Portal Room Cube via Stencils Guide
0.1 - 22.07.13

Unity 4.2 Pro introduces access to the stencil buffer, enabling various effects to be accomplished such as portals through masking areas of the screen.

This project is an exploration of using stencils to provide a portal system (that's the technique, nothing to do with the game of the same name), allowing for multiple views into the same space but showing different results depending upon which portal is in view.

Previously the only way to accomplish this would be through render to texture, though that has some drawbacks such as high texture memory usage, dependant upon screen size and number of portals. Though stencil portals don't suffer from increased texture memory they have their own set of problems such as potential inefficient rendering as polygons will still go through the pipeline until culled by the stencil mask. So ultimately the best method to use is very dependant upon your requirements.

The Stencil Buffer

Please refer to the Unity 4.2.0 documentation in the reference section for ShaderLab to learn more about stencils, and how to use them in shaders.

Stencil checks are performed before depth test in the render pipeline. This means that when a stencil test fails, it avoids almost all the per-sample costs in a fragment shader, but in order to get to this stage all the vertex processing and everything up to fragment processing still has to be performed.

Stencils in Unity

The current implementation in Unity (4.2.0) is not very streamlined and some what limited, certainly in the case for how its used in this project. The main issue is that setting up stencil parameters, operations and testing has to be done within the shader code and there is no way to adjust these parameter values from a material dynamically.

The upshot of which means you end up requiring not only a shader per portal stencil mask, but also one for each type of shader used within the portal (e.g. diffuse, specular, bump diffuse etc.). Thankfully the issue will be addressed in a forth coming update according to a post in the Unity forums shader section by Aras Pranckevičius, where it will become possible to assign variables to render settings in a shader and thus be able to update them via material/scripting (see http://forum.unity3d.com/threads/183785-Using-a-shader-for-both-image-effect-and-textures)

Another issue to be aware of is that currently there is no method to independently clear the stencil buffer or the depth buffer. Clearing the depth buffer will also clear the stencil buffer. While it should be possible to draw a full-screen quad and reset the stencil buffer, effectively clearing it, the same cannot be done for the depth buffer. It might be possible to get around this issue through use of a plug-in and direct access to the 3D api.

It should be noted that Unity makes extensive use of the stencil buffer for deferred rendering and as such it is probably impractical to use it in many cases.

The Basic Stencil Portal Concept

The basic concept of using stencils for creating portals is simple and straightforward. You first render a polygon or mesh that sets a specific value in the stencil buffer, effectively creating a mask. Subsequent meshes are then checked against this stencil mask at the pixel/fragment level, if the pixel to be drawn has the same stencil reference value it continues down the pipeline to possibly being rendered, if not it is culled.

A Single 'Portal Room Cube'

Though still simple in concept, with the current limitations of stencils in Unity the set up becomes more complicated though the additional requirements of creating and managing of multiple shaders, materials, models.

The single cube requires four stencil masks, one for each side of the cube (ignoring top and bottom). Each side requires a unique stencil mask shader, that renders a unique reference value into the stencil buffer. The stencil mask shader must be rendered prior to any other meshes and therefore must have depth buffer writes disabled, so that subsequent content can be rendered inside the cube and not culled by the depth of the stencil mask itself.

Complications arise in that the contents of each Portal room must have its own set of unique shaders. This is due to the fact that currently it is not possible to assign values to the stencil property block at run time. The upshot of which means you have to make a duplicate shader for each type of shader used within each unique portal/stencil instance.

For example we have four portals/stencil masks, that means if we want to have a 'bumped diffuse' material applied to models within each portal we will need 4 unique versions of that shader, each with a different reference index. If you want 'bumped Specular' material as well, then that’s another 4 unique shaders and so on. This can be seen if you look through the 'StencilShaders' folder of the project and the various 'Portal_<n>_Shaders', where 'n' is a number indicating which portal the shader is for.

