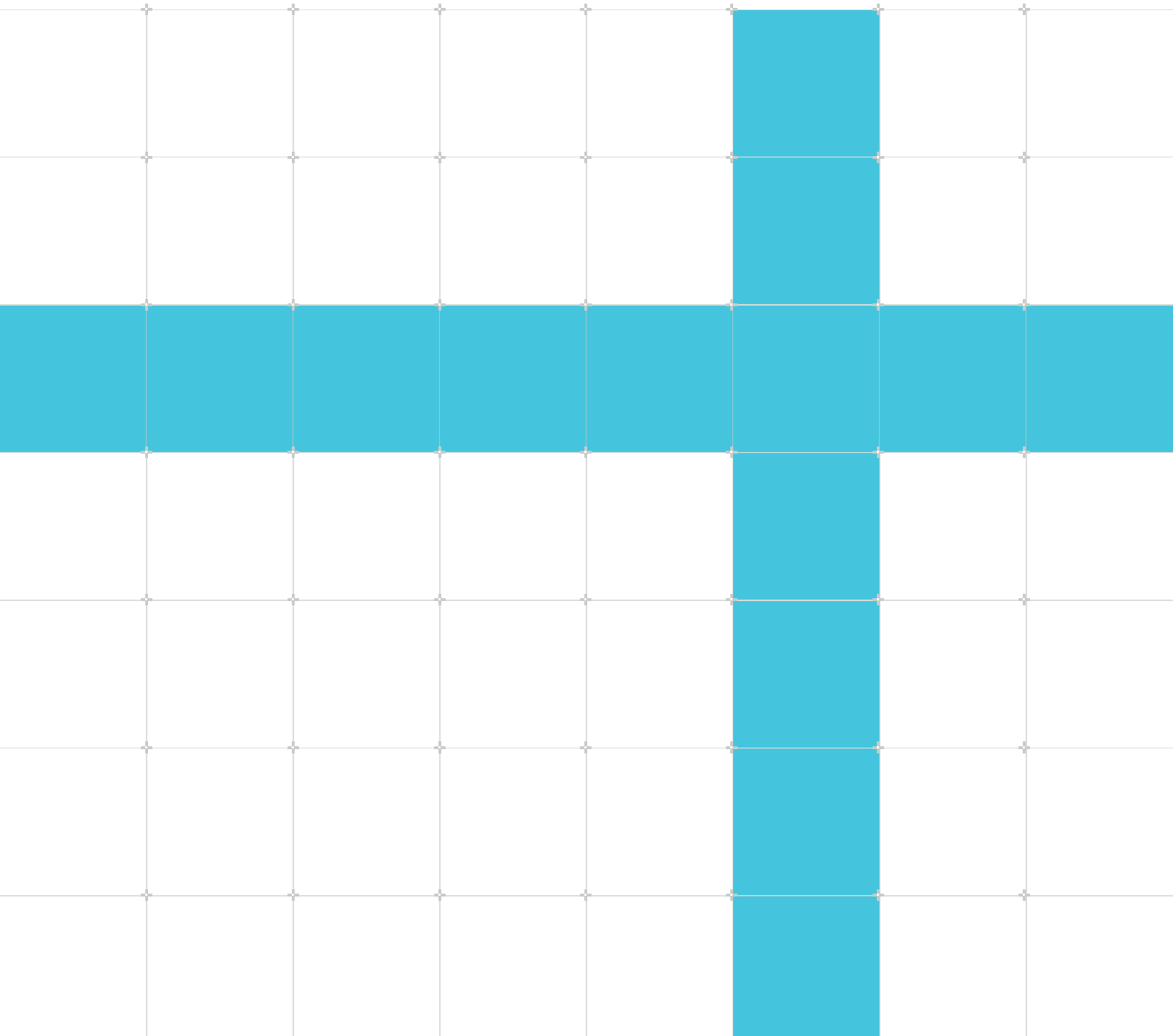
Arm Guide for Unity Developers - Special effects graphic techniques

Version 1.0

Issue 01
102259_0100_01_en



Arm Guide for Unity Developers - Special effects graphic techniques

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	1 January 2021	Non-Confidential	First release

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

This guide introduces several special effect techniques that you can use in your Unity programs, including:

- The dirty lens effect
- Fog effects
- Icy wall effect

Throughout this guide there are images showing how special effects are used in the examples shown the ice cave demo and the game Spellsouls by Nordeus.

Special effects are important in games. This is because they can make the story of the game more engaging by creating visual stimulation. However, sometimes they are sacrificed for mobile, but this guide shows efficient techniques for several effects so that games can afford these effects and increase immersion in games.

At the end of this guide, you will have learned:

- How to implement the dirty lens effect
- How to implement fog effects
- How to show the changing of time during the day using the procedural skybox

Before you begin

Before you work through this guide, you should be familiar with Unity and the fundamentals of shaders and reflections. To learn more about these topics, read our guides:

- Advanced graphic techniques - Getting started
- Local cubemap rendering

2. Specular effects in the Ice Cave demo

This section of the guide describes the specular effects that are used in the Ice Cave demo. Specular effects are an efficient technique that produce good results.

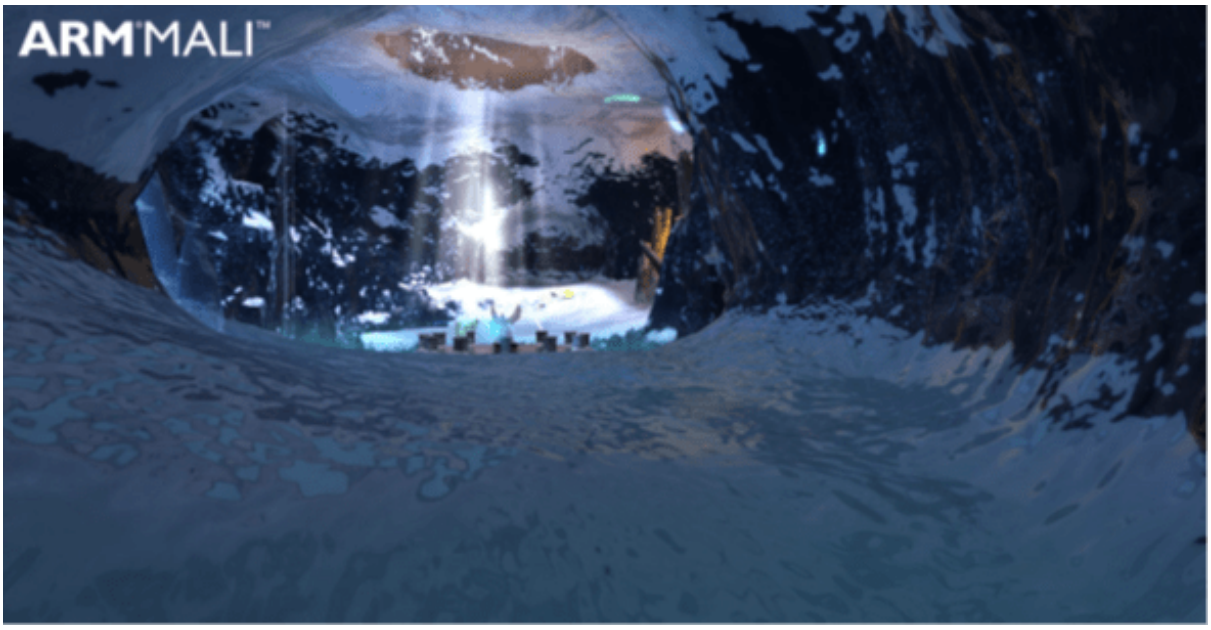
The specular effects that are used in the Ice Cave demo are implemented using the Blinn technique. The following code shows you how to implement specular effects with the Blinn technique:

```
// Returns intensity of a specular effect without considering shadows
float SpecularBlinn(float3 vert2Light, float3 viewDir, float3 normalVec, float4
 power)
{
        float3 floatDir = normalize(vert2Light - viewDir);
        float specAngle = max(dot(floatDir, normalVec), 0.0);
        return pow(specAngle, power);
}
```

A disadvantage of the Blinn technique is that it can produce incorrect results in certain circumstances. For example, specular effects can appear in regions that are in shadow.

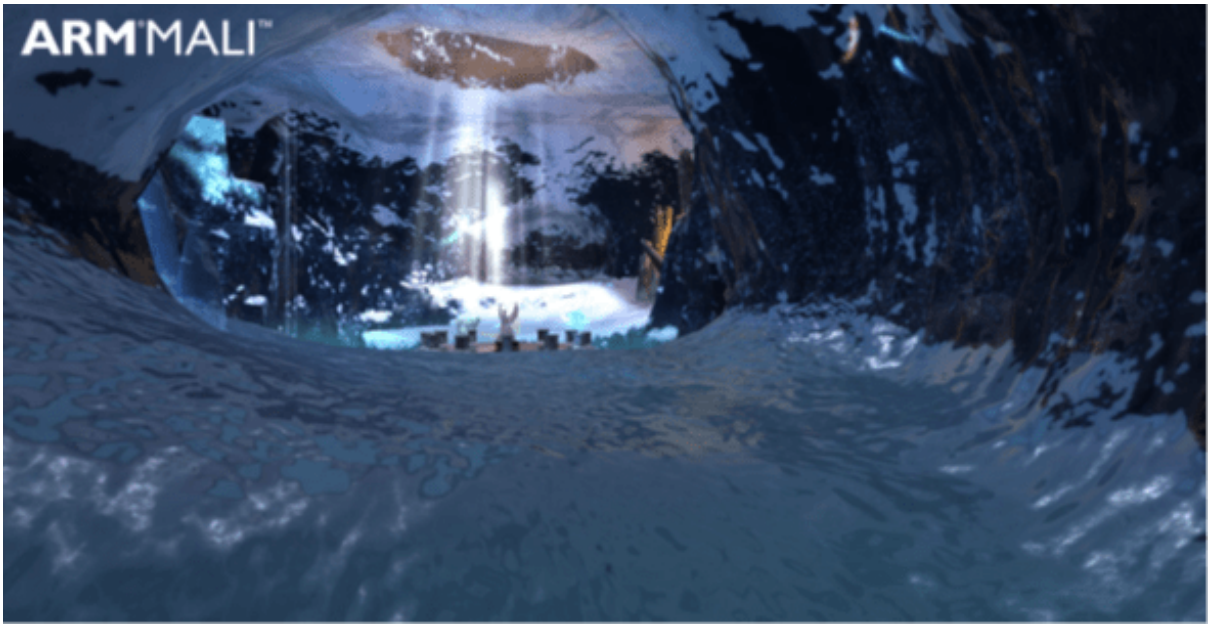
The following image shows an example of a shadowed region with no specular effects:

Figure 2-1: Shadowed region with no specular effects



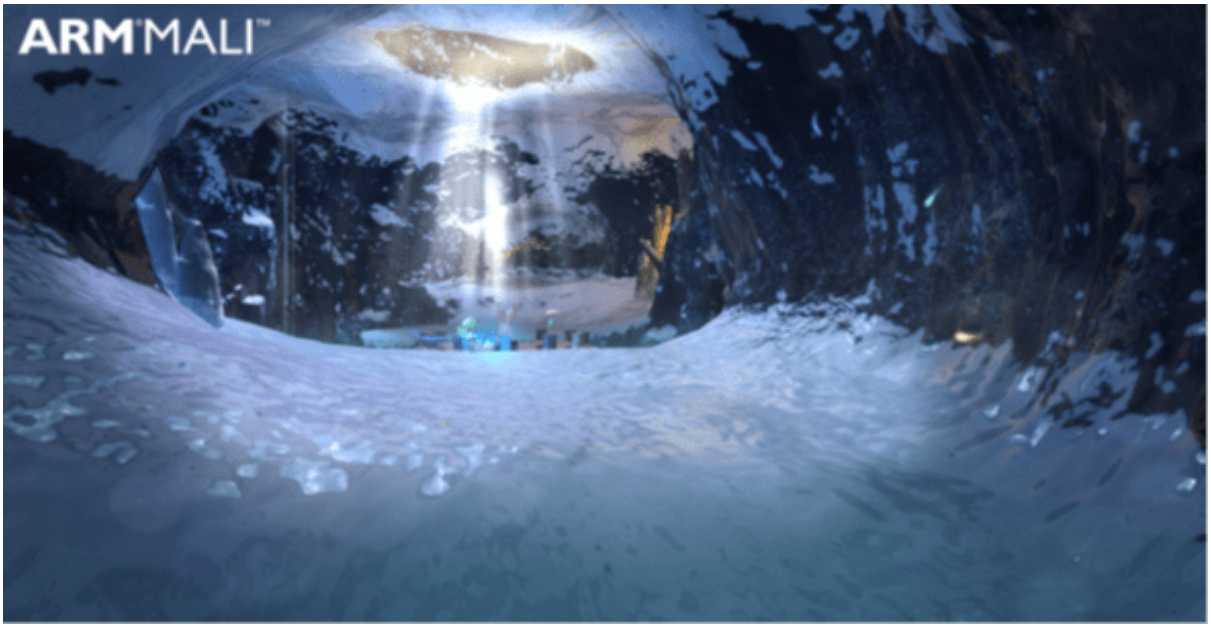
The following image shows an example of specular effects in a shadowed region, which are incorrect because the shadows are represented incorrectly:

Figure 2-2: Shadowed region with incorrect specular effects



The following image shows an example of specular effects in a lit region, which are correct:

Figure 2-3: Lit region with correct specular effects



The shadows make the specular effects intensity either stronger or weaker, depending on the light that is reaching the specular surface. All of the information that is required to correct the intensity of the specular effects is already in the Ice Cave demo, so fixing the specular effect is relatively simple.

The environment cubemap texture that is used for reflection and shadow effects contains two types of information:

- RGB: The RGB channels contain environment colors that are used for reflections.
- Alpha channels: The alpha channel contains opacity, which is used for shadows.

You can use the alpha channel to determine the specular intensity. This is because the alpha channel represents holes in the cave that let light into the environment. The alpha channel is, therefore, used to ensure that the specular effect is applied only to surfaces that are lit.

To use the alpha channel to correct the specular effect, calculate a corrected reflection vector in a fragment shader to make the reflection effect. Use the corrected reflection vector to fetch the RGBA texel from a cubemap texture. This is shown in the following code:

```
// Locally corrected static reflections
const half4 reflColor = SampleCubemapWithLocalCorrection(
ReflDirectionWS,
_ReflBBoxMinWorld,
_ReflBBoxMaxWorld,
input.vertexInWorld,
_ReflCubePosWorld,
_ReflCube);
```

For more information about creating the corrected reflection vector, or what the `SampleCubemapWithLocalCorrection()` function is, see Local cubemap rendering.

The `reflColor` value is in RGBA format where the RGB components contain color data for reflections and the alpha channel (A) contains the intensity of the specular effect. In the Ice Cave demo, the alpha channel is multiplied by the specular color, which is calculated with the Blinn technique. This is shown in the following code:

```
half3 specular = _SpecularColor.rgb *
SpecularBlinn(
input.vertexToLight01InWorld,
viewDirInWorld,
normalInWorld,
_SpecularPower) *
reflColor.a;
```

The specular value represents the final `specular` color. You can add the specular value to your lighting model, which is shown in the following code:

```
half3 pixCol = specular + reflections + diffuse + ambient;
```

3. Dirty lens effect

In this section of the guide, we explain a simple implementation of a dirty lens effect that is suitable for mobile devices. A dirty lens effect can invoke a sense of drama and is often used together with a lens flare effect.

