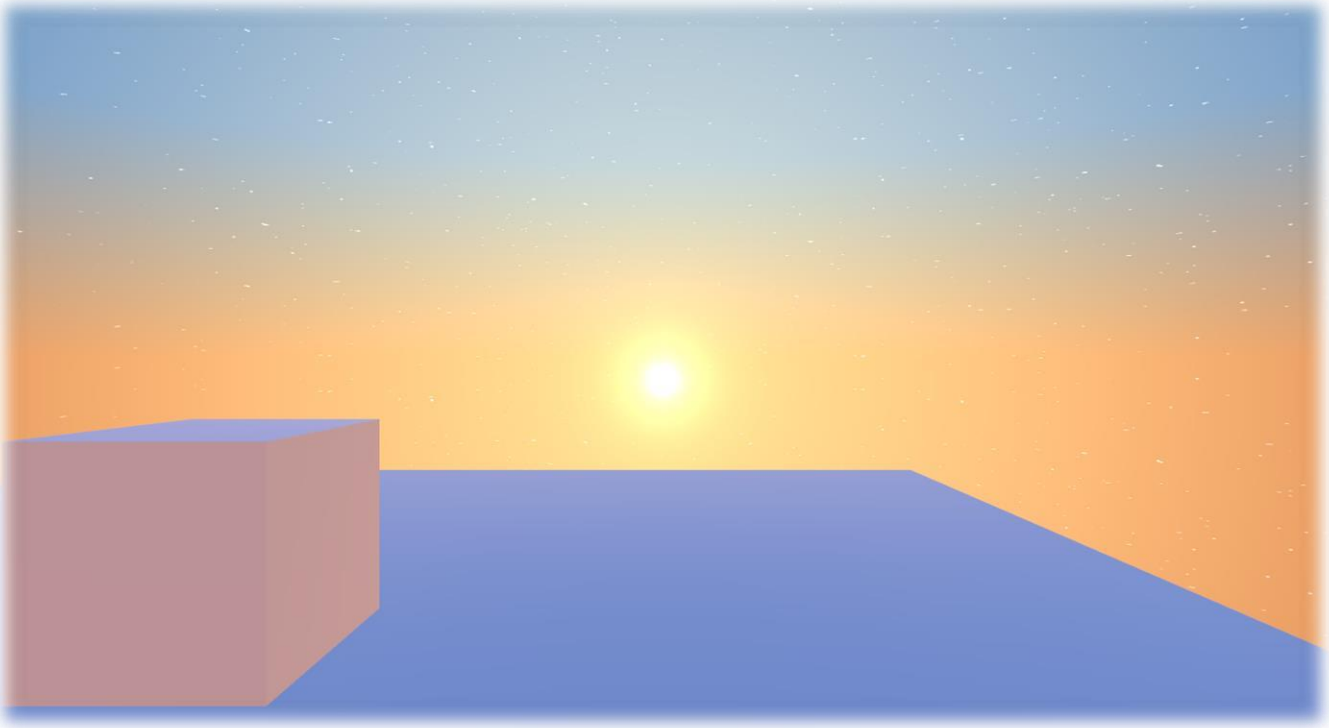


Skybox Universal RP

Documentation

_____

Creator: FlowingCrescent

Modified:

- Gabriel Henrique Pereira

- Lucas Gomes Cecchini **Pseudonym:** AGAMENOM

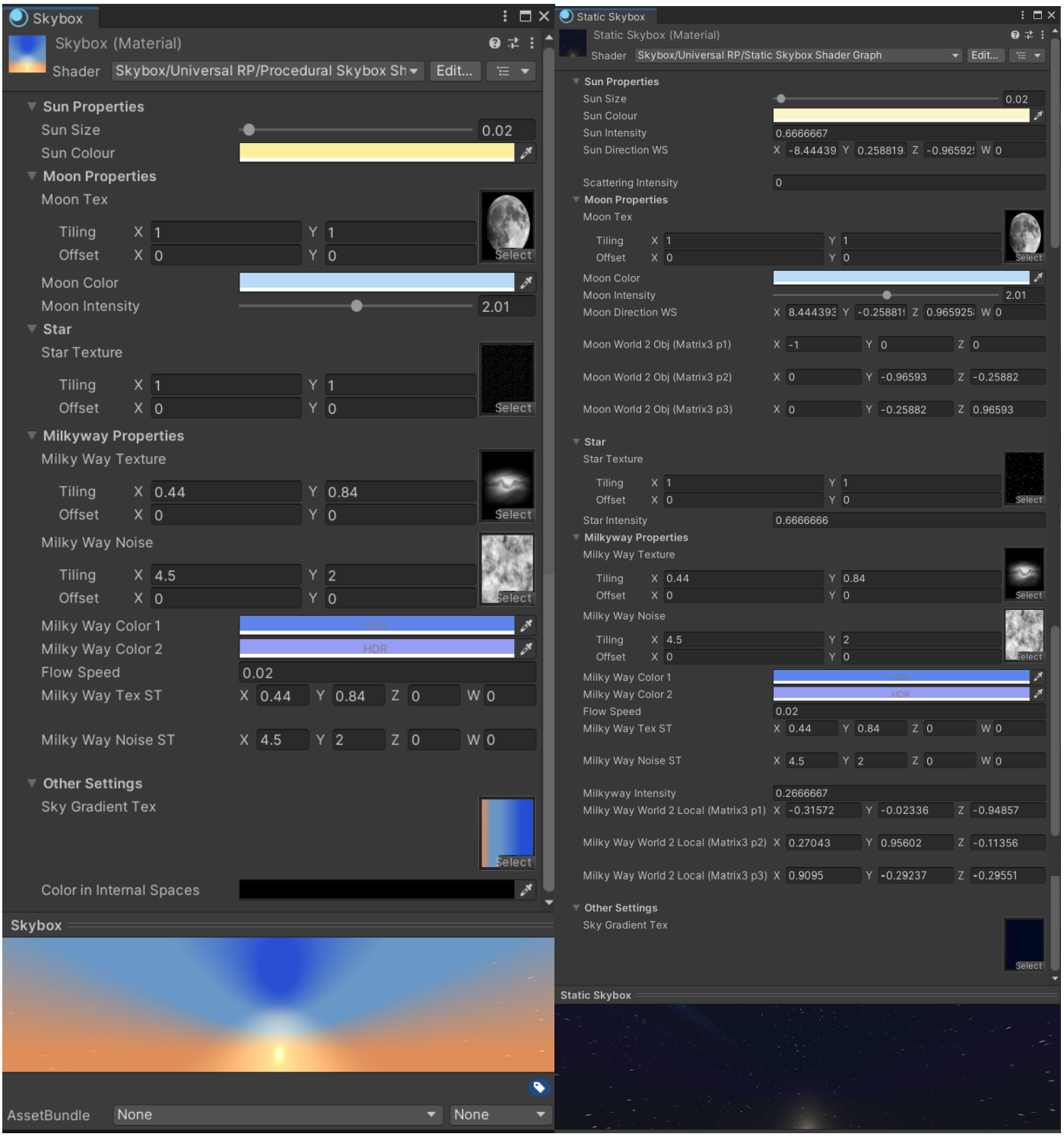
Simple Procedural Skybox

A simple procedural skybox which has day and night in unity universal render pipeline Unity version：6000.2.6f1, URP 17.2.0.

Please make sure your version of unity is similar for avoiding some errors due to version change.