Thankfully there isn't actually much you have to do to each unique shader, beyond changing the stencil reference value assigned within it. So its relatively straightforward to grab the Unity built-in shader code, create a duplicate and add in the necessary stencil parameters. See the ' StencilShaders' (grouped by Portal) in the example project for more information and the Stencil Shader section at the end of this document.

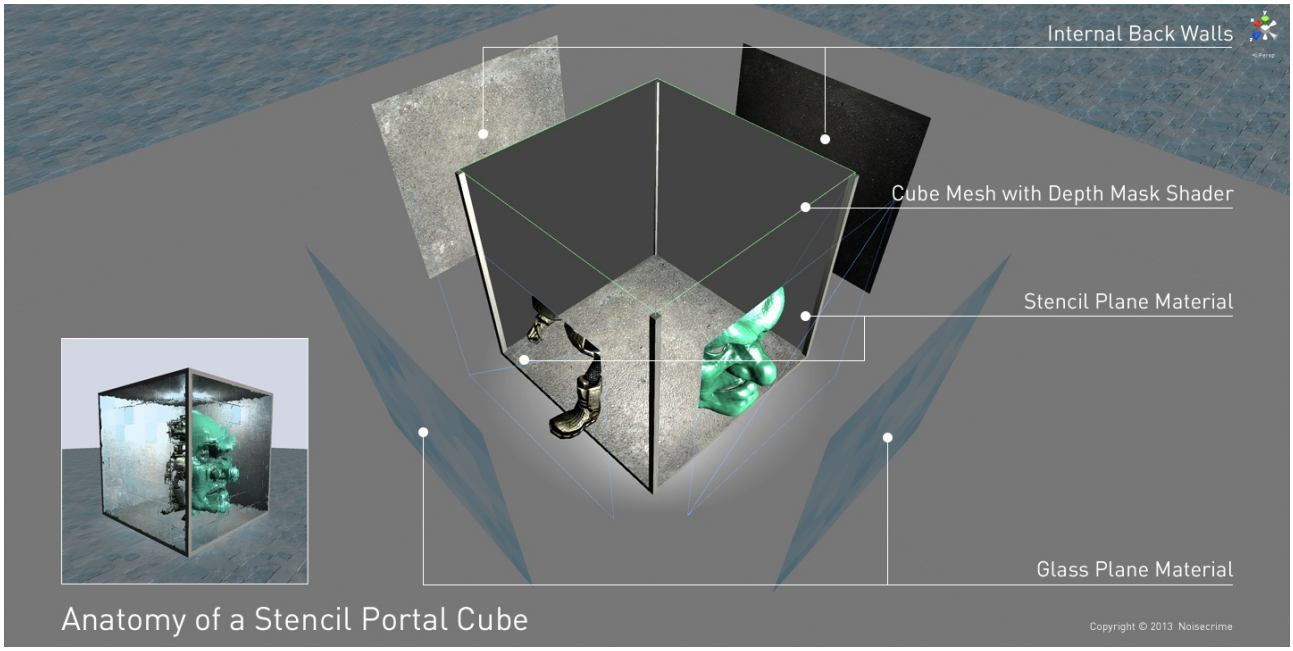
Other aspects that make this more complex are the contents within a portal, for example lighting or physics colliders as these may need to be defined on unique layers to avoid affecting other portal instances.

A Room of Portal Room Cubes

Again the current limitations of stencils in Unity make this increasingly more complex than it needs to be. The biggest issue here is that we can't clear the depth buffer after rendering of the stencil mask portal planes. This forces us to resort to using stencil Mask shaders that do not write to depth buffer and creative use of a secondary cube mesh .

Anatomy of a Portal Room Cube

Each 'Room Cube' is comprised of the same elements, as shown in the exploded view image below.



The Room its floor, ceiling, struts, internal walls and glass plane façades exist in the main world layer and are always rendered. Elements such as the internal walls and glass planes are single sided planes, so they are not rendered when facing away from the camera.

Just behind each glass plane façade is a plane with a unique stencil mask shader applied to it, see the StencilMask_X shaders in the project. There are four of these stencil mask shaders, one for each side of the cube and each having a unique stencil reference value (1 to 4 for simplicity).