In the Ice Cave demo, the dirty lens effect is implemented by a small shader that uses a full screen quad laid over the top of the scene. The intensity of the quad, and hence the effect, is variable, with a value passed in from a script. The intensity of the quad is passed to it by a script.

The shader renders the quad that uses additive alpha blending at the very end, after all transparent geometry has been rendered.

Dirty lens shader code

The following code shows the `DirtyLensEffect.shader` code:

```
half3 normalInWorld = half3(0.0,0.0,0.0);
half3 bumpNormal = UnpackNormal(tex2D(_BumpMapGlobal, input.tc));
half3x3 local2WorldTranspose = half3x3(input.tangentWorld,
input.bitangentWorld,
input.normalInWorld);
normalInWorld = normalize(mul(bumpNormal, local2WorldTranspose));
normalInWorld = normalInWorld*0.5 + 0.5;
return half4(normalInWorld,1.0);
```

Shader implementation

This subsection describes the Dirty lens shader.

The following subshader tag instructs the full screen quad to be rendered after all opaque geometry, and after nine other transparent objects:

```
Tags {"Queue" = "Transparent+10"}
```

The shader uses the following command to deactivate depth buffer writing, which prevents the quad from occluding the geometry behind it:

```
ZWrite Off
```

At the blending stage, the output of the fragment shader is blended with the pixel color that is already in the frame buffer.

The shader specifies the additive blending type: `Blend One One`. In this blending type, the source and the destination factors are both `float4 (1.0, 1.0, 1.0, 1.0)`.

This type of blending is often used for particle systems when they represent effects like fire that are transparent and emit light.

The shader deactivates culling and ZTest to ensure that the dirty lens effect is always rendered. This means that the vertices of the quad are defined in viewport coordinates, so that no vertex transformations take place in the vertex shader.

Then the fragment shader only fetches the texel from the texture and applies a factor that is proportional to the intensity of the effect.

The following image shows the full screen quad texture for the dirty lens effect:

Figure 3-1: Texture used for Dirty lens effect in Ice Cave demo

The following image shows how the Dirty Lens effect looks in the Ice Cave demo. Specifically, when the camera is oriented to the sunlight that is coming from the entrance of the cave:

Figure 3-2: Dirty lens effect as implemented in the Ice Cave demo

Script implementation

A simple script creates the full screen quad and calculates the intensity factor that is passed to the fragment shader.

The following function creates the mesh of the quad in the Start function:

```
void CreateQuadMesh()
{
        Mesh mesh = GetComponent().mesh;
        mesh.Clear();
        mesh.vertices = new Vector3[] {new Vector3(-1, -1, 0), new Vector3(1, -1,
 0), new Vector3(1, 1, 0), new Vector3(-1, 1, 0)};
        mesh.uv = new Vector2[] {new Vector2(0, 0), new Vector2(1, 0), new
 Vector2(1, 1), new Vector2(0, 1)};
        mesh.triangles = new int[] {0, 2, 1, 0, 3, 2};
        mesh.RecalculateNormals();
        //Increase bounds to avoid frustum clipping.
        bigBounds.SetMinMax(new Vector3(-100, -100, -100), new Vector3(100, 100,
 100));
        mesh.bounds = bigBounds;
}
```

When the mesh is created, its bounds are incremented to guarantee that it is never frustum clipped from being outside the camera view. Therefore, you must set the size of the bounds according to the size of your scene.

The script calculates the intensity factor and passes it to the fragment shader. The intensity factor is based on the relative orientation of the camera-to-sun vector and the camera forward vector. The maximum intensity of the effect takes place when the camera is looking directly at the sun.

The following code shows the calculation of the intensity factor:

```
Vector3 cameraSunVec = sun.transform.position - Camera.main.transform.position;
cameraSunVec.Normalize();
float dotProd = Vector3.Dot(Camera.main.transform.forward, cameraSunVec);
float intensityFactor = Mathf.Clamp(dotProd);
```

4. Light shafts

This section of the guide shows you how to implement light shafts into your Unity programs. Light shafts can be used to simulate the following effects:

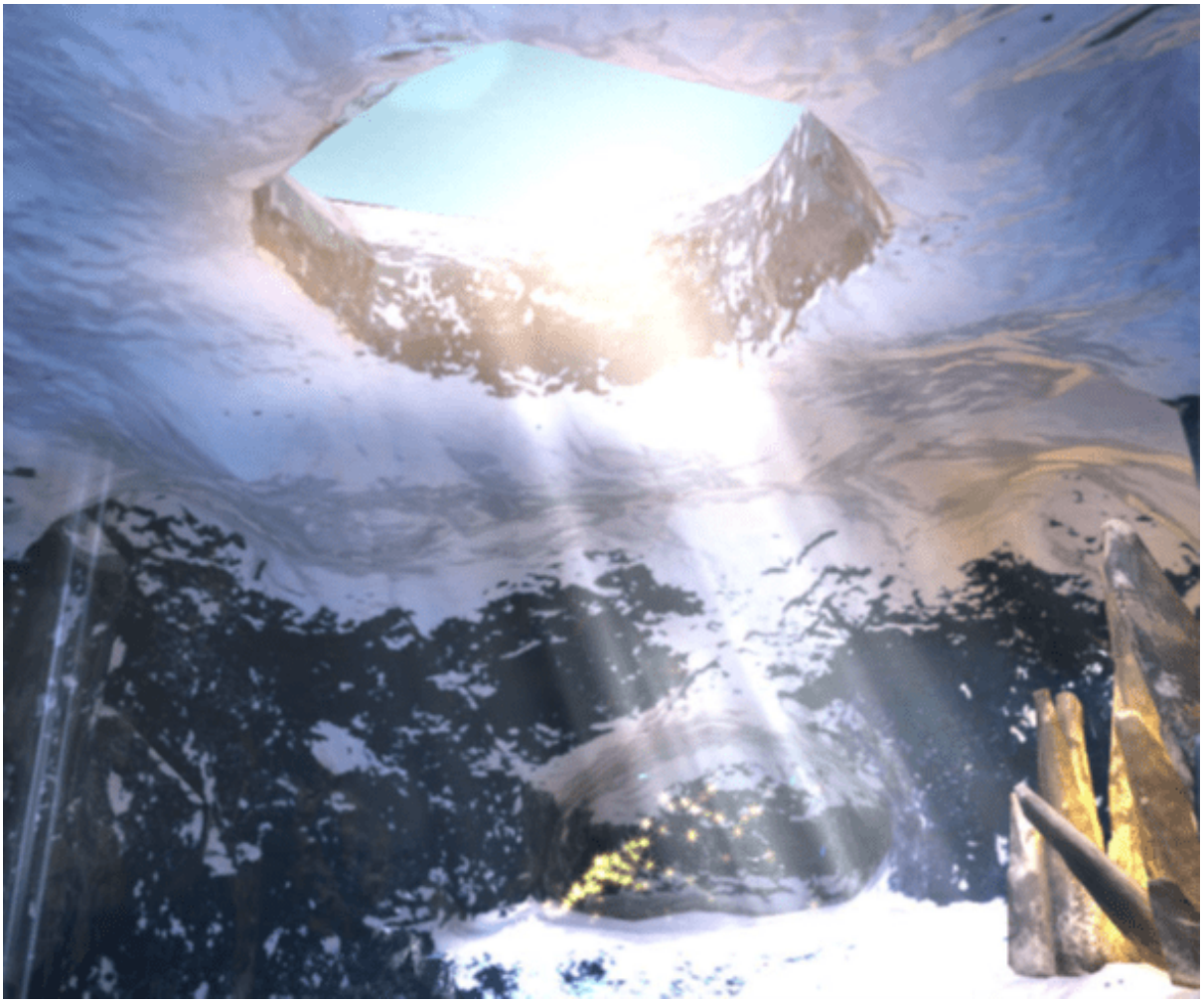
- Crepuscular rays
- Atmospheric scattering
- Shadowing

Implementing the preceding list of effects can add depth and realism to a scene.

In the Ice Cave demo, the light shaft simulates the scattering of the sun rays that are coming into the cave from the opening at the top of the cave.

The following image shows the light shafts in the Ice Cave demo:

Figure 4-1: Light shaft in the Ice Cave demo

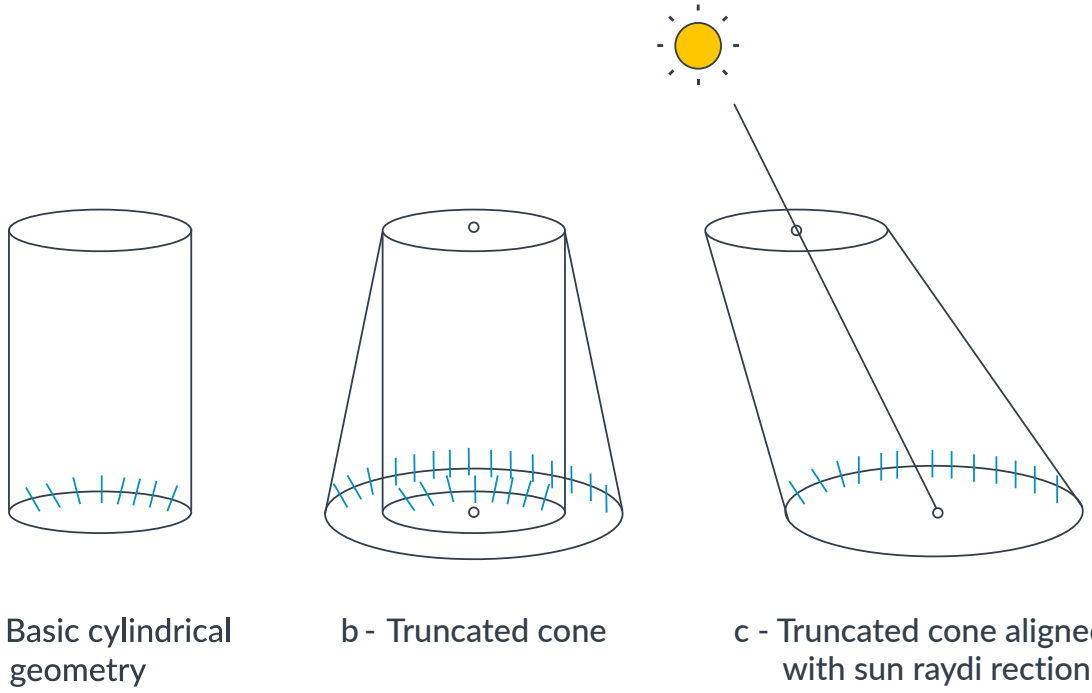


The light shaft is based on a truncated cone. The cone follows the orientation of the light rays in a manner that ensures that the upper section of the cone is always fixed.

The following diagram shows the geometry of the cone where:

- A shows that the basic geometry of the light shaft is a cylinder with two cross-sections at the top and bottom.
- B shows that the lower cross-section is expanded to achieve a truncated cone geometry based on the angle θ that you define.
- C shows that the upper cross-section is fixed, and that the lower cross-section is shifted horizontally according to the direction of the sun rays.

Figure 4-2: Light shaft geometry



A script can use the position of the sun to calculate the following values:

- The magnitude of the lower cross-section cone expansion based on the value of the angle θ that is provided as input.
- The direction and magnitude of the cross-section shift.

The vertex shader applies a transformation based on this data. The transformation is applied to the original vertices of the cylindrical geometry, in the local coordinates of the light shaft.

When rendering the light shaft, avoid rendering any hard edges that reveal its geometry. You can avoid rendering these edges by using a texture mask that smoothly fades out the top and bottom.

The following image shows the light shaft textures:

Figure 4-3: Light shaft textures

Fading out the cone edges

The following steps show you how to fade out the cone edges. This means that you fade out the light shaft intensity in the plane parallel to the cross section. The intensity of fade out depends on the relative camera-to-vertex orientation.

1. Project the camera position on the cross section, in the vertex shader.
2. Build a new vector from the center of the cross section to the projection and normalize it.
3. Calculate the dot product of this vector with the vertex normal, then raise the result to a power exponent.
4. Pass the result to the fragment shader as a varying. The result is used to modulate the intensity of the light shaft.

The vertex shader performs these calculations in the Local Coordinate System (LCS).

The following code shows the vertex shader:

```
// Project camera position onto cross section
float3 axisY = float3(0, 1, 0);
float dotWithYAxis = dot(camPosInLCS, axisY);
float3 projOnCrossSection = camPosInLCS - (axisY * dotWithYAxis);
projOnCrossSection = normalize(projOnCrossSection);
// Dot product to fade the geometry at the edge of the cross section
float dotProd = abs(dot(projOnCrossSection, input.normal));
output.overallIntensity = pow(dotProd, _FadingEdgePower) * _CurrLightShaftIntensity;
```

The coefficient `_FadingEdgePower` enables you to fine-tune the fading of the light shaft edges.

The script passes the coefficient `_CurrLightShaftIntensity`. Passing the coefficient makes the light shaft fade out when the camera gets close to it.

The following code shows a final touch that is added to the light shaft by slowly scrolling down the texture from a script:

```
void Update()
{
        float localOffset = (Time.time * speed) + offset;
        localOffset = localOffset % 1.0f;
        GetComponent().material.SetTextureOffset("_MainTex", new Vector2(0,
 localOffset));
}
The fragment shader fetches the beam and mask textures, and then combines them with
 the intensity factor:
float4 frag(vertexOutput input) : COLOR
{
        float4 textureColor = tex2D(_MainTex, input.tex.xy);
        float textureMask = tex2D(_MaskTex, input.tex.zw).a;
        textureColor *= input.overallIntensity * textureMask;
        textureColor.rgb = clamp(textureColor.a, 0, 1);
        return textureColor;
}
```

The light shaft geometry is rendered in the transparent queue after all the opaque geometry.

The light shaft uses additive blending to combine the fragment color with the corresponding pixel in the frame buffer.

The shader also disables culling and writing to depth buffer, so that it does not occlude other objects.

The settings for the pass are the following:

```
Blend One One
Cull Off
ZWrite Off
```

5. Fog effects

This section shows you how fog effects are used in the Ice Cave demo. A fog effect adds atmosphere to a scene. There are advanced fog implementations, but these are too expensive for mobile and a simple fog effect can be effective.

Fog is common in the real world, so adding this effect to your game adds to the sense of realism. In the real world, whether or not it is foggy, the colors fade the further you look in the distance. You can see this effect on sunny days, especially when looking at mountains.

This section describes two non-expensive versions of the fog effect:

- Procedural linear fog
- Particle-based fog

You can apply both effects simultaneously. The Ice Cave demo uses both techniques.

Procedural linear fog

Procedural linear fog ensures that the further away the object is, the more its colors fade to the defined fog color. To achieve this effect in your fragment shader, you can use simple linear interpolation between the fragment color and the fog color.

The following code shows you how to calculate the fog color in the vertex shader, based on the distance to the camera. This color is passed to fragment shader as a varying:

```
output.fogColor = _FogColor * clamp(vertexDistance * _FogDistanceScale, 0.0, 1.0);
```

The following list describes what is going on in the code:

- `vertexDistance` is the vertex to camera distance.
- `_FogDistanceScale` is a factor that is passed to the shader as a uniform.
- `_FogColor` is the base fog color that you define and is passed as a uniform.

In the fragment shader, the interpolated `input.fogColor` is combined with the fragment color `output.Color`. This is shown in the following code:

```
outputColor = lerp(outputColor, input.fogColor.rgb, input.fogColor.a);
```

The following image shows the result after applying the preceding code to your scene:

Figure 5-1: Linear fog based on distance

Note

Calculating the fog in a shader means that you are not required to do any extra post processing to produce the fog effect.