Original File: https://github.com/FlowingCrescent/SimpleProceduralSkybox_URP

Shader Explanations



Skybox Shader

General Concept

This shader is a **procedural skybox generator**, meaning it doesn't rely on static textures or cubemaps. Instead, it uses **mathematical gradients, color blending, and lighting calculations** to create a dynamic sky that reacts to **time of day, lighting direction, and atmosphere properties**.

🌞 Shader Overview

The shader is divided into functional areas that work together to produce the final visual effect:

1. Properties (User Controls)

These are adjustable values visible in Unity's Material Inspector or controlled by scripts like the *SkyController*.

They determine how the sky looks and behaves:

- **Sky Tint / Gradient Colors** – Defines the main colors of the sky, from the horizon to the zenith (top of the sky).
  Users can create smooth transitions (blue midday sky, orange at sunset, etc.).
- **Sun Direction / Sun Intensity** – The position and brightness of the sun in the sky.
  It's often driven by the scene's directional light.
- **Scattering Intensity** – Controls how atmospheric light scattering affects the sky.
  Higher values produce softer gradients and more diffuse color, simulating thick atmosphere.
- **Star Intensity / Milky Way Texture** – Defines how bright stars and galactic features appear when the sky transitions to night.
- **Exposure or Brightness** – Adjusts overall light intensity to adapt to various lighting setups.

These variables are not just colors or numbers; they represent **inputs to mathematical formulas** that define the visual transitions across the entire sky.

2. Vertex Stage

This section defines the geometry of the skybox — usually a large sphere or cube surrounding the scene.

The shader ensures that the skybox remains **infinitely distant** from the camera (it doesn't move as the player does).

Conceptually, this stage transforms the geometry to always fill the camera view, creating the illusion of a distant horizon.

3. Fragment / Pixel Stage

Here is where most of the visual computation happens.

Each pixel on the screen (representing a piece of the sky) is colored according to the properties defined above.

The shader calculates the color for each pixel using:

- The **view direction** (where the camera is looking in the sky).
- The **sun's position** (used to compute lighting and highlights).
- The **atmospheric scattering formulas** (how light fades through the atmosphere).

The color at each point of the sky is the result of blending several layers:

- A **base gradient** for the sky colors.
- A **sun glow** region around the sun's direction.
- **Horizon tinting** to simulate thickness of the atmosphere near the ground.
- **Starfield overlay** that fades in as the sun intensity decreases.

4. Lighting and Scattering Simulation

This part conceptually mimics how sunlight interacts with the atmosphere.

It uses simplified physics-based ideas like *Rayleigh scattering* (light spread in gases) and *Mie scattering* (larger particle haze).

The effect creates:

- Bright blue tones near the sun during the day.
- Orange or red hues near the horizon at sunset.
- Deep blues and blacks when transitioning to night.

The scattering strength and direction are adjusted using the **Scattering Intensity** property, and the sun's color/intensity affects the result directly.

5. Stars and Milky Way
At night, the shader gradually fades in the star layer.
The **Star Intensity** controls how visible they are, while the **Milky Way Intensity** blends in a soft galactic texture or procedural noise pattern.
This transition typically happens based on the **sun's height** below the horizon — once the sun intensity or elevation drops below a threshold, stars appear.

6. Time of Day Interaction
The shader itself doesn't manage time directly, but it responds to variables provided by the **SkyController** or **SkyControllerMenu** scripts.
Those scripts adjust parameters like:

- Current time (e.g., 0 to 24 hours)
- Time progression speed
- Whether time flows forward or backward
- Brightness and color transitions over time

As these values change, the shader updates in real time, producing a smooth day/night cycle without texture swapping.

7. Usage in Unity
To use this shader:

1. Create a **Material** using this shader.
2. Assign it to your scene's **Skybox** slot (under *Lighting Settings → Environment → Skybox Material*).
3. Optionally, assign it to a **SkyController** script that animates its parameters automatically over time.
4. Adjust the **gradients**, **intensities**, and **textures** to match your desired atmosphere.

The two UI scripts (SkyControllerMenu and SkyControllerMenuTMP) serve as **front-end interfaces** for controlling these shader properties interactively in-game or in the editor.

Conceptual Summary

Component	Purpose	Effect on Visuals
Sky Color Gradient	Defines base colors across the sky.	Day-to-night transition hues.
Sun Direction	Indicates where sunlight originates.	Determines highlight direction and glow.
Scattering Intensity	Controls atmospheric diffusion.	Adjusts softness of horizon and sky fade.
Star Intensity	Controls visibility of stars.	Brighter at night, hidden during day.
Milky Way Texture	Adds galactic details.	Enhances night sky realism.
Time Variables	Drive dynamic updates.	Enable animated sky transitions.

In short:
The shader builds a **living sky** — it changes color, brightness, and texture automatically depending on **time, light direction, and atmosphere settings**.
It doesn't use pre-rendered textures but instead relies on **math-driven gradients and blending**, making it lightweight, continuous, and customizable.

Script Explanations

Sky Time Data Copier

🌤 General Purpose

The script adds a new **menu option inside the Unity Editor** under:
Assets → Create → Skybox URP → SkyTimeData Library
When the user clicks this option, the script looks for the *"SkyTimeData"* folder located within the project structure at
Assets/Skybox Universal RP/Database.
If the folder is found, it is **copied** into whichever folder the user has currently selected in the *Project Window*.
If no folder is selected, it defaults to placing it inside the main *Assets* directory.

🧩 Script Structure Overview

The script is divided into two main regions:

1. **Folder Selection** – Handles detection of which folder the user currently has selected in Unity.
2. **SkyTimeData Copying** – Handles the logic for finding, validating, and copying the *SkyTimeData* folder.

🔷 Folder Selection Region

Method: GetSelectedFolderPath()
This method determines where the copied folder will be placed.
Process description:

- It begins with a default path of "Assets".
  This ensures there is always a valid fallback destination.
- It then checks what the user currently has selected in Unity's **Project window**.
- If the selected item is a **folder**, it uses that folder's path directly.
- If the selected item is a **file**, it instead retrieves the **parent folder** where that file is located.
- If nothing is selected, it simply returns "Assets" as the destination.

This method ensures that no matter what the user has clicked in the Project view, the script can determine a proper and safe destination for the copied folder.
Variable involved:

- currentFolderPath: Temporarily stores the selected or fallback folder path.

🔷 SkyTimeData Copying Region

Method: CopySkyTimeData()
This is the **main method** of the script, and it is triggered when the user selects **"SkyTimeData Library"** from the Unity menu.
Let's break down its flow step by step:

Step 1 — Searching for the source folder
The method searches through the project for any folder named "SkyTimeData".
Variable used:

- guids: Holds unique identifiers (GUIDs) of all found folders that match the name *"SkyTimeData"*.

If the script cannot find any such folder, it logs an error message and stops the process.

Step 2 — Confirming the correct folder location
Even if multiple *SkyTimeData* folders exist, the script expects the correct one to be inside:

Assets/Skybox Universal RP/Database
To enforce that, it compares the path of each found folder to this parent path.
Variables used:

- requiredParent: Stores the expected directory path where the folder should be located.
- sourceFolderPath: Will hold the exact path to the *SkyTimeData* folder if it is found inside the correct directory.

If no valid path matches the expected parent folder, the process stops and reports an error.

Step 3 — Determining where to copy the folder
The script calls the GetSelectedFolderPath() method to find where the user wants to copy the folder.
Variable used:

- destinationFolderPath: The path to the folder that the user selected (or "Assets" if none was selected).

If the destination or source folder does not exist in the filesystem, the script logs an error and stops safely.

