GIMME Cloud Shadows

Manual

Contents

1 Workflows

1.1 Basics

There are in essence four ways to get decent semi-transparent cloud shadows (I can think of). Those are: Using a **fullscreen image effect / fullscreen pass**, using a **shadowcaster with some form of dithering**, simulating the effect with a global texture in each shader in a **material-based way** and last but not least with **ray-tracing volumetric clouds**.

As ray-tracing would work by default without additional code *if* you and your customers have the hardware, this package focuses and provides solutions for the other three options. And all of those have the same thing in common: They need one or more **materials**.

The way this package is structured is therefore, that there is a global material controller (**GCSController)** that provides the shaders with the correct information for them to work. All the workflows are then just building upon this class.

Each option has some advantages and disadvantages regarding the effort required and the visual quality.

In addition to that, there is also a procedural texture generator included to create the shadow textures (but it can be used for much more).

1.1.1 Feature Comparison

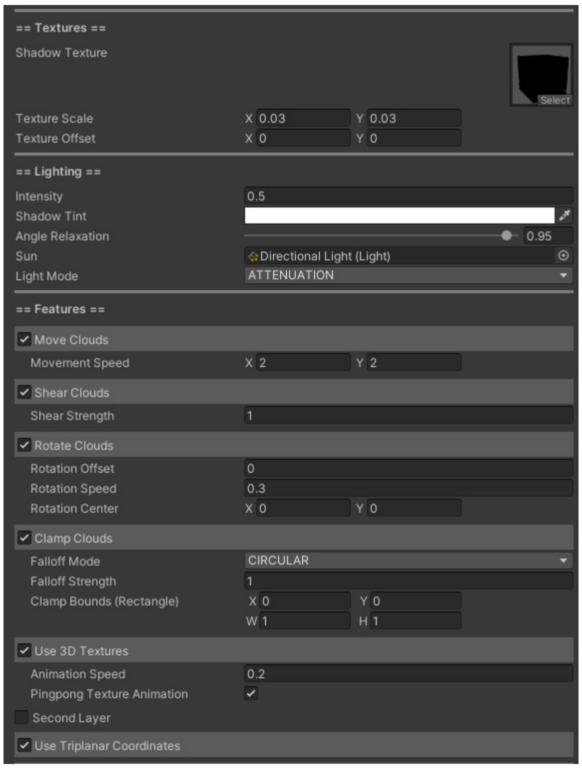
Comparison of the different Workflows		
Workflow	Pros	Cons
Image-Effect	Cheap Easy Setup Semi-Transparency	Overshadowing No control which material is affected
Shadowcaster	Cheap (Relatively) Easy Setup	No Real Semi-Transparency Results vary with shadow setup
Material	Best Visual Quality Very flexible Semi-Transparency	Requires good shader knowledge Requires a lot of work Performance

1.1.2 Material Controller

The controller is the heart piece of this package. You should place / instantiate it into the scene each time you want the cloud shadow effect regardless of which workflow you're using. The component talks to all the materials and shaders by using global properties. The code also has some methods that you can easily call to change the behaviour and the settings at runtime.

You can find a prefab with some reasonable default-values under **Runtime → Prefabs → GCSController**.

The controller has a lot of options regarding on how the shadows are rendered, and here are their explanations:



```
== Textures ==
Shadow Texture                                              Select
Texture Scale                  X 0.03        Y 0.03
Texture Offset                 X 0           Y 0
== Lighting ==
Intensity                      0.5
Shadow Tint
Angle Relaxation               ----------------------●-- 0.95
Sun                            Directional Light (Light)
Light Mode                     ATTENUATION
== Features ==
✓ Move Clouds
  Movement Speed               X 2           Y 2
✓ Shear Clouds
  Shear Strength               1
✓ Rotate Clouds
  Rotation Offset              0
  Rotation Speed               0.3
  Rotation Center              X 0           Y 0
✓ Clamp Clouds
  Falloff Mode                 CIRCULAR
  Falloff Strength             1
  Clamp Bounds (Rectangle)     X 0           Y 0
                               W 1           H 1
✓ Use 3D Textures
  Animation Speed              0.2
  Pingpong Texture Animation   ✓
☐ Second Layer
✓ Use Triplanar Coordinates
```