Try to merge as many effects as possible into one shader. However, check performance because if the cache is exceeded this means that the performance might reduce. Therefore, the shader may need to be split into two or more passes.

You can perform the fog color calculation in the vertex or fragment shader. The calculation in the fragment shader is more accurate but requires more compute performance. This is because it is calculated for every fragment.

The vertex shader calculation is lower precision but higher performance because it is only computed once per vertex.

Linear fog with height

Fog is applied uniformly across your scene. You can make fog more realistic by changing the density according to height. For example, make the fog denser lower down and thinner higher up. You can expose the height level, to adjust it manually.

The following image shows linear fog based on distance and height:

Figure 5-2: Linear fog based on distance and height

Non-uniform fog

The fog does not have to be uniform. You can make fog more visually interesting, for example non-uniform, by introducing some noise, for example, applying a noise texture.

For a more complex effect, you can also apply more than one noise texture and make those textures slide with different speeds. For instance, the noise texture that is further away slides more slowly than the texture that is closer to the camera.

You can apply more than one texture in a single pass in one shader, by blending the noise textures according to the distance.

Pre-baked fog

You can pre-bake fog into textures if the camera does not get close to them. This reduces the compute power required.

Volumetric fog that uses particles

You can simulate volumetric fog with particles. This technique can provide high-quality results.

Keep the number of particles to a minimum. Particles increase overdraw because more shaders are executed per fragment. If overdraw occurs, try using larger particles instead of creating more particles.

In the Ice Cave demo, a maximum of 15 particles are used at a time.

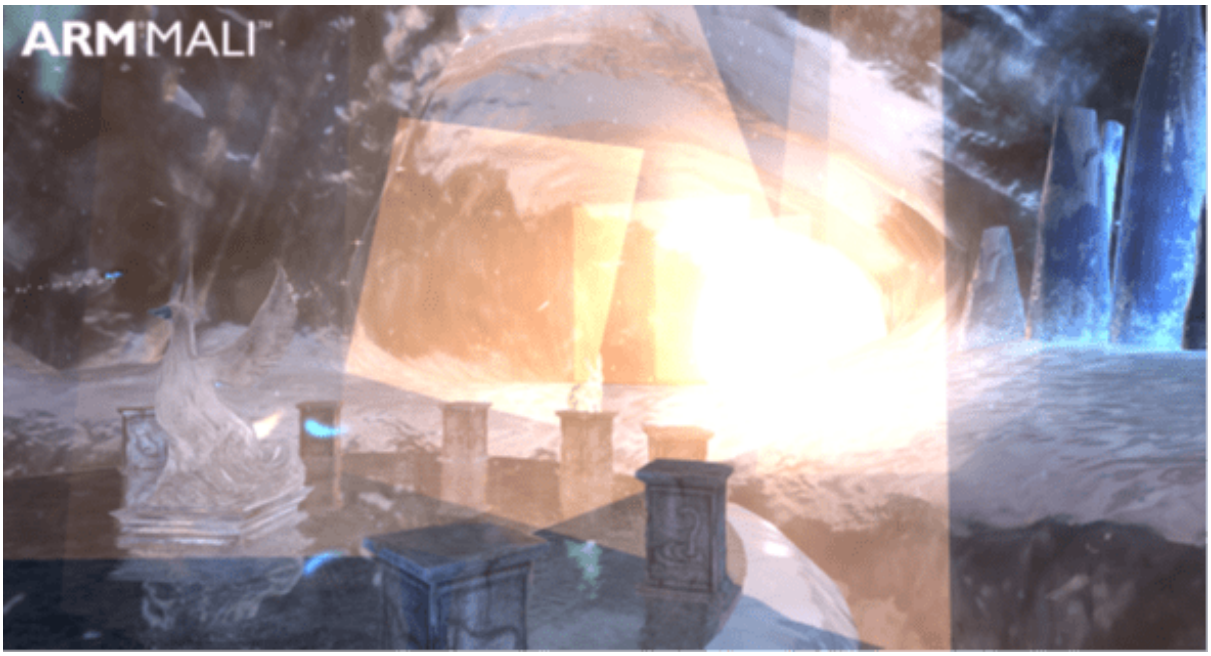
Let's look at different types of fog effects that use particles.

Geometry instead of billboard rendering

Set the Render Mode of your particle system to Mesh instead of Billboard. To achieve a volumetric effect, the individual particles must have random rotations.

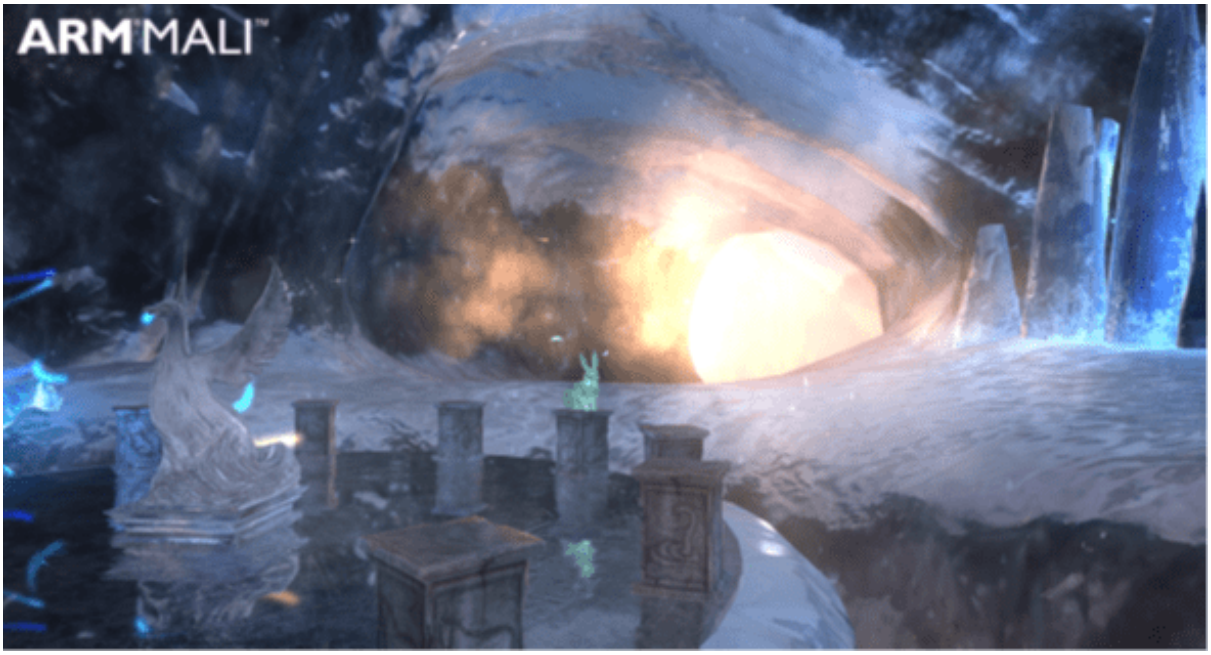
The following image shows the particle geometry without textures:

Figure 5-3: Particles without texture



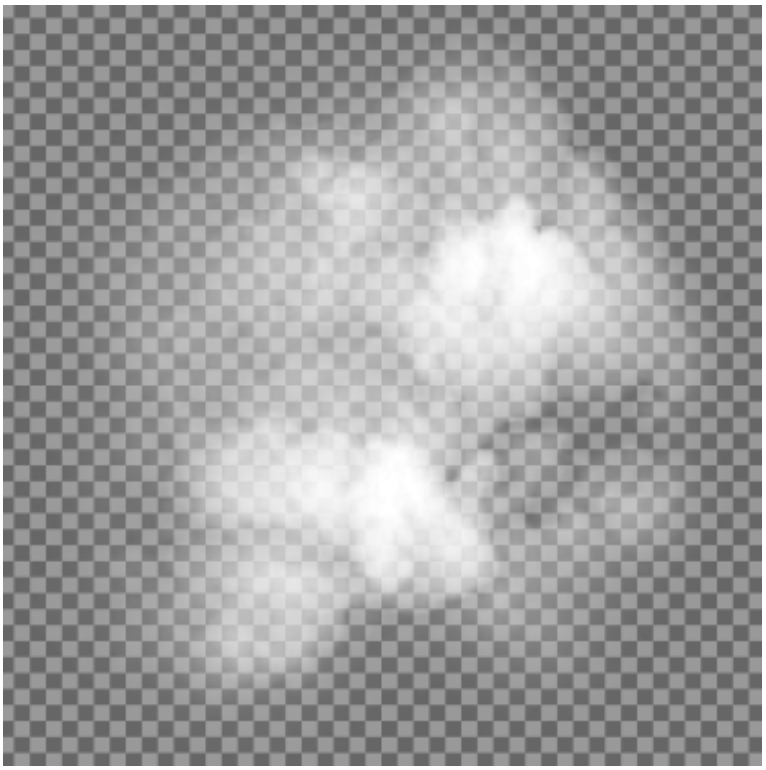
The following image shows the particle geometry with textures:

Figure 5-4: Particles with texture



The following image shows the texture that is applied to each particle:

Figure 5-5: Texture of an individual particle



Angle fade effect

Fade the individual particles in and out according to their orientation compared to the camera position. If individual particles do not fade in and out, the sharp edges of the particles are visible.

The following example code shows a vertex shader that performs the fading:

```
half4 vertexInWorld = mul(_Object2World, input.vertex);
half3 normalInWorld = (mul(half4(input.normal, 0.0), _World2Object).xyz);
const half3 viewDirInWorld = normalize(vertexInWorld - _WorldSpaceCameraPos);
output.visibility = abs(dot(-normalInWorld, viewDirInWorld));
output.visibility *= output.visibility; // instead of power of 2
```

The varying parameter `output.visibility` is interpolated across the particle polygon. Read this value in a fragment shader and apply an amount of transparency. This is shown in the following code:

```
half4 diffuseTex = _Color * tex2D(_MainTex, half2(input.texCoord));
diffuseTex *= input.visibility;
return diffuseTex;
```

Rendering particles

Use the following steps to render for particles:

1. Render particles as the last primitives within the frame, by adding the following line:

   ```
   Tags { "Queue" = "Transparent+10" }
   ```

   
   Note

   The +10 is present in this example because the Ice Cave demo renders nine other transparent objects before the particles.

2. Set the appropriate blending mode, by adding the following line at the beginning of a shader pass:

   ```
   Blend SrcAlpha One
   ```

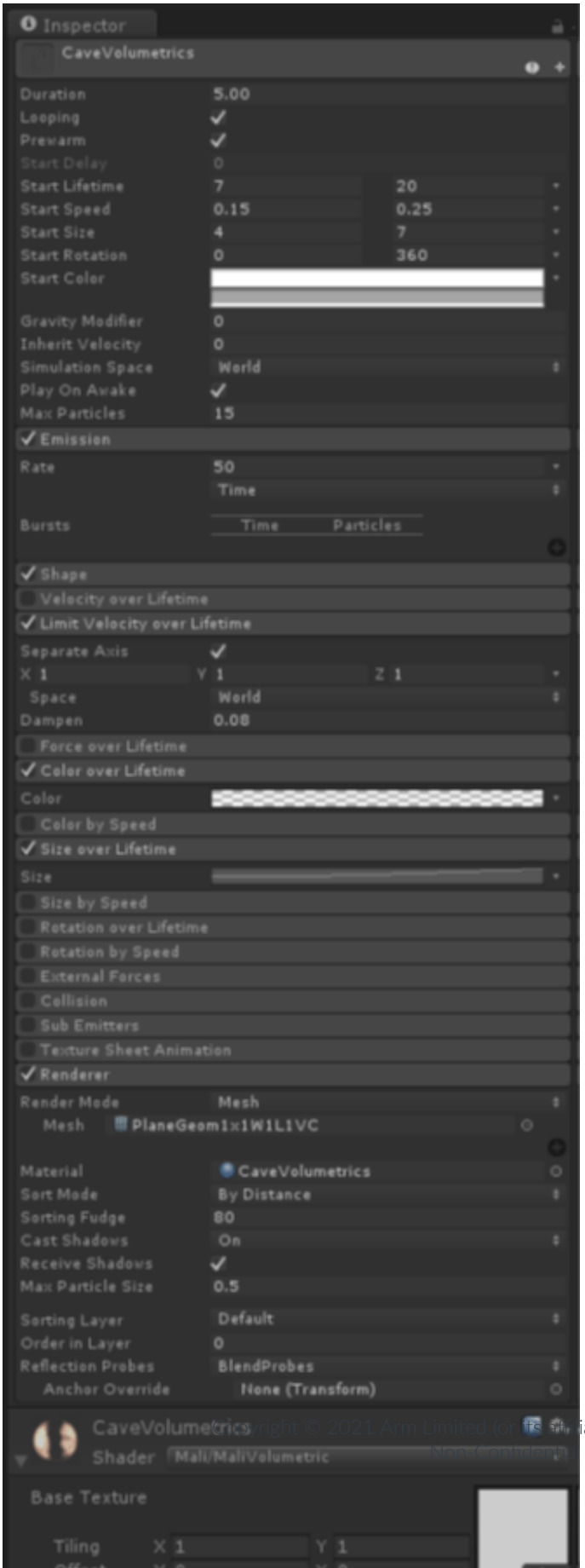
3. Disable writing to z-buffer, by adding the following line:

   ```
   ZWrite Off
   ```

Particle system settings

The following screenshot shows the settings that are used in the Ice Cave demo for the particle system:

Figure 5-6: Particle system parameters from the Ice Cave demo



The following screenshot shows the area where the particles are spawned, which is indicated by the box in the image. The box is defined by the built-in option Shape in the Unity Particle System:

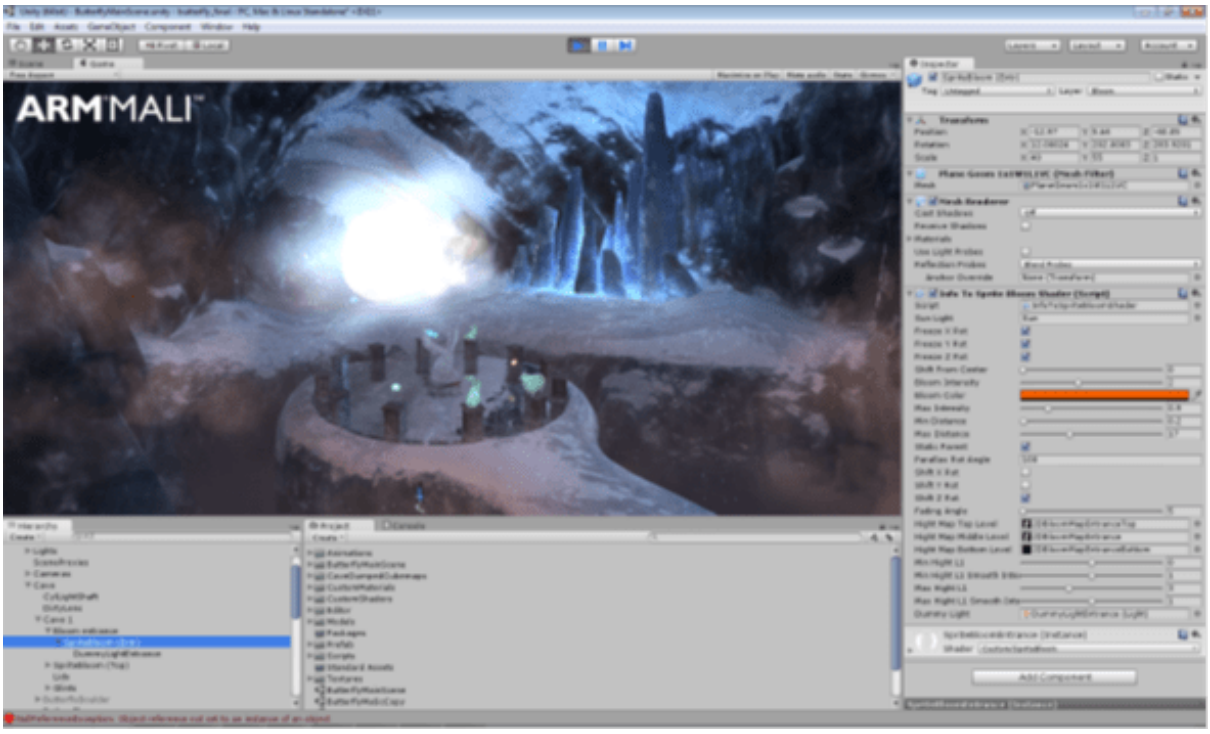
Figure 5-7: The particles box definition where the particles are spawned

6. Bloom

This section describes bloom effects. Bloom is used to reproduce the effects that occur in real cameras when taking pictures in a bright environment. The bloom effect simulates fringes of light that extend from the borders of bright areas, which creates the illusion of a bright light overwhelming the camera.

