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

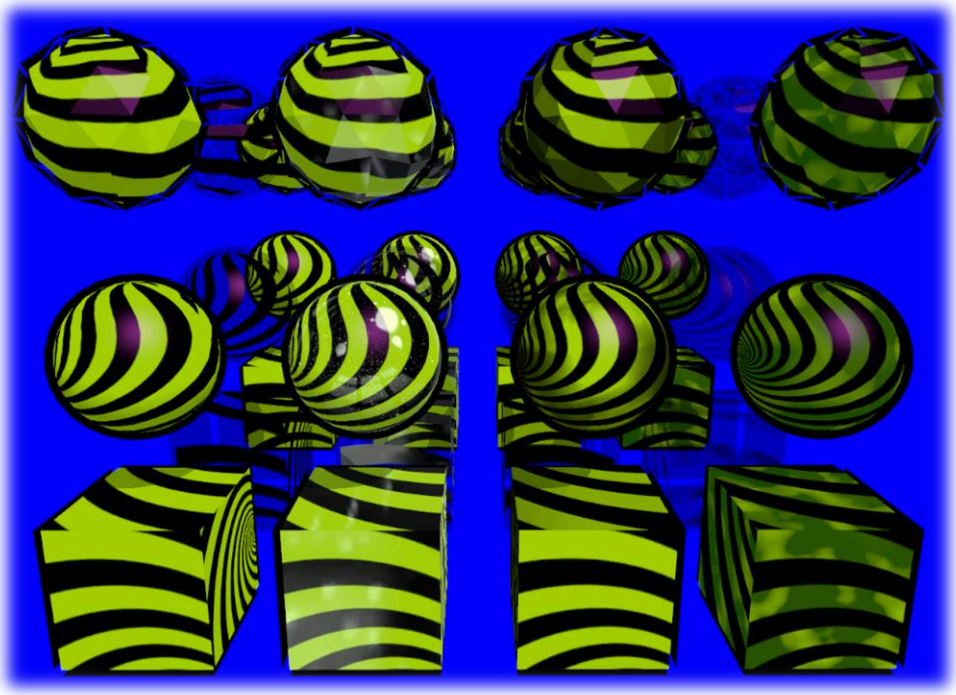
This package is compatible with:

- Unity 6000.2.6f2

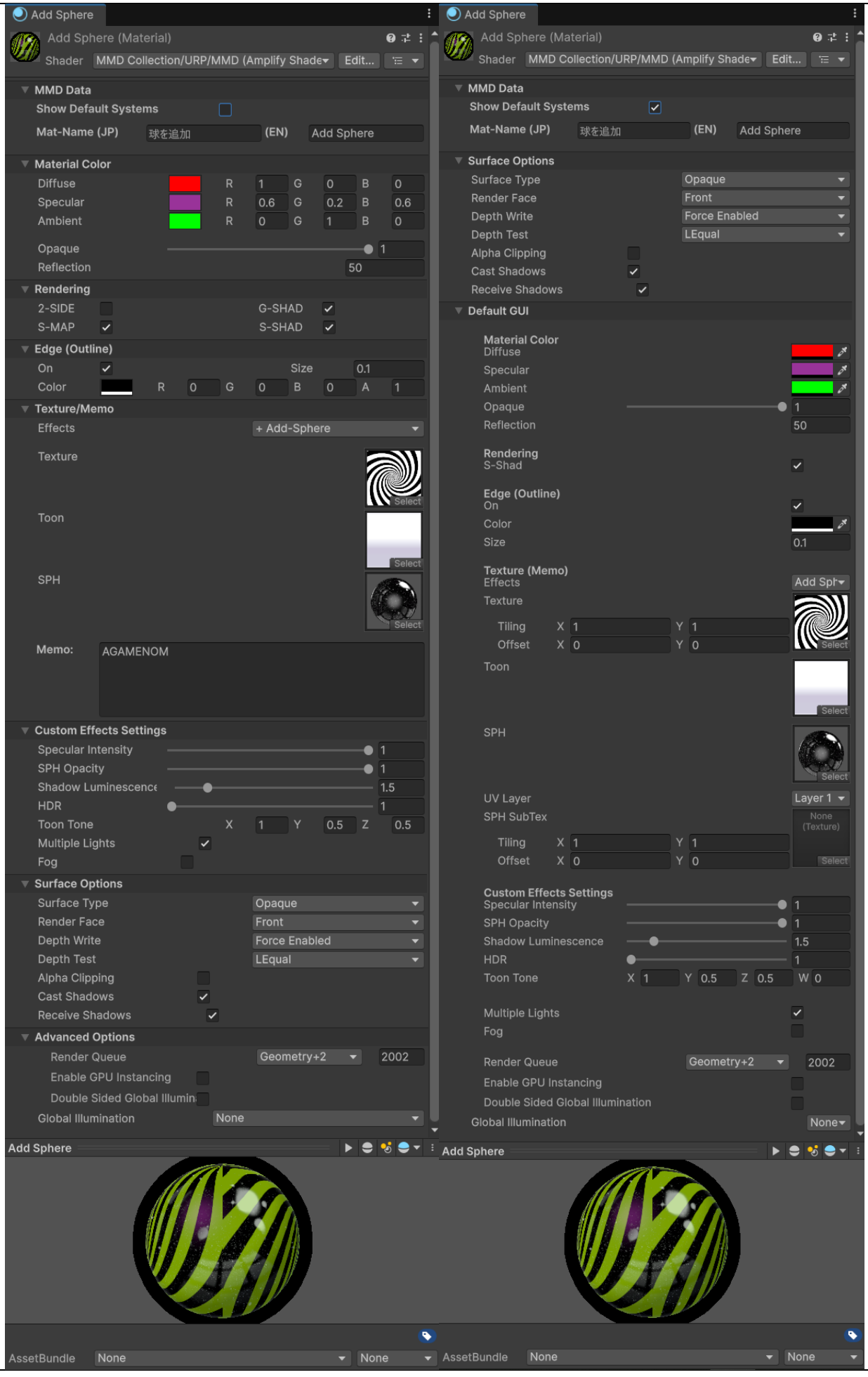
- Amplify Shader Editor v 1.9.9.8

- MMD4Mecanim 2020-01-05

Explicações do Shader



MMD Shader


Add Sphere
Add Sphere (Material)
Shader
MMD Collection/URP/MMD (Amplify Shade
Edit...
MMD Data
Show Default Systems
Mat-Name (JP)
球を追加
(EN)
Add Sphere
Material Color
Diffuse
Specular
Ambient
Opaque
Reflection
Rendering
2-SIDE
G-SHAD
S-MAP
S-SHAD
Edge (Outline)
On
Size
Color
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
Force Enabled
Depth Test
LEqual
Alpha Clipping
Cast Shadows
Receive Shadows
Advanced Options
Render Queue
Geometry+2
2002
Enable GPU Instancing
Double Sided Global Illumination
Global Illumination
None
Default GUI
Material Color
Rendering
S-Shad
Edge (Outline)
Texture (Memo)
Add Sph
Tiling
Offset
UV Layer
Layer 1
SPH SubTex
None (Texture)
Select
AssetBundle

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

Copy Rotation

Overview – What this component does
This Unity component is designed to **synchronize rotations between two sets of bones**.
Every frame (specifically at the end of the animation update cycle), it:

- Reads the **world rotation** of each bone in a source list.
- Applies that same rotation to the corresponding bone in a target list.

It is useful for:

- Animation retargeting.
- Bone mirroring.
- Driving one rig from another at runtime.
- Keeping multiple skeletons visually synchronized.

This version is intentionally **simple and deterministic**:

- No rotation limits.
- No smoothing or interpolation.
- No correction or clamping.
- Just a direct copy.

Inspector Fields (Configuration in the Editor)
These are the values you set in the Unity Inspector.

isCopyEnabled
Purpose
Controls whether the rotation copy process is active or not.
How it works

- When enabled, the component copies rotations every frame.
- When disabled, the component does nothing and exits immediately.

Why it exists

- Allows runtime control without removing the component.
- Useful for enabling/disabling synchronization during gameplay, cutscenes, or transitions.

copyBones
Purpose
Holds the list of **source bones**.
How it works

- Each element represents a bone whose rotation will be read.
- The order of this list is extremely important.

Important rule
The bone at index 0 in this list will copy its rotation to index 0 in the target list, index 1 to index 1, and so on.
Typical usage

- A fully animated skeleton.
- A control rig.
- A reference armature.

pasteBones
Purpose
Holds the list of **target bones**.
How it works

- Each element receives the rotation from its corresponding source bone.
- Must match the source list in length and order.

Typical usage

- A secondary model.
- A mirrored skeleton.
- A physics or proxy rig.
- A runtime-driven character.

Public Properties (Runtime Control)
These expose the internal settings so they can be controlled by other systems.

IsCopyEnabled
What it does
Allows other scripts or systems to:
- Enable rotation copying.
- Disable rotation copying.

Why this matters
- You can toggle synchronization dynamically.
- No need to touch the Inspector or remove the component.

CopyBones
What it does
Allows changing the source bone list at runtime.
Why this matters
- You can swap rigs.
- You can rebind bones dynamically.
- Useful for character swapping or modular characters.

PasteBones
What it does
Allows changing the target bone list at runtime.
Why this matters
- Supports dynamic targets.
- Enables re-targeting on the fly.

Execution Timing – Why LateUpdate Is Used
The component runs during **LateUpdate**, which happens:
- After animations are evaluated.
- After physics adjustments.
- After most transform changes.

Why this is important
- Ensures the source bones already have their final animation rotation.
- Prevents conflicts with Animator updates.
- Makes the copy result stable and predictable.

This is especially critical for animation synchronization tasks.

Internal Logic – Step-by-Step Behavior
Each frame, the component performs the following checks and actions:

1. Copy process enabled check
If rotation copying is disabled:
- The method stops immediately.
- Nothing is calculated or modified.

This avoids unnecessary processing.

2. Bone list validation
The component checks:
- Whether the source list exists.
- Whether the target list exists.

If either is missing:
- A warning is shown.
- No rotation is copied.

This prevents runtime crashes.

3. Length consistency check
The component verifies:
- Both bone lists have the same number of elements.

If they don't match:
- An error is logged.
- The operation stops.

This guarantees one-to-one bone mapping.

4. Per-bone safety check
For each bone pair:
- If either the source or target bone is missing:
  - A warning is shown.
  - That pair is skipped.
  - The rest continue normally.

This ensures partial setups do not break the entire system.

5. Rotation copying
For valid bone pairs:
- The **world rotation** of the source bone is read.
- That exact rotation is applied to the target bone.

There is:
- No conversion to local space.
- No blending.
- No filtering.

The result is a perfect rotational match in world space.

Usage Guide – How to Use This Component

Basic Setup
1. Add the component to any GameObject.
2. Assign the **source bones** to the copy list.
3. Assign the **target bones** to the paste list.
4. Ensure both lists:
   - Have the same length.
   - Are ordered correctly.
5. Keep the copy enabled.

Common Use Cases
- Drive a secondary character from a primary animated rig.
- Mirror bone motion to a duplicate skeleton.
- Synchronize bones between LOD models.
- Apply animation data to a physics-driven rig.
- Retarget rotations without using Unity's Animator retargeting.

When NOT to use it
- If you need rotation limits.
- If you need smoothing or blending.
- If you need local-space rotation copying.
- If bone hierarchies differ significantly.

This component is intentionally low-level and direct.

Summary
In short:
- This component copies **world rotations** from one bone list to another.
- It runs safely and consistently in LateUpdate.
- It is fully controllable at runtime.
- It prioritizes simplicity, performance, and predictability.

Free Fly Camera

Overview – What this camera system does

This component turns a camera into a **free-fly navigation tool**, similar to:

- Unity's Scene View camera
- Debug or inspection cameras
- First-person noclip cameras
- Cinematic scouting cameras

It allows you to:

- Move freely in all directions.
- Rotate the view using the mouse.
- Zoom forward and backward with the mouse wheel.
- Switch between slow precision movement and fast traversal.
- Customize all controls directly in the Inspector.

It is designed mainly for **editor tools, debugging, scene exploration, and runtime inspection**, but it can also be used in builds if desired.

Camera Settings – Movement, Rotation, and Zoom

These settings control **how the camera feels**.

movementSpeed

What it controls

The normal movement speed of the camera.

When it's used

- Active when moving without holding the fast-movement key.

Use case

- Precise navigation.
- Close inspection of objects.
- Cinematic framing.

fastMovementSpeed

What it controls

The movement speed when fast mode is active.

When it's used

- Active while holding the run key.

Use case

- Traversing large scenes quickly.
- Jumping across environments without waiting.

sensitivity

What it controls

How responsive the camera rotation is to mouse movement.

When it's used

- Only active while look mode is enabled.

Effect

- Higher values = faster rotation.
- Lower values = smoother, slower rotation.

zoomSensitivity

What it controls

How fast the camera moves forward or backward when using the mouse scroll wheel.

When it's used

- Normal zooming, without fast mode.

fastZoomSensitivity

What it controls

Zoom speed while fast mode is active.

Use case

- Quickly pushing the camera through large distances.
- Rapid scene traversal using the scroll wheel.

Key Settings – Movement and Control Keys
These settings define **which keys do what**.

left / right
What they do
Move the camera sideways relative to where it is facing.
Important detail
- Movement is based on the camera's orientation, not the world.

front / back
What they do
Move the camera forward or backward in the direction it is facing.
Common use
- Standard first-person navigation.

up / down (local)
What they do
Move the camera up or down relative to its own rotation.
Difference from global movement
- If the camera is tilted, "up" follows the camera's tilt.

upGlobal / downGlobal
What they do
Move the camera straight up or down using the world's vertical axis.
Why this exists
- Useful when the camera is rotated and you still want clean vertical movement.
- Ideal for architectural inspection or level design.

observe
What it does
Activates **look mode**.
While held
- Mouse movement rotates the camera.
- Cursor is hidden and locked.

When released
- Rotation stops.
- Cursor returns to normal.

run
What it does
Enables fast movement and fast zoom.
Effect
- Movement speed increases.
- Zoom speed increases.

Axis Settings – Mouse Input Configuration
These settings let you adapt the camera to **different input setups**.

zoom
What it represents
The input axis used for zooming forward and backward.
Typical source
- Mouse scroll wheel.

axisX
What it represents

Horizontal mouse movement.
Used for
- Turning the camera left and right.

axisY
What it represents
Vertical mouse movement.
Used for
- Looking up and down.

Runtime State – Internal Behavior

looking
What it represents
Whether the camera is currently in look mode.
When true
- Mouse movement rotates the camera.
- Cursor is locked and hidden.

When false
- Camera rotation stops.
- Cursor behaves normally.

Frame Update – What Happens Every Frame
Every frame, the camera performs the following steps:

1. Determine movement mode
- Checks whether the fast-movement key is being held.
- Chooses normal or fast speeds accordingly.