These are drawn before the contents of the Room cube so as to populate the stencil buffer with their respective reference values, though use of the 'Queue' Tag in the shader and setting the value to Geometry-100.

Between the glass façade and the stencil plane is a cube mesh with a 'depth mask' shader applied and which only renders back facing polygons. This is used to fill the depth buffer to prevent incorrect drawing order of the stencil planes due to the fact that they do not write to the depth buffer. It also uses a modified 'queue' value (Geometry-120) to ensure that it is the first thing rendered every frame.

Each frame the gameObjects are rendered in a specific order, determined via the queue tag in the custom shaders.

DepthMaskCube - Queue Tag 'Geometry-120'
StencilQuads - Queue Tag 'Geometry-100'
All other Content
Glass Façades - Queue Tag 'Transparent+100'

Controlling the Render Order of the Scene

The first objects rendered to the screen each frame are the '_DepthMaskCube' gameObjects for each 'Room', accomplished through the use of the 'Queue' tag (set to 'Geometry-120') defined in the 'DepthMask' shader itself. These cubes are slightly smaller than the actual room, its faces are therefore behind the glass planes, but in front of the Stencil Mask planes.

As the DepthMask shader uses 'ColorMask 0' nothing will actually be rendered to the screen, but the depth of the pixels will be rendered into the depth buffer. Most importantly though we are not rendering the front faces of the cubes, but the back faces (via 'Cull Front' in the shader) for reasons which will become clear.

The problem with stencil masks in this project is that we have to render them with depth writes disabled (I.e. disable writing the depth of each pixel into the depth buffer). The reason for this is because rendering the contents of each room cube needs to happen after the stencil mask planes and as the content is behind the stencil mask plane if we write the planes depth to the buffer the content would be culled (as its depth is further away) and never make it to the screen.

With the Stencil Mask planes not writing to the depth buffer there is no way to control their draw order with respect to distance from the camera (without resorting to some scripting solutions to sort them) and as such the Stencil Mask planes from other cubes can randomly overlap those of the cubes in front.

> *This can be seen if you load up the 'scene.StencilsDemo_DrawOrderIssue' scene in Unity and run it. All it does is disable the rendering of the all the '_DepthMaskCube' gameObjects and enable the 'Stencil Viewer' on the main camera. Moving the camera around in the game view will clearly show the incorrect and overlapping draw order problems caused by not writing to the depth buffer.*

The 'DepthMaskCube' is the simple solution to work around this problem (see following images), though it is very much dependant on the fact that this is a special case. By which I mean it takes advantage that the portals themselves are constructed around cubes which are convex and that the portal rooms inside each cube never extend beyond the cube. Without those special cases a more generalised solution would be needed, unless it becomes possible to clear the depth buffer part way through a render or between cameras.

The reason for the DepthMaskCube rendering only the back faces is that it fulfils all the requirements. It prevents portals from other cubes showing through, but importantly it wont prevent the content from within a cube being rendered as the depth buffer is being filed with the depths of the back walls of the cube, not the front walls (with respect to the camera viewpoint).