Figure 1: Material Controller

- **Shadow Texture**: The global texture that is sent to all the materials. When using the controller for shadows (see Light Mode), an alpha-value of 1 and a color of black means full shadow-strength
- **Texture Scale and Offset**: Applies scale and offset in world space (not dependent on the UVs of the objects)
- **Intensity**: 0 = No Shadows; 1 = Full Shadows; more than 1 = Darker than the normal Shadows
- **Shadow Tint**: Tints the color of the texture, which in turns can change the shadow color. I am wording this carefully as a tint can not be applied to a black texture (which is what you will usually have. So if it does not work → Texture is probably black to begin with)
- **Angle Relaxation**: This is best explained with an example. Imagine having the texture projected straight down from above upon the world.

Now imagine if we have a building there too with straight walls. If we did just that, then we would only sample one texture row of the shadows, which would result in dark stripes going down the walls. It would look like some visual artifact to the player. To remove that effect and sample from more than one row, we essentially skew and relax the normals of the walls a little bit so that the angle is not 90° to the light direction anymore internally. Which is exactly what this option does!

- **Sun**: Because of optimization reasons (calculating matrices on the CPU rather than the GPU), the main light source should be provided to the controller. Without it, shearing the shadows will not work (but otherwise nothing is affected)
- **Light Mode**: There are two modes available by default (easily extendable). They are attenuation and additive
- **Move Clouds**: It would be kind of boring if they couldn't move, wouldn't it. You can also simply call internal code to move the clouds in your own way
- **Shear Clouds**: Real clouds are not 2D texture projections, but have a 3D structure. This means that realistic shadows also stretch with the light direction (as any other non-flat 3D object). We can simulate this effect by shearing the texture with a strength representing the average cloud height
- **Rotate Clouds**: Rotating in this sense means that the texture itself is rotated. The further away you go from the center (also an option) the faster the rotation. A slow rotation and/or an initial offset can increase the visual variety
- **Clamp Clouds**: You can limit the shadows to a rectangle area in world space. Because those bounds can also be moved at runtime, this is useful for creating cloud shadow fronts. To avoid hard borders, the strength of the shadows falls off at the border using either a circular falloff model or a rectangular one (calculating the distance field of a rectangle). Another potential use is for tornadoes. Just saying.
- **3D Textures**: If you choose to use 3D Textures, the shadows are animated over time. Because it is already a struggle to make 2D Textures seamless, yet alone 3D ones, there is an option to let the timer value go back-and-forth (Pingpong), so as to never go "beyond" the 3D texture (i. e. when you have UVW coordinates, the W-coordinate is never going below 0 or above 1)

- **Second Layer**: Adds a second texture with a second set of options. This also increases variety (at the cost of memory and performance)
- **Triaplanar Coordinates**: This feature is more of a gimmick and only useful when using an image-effect shader or using additive light mode and the movement direction is important. It is a little bit technical: Essentially the clouds move in the wrong direction in areas that are already in the shadow. Usually that is no problem because you can't see those.

Without the GCS Controller, no workflow/option will work! Always have it somewhere in the scene.

1.2 Image-Effect Workflow

The image-effect works in a similar fashion to a regular post-processing effect. It adjusts the color to imitate shadows. Internally the world-space coordinates and world-space normals of each pixel are reconstructed and used in the regular cloud shadow shader logic. But because there is no way of knowing which pixel is already in a shaded area, you encounter a problem I labelled "overshadowing" in the feature comparison. In essence you're darkening already shaded areas again which can look a little bit weird.