2. Calculate movement direction
Based on which movement keys are pressed, the camera builds a combined movement direction that may include:
- Sideways motion.
- Forward or backward motion.
- Local vertical movement.
- Global vertical movement.

Multiple keys can be combined at once.

3. Apply movement
- The camera position is updated smoothly.
- Movement is frame-rate independent.
- Direction and speed are applied consistently.

4. Handle look mode rotation
If look mode is active:
- Horizontal mouse movement rotates the camera left and right.
- Vertical mouse movement rotates the camera up and down.
- Roll rotation is prevented to keep the camera stable.

5. Handle zoom input
If the mouse scroll wheel is used:
- The camera moves forward or backward.
- Speed depends on whether fast mode is active.

6. Handle look mode activation
- Pressing the observe key starts look mode.
- Releasing it stops look mode.

This allows quick and temporary camera control without toggles.

Component Disable Safety
When the component is disabled:
- Look mode is automatically turned off.
- Cursor visibility and locking are restored.

This prevents the cursor from getting stuck or hidden.

Look Mode Control Methods

StartLooking
What it does
- Enables look mode.
- Hides the cursor.
- Locks the cursor to the center of the screen.

Purpose
- Allows unrestricted mouse rotation.

StopLooking
What it does
- Disables look mode.
- Shows the cursor.
- Unlocks the cursor.

Purpose
- Restores normal user interaction.

Usage Guide – How to Use This Camera

Typical Setup
1. Add the component to a camera object.
2. Configure speeds and sensitivity.
3. Assign preferred keys and mouse axes.
4. Press Play.
5. Hold the observe key to look around.
6. Use movement keys to navigate.
7. Hold the run key for fast movement.

Ideal Use Cases
- Scene exploration tools.
- Cinematic camera scouting.
- Debugging spatial issues.
- Level design inspection.
- Runtime editor utilities.
- First-person noclip cameras.

Summary
In short:
- This is a **fully customizable free-fly camera**.
- Movement is smooth, precise, and frame-rate independent.
- Look mode mimics professional editor cameras.
- Fast mode allows instant large-scale navigation.
- All behavior is configurable and runtime-safe.

Custom MMD Data

Overview – What this data asset is for
This script defines a **custom data container** used to store **MMD-related metadata** inside Unity.
Instead of controlling behavior or logic, it acts as a **persistent database** that can be:
- Created as an asset.
- Edited in the Inspector.
- Shared across scenes.

- Reused by tools, editors, shaders, or runtime systems.

It is especially useful for **MikuMikuDance workflows**, where materials often need:

- Multilingual names.
- Extra notes.
- Centralized configuration.
- Tool-friendly organization.

Main Data Asset – Custom MMD Data

This asset represents **one complete collection of MMD-related configuration data**.

Think of it as:

- A settings file.
- A material registry.
- A metadata container for tools and pipelines.

showSystemsDefault

What it represents

A simple on/off flag that controls whether additional shader systems or advanced features should be shown by default.

How it's intended to be used

- Editor tools can read this value to decide whether to show advanced UI.
- Shader or material inspectors can use it as a default visibility state.
- Pipelines can adapt their behavior based on this preference.

Why it exists

- MMD workflows can be complex.
- This allows tools to start in a simplified or advanced mode without hardcoding behavior.

materialInfoList

What it represents

A collection of entries, where each entry describes **one MMD material**.

What it contains

- A reference to the actual material used in the project.
- Human-readable names in multiple languages.
- Free-form notes.

Why it exists

- MMD models often come with Japanese material names.
- Teams may need English equivalents.
- Technical notes are often needed for shaders, render pipelines, or conversion steps.

This list acts as a **bridge between raw materials and human understanding**.

Material Entry – Individual Material Information

Each entry in the material list represents **one material and its metadata**.

mmdMaterial

What it represents

A direct reference to a material used in the project.

Purpose

- Links metadata to a real, usable material.
- Allows tools to identify, modify, or display the correct material.

Typical usage

- Editor tools scan this reference to apply settings.
- Importers or converters match materials using this reference.

materialNameJP

What it represents

The material's name written in Japanese.

Why it matters

- Original MMD assets usually use Japanese naming.
- Preserving the original name avoids confusion.
- Helps with cross-referencing original files or documentation.

materialNameEN
What it represents
An English version of the material's name.
Why it matters
- Makes the project accessible to non-Japanese speakers.
- Useful for UI display, debugging, and team collaboration.
- Helps standardize naming conventions across tools.

materialMeno
What it represents
A free-text memo field for additional notes.
Typical content
- Shader quirks.
- Rendering issues.
- Conversion warnings.
- Artistic notes.
- Pipeline instructions.

Why it's important
- MMD materials often need special handling.
- This provides context that code alone cannot express.

How This Data Is Intended to Be Used

As an Editor Tool Backend
- Custom inspectors can read this data to display material lists.
- Tools can toggle advanced shader systems using the default flag.
- Material metadata can be shown in user-friendly panels.

As a Runtime Reference
- Systems can query material names for UI or debugging.
- Localization systems can use the multilingual names.
- Runtime tools can adjust behavior based on stored notes.

As a Pipeline Support Asset
- Importers can map original MMD materials to Unity materials.
- Conversion tools can store decisions and notes.
- Shader systems can adapt per material.

Design Philosophy
This script follows a **data-only design**:
- No behavior.
- No logic.
- No processing.
- Just structured, serializable information.

This makes it:
- Safe to reuse.
- Easy to extend.
- Friendly for tools.
- Stable across refactors.

Summary
In short:
- This script defines a **custom data asset** for MMD workflows.
- It stores:
  - A global visibility preference.
  - A detailed list of material metadata.
- Each material entry includes:
  - A material reference.

    - Japanese and English names.
    - A memo field for human notes.
- It is designed for **editor tools, pipelines, and organization**, not gameplay logic.

Draw Mesh Instanced

Overview – What this component does
This component is a **GPU instanced mesh renderer**.
Instead of creating separate objects for each mesh, it:

- Sends transformation data directly to the graphics processor.
- Draws meshes efficiently using instancing.
- Avoids per-object overhead in the scene hierarchy.

It is designed for:

- Rendering large numbers of repeated meshes.
- Tooling and editor visualization.
- Performance-critical rendering.
- Debug or procedural placement systems.

The component can:

- Draw continuously during runtime.
- Draw only when selected in the editor.
- Reuse materials across submeshes or assign unique ones.
- Visualize instance positions using scene gizmos.

Nested Data Structure – Instance Definition
Each entry in the internal list represents **one instanced mesh configuration**.
Think of it as:
“Draw this mesh, using these materials, at this transform.”

transform
What it represents
Defines where and how the mesh instance appears in the world.
It contains:

- Position.
- Rotation.
- Scale.

Why it matters

- This transform is converted into a matrix sent to the GPU.
- The mesh is drawn exactly at this transform’s location.

mesh
What it represents
The geometric shape to be rendered.
Important note

- The mesh may contain multiple sub-parts (submeshes).
- Each sub-part can use a different material.

materials
What it represents
A list of materials used to render the mesh.
Key requirement

- These materials must support GPU instancing.

How they are used

- Each submesh selects a material from this list.
- Materials can be reused or limited depending on settings.

Inspector Settings – Rendering Behavior

OnDrawSelected
What it controls

Whether meshes are drawn **only when the object is selected in the editor**.

Behavior

- When enabled:
  - Meshes are rendered only while selected.
- When disabled:
  - Meshes are rendered continuously.

Use cases

- Debug visualization.
- Editor-only tools.
- Avoid cluttering the scene when not needed.

reuseMaterials

What it controls

How materials are assigned to submeshes.

When enabled

- Materials repeat cyclically if there are more submeshes than materials.

When disabled

- Rendering stops once all provided materials are used.
- Extra submeshes are not drawn.

Why this exists

- Some meshes reuse materials.
- Others require strict one-to-one material assignment.

drawMeshInstancedLists

What it represents

The full list of instanced mesh configurations.

Each entry defines:

- Where to draw.
- What to draw.
- How it looks.

Why it's a list

- Allows rendering multiple different meshes.
- Each entry is independent.
- Supports complex tool setups.

Runtime Rendering Flow

Continuous Rendering

Every frame:

- The component goes through the entire list.
- Each valid entry is rendered using GPU instancing.
- No scene objects are created.

This ensures:

- Minimal overhead.
- Stable performance.

Editor Visualization – Scene Gizmos

When the object is selected:

Main object indicator

- A green sphere is drawn at the component's position.
- Helps identify the root of the system.

Instance indicators

For each entry with a valid transform:

- A red sphere is drawn at the instance position.
- Shows exactly where each mesh is being rendered.

Purpose

- Debug placement.

- Visualize instanced objects.
- Confirm transforms without spawning objects.

Internal Drawing Logic – How Rendering Happens
For each instanced entry:

1. **Validation**
    - Ensures a mesh exists.
    - Ensures materials exist.
    - Ensures at least one material is provided.
2. **Submesh iteration**
    - Each sub-part of the mesh is handled separately.
    - This allows multiple materials per mesh.
3. **Material selection**
    - Materials are chosen based on submesh index.
    - Can repeat or stop based on reuse setting.
4. **Transform conversion**
    - The transform is converted into a matrix.
    - This matrix is sent to the GPU.
5. **Instanced draw call**
    - The GPU renders the mesh directly.
    - No GameObjects are created.
    - Extremely efficient for repeated geometry.

Usage Guide – How to Use This Component

Basic Setup

1. Add the component to any object.
2. Create one or more entries in the instance list.
3. Assign:
    - A transform.
    - A mesh.
    - One or more materials.
4. Enable GPU instancing on the materials.
5. Choose whether materials should be reused.
6. Decide whether rendering should be editor-only.

Ideal Use Cases

- Rendering large static props.
- Procedural level decoration.
- Visual debugging tools.
- Editor-only previews.
- Crowd or environment instancing.
- MMD accessory rendering pipelines.

Design Philosophy
This component is built around:

- **Performance first** – no object overhead.
- **Tool friendliness** – clear visualization.
- **Flexibility** – reusable or strict materials.
- **Editor support** – draw-on-select mode.

It intentionally avoids:

- Physics.
- Per-instance behavior.
- Animation logic.

Summary
In short:

- This script draws meshes using **GPU instancing**.
- It avoids creating scene objects.

- Each entry defines:
  - A transform.
  - A mesh.
  - Materials.
- It supports:
  - Editor-only drawing.
  - Material reuse.
  - Submesh handling.
  - Visual debugging.

Manage Objects

Overview – What this tool is for

This component is an **editor-only scene management utility**.

Its purpose is to help developers:

- Organize complex scenes.
- Quickly show or hide groups of objects.
- Debug visibility-related issues.
- Reduce visual clutter while working.

It does **not** exist for gameplay or runtime logic.

When the project is built or run outside the editor, the component **automatically removes itself**, ensuring **zero performance cost** in production.

Core Concept – Managed Objects

At its heart, the tool keeps a **list of objects** and tracks whether each one is currently:

- Visible and active.
- Or hidden and inactive.

From a custom inspector interface, you can:

- Toggle objects individually.
- Toggle all objects at once.
- Add objects via drag & drop.
- Remove objects safely with undo support.
- Hide parts of the inspector UI to keep things clean.

Managed Object Data

Each managed entry represents **one scene object and its visibility state**.

Managed Object Reference

What it represents

A reference to an object in the scene hierarchy.

Why it exists

- This is the object whose visibility is being controlled.
- The tool does not create or destroy objects—only activates or deactivates them.

Object State

What it represents

The cached visibility state of the object.

Why it's stored

- Allows the editor UI to reflect the real visibility.
- Makes toggling predictable.
- Keeps the inspector buttons in sync with the scene.

Main Component Settings

Managed Object List

What it represents

The complete list of objects controlled by this tool.

Behavior

- Objects are stored in the order they were added.
- Each object has its own toggle button.

- The list supports undo operations for safety.

How objects are added

- Drag and drop directly into the custom inspector.
- Automatically prevents duplicates.

GlobalState

What it represents

A shared visibility state used when toggling all objects at once.

How it works

- Acts as the “master switch”.
- Flips between visible and hidden each time the global toggle is pressed.
- Applied to every managed object simultaneously.

HideObjects

What it represents

A UI preference, not a scene behavior.

What it does

- Hides the managed object list in the inspector.
- Does not affect the objects themselves.

Why it exists

- Keeps the inspector clean when the list is large.
- Useful once the setup phase is complete.

HideDefaultInspector

What it represents

Another UI preference.

What it does

- Hides Unity’s default component inspector.
- Shows only the custom tool interface.

Why it exists

- Prevents visual clutter.
- Focuses attention on the visibility management controls.

Core Actions – What the Tool Can Do

Toggle a Single Object

What happens

- The selected object’s visibility is flipped.
- If it was visible, it becomes hidden.
- If it was hidden, it becomes visible.

Safety features

- Index validation ensures no invalid access.
- Undo is recorded for:
    - The tool itself.
    - The object being modified.

Result

- One-click visibility control per object.

Toggle All Objects

What happens

- The global state is flipped.
- Every managed object is set to match that state.

Important behavior

- Only valid objects are affected.
- Missing or deleted references are skipped safely.

Undo support

- The global toggle.
- Every affected object.
- All changes can be reverted instantly.

Use cases

- Hide all debug helpers.
- Show all environment pieces.
- Quickly clean up the scene view.

Remove an Object from Management

What happens

- The object is removed from the tool's list.
- The object itself is NOT deleted from the scene.

Why this matters

- This tool manages visibility, not object lifetime.
- Safe to use without risking data loss.

Index Validation

Purpose

- Ensures all operations reference valid objects.
- Prevents null references or out-of-range errors.

Effect

- The tool never crashes due to bad data.
- Deleted or missing objects are handled gracefully.

Editor-Only Behavior

Automatic Self-Removal in Builds

What happens

- When running outside the editor:
  - The component deletes itself immediately.

Why this exists

- Zero runtime overhead.
- No unnecessary components in production builds.
- Clear separation between editor tools and gameplay logic.

Custom Inspector Interface

This tool includes a **fully custom inspector** designed for productivity and safety.

Drag & Drop Area

What it does

- Allows adding objects by dragging them from the hierarchy.
- Works with multiple selected components at once.

Safety features

- Prevents duplicate entries.
- Records undo for all affected components.

Workflow benefit

- Extremely fast setup.
- No manual list editing.

Global Toggle Button

What it does

- Toggles all managed objects in one click.

Visual feedback

- Green: all visible.
- Red: all hidden.
- Yellow: mixed states across multiple selections.

Multi-Object Editing Support

What it supports

- Multiple tool instances selected at once.
- Shared toggles with mixed-value handling.

What it restricts

- Object list editing is disabled when multiple tools are selected.

Why

- Prevents accidental list corruption.
- Keeps behavior predictable.

Managed Object List UI

For each managed object:

- A colored button shows its current state.
- Clicking toggles visibility.
- A remove button unregisters the object.

Color meaning

- Green: visible.
- Red: hidden.

