y, position.width, lineHeight);'
'EditorGUI.LabelField(labelRect, label);'

- Draws a label for the whole property (optional but improves clarity).

🔗 Bone Field

'Rect boneRect = new(...);'
'EditorGUI.PropertyField(boneRect, property.FindPropertyRelative("bone"), new GUIContent("Bone"));'

- Shows the reference to a Transform that represents the bone.
- It's a reference field, allowing users to drag/drop a transform.

🎛️ Toggle Row

This section handles three boolean flags:

- rotate: Whether this bone should rotate.
- move: Whether this bone should move.
- visible: Whether this bone should be shown.

'Rect toggleRow = new(...);'

- Calculates a horizontal row for the toggles.
- Splits the row into 3 equally-sized toggle buttons using toggleWidth.

Before drawing them:

- It reduces EditorGUIUtility.labelWidth to 50, so the label names don't waste too much space.

Each toggle is drawn using EditorGUI.PropertyField, and the labels are custom-set as "Rotate", "Move", "Visible".
After drawing, it restores the label width to its previous value.

📏 **Method: GetPropertyHeight(property, label)**

This method tells Unity how tall the whole drawer should be.

- It uses:
  - 3 lines of height: one for the label, one for the bone field, one for the toggles.
  - 2 gaps of vertical spacing between the 3 lines.

So the return value is:

'lineHeight * 3 + spacing * 2'

This ensures everything fits neatly.

🧠 **Summary of Usage**

When a developer adds a BonesTransform field to a MonoBehaviour or ScriptableObject, this drawer:

- Displays the field as a small custom box in the Inspector.
- Shows the bone reference and the three flags (rotate, move, visible) in a tidy and intuitive layout.
- Makes editing bone-related data much easier during development and animation setup.