Azure[Sky] Dynamic Skybox v8.0.0[1] [2]

1 Documentation created using the LTS version of Unity 2022.3.15f1 as reference.

2 English is not my native language, so I’m sorry for possible grammar errors.

Sumário

Getting Started Using Legacy Render Pipeline:

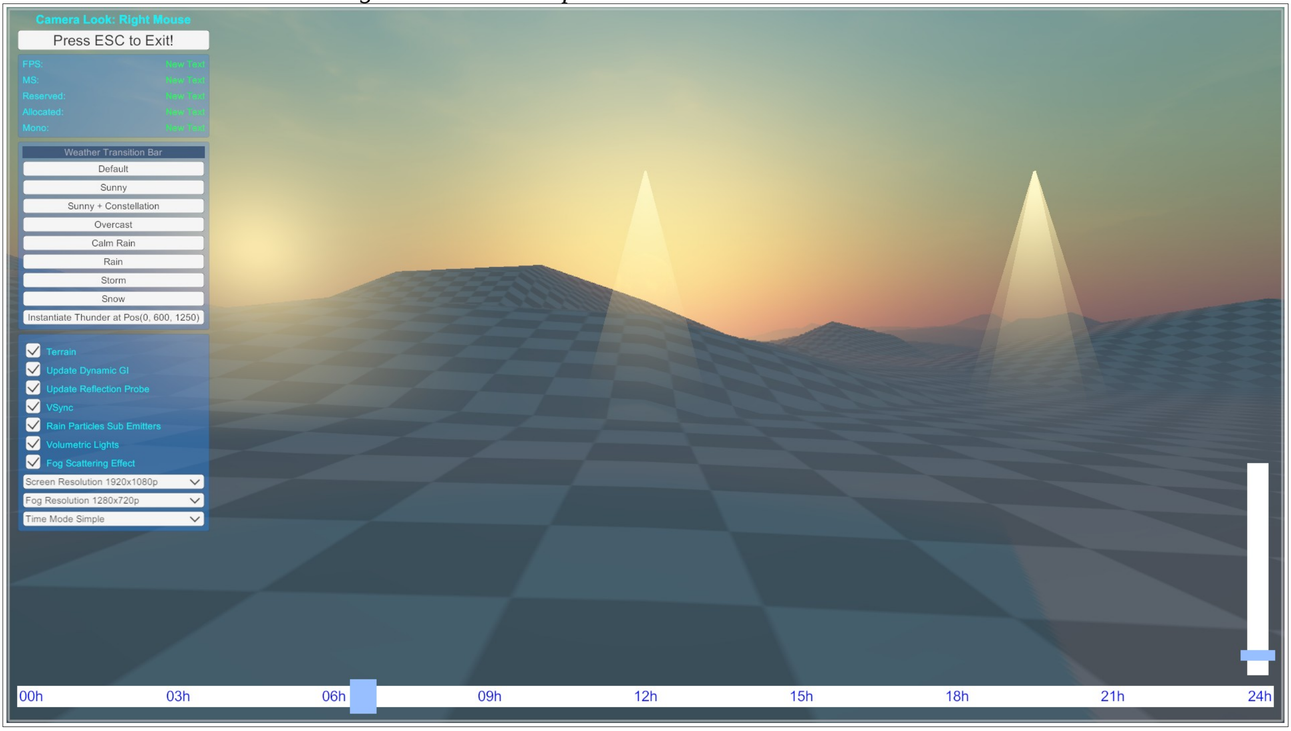
- First import the **Azure[Sky] Dynamic Skybox** package to your project.
- Drag the Azure[Sky] prefab to the scene.
  Assets/Plugins/Azure[Sky] Dynamic Skybox/Core/Prefabs.
- Delete any pre-existing '***Directional Light***' in the scene. The prefab already comes with a directional light as a child game object.
- Add the Fog Scattering effect to the Camera by attaching the ***AzureFogScatteringRenderer.cs*** to the camera or by using the component menu:
  Component>Azure[Sky] Dynamic Skybox>Azure Fog Scattering Renderer.
- Disable the '***Anti-Aliasing***' located at the '***Quality***' tab of the '***Project Settings***' window. The default Unity's anti-aliasing can cause artifacts on the terrain edge when there is only the background sky rendered behind it. It is recommended to use a post-processing anti-aliasing effect instead.
- Disable Unity's default fog in the '***Lighting***' window.

Getting Started Using Universal Render Pipeline:

- First import the **Azure[Sky] Dynamic Skybox** package to your project.
- Import the '***Replacement.unitypackage***' from the folder '***Replacements***' located at *Assets/Plugins/Azure[Sky] Dynamic Skybox/Replacements*. The folder may contain more than one unitypackage file, so just import the one who is more indicated according to the URP version installed in your project.
- Drag the Azure[Sky] prefab to the scene.
  Assets/Plugins/Azure[Sky] Dynamic Skybox/Core/Prefabs.
- Delete any pre-existing '***Directional Light***' in the scene. The prefab already comes with a directional light as a child game object.
- Create a new '***UniversalRendererData.asset***' by the menu:
  Create>Rendering>URP Universal Renderer, and add the '***AzureFogScatteringFeature***' to it. Or just add the fog renderer feature to a pre-existing universal renderer data already in use by your project.
- Remember to add the '***UniversalRendererData.asset***' to the '***Renderer List***' of the '***UniversalRenderPipeline.asset***' in use by your project, in case you have created a new one.
- Select your main camera and set the '***Depth Texture***' option to '***On***', its is located at the '***Rendering***' tab of the '***Camera***' component Inspector.
- Still in the main camera, set the '***Renderer***' option to use the '***UniversalRendererData***' that contains the fog renderer feature attached to it. This will force that camera to render the fog scattering effect.
- Disable the '***Anti-Aliasing***' located at the '***UniversalRenderPipeline.asset***'. The default Unity's anti-aliasing can cause artifacts on the terrain edge when there is only the background sky rendered behind it. It is recommended to use a post-processing anti-aliasing effect instead.
- Disable Unity's default fog in the '***Lighting***' window.

After finishing the ‘Getting Started’ instructions, you can open the sample scene located at the folder: *Assets/Plugins/Azure[Sky] Dynamic Skybox/Demos/Scenes*. If everything went as expected, you should see the sample scene similar to the image below.

Note that the sample scene may change in future updates.
*It is recommended to set the ‘**Color Space’** to ‘**Linear’** in the ‘**Project Settings’** for a better lighting. The lighting of the demo scene were customized using the linear color space.*



✓ Terrain
✓ Update Dynamic GI
✓ Update Reflection Probe
✓ VSync
✓ Rain Particles Sub Emitters
✓ Volumetric Lights
✓ Fog Scattering Effect
Screen Resolution 1920x1080p
Fog Resolution 1280x720p
Time Mode Simple

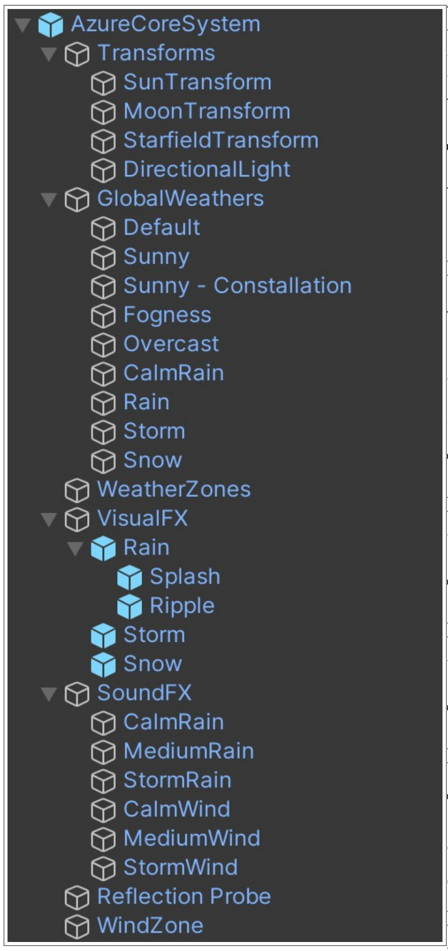
*See the ‘**Known Issues’** section for an important information regarding the prefab and the Unity’s undo system feature*.

Simple Volumetric Lights Effect:

Azure[Sky] supports the rendering effect of simple volumetric lights (*point and spot only*). The current volumetric lights effect is recommended for prototyping purposes or for simple games (*current it does not supports volumetric shadows*). To enable this effect in your game, you need to follow some steps:

- Add the ‘***AzureVolumetricLight.cs***’ component to any spot or point light.
  Component>Azure[Sky] Dynamic Skybox>Azure Volumetric Light.
- Attach the appropriated mesh to the ‘***Light Mesh***’ field in the Inspector. Azure comes with one Sphere mesh for the point lights, and two Cone meshes for the spot lights. The meshes are located at the folder: *Assets/Plugins/Azure[Sky] Dynamic Skybox/Core/Meshes*.
- Add the ‘***AzureVolumetricLightRenderer.cs***’ to the camera you want to render the volumetric lights.
  Component>Azure[Sky] Dynamic Skybox>Azure Volumetric Light Renderer.

*See the ‘**Known Issues’** section for an important information regarding the volumetric lights effect feature*.



```
AzureCoreSystem
  Transforms
    SunTransform
    MoonTransform
    StarfieldTransform
    DirectionalLight
  GlobalWeathers
    Default
    Sunny
    Sunny - Constallation
    Fogness
    Overcast
    CalmRain
    Rain
    Storm
    Snow
  WeatherZones
  VisualFX
    Rain
      Splash
      Ripple
    Storm
    Snow
  SoundFX
    CalmRain
    MediumRain
    StormRain
    CalmWind
    MediumWind
    StormWind
  Reflection Probe
  WindZone
```

Prefab Structure:

Inside the prefab, you will find some important game objects for the sky system operation.

Transforms: Inside the 'Transforms' game object are the transforms manipulated by the 'Time System'. After the 'Time System' ends the celestial bodies calculations, it sets the rotation/direction of these transforms according to its purposes. And later their rotation/direction are used by the '***AzureSkyRenderer.cs***' component as reference to render some elements of the skybox (*sun texture, moon texture, stars and more*).

The "SunTransform" receives the rotation/direction that will represent the sun position in the sky. You can use this transform for flare effect or any effect dependent of the sun direction.

The "MoonTransform" receives the rotation/direction that will represent the moon position in the sky. You can use this transform for flare effect or any effect dependent of the moon direction.

The "StarfieldTransform" receives the rotation/direction that will represent the entire starfield, as Milky Way and Regular Stars.

The "DirectionalLight" is the main light used to lighting the scene as any other directional light, but for this use case, it works a bit different. When the sun sets below the horizon line and there is moon in the sky at night, the directional light will automatically receive the rotation/direction of the 'MoonTransform'. But if the sun is above the horizon line, the directional light will automatically receive the rotation/direction of the 'SunTransform', even if the moon is visible at daytime. There is no need for two directional lights in the scene (*it is bad for performance*) because when the sun and moon are both visible in the sky (*above the horizon line*), it means that it is daytime, and at the daytime, the light reflected by the moon is irrelevant for the scene lighting and atmospheric scattering effect.

GlobalWeathers: Inside the 'GlobalWeathers' game object are the weather presets used by the 'Weather System'. I chose for design and organization to put the global weathers inside a separate transform, each global weather game object stores a different preset according to its designation. You can delete or create any global weather presets as you want. See how it works later…

WeatherZones: Here you can place the weather zones used by the 'Weather System' in case you plan to use this feature in your scene. Of course, you can create the weather zones and global weathers outside the prefab, but this way, I consider that it is better for organization purposes. See how it works later…

VisualFX: Inside the 'VisualFX' game object are the particle systems as rain, snow, ripples, etc…

SoundFX: Inside the 'SoundFX' game object are the sound effects as wind, rain, storm, etc…

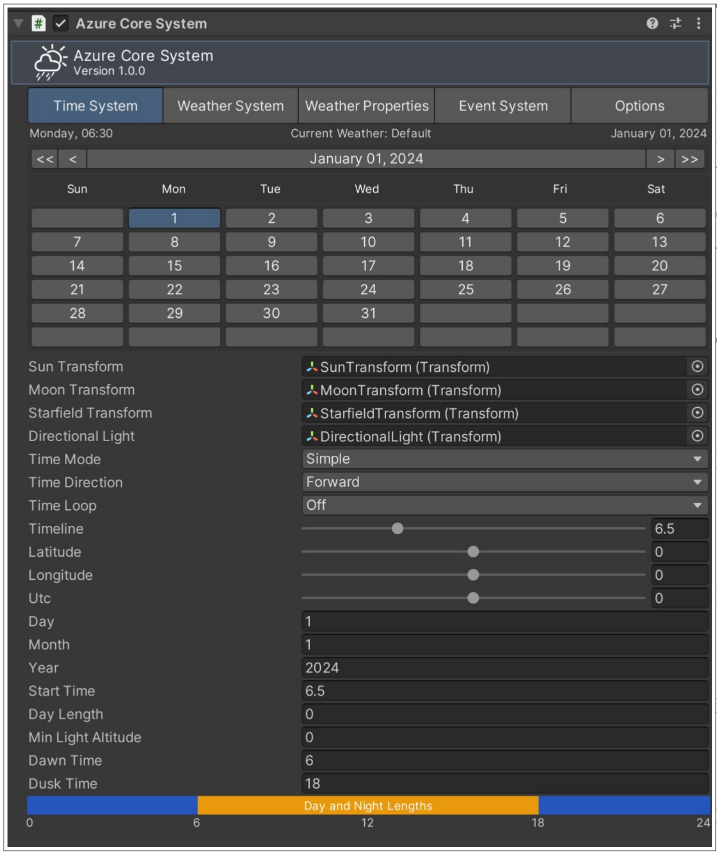
ReflectionProbe: This reflection probe is handled by the core system and is updated synchronized with the core system update, this is useful for performance. Why update the reflection probe every frame? This one will be updated only when there is a change in the sky or in the lighting caused by the change in the time of day. Check the reflection probe box size to best fit your scene, it is configured as (10000, 10000, 10000) by default to work with ocean water systems, maybe it is too high for your scene.