A requirement for reconstructing the information is that your graphics API supports depth textures. Most of them do, but you can check here: Cameras and depth textures

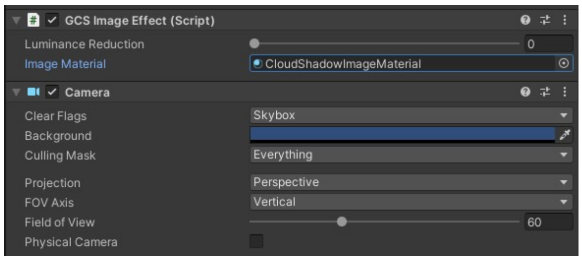
The shader used for all of this is called **GimmeCloudShadows→Cloud Shadows Image** and there are variants for each supported rendering pipeline. The setup is done in two steps.

The first one is to create a material from this shader. No values need to be provided, the material alone somewhere is enough.

The next step is then different between the rendering pipelines.

1.2.1 Builtin

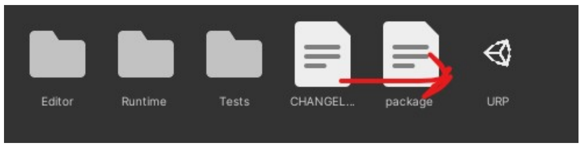
For the Built-in Rendering Pipeline you add the component **"GCS Image Effect"** to the main camera and drag-and-drop the material you created into the **"Image Material"** slot. And... that's it. Done.



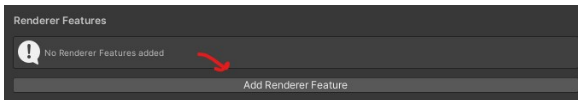
GCS Image Effect (Script)
Luminance Reduction 0
Image Material CloudShadowImageMaterial
Camera
Clear Flags Skybox
Background
Culling Mask Everything
Projection Perspective
FOV Axis Vertical
Field of View 60
Physical Camera

1.2.2 URP

For the Universal Rendering Pipeline, you first locate the URP.unitypackage and extract the content.

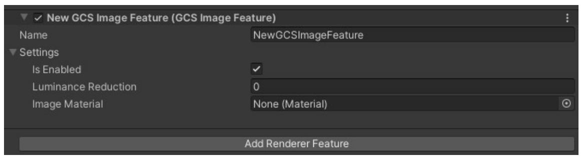


Afterwards you will have to locate your Renderer Data. It is usually under **Assets/Settings**. In this case, using forward rendering, the scriptable-object-file is called **"ForwardRenderer.asset"**. Looking at the inspector, you should be able to add a Render Feature like here:



Renderer Features
No Renderer Features added
Add Renderer Feature

Add the **GCS Image Feature**. The game view will turn black and your inspector should look something like this:



New GCS Image Feature (GCS Image Feature)
Name NewGCSImageFeature
Settings
Is Enabled ✓
Luminance Reduction 0
Image Material None (Material)
Add Renderer Feature

Drag-and-drop the material you created earlier in the **"Image Material"** slot. When done correctly, the game view should look normal again.

1.3 Shadowcaster Workflow

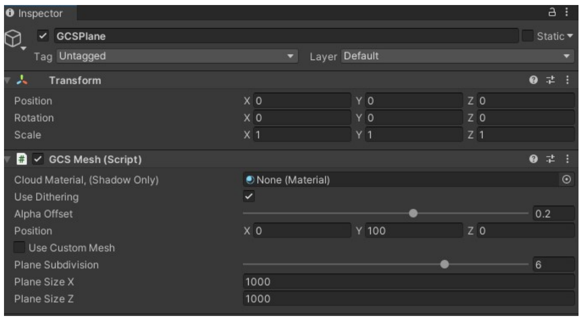
A shadowcaster is nothing more than an object that casts shadows. In this case it is an invisible mesh with some geometry that uses a special shader. You can find it as **GimmeCloudShadows→Cloud Shadows Only**.

You can't use the additive light mode of the controller for this workflow... you can only cast shadows.

Now, because it is so dependent on the Unity Shadows, the quality of this method varies a lot, depending on your setup of the shadow map resolution and shadow cascades. Also, there is no natively supported semi-transparency for shadows, which is why this is approximated using a dithering method. This is especially apparent when getting very close to a shadow or if the main light uses hard shadows.