Design Philosophy

This tool is built around:

- **Editor-first workflow**
- **Undo safety**
- **Zero runtime cost**
- **Non-destructive control**
- **Clean, scalable UI**

It intentionally avoids:

- Runtime logic.
- Scene modification beyond visibility.
- Object creation or deletion.

Typical Usage Scenarios

- Organizing large MMD scenes.
- Toggling debug helpers.
- Managing complex rig setups.
- Controlling visibility layers without layers.
- Temporarily hiding heavy objects.
- Editor-only tool pipelines.

Summary

In short:

- This is an **editor-only visibility manager**.
- It lets you register objects and toggle them safely.
- It supports:
  - Individual toggles.
  - Global toggles.
  - Drag & drop setup.
  - Undo at every step.
- It removes itself in builds for zero overhead.
- It improves workflow, clarity, and scene organization.

MMD Bone Renderer

General Purpose

This tool is an **editor-only visualization utility** designed to **draw bone hierarchies directly in the Scene view**.

It does **not** affect gameplay, animation, or runtime behavior.

Its only role is to help developers **inspect, debug, and understand skeletal structures visually** while working inside the editor.

The tool:

- Extracts parent-child relationships between bones.
- Draws visual connections between them.
- Identifies bone tips (end bones).
- Supports grouping bones with different colors.

- Respects Scene visibility rules and layer filters.

High-Level Workflow

1. You assign bones to one or more **bone groups**.
2. Each group has a **visual color**.
3. When the component is enabled or modified:
    - The tool scans all assigned bones.
    - It detects valid parent → child relationships.
    - It caches:
        - Bone connections.
        - Bone tips (bones with no valid children).
4. A separate drawing system (not shown here) uses this cached data to render:
    - Lines between bones.
    - Points for joints and tips.

Data Structures

Bone Group Data

This represents a **logical group of bones**.
Each group contains:

- A **color**, used to draw all bones in that group.
- A **list of bone transform data**, defining which bones belong to the group.

Why this exists:

- Allows visual separation of different skeleton parts.
- Makes complex rigs easier to read.
- Supports multiple overlapping bone systems.

Bone Connection Pair

This represents **one visual bone connection**.
It stores:

- A **parent bone**.
- A **child bone**.

Why this exists:

- Bone rendering is based on connections, not individual bones.
- Each pair becomes a line in the Scene view.

Serialized Settings (Inspector Controls)

Draw Bones Toggle

Controls whether bone visualization is active.
If disabled:

- No bones or joints are drawn.
- Cached data still exists but is not used visually.

Purpose:

- Quickly enable or disable visualization without removing the component.

Joint Size

Controls the **visual size of joint points**.
This affects:

- How large bone joints appear in the Scene view.
- Readability at different zoom levels.

It has no effect on:

- Bone relationships.
- Scene data.
- Runtime behavior.

Bone Group List

This is the **core configuration of the tool**.
It stores:

- All bone groups.
- Their colors.

- Their assigned bones.

Default behavior:

- Starts with two predefined groups using different colors.
- Can be expanded with additional groups.

This list defines **everything the tool visualizes**.

Cached Internal Data

Bone Connections Cache

A list of all valid parent → child bone connections.

Why it is cached:

- Bone extraction can be expensive.
- Rendering needs fast access to relationships.
- Prevents recalculating data every frame.

This cache is rebuilt only when:

- The component is enabled.
- Bone groups are modified.
- The user manually forces a refresh.

Bone Tips Cache

A list of bones that:

- Have no valid child bones inside the configured bone set.

Why tips matter:

- Tips are often drawn differently.
- Useful for identifying chain endings.
- Helps debug broken or incomplete rigs.

Public Properties (Controlled Access)

These provide controlled access to:

- Whether bones are drawn.
- Joint size.
- The list of bone groups.
- Extracted bone connections.
- Extracted bone tips.

Purpose:

- Allow editor tools or other systems to read or modify the tool safely.
- Prevent direct manipulation of internal cached data.

Global Events

Add Bone Renderer Event

Triggered when this component becomes active.

Purpose:

- Allows global editor tools to detect active bone renderers.
- Useful for centralized visualization managers.

Remove Bone Renderer Event

Triggered when this component is disabled.

Purpose:

- Keeps editor tools in sync.
- Ensures removed renderers are no longer considered.

Lifecycle Behavior

When the Component Is Enabled

- Bone data is extracted immediately.
- All bone relationships and tips are rebuilt.
- A global notification is sent.

This ensures:

- Visualization is always up-to-date.
- No stale data remains.

When the Component Is Disabled

- A removal notification is sent.
- Cached data remains but is no longer used.

Public Utility Actions

Invalidate (Rebuild Data)

Forces a full recalculation of:

- Bone connections.
- Bone tips.

Use this when:

- Bones are added or removed.
- Bone hierarchy changes.
- Visibility rules change.

Reset (Clear Data)

Completely removes:

- Cached bone connections.
- Cached bone tips.

Use this for:

- Debugging.
- Temporarily disabling visualization logic.

Add Bone Group

Creates a new bone group with:

- A specified color.
- A list of bones.

Behavior:

- Automatically initializes missing data.
- Prevents null references.
- Rebuilds cached bone data immediately.

Use case:

- Organizing bones by function (arms, legs, face, physics, etc.).

Add Bone to Group

Adds a single bone to an existing group.

Safety rules:

- Invalid group indices are ignored.
- Null or invalid bones are ignored.
- Duplicate bones in the same group are prevented.

After adding:

- All bone relationships are recalculated.

Bone Extraction Logic

This is the **core intelligence of the tool**.

Step 1: Collect Valid Bones

- All bones from all groups are collected into a fast lookup set.
- This allows quick checks for valid parent-child relationships.

Step 2: Iterate Through All Bones

For each bone:

- Hidden objects are ignored.
- Objects outside visible editor layers are ignored.

This ensures:

- Visualization matches the Scene view.
- Hidden or filtered objects do not clutter the view.

Step 3: Detect Child Relationships

For each bone:

- All children are inspected.

- If a child is also a registered bone:
    - A connection is created.
    - The bone is not considered a tip.

Step 4: Detect Bone Tips
If a bone:
- Has no valid registered children,
- It is classified as a **tip**.

Step 5: Cache Results
- All connections are stored as bone pairs.
- All tips are stored separately.
- Both are cached for fast rendering.

Editor-Only Utility: Automatic Bone Collection
Auto Get Bones
This is a **debug and convenience feature**, not the intended workflow.
What it does:
- Replaces all existing bone groups.
- Collects every child transform under this object.
- Assigns them all to a single bone group.

Warnings:
- Ignores proper rig structure.
- May include non-bone objects.
- Intended only for quick tests or debugging.

A confirmation dialog prevents accidental misuse.

Intended Usage
This tool is meant to be used when:
- Debugging skeletal hierarchies.
- Inspecting imported MMD rigs.
- Understanding complex bone chains.
- Visualizing control rigs or helpers.
- Debugging animation or constraint issues.

It is **not** intended to:
- Modify bones.
- Animate bones.
- Replace proper rigging tools.
- Run at runtime.

Design Philosophy
- **Editor-only**: zero runtime impact.
- **Non-destructive**: never modifies scene data.
- **Visibility-aware**: respects editor visibility rules.
- **Performance-safe**: caches extracted data.
- **Debug-focused**: prioritizes clarity over automation.

Summary
In short:
- This component visualizes bone hierarchies in the Scene view.
- Bones are organized into colored groups.
- Parent-child relationships are detected automatically.
- Bone tips are identified explicitly.
- Data is cached and refreshed only when necessary.
- The tool exists purely for editor inspection and debugging.

Bones Transform

General Purpose

This script defines:

1. A **bone configuration data object** that describes how an individual bone should behave.
2. A **custom Inspector layout** that makes editing this data intuitive and compact inside the editor.

Its main goal is to let tools such as:

- bone visualizers,
- animation helpers,
- rig debugging systems,

know **which bone to use** and **which aspects of that bone should be processed** (rotation, position, visibility).
This script does **not** animate, render, or modify bones by itself.
It only **describes intent** and provides a clean way to edit that intent.

Bone Configuration Data
Bone Reference
This field stores a reference to a **single bone object** in the scene hierarchy.
Purpose:

- Identifies which bone this configuration applies to.
- Acts as the anchor for all other options.

Usage:

- Tools read this reference to know *which* bone they should inspect, visualize, or synchronize.

Rotation Toggle
This option indicates whether **rotation data** should be considered for this bone.
If enabled:

- Systems consuming this data may read or apply rotation.
- Bone rotation may be visualized or synchronized.

If disabled:

- Rotation is ignored for this bone.
- The bone may still exist but will not rotate through the consuming system.

Typical use cases:

- Ignoring rotation on helper bones.
- Locking rotation on specific joints.
- Debugging partial animation chains.

Position Toggle
This option indicates whether **position data** should be considered.
If enabled:

- Bone position can be read, modified, or visualized.
- The bone participates in movement-based logic.

If disabled:

- The bone is treated as position-locked.
- Useful for bones that only rotate.

Typical use cases:

- Preventing translation on rigid bones.
- Separating rotation-only and movement bones.

Visibility Toggle
This option controls whether the bone is **considered visible** by systems that render or debug bones.
If enabled:

- The bone may be drawn in the Scene view.
- Debug visuals may include this bone.

If disabled:

- The bone is hidden from visualization tools.
- The bone still exists logically.

This is especially useful for:

- Hiding helper or technical bones.
- Reducing visual clutter in complex rigs.

Custom Inspector Layout

The second part of the script defines **how this bone configuration appears in the Inspector**.
Instead of showing fields in a vertical list, it organizes them in a structured layout for clarity and speed.

Inspector Rendering Behavior
Top Label
The property name is displayed at the top as a header.
Purpose:
- Keeps the Inspector readable when multiple bones are listed.
- Makes it clear where each bone entry begins.

Bone Selection Row
The first row contains:
- A field for selecting the bone reference.

This is the most important field and is visually separated to:
- Emphasize its importance.
- Prevent confusion with the toggles below.

Toggle Row (Rotate, Move, Visible)
The second row displays three compact toggles side by side:
- Rotation
- Position
- Visibility

Design goals:
- All behavioral flags visible at once.
- Quick comparison between bones.
- Faster editing when working with many entries.

Label widths are reduced to:
- Save horizontal space.
- Keep the layout clean and aligned.

Property Height Calculation
The drawer explicitly defines how much vertical space it needs.
This ensures:
- No overlapping UI elements.
- Consistent spacing in lists and arrays.
- Proper layout even when nested inside other data structures.

The total height accounts for:
- The label row.
- The bone field row.
- The toggle row.
- Standard spacing between lines.

Editor-Only Scope
Everything related to the custom Inspector exists **only inside the editor**.
This means:
- No runtime cost.
- No build impact.
- No effect on performance or memory in final builds.

The bone configuration data itself can exist at runtime, but the visual editing tools do not.

How This Is Meant to Be Used
This system is designed to be used as part of:
- Bone visualization tools.
- Animation pipelines.
- Rig debugging systems.
- Editor utilities for skeletal inspection.

Typical workflow:
1. Add a list of bone configurations to a tool.
2. Select a bone for each entry.

3. Enable or disable rotation, movement, and visibility as needed.
4. Other systems read this data and behave accordingly.

Design Philosophy

- **Data-driven**: This script describes *what should happen*, not *how it happens*.
- **Non-invasive**: Never modifies bones directly.
- **Editor-friendly**: Optimized for clarity and speed.
- **Extensible**: Additional flags can be added without breaking existing tools.
- **Reusable**: Can be consumed by multiple systems simultaneously.

Summary

In short:

- This script defines a **single bone configuration unit**.
- Each unit specifies:
  - Which bone is targeted.
  - Whether rotation is used.
  - Whether position is used.
  - Whether the bone is visible.
- A custom Inspector layout makes editing these settings fast and readable.
- The script acts as a **bridge between bone data and editor tools**, not as a bone controller.

MMD Bone Renderer Inspector

Overall Purpose

This script defines a **custom Inspector interface** for the bone renderer component.
Its job is to:

- Detect **any change made in the Inspector**.
- Detect **Undo and Redo actions**.
- Automatically force the bone renderer to **recalculate its internal bone data** whenever something changes.

In short:

👉 It guarantees that **bone visualization always stays in sync with Inspector edits**, without requiring manual refreshes.
This script does **not** draw bones, manage data, or store settings.
It only **reacts to Inspector interactions** and keeps the bone renderer updated.

Editor Scope and Limitations

This inspector exists **only inside the Unity Editor**.
That means:

- It never runs in builds.
- It has zero runtime cost.
- It is purely a workflow and debugging aid.

Multi-Object Support

The inspector explicitly supports **editing multiple bone renderer components at once**.
This allows:

- Selecting multiple objects in the scene.
- Editing shared settings.
- Ensuring all selected bone renderers stay synchronized.

Inspector Behavior Explained

Default Inspector Rendering

The script intentionally displays the **standard Inspector layout** for the bone renderer component.
Why this matters:

- No custom UI is reimplemented.
- All existing fields remain visible and editable.
- Future changes to the bone renderer automatically appear here.

This keeps maintenance low and behavior predictable.

Change Detection
The inspector actively checks whether **any value changed** while the Inspector was drawn.
This includes:

- Toggling visualization options.
- Changing joint size.
- Editing bone groups.
- Modifying bone lists.
- Adjusting colors.

If *anything* changes, the inspector marks the component as needing an update.

Undo and Redo Awareness
In addition to direct edits, the inspector also listens for:

- Undo actions.
- Redo actions.

This is critical because:

- Undo/Redo does not always trigger normal change detection.
- Bone data may become outdated if not refreshed.

By explicitly detecting Undo/Redo events, the inspector ensures:

- Reverted changes immediately reflect in bone visualization.
- No stale or mismatched bone data remains.

Bone Data Refresh Logic
When a change or Undo/Redo is detected:

1. The inspector loops through **all selected bone renderer components**.
2. For each one, it requests a **full bone data rebuild**.
3. The bone renderer re-extracts:
    - Bone connections.
    - Bone tips.
    - Parent-child relationships.

This guarantees:

- Scene view visualization updates immediately.
- No manual refresh button is needed.
- Data always matches what the Inspector shows.

Why This Script Is Necessary
Without this inspector:

- Bone extraction might only occur on enable.
- Inspector edits could leave internal data outdated.
- Undo/Redo could silently desynchronize visuals.
- Developers would need manual refresh methods.

This script solves all of that automatically.

Intended Usage
You do not interact with this script directly.
Typical usage looks like this:

1. Add the bone renderer component to an object.
2. Edit its settings normally in the Inspector.
3. Add, remove, or adjust bones.
4. Use Undo and Redo freely.
5. Bone visualization updates instantly and correctly.

This script works entirely **behind the scenes**.

Design Philosophy

- **Non-invasive**: Does not replace the default Inspector.
- **Automatic**: No buttons or manual refresh required.
- **Robust**: Handles Undo/Redo correctly.
- **Multi-object safe**: Works across multiple selections.
- **Maintenance-friendly**: Automatically adapts to future field changes.