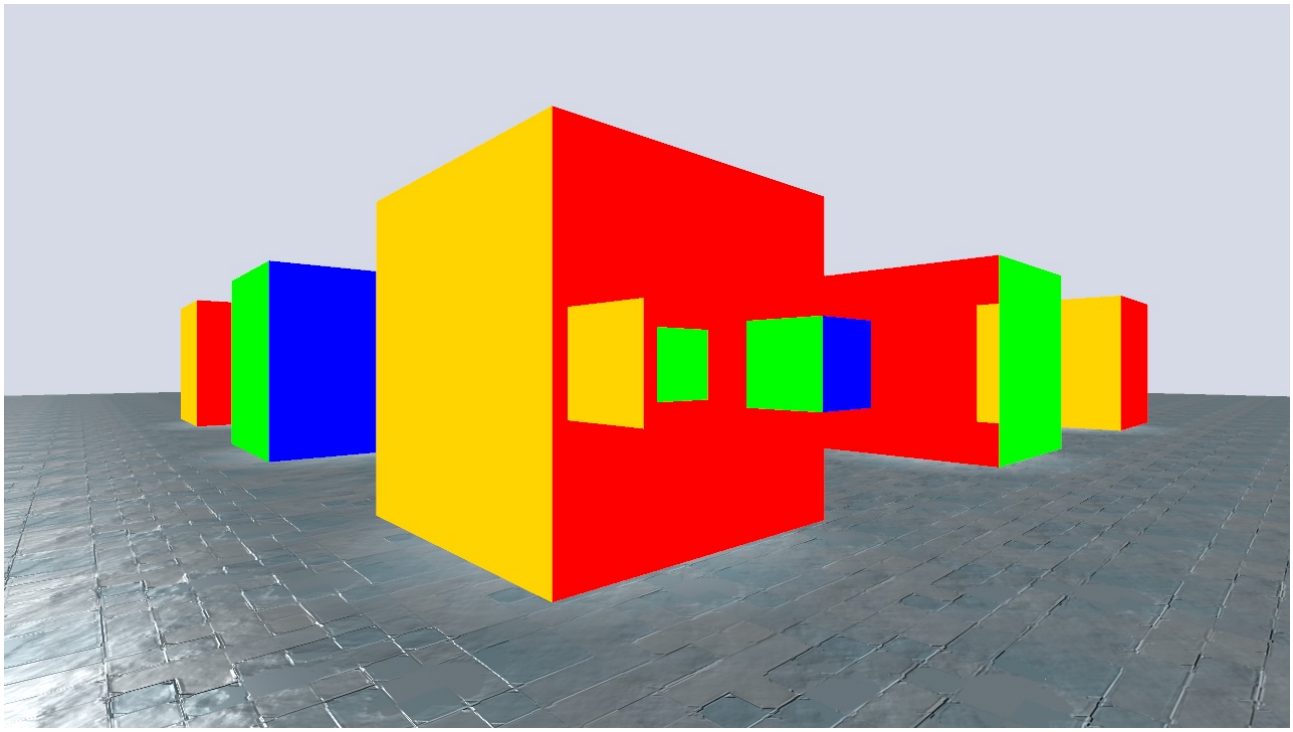


Illustration 1: Without the DepthmaskCube along with the stencil mask planes not writing the to depth buffer results in incorrect draw order, and random overlapping of stencil masks.