Step 4 — Preparing the destination path
It extracts just the name of the source folder (*SkyTimeData*) and appends it to the destination folder path.
Variables used:

- folderName: Stores only the last part of the path (in this case, "SkyTimeData").
- destinationPath: Combines the destination folder and folder name to form the final target location.

Step 5 — Copying the folder
The actual copy operation is performed using Unity's built-in file utility.
This copies the entire folder structure and its contents to the new location.
Once the copy is complete, the script forces Unity to **refresh the AssetDatabase** so that the copied files appear immediately in the Project window.
Finally, it displays a message confirming the successful copy and shows the path where the folder was placed.

🧠 Summary of Key Variables

Variable	Purpose
guids	Holds the unique identifiers of all found "SkyTimeData" folders in the project.
requiredParent	Defines where the correct source folder must be located (Assets/Skybox Universal RP/Database).
sourceFolderPath	Stores the exact path to the correct SkyTimeData folder.
destinationFolderPath	The path where the folder will be copied, based on user selection.
folderName	The name of the folder being copied ("SkyTimeData").
destinationPath	The full final destination path for the copied folder.
currentFolderPath	Used inside GetSelectedFolderPath() to hold the current selection path or default to "Assets".

🧭 Usage in Unity

1. In Unity's top menu bar, go to:
   Assets → Create → Skybox URP → SkyTimeData Library
2. Select a folder in the Project window where you want to copy the *SkyTimeData* files.
   If no folder is selected, the script will copy them to the main *Assets* directory.
3. The script will automatically locate the correct *SkyTimeData* folder under **Assets/Skybox Universal RP/Database** and duplicate it.

4. Once finished, the new *SkyTimeData* folder will appear in the destination you selected, ready for editing or customization.

📜 **In Summary**
This script is a **Unity Editor automation tool**.
It removes the need to manually locate and duplicate the *SkyTimeData* folder by handling the search, validation, and copying automatically.
Its purpose is to make it easier for developers using the **Skybox Universal RP** system to quickly generate working copies of the *SkyTimeData* library for testing or modification — directly from the Unity Editor menu, without having to manually browse file directories.

Sky Controller Create

🔷 **Overview**
This script is a **Unity Editor utility** designed to make it easier to create skybox-related GameObjects in a scene.
It adds new tools under the **Unity menu "GameObject → 3D Object → Skybox"**, allowing you to quickly create three types of objects:

1. A **Sky Controller**, based on a prefab stored in your project's database.
2. A **SkyController Trigger**, which includes a collider used for interaction or zone-based control.
3. A **Screenshot Skybox**, containing six cameras that capture the environment in all directions — useful for generating skybox textures.

Each menu option automatically sets up the object, applies necessary components, and ensures it integrates smoothly into your Unity scene.

🔷 **Script Organization**
The script is divided into **three logical regions**, each responsible for a specific type of object creation:

1. **Sky Controller Creation**
2. **SkyController Trigger Creation**
3. **Screenshot Skybox Creation**

Each section is explained in detail below.

🔷 **1. Sky Controller Creation**
Purpose
This part creates a **new Sky Controller GameObject** in the scene.
It uses an existing **prefab** (a preconfigured template) located in the folder Assets/Skybox Universal RP/Database.
The prefab is duplicated, unpacked (so you can edit it), and placed in your scene.
Step-by-Step Explanation

1. The tool looks through all assets in the project for a prefab named **"[Sky Controller]"**. If it can't find one, it shows an error message in Unity's Console and stops.
2. It verifies that the prefab is located inside the required folder — this prevents accidental use of incorrect files with the same name.
3. Once found, it loads that prefab into memory.
4. It checks if you have a GameObject selected in the scene hierarchy.
    - If yes, the new Sky Controller will be created as a **child** of the selected object.
    - If not, it will be placed **at the root level** of the scene.
5. The prefab is **instantiated** (a copy is created in the scene).
6. The script **renames** the new object to **"[Sky Controller]"** to maintain a consistent naming convention.

7. The prefab is then **unpacked**, meaning it becomes an independent GameObject that can be freely modified without affecting the original prefab.
8. Finally, the operation is recorded in Unity's **Undo system**, so you can undo it if needed.
9. A confirmation message appears in the Console indicating successful creation.

Result

A complete **Sky Controller** setup is added to your scene, ready to be customized.

🔷 2. SkyController Trigger Creation

Purpose

This part creates a **Trigger Object** used to detect when something enters a specific zone — for example, when transitioning between skyboxes or lighting conditions.
It's automatically configured with a collider and a specific script.

Step-by-Step Explanation

1. The script defines a **base name**: "[SkyController Trigger]".
2. It checks if an object with that name already exists in the scene.
    - If it does, it automatically generates a new name by adding a number, like "[SkyController Trigger] (1)", "[SkyController Trigger] (2)", and so on.
3. A new **empty GameObject** is created using the generated name.
4. If you have an object selected in the hierarchy, the new trigger will become its **child**.
5. The creation is recorded in the **Undo history**.
6. A **Box Collider** component is added automatically and marked as a **trigger**, meaning it can detect overlapping objects without acting as a physical barrier.
7. The script **SkyControllerTrigger** (a separate script not shown here) is then added to handle the logic when something enters or leaves the trigger zone.
8. The new trigger object is registered for Undo and a confirmation message appears in the Console.

Result

You get a new **trigger GameObject** that can be used to manage interactions such as skybox changes when a player enters a specific area.

🔷 3. Screenshot Skybox Creation

Purpose

This section creates a **Screenshot Skybox Object**, which includes **six cameras** arranged in different directions.
The setup allows you to capture **360-degree screenshots** of your skybox — useful for generating cube maps or testing environment visuals.

Step-by-Step Explanation

1. The base name is set to "[Screenshot Skybox]".
2. The script checks whether an object with that name already exists.
   If it does, it automatically generates a unique name by adding a counter number.
3. A **new parent GameObject** is created using the name.
4. If you have a GameObject selected, the new one becomes a **child** of that selection.
5. The creation is recorded in Unity's Undo system.
6. A **ScreenshotSkybox component** is added to the new object.
   This component is responsible for managing the capture process and will later use the cameras that the script creates.
7. The script defines a list of six camera directions:
    - Front
    - Left
    - Back
    - Right
    - Up
    - Down

8. For each direction:
    - A new GameObject is created with the corresponding name.
    - A **Camera component** is added and configured with:
        - A 90° field of view.
        - A near clipping plane of 0.01.
        - A far clipping plane of 1.0.
    - The camera is parented to the main Screenshot Skybox object.
    - Depending on its name, a **specific rotation** is assigned so that each camera faces one of the six cube map directions.
9. After all cameras are created, they are stored inside an array and assigned to the **ScreenshotSkybox component**.
10. The newly created Screenshot Skybox is automatically **selected** in the Unity Editor so you can see and configure it immediately.
11. A confirmation message is printed in the Console once the process is complete.

Result

You get a ready-to-use setup consisting of six cameras pointing in all directions.
This configuration can be used to capture full-environment images or generate texture data for custom skyboxes.

🔷 How to Use the Script in Unity

1. **Open Unity.**
   In the top menu, go to:
   GameObject → 3D Object → Skybox
2. **Choose what to create:**
    - **Sky Controller:** Creates and unpacks a prefab for managing your skybox system.
    - **SkyController Trigger:** Creates a trigger area for skybox transitions or effects.
    - **Screenshot Skybox:** Creates a camera rig for capturing skybox images.
3. **Edit or position the new object** as desired in the scene.
   Each creation is automatically registered in Undo, so you can revert easily if needed.
4. **Use them together if desired:**
   For example, you can have a Sky Controller managing the lighting and colors,
   a SkyController Trigger that changes the controller's behavior when the player enters a zone,
   and a Screenshot Skybox used for generating the skybox textures that appear in the scene.