Summary
In simple terms:

- This script watches the Inspector.
- If *anything* changes — including Undo/Redo — it reacts.
- It forces the bone renderer to rebuild its internal data.
- This keeps visualization accurate and reliable at all times.

It acts as a **silent synchronization layer** between:

- Inspector edits
  and
- Scene view bone visualization.

MMD Bone Renderer Utils

High-Level Purpose
This script is an **editor-only rendering and interaction system** for MMD bone structures inside the **Unity Scene view**.
Its responsibilities are:

- Visually rendering bones efficiently using GPU instancing.
- Drawing bones as solid + wireframe pyramid shapes.
- Supporting large skeletons without performance degradation.
- Allowing bones to be **hovered, selected, highlighted, dragged, and transformed**.
- Synchronizing bone visibility with Unity's layer and visibility systems.
- Automatically reacting to hierarchy changes, visibility changes, and prefab stages.

This is the **visual and interactive backbone** of the entire MMD bone visualization system.

Core Architecture Overview
The system is built around **four main layers**:

1. **Global static manager**
   Keeps track of all active bone renderer components.
2. **Batch rendering system**
   Uses GPU instancing to efficiently draw thousands of bones.
3. **Scene view interaction logic**
   Handles hover, selection, dragging, and tool switching.
4. **Editor lifecycle hooks**
   Automatically reacts to scene changes, visibility changes, and prefab context.

Static Nature of the System
This utility is **static and auto-initialized**.
That means:

- It loads automatically when the editor starts.
- You never add it manually.
- It manages all bone renderers globally.
- It exists only once per editor session.

Internal Batch Renderer (Performance Core)
Purpose
The internal batch renderer is responsible for **efficient drawing**.
Instead of drawing each bone individually, it:

- Groups bones together.
- Sends them to the GPU in large batches.
- Avoids per-bone draw calls.

This is essential for MMD models, which often have **hundreds or thousands of bones**.

Key Responsibilities

- Store transformation data for each bone.
- Store base colors and highlight colors.
- Split rendering into chunks to respect GPU limits.
- Render both solid faces and wireframe overlays.

Important Internal Concepts

- **Instance limit**
  The GPU can draw a limited number of instances at once, so bones are split into chunks automatically.
- **Two visual passes**
  - Solid pyramid (bone body).
  - Wireframe pyramid (bone outline).
- **Highlight support**
  Bones can change color when hovered or selected without re-rendering geometry.

Global Static State

Registered Bone Renderers

A shared list stores **all active bone renderer components** in the scene.
Whenever:

- A bone renderer is enabled → it is registered.
- A bone renderer is disabled → it is removed.

This allows centralized rendering and interaction.

Shared Rendering Resources

The system lazily creates and reuses:

- A shared material.
- A shared pyramid mesh.
- A shared batch renderer.

This avoids duplication and reduces memory usage.

Pyramid Bone Mesh

Shape Choice

Bones are visualized as **pyramids**, because:

- Direction is obvious (tip points forward).
- Orientation is easy to read.
- Depth and hierarchy are visually clear.

Mesh Structure

The pyramid mesh contains:

- One solid sub-mesh for faces.
- One line sub-mesh for wireframe.

Both are drawn in a single batched process.

Scene Rendering Loop

Scene View Hook

The system hooks directly into the Scene view rendering pipeline.
Every frame:

- It clears previous bone data.
- Iterates through all registered bone renderers.
- Collects visible bones.
- Queues rendering data.
- Executes a single batched render pass.

Visibility Awareness

Bones are automatically skipped if:

- Their GameObject is hidden.
- Their layer is not visible.
- They are outside the current prefab editing stage.

This ensures correct editor behavior.

Bone Group Handling

Bones are organized into **groups**, each with:

- Its own color.
- Its own visibility rules.

- Its own list of bones.

Only bones that:

- Belong to the same group.
- Are explicitly marked as visible.

are rendered together.

Bone Rendering Logic

Bone Connections

- Bones with children are rendered as directional pyramids.
- Bones without valid children are treated as **tips**.

Tip bones:

- Render shorter.
- Still remain interactive.

Transform Matrix Computation

For each bone:

- The direction is calculated from parent to child.
- A local orientation basis is built.
- Scale depends on bone length.
- Position is offset so the pyramid sits correctly.

This ensures:

- Bones always point correctly.
- Scaling remains consistent across different skeleton sizes.

Interaction System

Each bone supports **full Scene view interaction**.

Hover Detection

When the mouse moves over a bone:

- The bone highlights.
- The Scene view repaints automatically.

Selection

Clicking a bone:

- Selects the bone's GameObject.
- Respects scene visibility and picking rules.
- Highlights the bone.

Tool Switching Logic

Based on bone settings:

- Rotation enabled → rotation tool is activated.
- Movement enabled → move tool is activated.
- Both enabled → full transform tool is activated.

This makes bone manipulation **context-aware** and intuitive.

Drag-and-Drop Support

Dragging a bone:

- Starts a drag operation.
- Allows bones to be referenced or reused.
- Integrates with Unity's drag system.

Highlighting Rules

Bone color changes based on state:

- Normal → group color.
- Hovered → preselection color.
- Selected → selection color.

Highlighting is dynamic and does not require re-batching.

Automatic Refresh Triggers

The system automatically refreshes when:

- Scene visibility changes.
- Layers change.
- Hierarchy changes.
- Bone renderers are added or removed.

Each refresh:

- Invalidates bone data.
- Rebuilds connections.
- Repaints the Scene view.

Usage Summary

From a user perspective:

1. Add a bone renderer component to a character.
2. Configure bone groups and visibility.
3. Open the Scene view.
4. Bones appear automatically.
5. Hover, click, drag, and manipulate bones naturally.
6. Performance remains stable even with large rigs.

No manual setup for this utility is required.

Design Philosophy

- **Editor-only**: zero runtime cost.
- **Centralized**: one renderer for all skeletons.
- **Scalable**: handles thousands of bones.
- **Interactive**: full Scene view integration.
- **Non-invasive**: respects Unity visibility and prefab systems.

In One Sentence

This script is the **high-performance, interactive Scene view engine** that renders, highlights, selects, and manipulates MMD bones in real time while staying fully synchronized with Unity's editor systems.

Check Shader Keywords

Overall Purpose

This script is an **editor utility** that lets you **inspect which global shader keywords are defined by the shaders used in selected materials**.

In short, it helps you answer questions like:

- "Which global keywords does this shader expose?"
- "Is this shader enabling features globally that might affect performance or compatibility?"
- "Why is a shader behaving differently across projects or scenes?"

It adds a **right-click menu option in the Assets window** so you can run this check on materials directly.

Where and When This Script Works

- It works **only inside the Unity Editor**.
- It does **not run in builds or at runtime**.
- It is triggered manually by the user through the **Assets context menu**.

Core Idea Behind the Script

Unity internally knows which **global shader keywords** are available for a shader, but:

- That information is **not publicly accessible**.
- Unity does not provide a built-in inspector view for it.

This script:

- Accesses Unity's internal systems.
- Extracts the global keyword list.
- Logs it in a readable format.

Static Nature of the Script

The script is designed as a **static utility**:

- No instances are created.
- No components are attached to objects.

- Everything runs globally and on demand.

This makes it lightweight and ideal for diagnostic tasks.

Variable Explanation

Cached Internal Method Reference

The script stores a **cached reference** to Unity's internal function that retrieves global shader keywords.

Why this exists:

- Accessing internal Unity systems requires reflection.
- Reflection is slow if done repeatedly.
- By caching the method once, performance is improved.

What this variable represents:

- A pointer to Unity's hidden keyword retrieval function.
- It is initialized only once and reused afterward.

Menu Command Function (Main Entry Point)

Purpose

This is the function that runs when you click:

Assets → MMD Collection → Check Shader Keywords

It acts as the **controller** of the entire process.

What It Does Step by Step

1. **Reads current selection**
    - Looks at everything you currently have selected in the Project window.
2. **Filters materials**
    - If the selected object is a material, it proceeds.
    - If not, it logs a warning and skips it.
3. **Validates shader assignment**
    - If a material does not have a shader, it logs an error.
    - This prevents invalid or broken materials from causing failures.
4. **Delegates keyword inspection**
    - For valid materials, it passes the shader to the utility function that extracts keywords.

This separation keeps responsibilities clean and readable.

Utility Function: Keyword Extraction and Logging

Purpose

This function does the **actual work** of retrieving and displaying global shader keywords.

It is intentionally private because:

- It is not meant to be called directly by users.
- It exists only to support the menu command.

Internal Workflow

1. **Null safety check**
    - If the shader reference is missing, the function exits safely.
2. **Lazy initialization**
    - If the internal Unity method has not been retrieved yet:
        - The script searches for it using reflection.
        - The result is cached for future calls.
3. **Failure handling**
    - If Unity changes its internals and the method cannot be found:
        - An error is logged.
        - The process stops safely.
4. **Keyword retrieval**
    - The internal Unity method is executed.
    - The result is expected to be a list of keyword names.
5. **Result validation**
    - If no keywords exist:
        - A clear message is logged stating that.
    - If keywords exist:
        - They are printed line by line for readability.

Why Reflection Is Used
Unity does not publicly expose global shader keyword information.
Reflection is used because:
- It allows access to internal editor-only APIs.
- It enables powerful inspection tools without modifying Unity itself.

This is safe as long as:
- It runs only in the editor.
- Errors are handled gracefully (which this script does).

Output Behavior
All results are logged to the **Unity Console**:
- Each shader is clearly identified by name.
- Keywords are printed one per line.
- Messages are context-aware and reference the material or shader asset.

This makes debugging easy when working with multiple materials.

Typical Usage Scenario
1. Select one or more materials in the Project window.
2. Right-click on them.
3. Choose **Check Shader Keywords** from the menu.
4. Open the Console.
5. Review the list of global shader keywords for each shader.

Why This Tool Is Useful
- Debugging shader variants.
- Verifying feature flags in complex shaders.
- Auditing third-party or imported shaders.
- Diagnosing unexpected rendering behavior.
- Understanding shader compilation differences between projects.

Design Philosophy
- **Non-invasive**: Does not modify assets.
- **Editor-only**: Zero runtime cost.
- **Safe**: Defensive checks prevent crashes.
- **Performant**: Reflection is cached.
- **Clear feedback**: Logs meaningful messages.

One-Sentence Summary
This script is an **editor diagnostic tool** that lets you inspect and log **global shader keywords used by selected materials**, using Unity's internal systems in a safe, efficient, and user-friendly way.

Create Prefab From Model

Overall Purpose
This script is a **Unity Editor utility** that turns selected **3D model assets** (such as FBX files) into **standard prefabs**.
Why this is useful:
- Imported models are usually *model prefabs*, which are limited.
- This tool extracts the **meshes and materials** from the model.
- It rebuilds them as **regular GameObjects**.
- It saves everything as a **new prefab** in the same folder as the original model.

The result is a prefab you can freely edit, instance, or extend without being tied to the original model importer.

Where and When This Script Works
- Runs **only in the Unity Editor**.
- Appears as a **right-click menu option in the Assets window**.
- Does **not run at runtime** and does not affect builds.

High-Level Workflow

1. You select one or more **3D model assets** in the Project window.
2. You use the custom menu command.
3. The script:
    - Finds all meshes inside each model.
    - Converts each mesh into a standalone GameObject.
    - Preserves the original materials.
    - Combines everything into a new prefab.
4. The prefab is saved next to the original model file.

Script Structure Overview

The script is divided into three conceptual areas:

1. **Menu entry and selection handling**
2. **Model-to-GameObject conversion**
3. **Prefab creation and saving**

Each part has a clear responsibility.

Menu Entry (How You Trigger It)

Menu Command

The script adds a menu item to Unity:
Assets → MMD Collection → Create Prefabs From Selected Model
This is the **only entry point** for the tool.

What Happens When You Click It

- Unity gives the script a list of everything you currently selected.
- The script filters that list to keep **only valid 3D model assets**.
- If nothing valid is selected:
    - A warning is shown.
    - The process stops safely.
- Each valid model is then processed individually.

This ensures:

- No accidental prefab creation.
- No crashes from unsupported asset types.

Model Conversion Logic

Purpose

For each selected model, the script converts its internal mesh data into **new GameObjects** that can live independently from the model importer.

What Gets Extracted

The script searches through the entire model hierarchy and looks for:

- **Skinned meshes** (used for animated characters).
- **Static meshes** (used for props and environment objects).

Both types are handled so the tool works for:

- Characters
- Props
- Environments
- Mixed models

Conversion Process

For each mesh found:

1. The script reads:
    - The mesh geometry.
    - The materials assigned to it.
2. A new empty GameObject is created.
3. The mesh is assigned to it.
4. The materials are restored exactly as they were.
5. The object is stored in a temporary list.

This guarantees:

- No data loss.
- Visual fidelity stays intact.
- Materials are not duplicated unnecessarily.

Handling Different Mesh Types

Skinned Meshes

- Used for characters and deformable meshes.
- The script extracts:
    - The shared mesh.
    - All assigned materials.

Static Meshes

- Used for props and rigid objects.
- The script:
    - Reads the mesh from the mesh container.
    - Reads materials from the renderer.

If a mesh has no materials or is improperly configured:

- The script logs an error.
- That specific object is skipped.
- The rest of the model continues processing.

Prefab Creation Logic

Purpose

Once all mesh GameObjects are created, the script packages them into a prefab.

Single Mesh Case

If the model contains only one mesh:

- That GameObject becomes the prefab root.
- The prefab is named after the model.

Multiple Mesh Case

If the model contains multiple meshes:

- A new root GameObject is created.
- All generated mesh objects become its children.
- This mirrors how complex models are usually structured.

Saving the Prefab

- The prefab is saved in the **same folder** as the original model.
- The name clearly indicates it is a prefab version.
- If saving succeeds:
    - A confirmation message is logged.
- If saving fails:
    - A clear error message is shown.

After saving:

- All temporary objects are immediately destroyed.
- The scene is left clean and unchanged.

Safety and Error Handling

The script is defensive and editor-safe:

- Ignores invalid selections.
- Handles missing meshes or materials.
- Prevents null references.
- Catches file system or prefab save errors.
- Always cleans up temporary objects.

This makes it safe to use repeatedly, even in large projects.

Practical Usage Example

1. Import an FBX or similar 3D model.
2. Select the model in the Project window.

3. Right-click → **Create Prefabs From Selected Model**.
4. A new prefab appears in the same folder.
5. You can now:
    - Add scripts.
    - Modify hierarchy.
    - Use prefab variants.
    - Edit materials freely.

Why This Tool Is Valuable

- Converts locked model prefabs into editable assets.
- Preserves original mesh and material data.
- Speeds up character and prop workflows.
- Ideal for MMD models, imported assets, and kitbashing.
- Eliminates manual prefab rebuilding.