WindZone: This wind zone is handled by the weather system, you can configure its intensity and direction in each weather preset, so you can get different wind settings as the weather changes.

Azure Core System:

Component>Azure[Sky] Dynamic Skybox>Azure Core System.

The AzureCoreSystem is the main component in the entire system and by default it is attached to the main parent game object that compose the Azure[Sky] prefab, and as you can see in the previous page, it also is used to name the prefab. The core system is composed by the '***Time System***', '***Weather System***', '***Event System***' and a "simple" '***Update System***' as well as the '***Weather Properties***' and '***Override System***'. In other words, all the sky system turns around the core system, and you can easily implement your custom scripts or third party assets to work with the '***Weather System***' using the custom '***Weather Properties***' and the '***Override System***'. You can navigate in the header toolbar to change the Inspector options according to the system you want to configure.



Azure Core System
Azure Core System
Version 1.0.0

Time System | Weather System | Weather Properties | Event System | Options

Monday, 06:30 — Current Weather: Default — January 01, 2024

<< < January 01, 2024 > >>

Sun	Mon	Tue	Wed	Thu	Fri	Sat
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Sun Transform	SunTransform (Transform)
Moon Transform	MoonTransform (Transform)
Starfield Transform	StarfieldTransform (Transform)
Directional Light	DirectionalLight (Transform)
Time Mode	Simple
Time Direction	Forward
Time Loop	Off
Timeline	6.5
Latitude	0
Longitude	0
Utc	0
Day	1
Month	1
Year	2024
Start Time	6.5
Day Length	0
Min Light Altitude	0
Dawn Time	6
Dusk Time	18

Time System:

In the '<u>Time system</u>' are all the settings relative to date, location and time of day. It is very intuitive to use, so I will point only the most important ones.

Time Mode: Set if you want the celestial bodies to be represented as a Simple or a Realistic position in the sky.

The simple time mode uses only a few quaternion operations and the moon is always in the opposite position in the sky relative to the sun, this mode is very cheap for the performance.

In the realistic time mode, the sun and the moon position in the sky will be realistically calculated based on the date, time and geographical location taking the Earth as reference. The phases of the moon will change automatically and will be the same as in the real world. Some architects use this mode to simulate the real world sun lighting in their projects.

If you want to know the north direction to reference your scene for your architectural project, it is very simple. In Unity, the north direction will be the direction that any camera is pointing when you set its rotation to (0, 0, 0).

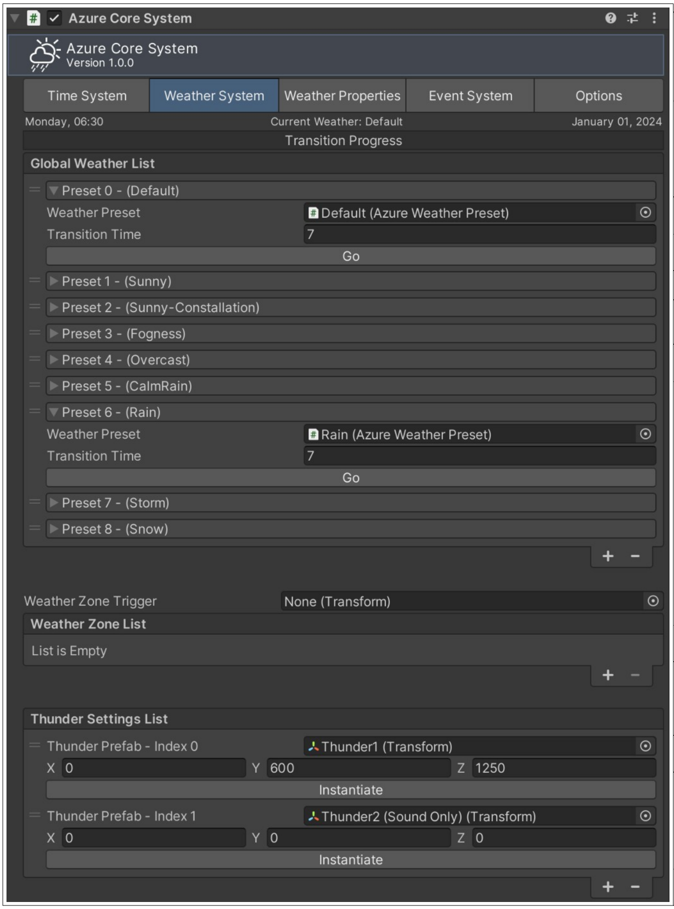
Timeline: Represents the 24 hours day cycle, but do not confuse with the time of day. Generally the time of day will be the same as the timeline value, but in case you change the 'Dawn Time' and the 'Dusk Time' to alter the length of the daytime and nighttime, the time of day will be different than the timeline value. The actual time of day value is displayed at left-top of the component Inspector, next to the day of the week display.

Start Time: Is the default value that the timeline will be initialized when the scene starts. Note that no matter the value you set in the timeline slider, the scene will always starts using the "Start Time" value as reference. This is useful while in the edit mode because you can change the timeline to test if any weather property setting is good at that time of day, this way you don't have to worry if you forgot to set the timeline slider back to its original value that you want the scene to start.

Day Length: Controls the duration of the day cycle in minutes. Example, if you set it to 30, the entire day cycle will last 30 minutes in the real world time. If you do not want the time of day to change, just set the 'Day Length' to zero, this will keep the time of day static in the start position. If the day length is set to zero, you still can change the timeline by code or by moving the slider in the Inspector.

Min Light Altitude: Limits the angle the directional light can get near to the horizon line. When the sun is near the horizon line, the light affects the scene almost horizontally, causing very long and stretched shadows. This is not good for performance due to the way shadows are made in graphics programmer, this simple thing can increase a lot the triangle count of your scene drastically impacting the performance of your game. So you can try to set a minimum light altitude for the directional light direction to avoid stretched shadows.

Dawn Time and Dusk Time: It controls the day and night lengths. Example, if you set the dawn time to 4, and set the dusk time to 20, the daytime will get the double of time in the timeline than the nighttime. This mean that if you set the 'Day Length' to 30 minutes, the daytime will get 20 minutes while the nighttime will get only 10 minutes of the day-night cycle in the real world time. And the "time of day" should display 6:00am when in the timeline it is 4h, and the "time of day" should display 6:00pm when in the timeline it is 20h. This is why you should not confuse the "Timeline" with the "Time of Day".



Azure Core System

Azure Core System
Version 1.0.0

Time System | Weather System | Weather Properties | Event System | Options

Monday, 06:30 | Current Weather: Default | January 01, 2024

Transition Progress

Global Weather List

Preset 0 - (Default)
Weather Preset: Default (Azure Weather Preset)
Transition Time: 7
Go

Preset 1 - (Sunny)
Preset 2 - (Sunny-Constallation)
Preset 3 - (Fogness)
Preset 4 - (Overcast)
Preset 5 - (CalmRain)

Preset 6 - (Rain)
Weather Preset: Rain (Azure Weather Preset)
Transition Time: 7
Go

Preset 7 - (Storm)
Preset 8 - (Snow)

Weather Zone Trigger: None (Transform)

Weather Zone List
List is Empty

Thunder Settings List

Thunder Prefab - Index 0: Thunder1 (Transform)
X 0 Y 600 Z 1250
Instantiate

Thunder Prefab - Index 1: Thunder2 (Sound Only) (Transform)
X 0 Y 0 Z 0
Instantiate

Weather System:

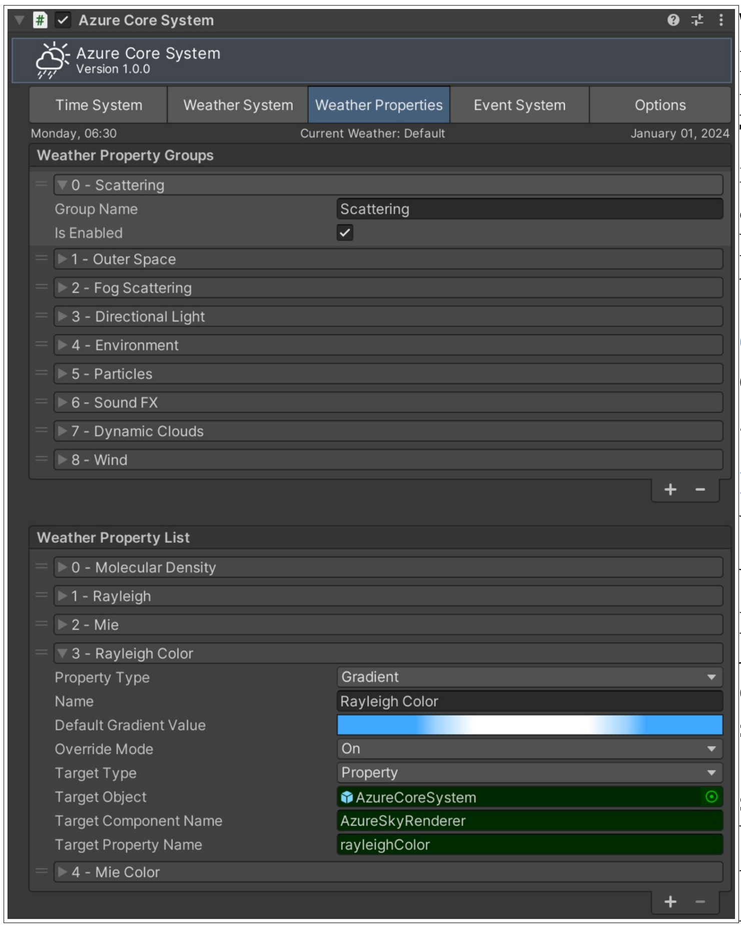
Here you will find the stuffs used to manipulate the weather.

In the 'Global Weather List' you can attach the weather presets, set a time that the weather transition should last when changing the weather to that preset. You can use the 'Go' button for a fast test of the weather transition while in play mode. See later how to change the global weather by code.

In the 'Weather Zone List' you can add the local weather zone game objects, you should organize each weather zone according to its priority. Generally you will set as 'Weather Zone Trigger' the player or the camera transform, so when the trigger enters any local weather zone influence, the weather will change to the weather preset attached to that weather zone. See later how to configure a weather zone.

The 'Thunder Settings List' is responsible for the thunder instantiations. Azure[Sky] comes with two thunder prefabs by default, and after attaching a thunder prefab to any list element, you can set its instantiation position and press the 'Instantiate' button to test it while in the editor play mode. The instantiated thunder prefabs are automatically deleted from the scene when its sound fx ends (*note that the second thunder in the list plays the sound fx only*). To instantiate a thunder prefab by code, you can use the following methods: *Or you can just use your own code and instantiate it as you do for any other prefab.*

```
public void InstantiateThunderPrefab(int index);
public void InstantiateThunderPrefab(int index, Vector3 position);
```


Azure Core System

Azure Core System
Version 1.0.0

Time System | Weather System | Weather Properties | Event System | Options

Monday, 06:30 Current Weather: Default January 01, 2024

Weather Property Groups

0 - Scattering
Group Name: Scattering
Is Enabled: ✓
1 - Outer Space
2 - Fog Scattering
3 - Directional Light
4 - Environment
5 - Particles
6 - Sound FX
7 - Dynamic Clouds
8 - Wind

Weather Property List

0 - Molecular Density
1 - Rayleigh
2 - Mie
3 - Rayleigh Color
Property Type: Gradient
Name: Rayleigh Color
Default Gradient Value
Override Mode: On
Target Type: Property
Target Object: AzureCoreSystem
Target Component Name: AzureSkyRenderer
Target Property Name: rayleighColor
4 - Mie Color

Weather Properties:

Here you can create custom weather properties and organize them into groups. These weather properties will be available in each weather preset to be customized according to its specifications. See later how to create a custom weather property to control variables from other scripts.