🔷 Summary

This script acts as a **toolbox for skybox development** inside Unity.
It streamlines repetitive setup tasks by automating:

- Prefab duplication and unpacking for the main sky controller.
- Trigger zone creation for environmental interaction.
- Full 360° camera rig setup for skybox texture generation.

Main benefits:

- Saves time during skybox setup.
- Reduces manual configuration errors.
- Provides immediate integration with existing skybox systems.
- Works directly within Unity's editor workflow under familiar menus.

Essentially, it turns what would normally take several manual steps into **one-click operations**, making skybox management faster, cleaner, and more efficient.

Skybox Materials Creator

🔷 Overview

This script is a **Unity Editor utility** that helps you **create skybox materials automatically**.
It adds **two new menu options** inside Unity's **Assets > Create > Skybox URP > Material/** menu:

- **"Skybox Procedural"**
- **"Skybox Static"**

When one of these options is selected, the script **duplicates a predefined base material** and places the copy into the **currently selected folder** in the Project window.
It also ensures that the new material:

- Has a **unique name** (avoiding file overwrites).
- Is **automatically selected** in the Project window.
- Enters **rename mode**, so you can immediately rename it.

The purpose of this tool is to make skybox material creation **fast, consistent, and user-friendly** within Unity's workflow.

🔷 Script Structure

The script is divided into three logical regions:

1. **Folder Selection**
2. **Material Creation Core**
3. **Menu Items**

🔷 Region 1: Folder Selection

Method: GetCurrentFolderPath
This method determines **where the new material should be saved**.
Step-by-step logic:

1. By default, it assumes the path is **"Assets"**, which is the project's root folder.
2. It checks which folder or asset is currently selected in Unity's **Project window**.
3. If a **folder** is selected, it uses that folder's path directly.
4. If a **file** (like an image or material) is selected, it finds that file's **parent folder** instead.
5. It stops after processing the first selected item and returns the folder path.

Purpose:
To ensure that when you create a new skybox material, it appears exactly in the folder you currently have open or selected.

🔷 Region 2: Material Creation Core

Method: CreateSkyboxMaterial
This is the **central function** responsible for creating a new skybox material.
It takes two text parameters:

- The **name of the base material** to duplicate (either "Skybox Procedural" or "Skybox Static").
- The **default name** for the new material (e.g., "New Skybox Procedural").

Here's how it works step by step:

1. **Searching for the base material:**
   It searches all project assets for a material that matches the given base material name.
2. **Validating results:**
   If no material is found, an error is printed in the Unity Console, and the process stops.
3. **Locating the correct source folder:**
   The script only accepts base materials located inside the **"Assets/Skybox Universal RP/Database"** folder.
   This ensures that the duplicates are always created from the official base templates.
4. **Loading the base material:**
   Once the correct material is found, it is loaded into memory.
5. **Validating the load:**
   If loading fails for any reason, an error is shown, and the function stops.

6. **Creating the new material:**
   The script duplicates the base material, effectively cloning all of its properties.
7. **Determining the save location:**
   It uses the earlier method (**GetCurrentFolderPath**) to find where to save the new file.
8. **Generating a unique path:**
   To avoid overwriting existing materials, Unity's asset system generates a unique filename if one already exists.
9. **Saving the new material:**
   The duplicated material is saved as a new asset file in the selected folder.
10. **Updating the Asset Database:**
    All changes are saved and the asset list is refreshed so the new file appears immediately in the Project window.
11. **Selecting the new asset:**
    The newly created material is automatically selected in Unity's interface.
12. **Initiating rename mode:**
    Using Unity's delayed execution system, the script waits a moment and then triggers the rename action (as if the user pressed F2).
    This lets the user instantly rename the new material, improving workflow.
13. **Logging confirmation:**
    A message is printed in the Console confirming that the new skybox material has been created successfully.

🔷 Region 3: Menu Items

This section defines two public menu commands that call the material creation process.

Menu Option 1: "Skybox Procedural"

- Appears under: **Assets > Create > Skybox URP > Material > Skybox Procedural**
- When selected, it runs the creation process for a **procedural skybox** using the base material named "Skybox Procedural."
- The new material will have the default name **"New Skybox Procedural."**

Menu Option 2: "Skybox Static"

- Appears under: **Assets > Create > Skybox URP > Material > Skybox Static**
- When selected, it duplicates the **static skybox** base material named "Skybox Static."
- The new file will have the default name **"New Skybox Static."**

Both options rely on the same **CreateSkyboxMaterial** function for the actual logic.

🔷 How to Use the Tool in Unity

1. **Open the Project Window:**
   Navigate to the folder where you want to create the new material.
2. **Access the Menu:**
   In Unity's top menu bar, go to:
   Assets → Create → Skybox URP → Material
3. **Choose an Option:**
   - Select **Skybox Procedural** to create a dynamic skybox based on Unity's procedural shader.
   - Select **Skybox Static** to create a static image-based skybox.
4. **Automatic Creation:**
   The script:
   - Finds the base material inside **Skybox Universal RP/Database**.
   - Duplicates it.
   - Saves the new version in your selected folder.
5. **Rename the Material:**
   Once created, Unity automatically highlights the new material and enters rename mode, allowing you to name it right away.

🔷 Summary

In essence, this script is a **workflow enhancement tool** for Unity developers working with skyboxes.
It removes the need to manually locate, duplicate, and rename materials every time you want a new skybox.
Key benefits:

- Prevents naming conflicts by generating unique asset names.
- Automatically places materials in the correct location.
- Speeds up the creation process with one-click menu commands.
- Enhances user experience by automatically focusing and renaming the new asset.

This utility is particularly useful for projects that frequently create multiple skybox materials (for different lighting setups, times of day, or visual styles) and need a clean, organized workflow.

Sky Time Data

This script defines a **data container** used to store and organize information that controls how the sky looks and behaves in a **custom skybox system** for Unity's **Universal Render Pipeline (URP)**. It doesn't perform active calculations or rendering itself — instead, it provides **configurable values** that other scripts can read and use to generate the appearance of the sky, sun, and stars.

🔵 General Purpose

The script introduces an asset type called **SkyTimeData**, which developers can create from Unity's **"Create" menu**.
Each *SkyTimeData* asset represents a **snapshot of the sky's state** — for example, one could represent dawn, another midday, and another night.
Other parts of the skybox system can then use these assets to interpolate or blend between different times of day.
These assets make it easy to fine-tune the appearance of the sky without directly modifying shader code or material settings.

🧩 Structure Overview

The script defines a **ScriptableObject**, which is a special Unity asset used to store persistent data.
It contains several public variables grouped into logical categories, each annotated with Unity's attribute system to enhance usability in the Inspector (e.g., Header, Tooltip).
There are **no methods** in this script — it serves purely as a structured data definition.

🔷 Inspector and Menu Attributes

Before looking at the variables, it's useful to understand the attributes attached to the class:

- **HelpURL**
  Adds a clickable help link in the Unity Inspector. Clicking it opens the GitHub page for the original skybox system.
- **CreateAssetMenu**
  Adds a menu item under:
  Assets → Create → Skybox URP → SkyTimeData
  This allows developers to easily create a new *SkyTimeData* asset directly from the Unity Editor.