One-Sentence Summary

This script is an **editor tool that converts selected 3D model assets into fully editable prefabs by extracting meshes and materials, rebuilding them as GameObjects, and saving the result safely and automatically**.

Find Missing Scripts

Overall Purpose

This script is an **Editor utility** that helps you **find broken objects in a Unity scene**.
Specifically, it:

- Scans **every GameObject in all loaded scenes**.
- Detects **missing scripts** (components that were deleted or no longer exist).
- Logs how many objects are affected.
- Automatically **selects those objects** in the Editor so you can fix them.

Missing scripts often happen when:

- A script file is deleted or renamed.
- A package is removed.
- A merge goes wrong.
- A prefab was created from an older project version.

This tool makes finding those issues fast and painless.

Where and When This Script Works

- Works **only in the Unity Editor**.
- Does **not run in play mode or builds**.
- Appears in the **GameObject menu**, not the Assets menu.

How You Use It

1. Open a scene (or multiple scenes).
2. In the top menu bar, click:
   GameObject → MMD Collection → Find Missing Scripts
3. The script:
    - Searches the scene(s).
    - Logs how many broken objects it found.
    - Selects them all automatically.
4. You can now:
    - Inspect each object.
    - Remove missing components.
    - Replace scripts.
    - Fix prefabs safely.

Script Structure Overview

The script is intentionally small and focused.
It contains:

1. **One menu command**

2. **One main method**
3. **A few helper checks**
4. **No persistent state**

This makes it safe, fast, and easy to maintain.

Variables Explained

List of Objects With Missing Scripts

- This list stores **every GameObject that has at least one missing script**.
- It starts empty.
- Objects are added only if a problem is detected.
- At the end, this list becomes the Editor selection.

Purpose:

- Keeps results organized.
- Allows batch selection in the Editor.

Collection of All GameObjects

- This is a snapshot of **every GameObject Unity knows about right now**.
- Includes:
  - Scene objects
  - Inactive objects
  - Objects in loaded scenes
- Excludes objects that are not loaded into memory.

Purpose:

- Ensures nothing is missed, even disabled or hidden objects.

Main Method Explanation

Menu Action Method

This is the method that runs when you click the menu item.

Its responsibilities are:

1. Gather all GameObjects.
2. Filter out objects that should not be checked.
3. Inspect each object's components.
4. Detect missing scripts.
5. Report and select the results.

Step 1: Scene and Visibility Filtering

Before checking components, the script ignores objects that:

- Do not belong to a valid scene.
- Are in scenes that are not loaded.
- Are hidden or marked as non-editable by Unity.

Why this matters:

- Prevents false positives.
- Avoids editor-internal objects.
- Keeps results relevant and clean.

Step 2: Component Inspection

For each valid GameObject:

- The script retrieves **all attached components**.
- Unity represents missing scripts as **empty component slots**.
- If **any component is empty**, the object is considered broken.

This is the key detection mechanism.

Step 3: Collecting Broken Objects

- If an object has at least one missing script:
  - It is added to the results list.
- Objects with no issues are ignored.

This ensures:

- Each GameObject appears only once.
- The list stays concise and useful.

Final Actions
Logging the Results
After scanning everything:
- The script logs how many objects were found.
- This gives immediate feedback without opening the console spam.

Selecting the Objects
- All problematic GameObjects are selected at once.
- This allows:
  - Quick inspection in the Hierarchy.
  - Easy bulk fixing.
  - Immediate visibility of the problem areas.

This is the most important usability feature.

Why This Script Is Useful
- Saves hours of manual searching.
- Prevents runtime errors caused by missing scripts.
- Helps clean up broken prefabs and scenes.
- Essential for large or long-running projects.
- Safe to run at any time.

Typical Use Cases
- After deleting or renaming scripts.
- After upgrading Unity versions.
- After importing or removing packages.
- After resolving merge conflicts.
- Before committing or releasing a build.

One-Sentence Summary
This script is an **Editor-only cleanup tool that scans all loaded scenes, finds GameObjects with missing scripts, reports how many exist, and automatically selects them so you can fix them quickly**.

Material Property Cleaner

Overall Purpose
This script is a **Unity Editor cleanup tool for Materials**.
Its job is to:
- Remove **obsolete or invalid material properties** that no longer exist in the material's shader.
- Remove **invalid shader keywords** that are no longer supported by the shader.
- Keep materials clean, consistent, and safe after shader changes.

This is especially useful after:
- Changing or upgrading shaders.
- Switching render pipelines.
- Importing assets from other projects.
- Removing shader features or variants.
- Fixing warnings, broken visuals, or bloated materials.

Where and When This Script Works
- **Editor-only tool**
- Does **not run in play mode**
- Works on **selected Material assets**
- Appears in the **Assets context menu**

How You Use It
1. In the **Project window**, select one or more **Material assets**.
2. Right-click the selection (or use the top menu).
3. Choose:

Assets → MMD Collection → Clean Invalid Material Properties

4. Confirm the warning dialog.
5. The script:
    - Cleans unused material properties.
    - Removes unsupported shader keywords.
    - Saves and refreshes the assets.

⚠ This action **cannot be undone**, so it asks for confirmation first.

What Problems This Script Solves

- Materials storing **old properties from previous shaders**.
- Hidden texture slots that no longer exist.
- Float, color, or integer values that the shader never reads.
- Shader keywords left behind by removed features.
- Larger file size and unpredictable behavior.

Script Structure Overview

The script is divided into three logical parts:

1. **Cached reflection data**
2. **Menu action**
3. **Cleanup utilities**

Each part has a single responsibility.

Variables Explained

Cached Shader Keyword Method

- Stores a reference to an **internal Unity editor method** that lists valid shader keywords.
- Retrieved only once and reused later.
- Improves performance and avoids repeated lookups.

Purpose:

- Allows checking which keywords are still valid for a shader.
- Prevents accidental removal of legitimate keywords.

Main Menu Method

"Clean Invalid Material Properties"

This is the **entry point** when you click the menu item.

Its responsibilities:

1. Ask the user for confirmation.
2. Loop through all selected assets.
3. Process only Materials.
4. Clean invalid properties.
5. Clean invalid shader keywords.
6. Save the changes.

Confirmation Dialog

Before doing anything destructive:

- A dialog warns the user that:
    - The operation affects assets.
    - The operation cannot be undone.
- If the user cancels, nothing happens.

Purpose:

- Prevents accidental asset damage.
- Encourages safe usage.

Selection Processing

For each selected object:

- If it is **not a Material**, a warning is shown.
- If it **is a Material**, it continues.
- Materials without shaders are skipped with an error message.

This keeps the tool safe and predictable.

Undo Registration
The material is registered for Editor tracking before changes.
Purpose:
- Allows Unity to track modifications.
- Ensures changes are properly saved and serialized.

Cleaning Material Properties
Collecting Valid Shader Properties
- The script asks the shader which properties it currently supports.
- All valid property names are stored in a lookup set.

Purpose:
- This becomes the **reference list** for cleanup.
- Anything not in this list is considered invalid.

Accessing Stored Material Data
- The material's serialized data is accessed directly.
- This includes:
    - Textures
    - Floats
    - Integers
    - Colors

Unity keeps these even if the shader no longer uses them.

Removing Invalid Properties
Each property list is scanned:
- If a property name **does not exist in the shader**, it is removed.
- The scan runs backwards to avoid indexing issues.

This removes:
- Ghost properties.
- Legacy shader leftovers.
- Corrupted or unused data.

Cleaning Shader Keywords
Why Keywords Matter
Shader keywords control:
- Feature toggles
- Variants
- Conditional rendering paths

Invalid keywords:
- Increase build size.
- Cause wrong visuals.
- Trigger warnings.

Getting Valid Keywords
- The script queries Unity internally to get all **valid global keywords** for the shader.
- This requires reflection because Unity does not expose it publicly.

The result becomes the allowed keyword list.

Removing Invalid Keywords
For each keyword on the material:
- If the shader does **not recognize it**, the keyword is disabled.
- Removed keywords are logged for transparency.

This ensures:
- Materials only use supported shader features.
- Cleaner and more predictable rendering.

Finalization
After cleaning:
- Changes are applied.

- The material is marked as modified.
- Assets are saved and refreshed.

This guarantees:

- No data loss due to unsaved changes.
- Immediate visibility in the Editor.

Typical Use Cases

- After switching shaders on many materials.
- After removing shader features or passes.
- After upgrading Unity or render pipelines.
- Before shipping or optimizing a project.
- When materials behave inconsistently.

Safety Notes

- The operation **cannot be undone**.
- Always back up or use version control.
- Only affects **selected materials**.

One-Sentence Summary

This script is an **Editor cleanup tool that removes obsolete material properties and unsupported shader keywords from selected materials, keeping them clean, lightweight, and compatible with their current shaders**.

Material Shader Converter

1. Overall Purpose

This tool is an **Editor-only conversion utility** for Unity projects that use **MMD4Mecanim materials** and need to migrate them to **URP-compatible shaders**.

Its main goals are:

- Detect which MMD shader a material is using.
- Replace it with the correct URP shader variant.
- Preserve as much visual behavior as possible:
  - Colors.
  - Transparency.
  - Outline (edge) rendering.
  - Tessellation.
  - Shadow and self-shadow behavior.
- Clean up invalid properties and keywords left behind after the conversion.

In short:

It automates what would otherwise be a very tedious, error-prone manual shader migration.

2. Where and When It Runs

- This tool **only exists inside the Unity Editor**.
- It appears as a **right-click menu option in the Project window**.
- It works on **selected material assets**, not on GameObjects in the scene.

Because it is Editor-only:

- It does not affect builds.
- It is safe to use repeatedly.
- Undo support is included.

3. Shader Model Concept

The tool classifies shaders into **conversion models**.

Each model represents a **rendering behavior**, not just a visual style.

The shader models are:

- **Default**
  Standard URP shader with no special passes.
- **Tessellation**
  Uses tessellation features to add geometry detail.
- **Empty**

Minimal “dummy” shader used for placeholder or invisible materials.
- **Four Layers / Eight Layers**
  Special multi-pass shaders that render multiple outline layers.
- **No Shadow**
  Shader explicitly configured to never cast or receive shadows.
- **No Shadow + Tessellation**
  Combines tessellation with disabled shadow behavior.

This classification lets the tool choose **how** to convert a material, not just **which shader to assign**.

4. Shader Mapping Table (The Core Brain)
At the heart of the tool is a **mapping table**.
For each original MMD shader name, the table defines:
- Which URP shader should replace it.
- Whether the material is:
  - Transparent.
  - Using outlines.
  - Double-sided.
  - Using global shadow casting.
- Which shader model category it belongs to.

This table allows the converter to:
- Make **deterministic decisions**.
- Avoid guessing or heuristics.
- Be easily extended in the future.

If a shader is **not in the table**, the tool refuses to convert it and warns you.

5. Menu Action (Entry Point)
When you click:
Assets → MMD Collection → Convert Material Shader (MMD4Mecanim)
the tool:
1. Looks at everything you selected.
2. Filters out anything that is not a material.
3. For each valid material:
   - Reads its current shader name.
   - Searches the mapping table.
   - Converts it if a match is found.
   - Logs a warning if no mapping exists.

This design ensures:
- No silent failures.
- No accidental conversions.
- Clear feedback in the console.

6. Conversion Flow (High-Level)
For each material being converted, the tool follows this flow:
1. **Record Undo state**
   So the user can safely revert.
2. **Backup important render settings**
   Such as:
   - GPU instancing.
   - Global illumination flags.
   - Double-sided GI.
3. **Apply the correct conversion method**
   Based on the shader model:
   - Standard shader conversion.
   - Empty shader conversion.
   - Multi-pass outline conversion.
4. **Restore render settings**
   Ensures visual behavior remains consistent.
5. **Clean invalid data**
   Removes unused properties and keywords that could cause warnings or bugs.

6. **Mark the material as modified**
   So Unity saves the changes correctly.

7. Standard Shader Conversion (Most Common Path)
This is the most complex and important path.
What gets preserved:
- Base color (diffuse).
- Specular color and intensity.
- Ambient lighting color.
- Shininess.
- Textures:
  - Main texture.
  - Toon texture.
  - Sphere map.
- Edge (outline) color and thickness.
- Transparency state.
- Shadow behavior.
- Render queue ordering.
- Tessellation values (if applicable).

Key behaviors handled:
- **Transparency**
  - Automatically switches render mode.
  - Moves the material into the correct transparent render queue.
  - Handles transparent outlines correctly.
- **Outlines**
  - Enables or disables outline rendering.
  - Transfers color and thickness.
  - Adjusts size scaling to match URP behavior.
- **Double-sided rendering**
  - Adjusts face culling properly.
- **Shadows**
  - Enables or disables shadow casting.
  - Controls self-shadowing and shadow receiving.
- **Tessellation**
  - Preserves edge length.
  - Preserves phong smoothing strength.
  - Preserves extrusion amount.

This ensures the converted material **looks as close as possible** to the original MMD version.

8. Empty Shader Conversion
Used for dummy or placeholder materials.
This path:
- Assigns a minimal URP shader.
- Forces transparency.
- Disables all shadows.
- Uses default back-face culling.
- Pushes the material into the transparent render queue.

This avoids unnecessary rendering cost and visual artifacts.

9. Multi-Pass Outline Conversion
Used for advanced MMD outline shaders that rely on **multiple rendering passes**.
This conversion:
- Assigns a special URP shader designed for layered outlines.
- Configures whether the shader uses:
  - Four outline layers, or
  - Eight outline layers.
- Transfers outline color and thickness.
- Keeps the original render queue intact.

This preserves stylized anime-style outlines accurately.

10. Render Queue Adjustment Logic
The tool carefully adjusts render queues when transparency is involved.
Why this matters:
- Opaque objects must render before transparent ones.
- Transparent objects must sort correctly to avoid depth artifacts.

The converter:
- Detects if the original queue was opaque or transparent.
- Moves it into the correct transparent range when necessary.
- Leaves it unchanged if no conversion is needed.

11. Material Cleanup (Critical for Stability)
After shader replacement, many properties and keywords become **invalid**.
The cleanup phase:
Property Cleanup
- Scans the new shader to find valid property names.
- Removes:
  - Invalid textures.
  - Invalid floats.
  - Invalid integers.
  - Invalid colors.
- Prevents:
  - Inspector warnings.
  - Broken serialization.
  - Hidden performance issues.

Keyword Cleanup
- Retrieves the shader's valid global keywords.
- Disables any material keyword that the shader does not support.
- Prevents shader variant pollution and incorrect rendering paths.

This step is **essential** and often skipped in simpler converters.

12. Reflection Usage (Why It Exists)
The tool uses reflection to access **internal Unity Editor APIs** that are not publicly exposed.
Specifically:
- It retrieves the list of valid global shader keywords.

Why this matters:
- Unity does not provide a public API for this.
- Without it, keyword cleanup would be incomplete.
- Performance impact is minimized by caching the reflected method.