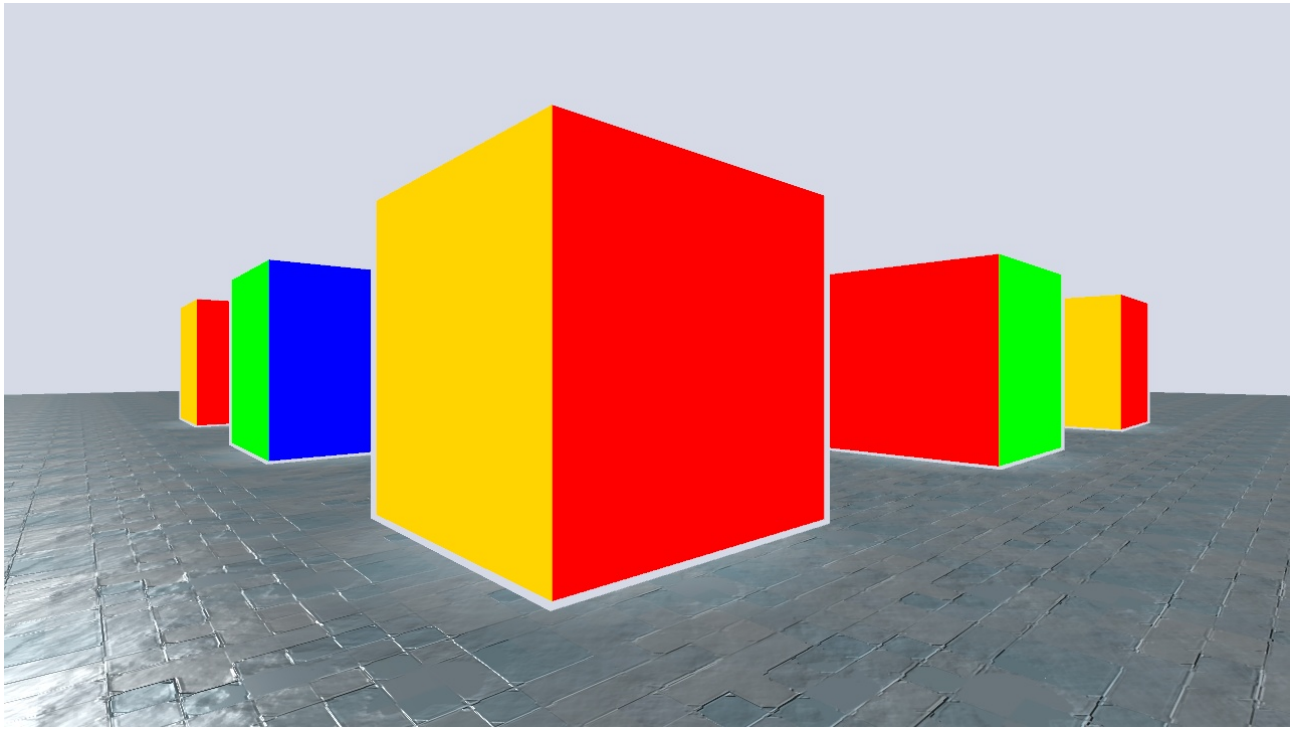


Illustration 2: With the DepthmaskCube and due to the fact that cubes are convex we get correct rendering of the stencil masks.

The Stencil Shaders - Stencil Masks

There are four stencil masks one for each side of the Room Cube, each one differs only in terms of its stencil 'ref', which is the value written to the stencil buffer for any pixels affected by this shader. This value can then be compared against in by any subsequent rendering to the same pixel.

The key element to these shaders is the stencil property block

```
Stencil
{
	Ref 1
	Comp always
	Pass replace
}
```

In the code above the shader will render a value of '1' to the stencil buffer for every pixel that is covered by the mesh being rendered. The 'Comp' keyword specifies the comparison and is similar to those used in the depth buffer. In this case it is set to 'always' meaning no comparison is made, and the stencil ref value is always written to the stencil buffer. The 'Pass' keyword determines what to do between existing stencil buffer values and the new one, here we tell it to replace any previous value with our 'ref' value, i.e. over-writing it.

More information about the specific keywords used and their meaning can be found in the Unity ShaderLab documentation.

The Stencil Shaders - General Shaders

Each portal requires its own set of unique shaders, one for each shader type. This is due to a current limitation of Unity (4.2.0) and its inability to dynamically change property states within a shader at run-time.

In the project all these shaders are simply taken directly from the Unity Built-in shader source which can be found at http://unity3d.com/unity/download/archive. The only difference is the addition of the stencil block, with the correct 'ref' value based on which portal the shader is for. e.g.

```
Stencil
{
	Ref 1
	Comp equal
	Pass keep
	Fail keep
}
```

In the code above we state that we are interested in stencil buffer values that have a reference (ref) of '1', the compare function looks for when the value in the buffer is equal to the value for this shader (i.e. '1'). If they are equal rendering continues down the pipeline possibly drawing the pixel on screen, if it fails then this pixel is culled. This is essentially what provides the masking ability of stencils.

The 'Pass' and 'Fail' keywords specifiy how to treat the stencil buffer should the

comparison fail and in both cases it keeps the value already in the stencil buffer since we do not want to affect it.

The Stencil Shaders – Stencil View Ref Shaders

These four shaders simply render a specific colour to the color buffer to illustrate where stencil portals are being rendered to on screen and are useful for debugging with.

Its a bit like setting the scene camera to render 'alpha', except here they are only visible in the 'game' view and only if the 'Stencil Viewer' script is enabled on the main camera.