🔶 Variables and Their Roles

Each variable defines a specific aspect of how the sky should appear.

🟦 Sky and Lighting Settings

These values control the color and brightness of the general sky and sun.

1. **skyColorGradient**
    - A gradient defining how the sky color changes throughout the day.
    - For example, it might be orange at sunrise, blue at midday, and dark at night.
    - It's used to smoothly transition between sky colors based on time.
2. **sunIntensity**
    - A floating-point value controlling how bright the sunlight appears at a particular time.
    - Higher values make the light more intense, while lower ones simulate dusk or night.
3. **scatteringIntensity**
    - Determines how strong the atmospheric scattering effect is.
    - This affects how light diffuses through the air, impacting realism and color blending, especially at dawn or sunset.

🌠 Stars and Milky Way Settings

These parameters adjust how visible celestial features appear in the sky during nighttime.

4. **starIntensity**
    - Controls how bright the stars appear.
    - Higher values make the stars stand out more against the dark sky.
5. **milkywayIntensity**
    - Defines how strong the Milky Way texture appears.
    - Useful for adjusting the overall visibility of the galactic band in the skybox.

🌈 Texture References

These fields provide optional texture data that can represent or visualize sky colors.

6. **skyColorGradientTex**
    - A texture version of the sky color gradient.
    - It can be used for debugging, visualization, or as an input to shaders that render the procedural sky.

🧠 How It Fits in the System

This class doesn't work alone.
It's designed to be used by other scripts in the *Skybox Universal RP* system — particularly those that:

- Generate the procedural skybox.
- Transition between different times of day.
- Apply lighting and scattering parameters to materials or shaders.

For example, a controller script could load multiple *SkyTimeData* assets (e.g., *Dawn*, *Noon*, *Night*) and blend their parameters in real time based on the in-game time, gradually changing the appearance of the sky.

🧭 Usage in Unity

1. In the Unity menu bar, go to:
   Assets → Create → Skybox URP → SkyTimeData
2. Name the new asset (for example: "Time – Morning").
3. In the Inspector, fill in the properties:
    - Define a **skyColorGradient** with the desired color transitions.
    - Adjust **sunIntensity**, **scatteringIntensity**, **starIntensity**, and **milkywayIntensity** to fit your desired lighting.
    - Optionally assign a **skyColorGradientTex** for reference or shader use.

4. Use this asset in your custom sky system — for example, assigning it to a *Sky Controller* component or script that updates the sky appearance dynamically.

Summary

Category	Variable	Description
Sky and Lighting	skyColorGradient	Controls the sky color transitions throughout the day.
	sunIntensity	Defines the brightness of the sunlight.
	scatteringIntensity	Controls how strongly the atmosphere scatters light.
Stars and Milky Way	starIntensity	Determines how bright the stars appear.
	milkywayIntensity	Adjusts the visibility of the Milky Way.
Textures	skyColorGradientTex	Optional texture representing the color gradient for use in shaders or previews.

In Summary

The *SkyTimeData* script is a **data template** used to store sky lighting and color parameters.
Each asset created from it represents a unique **time of day configuration**.
It allows for clean separation between visual configuration and logic — meaning developers and artists can tweak sky visuals directly from the Unity Editor without touching code.

Sky Time Data Controller

Overall Function

The script works as a **sky controller**.
It uses several **sky data profiles** (called *SkyTimeData*) that describe what the sky should look like at different times (midnight, dawn, noon, etc.).
When given a time value (from 0 to 24), it **interpolates** between two nearby profiles to create a blended sky and lighting setup that matches that exact moment.

Main Components

1. SkyTimeDataCollection

This is a **container** that holds all the key *SkyTimeData* assets representing different times of the day.
There are eight slots, each corresponding to:

- **time0** → midnight
- **time3** → early morning (3 AM)
- **time6** → sunrise (6 AM)
- **time9** → morning (9 AM)
- **time12** → noon
- **time15** → afternoon (3 PM)
- **time18** → sunset (6 PM)
- **time21** → evening (9 PM)

Each of these references a *SkyTimeData* object with information about the sky's color, brightness, and lighting intensities at that moment.

2. Main Controller (SkyTimeDataController)

This is the active component placed in the Unity scene. It handles:

- Storing all sky data references.
- Generating the interpolated sky appearance.
- Updating the environment lighting to match the sky.

🌤️ Variables Explained

skyTimeDataCollection
Holds the group of *SkyTimeData* profiles used for interpolation.
Without it, the controller wouldn't know what sky settings correspond to each time of day.

updateEnvironmentLighting
A toggle that decides whether the script will automatically adjust the ambient lighting in the scene (the indirect light affecting all objects).

defaultColorEnvironmentLighting
A backup color used when automatic lighting updates are turned off.
The scene will use this static color instead of calculating sky-based colors.

inInternalSpace
A flag that tells the system if the player or camera is indoors.
When true, outdoor lighting is replaced with a single indoor color.

colorEnvironmentLighting
The color used when the system detects that the player is indoors.

newData
A temporary *SkyTimeData* instance that stores the final blended sky information after interpolation.
This is what gets returned or applied to the skybox.

🧩 Core Methods and Logic

OnEnable
When the controller activates (e.g., when entering play mode or enabling the component), it creates an empty *SkyTimeData* instance.
This will be used to store and return interpolated data during runtime.

GetSkyTimeData(time)
This is the **heart of the system**.
Purpose:
Given a specific time of day (from 0 to 24 hours), it determines which two sky profiles should be blended, then creates a smooth transition between them.
Steps:

1. **Identify the time segment:**
   The method checks which two *SkyTimeData* profiles the current time lies between.
   For example, 7:30 AM falls between time6 (sunrise) and time9 (morning).
2. **Calculate the interpolation value (lerpValue):**
   It calculates a number between 0 and 1 to represent how far the current time is between the two points.
   Example: 7:30 AM would be halfway (0.5) between 6 AM and 9 AM.
3. **Generate a new gradient texture:**
   It calls another method to blend the two sky color gradients, creating a smooth color transition for the sky.
4. **Update lighting:**
   It triggers the lighting update method so the ambient light in the scene matches the sky's new colors.
5. **Interpolate sky intensities:**
   It blends all numeric values between the two sky profiles:
   - Star brightness
   - Milky Way intensity
   - Sunlight intensity
   - Atmospheric scattering

6. **Return the blended result:**
   The resulting *SkyTimeData* (stored in newData) represents the exact appearance of the sky for that moment in time.

GenerateSkyGradientColorTex(startGradient, endGradient, resolution, lerpValue)
This method **creates a new sky texture** that smoothly transitions between two color gradients.
How it works:
- A new texture is created with a specified width (resolution).
- For each pixel across the width:
  - It finds the color from the first gradient and the second gradient.
  - It then blends them based on the interpolation factor (lerpValue).
- The resulting texture is applied to the skybox to create a realistic color fade across the horizon.

This is what makes sunrises, sunsets, and day-to-night transitions look smooth.

UpdateEnvironmentLighting(start, end, lerpValue)
This method updates the **ambient lighting colors** in Unity's environment settings.
How it works:
1. **Interpolate three lighting components:**
   - **Sky color:** the light from the sky dome.
   - **Equator color:** the middle part of the sky (used for balanced light).
   - **Ground color:** the light reflected upward from the ground.
2. **Check lighting settings:**
   - If lighting updates are disabled, it forces all three colors to use the default ambient color.
   - If the system is indoors, it overrides all colors with the indoor lighting color.
3. **Apply the result:**
   It updates Unity's internal lighting system (RenderSettings) so the scene's global illumination matches the sky appearance.