13. Practical Usage Guide
How to use it:
1. Select one or more **materials** in the Project window.
2. Right-click.
3. Choose:
   MMD Collection → Convert Material Shader (MMD4Mecanim)

Best practices:
- Run it **before** doing manual material tweaks.
- Keep your materials backed up (Undo is supported).
- Check the console for warnings about unmapped shaders.

14. Why This Tool Is Well-Designed
- Deterministic behavior (no guessing).
- Preserves visual fidelity.
- Undo-safe.
- Handles edge cases like outlines, tessellation, and shadows.
- Cleans up after itself.
- Extensible via the mapping table.

In short:

This is not just a shader swapper — it's a full material migration system.

Paste As Child Multiple

1. What This Tool Is For

This is a **Unity Editor utility window** designed to speed up a very common workflow:
Take one object (a prefab or a scene object) and paste it as a child under many selected objects at once.
Instead of:

- Selecting a parent
- Copying
- Pasting
- Repeating dozens of times

This tool does it **in one operation**, with extra quality-of-life features:

- Undo support.
- Optional automatic name numbering.
- Automatic expansion of the hierarchy so you can immediately see the result.

2. Where It Lives and How It's Accessed

- It only exists **inside the Unity Editor**.
- It appears in the menu under:

GameObject → MMD Collection → Paste as Multiple Children
When selected, it opens a **small custom window**, not a context action.
This is intentional:

- It allows configuration (name enumeration).
- It avoids accidental execution.

3. Internal State (Fields)

Object to Copy

This variable stores the **source object**:

- Can be a prefab from the Project view.
- Can be a scene object from the Hierarchy.

This object is **never modified**.
It is only used as a template to create new instances.

Enumerate Names

This is a simple on/off option.
When enabled:

- Every pasted child gets a numbered name.
- Example:
  - Object (1)
  - Object (2)
  - Object (3)

When disabled:

- All children keep the original object name.

The numbering is **global across all parents**, not per parent.

4. Editor Window Initialization

Window Creation

When the menu option is clicked:

- A custom editor window is created or reused.
- The window title is set.
- The size is locked so it cannot be resized.

This keeps the tool:

- Focused.
- Clean.
- Impossible to visually break.

5. User Interface Layout

The window contains four main elements:

1. **Title label**
   Clearly explains what the user should do.
2. **Object selector**
   - Lets you pick:
     - A prefab, or
     - A scene object.
   - Drag-and-drop supported.
   - Only accepts objects that can exist in a scene.
3. **Enumerate toggle**
   - Enables or disables name numbering.
4. **Action button**
   - Disabled if no object is selected.
   - Prevents invalid usage.
   - Clearly labeled to describe exactly what will happen.

The interface prevents misuse before execution.

6. Main Operation Logic
When the user clicks **Paste As Child**, the tool performs several checks and actions in order.

Step 1: Read Selected Parents
The tool gathers all currently selected objects in the Hierarchy.
If:
- Nothing is selected → it stops and shows a warning.

This prevents:
- Accidental creation at the root.
- Confusing results.

Step 2: Validate Source Object
If no object was chosen to copy:
- The operation stops.
- A warning is shown.

This ensures the tool never runs in an undefined state.

Step 3: Prepare Tracking
The tool prepares:
- A list to track all newly created objects.
- A counter for name enumeration.

This allows:
- Final selection of results.
- Correct numbering.

Step 4: Process Each Parent
For every selected parent object:
a) Create a New Instance
The tool detects whether the source object is:
- A prefab asset, or
- A scene object.

Depending on that:
- Prefabs are instantiated in a prefab-safe way.
- Scene objects are duplicated normally.

Undo support is registered so the operation can be reversed cleanly.

b) Attach to Parent
The new object is:
- Assigned as a child of the current parent.
- Reset to default local position.
- Reset to default local rotation.
- Reset to uniform scale.

This guarantees:

- No unexpected offsets.
- No inherited scale issues.
- Predictable placement.

c) Apply Name Enumeration (Optional)
If enumeration is enabled:
- The object name is changed.
- A unique number is appended.

The number increases globally, not per parent.

d) Register Undo and Track Result
Each created object:
- Is registered with the Undo system.
- Is added to the internal result list.

This ensures:
- Full Undo support.
- Final selection works correctly.

e) Expand Hierarchy
After adding the child:
- The parent is automatically expanded in the Hierarchy.
- The new child becomes immediately visible.

This avoids the common “Where did it go?” moment.

Step 5: Error Handling
If anything goes wrong during creation:
- The error is caught.
- A message is logged.
- The tool continues processing other parents.

One failure does not cancel the entire operation.

Step 6: Final Selection
After all parents are processed:
- All newly created objects become selected in the editor.

This allows:
- Immediate batch editing.
- Quick inspection.
- Easy deletion or adjustment.

Step 7: Window Cleanup
Once finished:
- The tool window closes automatically.

This reinforces:
- One-shot usage.
- Clean workflow.
- No unnecessary UI clutter.

7. Hierarchy Expansion Helper
This helper exists for **editor convenience only**.
What it does:
- Forces the Unity Hierarchy window to expand a specific object.
- Makes newly created children visible instantly.

How it works conceptually:
- Accesses Unity’s internal hierarchy window.
- Calls a hidden expansion function.
- Uses defensive error handling.

If this fails:
- The operation still succeeds.
- Only auto-expansion is skipped.

This keeps the tool safe and non-blocking.

8. Typical Usage Scenario
Example workflow:
1. Select multiple character bones, props, or scene nodes.
2. Open the tool from the GameObject menu.
3. Drag a prefab (for example, a collider, helper object, or effect).
4. Enable name enumeration if needed.
5. Click **Paste As Child**.

Result:
- Each selected object receives a child.
- All children are visible.
- All changes are undoable.
- All new objects are selected.

9. Why This Tool Is Useful
This tool shines when:
- Rigging or adding helper objects.
- Adding repeated effects or markers.
- Preparing complex hierarchies.
- Working with MMD or imported rigs.
- Avoiding repetitive manual work.

It reduces:
- Human error.
- Time spent clicking.
- Hierarchy clutter confusion.

10. Design Strengths
- Clear UI.
- Safe execution checks.
- Full Undo support.
- Prefab-aware behavior.
- Automatic hierarchy visibility.
- Non-destructive workflow.

In short:
It's a precision productivity tool, not a brute-force shortcut.

Custom MMD Data Utility Editor

1. What This Script Is For
This utility exists to **manage a single shared CustomMMDData asset inside the Unity Editor**.
Its responsibilities are very focused:
- Make sure **one valid CustomMMDData asset always exists**.
- Automatically **find an existing asset** if it was already created.
- **Create the asset automatically** if it doesn't exist yet.
- **Clean invalid material references** inside that asset to keep it consistent and safe.

Think of it as a **guardian and entry point** for CustomMMDData, ensuring:
"Whenever something in the editor needs CustomMMDData, it will always get a valid one."

2. Scope and Context
- This script only runs in the **Unity Editor**, not in a built game.
- It is meant to be called by:
  - Other editor tools.
  - Importers.
  - Setup or conversion utilities.
- It is **not** a user-facing window or menu item.
- It works silently in the background.

3. Stored Variable (Internal Cache)

Cached CustomMMDData Reference
There is one stored reference that holds a previously found or created CustomMMDData asset.
Purpose of this cache:
- Prevents searching the entire project every time.
- Ensures all tools are working with the **same instance**.
- Improves performance.
- Avoids asset duplication.

Once this reference is set, the utility reuses it.

4. Main Public Entry Points
These are the methods other tools are expected to call.

Get or Create CustomMMDData
This is the **most important method in the script**.
What it guarantees:
- It always returns a valid CustomMMDData asset.
- It never returns “nothing”.

How it works conceptually:
1. First, it checks whether a cached instance already exists.
2. If not:
    - It searches the project for an existing CustomMMDData asset.
3. If the search finds nothing:
    - It creates a brand-new CustomMMDData asset.
4. The result is cached and returned.

Why this matters:
- Other editor tools don’t need to care whether the asset exists.
- They can safely call this method and continue working.

This design prevents:
- Duplicate data assets.
- Manual setup steps.
- Null reference errors.

Remove Invalid Materials
This method is about **data integrity**.
Its job:
- Clean up the material list inside a CustomMMDData asset.
- Remove entries that reference materials that no longer exist.

What problem it solves:
- Materials can be deleted, renamed, or lost during imports.
- The data asset may still reference them.
- Invalid references cause errors, warnings, or broken tools.

What it does conceptually:
- Scans the material list.
- Keeps only entries that still point to a valid material.
- Silently removes broken entries.

It does nothing if:
- The data asset itself is missing.
- Everything is already valid.

This makes it safe to call repeatedly.

5. Internal Helper Logic (Private Responsibilities)
These methods are not meant to be called directly by other tools.

Finding an Existing CustomMMDData Asset
This step searches the project database for assets of the correct type.
How it behaves:
- Looks for assets labeled as CustomMMDData.
- If at least one is found:
    - Loads the first one.

- If none are found:
    - Reports failure internally.

Why it works this way:

- Enforces a **single authoritative data asset**.
- Avoids ambiguity.
- Keeps the system simple.

Creating a New CustomMMDData Asset

This method handles asset creation in a safe and predictable way.

What it does conceptually:

1. Ensures a known folder exists where the asset should live.
2. Generates a unique name so it never overwrites anything.
3. Creates a new CustomMMDData instance.
4. Saves it to disk.
5. Refreshes the editor so Unity recognizes it.

Why this matters:

- No manual asset creation required.
- No naming conflicts.
- Works in fresh projects automatically.

6. Typical Usage Flow

Here's how this utility is expected to be used by other tools:

1. An editor tool needs access to CustomMMDData.
2. It calls **Get or Create CustomMMDData**.
3. It receives a valid asset every time.
4. Before using material data, it may call **Remove Invalid Materials**.
5. The tool proceeds safely with clean data.

No setup steps.

No assumptions.

No fragile dependencies.

7. Design Intent and Strengths

This script is designed around **editor safety and automation**.

Key strengths:

- Single source of truth.
- Zero manual setup.
- Defensive programming.
- Safe repeated calls.
- Asset integrity maintenance.
- Clean separation of responsibilities.

It behaves like:

A silent infrastructure service that keeps editor tools stable.

8. Why This Script Is Important

Without this utility:

- Multiple tools might create duplicate data assets.
- Some tools might fail if the asset doesn't exist.
- Broken material references could accumulate silently.

With this utility:

- The editor environment stays clean.
- Tools become simpler.
- Errors are prevented rather than patched.

Header Group GUI

1. What This Script Is For

This script provides a **reusable collapsible header group** for:

- Custom Inspectors
- Custom Editor Windows

Its main purpose is to let you visually group settings under a header that can be **expanded or collapsed**, while also **remembering its open/closed state** between editor sessions.
In short, it solves three common editor problems:

1. Inspectors getting too long and cluttered
2. Foldout states resetting every time the editor refreshes
3. Repeating foldout logic across multiple tools

This utility centralizes all of that into a **single, lightweight helper**.

2. Where and When It Is Used
This script is:

- Editor-only
- Not attached to GameObjects
- Not visible to end users

It is designed to be **called from inside other editor tools**, such as:

- Custom inspectors
- Custom property drawers
- Editor windows

You don't "open" this tool — you **use it as a building block**.

3. Internal State Storage (Foldout Cache)
Foldout State Dictionary
The script keeps an internal table that maps:

- A group's unique name
  → whether that group is currently expanded or collapsed

Why this exists:

- Accessing editor preferences repeatedly is slow.
- This cache avoids unnecessary reads.
- It ensures instant responsiveness when drawing inspectors.

What it represents conceptually:
"For each group header, remember if the user left it open or closed."
This state is shared across all uses of the same group name.

4. Main Public Feature (What You Actually Use)
Draw Group
This is the **only method most users ever interact with**.
Its job:

- Draw a header bar
- Allow it to be expanded or collapsed
- Persist its state automatically
- Draw content only when expanded

You provide two things:

1. The header's label (text + optional tooltip)
2. A block of logic that draws the contents inside the group

The script handles everything else.

What Happens Internally When a Group Is Drawn
Step by step, conceptually:

1. The script checks whether this group was previously open or closed.
2. It calculates the correct size and position for the header.
3. It draws:
   - A subtle top border
   - A lightly shaded background
   - A bold foldout label
4. It detects whether the user clicks to expand or collapse the group.
5. If the state changes:
   - The new state is saved immediately.
6. If the group is open:
   - The provided content-drawing logic is executed.
   - The layout is slightly indented for clarity.

This all happens every time the inspector redraws.

5. Visual Design Choices (Why It Looks the Way It Does)
The header is intentionally styled to be:
- Minimal
- Lightweight
- Non-intrusive
- Consistent with Unity's inspector style

Design details:
- Subtle background tint instead of a heavy box
- Thin separator line for visual grouping
- Bold text for clarity
- Full-width layout for clean alignment

The goal is clarity without visual noise.

6. Persistent State Handling (Why It Remembers)
Retrieving a Group's State
When a group is drawn:
- The script first checks its internal cache.
- If the state isn't cached yet:
  - It retrieves the saved value from editor preferences.
  - Defaults to "open" if no value exists.
- The result is stored in memory for fast reuse.

This ensures:
- First-time groups start open
- User preferences are respected forever

Saving a Group's State
When the user toggles a group:
- The new state is written to:
  - The internal cache (for performance)
  - Editor preferences (for persistence)

This means:
- Closing and reopening Unity preserves layout
- Reloading scripts does not reset foldouts

7. Unique Group Identification
Each group's state is saved using a unique key derived from its name.
Why this matters:
- Multiple inspectors can reuse this system safely
- Each group remembers its own state
- No conflicts between unrelated tools

The group name effectively becomes:
The identity of that foldout across the editor

8. Typical Usage Pattern
A common workflow looks like this:
1. You create a custom inspector or editor window.
2. You want to group related settings together.
3. You call this utility and pass:
   - A label like "Rendering Settings"
   - A function that draws those settings
4. The group appears collapsible.
5. Unity remembers the user's preference automatically.

No extra setup.
No manual preference handling.
No duplicated foldout logic.

9. Why This Script Is Useful

Without this utility:
- Each inspector reimplements foldouts.
- Foldout state is often lost.
- Code becomes repetitive and inconsistent.

With this utility:
- Inspectors stay clean and readable.
- Users control what they see.
- State persistence feels native and polished.
- Editor tools scale better as they grow.

This script is a **quality-of-life infrastructure piece**.

10. Design Philosophy

This tool follows a few strong principles:
- One responsibility: collapsible groups
- Zero editor clutter
- Fast redraws
- Persistent UX
- Reusable everywhere

It doesn't try to do too much — and that's exactly why it's effective.

Material Utility

High-level purpose

This script is a **shared toolbox for custom material inspectors** used in MMD shaders inside the Unity Editor.

Instead of each material inspector re-implementing the same UI logic (surface options, blending, textures, toggles, render queues, metadata, etc.), this utility centralizes everything in one place.