On the other hand... this can also make your game really stand out and could be interpreted as a stylistic choice (dithering is very common in games that use a cartoon-ish style). Alternatively, you could just simply don't use dithering, having a shadow edge like all the other objects in your scene.

The setup is very simple. First, create a material using **GimmeCloudShadows→Cloud Shadows Only**. Then choose any GameObject and add the **GCSMesh**-component to it, like so:



Inspector

GCSPlane ✓ — Static ☐
Tag: Untagged — Layer: Default

Transform

	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	1	1	1

GCS Mesh (Script) ✓

Cloud Material, (Shadow Only)	None (Material)
Use Dithering	✓
Alpha Offset	0.2
Position	X 0, Y 100, Z 0
Use Custom Mesh	☐
Plane Subdivision	6
Plane Size X	1000
Plane Size Z	1000

Then add the material you created first to the **Cloud Material** slot. And that, in theory, is it (works for each rendering pipeline). However, depending on your world-, light- and shadow-setup you might not see the shadows immediately, so maybe you have to move around the plane and the main directional light a little bit.

The component itself by default creates a subdivided plane at the start, applies the material and moves it to the position (provided in the inspector). There are a few settings to adjust, so here is the explanation:

- **Dithering**: Uses dithering to approximate semi-transparent shadows. Works better with soft shadows
- **Alpha Offset**: Subtracts the value from the actual alpha value in the texture
- **Position**: Moves the created plane to these coordinates
- **Plane Subdivision**: The plane has to be subdivided into smaller parts because of an issue Unity affectionately calls **Shadow Pancaking**. You can find more information here: Shadow troubleshooting

- **Plane Size**: The world-space size of the plane
- **Custom Mesh**: When using a custom mesh, no plane is created and instead the geometry of your provided mesh renderer is used (a good choice would be a hemisphere). Make sure that the mesh is also not affected by **Shadow Pancaking**.

1.4 Material Workflow (Advanced)

This workflow yields the best visual results, but the trade-off is some work and performance. In principle, you include the shader-code from the file **Runtime/Shader/GimmeCloudShadows.cginc** into the shaders you're using and call the method **CloudShadow** in the fragment-part. Then you use the color it returns and (usually) compare it against the shadow attenuation.

When you're using the method, the controller is still going to work for this shader (because it is global) and therefore all your materials will work as well.

There are some examples, that show how one could do that, included in the package. The one worth mentioning is the **GimmeCloudShadows/-Cloud Shadows Default**-shader that works on all rendering pipelines that are supported. You can use this one as a baseline for creating more advanced shaders. Another useful one might be the surface shader **GimmeCloud-Shadows/Cloud Shadows Surface Lit (Only Builtin)**.

You can download all the built-in shaders at the Unity Download Page. For the shaders of the rendering pipelines: You can find them inside your packages folder