This ensures objects in the world reflect realistic light depending on the time of day.

How to Use It
1. **Create several SkyTimeData assets**
   Each one represents a specific time of day (e.g., 6 AM, 12 PM, 6 PM).
   Adjust gradients, sun intensity, scattering, stars, and Milky Way settings for each.
2. **Create a SkyTimeDataCollection**
   Assign all these time-based profiles to their corresponding slots.
3. **Add the SkyTimeDataController** to a GameObject in your scene.
   Assign the *SkyTimeDataCollection* to it.
4. **Call GetSkyTimeData(time)**
   From another script or system that tracks in-game time.
   This will return a *SkyTimeData* that visually represents the current moment.
5. **Apply the resulting data** to your custom skybox shader or system.
   The system will automatically handle ambient lighting transitions if enabled.

Summary
In essence:
- **SkyTimeData** = defines *how* the sky looks at a specific moment.
- **SkyTimeDataCollection** = holds all key moments of the day.
- **SkyTimeDataController** = calculates smooth transitions between them in real time.

It's a **dynamic procedural sky manager**, producing seamless day/night cycles and adaptive lighting for atmospheric realism.

Sky Controller

General Overview

The script's goal is to control how the **skybox and lighting evolve over time**.
It tracks the current time of day, interpolates colors and intensities from data assets, moves celestial objects, updates lighting, and can automatically transition between **day and night**.
It also includes a **custom Unity Editor section** that allows exporting the current sky gradient as a texture file.

Main Components and Their Purpose

TimeMode Enumeration

Defines how time advances in the simulation:

- **Disabled** → Time doesn't move.
- **OnlyOnStart** → Time updates once at the start.
- **Continuous** → Time continuously progresses every frame.
- **BySpeed** → Time moves at a custom speed multiplier.
- **ByElapsedTime** → Time progresses based on normalized elapsed time (useful for syncing to real time or a timeline).

Variables and What They Represent

Execution

- **executeInEditMode** – Allows updates to run even when the Unity editor is not in Play Mode. Useful for previews.

Time Settings

- **currentTime** – Represents the time of day in hours (0 = midnight, 12 = noon, etc.).
- **timeMode** – Selects how time progression behaves (from the TimeMode enum).
- **isReversedTime** – Determines if time runs backward.
- **timeMultiplier** – Controls how fast time moves in BySpeed mode.
- **dayDuration** – Defines how long a full day lasts (in seconds).

Scene References

- **skyboxMaterial** – Material controlling the skybox's appearance.
- **sunLight** / **moonLight** – Directional lights representing the sun and moon.
- **sunPivot**, **moonPivot**, **milkyWayPivot** – Transforms used to rotate and position celestial objects.
- **milkyWayDefaultRotation** – Defines the default angle for the Milky Way texture.

Environment Lighting

- **isIndoorEnvironment** – Defines if the current scene should simulate indoor lighting.
- **autoUpdateLighting** – Enables automatic updates to Unity's ambient lighting.
- **ambientBaseColor** – Base color for ambient light (when no dynamic lighting is applied).
- **maxLightIntensity** – Sets the maximum brightness for sun and moon.

Runtime State

- **giUpdateTimer** – Timer used to trigger global illumination refresh.
- **isTransitioningDayNight** – Flag that prevents overlapping day/night transitions.
- **autoCalculateIntensity** – Enables automatic brightness calculation for celestial lights.
- **timeProgressRatio** – Normalized progress of the day (0–1) when using ByElapsedTime mode.

Private Fields

- **hasStarted** – Prevents repeated execution in OnlyOnStart mode.
- **skyTimeDataController** – Reference to another component responsible for interpolating between different sky data sets.
- **currentSkyTimeData** – Stores the sky data currently being applied (colors, intensities, etc.).

⚙ Main Methods and Their Role

OnEnable

Runs when the component is activated.
It links to the SkyTimeDataController component on the same GameObject.

Update

This is the **main loop** of the controller.
Every frame, it performs:

1. **Time progression** (via HandleTimeProgression).
2. **Day/Night transitions** when the clock hits 6:00 (sunrise) or 18:00 (sunset).
3. **Global illumination updates** if enough time has passed.
4. **Sky updates**, **environment settings**, and **celestial rotations**.
5. **Applies interpolated properties** to the skybox material.

🕒 Time Management

HandleTimeProgression

Manages how time moves depending on the selected mode:

- In **Continuous** mode, time moves smoothly and loops every 24 hours.
- In **BySpeed** mode, it advances based on the multiplier.
- In **ByElapsedTime** mode, it calculates progress as a ratio between 0 and 1.
- In **OnlyOnStart**, it updates once, then stops.
- If **isReversedTime** is true, time runs backward.

It also increments the **global illumination timer** for light updates.

☁ Sky and Lighting Updates

UpdateSkyData

Retrieves interpolated SkyTimeData for the current time using the connected SkyTimeDataController.
This data includes the **sky gradient**, **sunlight intensity**, **star brightness**, and **atmospheric scattering**.

ApplyEnvironmentSettings

Updates environment-related variables before applying them to the scene:

- Syncs environment lighting parameters with the controller.
- Applies colors and toggles between indoor/outdoor lighting states by enabling shader keywords in the skybox material.

UpdateCelestialTransforms

Rotates the **sun**, **moon**, and **Milky Way** based on time:

- The **sun** moves across the sky between 6:00 and 18:00.
- The **moon** moves during the night, offset from the sun's rotation.
- Updates pivot transforms to ensure proper world-space alignment.
- Sends updated light directions to the skybox material for shader calculations.

If **autoCalculateIntensity** is active, it automatically adjusts the brightness of the sun and moon so that when one rises, the other fades.

CalculateLightIntensity

Computes a value between 0 and 1 for how bright the sun or moon should be.
Uses a cosine wave based on the time of day — making noon the brightest for the sun and midnight the brightest for the moon.

ApplySkyProperties

Applies current interpolated sky data to the skybox material:

- Sets gradient textures, scattering levels, and intensities.
- Updates shader matrices for moon and Milky Way alignment.

Day/Night Transitions

TransitionToNight and TransitionToDay

Smoothly blend lighting between day and night.
They run as coroutines to gradually adjust the **sun** and **moon** intensities over time.

- During **TransitionToNight**, the sun fades out while the moon fades in.
- During **TransitionToDay**, the moon fades out and the sun lights up again.

These transitions prevent abrupt lighting changes and maintain visual realism.

Editor Customization (in Unity Editor only)

SkyControllerEditor

A custom editor script that enhances usability:

- Adds a **button** to export the current sky gradient as a PNG image.
- Automatically opens a folder selection dialog for saving the file.
- Reads the current gradient texture from the material and saves it to disk.
- Keeps the default Unity inspector layout below the custom button.

This helps artists preview and reuse gradient textures outside of the procedural system.

Usage in Unity

1. **Attach** this component to a GameObject in the scene (e.g., “SkyController”).
2. **Assign** the required references:
    - A **Skybox Material** that supports the procedural URP shader.
    - **Sun** and **Moon** lights (Directional).
    - **Pivots** for rotation (optional but recommended).
3. **Link** a SkyTimeDataController component to provide sky gradient data.
4. Adjust:
    - **Day duration**, **time mode**, **multiplier**, and **environment lighting**.
5. In Play Mode, the sky dynamically transitions based on time — sun rises, moon sets, lighting adapts, and the skybox gradient evolves.