Group Name: The name that will be displayed in the weather preset for this group.

Is Enabled: Enables or disables the weather processing of this group.

When you click to expand a weather property group, the '***Weather Property List***' will display all the weather properties created inside that current group. In the screenshot at the left, you can see that the '***Scattering***' weather group is current selected, and in the weather property list below are the weather properties current in that group (*Molecular Density, Rayleigh, Mie, Rayleigh Color and Mie Color*). If you select another weather property group, the weather property list will be updated to show the weather properties of this new selected group.

Property Type: The type that the weather property will be available for customization in the weather presets. It supports the following types: Float, Curve, Color, Gradient, Direction (*Vector3*) and Position (*Vector3*).

Default Value: The default value that weather property will get when the weather preset is reset.

Min Value and Max Value: If the property type is a float or a curve, the Inspector will show the min and max values limits available for customization of this weather property in the weather presets.

Override Mode: Enables the '***Override System***' for this custom weather property.

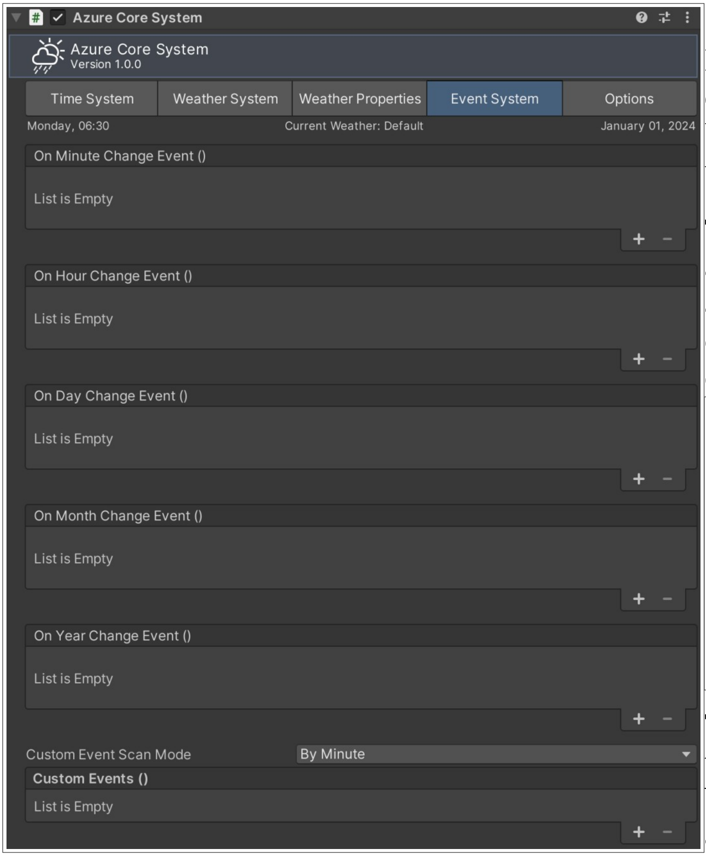
How the '***Override System***' works? When the '***Weather System***' performs all the weather presets blends and transitions, each custom weather property stores its own final resulting values that we can call '***Outputs***'. The outputs stay there to be accessed later to control variables from other scripts. You can access these outputs and sent its values to the target variables using your own custom scripts, or you can just enable and configure the '***Override System***' to do this work for you automatically. All the properties you see for customization in the '***Weather Presets***' are custom '***Weather Properties***' and all of them are using the "***Override System***' configured by default to handle the target variables automatically. The prefab is already configured to use the custom '***Weather Properties***' and the '***Override System***' to control the sky render parameters, the directional light intensity and color, the particles intensity, the sounds volume, the fog parameters, etc…

Target Type: The type of the variable you want to override using the output value from this custom 'Weather Property'. The types supported are: ***Property***, ***Field***, ***Material Property***, ***Global Shader Uniform***, ***Global Property***, ***Global Field***. Note that the target type should match the output type.

Target Object: The game object in the scene that contains the variable you want to override.

Target Component Name: The name of the component in the target object that contains the variable you want to override.

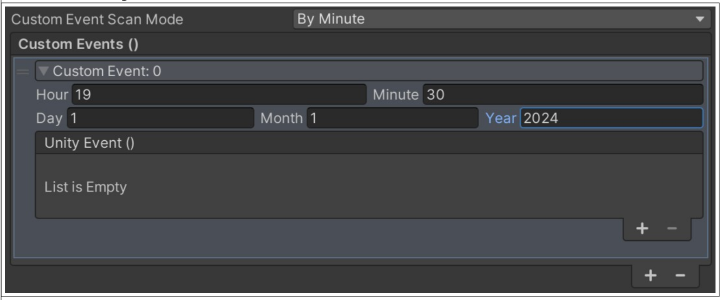
Target Property Name: The name of the variable you want to override in that script component. Note that you should use exactly the same name used in the script, and not the name that shows in the Inspector.


Azure Core System
Azure Core System
Version 1.0.0
Time System | Weather System | Weather Properties | Event System | Options
Monday, 06:30 Current Weather: Default January 01, 2024
On Minute Change Event ()
List is Empty
On Hour Change Event ()
List is Empty
On Day Change Event ()
List is Empty
On Month Change Event ()
List is Empty
On Year Change Event ()
List is Empty
Custom Event Scan Mode: By Minute
Custom Events ()
List is Empty

Event System:

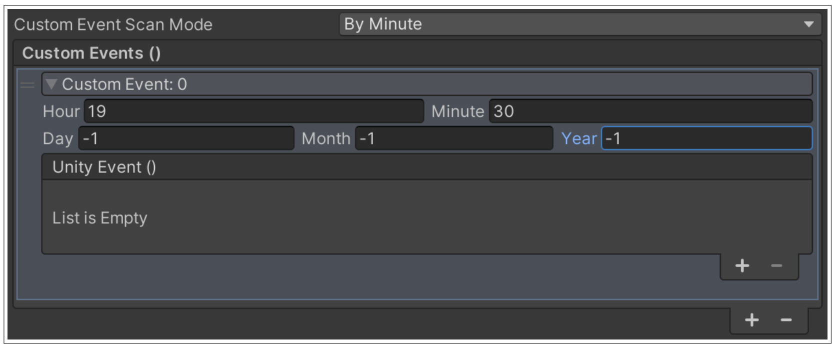
Here are all the events related to the time and date system. It is integrated with the built-in Unity's event system and it is very easy to use.

The '***Custom Events***' are a bit different to use and require a better explanation. When you add a new custom event to the list, you can configure a time and a date to trigger that event, as you can see in the screenshot below.

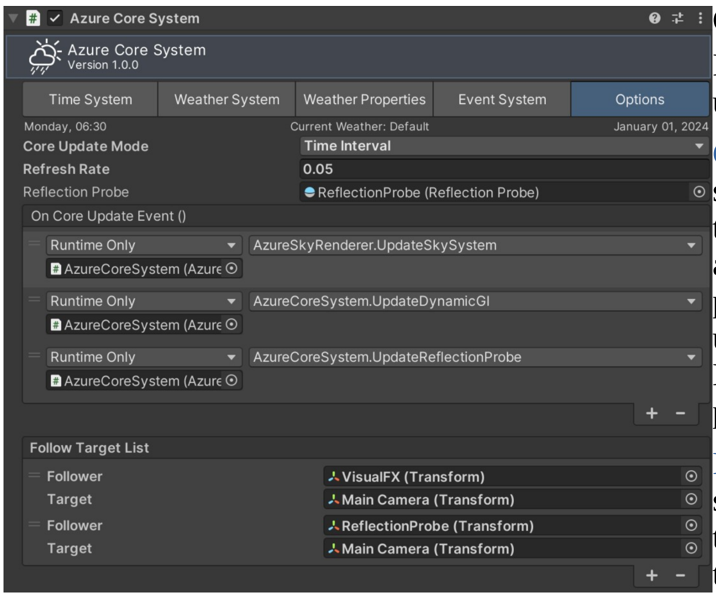

Custom Event Scan Mode: By Minute
Custom Events ()
Custom Event: 0
Hour 19 Minute 30
Day 1 Month 1 Year 2024
Unity Event ()
List is Empty

This custom event is configured to be triggered at *07:30pm January 01, 2024*. When the '***Time System***' reaches that time and date, all the function from the '***UnityEvent()***' list will be executed.

But, if you set the value of '**-1**' to any time or date field, this will force that field to be ignored by the event scanner. In the screenshot below, the event is configured to be triggered every time the 'Time System' reach the time of 07:30pm, ignoring the day, month and year. So, the event below will be executed every day when the time of day get at 07:30pm.


Custom Event Scan Mode: By Minute
Custom Events ()
Custom Event: 0
Hour 19 Minute 30
Day -1 Month -1 Year -1
Unity Event ()
List is Empty

Custom Event Scan Mode: Defines if the event will be scanned every minute or every hour (*in the game time*). Setting it '***By Hour***' is a bit better for performance, but you lose the minute precision check.



Monday, 06:30 Current Weather: Default January 01, 2024

Core Update Mode: Time Interval
Refresh Rate: 0.05
Reflection Probe: ReflectionProbe (Reflection Probe)

On Core Update Event ()

Runtime Only	AzureSkyRenderer.UpdateSkySystem
AzureCoreSystem (Azure	
Runtime Only	AzureCoreSystem.UpdateDynamicGI
AzureCoreSystem (Azure	
Runtime Only	AzureCoreSystem.UpdateReflectionProbe
AzureCoreSystem (Azure	

Follow Target List

Follower	VisualFX (Transform)
Target	Main Camera (Transform)
Follower	ReflectionProbe (Transform)
Target	Main Camera (Transform)

Options:

In the options tab you can find the core update options and the follow targets feature.

Core Update Mode: Defines if the core system will be updated every frame or in time intervals. I recommend using it defined as time intervals because the gain in performance is a big deal. This way you can update the Reflection Probe and Unity Dynamic GI in real time with a low performance cost.

Refresh Rate: The time interval that the core system should be updated. By default it is set to 0.05 because in the demos scene there is the possibility to change the time of day very fast using the timeline slider, so we need a fast refresh time. But if in your game the day-night cycle moves slow and is configured to complete a cycle in 24 minutes or more, you can easily increase the refresh rate value without noticing any freezing/jumping in the rendering. The old GTA games like San Andreas used a similar feature to update the lighting, shadows and time of day with a very high time intervals to save the most performance as possible.

On Core Update Event(): It is a UnityEvent that you can use to update other systems in synchronization with the core system, as you can see, it is used to update the AzureSkyRenderer component as well the Unity's Dynamic GI and also the reflection probe every time the core system is updated.

AzureCoreSystem Execution Order:

- Update Time of Day (*Internally*);
- Update of the Celestial Bodies (*Internally*);
- Update the Weather System (*Internally*);
- Override the Target Properties (*Internally*);
- AzureSkyRenderer.UpdateSkySystem (*OnCoreUpdateEvent*);
- AzureCoreSystem.UpdateDynamicGI (*OnCoreUpdateEvent*);
- AzureCoreSystem.UpdateReflectionProbe (*OnCoreUpdateEvent*);

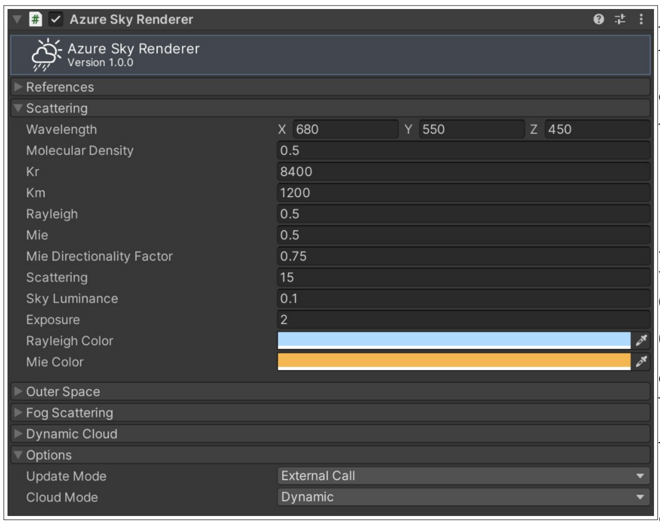
Note that in the options tab of the '***AzureSkyRenderer***' component, the '***Update Mode***' of that component is set by default to '***Externally Call***' instead of '***Locally Every Frame***'.

Follow Target List: This feature is used to update the position of the reflection probe and the particles to always be near the camera as it moves around the scene, otherwise the particles will stay fixed at the start prefab position and the rain/snow will not show if the camera is far away in the scene. Remember to attach your camera or player transform to the '***Target***' field in case you can not see the particles in your scene.