1.4.1 Talking Code

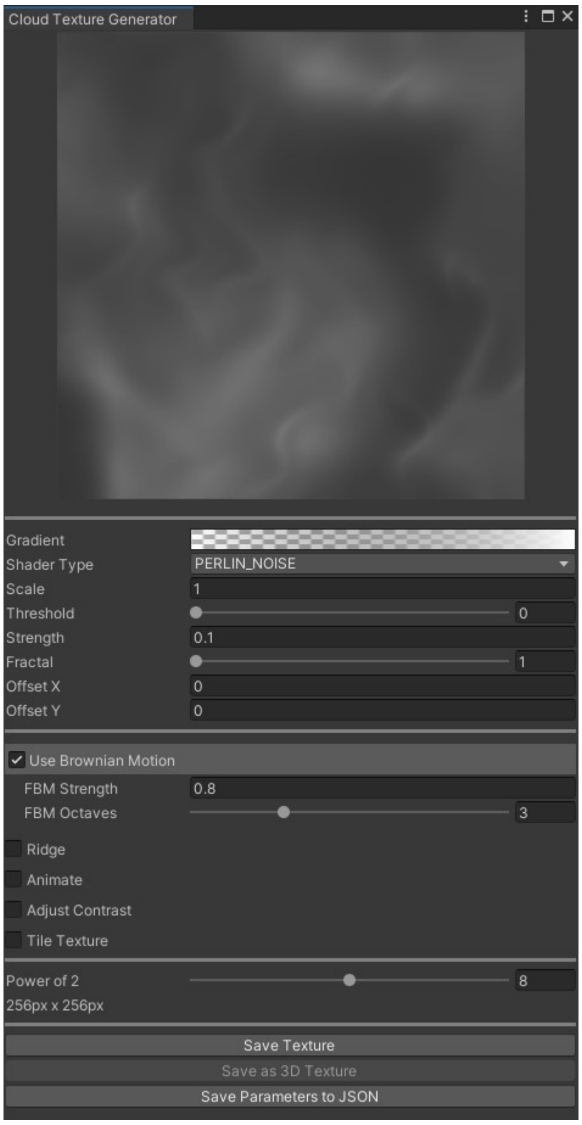
The process can not really be automated (because it depends on what you're using). But here is a general outline on what has to be done:

```
Shader "MyShaders/Custom Shader (with Cloud Shadows)"
{
  Properties
  {
    ...
  }
  SubShader
  {
    Tags { ... }

    Pass
```

```
{
  Name "MyPass"

  CGPROGRAM

  #pragma target 3.0

  #include "UnityCG.cging"
  #include "Lighting.cginc"

  //Here you'll have to enter the correct path
  //to GimmeCloudShadows.cginc
  //Alternatively you can also just copy and paste the
  //code from the file
  #include "/MyShaderLibrary/GimmeCloudShadows.cginc

  #pragma vertex vert
  #pragma fragment frag

  ...

  //We need the position and normals...
  struct MyVertexData
  {
    ...
    float4 pos : POSITION;
    float3 normal : NORMAL;
    ...
  }

  //... for later calling CloudShadow()
  //in the fragment part
  struct MyFragmentData
  {
    ...
    float3 worldPos : TEXCOORD1;
    float3 worldNormal : TEXCOORD2;
    ...
  }

  ...

  MyFragmentData vert(MyVertexData v)
  {
    MyFragmentData o;

    ...
  o.worldPos = mul(unity_ObjectToWorld, v.pos);

  half3 worldNormal = UnityObjectToWorldNormal(v.normal);
  o.worldNormal = normalize(worldNormal);

  ...

  return o;
}

...

float4 frag(MyFragmentData i)
{
  //We have to get the already existing shadow
  //first somehow. This is done differently
  //depending on the rendering pipeline
  float atten = GetShadowAttenuation(i);

  ...

  //This is the important bit. The method needs
  //just two inputs for each pixel:
  //  - The world position
  //  - The world normal (at that position)
  //As long as you can provide that information
  //the correct color is returned
  float4 cloudColor = CloudShadow(i.worldPos, i.worldNormal);

  ...

  //Now we have to apply the cloud color
  //to the regular color

  //These are the light modes of the controller
  #if GCS_ATTENUATION

  //This is just an example implementation
  //but you can use the cloud color however
  //you like at this point

  //When alpha = 1, we want the attenuated
  //shadow color, nothing else
  cloudColor.rgb
  = lerp(cloudColor.rgb, atten, 1 - cloudColor.a);

  //We also want to adjust depending on the
  //intensity setting in the controller
  cloudColor = lerp(cloudColor, atten, 1 - _GCSIntensity);
```

```
        //Get all the other lighting influences
        float4 lighting
        = diffuse * min(atten, cloudColor) + ambient;

        //And then finally adjust the color
        color = color * lighting;

        #elif GCS_ADDITIVE

        ...

        #endif

        return color;
      }


      ENDCG
    }
  }
}
```

2 Noise Texture Generator



Gradient
Shader Type PERLIN_NOISE
Scale 1
Threshold 0
Strength 0.1
Fractal 1
Offset X 0
Offset Y 0
✓ Use Brownian Motion
FBM Strength 0.8
FBM Octaves 3
Ridge
Animate
Adjust Contrast
Tile Texture
Power of 2 8
256px x 256px

Figure 2: Texture Generator

You can find the Noise Texture Generator under **Window → Gimme Cloud Shadows → Generate Cloud Textures**. Although the intention was to use it to create the shadow textures, it turns out that this little tool is surprisingly a lot more useful than envisioned. Based on noise-generating compute shaders it is possible to create all sorts of procedural patterns, still or animated.