What this gives you:
- Consistent UI across all MMD material inspectors
- Correct handling of multi-material editing
- Undo/Redo safety
- Automatic synchronization between UI, shader properties, keywords, render queues, and passes
- Cleaner and smaller custom inspectors

Think of this class as a **material inspector framework**, not a single feature.

1. Custom MMD Data Handling

Purpose

This section handles **extra metadata that does not belong to the shader**, such as:
- Japanese material name
- English material name
- Memo/notes
- Whether the inspector should fall back to Unity's default material inspector

This data is stored externally in a **custom MMD data container**, not inside the material itself.

LoadData

What it does
- Reads previously saved metadata for a material
- Outputs the stored values so the inspector can display them

How it works conceptually
1. Start with safe defaults (empty strings, disabled flags)
2. If either the data container or material is missing → stop
3. Read the global "use default inspector" flag
4. Look through the stored material list
5. If a matching material is found:
    - Copy its stored names and memo
    - Stop searching immediately

Why this matters
- Prevents null errors
- Allows multiple materials to share one data source

- Keeps editor UI responsive and predictable

SaveData
What it does
- Writes material metadata back into the shared data container

How it works
1. Abort if inputs are invalid
2. Store the global inspector mode flag
3. Search for an existing entry for the material
4. If found → update it
5. If not found → create a new entry

Why this matters
- Prevents duplicate entries
- Keeps metadata synchronized with materials
- Allows safe incremental updates

2. Global Illumination Control
RenderLightmapFlags
Purpose
Controls how materials contribute to **global illumination**:
- None
- Realtime
- Baked
- Emissive only

Key behaviors
- Supports editing multiple materials at once
- Detects when selected materials have different GI modes
- Shows Unity's "mixed value" state when needed
- Applies changes to all materials consistently
- Fully supports Undo/Redo

Why it exists
Unity's default GI UI is not always exposed or clear in custom inspectors.
This method ensures GI behavior is always visible and editable.

3. Basic Property Drawers
These methods are **building blocks** for material inspectors.
They all share common goals:
- Compact layout
- Multi-material safety
- Mixed value detection
- Undo support

ColorProperty
What it draws
- A color picker
- Optional alpha channel
- Optional HDR values
- Individual numeric controls for R, G, B, (A)

Why this design
- Artists often want precise numeric control
- Color pickers alone are not always enough
- Per-channel control avoids guesswork

FloatProperty
What it draws
- A single numeric input field

Why it exists
- Many shader parameters are simple scalars
- This avoids boilerplate code in inspectors

FloatSliderProperty
What it draws
- A slider constrained to a specific range

Why it exists
- Prevents invalid values
- Makes artist intent clearer
- Ideal for normalized values like thresholds or intensities

VectorProperty
What it draws
- A vector with 2, 3, or 4 components

Why it exists
- One unified method handles UVs, directions, masks, and custom data
- Avoids separate implementations for each vector size

4. Toggle & Keyword Controls
This section handles **boolean-style options**, even when the shader stores them as numbers, passes, or keywords.

DoubleToggleProperty
Purpose
Controls whether the material renders:
- One side only
- Both sides

Key behavior
- Interprets render face settings
- Warns when values are inconsistent
- Uses visual cues (color and enable/disable states)
- Forces correct values when toggled

Why this matters
Double-sided rendering is a common MMD requirement and easy to misconfigure.

FloatToggleProperty
Purpose
Turns a numeric property into a simple on/off toggle
Why it exists
Many shaders store booleans as numbers for performance or compatibility.

PassToggleProperty
Purpose
Enables or disables a **specific shader pass** (for example, shadow casting)
Key behavior
- Works across multiple materials
- Detects mixed pass states
- Applies changes safely

KeywordToggleProperty
Purpose
Links a toggle to:
- A numeric property
- A shader keyword

Why it's powerful
- Keeps shader variants in sync
- Supports inverted keywords
- Prevents keyword/property mismatches

5. Dropdown Controls
These methods turn numeric shader values into **human-readable choices**.

FloatDropdownProperty
What it does
- Maps display strings to numeric values
- Behaves like an enum selector
- Supports mixed material editing

Why it exists
Shaders don't have real enums, but inspectors should.

SurfaceTypeDropdownProperty
Purpose
Switches between:
- Opaque
- Transparent

What it automatically updates
- RenderType tags
- Render Queue
- Transparency consistency

Why it's important
Surface type affects:
- Sorting
- Depth writing
- Blending
- Performance

This method ensures all of that stays correct.

BlendingModeDropdownProperty
Purpose
Controls how transparent materials blend with the background
What it manages
- Blend mode selection
- Source blend factor
- Destination blend factor

Why it matters
Incorrect blending causes:
- Dark halos
- Incorrect transparency
- Visual artifacts

This method enforces valid combinations.

DepthWriteDropdownProperty
Purpose
Controls how the material writes to the depth buffer
Modes
- Automatic
- Forced enabled
- Forced disabled

Extra behavior
- Automatically toggles the DepthOnly pass
- Keeps depth logic consistent with surface type

6. Texture Controls
TextureProperty
What it draws
- Texture slot (regular or cubemap)
- Optional tiling and offset controls

Key features
- Mixed value detection
- Undo support

- Compact layout
- Optional advanced controls

DrawScaleOffsetInRect
Purpose
Draws tiling and offset controls in a precise layout area
Why it exists
- Unity's default layout is often too large
- This allows fine UI placement
- Supports mixed values across materials

7. Surface Options Composite UI
DrawSurfaceOptions
Purpose
This is a **ready-to-use surface options block**, similar to URP's standard inspector.
What it includes
- Surface type
- Blending (only when transparent)
- Face culling
- Depth write
- Depth test
- Alpha clipping
- Alpha cutoff (conditional)
- Shadow casting
- Shadow receiving

Why this is important
All MMD materials behave consistently and predictably without repeating code.

8. Render Queue Conversion
ConvertRenderQueueToTransparent
Purpose
Moves a material's render queue into the transparent range while preserving order
Why
Prevents sorting issues when switching surface types.

ConvertRenderQueueToOpaque
Purpose
Restores a transparent queue back to a solid rendering range
Why
Keeps render order stable when toggling transparency off.

9. Validation & Helpers
HasFloatPropertyValue
Purpose
Checks whether **all selected materials** share the same numeric value
Why
Used to decide whether dependent UI should appear.

CheckBlendingMode
Purpose
Ensures blending factors remain valid based on:
- Surface type
- Selected blending mode

Why
Prevents invalid or contradictory shader states, even when values are changed indirectly.

Usage Summary
You use this class when writing **custom material inspectors**:
- Call individual helpers for specific controls

- Or call DrawSurfaceOptions to get a full, standardized surface UI
- Use metadata helpers to store MMD-specific info
- Rely on automatic syncing instead of manual fixes

Result:

- Less code
- Fewer bugs
- Consistent artist experience
- Correct rendering behavior by default

MMD Material Custom Inspector Base

High-level purpose

This script is a **base inspector framework** for **custom MMD material inspectors** inside Unity.
Instead of every MMD shader having its own completely separate inspector logic, this class centralizes **shared behavior**, such as:

- Handling **MMD-specific metadata** (Japanese name, English name, memo).
- Supporting **multi-material editing** cleanly.
- Providing **Undo / Redo safety**.
- Managing **custom collapsible UI sections**.
- Allowing a **fallback to Unity's default material inspector**.
- Applying **shared and shader-specific rendering rules**.

Any custom MMD material inspector is expected to **inherit from this class**, then add only the shader-specific UI and logic.
Think of it as a **foundation layer** that guarantees consistency, safety, and UX across all MMD shaders.

Context section (editor state)

These variables store references Unity provides while the inspector is active.

materialEditor

This is Unity's internal editor controller for materials.
It is used to draw standard material fields, rendering options, and to trigger repainting.
You don't create it — Unity gives it to you when the inspector is drawn.

properties

This is the full list of shader properties belonging to the material(s) being inspected.
It allows:

- Reading shader values.
- Drawing surface options.
- Applying rendering rules consistently.

It is cached so derived inspectors don't have to request it repeatedly.

targets

This represents **everything currently selected** in the inspector.
Usually this is:

- One material
- Or multiple materials when multi-selection is active

This array is essential for:

- Detecting mixed values.
- Applying changes to all selected materials safely.

MMD custom data section

This block handles **MMD-specific metadata**, which Unity materials do not support natively.

mmdData

A shared data asset that stores:

- Japanese material name
- English material name
- Memo text
- Inspector mode flags

This data is **external to the material**, allowing richer metadata without polluting shader properties.

nameJP
Cached Japanese material name.
Used for:
- Display in the inspector
- Export or MMD compatibility logic

Cached locally for performance and mixed-value detection.

nameEN
Cached English material name.
Purely for readability and workflow convenience.

memo
A free-form notes field.
Used for:
- Documentation
- Artist notes
- Technical reminders

Fully Undo-safe and multi-object aware.

showDefault
Controls whether:
- The **custom MMD inspector** is shown
  or
- Unity's **default material inspector** is used instead

This is extremely useful for debugging or advanced tweaking without removing custom logic.

Main inspector flow (OnGUI)
This is the **entry point** — it runs every time the inspector is drawn or refreshed.
What happens step-by-step
1. **Cache Unity references**
   - Material editor
   - Shader properties
   - Selected targets
2. **Ensure MMD data exists**
   - Loads or creates the shared MMD data asset
   - Reads initial values from the first selected material
3. **Register Undo / Redo handling**
   - Ensures inspector refreshes correctly after undo operations
4. **Begin change tracking**
   - Unity starts watching for user edits
5. **Draw the MMD metadata header**
   - Japanese name
   - English name
   - Memo
   - Default inspector toggle
6. **Choose UI mode**
   - If custom mode is active:
     - Draw shader-specific UI
     - Draw shared surface options
     - Draw advanced rendering options
   - If default mode is active:
     - Show Unity's default inspector inside a collapsible group
7. **Apply rules only if something changed**
   - Synchronizes blending modes
   - Applies common rendering rules
   - Applies shader-specific rules

This ensures:
- No unnecessary recalculation
- Clean, predictable behavior

Undo / Redo handling
Why this exists
Unity's Undo system does not automatically refresh custom cached data.
Without this:
- Values may appear wrong after undo
- Inspector UI may desync from actual data

EnsureUndoHook
Guarantees that:
- Undo callbacks are registered **once**
- No duplicate listeners are created

This prevents memory leaks and repeated refresh calls.

OnUndoRedo
Triggered when the user presses Undo or Redo.
It:
- Clears cached MMD data
- Forces the inspector to repaint
- Forces scene and editor refresh

This guarantees that:
- The inspector always reflects the real state
- No stale data survives Undo

Header UI (MMD Data section)
Purpose
This section exposes MMD-specific metadata in a structured, collapsible UI.

Default inspector toggle
A switch that controls whether:
- The custom MMD UI is used
- Or Unity's default inspector is shown

When toggled:
- Changes are recorded for Undo
- The value is saved per material
- All selected materials are updated consistently

Material name fields
Two side-by-side fields:
- Japanese name
- English name

Features:
- Supports mixed values
- Fully Undo-safe
- Applies changes to all selected materials

Memo UI
Purpose
Provides a large, editable notes area tied to MMD data.
Features:
- Multi-material safe
- Mixed-value aware
- Undo-safe
- Stored externally from the material

This is ideal for:
- Documentation
- Export notes
- Collaboration workflows

Mixed-value string handling
Why this exists
When multiple materials are selected:
- Values may differ
- Unity needs to show a “mixed” state

This method:
- Detects differences across selections
- Shows the mixed indicator correctly
- Applies edits to all selected materials uniformly

It is reused for:
- Japanese name
- English name
- Any future string-based metadata

Custom UI extension points
DrawUI
This is where **shader-specific controls** go.
Derived inspectors override this to draw:
- Custom sliders
- Toggles
- Texture slots
- MMD-specific parameters

The base class does nothing here by default.

Rendering rules system
ApplyCommonRules
Intended for logic shared across all MMD shaders, such as:
- Enforcing blend mode consistency
- Fixing incompatible settings
- Syncing hidden properties

ApplyShaderSpecificRules
Reserved for per-shader logic, such as:
- Automatically adjusting keywords
- Forcing render queue changes
- Enabling/disabling features based on values

Both methods are only called **when something actually changes**.

Custom data loading
EnsureCustomDataLoaded
This method guarantees that:
- The shared MMD data asset exists
- Invalid references are cleaned
- Cached values are initialized from the selected material

It only runs once per inspector lifecycle unless explicitly invalidated.
This avoids:
- Redundant asset loading
- Performance issues
- Inconsistent data states

Intended usage
How you’re supposed to use this class
1. Create a new custom material inspector
2. Inherit from this base class
3. Override:
    - DrawUI for shader controls
    - ApplyShaderSpecificRules if needed
4. Let the base class handle:
    - Metadata

- Undo
- Multi-selection
- Default inspector fallback
- Rendering rules

Mental model summary
This class is:
- A **stable spine** for all MMD material inspectors
- A **UX consistency layer**
- A **data safety net**
- A **workflow accelerator**

It deliberately separates:
- Metadata
- UI
- Rules
- Unity plumbing

So shader inspectors stay clean, focused, and predictable.

MMD Material SG

High-level purpose
This script is a **custom material inspector** specifically designed for an **MMD Shader Graph material**.
Its goals are:
- Organize a complex MMD shader into **clear, logical UI sections**.
- Provide **artist-friendly controls** for colors, textures, outlines, shadows, and effects.
- Automatically **synchronize rendering behavior** (transparency, blending, shadows) with the shader's internal keywords.
- Integrate seamlessly with the **shared MMD inspector framework** you built earlier.

This inspector is not standalone — it **extends the MMD base inspector**, inheriting metadata handling, undo safety, multi-material support, and default-inspector fallback.

Relationship to the base system
This inspector inherits everything from the **MMD material inspector base**, which means:
- Japanese name, English name, memo, and default-GUI toggle are already handled.
- Undo / Redo safety is automatic.
- Multi-material editing works by default.
- Shared rendering rules will always be applied at the right time.

This script focuses only on:
- **How the UI looks**
- **How this specific shader behaves**

Inspector UI section (DrawUI)
This is where the **entire visible interface** of the shader is defined.
The UI is divided into **collapsible groups**, each representing a logical concept rather than raw shader parameters.

Material Color group
Purpose:
Controls how the surface looks under light.
What it exposes conceptually:
- **Diffuse color**
  The base color of the surface.
- **Specular color**
  The color of highlights and shiny reflections.
- **Ambient color**
  How the material responds to indirect or environmental lighting.
- **Opacity (Opaque slider)**
  Controls transparency by adjusting alpha.
- **Reflection (Shininess)**

Controls how strongly the surface reflects light.

Why this group exists:
Artists expect all “how does it look” values in one place, not scattered across technical settings.