The following screenshot shows bloom at the entrance of the cave in the Ice Cave demo:

Figure 6-1: Bloom at the entrance of the cave as implemented in the Ice Cave demo



The bloom effect occurs on real lenses because they cannot focus perfectly. When the light passes through the circular aperture of the camera, some light is diffracted, creating a bright disk around the image. This effect is not typically noticeable under most conditions, but it is visible under intense lighting.

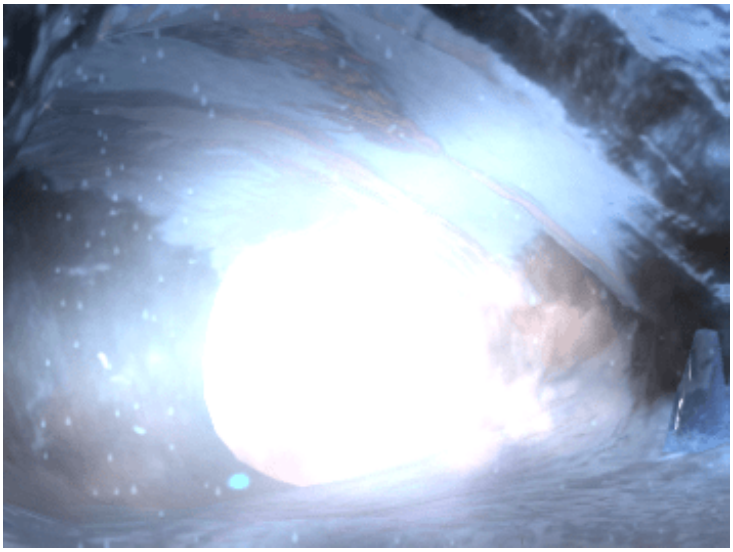
Implementing bloom

Bloom effects are typically implemented as a post-processing effect. Generating effects this way can use a large amount of computing power, so if possible, avoid them in mobile games. This is especially true if your game, like the Ice Cave demo, uses many complex effects.

One alternative approach uses a simple plane. This is where you place the plane between the camera and the light source. The plane must be oriented with the normal pointing to the area where the camera will be situated.

To create a bloom effect this way, place a plane where you want the bloom effect to occur. Modulate the intensity of the effect using a factor that is based on the alignment of the view vector, with the plane normal, and the light source. This method is implemented in the Ice Cave demo, as shown in the following image:

Figure 6-2: Plane at the entrance of the cave implementing bloom



Another way to avoid post-processing is to bake your bloom. If we have a map, for example a glossiness map, we can use that to determine where bloom must occur. A map can work for the static areas of your scene.


Note

If we do resort to post-processing, there are more efficient ways to do the bloom, so we prefer the Dual Filtering technique, for more information see Post-processed bloom.

Planar bloom: creating a bloom modulation factor script

The alignment factor can be calculated using a script. This factor alters the bloom effect depending on the angle that the camera is viewing the effect from.

For example, the following script calculates the alignment factor for a plane that it is attached to:

```
// Light-plane normal-camera alignment
Vector3 planeToCamVec = Camera.main.transform.position -
 gameObject.transform.position;
planeToCamVec.Normalize();
Vector3 sunLightToPlainVec = origPlainPos - sunLight.transform.position;
sunLightToPlainVec.Normalize();
float sunLightPlainCameraAlignment = Vector3.Dot(plainToCamVec, sunLightToPlainVec);
sunLightPlainCameraAlignment = Mathf.Clamp (sunLightPlainCameraAlignment, 0.0f,
 1.0f);
```

The alignment factor `sunLightPlaneCameraAlignment` is passed to the shader to modulate the intensity of the rendered color.

Planar bloom; the vertex shader

The vertex shader receives a `sunLightPlainCameraAlignment` factor and uses it to modulate the intensity of the rendered bloom color.

The vertex shader applies the MVP matrix, which passes the texture coordinates to the fragment shader, and outputs the vertex coordinates. This is shown in the following code

```
vertexOutput vert(vertexInput input)
{
        vertexOutput output;
        output.tex = input.texcoord;
        output.pos = mul(UNITY_MATRIX_MVP, input.vertex);
        return output;
}
```

Planar bloom: the fragment shader

The fragment shader fetches the texture color and increments it in using a `CurrBloomColor` tint color, which is passed as a uniform variable. The alignment factor modulates the color before it is used.

The following code shows how this process is performed:

```
half4 frag(vertexOutput input) : COLOR
{
        half4 textureColor = tex2D(_MainTex, input.tex.xy);
        textureColor += textureColor * _CurrBloomColor;
        return textureColor * _AligmentFactor;
}
```

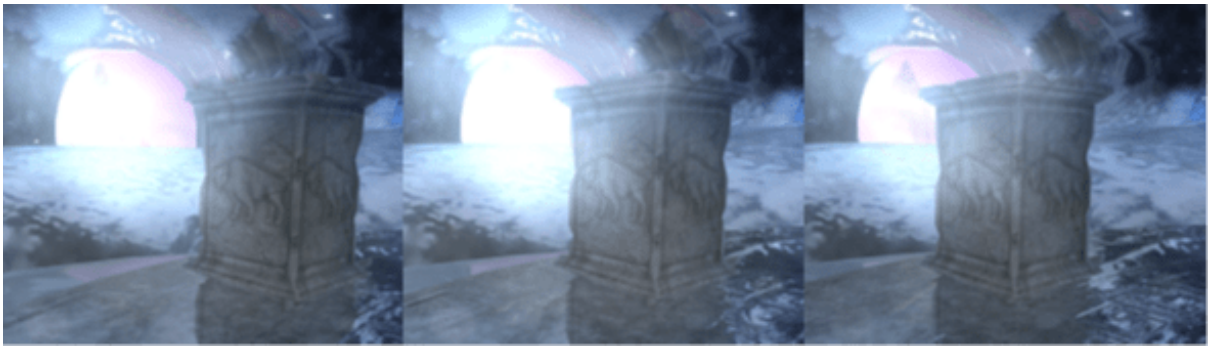
The bloom plane is rendered in the transparent queue after all opaque geometry. In the shader, use the following queue tag to set the rendering order:

```
Tags { "Queue" = "Transparent" + 1}
```

The bloom plane is applied using additive one-to-one blending, to combine the fragment color with the corresponding pixel which is already stored in the frame buffer. The shader also disables writing to the depth buffer using the instruction, `ZWrite Off`, so that existing objects are not occluded.

The following image shows the texture that the Ice Cave demo uses for its bloom effect, and the result that it creates:

Figure 6-3: Entering the occlusing zone behind an opaque plinth



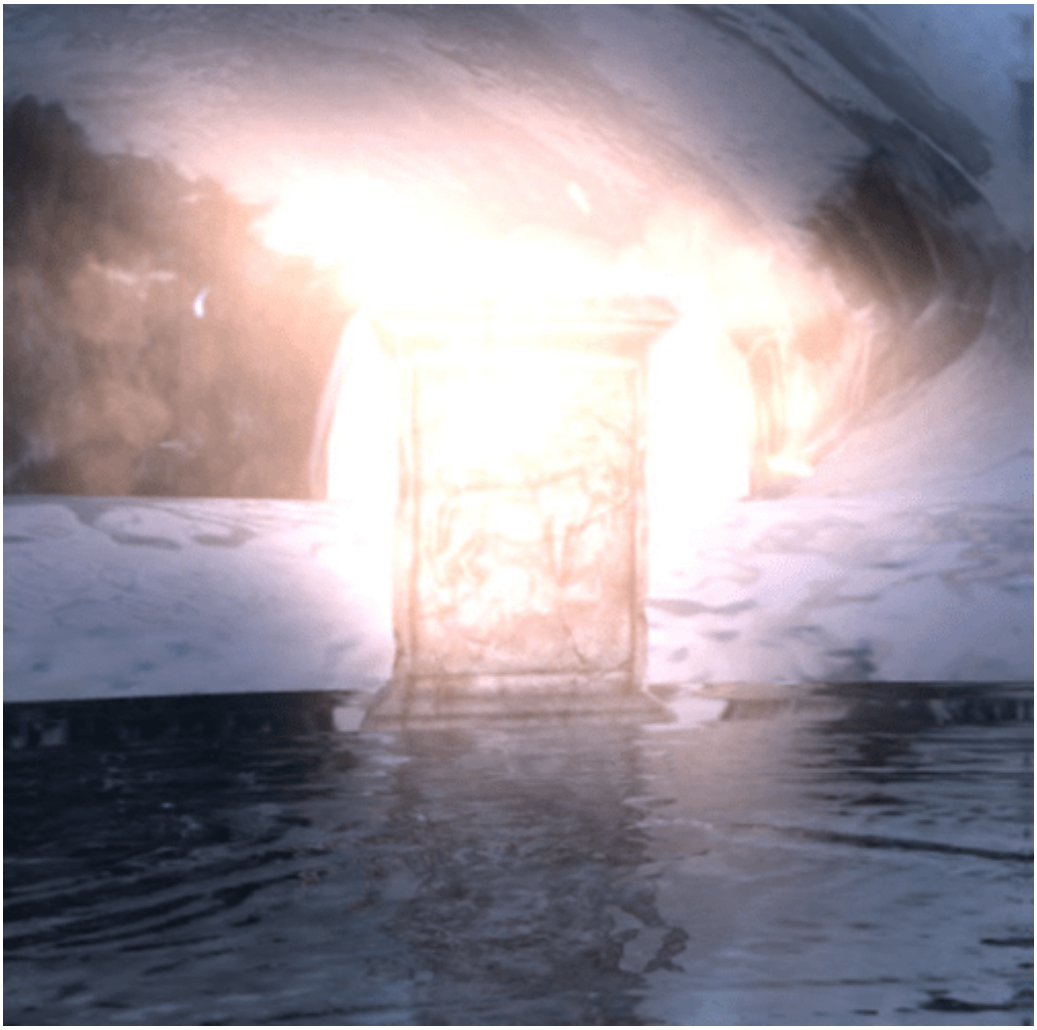
Planar bloom: correcting the bloom effect

Using a plane to produce a bloom effect is simple and suitable for mobile devices. However, it does not produce the expected results when an opaque object is placed between the light source producing the bloom and the camera. You can produce the expected results by using an occlusion map.

The incorrect appearance of the bloom effect behind an opaque object occurs because the effect is rendered in the transparent queue. Therefore, the blending occurs on top of any objects that are in front of the bloom plane.

The following image shows an example of the incorrect bloom effect that is produced without any fixes:

Figure 6-4: Incorrect bloom effect



To prevent an incorrect bloom effect from occurring, you can alter the script which calculates the alignment factor so that it processes an extra occlusion map. This occlusion map describes the areas where the camera normally applies the bloom effect incorrectly. These conflicting areas are assigned an occlusion map value of zero. This factor combines with the alignment factor to set the intensity of factor in these places to zero. This change sets the bloom to zero so that it no longer renders over objects which must occlude it.

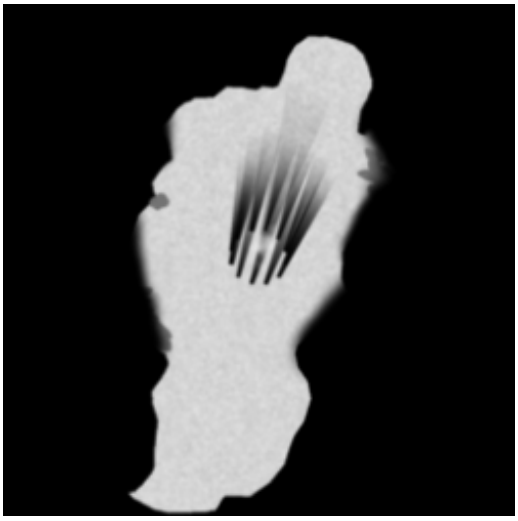
The following code implements this adjustment:

```
IntensityFactor = alignmentFactor * occlusionMapFactor
```

The Ice Cave demo camera can move freely in three dimensions. Therefore, three gray scale maps are used, each one covering a different height. Black means zero, so the occlusion map factor multiplied by the alignment factor makes the final intensity zero, occluding the bloom. White means one, so the intensity is equal to the alignment factor and the bloom is not occluded.

The following image shows the `occlusionMapFactor` at ground level:

Figure 6-5: Ground level bloom occlusion map

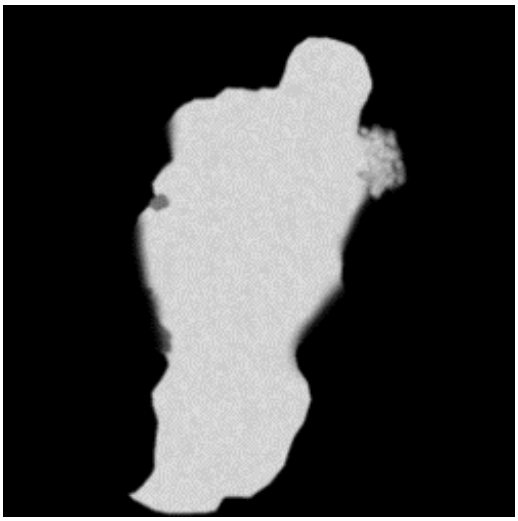



Note

The black radiating areas in the middle of the map are zones behind the opaque plinths that occlude the bloom effect from the entrance of the cave.

There are no occluding objects at a height Hmax above ground level. Therefore, the bloom occlusion map above ground level is white. The following image shows the `occlusionMapFactor` above ground level:

Figure 6-6: Above ground level bloom occlusion map



Below the height Hmin, and below ground level the bloom effect is completely occluded. Therefore, the bloom occlusion map below ground level is completely black. The white contour is

added so that the map is visible. The following image shows the `occlusionMapFactor` below ground level:

Figure 6-7: Below ground level bloom occlusion map

Planar bloom: interpolation between occlusion maps

In the Ice Cave demo, the script calculates the XZ projection of the camera position on the 2D cave map every frame. The script also normalizes the result between zero and one.