Upon opening the tool, you'll be asked if you want to create a new texture or load one from a JSON File. You can find some example textures in the Folder **Editor → NoiseParameters**.

Once decided, there are a lot of parameters to play around with (Note: There are tooltips on some of them). I encourage to play around with everything to get a feel for the tool, but here a rough explanation of the values:

- **Gradient**: Maps the defined colors (max. 8) to the noise values (which range from 0 to 1).
- **Noise Type**: The type of the base noise map
- **Scale**: The scale of all the noise maps (including FBM)

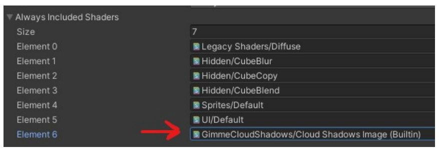
- **Threshold**: You can define a threshold to set all the noise values below this number to the color at the left in the gradient. E.g. when the left-most point of the gradient is black, and the threshold is 0.5, all noise values below 0.5 are mapped to black
- **Strength**: The base noise map is multiplied by this value
- **Fractal**: Adds octaves to the base noise map. In essence adding the same map at double the scale with half the strength.
- **Offset**: An offset to the start of all noise maps
- **Brownian Motion**: Fractal brownian motion noise is a special type of noise that usually goes well with others. It also goes well for doing domain warping, and therefore has to be activated when you want to animate the texture
- **FBM Strenghth**: The FBM map is multiplied by this value (and then added to the base noise map)
- **FBM Octaves**: Adds octaves to the FBM map. Kind of the same as the fractal setting (kind of)
- **Ridge**: Inverts the values and scales them to a power. When you use a simplex base noise alone and try out this setting a little bit, you'll see the "Ridges" very easily
- **Ridge Offset**: This is the value from which the noise is subtracted. E.g. a value of one means, the result is (1 - color) etc.
- **Ridge Power**: The values are taken to the power of... the Ridge Power
- **Animate**: Enables the moving and warping of the noise along time. The result can then be saved to a 3D Texture
- **Movement**: Moves the texture along the direction over time
- **Warp Strength**: Applies domain-warping to the noise. This option is only available when using FBM. The precise algorithm was developed based on gut-feeling, and I encourage you to play around with the code (it is great fun)
- **Warps**: You can warp the input noise multiple times, creating a more fluid-ish animation
- **Time**: A value going from 0 to 1. When saving a 3D Texture, every layer is sampled with a different time value (as you would expect)

- **Contrast**: Basically does the same as an image editor
- **Tile Texture**: Tries to tile the texture to make it seamless. However, the algorithm is not perfect (it works better with images that have not too much color-variety)
- **Blur Steps**: Blurs the texture to reduce the visibility of the seams

The rest of the settings regard the saving of the results. You can either save it into a 2D Texture, an 3D Texture (when animating) or into a JSON-File (which just contains all the parameters of this tool).

3 Shader Stripping

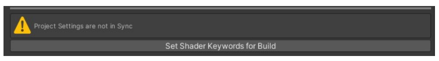
Because **Gimme Cloud Shadows** uses global properties and textures (not on a per-material basis), we have to tell Unity which variants it needs to compile in a build. Otherwise it would automatically ignore ***all*** of the cloud shadow shader code. On the other hand, if we would tell Unity to compile the complete shader, this would take hours to days (depending on the shader). Therefore, a class is provided that tells Unity which variants it can safely ignore and therefore leaves us with all the ones we'd like to have. It is called **GCSShaderProcessor**.