Summary of Behavior

- Controls **time simulation** (fast-forward, real-time, reverse).
- Interpolates **sky appearance** through time.
- Animates **sun**, **moon**, and **Milky Way** rotations.
- Updates **environment lighting** and **global illumination**.
- Allows **manual texture export** for visual debugging or reuse.

Sky Controller Trigger

Purpose and Usage

This component is designed to be attached to a **GameObject** that has a **Collider** configured as a **trigger**.
When the player (or any GameObject with a specific tag) enters the trigger zone, the script updates the **SkyController’s** IsIndoorEnvironment property — controlling whether the scene lighting and skybox should behave like an *indoor* or *outdoor* environment.
Example usage:

- Attach this script to a doorway, building entrance, or cave trigger.
- When the player enters, the lighting switches to an indoor mode.
- When leaving, another trigger can switch it back to outdoor mode.

⚙️ Inspector Settings

inInternalSpace

- Type: Boolean (true/false)
- Purpose: Defines whether the **SkyController** should consider the player to be *inside* an internal space.
- Example:
  - Set **true** on a building interior trigger (to simulate entering).
  - Set **false** on a doorway exit trigger (to simulate leaving).

This directly controls the SkyController's IsIndoorEnvironment variable.

playerTag

- Type: String
- Default: "Player"
- Purpose: Determines which GameObjects can activate the trigger.
  Only colliders with this tag will affect the SkyController.

Example:
If your player object is tagged as "Player", only the player entering will change the lighting — other objects (like enemies or physics props) won't affect the sky.

🔒 Private Reference

skyController

- Type: Reference to a **SkyController** instance in the scene.
- Purpose: Stores a link to the SkyController so the trigger can send commands to it.
- Automatically assigned when the script starts via a search in the scene.
- If no SkyController is found, the script logs a warning message.

🧭 Public Properties

These allow runtime or inspector access to the private fields.

InInternalSpace

- Getter and setter for the inInternalSpace variable.
- Allows external scripts or UI elements to dynamically change whether the trigger activates indoor or outdoor mode.

PlayerTag

- Getter and setter for the playerTag variable.
- Lets you change which tag activates the trigger without editing the script.

🔁 Unity Event Methods

OnEnable()

- Called automatically by Unity when the GameObject becomes active.
- The method searches the entire scene for a **SkyController** instance using FindAnyObjectByType.
- If no SkyController exists, a warning message appears in the Unity Console:

"No SkyController found in the scene. The trigger will have no effect until one is present."
This ensures that the script doesn't break if it's active before the SkyController loads.

OnTriggerEnter(other)

- Triggered automatically when another Collider enters the trigger zone of this GameObject.
- The method performs three steps:

1. **Checks the tag:**
   It verifies if the entering Collider's GameObject has the same tag as playerTag.
2. **Verifies the SkyController:**
   If no SkyController was found earlier, it warns the developer.

3. **Updates environment state:**
   If both conditions are met, it sets the SkyController's IsIndoorEnvironment to match the value of inInternalSpace.

It also logs a confirmation message:
"SkyController internal space set to true."
or
"SkyController internal space set to false."
This message helps confirm in the console when a trigger is successfully activated.

🧠 Summary of the Process

1. The script looks for a **SkyController** in the scene when it becomes active.
2. A Collider configured as a **trigger** detects when an object enters its area.
3. If that object has the **configured tag** (usually "Player"), the script sends a command to the SkyController:
   - If inInternalSpace is **true**, lighting and skybox behave as if the player is *indoors*.
   - If **false**, the system switches back to *outdoor* conditions.
4. All state changes are logged in the Console for debugging clarity.

📦 Typical Setup Example

1. Create an **empty GameObject** (e.g., "IndoorZoneTrigger").
2. Add a **Box Collider** component.
3. Enable **"Is Trigger"**.
4. Attach the **SkyControllerTrigger** script.
5. Configure:
   - In Internal Space → **true** (or false for exit zones)
   - Player Tag → "Player"
6. Ensure a **SkyController** component exists in your scene.
7. When the player enters or exits the area, the skybox lighting changes smoothly according to the SkyController logic.

🧩 Key Takeaway

This script provides a **simple interaction layer** between gameplay elements and environmental rendering.
It doesn't directly modify visual effects but instead communicates with the **SkyController**, allowing environmental lighting and sky appearance to respond dynamically to player movement — creating immersive transitions between indoor and outdoor spaces.

Sky Controller Menu / Sky Controller Menu TMP

Both scripts are **UI controllers** that connect Unity's visual interface (sliders, dropdowns, toggles, text fields) to a **sky simulation system** managed by another component called SkyController.
They serve as the "bridge" between the player or developer adjusting time-related settings through the UI and the internal logic that updates the virtual sky.

🧭 Overall Purpose

The purpose of these scripts is to allow users to **control the time of day** and **sky behavior** interactively, such as changing the hour, day duration, or whether time flows forward or backward.
There are **two versions**:

- **Legacy version** — uses Unity's default UI elements (Text, Dropdown, InputField).
- **TMP version** — uses **TextMeshPro (TMP)** equivalents (TMP_Text, TMP_Dropdown, TMP_InputField), which offer better text rendering and customization.

Despite the different UI systems, **both scripts perform the same logic and have identical functionality**.

🧩 Structure Overview

Each script is organized into four main regions:

1. **Fields** – Contain references to the sky system and UI elements.
2. **Properties** – Public accessors for those fields, allowing other scripts or the Unity Inspector to access them safely.
3. **Unity Lifecycle** – Contains OnEnable() and OnDisable() methods, which handle initialization and event binding.
4. **Private Methods** – Handle user interactions, input validation, and visual updates.

🌞 Field Descriptions

1. skyController

Reference to the external SkyController component.
This is the object being controlled — it manages the actual sky simulation such as time progression, light intensity, and color transitions.

2. timeSlider

A horizontal slider that represents the **current time of day** (for example, from 0 to 24).
When the user moves the slider, it updates the CurrentTime variable inside the SkyController.

3. timeLabel

A text display that shows the **current time value** as a rounded number (like H: 12).
It is automatically refreshed whenever the slider changes.

4. timeProgressionDropdown

A dropdown menu used to select the **time progression mode**.
This could be:

- **By Speed:** Time flows at a certain speed multiplier.
- **By Elapsed Time:** A full day completes in a fixed duration (for example, 5 real minutes).

5. reverseTimeToggle

A simple toggle button that determines if **time flows backward** instead of forward.
When checked, the SkyController reverses the sun and sky movement direction.

6. timeSpeedInput

A text input field that defines how fast time progresses when in *speed-based mode*.
Example: entering “2” doubles the day-night cycle speed.

7. dayLengthInput

Another text input field that defines **how long a day lasts** (in seconds or minutes) when in *time-based mode*.

📦 Property Section

Each field has a corresponding **getter and setter property**, allowing other parts of the program to modify them dynamically or read their current state.
For instance:

- TimeSlider allows accessing or replacing the time slider.
- TimeLabel allows accessing the text element that shows the time.

This section improves modularity — other scripts can reuse or swap UI components without directly modifying private fields.