Azure Sky Renderer:

***Component>Azure[Sky] Dynamic Skybox>Azure Sky Renderer*.**

This component handles the render of the skybox and fog scattering effect. The most effective way to know what each parameter does is playing with it and see what is changing in the sky/fog.



Azure Sky Renderer

Azure Sky Renderer
Version 1.0.0

References

Scattering

Wavelength	X 680 Y 550 Z 450
Molecular Density	0.5
Kr	8400
Km	1200
Rayleigh	0.5
Mie	0.5
Mie Directionality Factor	0.75
Scattering	15
Sky Luminance	0.1
Exposure	2
Rayleigh Color	🟦
Mie Color	🟧

Outer Space

Fog Scattering

Dynamic Cloud

Options

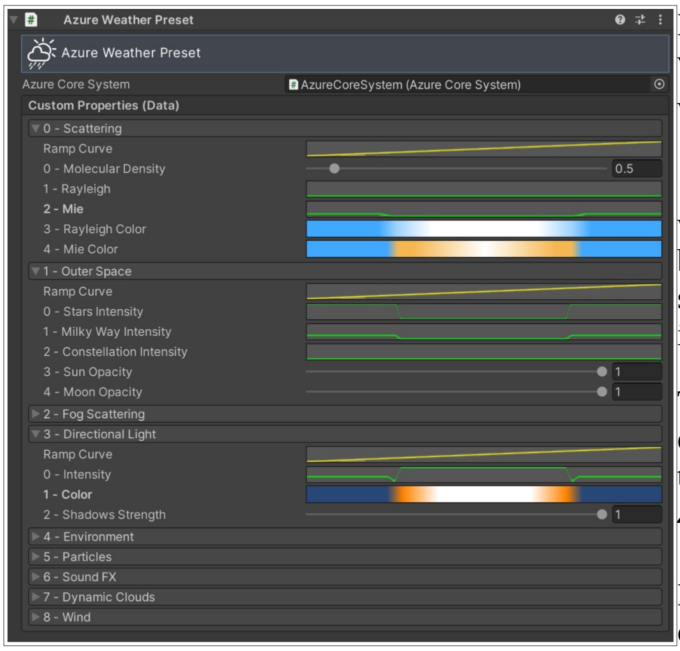
Update Mode	External Call
Cloud Mode	Dynamic

An especial note to the ‘***Update Mode***’ that is set to ‘***External Call***’, this is because the sky update are being handled by the ‘***Update System***’ from the ‘***AzureCoreSystem***’ component.

Important: It is likely that you will try to change some parameters in this component or even in the directional light and you will not be allowed to change, the parameter will appear to be frozen at the same value. This is because these parameters are being handled by the ‘Weather System’ from the ‘AzureCoreSystem’, so when you change them in the Inspector, they will be immediately overridden by the core system.

Azure Weather Preset:

***Component>Azure[Sky] Dynamic Skybox>Azure Weather Preset*.**



Azure Weather Preset

Azure Weather Preset

Azure Core System: AzureCoreSystem (Azure Core System)

Custom Properties (Data)

0 - Scattering
- Ramp Curve
- 0 - Molecular Density: 0.5
- 1 - Rayleigh
- 2 - Mie
- 3 - Rayleigh Color
- 4 - Mie Color

1 - Outer Space
- Ramp Curve
- 0 - Stars Intensity
- 1 - Milky Way Intensity
- 2 - Constellation Intensity
- 3 - Sun Opacity: 1
- 4 - Moon Opacity: 1

2 - Fog Scattering

3 - Directional Light
- Ramp Curve
- 0 - Intensity
- 1 - Color
- 2 - Shadows Strength: 1

4 - Environment

5 - Particles

6 - Sound FX

7 - Dynamic Clouds

8 - Wind

Here is where you customize any weather variation for the ‘***Weather System***’.

When you create and configure a new ‘***Weather Group***’ and add many custom ‘***Weather Properties***’ to it as you want, they will be displayed for customization organized by group tabs. This is the reason you have to set good names for the ‘***Weather Groups***’ and its custom ‘***Weather Properties***’.

The weather presets are used as customization data storage, so you should to reference it in the ‘***Global Weather***’ list or in the ‘***Weather Zones***’ to make use of them.

Each property will be displayed for customization according to its type and the configuration you did in the ‘***AzureCoreSystem***’ component.

Creating a New Weather Preset:

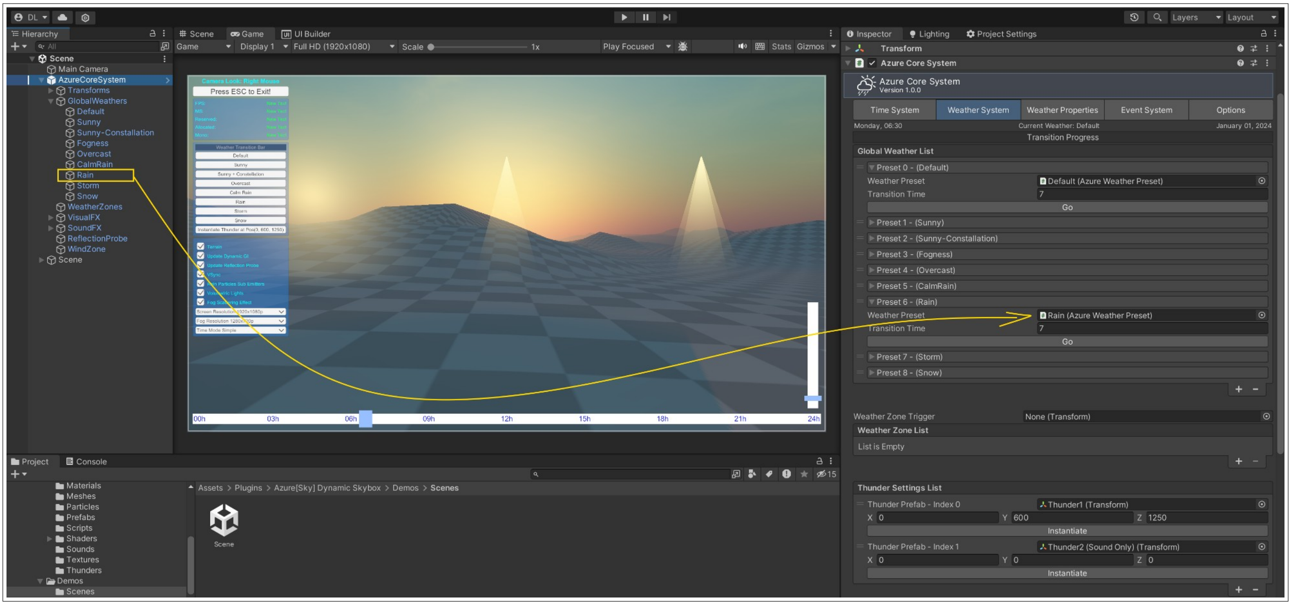
- You can just select the '***Default***' weather preset game object that comes with the prefab.
- Duplicate it using '**Ctrl +D**', and start customizing from there.

Or can create a new one from scratch:

- Create an empty game object.
- Attach the '***AzureWeatherPreset.cs***' script to the new game object.
  Component>Azure[Sky] Dynamic Skybox>Azure Weather Preset**.**
- Reference the '***AzureCoreSystem***' component to the '***Azure Core System***' field in the Inspector, so it can display all the groups and custom weather properties available in that core system component.
- Reset the '***AzureWeatherPreset***' script component using the Unity's component menu. This will force all the properties to get the default values configured for the custom '***Weather Properties***' in the '***AzureCoreSystem***' component.

Creating a New Global Weather:

Create a new element to the '***Global Weather***' list in the '***Weather System***' tab of the '***AzureCoreSystem***' component and attach to it the new '***Weather Preset***' that you just created.



Transform
Azure Core System
Azure Core System
Version 1.0.0

Time System | Weather System | Weather Properties | Event System | Options

Monday, 06:30 Current Weather: Default January 01, 2024
Transition Progress

Global Weather List
Preset 0 - (Default)
Weather Preset: Default (Azure Weather Preset)
Transition Time: 7
Go
Preset 1 - (Sunny)
Preset 2 - (Sunny-Constallation)
Preset 3 - (Fogness)
Preset 4 - (Overcast)
Preset 5 - (CalmRain)
Preset 6 - (Rain)
Weather Preset: Rain (Azure Weather Preset)
Transition Time: 7
Go
Preset 7 - (Storm)
Preset 8 - (Snow)

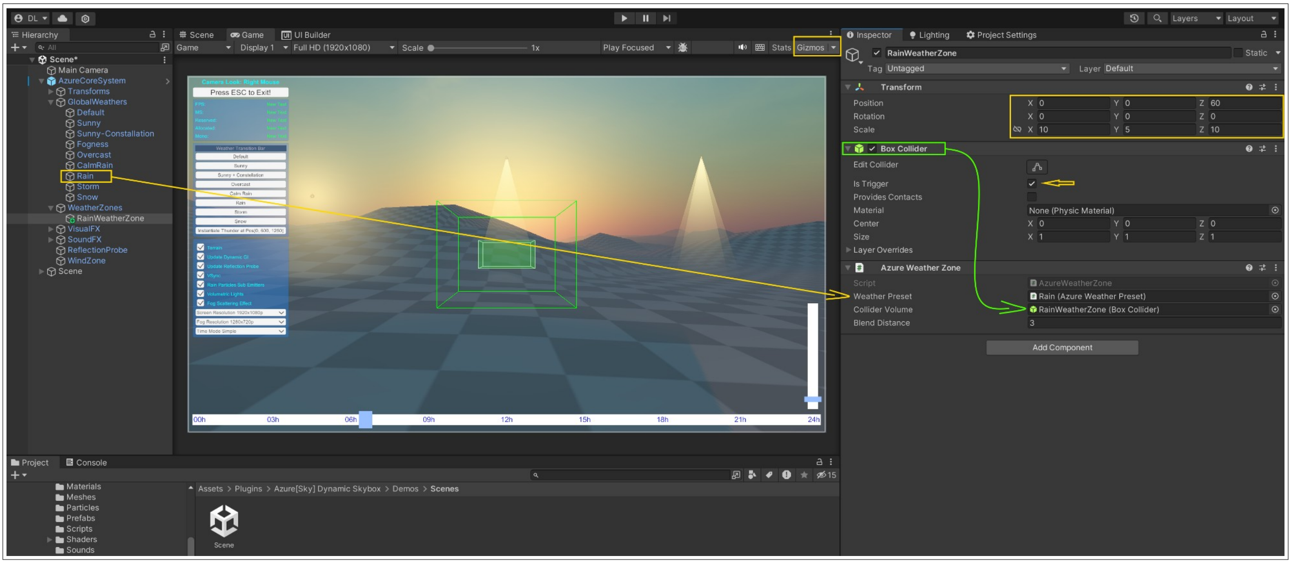
Weather Zone Trigger: None (Transform)
Weather Zone List
List is Empty

Thunder Settings List
Thunder Prefab - Index 0: Thunder1 (Transform)
X 0 Y 600 Z 1250
Instantiate
Thunder Prefab - Index 1: Thunder2 (Sound Only) (Transform)
X 0 Y 0 Z 0
Instantiate

Creating a Weather Zone:

- Create an empty game object in the scene.
- Add a '***Box Collider***' to it and check the '***Is Trigger***' option.
- Add the '***AzureWeatherZone.cs***' component to it.
  Component>Azure[Sky] Dynamic Skybox>Azure Weather Zone**.**
- Attach some '***Weather Preset***' game object to the '***Weather Preset***' field in the Inspector, example attach the '***Rain***' weather preset ti the field in the Inspector.
- Reference the '***Box Collider***' component to the '***Collider Volume***' field in the Inspector.

- Configure the ‘Blend Distance’ for a smooth transition.
- Enable the ‘***Gizmos***’, so that the weather zone volume can be seen in the ‘***Game***’ window.
- For this example, configure the ‘***Transform***’, ‘***Box Collider***’ and the ‘***AzureWeatherZone***’ as the screenshot below.

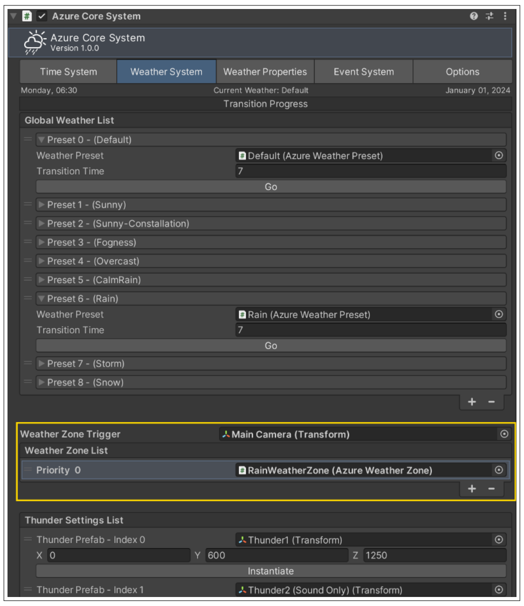

RainWeatherZone
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 60
Rotation X 0 Y 0 Z 0
Scale X 10 Y 5 Z 10

Box Collider
Is Trigger ✓
Provides Contacts
Material None (Physic Material)
Center X 0 Y 0 Z 0
Size X 1 Y 1 Z 1

Azure Weather Zone
Script AzureWeatherZone
Weather Preset Rain (Azure Weather Preset)
Collider Volume RainWeatherZone (Box Collider)
Blend Distance 3

When the gizmos is enabled, you can see the collider volume in the Game window. As you can see in the screenshot above, the filled green box is the volume area, and the wire-frame green box is the transition area for that weather zone. The weather will be fully changed to that weather zone when the ‘***Weather Zone Trigger***’ is completely inside the volume area (*filled green box*).