Always Included Shaders	
Size	7
Element 0	Legacy Shaders/Diffuse
Element 1	Hidden/CubeBlur
Element 2	Hidden/CubeCopy
Element 3	Hidden/CubeBlend
Element 4	Sprites/Default
Element 5	UI/Default
Element 6	GimmeCloudShadows/Cloud Shadows Image (Builtin)

Figure 3: Image Effect Shader is included

But in order for it to work properly, two things have to be done. First, we have to include the shaders under **Project Settings→Graphics**. What you have to include depends on your workflow. If you use an image effect, include the **Cloud Shadows Image Shader**, if you use a shadow-caster, include the **Cloud Shadows Only Shader**. And if you're using the material-workflow... well, you will have to include all your shaders there (and do additional shader stripping on your own).



Project Settings are not in Sync

Figure 4: Shader Keywords not in sync

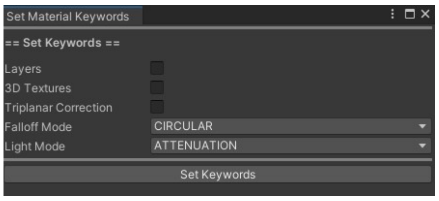
The second part is the easy one. Find the controller and press the button that says **Set Shader Keywords for Build**. This automatically searches for the keywords that can be ignored and adds them to a list, which you can find in **Project Settings→Gimme Cloud Shadows**. You can manually change that list if you want (and dare to).

4 Additional Tools

4.1 Shader Tools (Advanced)

There are two tools, that provide you with an automated way to modify the shaders and materials without coding. The first is the **Material Keyword Tool** and the second one is the **Shader Property Exposing Tool**. Both of them are used when you want to have multiple materials with different properties. In other words if you want to have materials or shaders with local keywords or local properties that are not controlled by the global **GCS Controller**.

The **Material Keyword Tool** allows you to set shader/material keywords for a given material. It has the same functionality as writing the keywords (which you can find in the static class **GCSShaderKeywords**) into the material using the debug inspector. It is important to know, that as soon as the material differs from the settings in the **GCS Controller**, you will have to adjust the shader stripping in order for it to work in builds.



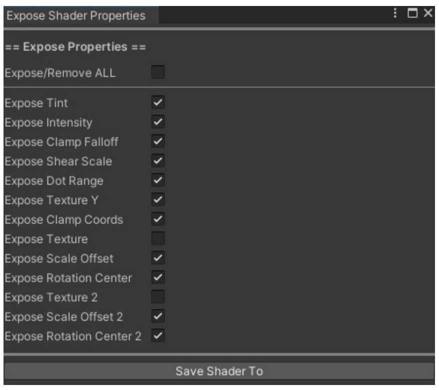
Set Material Keywords

== Set Keywords ==

Layers	☐
3D Textures	☐
Triplanar Correction	☐
Falloff Mode	CIRCULAR
Light Mode	ATTENUATION

Set Keywords

Figure 5: Material Keyword Tool



Expose Shader Properties

== Expose Properties ==

Expose/Remove ALL	☐
Expose Tint	✓
Expose Intensity	✓
Expose Clamp Falloff	✓
Expose Shear Scale	✓
Expose Dot Range	✓
Expose Texture Y	✓
Expose Clamp Coords	✓
Expose Texture	☐
Expose Scale Offset	✓
Expose Rotation Center	✓
Expose Texture 2	☐
Expose Scale Offset 2	✓
Expose Rotation Center 2	✓

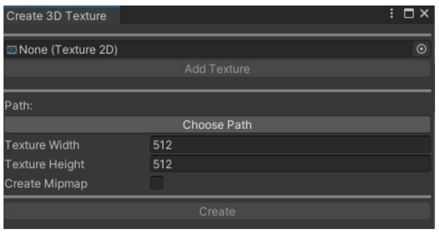
Save Shader To

Figure 6: Shader Property Exposer

The **Shader Property Exposing Tool** is a handy tool allowing you to override the global properties with local ones by exposing them. This can be useful, say, if you want for example different shadow textures for different shaders / materials. Contrary to overriding keywords, this does not affect the shader variants (except that you have two shaders now instead of one of course). Depending on your situation this might require some additional code written from your side, such that the **GCS Controller** remains global, or additional work by having to update the different materials separately.