⚙️ Unity Lifecycle Section

OnEnable()

This method runs whenever the UI element becomes active in the scene.
It performs several tasks:

1. **Initial synchronization:**
   Reads the current values from the SkyController and applies them to all UI elements so the user sees the correct state immediately.
2. **Event subscription:**
   Each UI element is linked to a specific handler method:
   - Slider → OnTimeChanged
   - Dropdown → OnTimeProgressionChanged
   - Toggle → OnReverseTimeChanged
   - Input fields → OnTimeSpeedChanged and OnDayLengthChanged
3. **Input validation setup:**
   The script configures the text fields to only accept numeric characters and one decimal point.
4. **UI refresh:**
   It calls UpdateTimeProgressionInteractable() to enable or disable fields depending on the selected mode, and UpdateDisplayedTime() to show the current hour.

OnDisable()
This method executes when the UI is turned off or destroyed.
It removes all event listeners and input validators to prevent errors or memory leaks when the UI is reactivated later.

🧠 **Private Method Descriptions**

ConfigureInputField(field)
Prepares a text input to only allow **numbers and a decimal point**.
It ensures users can't type letters or invalid symbols.

OnTimeChanged(value)
Triggered whenever the user moves the **time slider**.
It updates the CurrentTime value in the SkyController and refreshes the displayed text.

UpdateDisplayedTime(value)
Formats and displays the current time value as an integer hour (e.g., H: 13).

OnTimeProgressionChanged(value)
Triggered when the **time mode dropdown** changes.
It converts the dropdown's selected index into a time mode and updates both the sky system and UI interactivity accordingly.

OnReverseTimeChanged(value)
Runs when the user toggles the **reverse time** option, updating the sky simulation to move backward or forward.

OnTimeSpeedChanged(value)
Triggered when the user edits the **speed input**.
If the input is a valid number, it updates the sky system's **time speed multiplier**.

OnDayLengthChanged(value)
Triggered when the user edits the **day length input**.
If valid, it updates how long a full day cycle lasts.

UpdateTimeProgressionInteractable(mode)
This method adjusts which inputs are **enabled** or **disabled** based on the chosen progression type:

- If the mode is *By Speed*, the timeSpeedInput field is active.
- If the mode is *By Elapsed Time*, the dayLengthInput field is active.

ValidateNumericInput(text, index, char)
Custom character validation.
It ensures that only digits (0–9) and a single decimal point (.) can be entered in the numeric fields.

🧭 **Usage Summary**

1. Attach either SkyControllerMenu or SkyControllerMenuTMP to a GameObject that holds UI components.
2. Assign references to:
    - A SkyController component.
    - UI elements (slider, text, dropdown, toggle, input fields).
3. When the scene runs:
    - The menu reads settings from the controller and populates the UI.
    - User interactions automatically update the sky system in real time.

⚖️ Neutral Comparison (Legacy vs TMP)

Feature	Legacy Version	TMP Version
Text Handling	Uses Unity's default Text, Dropdown, InputField.	Uses TMP_Text, TMP_Dropdown, TMP_InputField for higher quality text rendering.
Functionality	Identical logic and behavior.	Identical logic and behavior.
Recommended Use	Suitable for older projects or when TMP is not installed.	Recommended for modern Unity projects.

✅ In short:

These scripts serve as **user interfaces** for controlling a dynamic sky system.
They synchronize UI elements with sky simulation parameters, process user input safely, and visually display the current time of day, ensuring a responsive and intuitive way to interact with the environment's lighting and time cycle.

Screenshot Skybox

🔷 Overview

This script is designed for **Unity** and allows you to **capture a full 360-degree screenshot of a skybox** using **six cameras**, each facing a different direction.
It saves each camera's view as a **PNG image file** inside a **"Screenshots" folder** located within the project's **Assets/Editor** directory.
Additionally, the script comes with a **custom Unity Editor interface** that adds two buttons directly into the **Inspector panel**:

- One button captures the screenshots.
- The other opens the folder containing them.

🔷 Variables and Fields

1. pixelSize

This variable determines the **resolution** (in pixels) of each captured image.
For example, a value of 2048 means that every screenshot will be 2048x2048 pixels.
It directly affects image quality and memory usage.

2. cameras

This is an **array of six Camera objects**, which must be assigned manually in the Unity Inspector.
Each camera represents one of the six directions in a cube map:

- **Front**
- **Right**
- **Back**
- **Left**
- **Up**
- **Down**

The combination of all six creates a **complete 360-degree panoramic capture**.

3. screenshotFolder

This is a text string that defines the **destination path** for the saved images.
By default, it points to a folder named **"Screenshots"** inside **Assets/Editor**.
If the folder does not exist, the script automatically creates it.

🔷 Properties

Two public properties exist for better control in the Unity Editor or via other scripts:

1. **PixelSize**
    - Allows reading or modifying the screenshot resolution.
2. **Cameras**
    - Allows reading or replacing the array of six camera references.

🔷 Public Methods

1. TakeScreenshot

This is the main function that performs the 360° capture process.

Step-by-step process:

1. It checks that exactly six cameras have been assigned.
   If not, an error message appears in the Unity Console.
2. It calls a private helper method to **align all cameras** correctly so that each one points toward one of the cube map directions.
3. It checks if the screenshot folder exists, creating it if necessary.
4. For every camera:
    - A temporary render texture is created to store the camera's view.
    - The camera renders its current view into that texture.
    - The script copies the pixels from the render texture into a 2D texture in memory.
    - That texture is then **encoded as a PNG file**.
    - A unique filename is created using the camera's name and a timestamp (for example: "Screenshot-Front-2025-11-03-19-24-00.png").
    - The image file is written to disk.
    - Temporary textures are destroyed to free memory.
    - A confirmation message is printed in the Console showing where the image was saved.

This method ensures all six directions of the skybox are captured, giving you a full cube map-style set of images.

2. OpenScreenshotFolder

This function opens the folder where screenshots are stored using the system's file explorer.
It's a quick way to access your captures without manually browsing through project folders.

🔷 Private Methods

ApplyCameraPositionsAndRotations

This helper method assigns the **correct orientation** (rotation) to each of the six cameras.
Each camera is placed at the **world origin (0, 0, 0)** and rotated so that they collectively face all six cube faces:

- Camera 0 → Forward
- Camera 1 → Right
- Camera 2 → Back
- Camera 3 → Left
- Camera 4 → Up
- Camera 5 → Down

This setup ensures that all angles of the environment are captured seamlessly.

🔷 Custom Editor Section

At the bottom, there's a separate code block that defines a **Custom Unity Editor** for this script.
This part is only compiled inside the **Unity Editor** (not in builds).
It modifies how the component appears in the **Inspector** window.
Custom features added:

1. A button labeled **"Take Screenshot"** — directly triggers the screenshot capture method.
2. A button labeled **"Open Folder"** — opens the screenshot folder on the system.

Below these buttons, the default Inspector layout (showing the pixel size and camera list) is displayed.

🔷 Usage in Unity

1. **Attach the Script:**
   Add the Screenshot Skybox component to an empty GameObject in your scene.
2. **Assign Cameras:**
   In the Inspector, drag and drop six cameras into the "Cameras" list, each corresponding to one direction.
3. **Set Resolution (Optional):**
   Adjust the "Pixel Size" field to the desired screenshot resolution.
4. **Capture the Skybox:**
   - Click the **"Take Screenshot"** button in the Inspector.
   - The script will automatically create a "Screenshots" folder and save six PNG files (one per direction).
5. **Open the Folder:**
   - Click the **"Open Folder"** button to view the saved images.

🔷 Summary

In essence, this script automates the process of **capturing a cube map representation of your scene's skybox** in Unity.
By combining the six images, you can later use them for environment maps, lighting probes, or even generating HDR textures.
It's a practical developer tool that integrates neatly with Unity's workflow, providing instant skybox captures with minimal setup.