Azure Core System
Version 1.0.0
Time System | Weather System | Weather Properties | Event System | Options
Monday, 06:30 Current Weather: Default January 01, 2024
Transition Progress

Global Weather List
Preset 0 - (Default)
Weather Preset Default (Azure Weather Preset)
Transition Time 7
Go
Preset 1 - (Sunny)
Preset 2 - (Sunny-Constallation)
Preset 3 - (Fogness)
Preset 4 - (Overcast)
Preset 5 - (CalmRain)
Preset 6 - (Rain)
Weather Preset Rain (Azure Weather Preset)
Transition Time 7
Go
Preset 7 - (Storm)
Preset 8 - (Snow)

Weather Zone Trigger Main Camera (Transform)
Weather Zone List
Priority 0 RainWeatherZone (Azure Weather Zone)

Thunder Settings List
Thunder Prefab - Index 0 Thunder1 (Transform)
X 0 Y 600 Z 1250
Instantiate
Thunder Prefab - Index 1 Thunder2 (Sound Only) (Transform)

- Now select the ‘***AzureCoreSystem***’ component and add a new element to the ‘***Weather Zone List***’.
- Attach the weather zone game object we just created to the object field in the list.
- A do not forget to attach the camera to the ‘Weather Zone Trigger’ field that is located above the weather zone list.
- Now play the game and move the camera inside the weather zone, the weather should change to the weather preset attached in the weather zone component.

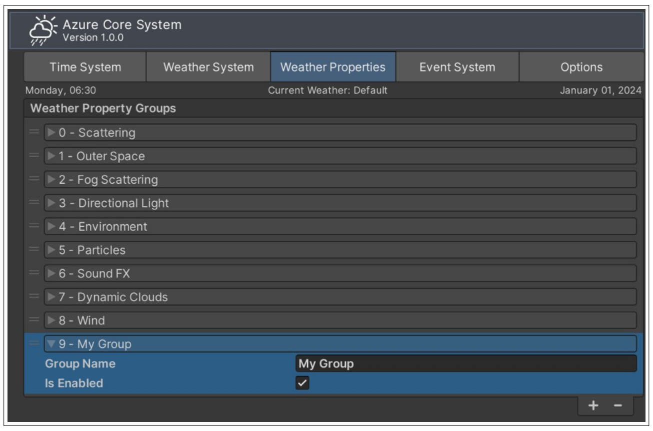
The weather zone feature will be ignored if there is no weather zone trigger selected.

You can have one weather zone volume inside another, but you need to order it in the weather zone list according to its priorities.

The weather system will ignore the weather zone game objects not attached to the weather zone list.

Configuring a New Weather Property:

Select the ‘***Weather Properties***’ tab in the header toolbar of the ‘***AzureCoreSystem***’ component and create a new ‘***Weather Property Group***’.



Azure Core System
Version 1.0.0

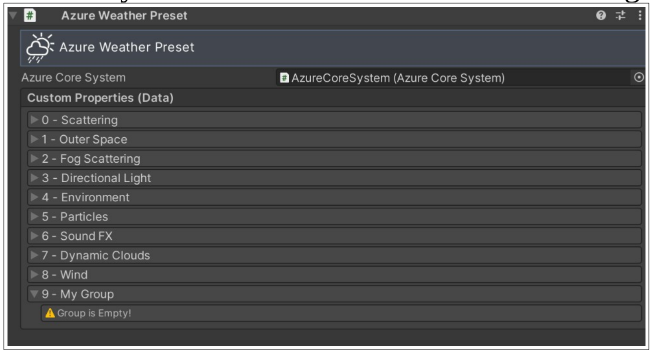
Time System | Weather System | Weather Properties | Event System | Options

Monday, 06:30 Current Weather: Default January 01, 2024

Weather Property Groups
- 0 - Scattering
- 1 - Outer Space
- 2 - Fog Scattering
- 3 - Directional Light
- 4 - Environment
- 5 - Particles
- 6 - Sound FX
- 7 - Dynamic Clouds
- 8 - Wind
- 9 - My Group

Group Name: My Group
Is Enabled: ✓

Select any ‘***Weather Preset***’ to see that this new group is now available in its Inspector.



Azure Weather Preset

Azure Core System: AzureCoreSystem (Azure Core System)

Custom Properties (Data)
- 0 - Scattering
- 1 - Outer Space
- 2 - Fog Scattering
- 3 - Directional Light
- 4 - Environment
- 5 - Particles
- 6 - Sound FX
- 7 - Dynamic Clouds
- 8 - Wind
- 9 - My Group

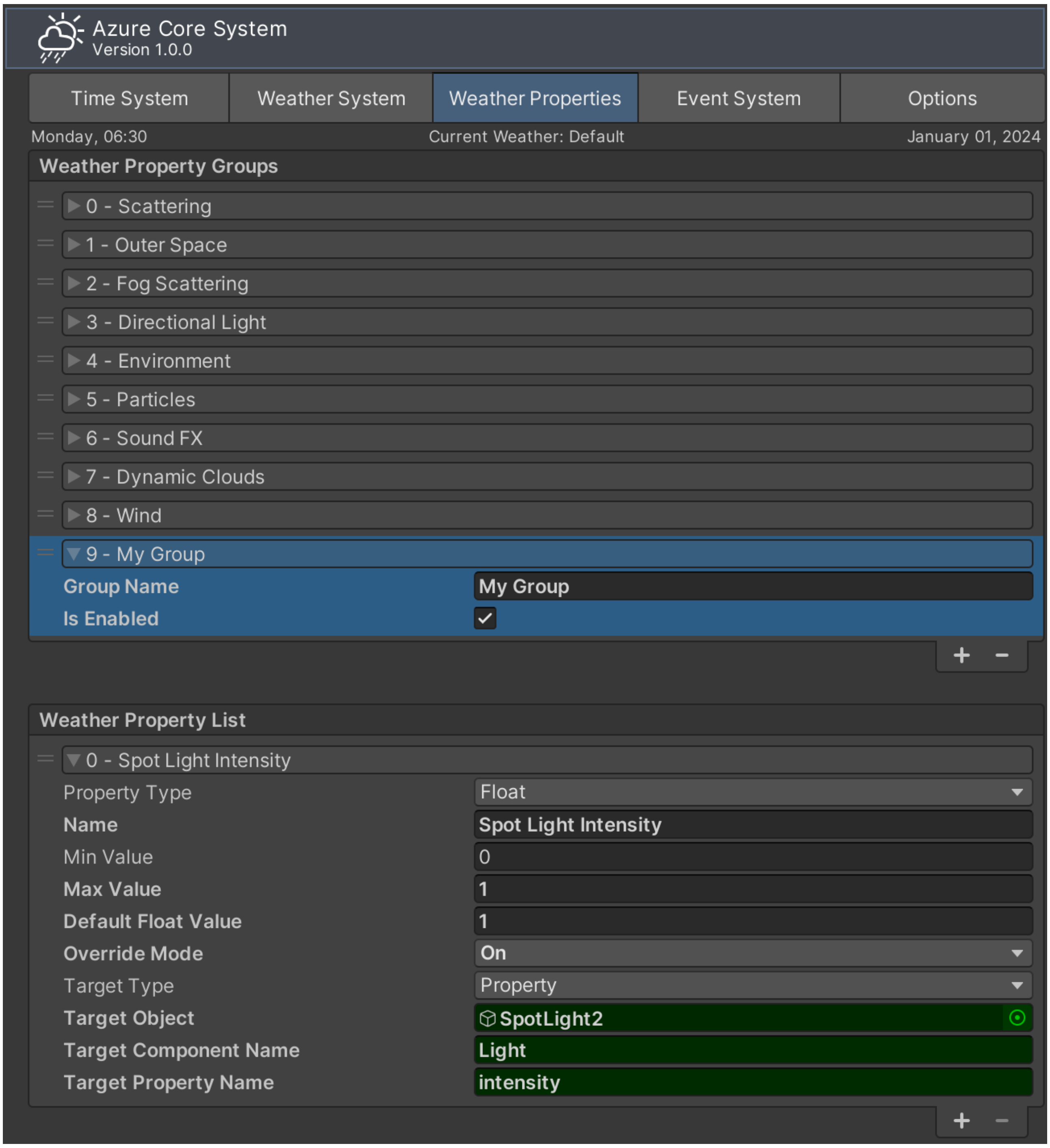
⚠ Group is Empty!

As you can see, the new group is there, but there is a message saying is is empty. So back to the ‘***AzureCoreSystem***’ and with this new group selected in the ‘***Weather Group***’ list, add a new element to the ‘***Weather Property***’ list.

- Set the ‘***Property Type***’ to ‘***Float***’. This custom property will be used to control the intensity of a spot light in the scene, so its type should return a float value. You can also set it as a ‘Curve’ if you need different values along the day cycle progression.
- Set the ‘***Name***’ as ‘***Spot Light Intensity***’ or any name you want.
- Set the ‘***Min Value***’ to 0.
- Set the ‘***Max Value***’ to 1.
- Set the ‘***Default Value***’ to 1. So every time you create a new weather preset or reset it, the custom property in the weather preset will start with this value as default.
- Set the ‘***Override Mode***’ to ‘***On***’. Using the override system avoid the use of scripting to send the output value from the custom weather property to the target variable.
- Set the ‘***Target Type***’ to ‘***Property***’ because the Unity’s code standards generally encapsulate the fields into properties. If your target property is not encapsulated, set it as ‘***Field***’ instead.
- Drag one of the volumetric spot lights from the demo scene to the ‘***Target Object***’ field.

- Set the “***Target Component Name***’ to ‘***Light***’, because we want to override a variable located in the ‘***Light***’ component of the spot light game object.
- Set the ‘***Target Property Name***’ to ‘***intensity***’ because we want to override the intensity property of the light component.

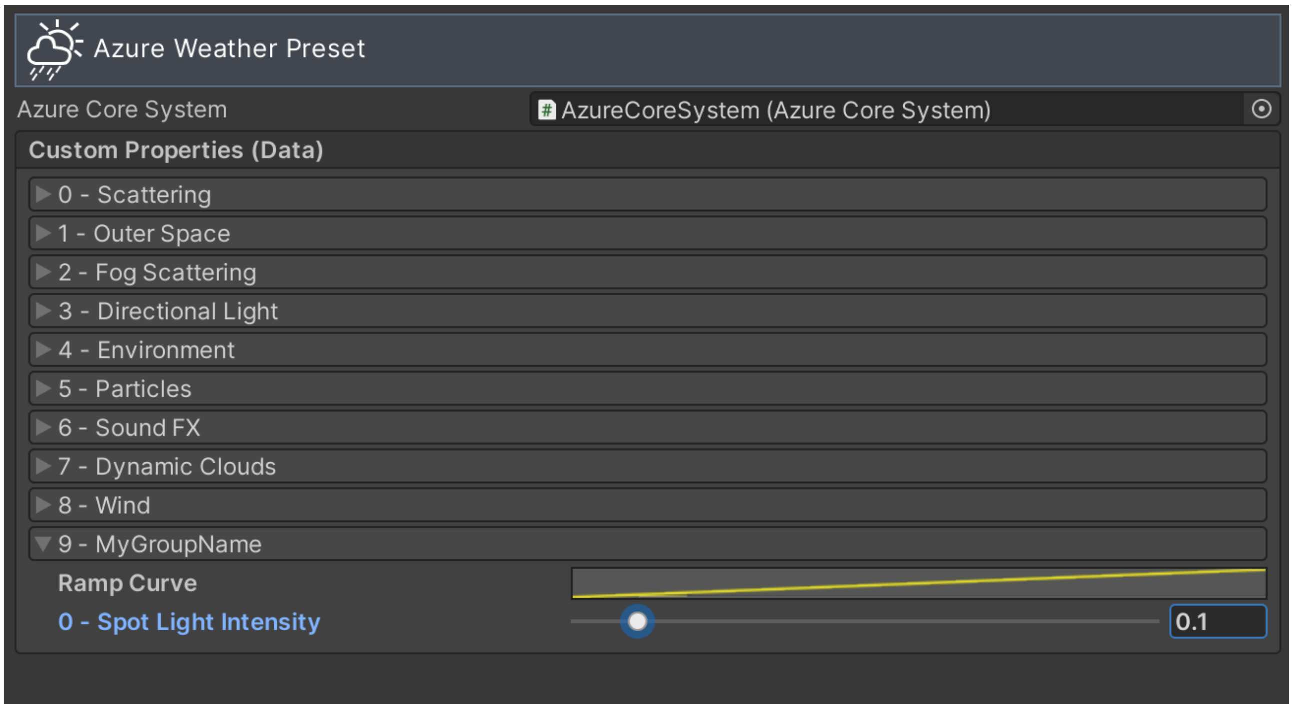
This is more or less how your configuration should look:



Group Name	My Group
Is Enabled	✓

Property Type	Float
Name	Spot Light Intensity
Min Value	0
Max Value	1
Default Float Value	1
Override Mode	On
Target Type	Property
Target Object	SpotLight2
Target Component Name	Light
Target Property Name	intensity

Now select the '***Default***' weather preset in the prefab. If all goes well, the new custom weather property should be available for customization inside the new group tab. It should be with a zero value because the weather preset were already created and were not reset to its default value, so set a value to 0.1, otherwise the spot light should not be visible in the scene.