4.2 3D Texture Tools (Advanced)

For the sake of completion, there are two tools regarding 3D Textures. You're likely to have some of them already, either from other packages or self-made. The two provided are the **3D Texture Creator** and the **3D Texture Combiner**.



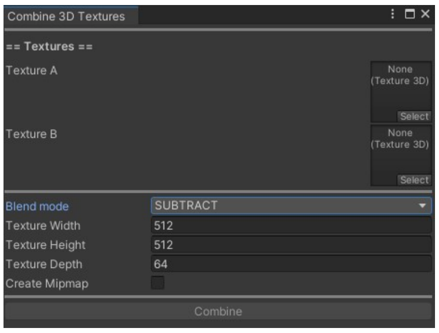
Path:

Texture Width 512
Texture Height 512
Create Mipmap

Figure 7: 3D Texture Creator

The creator is used to combine separate 2D Textures into a single stacked 3D Texture. The resolution can be freely chosen, so it is possible to combine different-sized textures (they are sampled with bilinear filtering). The reason this tool is included, is that you might want to create the 3D Texture in another program, for example by rendering the frames in an external software like **Blender**.

Speaking of blending, the combiner there to weld different 3D Textures together. This can be used to further add some details (for example by combining different cloud textures with different noise scale), but it is also a way to save memory, by combining two cloud layers into one with little sacrifice (assuming the textures are largely transparent). Again, the resolution can be chosen freely (because the values are sampled).



Texture A None (Texture 3D)
Texture B None (Texture 3D)
Blend mode SUBTRACT
Texture Width 512
Texture Height 512
Texture Depth 64
Create Mipmap

Figure 8: 3D Texture Combiner

5 Useful Information

5.1 Performance

As an fullscreen image-effect and a shadowcaster are actually quite cheap, the most performance is lost using the material-based approach. However, the performance-impact should not be that much greater than with other additional material properties (e.g. using a normal map or an occlusion map). So to improve GPU performance, the same rules should be applied as always (here a quick reminder):

- Use as few materials as possible (for example by using a texture atlas)

- Use GPU instancing or static + dynamic batching
- Reduce the quality and the amount of features necessary for materials, especially those used on small objects
- Reduce the amount of additional lights (not the main light source) when using Forward Rendering

5.2 Memory

Because global textures are used, the memory footprint in RAM and VRAM is actually quite low. However, especially when using animated textures (3D Textures), the space-requirement can become very high. You can either reduce the spatial dimensions (the texture resolution) or the animation quality (the height of the 3D Texture, i. e. less frames).
An alternative solution, creating the textures on the fly, may be implemented in the future (but this solution has a performance-cost attached to it) or can also be applied using the provided Compute Shaders.
Another point of optimization are the shader variants (because they have to be kept in the memory as well). Please make sure therefore that all the features you don't need are in the exclusion list in the **Unity Project Settings** (this is automatically done when pressing **"Set Shader Keywords for Build"** in the **GCS Controller**)

5.3 Ideas

Here are some ideas, on what else you can do with the code in the package:

- Using the Compute Shaders of the texture generator, one could create an infinite variety of seamless, low-res textures at runtime (e.g. simulating flowing water or molten lava)
- It is possible to combine the workflows to create interesting effects (e.g. you can have cloud shadows (attenuative shadowcaster) and a lightning effect (additive image-effect) at the same time)
- Because you can adjust the color additively too, you can simulate all sorts of lighting effects as well. There is a sample in the package that shows on how to emulate caustics in a bowl of water just as an example

6 Contact

For any questions or suggestions, you can reach me anytime by the following email-adress:

blenderfan@gmx.at

There is also a discord server, which is usually the fastest way to reach me:

Parable Games - Discord

Alternatively, you can also find some social media links and contact information on my website:

https://parable-games.com

7 Future Plans

Some additional features I plan to implement in the near future:

- HDRP Support
- Option in controller to generate the cloud textures at runtime using a profile
- Generator: A better algorithm for seamless textures
- Generator: New base-noises

Thank You!

Your purchase of **Gimme Cloud Shadows** enables me to continue developing code and techniques for game-development in an independent way!