If the camera height is below Hmin or above Hmax, the normalized coordinates are used to fetch the color value from a single map. If the camera height is between Hmin and Hmax, the color is retrieved from both maps and interpolated.

This interpolation creates a smooth transition between the effects at different heights.

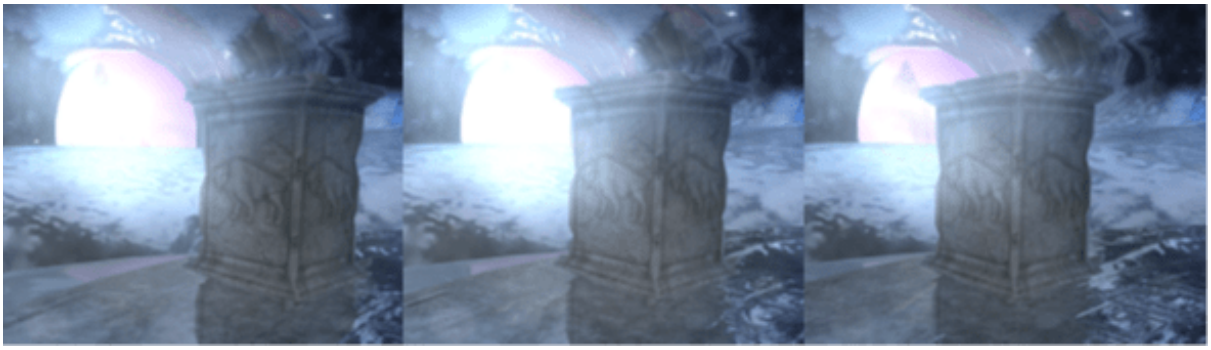
The Ice Cave demo uses the `GetPixelBilinear()` function to retrieve the colors from the map or maps.

This function returns a filter color value that uses the following code:

```
float groundOcclusionFactor =
 groundOcclusionMap.GetPixelBilinear(camPosXZNormalized.x, camPosXZNormalized.y)).r;
```

Using the bloom occlusion maps prevents the blending of the bloom on top of occluding opaque object. The following image shows a sequence of images with the resulting effect. When the camera is entering and leaving, the occluding black is behind the opaque plinth, preventing the incorrect bloom:

Figure 6-8: Sequence showing the camera entering the occluding zone behind the opaque plinth



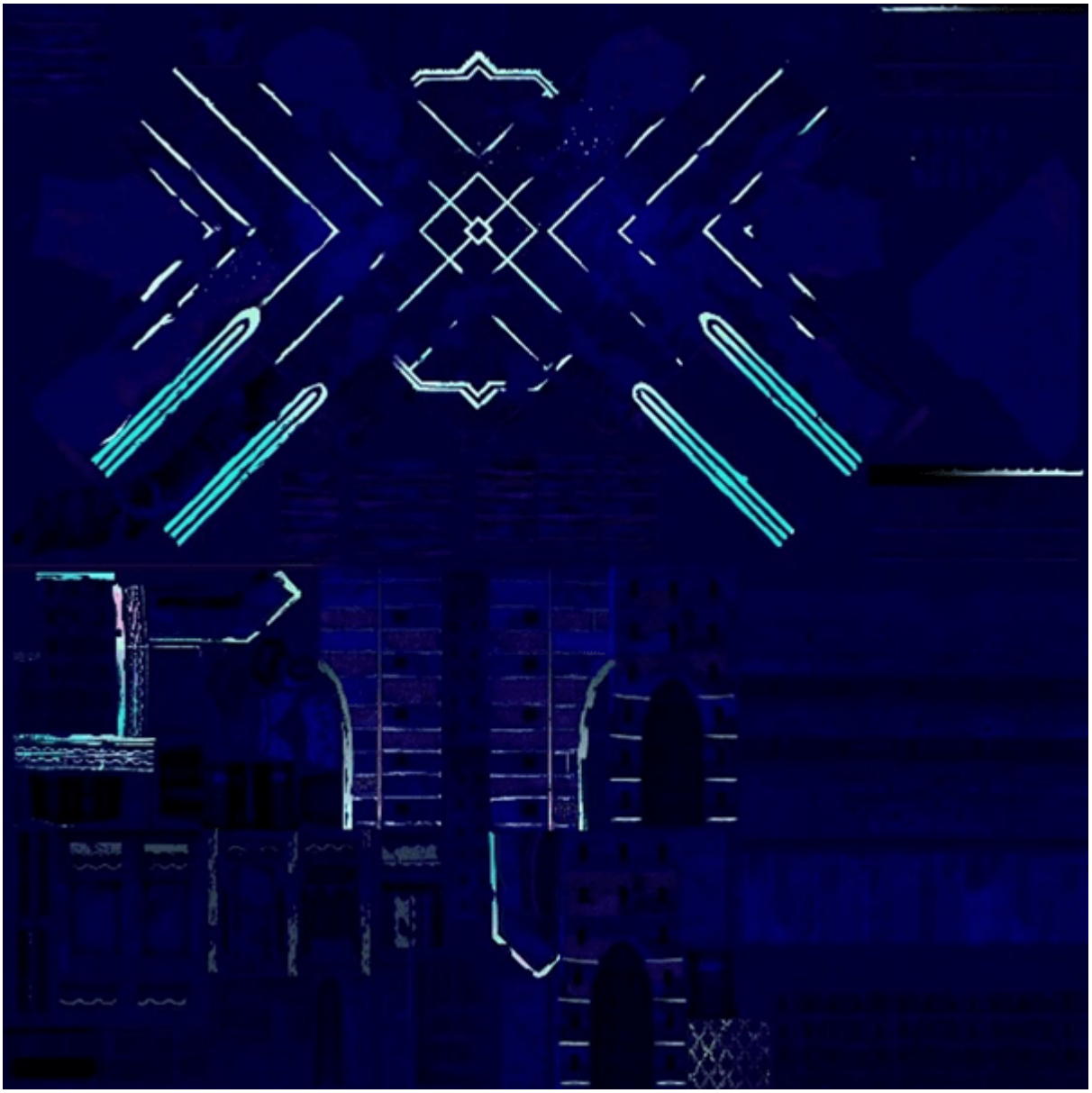
Baking bloom

For static terrain, it is possible to use a baked bloom map to efficiently produce the bloom effect.

In the game Spellsouls by Nordeus, a glossiness map from the terrain was a good starting point to produce a baked bloom map.

The following image shows the initial glossiness texture:

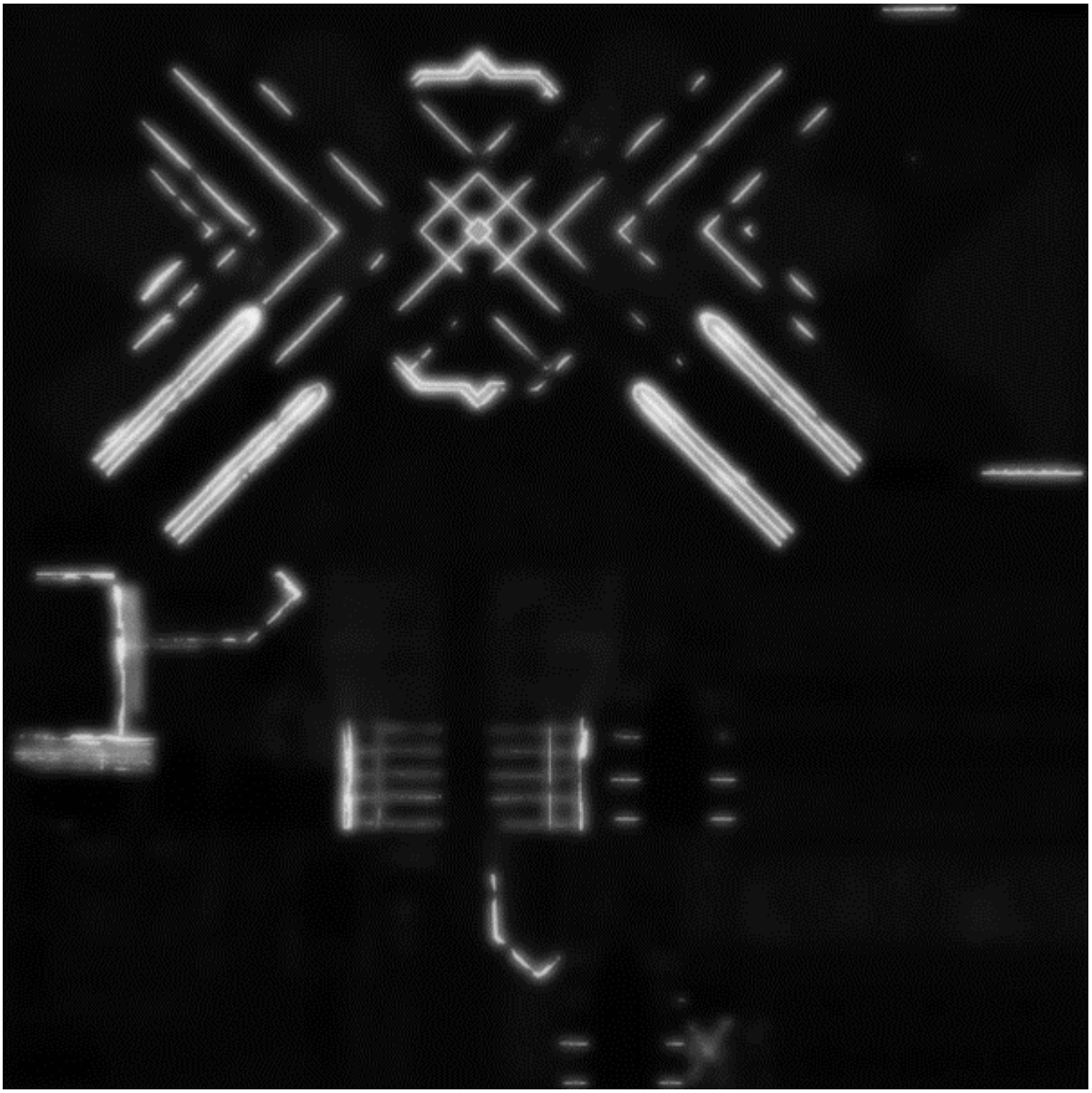
Figure 6-9: Initial glossiness texture



From the initial glossiness texture, we can generate the bloom map by creating an R8 scaled luminance map, and then blurring. This texture can then be stored in the alpha channel of the glossiness map itself. This means that you have one less texture access.

The following image shows a luminance map, stored in alpha channel:

Figure 6-10: Luminance map stored in alpha channel



In the vertex shader, we compute an alignment factor between the reflected light vector and the camera vector. This is shown in the following code:

```
// Vertex shader
float lightObjCameraAlignment = dot(objToCam, reftLightDir);
half alignmentFactor = clamp(lightObjCameraAlignment, 0.0, 1.0);
```

Then in the fragment shader we make a pixel brighter based on the following:

- The data we sample from the bloom map
- The alignment factor that we previously computed

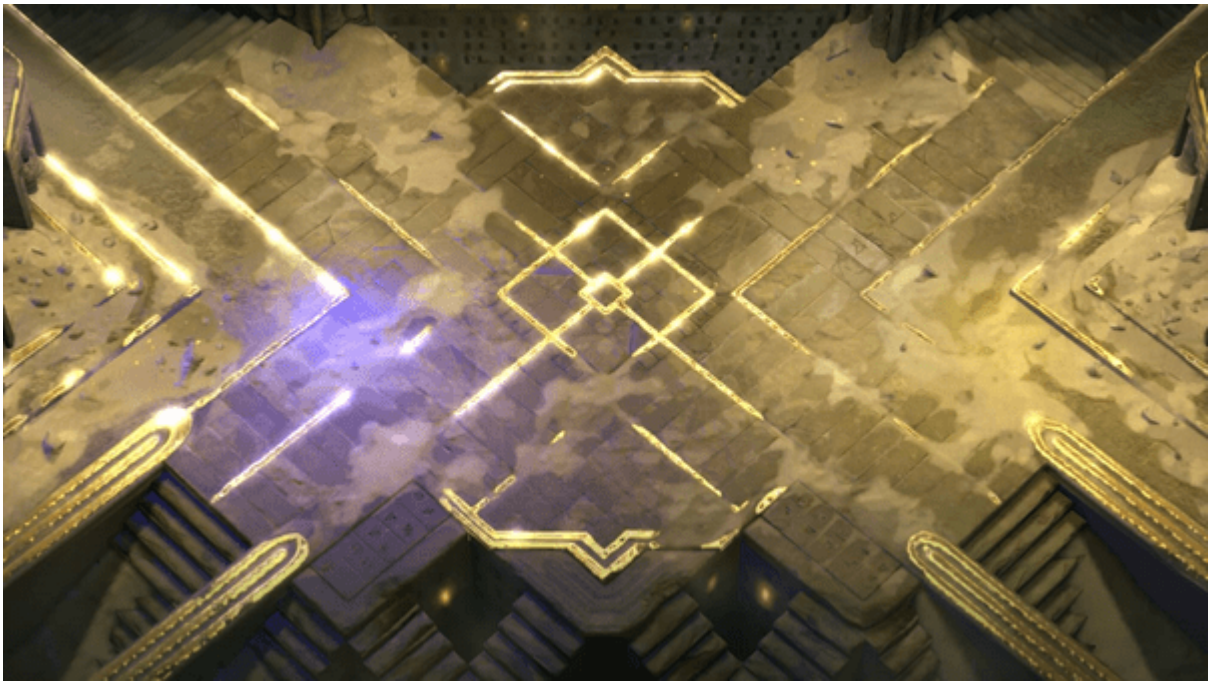
- A `BloomStrength` factor that we can tune

How to make a pixel brighter, based on the preceding list, is shown in the following code:

```
// Fragment shader
half bloom = rawGlossMap.a;
finalColor += finalColor * bloom * i.alignmentFactor * _BloomStrength;
```

The following image shows a Spellsouls terrain, with bloom applied:

Figure 6-11: Spellsouls bloom with Dual Filtering



This technique might produce a less compelling bloom. But the cost is massively reduced, so the trade-off is often worth it.

However, this technique does not work well for dynamic objects because the bloom cannot bleed off the object well. Therefore, game designer Nordeus, used the planar bloom, that is described earlier in this section, for the characters in Spellsouls.

Post-processed bloom

It is important to choose the most efficient bloom technique possible, even if you have enough rendering overheard to be able to handle a post-processing pass.

Before Nordeus in Spellsouls used the Baked and Planar bloom for their terrain and characters respectively, there were experiments on getting the most efficient post-processed bloom.

On mobile, the standard Unity post-processing bloom is too slow, but can also be too general. For example, you cannot choose to only have certain materials blooming. You must have either all materials blooming, or none.