Azure Weather Preset

Azure Core System: AzureCoreSystem (Azure Core System)

Custom Properties (Data)

- 0 - Scattering
- 1 - Outer Space
- 2 - Fog Scattering
- 3 - Directional Light
- 4 - Environment
- 5 - Particles
- 6 - Sound FX
- 7 - Dynamic Clouds
- 8 - Wind
- 9 - MyGroupName
  - Ramp Curve
  - 0 - Spot Light Intensity: 0.1

Select any other weather preset, for example the '***Rain***' weather preset and set a value of 1.0.

Now play the game and alternate the weather between the '***Default***' and the '***Rain***' using the '***Go***' button located in the '***Global Weather***' list. The intensity of the spot light should be changing according to the selected weather.

Important: If you try to change the intensity of that spot light directly in its '***Light***' component, you will not be able because the value will be immediately overridden by the core system.

You can use this same process to control any other variable you want, you just need to care about its type and configure the custom 'Weather Property' according it. Take a look at the other 'Weather Properties' configuration to understand better how it was configured by default. All the custom weather properties types was already in use by the default prefab settings.

How to Change the Global Weather by Scripting:

```
using UnityEngine;
using UnityEngine.AzureSky; // Always include it to be able to access the Azure[Sky] classes.

public class Example : MonoBehaviour
{
    // The reference to the core system component.
    // You need to reference it manually in the Inspector.
    public AzureCoreSystem azureCoreSystem;

    private void Start()
    {
        // Changing the global weather to the 'Storm' weather preset.
        // Note that in the 'Global Weather List', by default the 'Storm' weather preset...
        // is attached to the element number 7. We need to pass as parameter, the element number
        // of the target weather preset in the global weather list.
        azureCoreSystem.weatherSystem.SetGlobalWeather(7);
    }
}
```

How to Change the Timeline by Scripting:

```
using UnityEngine;
using UnityEngine.AzureSky; // Always include it to be able to access the Azure[Sky] classes.

public class Example : MonoBehaviour
{
    // The reference to the core system component.
    // You need to reference it manually in the Inspector.
    public AzureCoreSystem azureCoreSystem;

    private void Update()
    {
        if (Input.GetKeyUp(KeyCode.Space))
        {
            // Accessing the time system instance inside the core system and setting the
            // timeline to 12 hours, note that the timeline requires a value between 0 and 24.
            azureCoreSystem.timeSystem.timeline = 12.0f;
        }
    }
}
```

How to Change the Date by Scripting:

```
using UnityEngine;
using UnityEngine.AzureSky; // Always include it to be able to access the Azure[Sky] classes.

public class Example : MonoBehaviour
{
    // The reference to the core system component.
    // You need to reference it manually in the Inspector.
    public AzureCoreSystem azureCoreSystem;

    private void Update()
    {
        if (Input.GetKeyUp(KeyCode.Space))
        {
            // Accessing the time system instance inside the core system and setting the
            // date to January 1, 2024. Note the parameters ordering (day, month, year).
            azureCoreSystem.timeSystem.SetDate(1, 1, 2024);
        }
    }
}
```

Overriding a Target Variable by Scripting:

You may opt to disable the ‘Override System’ of all the custom ‘Weather Properties’ because it can generated memory garbage for the use of ‘Reflection’. The use of ‘Reflection’ is required to be able to access a target variable using its string name only. So, if you want to prevent garbage, the way is to disable the ‘Override System’ in each custom ‘Weather Property’ and send its outputs value manually using your custom script.

```
using UnityEngine;
using UnityEngine.AzureSky; // Always include it to be able to access the Azure[Sky] classes.

public class Example : MonoBehaviour
{
    // The reference to the core system component.
    // You need to reference it manually in the Inspector.
    public AzureCoreSystem azureCoreSystem;

    // The reference to a Light component.
    // You need to reference it manually in the Inspector.
    public Light myLight;

    // Registering to Azure[Sky] event.
    private void OnEnable()
    {
        AzureNotificationCenter.OnAfterWeatherSystemUpdate += OnAfterWeatherSystemUpdate;
    }

    // Unregistering from Azure[Sky] event.
    private void OnDisable()
    {
        AzureNotificationCenter.OnAfterWeatherSystemUpdate -= OnAfterWeatherSystemUpdate;
    }

    // It will be executed every time the Weather System is updated by the AzureCoreSystem component.
    // It is good for performance because it may not be executed every frame if you are using the
    // 'Core Update Mode' defined to 'Time Interval'.
    private void OnAfterWeatherSystemUpdate(AzureWeatherSystem azureWeatherSystem)
    {
        // It gets the output float value from the group element number 3 and
        // the weather property element number 0 and send it to the target light intensity.
        // If you check in the weather preset (default prefab setting) you will see that
        // it is referencing to the 'intensity' weather property of the 'Directional Light' group.
        myLight.intensity = azureCoreSystem.weatherSystem.GetFloatOutput(3, 0);

        // It gets the output color value from the group element number 3 and
        // the weather property element number 1 and send it to the target light color.
        // It is referencing to the 'Color' weather property from the 'Directional Light' group.
        myLight.color = azureCoreSystem.weatherSystem.GetColorOutput(3, 1);
    }
}
```

Adding Fog Scattering Support to Custom Transparent Shaders:

1) - Add this line of code shortly after the CGPROGRAM starts.

```
#include "Assets/Plugins/Azure[Sky] Dynamic Skybox/Core/Shaders/Transparent/AzureFogCore.cginc"
```

2) - Add 1 extra TEXCOORD definition to the "vertex to fragment" struct to store the world position.

```
struct v2f
{
    float3 worldPos : TEXCOORD0;
};
```

3) - Now in the vertex shader, it is necessary to calculate the vertex world position and store it in the extra TEXCOORD that we just created.

```
o.worldPos = mul(unity_ObjectToWorld, v.vertex);
```

4) - And finally, you need to apply the fog scattering color to the output color of the fragment shader(pixel shader). Using the following command:

```
col = ApplyAzureFog(col, i.worldPos);
```

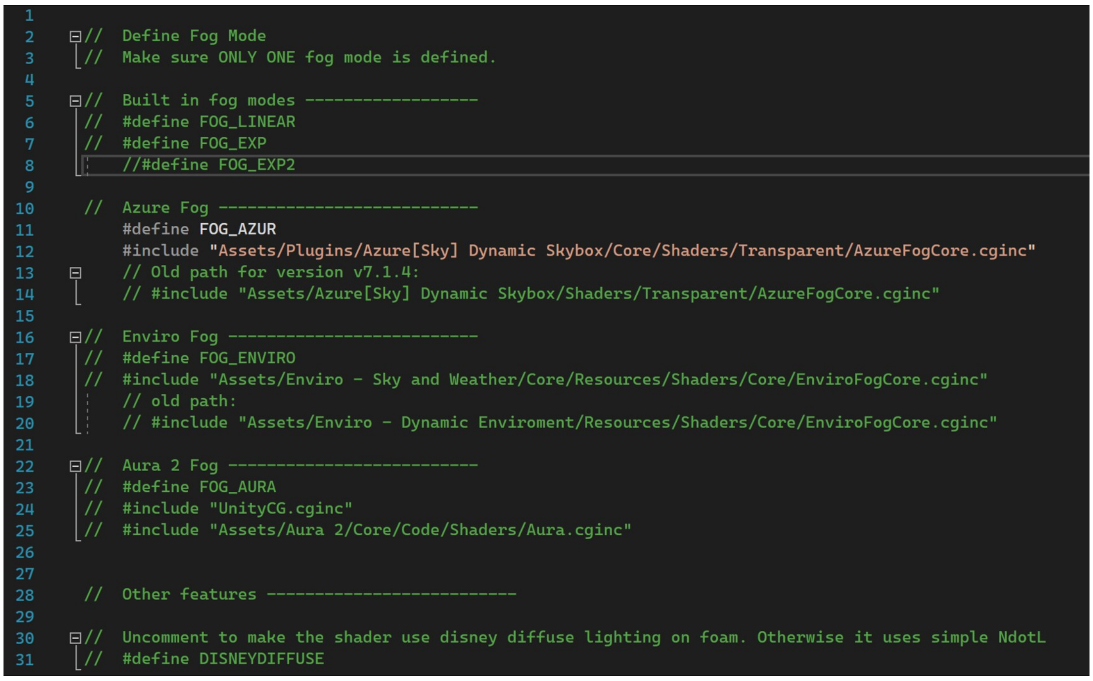
Modified Shader Example to Support the Fog Scattering Effect:

```
Shader "ModifiedShaderExample"
{
    Properties
    {
        _MainTex ("Base (RGB) Trans (A)", 2D) = "white" {}
    }

    SubShader
    {
        Tags {"Queue"="Transparent" "IgnoreProjector"="True" "RenderType"="Transparent"}
        LOD 100

        ZWrite Off
        Blend SrcAlpha OneMinusSrcAlpha

        Pass
        {
            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag
            #pragma target 2.0
            #pragma multi_compile_fog
            #include "UnityCG.cginc"

            // Azure[Sky] - Start of the Instruction number 1
            #include "Assets/Plugins/Azure[Sky] Dynamic Skybox/Core/Shaders/Transparent/AzureFogCore.cginc"
            // Azure[Sky] - End of the Instruction number 1

            sampler2D _MainTex;
            float4 _MainTex_ST;

            struct appdata_t
            {
                float4 vertex : POSITION;
                float2 texcoord : TEXCOORD0;
                UNITY_VERTEX_INPUT_INSTANCE_ID
            };

            struct v2f
            {
                float4 vertex : SV_POSITION;
                float2 texcoord : TEXCOORD0;
                UNITY_FOG_COORDS(1)

                // Azure[Sky] - Start of the Instruction number 2
                float3 worldPos : TEXCOORD2;
                // Azure[Sky] - End of the Instruction number 2

                UNITY_VERTEX_OUTPUT_STEREO
            };

            v2f vert (appdata_t v)
            {
                v2f o;
                UNITY_SETUP_INSTANCE_ID(v);
                UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o);
                o.vertex = UnityObjectToClipPos(v.vertex);
                o.texcoord = TRANSFORM_TEX(v.texcoord, _MainTex);

                // Azure[Sky] - Start of the Instruction number 3
                o.worldPos = mul(unity_ObjectToWorld, v.vertex);
                // Azure[Sky] - End of the Instruction number 3

                UNITY_TRANSFER_FOG(o, o.vertex);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                fixed4 col = tex2D(_MainTex, i.texcoord);
                UNITY_APPLY_FOG(i.fogCoord, col);

                // Azure[Sky] - Start of the Instruction number 4
                col = ApplyAzureFog(col, i.worldPos);
                // Azure[Sky] - End of the Instruction number 4

                return col;
            }

            ENDCG
        }
    }
}
```

Adding Fog Support to Lux Water:
https://assetstore.unity.com/packages/vfx/shaders/lux-water-119244

- Azure[Sky] side comes fully integrated with Lux Water by default. So you do not need to change anything in Azure[Sky] side.