Rendering group
Purpose:
Controls how the material interacts with lighting, shadows, and geometry.
Key concepts:

- **Double-sided rendering**
  Whether the material is visible from both sides of the mesh.
- **Shadow casting (G-SHAD)**
  Determines if the object casts shadows onto other objects.
- **Shadow receiving (S-MAP)**
  Determines if the object receives shadows from others.
- **Self-shading (S-SHAD)**
  Controls whether the object shades itself or only reacts to its own lighting.

Why this group exists:
These settings directly affect performance, realism, and correctness — they need to be obvious and grouped.

Edge (Outline) group
Purpose:
Controls the MMD-style outline effect.
Conceptual behavior:

- Toggle for enabling/disabling outlines.
- Control over outline thickness.
- Control over outline color and transparency.

Important detail:
This entire section is currently **disabled**, meaning:

- The UI shows the settings.
- The user cannot modify them yet.

This is useful for:

- Future implementation
- Visual documentation
- Preventing broken or unsupported features

Texture / Memo group
Purpose:
Central hub for textures, special effects, and material notes.
Concepts covered:

- **Effect mode selector**
  Chooses how special sphere-based effects behave:
  - Disabled
  - Multiplicative glow
  - Additive glow
  - Sub-texture mode
- **Main texture**
  The primary image applied to the material.
- **Toon texture**
  Controls stylized shading and highlights.
- **Conditional texture logic**
  - If “Sub-Tex” mode is active:
    - User chooses a UV layer
    - A sub-texture is exposed
  - Otherwise:
    - A sphere or cube reflection texture is used
- **Memo field**
  Uses the shared MMD metadata system to store notes.

Why this group exists:

This mirrors real MMD workflows where textures and notes are tightly linked.

Custom Effects Settings group
Purpose:
Advanced fine-tuning of lighting and shading behavior.
Concepts exposed:

- Specular intensity
- Sphere reflection opacity
- Shadow brightness
- HDR intensity
- Toon shading tone vector
- Multiple light support
- Fog interaction

These settings are:

- Power-user focused
- Optional
- Non-destructive

They live in their own group to avoid overwhelming casual users.

Common rendering rules section
This is **the most important technical part** of the script.
Why this exists
Shader Graph materials rely on:

- Numeric properties
- Shader keywords

If these get out of sync, the material behaves incorrectly.
This method guarantees:

- What the user sees in the inspector
- Always matches how the shader actually renders

Transparency & blending logic
Conceptual behavior:

- If alpha clipping is enabled:
  - Alpha testing is activated in the shader
- If the surface is transparent:
  - Transparency keywords are enabled
- Blend mode determines alpha behavior:
  - Alpha → normal transparency
  - Premultiply / Additive → special handling
  - Multiply → conditional alpha modulation

This prevents:

- Incorrect blending
- Washed-out materials
- Broken transparency

Shadow synchronization logic
Conceptual behavior:

- A single “receive shadows” toggle controls:
  - Internal float flags
  - Multiple shader keywords
  - Main light shadows
  - Additional light shadows

When shadows are disabled:

- All related shadow keywords are turned off
- Internal flags are updated consistently

When shadows are enabled:

- Everything is turned back on in sync

This avoids:

- Ghost shadows

- Performance waste
- Visual mismatches

Multi-material awareness
All rules are applied to:
- Every selected material
- Independently
- Safely

No assumptions are made about single selection.

Utility logic
Keyword control helper
Purpose:
Centralizes how shader features are turned on and off.
Instead of duplicating logic everywhere:
- One helper ensures keywords are always toggled correctly

This improves:
- Readability
- Consistency
- Maintenance

Shader-specific rules section
Current state
This method exists but does nothing right now.
Why it exists anyway
It provides a **future-proof hook** for:
- Shader-unique constraints
- Automatic fixes
- Conditional feature enforcement

Other MMD shaders can override this without touching shared logic.

How this inspector is meant to be used
Typical workflow
1. Assign the MMD Shader Graph shader to a material.
2. Select the material in the Inspector.
3. This custom inspector automatically appears.
4. Artist:
    - Adjusts colors and textures
    - Enables effects
    - Tunes shadows and lighting
5. Technical rules are applied automatically in the background.
6. MMD metadata is preserved across sessions and Undo/Redo.

Mental model summary
This script is:
- A **presentation layer** for a complex shader
- A **rule enforcer** that prevents invalid states
- A **bridge** between artist intent and shader correctness
- A **specialized extension** of your MMD inspector ecosystem

It keeps **creative controls simple**, while **technical complexity stays hidden and synchronized**.

MMD Material ASE

Overall purpose of the script
This script defines a **custom material inspector** specifically for **MMD materials created with Amplify Shader Editor (ASE)**.
Its main goals are:
- Present a **clean, MMD-oriented UI** instead of a raw shader parameter list.
- Group material controls into **logical sections** that match how artists think.

- Integrate with the **shared MMD inspector framework** (metadata, undo safety, multi-material editing).
- Keep **shader keywords and rendering behavior synchronized** with the inspector values.
- Avoid exposing unnecessary or confusing ASE internals.

In short:

👉 *This is the "artist-facing control panel" for ASE-based MMD materials.*

Inheritance and context

This inspector **inherits from the common MMD material inspector base**.
That means it automatically gets:

- Support for:
  - Japanese name
  - English name
  - Memo text
  - "Show Default Systems" toggle
- Multi-material editing
- Undo / Redo safety
- Shared infrastructure for rendering rules
- Optional fallback to Unity's default material inspector

This script **only focuses on ASE-specific UI and rules**, not shared infrastructure.

Inspector UI section (DrawUI)

This method defines **everything the user sees** when selecting an ASE-based MMD material.
The UI is split into **collapsible groups**, each representing a conceptual category.

Material Color group

Purpose:

Controls the visual identity of the surface.

Conceptual controls:

- **Diffuse**
  The base surface color.
- **Specular**
  The color of highlights and reflective shine.
- **Ambient**
  How the material responds to ambient or indirect light.
- **Opaque (Alpha)**
  Controls how transparent the material is.
- **Reflection**
  Controls the intensity of reflective highlights.

Why it exists:

These are the most frequently adjusted artistic properties and should always be easy to find.

Rendering group

Purpose:

Controls how the material behaves in the render pipeline.

Concepts exposed:

- **Double-sided rendering**
  Determines whether both sides of the mesh are visible.
- **G-SHAD (shadow casting)**
  Controls whether this object casts shadows onto others.
- **S-MAP (shadow receiving)**
  Controls whether this object receives shadows.
- **S-SHAD (self shading)**
  Determines whether the object shades itself only.

Why this group exists:

These settings affect realism, performance, and correctness, and are critical in MMD workflows.

Edge (Outline) group

Purpose:

Controls the classic MMD outline effect.
Concepts exposed:
- Enable/disable outline rendering
- Outline thickness
- Outline color (including transparency)

Difference from Shader Graph version:
In the ASE version, **outline controls are fully active**, reflecting ASE's more direct handling of outline data.

Texture / Memo group
Purpose:
Handles textures, special effects, and metadata notes in one place.
Conceptual features:
- **Effect mode selector**
  - Disabled
  - Multi-sphere (multiplicative glow)
  - Add-sphere (additive glow)
  - Sub-texture (alternate UV-based effect)
- **Main texture**
- **Toon texture**
- **Conditional logic**
  - If Sub-Tex mode is active:
    - Select UV layer
    - Use sub-texture
  - Otherwise:
    - Use sphere or cube reflection texture
- **Memo field**
  - Stores freeform notes in shared MMD data
  - Works across multiple materials
  - Fully Undo-safe

Why memo is here:
In MMD workflows, textures and notes often go together (author notes, usage hints, export comments).

Custom Effects Settings group
Purpose:
Advanced fine-tuning for lighting and shading behavior.
Conceptual controls:
- Specular intensity
- Sphere-map opacity
- Shadow brightness
- HDR lighting intensity
- Toon tone vector
- Multiple light support
- Fog interaction

These controls are intended for **advanced users** and technical artists.

Common Rendering Rules section
This section enforces **automatic consistency** between inspector values and shader behavior.
Why this exists
ASE shaders often rely on:
- Float parameters
- Shader keywords

If these fall out of sync, the material can render incorrectly.
This method ensures:
- Inspector intent → shader behavior
- Always consistent
- Always safe

What this implementation does

For every selected material:

- Checks whether **alpha clipping** is enabled.
- Enables or disables the **alpha test shader keyword** accordingly.

This is intentionally minimal compared to the Shader Graph version, because:

- ASE shaders often already manage blending internally.
- Over-aggressive rule enforcement could break user setups.
- This inspector aims to be **lightweight and predictable**.

Utility method (keyword control)
Purpose:
Centralizes how shader features are turned on or off.
Conceptually:

- If a feature should be active → keyword enabled
- If not → keyword disabled

This avoids:

- Duplicate logic
- Inconsistent keyword handling
- Hard-to-track rendering bugs

Shader-specific rules section
Current state:
Empty.
Why it still exists:

- Provides a **dedicated hook** for:
  - ASE-only constraints
  - Fixups for specific shader graphs
  - Future extensions

This keeps the architecture clean and scalable.

How this inspector is meant to be used
Typical workflow

1. Create or assign an **ASE-based MMD shader** to a material.
2. Select the material in the Inspector.
3. This custom inspector appears automatically.
4. Artist:
   - Adjusts colors and textures
   - Enables outlines and effects
   - Tunes shadows and lighting
5. The inspector:
   - Keeps shader keywords synchronized
   - Preserves MMD metadata
   - Supports Undo/Redo and multi-editing

Conceptual difference vs Shader Graph inspector
Compared to the Shader Graph version:

- **Simpler rule enforcement**
- **More direct outline control**
- **Less reliance on URP surface abstractions**
- **Closer alignment with traditional MMD + ASE workflows**

This makes it ideal for:

- Legacy pipelines
- ASE-heavy projects
- Artists who prefer explicit control

Mental model summary
This script acts as:

-  A **clean control surface** for ASE-based MMD materials
-  A **safety layer** that prevents keyword desync
-  A **modular extension** of the MMD inspector framework

- 🎨 A **workflow-first UI**, not a shader debug panel

MMD Material ASE T

1. What this script is and why it exists

This script defines a **custom material inspector** for a specific type of material:

- **MMD-style materials**
- Built using **Amplify Shader Editor (ASE)**
- With **tessellation support**

Instead of showing Unity's default material UI (which can be messy or too generic), this inspector:

- Organizes properties into **clear collapsible sections**
- Matches **MMD workflows and terminology**
- Keeps rendering behavior consistent (transparency, shadows, keywords)
- Exposes **advanced shader features** in a controlled, artist-friendly way

In short:

👉 *It turns a complex shader into a clean, structured, MMD-focused editor UI.*

2. Overall structure and inheritance

The class is built on top of a **shared base inspector** (MMDMaterialCustomInspectorBase).
That base class already provides:

- Access to material properties
- Utility drawing functions (sliders, toggles, textures, dropdowns)
- A lifecycle for applying rendering rules

This script **does not reinvent those tools** — it *assembles them* into a coherent UI and applies shader-specific logic where needed.

3. Inspector UI: how the material editor is built

The heart of the script is the **UI drawing method**, which defines *what the artist sees*.
Each section is wrapped in a **collapsible header group**, so the inspector stays readable even with many options.

3.1 Material Color group

This section controls the **core visual appearance** of the material.
It exposes:

- **Diffuse color** → the base color of the surface
- **Specular color** → highlight or shine color
- **Ambient color** → how the material reacts to overall lighting
- **Opacity slider** → controls transparency
- **Reflection strength** → how shiny or reflective the surface feels

Purpose:

- Matches classic MMD material parameters
- Centralizes all color and reflection controls in one place

3.2 Rendering group

This section controls **how the material behaves in the render pipeline**, not how it looks.
Here the user can:

- Enable or disable **double-sided rendering**
- Control **shadow casting**
- Control **shadow receiving**
- Enable **self-shadow-only behavior**

Purpose:

- Keep shadow behavior consistent with MMD expectations
- Prevent users from accidentally breaking lighting behavior
- Synchronize material toggles with shader keywords and passes

3.3 Edge (Outline) group

This section controls the **MMD-style outline effect**.
It allows:

- Turning the outline on or off

- Adjusting outline thickness
- Choosing outline color (including transparency)

Purpose:

- Provide anime-style edge control
- Keep outline logic visually isolated from other material settings

3.4 Texture / Memo group

This is one of the most flexible sections.
It controls:

- **Main texture**
- **Toon shading texture**
- **Sphere / cube maps (SPH)**
- **Special effects modes**
- Optional **sub-textures with alternate UV layers**
- A **memo area** for notes or metadata

Important behavior:

- The UI **changes dynamically** depending on the selected effect mode
  (for example, showing UV selection only when sub-texture mode is active)

Purpose:

- Support classic MMD texture workflows
- Keep advanced effects accessible but not overwhelming
- Allow artists to document material intent directly in the inspector

3.5 Custom Effects Settings group

This is the **advanced control panel**.
It includes:

- Specular intensity
- SPH opacity
- Shadow brightness control
- HDR lighting intensity
- Toon tone vectors
- Multiple light support
- Fog toggle
- Tessellation controls:
  - Edge length
  - Phong smoothing strength
  - Extrusion amount

Purpose:

- Expose powerful shader features safely
- Give fine control without forcing users into shader graphs
- Group all “expert” parameters in one place

4. Common Rendering Rules

This part runs **after the UI is drawn**.
What it does:

- Iterates over all selected materials
- Checks specific material properties
- Enables or disables shader features automatically

Example behavior:

- If alpha clipping is enabled in the UI, the corresponding shader keyword is activated
- If it’s disabled, the keyword is removed

Purpose:

- Ensure visual settings and shader behavior never fall out of sync
- Support multi-material editing reliably
- Centralize shared rendering logic across materials

5. Utility logic

The script includes a small internal helper whose only job is:

- Turning shader features **on or off** at the material level

Why this matters:

- Keeps the code readable
- Prevents duplicated logic
- Makes rendering rules easy to audit and extend later

6. Shader-specific rules section

This section is intentionally empty.
Why it exists:

- Acts as a **future extension point**
- Keeps the inspector architecture consistent with other shaders
- Makes it easy to add special logic later without refactoring

This is a design choice focused on **maintainability**, not necessity.

7. How this script is used in practice

From a user's perspective:

1. Create or select a material using the MMD ASE tessellation shader
2. Unity automatically displays this custom inspector
3. The artist:
    - Expands only the sections they need
    - Adjusts colors, textures, outlines, and effects
    - Never touches raw shader keywords or passes
4. The script:
    - Updates material properties
    - Synchronizes shader keywords
    - Keeps rendering behavior correct and predictable

8. Design philosophy behind this inspector

This script is built around a few clear principles:

- **MMD-first workflow**
- **Artist-friendly UI**
- **Clear separation of concerns**
- **Low cognitive load**
- **Future-proof structure**

It doesn't just expose shader parameters —
it **curates them into a workflow**.