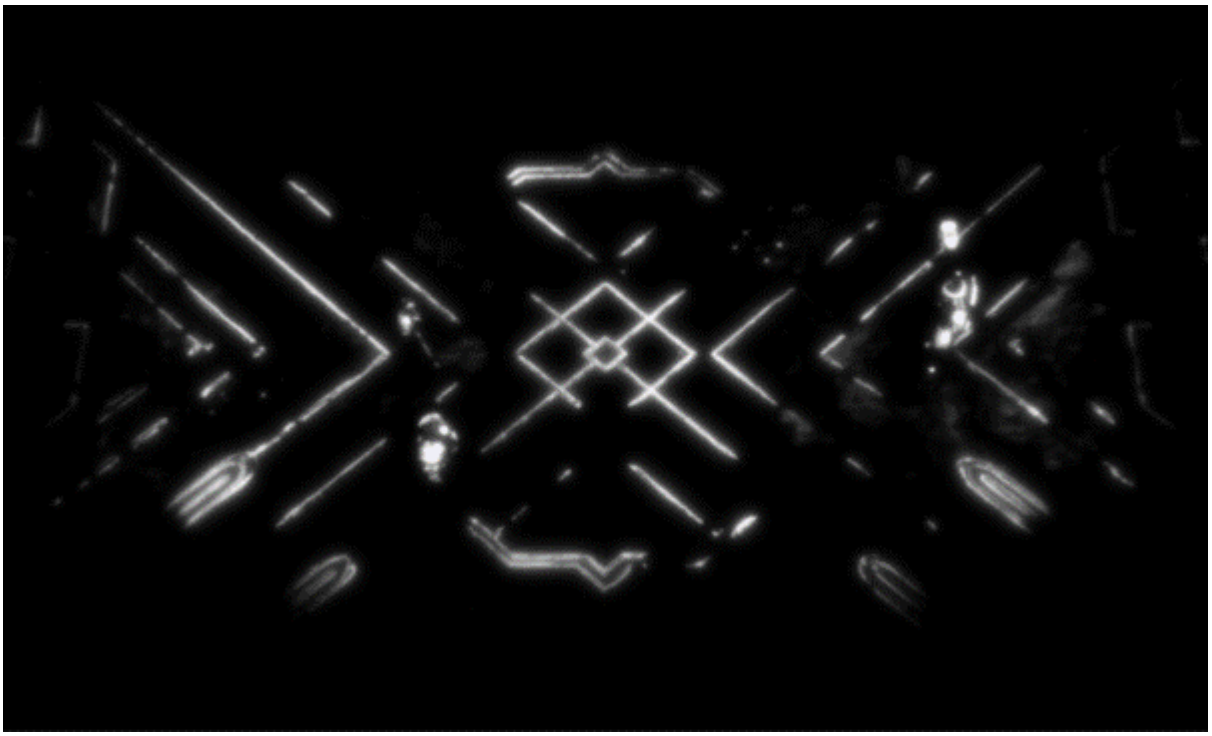
For a more efficient post-process technique, you can use Multiple Render Target (MRT) to render only those objects that you want to bloom into a separate texture. For your blooming objects, render where there is specular greater than 1 into the separate texture. This texture will be the input to a custom bloom pipeline.

When developing for mobile, we must save the power and heat that is caused by excess memory bandwidth. Therefore, we use an R8 frame buffer, storing just the scaled luminance of the pixel.

With the values greater than one being captured into the texture, an HDR-like effect is achieved without an HDR frame buffer. In the MRT step, apply a scaling factor to represent values greater than 1 with fixed point.

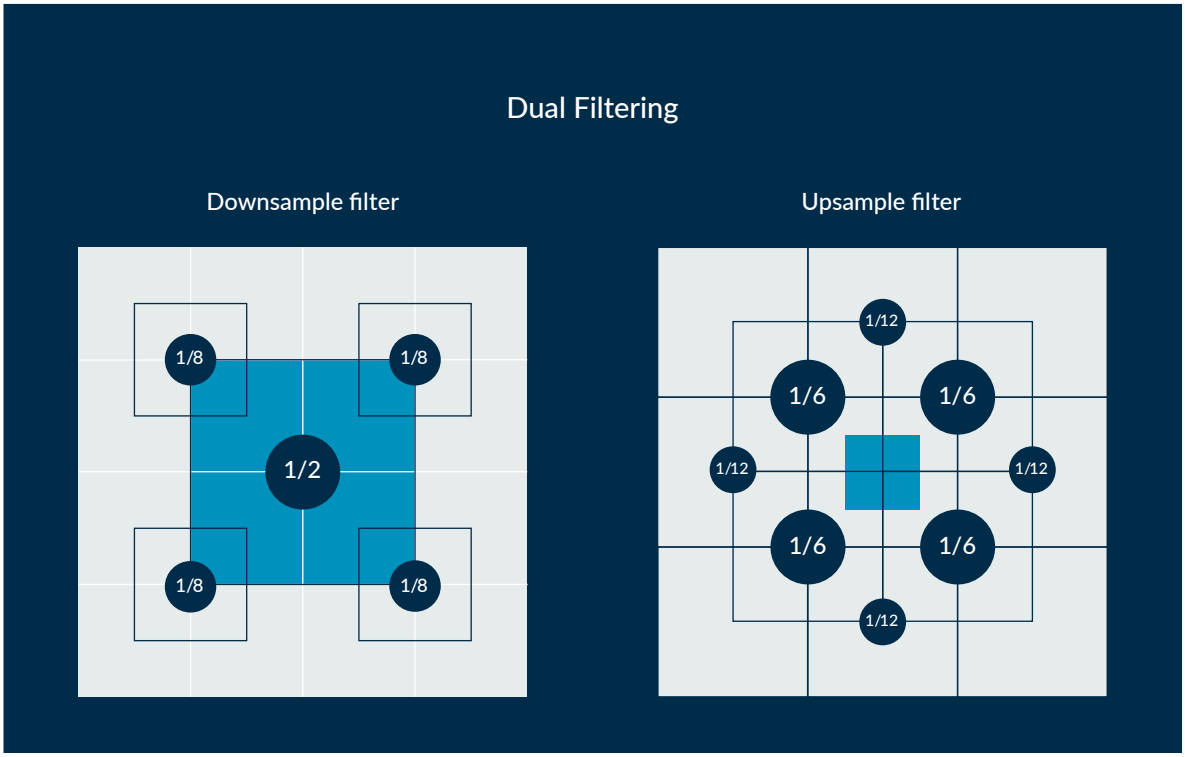
The following image shows MRT R8 bloom texture:

Figure 6-12: MRT R8 bloom texture



The blur step of bloom is the most expensive part, so this is where it is most important to use the correct technique. Our recommendation is called dual filtering, which features optimized downscaling or upscaling filters. Dual Filtering achieves a stronger effect, like a larger Gaussian radius, at a much lower cost, to be specific it has 14 times performance improvement at 1080p. An example of dual filtering is shown in the following image:

Figure 6-13: Dual Filtering



Dual filtering uses bilinear filters that are in the GPU hardware to allow fewer samples to be done. By sampling between pixels, we get a weighted average of those pixel values in one sample.

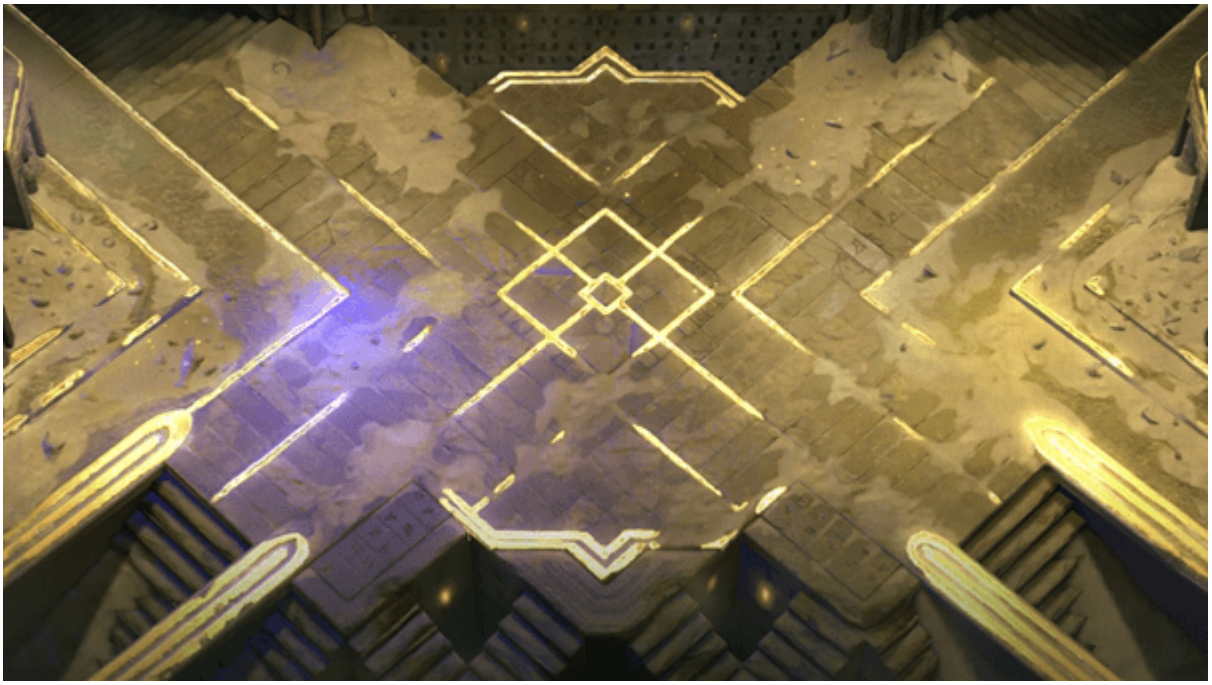
On the down sample, four samples are required to smudge in some of all the neighboring pixels, along with one sample of the center pixel itself. On the up sample, we reconstruct information from the down sample pass. The up-sample pattern was chosen to get a smooth circular shape to the blur. Different shapes can be chosen for different artistic looks, and different distances to make the blur bigger or smaller.

Note

For further reading on the filter, see Marius Bjorges Dual Filtering presentation notes.

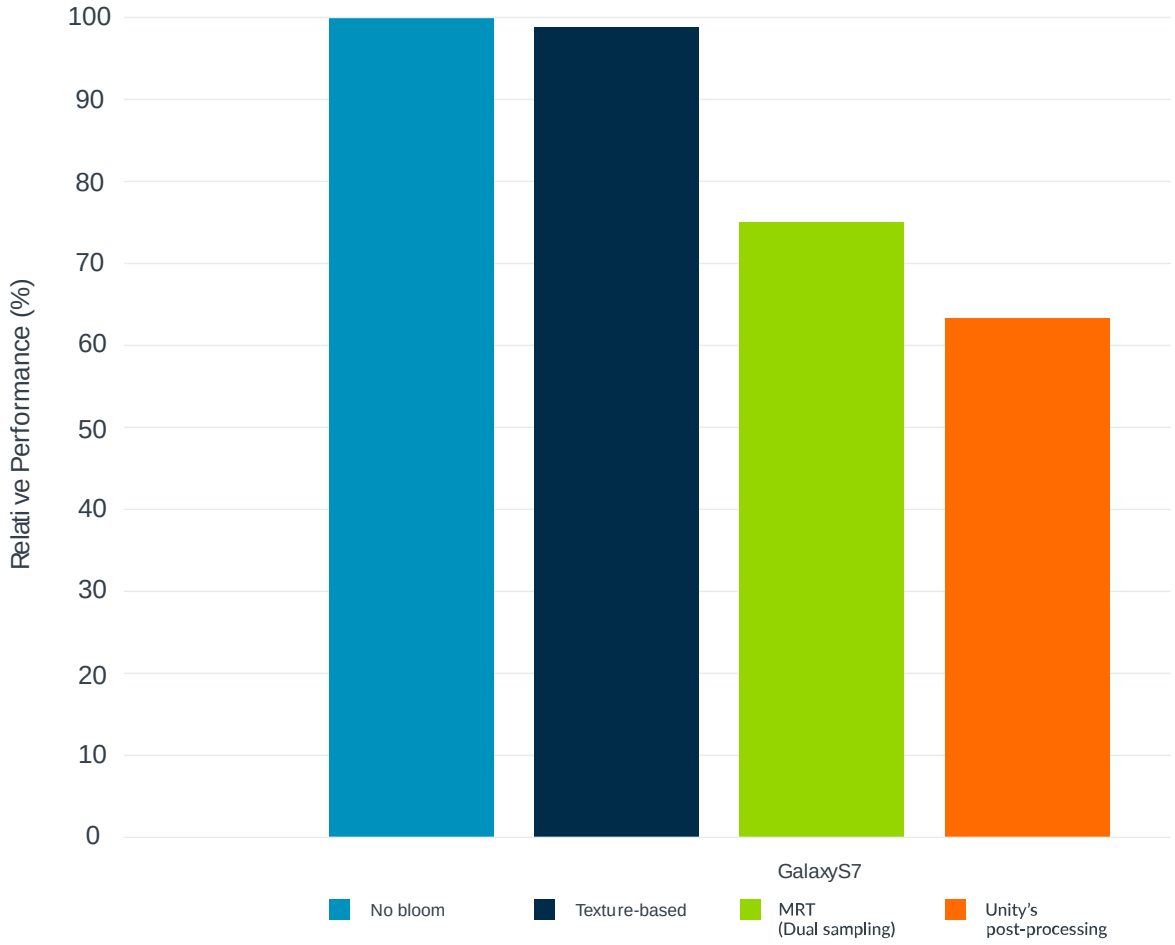
After the blur is calculated, that the bloom can be added to the rest of the scene, producing the final look. This is shown in the following image:

Figure 6-14: Spellsouls terrain with bloom applied



The following graph shows that efficient post-processing comes with significant cost however, adding in a texture to bake in the bloom costs virtually nothing:

Figure 6-15: Performance comparison between different bloom methods



7. Icy wall effect

This section shows you how to implement the icy wall effect in your own Unity programs.

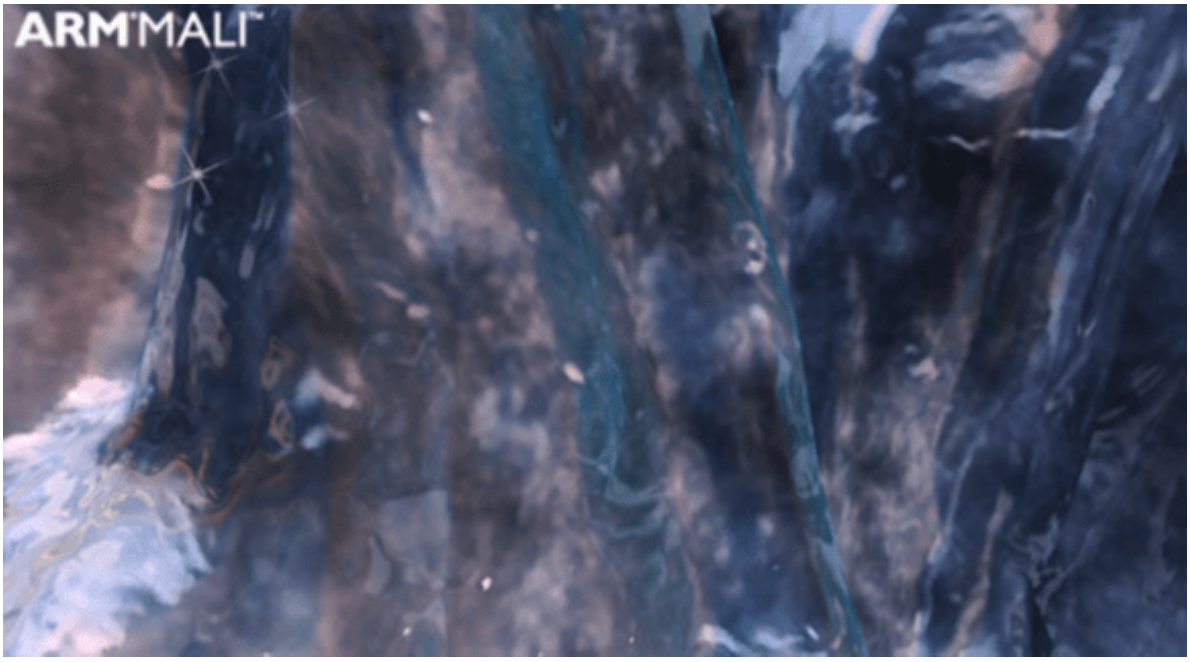
The Ice Cave demo uses a reflection effect in the icy walls of the cave. This effect is subtle but adds extra realism and atmosphere to the scene.