- In LuxWater you need to edit the “LuxWater_Setup.cginc” file located in:
Assets/LuxWater/Shaders/Includes
*A*nd uncomment the #define and #include corresponding to Azure[Sky] fog and make sure to comment all the other #defines. Note that for the Azure[Sky] version cycle 8.0.0 and higher, the package path were changed and you need to edit it in Lux Water file, maybe the Lux Water developer already did it in future updates. The “LuxWater_Setup.cginc” file should be edited like in the screenshot below.



```

//  Define Fog Mode
//  Make sure ONLY ONE fog mode is defined.

//  Built in fog modes ------------------
//  #define FOG_LINEAR
//  #define FOG_EXP
    //#define FOG_EXP2

//  Azure Fog ---------------------------
    #define FOG_AZUR
    #include "Assets/Plugins/Azure[Sky] Dynamic Skybox/Core/Shaders/Transparent/AzureFogCore.cginc"
    // Old path for version v7.1.4:
    // #include "Assets/Azure[Sky] Dynamic Skybox/Shaders/Transparent/AzureFogCore.cginc"

//  Enviro Fog --------------------------
//  #define FOG_ENVIRO
//  #include "Assets/Enviro - Sky and Weather/Core/Resources/Shaders/Core/EnviroFogCore.cginc"
    // old path:
    // #include "Assets/Enviro - Dynamic Enviroment/Resources/Shaders/Core/EnviroFogCore.cginc"

//  Aura 2 Fog --------------------------
//  #define FOG_AURA
//  #include "UnityCG.cginc"
//  #include "Assets/Aura 2/Core/Code/Shaders/Aura.cginc"


//  Other features --------------------------

//  Uncomment to make the shader use disney diffuse lighting on foam. Otherwise it uses simple NdotL
//  #define DISNEYDIFFUSE
```

- Then right-click over the “LuxWaterCG” and “LuxWater CG Tessellation” shaders, and reimport it.

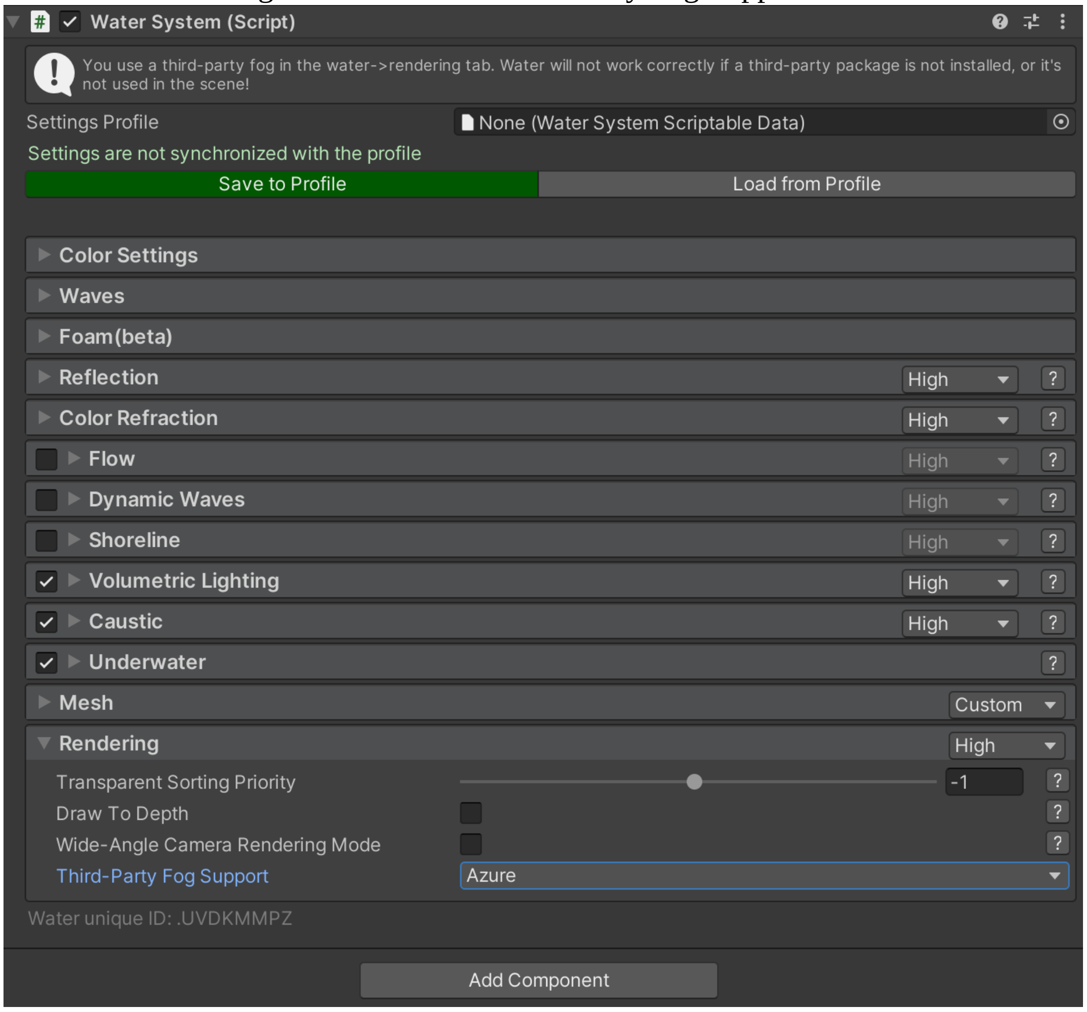
- In the camera you have the fog scattering effect, reorder the Azure[Sky] fog scattering script to be rendererd before the LuxWater_UnderWaterRendering script. This way the under water effect will work properly.

- Set the Directional Light from the Azure[Sky] prefab to the “Sun” property of the LuxWater_UnderWaterRendering script.

Adding Fog Support to KWS Water System:
https://assetstore.unity.com/packages/tools/particles-effects/kws-water-system-standard-rendering-191771

- Just select the Water System in your scene.

- Go to the 'Rendering' tab and set the 'Third-Party Fog Support' to 'Azure'.



Water System (Script)

You use a third-party fog in the water->rendering tab. Water will not work correctly if a third-party package is not installed, or it's not used in the scene!

Settings Profile: None (Water System Scriptable Data)

Settings are not synchronized with the profile

	Reflection	High
	Color Refraction	High
☐	Flow	High
☐	Dynamic Waves	High
☐	Shoreline	High
✓	Volumetric Lighting	High
✓	Caustic	High
✓	Underwater	
	Mesh	Custom
	Rendering	High

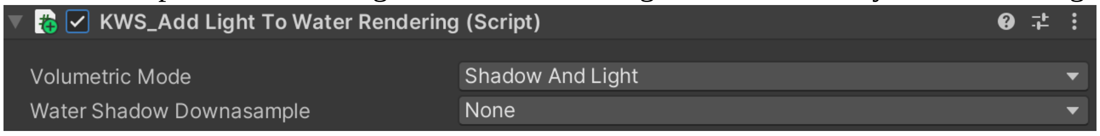
Transparent Sorting Priority	-1
Draw To Depth	☐
Wide-Angle Camera Rendering Mode	☐
Third-Party Fog Support	Azure

Water unique ID: .UVDKMMPZ

- The KWS Water System will scan the Azure[Sky] package files and automatically include the AzureFogCore.cginc to their shaders, and recompiling its shaders. So firstly, the Azure[Sky] package should be imported in the project.

- Make sure the Azure[Sky] prefab is in the scene.

- Add the script 'KWS_AddLightToWaterRendering' to the Azure[Sky] directional light.



KWS_Add Light To Water Rendering (Script)

Volumetric Mode	Shadow And Light
Water Shadow Downsample	None

Adding Fog Support to Crest Ocean System – GitHub Version:

https://github.com/wave-harmonic/crest

- Open the Crest's Ocean.shader and add this line of code along with the other #include files of the first shader pass. This will import the fog data to be used in the shader.
```
#include "Assets/Plugins/Azure[Sky] Dynamic Skybox/Core/Shaders/Transparent/AzureFogCore.cginc"
```

- The Crest shader by default already have the world position coordinates, so it is not necessary to add new calculations since we can use the coordinates that are already in the shader.

- Replace in the fragment shader the Unity built-in fog implementation:
```
UNITY_APPLY_FOG(input.fogCoord, col);
```
By the Azure[Sky] fog implementation:
```
col = ApplyAzureFog(float4(col, 1.0f), input.worldPos);
```

- Remove any other directional light source from the scene, the Azure[Sky] prefab already have a directional light.

- Add the Azure[Sky] directional light to the "Primary Light" on the "Ocean Render" component.

- Add the Azure[Sky] directional light to the "Sun Source" on the Unity Lighting window.

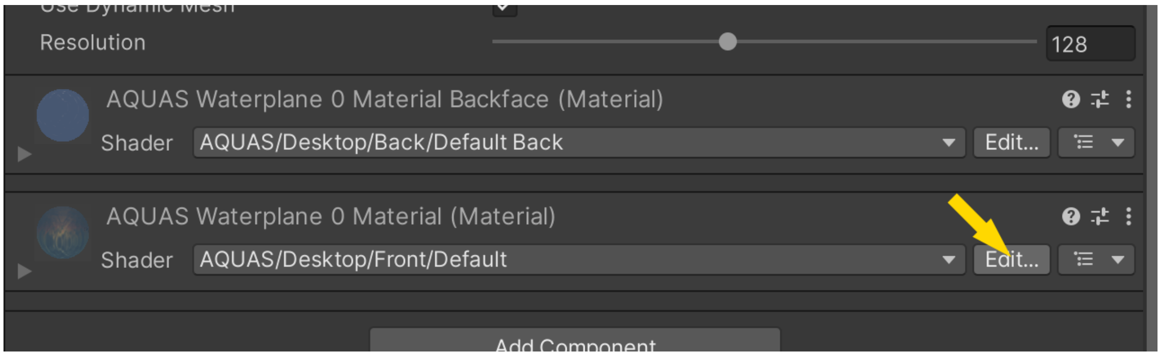
- Remember to add the Azure[Sky] fog scattering effect to the camera, so the other game objects get the fog too. But always place it before the underwater post processing effect scripts.

Adding Fog Support To Aquas Water:

https://assetstore.unity.com/packages/tools/particles-effects/aquas-built-in-render-pipeline-138749

- In the scene, select the 'Aquas Waterplane' game object.

- In the Inspector, click in the 'Edit' button of the material tab. This will make sure you will be editing the correct shader current in use. For this example, I chose the shader "*AQUAS/Desktop/Front/Default*"

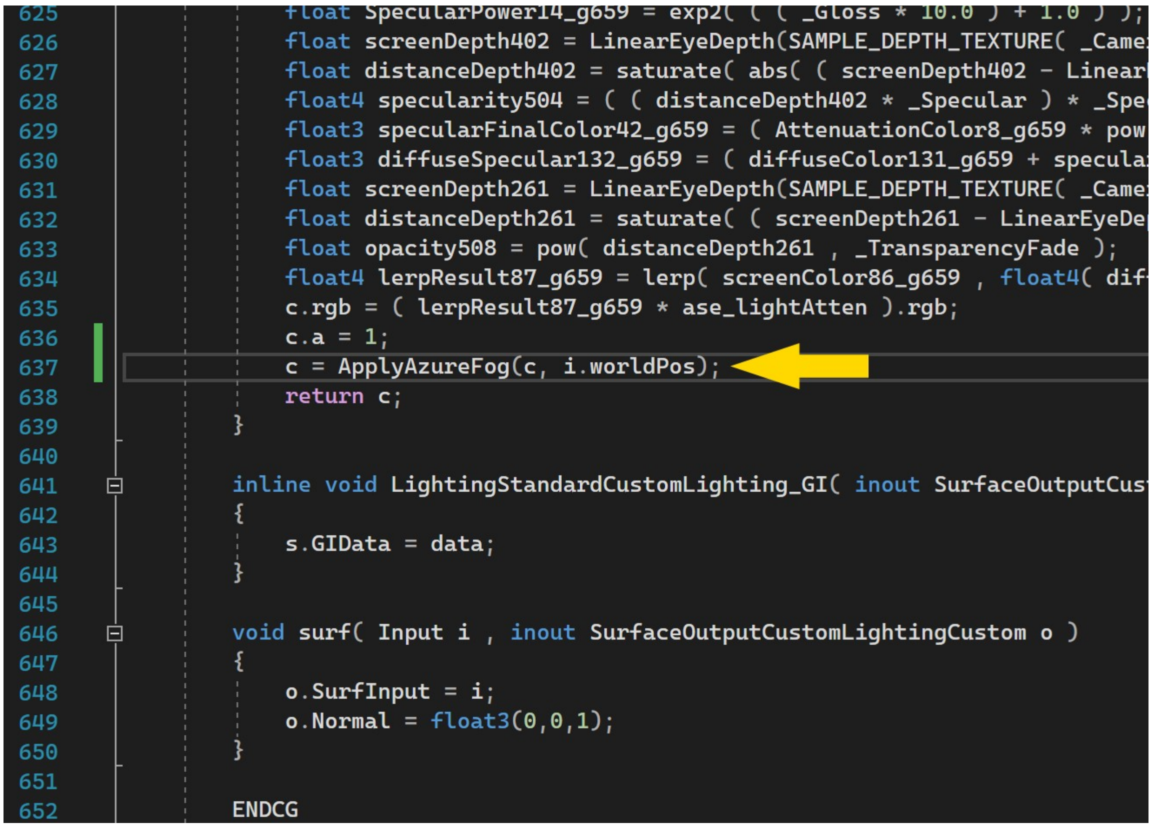

Resolution 128

AQUAS Waterplane 0 Material Backface (Material)
Shader AQUAS/Desktop/Back/Default Back