Ice can be a difficult material to replicate because light scatters off it in different ways, depending on the small details of its surface. The reflection can be completely clear, completely distorted, or anywhere in between. The Ice Cave demo shows this effect and includes a parallax effect for greater realism. The following image shows the Ice Cave demo, demonstrating the icy wall effect:

Figure 7-1: The Ice Cave demo with the icy wall effect

The following image shows a close up of this effect:

Figure 7-2: Close up of the reflective icy walls



Modifying and combining normal maps to affect reflections

The reflection effect in the Ice Cave demo uses the tangent space normal maps and calculated gray scale fake normal maps. Combining both maps with some modifiers creates the effect in the demo.

The gray scale fake normal maps are a gray scale of the tangent space normal maps. In the Ice Cave demo, most values in the gray scale maps are in the range 0.3-0.8. The following image shows the tangent space normal maps and gray scale fake normal maps that the Ice Cave demo uses:

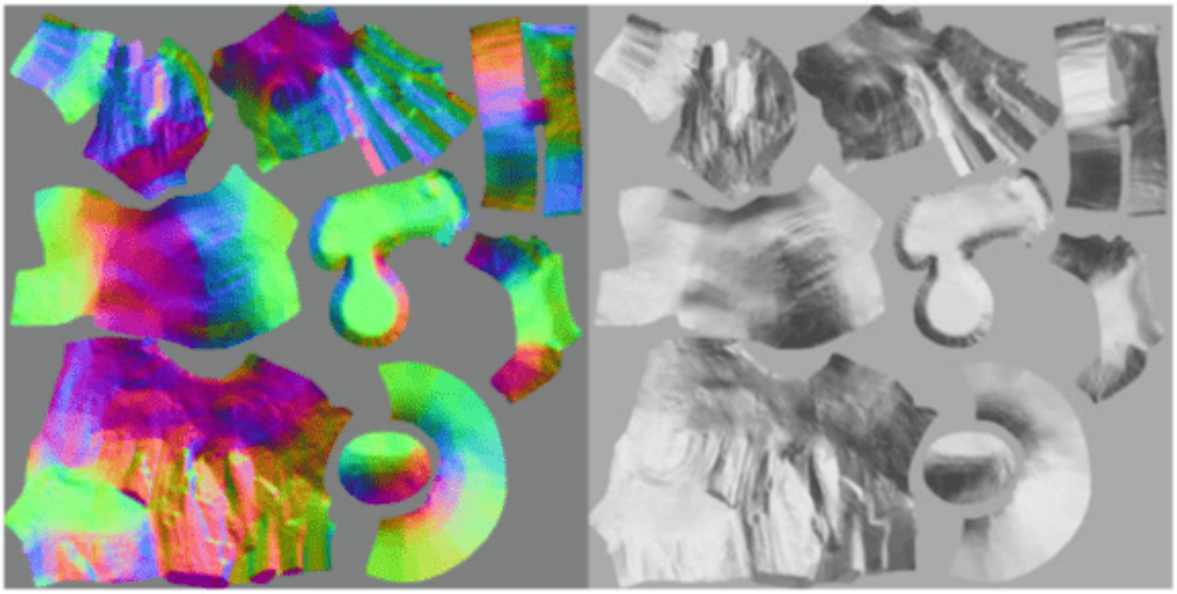
Figure 7-3: Tangent sapce normal maps and greyscale fake normal maps

Why tangent space normal maps are used?

The tangent space normal maps are used in the Ice Cave demo, because they have a small range of values in the resulting gray scale. Therefore, the tangent space normal maps work well in the later stages of this rendering process.

An alternative to tangent space normal maps is to use object space normal maps. These maps show the same details that the tangent space normal maps show, but also show where light hits. Therefore, the range of values in the resulting object space normal map gray scale is too large to work well in the later stages of the rendering process. The following image shows this effect:

Figure 7-4: Object space normal maps and greyscale effect

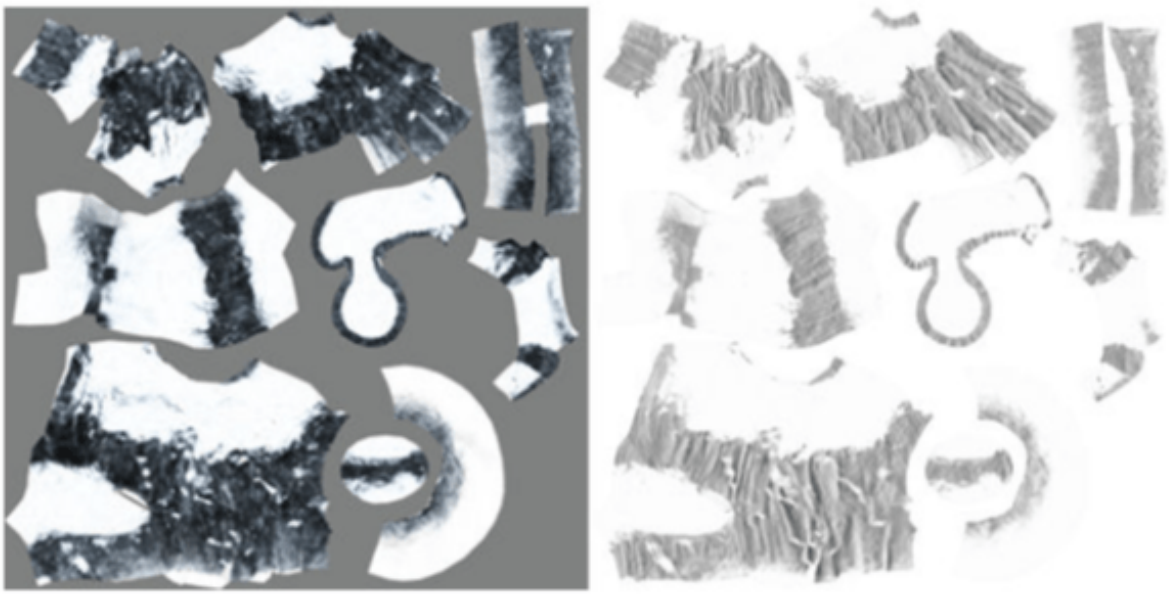


Applying transparency to parts of the normal maps

The gray scale fake normal maps are only applied to the areas without snow. To prevent them from being applied to the snowy areas, the gray scale fake normal maps are modified using the diffuse texture maps for the same surfaces. This modification increases the alpha component of the gray scale fake normal maps where snow appears in the texture maps.

The following image shows the diffuse texture maps for the Ice Cave demo static surfaces and the result when they are applied to the gray scale fake normal maps:

Figure 7-5: Diffuse texture and final fake normal maps with transparent areas



Creating the reflections from the different normal maps

The combination of the transparency-adjusted gray scale fake normal maps and the true normal maps creates the reflection effects that are shown in the Ice Cave demo.

The transparency-adjusted gray scale fake normal maps, `bumpFake`, and the true normal maps, `bumpNorm`, are combined proportionally. The combination uses the following function:

```
half4 bumpNormalFake = lerp(bumpNorm, bumpFake, amountFakeNormalMap);
```

This code means that in the darker parts of the cave, the main reflection contribution comes from the gray scale fake normals. In the snowy parts of the cave, the effect comes from the object space normals. To apply the effect using the gray scale fake normal maps, the gray scale must be converted into normal vectors. To start this process, the three components of the normal vectors are set equal to the gray scale value. In the Ice Cave demo, the components vectors are between (0.3, 0.3, 0.3) and (0.8, 0.8, 0.8). This is because all the components are set so that they are the same and all the normal vectors point in the same direction.

The shader applies a transformation to normal components. This uses the transform that is usually used to transform values in the range 0-1 to the range -1 to 1. The equation which performs this change is the following:

```
result = 2 * value - 1.
```

This equation changes the normal vectors so that they either point in the direction they pointed before, or the opposite direction. For example, if the original has the components (0.3, 0.3, 0.3), this means that the resulting normal is (-0.4, -0.4, -0.4). If the original has the components (0.8, 0.8, 0.8), this means that the resulting normal is (0.6, 0.6, 0.6).

After the transformation to the range -1 to 1, the vectors are fed to the `reflect()` function. This function is designed to work using normalized normal vectors. However, in this case the non-normalized normal is passed to the function. The following code shows how the shader built-in function `reflect()` works:

```
R = reflect(I, N) = I - 2 * dot(I, N) * N
```

Using this function with a non-normalized input normal with a length less than one, causes the reflection vector to deviate more than expected from the normal. This is because of what the reflection law states.

When the value of the components of the normal vector goes below 0.5, the reflection vector switches to the opposite direction. When the switch takes place the reflection vector switches to the opposite direction, a different part of the cubemap is read. This switching between different parts of the cubemap creates the effect of uneven spots. The uneven spots reflect the white parts of the cubemap next to the reflections of the rocky parts of the cubemap. This is because the areas in the gray scale fake normal maps that cause the switches between positive and negative normal are also the areas that produce the most distorted angles on the reflected vectors. This creates an appealing swirl effect. The following image shows this effect:

Figure 7-6: Swirl effect in the ice reflection

If the shader is programmed to use the fake normal values without the non-normalized clamped stage, the result features diagonal bands. The result featuring diagonal bands is a significant difference and shows that these stages are important. The following image shows the resulting effect without the clamp stage:

Figure 7-7: Ice reflection without the clamp stage

Applying local correction to the reflections

Applying local correction to the reflection vector improves the realism of the effect. Without this correction, the reflection does not change with the position of the camera like reflections in real

life do. This is especially true for lateral movements of the camera. For more information, see Generating correct reflections with a local cubemap.

8. Procedural skybox

This section of the guide describes the procedural skybox and how you can use it in your own Unity programs.

The Ice Cave demo uses a time-of-day system to show the dynamic shadowing effects that you can achieve with local cubemaps.

To achieve a dynamic time of day effect, combine the following elements:

- A procedurally generated sun
- A series of fading skybox background cubemaps that represent the day to night cycle
- A skybox clouds cubemap

The procedural sun and the separated cloud texture also create a computationally cheap subsurface scattering effect.

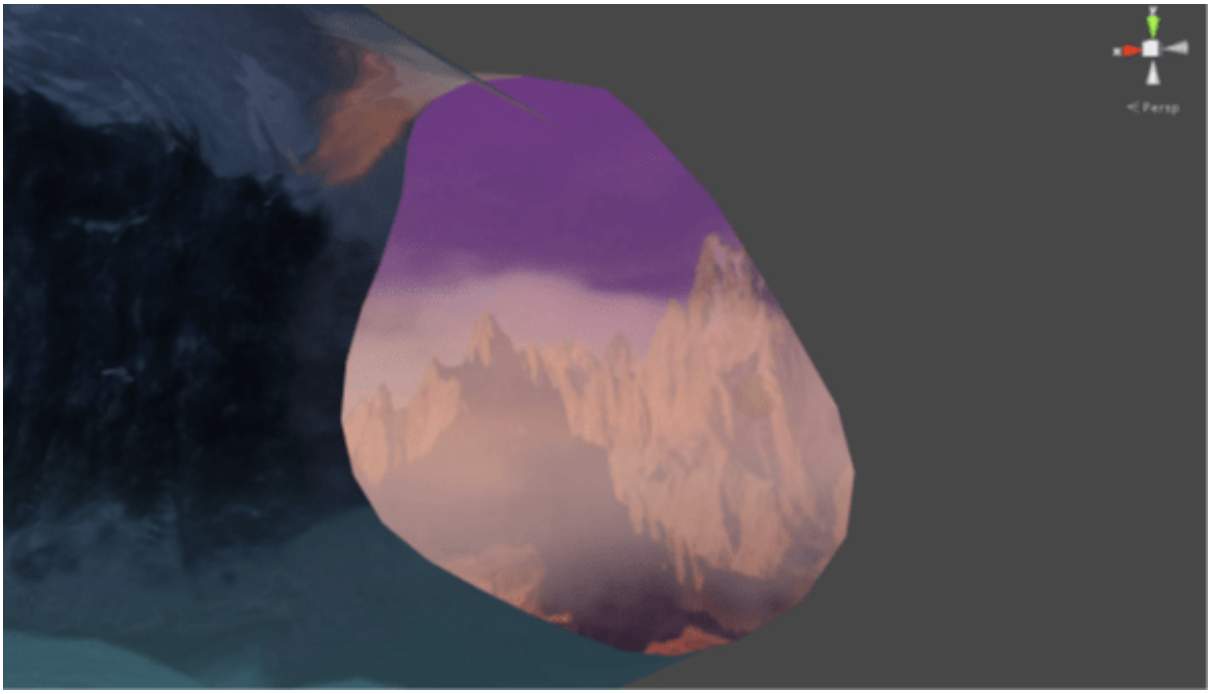
The following image shows a view of the skybox:

Figure 8-1: Sun rendering with subsurface scattering in the Ice cave demo

The Ice Cave demo uses the view direction to sample from a cubemap. This enables the demo to avoid rendering a hemisphere for the skybox. Instead, the demo just uses planes near the holes in the cave. Therefore, the performance increases, when compared to rendering a hemisphere that is then mostly occluded by other geometry.

The following image shows the skybox rendered with the use of a plane:

Figure 8-2: Skybox rendered that uses a plane



Managing the time of day

This effect uses a C# script to manage the mathematics for the time of day, and the animation of the day-night cycle. A shader then combines the sun and skymaps.

You must specify the following values for the script:

- The number of fading skybox background phases
- The maximum duration of the day to night cycle

For each frame, the script selects the skybox cubemaps to blend. The selected skyboxes are set as textures for the shader and the script blends the textures together while the frame is rendering.

To set the textures, the Ice Cave demo uses the Unity Shader global features. This enables you to set a texture in one place that all the shaders in your application can use. This is shown in the following code:

```
Shader.SetGlobalTexture (ShaderCubemap1, _phasesCubemaps [idx1]);
Shader.SetGlobalTexture (ShaderCubemap2, _phasesCubemaps [idx2]);
Shader.SetGlobalFloat (ShaderAlpha, blendAlpha);
Shader.SetGlobalVector (ShaderSunPosition, normalizedSunPosition);
Shader.SetGlobalVector (ShaderSunParameters, _sunParameters);
Shader.SetGlobalVector (ShaderSunColor, _sunColor);
Shader.SetGlobalTexture (ShaderCloudsCubemap, _CloudsCubemap);
```

The name of the sampler used must not conflict with the one that is locally defined by the shader.

The following items are set in the script code:

- The two cubemaps that are interpolated.
- The values `idx1` and `idx2` are computed based on the elapsed time.
- A `blendAlpha` factor, that is used in the shader to blend the two cubemaps.
- A normalized sun position used to render the sun sphere.
- Parameters for the sun
- The color of the sun
- The cloud cubemap. Specifically, in the code these are: `ShaderCubemap1` and `ShaderCubemap2` which are two strings that contain the unique sampler names, in this case `_SkyboxCubemap1` and `_SkyboxCubemap2`.