AQUAS Waterplane 0 Material (Material)
Shader AQUAS/Desktop/Front/Default

- Add this line of code along with the other #include files of the first shader pass. This will import the fog data to be used in the shader.
```
#include "Assets/Plugins/Azure[Sky] Dynamic Skybox/Core/Shaders/Transparent/AzureFogCore.cginc"
```

- Add this code just before the return of the ‘LightingStandardCustomLighting’ function.
```
c = ApplyAzureFog(c, i.worldPos);
```



```
        float SpecularPower14_g659 = exp2( ( ( _Gloss * 10.0 ) + 1.0 ) );
        float screenDepth402 = LinearEyeDepth(SAMPLE_DEPTH_TEXTURE( _Came
        float distanceDepth402 = saturate( abs( ( screenDepth402 - Linear
        float4 specularity504 = ( ( distanceDepth402 * _Specular ) * _Spe
        float3 specularFinalColor42_g659 = ( AttenuationColor8_g659 * pow
        float3 diffuseSpecular132_g659 = ( diffuseColor131_g659 + specula
        float screenDepth261 = LinearEyeDepth(SAMPLE_DEPTH_TEXTURE( _Came
        float distanceDepth261 = saturate( ( screenDepth261 - LinearEyeDe
        float opacity508 = pow( distanceDepth261 , _TransparencyFade );
        float4 lerpResult87_g659 = lerp( screenColor86_g659 , float4( dif
        c.rgb = ( lerpResult87_g659 * ase_lightAtten ).rgb;
        c.a = 1;
        c = ApplyAzureFog(c, i.worldPos);
        return c;
    }

    inline void LightingStandardCustomLighting_GI( inout SurfaceOutputCus
    {
        s.GIData = data;
    }

    void surf( Input i , inout SurfaceOutputCustomLightingCustom o )
    {
        o.SurfInput = i;
        o.Normal = float3(0,0,1);
    }

    ENDCG
```

- This should be enough. As I can see, for any other shader variation it is the same process.

- Remember that it is necessary to have the Azure[Sky] prefab in the scene, also attach the Azure[Sky] fog scattering effect to the camera before the Aquas under water effect.

Adding Fog Support to Hidroform Ocean System:

https://assetstore.unity.com/packages/tools/particles-effects/hydroform-ocean-system-76960

- Open the **WaterFunc.cginc** file and add this line at the top in any place, but make sure to not put it inside some function. This will import the fog data to be used in the ocean shader.
```
#include "Assets/Plugins/Azure[Sky] Dynamic Skybox/Core/Shaders/Transparent/AzureFogCore.cginc"
```

- In the same file, change the "**return**" of the **calcPixel** function from this:
```
return saturate( waterColor );  // saturate fixes issues with Linear/HDR turned on
```
To this one:
```
return saturate(ApplyAzureFog(saturate(waterColor), inData.worldPos));
```

- To avoid duplicate fog generated by the GrabPass, open the **Water.shader** and change the Tag setting from this:
Tags { "Queue" = "AlphaTest" "RenderType" = "Opaque" "ForceNoShadowCasting" = "true" }
To this one:
Tags {"Queue"="Transparent-2" "RenderType"="Transparent" "ForceNoShadowCasting" = "True"}

- For the same reason, open the **Brim.shader** and change the Tag setting from this:
Tags { "Queue" = "AlphaTest+1" "RenderType" = "Opaque" "ForceNoShadowCasting" = "true" }
To this one:
Tags {"Queue"="Transparent-1" "RenderType"="Transparent" "ForceNoShadowCasting" = "True"}

- Set the Azure[Sky] directional light to the "Sun Light" field in the "Surface FX" section of the 'HidroformComponent'.

- Remember to attach the fog scattering effect to the camera, before the under water script.

- In the 'Reflect FX' section of the 'HidroformComponent', check the option 'Skybox Capture'. This will update the water cubemap using the skybox as reference.

- For an another approach to update Hidroform cubemap, see the post number #628 from the Hidroform official forum.

Single-pass Instanced Rendering:

The single-pass instanced rendering feature is not advertised in the Asset Store description because I don't have a VR device for testing. So I have edited the shaders following the instructions from Unity's documentation website: https://docs.unity3d.com/Manual/SinglePassInstancing.html

In this website there are some good information too:
https://medium.com/@jollytheorysteam/how-i-modified-my-unity-post-processing-effect-to-work-with-single-pass-instanced-vr-dc41863c34f5

To use the edited shaders with the support to the single-pass instanced rendering, just import the Unity's package called '***Single_Pass_Instanced***' located at the '***Replacements***' folder:
Assets/Plugins/Azure[Sky] Dynamic Skybox/Replacements/Single_Pass_Instanced.unitypackage

I think all the shaders were modified correctly, but my only preoccupation is regarding the 'Fog Scattering Effect'. I know Unity uses a texture array sampler in shader when using single-pass instanced rendering to render to correct texture slice/layer in the screen.

But in the fog scattering effect there is the option to select its resolution, so you can save performance by setting a low texture resolution to it. To achieve this feature, first we render the fog data into a separate RenderTexture with a low resolution, and then we render this low resolution RenderTexture to the screen stretching it to the screen resolution.

This low resolution RenderTexture uses a default render texture construction, maybe it need to be changed treated as a Tex2DArray.

```
// Default constructor
public void SetFogScatteringResolution(int width, int height)
{
    m_fogRenderTextureWidth = width;
    m_fogRenderTextureHeight = height;

    m_fogScatteringRT = new RenderTexture(width, height, 0, RenderTextureFormat.ARGB32, RenderTextureReadWrite.Default)
    {
        name = "Fog Scattering RT",
        wrapMode = TextureWrapMode.Clamp,
        filterMode = FilterMode.Bilinear
    };

    m_fogScatteringRT.Create();
}
```

I'm not sure, but maybe this render texture should be created as an array texture. So try to change its constructor in the 'AzureFogScatteringRenderer.cs' to some like this:

```
// New constructor
public void SetFogScatteringResolution(int width, int height)
{
    m_fogRenderTextureWidth = width;
    m_fogRenderTextureHeight = height;

    m_fogScatteringRT = new RenderTexture(width, height, 0, RenderTextureFormat.ARGB32, RenderTextureReadWrite.Default)
    {
        name = "Fog Scattering RT",
        wrapMode = TextureWrapMode.Clamp,
        filterMode = FilterMode.Bilinear
    };

    m_fogScatteringRT.dimension = TextureDimension.Tex2DArray;
    m_fogScatteringRT.volumeDepth = 2;

    m_fogScatteringRT.Create();
}
```

If you are in a URP project, the script you should try this change is 'AzureFogScatteringFeature.cs', instead of the 'AzureFogScatteringRenderer.cs' because this one only works in the built-in render pipeline. I'm sorry not being able to implementing it completely, but unfortunately I don't have a VR device for testing and Unity documentation is lacking information.

Known Issues:

Weather Property and Undo System:

For some unknown reason, after you drag the sky prefab into the scene and starts creating new '***Weather Groups***' and '***Weather Properties***' to edit later in the weather presets, for some reason these new custom weather properties will not work with the undo system. If you make a change and apply the undo 'Ctrl +Z', it will broke all the new weather groups and weather properties created after the prefab creation.

How to fix it: Just after you drag the prefab into the scene, right click over it and go to the option '*Prefab>Unpack Completely*'. Now you can create a lot of additional weather groups and weather properties and safe edit it and use the undo system.

Volumetric Lights Effect Disappearing:

Sometimes, the volumetric render disappears after Unity's editor ends recompiling the scripts, to see the volumetric lights render again, you just need to enter play mode, so that the render command buffer is created again. If I'm not wrong, it is caused because the CommandBuffer seems to be not serializable.

IL2CPP and Code Stripping:

When using IL2CPP as 'Scripting Backend', the engine code stripping can't be disabled in the 'Player' settings, or even if you are not using IL2CPP, but you have opted to enable the engine code stripping, here goes some useful information.

What is code stripping? This option that you can set in the 'Player' tab of the 'Project Settings' window, forces the compiler to remove unused engine API from the build, this reduces the final build size. For example, if you not set any Light intensity by script, the 'Set' accessor of the Light's intensity property will be removed from the final build because it were never used in your game.

But, when the 'Override Mode' of the custom 'Weather Properties' is enabled, Azure[Sky] uses System.Reflection to set the target variables based only by its string names (*this is why it is possible to configure the custom weather properties to automatically overwrite the target variables without using code*).

This feature (*Override System*) requires the use of 'PropertyInfo' and 'FieldInfo' to store the information of the target variables, and the use of its 'GetValue' and 'SetValue' to overwrite it. The problem is that the compiler does not consider it as valid 'Getter' and 'Setter' accessors, so there is the possibility that some target variables handled by the override system does not have a 'Set' accessor in the final build, and when the 'SetValue' of the 'PropertyInfo' and 'FieldInfo' tries to overwrite it with the output values from the custom weather properties, it can crash the game or fail the project building because of a missing set accessor (*the 'SetValue' will tries to use it later in the build, but it does not exist*).

How to fix it? Make sure the 'Set' accessor of the target variables from your custom properties have been used at last one time in the game logic. Or you can just use the Azure[Sky] prefab called 'AzureCoreSystem_OverrideByCode', in this prefab, the override system of all the custom weather properties are disabled by default, and the output values from the weather properties are sent to their target variables using a custom script called 'AzureWeatherPropertyLinker' using the following weather system's API:

```
public float GetFloatOutput(int groupIndex, int propertyIndex)
public Color GetColorOutput(int groupIndex, int propertyIndex)
public Vector3 GetVector3Output(int groupIndex, int propertyIndex)
```

Memory Garbage:

You may notice that the Azure[Sky] prefab can be generating memory garbage, it is caused by the use of ‘System.Reflection’, more precisely, the “SetValue’ of the ‘PropertyInfo’ and ‘FieldInfo’.

How to fix it? There is no way to fix it if you want to use the ‘Override System’ to automatically overwrite the target variables of the custom weather properties without using code.

To get rigid of the memory garbage, you need to disable the “Override Mode’ for each weather property, and manually overwrite its target properties using a custom script. Actually, it is very easy to do, you can just use the following API from the AzureWeatherSystem component to get the output values from the custom weather properties and send it to the target variables.

```
public float GetFloatOutput(int groupIndex, int propertyIndex)
public Color GetColorOutput(int groupIndex, int propertyIndex)
public Vector3 GetVector3Output(int groupIndex, int propertyIndex)
```

By default, Azure[Sky] provides a version of the prefab with the ‘Override Mode’ of all the custom weather properties used by it already disabled. There is a script attached to it called ‘AzureWeatherPropertyLinker’ that get the output values and send it to the targets.

The prefab variation is called ‘AzureCoreSystem_OverrideByCode’, by using this prefab you should be free of the garbage generation, and also the performance should be better than the default prefab.

Just open the script ‘AzureWeatherPropertyLinker’ to see how simple is to manually use the custom weather properties output to control variables from other scripts and component by using your own code.

You can just follow the same process if you want to create more custom weather properties.

The downside of disabling the automatic ‘Override System’ and set the target variables by your own code, is that it can be a pain if you change the order of the weather groups and its weather properties in the AzureCoreSystem list or even if you delete some weather group or weather property. It will require you to fix the indexation in the calls of the `GetFloatOutput()`, `GetColorOutput()`, and `GetVector3Output()` from the ‘AzureWeatherPropertyLinker’ script because the order of the weather group and weather properties were changed.

I would disable the ‘Override System’ to get rigid of the garbage generation only when the game development is near the end, when the project is in its polishing and refinement stage and all the weather properties and weather groups are already done and there is no more need to change it. Otherwise you could waste a lot of time fixing the indexation every time you change something.

Importing error:

The type or namespace name 'VisualScripting' does not exist in the namespace 'Unity' (are you missing an assembly reference?)

This error can appears in the console after you import Azure[Sky] to your project if you do not have the Unity’s ‘Visual Scripting’ package installed in the project. All Unity projects have it installed by default, but there is the possibility you have removed it from the project.

Azure[Sky] is using an important function from the ‘VisualScripting’ namespace. It is a custom ‘SetValue’ for the ‘PropertyInfo’ and ‘FieldInfo’ used by Azure[Sky] to perform the ‘Override System’. This function is called ‘SetValueOptimized’, it is an optimized variation used the Unity’s visual scripting feature that should generate less memory garbage as described in the in the issue above, now you know Unity is also using ‘System.Reflection’ in their scripting node system.

How to fix it? You can just import the ‘Visual Scripting’ using the ‘Package Manager’ window, it is located in the ‘Unity Registry’ section. Or you can just open the Azure[Sky] script called ‘AzureWeatherSystem’ and replace all the ‘SetValueOptimized’ calls by its original variation ‘SetValue’.