To access these textures in the shader, you must declare them with the following code:

```
samplerCUBE _SkyboxCubemap1;
samplerCUBE _SkyboxCubemap2;
```

The script selects the sun color and an ambient color based on the list that you specify. These colors are interpolated for each phase.

The sun color is passed to the shader for rendering the sun with the correct color.

The ambient color is used to dynamically set the Unity variable `RenderSettings.ambientSkyColor`. This is shown in the following code:

```
RenderSettings.ambientSkyColor = _ambientColor;
```

Setting this variable enables all the materials to receive the correct ambient color. In the Ice Cave demo, this effect causes a gradual variation of the overall color of the scene based on the phase of the day it is.

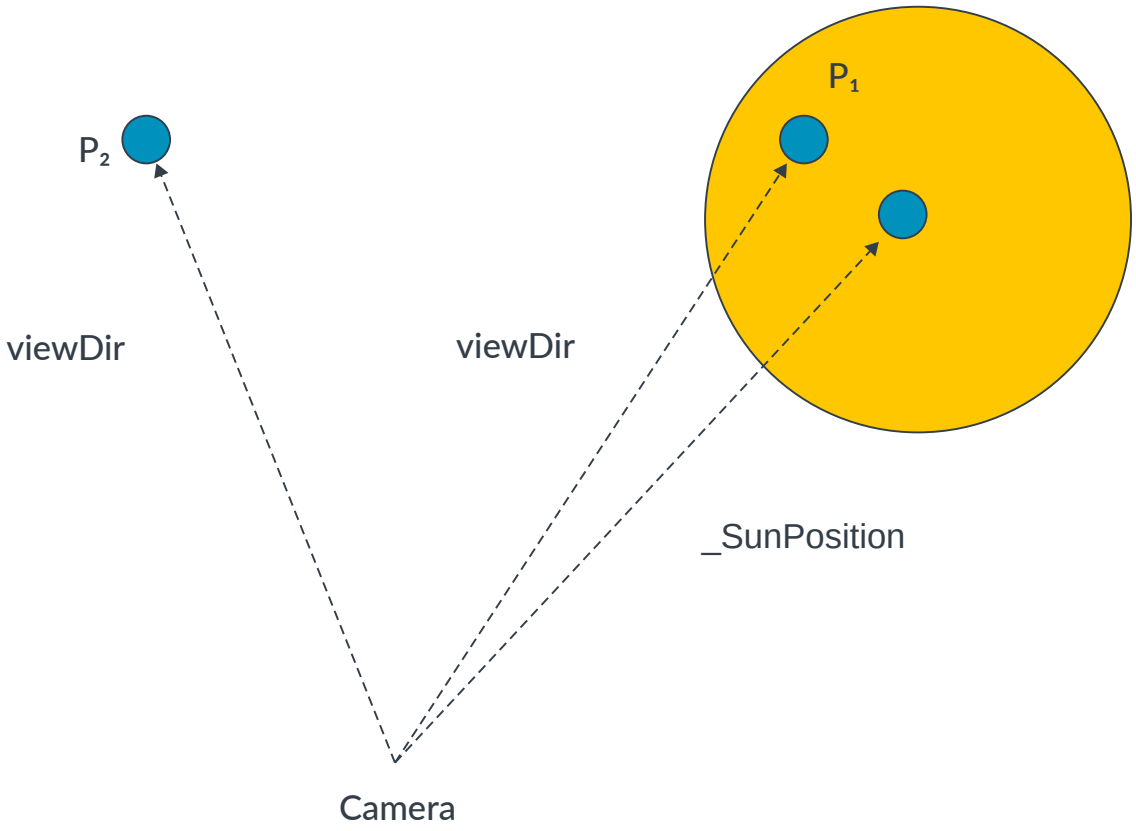
Rendering the sun

To render the sun, in the fragment shader, for each pixel of the skybox you must check whether it is inside the circumference of the sun.

To check whether a pixel is inside the circumference of the sun, the shader computes the dot product of the normalized sun position vector and the normalized view direction vector in world coordinates of the pixel being rendered.

The following diagram shows the rendering of the sun:

Figure 8-3: Rendering the sun



Note: The dot product of the normalized view vector and the sun position vector is used to determine if the fragment is rendered, as sky as P2, or rendered as a sun as P1.

The normalized sun position vector is passed to the shader by the C# script, if either of the following occur:

- The result is greater than a specific threshold. The pixel is colored with the sun color.
- The result is less than the threshold. The pixel is colored with sky color.

The dot product also creates a fading effect towards the edges of the sun. This is shown in the following code:

```
half _sunContribution = dot(viewDir,_SunPosition);
```

The following image shows a clear view of the sun:

Figure 8-4: Procedural sun rendering in clear sky conditions

Fading the sun behind mountains

There is a problem if the sun is low in the sky, because in real life it would disappear behind the mountains.

To create this effect, the alpha channel of the cubemap stores a value of 0 if the texel represents the sky. This channel stores a value of 1 if the texel represents the mountain.

While rendering the sun, the texture is sampled, and the alpha is used to make the sun fade behind the mountains. This sampling is effectively free. This is because the texture is already sampled to render the mountains.

You can also gradually fade the alpha near the edges or the areas where there is snow on the mountain. This produces an effect of the sun bouncing off the snow for little computational effort.

A similar technique is used to create a computationally cheap subsurface scattering effect for the clouds.

The original phase cubemaps are split into the following groups:

- One set of cubemaps contain the sky and the mountains. The alpha value is set to 0 for the sky and set to 1 for the mountains.
- The other set of cubemaps contains the clouds. The alpha value is set to 0 in the areas without clouds and gradually increases to 1 in areas where the clouds are denser.

The following image shows a mountains texture:

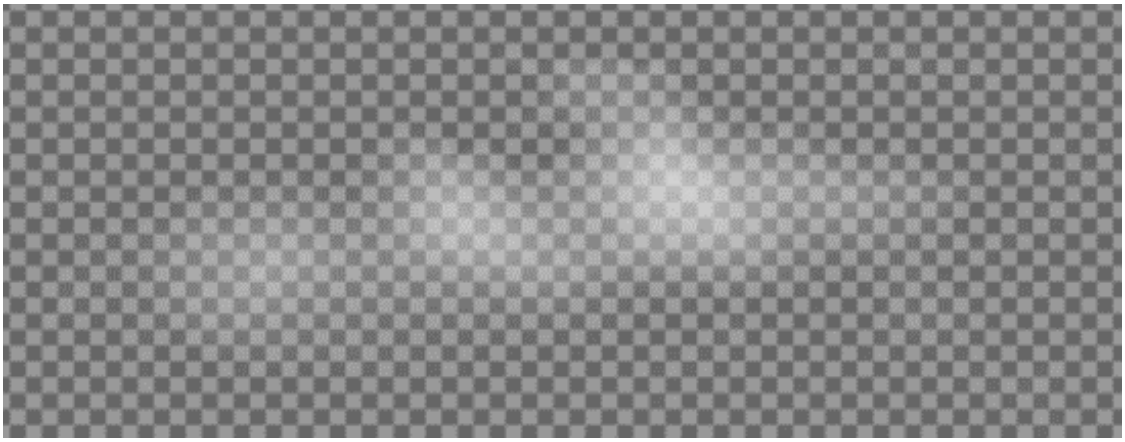
Figure 8-5: Mountains textureg

Note

The mountains have an alpha value of one, the sky has a value of zero.

The clouds skybox has alpha 0 for the empty areas and gradually fades to 1 when occupied by a cloud. The alpha channel fades smoothly to ensure that the clouds do not look like they have been artificially put on top of the sky. The following image shows a cloud texture with alpha:

Figure 8-6: Cloud texture with alpha



The shader performs the following steps:

1. Samples the two skyboxes that represent the current time of day phase.
2. Blends the two colors based on a blending factor that is calculated by the C# script.
3. Samples the cloud skybox.
4. Blends the color from point 2 with the color from the clouds using the cloud alpha.
5. Adds the cloud alpha and the skybox alpha.
6. Calls a function to compute the sun color contribution for the current pixel.
7. Adds the result of point 6 with the previously blended skybox and cloud color.


Note

You can optimize this sequence by putting the sky, mountains, and clouds into a single skybox. The sky, mountains, and clouds are separate in the Ice Cave demo, so that the artist can easily modify the skyboxes and the clouds separately.

Subsurface scattering

You can add a subsurface scattering effect that uses the alpha information to increase the radius of the sun.

In real life when the sun is in a clear sky, it looks small. This is because there are no clouds that can deviate, or scatter, the light toward your eyes. What you see are the rays directly from the sun, spread only very slightly by the atmosphere.

When the sun is occluded by clouds, but they do not completely obscure it, part of the light is scattered around the cloud. This light can reach your eyes from directions that are some distance away from the sun, making the sun appear larger than it really is.

The following code is used to achieve this effect in the Ice Cave demo:

```
half4 sampleSun(half3 viewDir, half alpha);
```

```
{
        half _sunContribution = dot(viewDir,_SunPosition);
        half _sunDistanceFade = smoothstep(_SunParameters.z -(0.025*alpha), 1.0,
 _sunContribution);
        half _sunOcclusionFade = clamp( 0.9-alpha, 0.0, 1.0);
        half3 _sunColorResult = _sunDistanceFade * _SunColor * _sunOcclusionFade;
        return half4( _sunColorResult.xyz, 1.0 );
}
```

Here is an explanation of the preceding code:

- The parameters of the function are `viewDirection` and an alpha value that is computed by the cloud alpha and skybox alpha being added together.
- The dot products of the sun position and view directions are used to compute a scaling factor. The scaling factor represents the distance of the current pixels from center of the sun.
- The `_sunDistanceFade` calculation uses the `smoothstep()` function to provide a more gradual fade from the center of the sun to the sky near the edges.
- This function has a variable domain based on the alpha, if there is clear sky, the alpha is 0 and the range is within `_SunParameters.z` and `1.0`. In this case `_SunParameters.z` is initialized to `0.995` in the C# script, `_SunParameters.z` corresponds to a sun of diameter of 5 degrees: `cos(5 degrees) = 0.995`
- If the pixel that is being processed contains a cloud, the radius of the sun is increased to 13 degrees. This enables an elongated scattering effect when approaching the clouds.
- The `_sunOcclusionFade` factor is used to fade down the contribution of the sun based on the occlusion received from the mountains and the clouds.

The following image shows the sun when it is not occluded by the clouds:

Figure 8-7: Sun not occluded by clouds

The following image shows the sun when it is occluded by clouds:

Figure 8-8: Sun occluded by clouds

9. Fireflies

This section of the guide shows you how to implement fireflies into your Unity programs. Fireflies are bright flying insects that are used in the Ice Cave demo to add more dynamism.

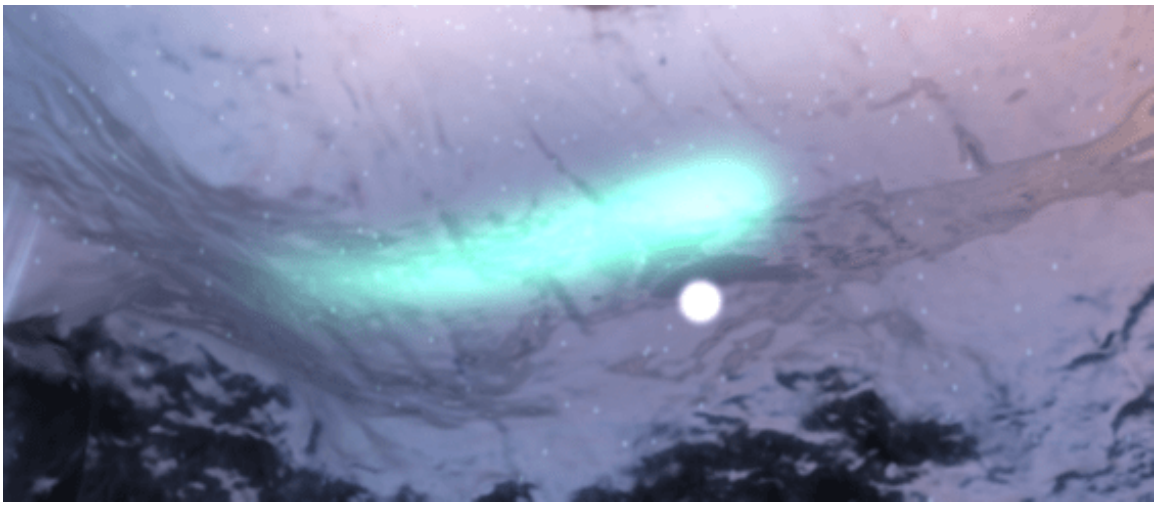
The fireflies are composed of the following components:

- A prefab object that is instantiated at runtime
- A box collider that is used to limit the area the fireflies can fly around in

These two components are combined in a C# script that manages the movement of the fireflies and defines the path that they follow.

The following image shows a firefly:

Figure 9-1: Firefly



The firefly prefab uses the Unity standard particle system to generate the trail of the firefly.

The following image shows the Unity Particle System settings that the fireflies use:

Figure 9-2: Unity Particle System settings the fireflies use

Depending on the effect that you want to achieve, you can use a Unity Trail Renderer to provide a more continuous look.

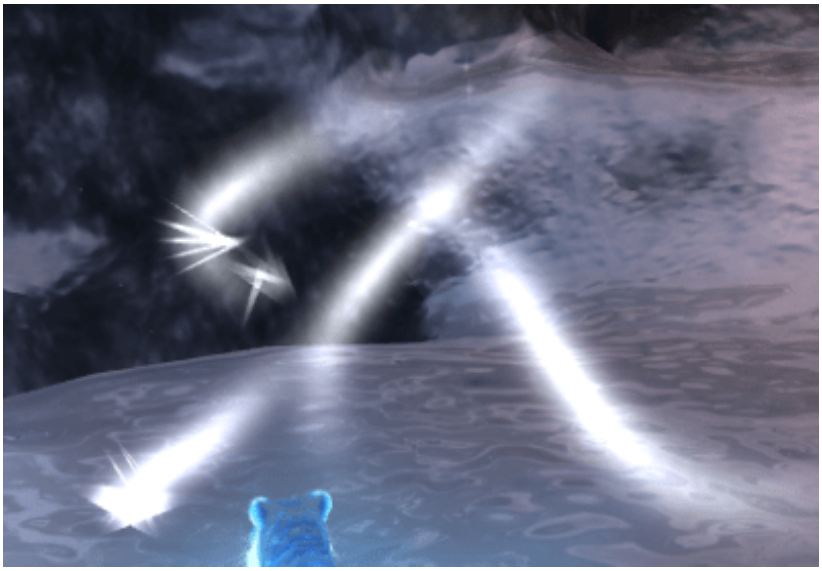
The Trail Renderer generates many triangles for each trail. You can change the number of triangles by modifying the Trail Renderer setting Min Vertex Distance. However, if the source is moved too fast, a high value can cause a jerky movement of the trail.

The Min Vertex Distance option defines the minimum distance between the vertices that form the trail. A high number is okay for straight trails, but it does not look smooth for curved trails.

The trail that is generated always faces the camera. Therefore, any abrupt movement of the source can cause the trail to overlap with itself. This causes artifacts that are generated by the blending of the triangles that form the trail.

The following image shows artifacts caused by an overlapping trail:

Figure 9-3: Overlapping trail artifacts



The final component to add to the prefab is a point light that casts light in the scene while it is moving.

The firefly generator prefab

The firefly generator prefab manages the creation of the fireflies and updates them in each frame. The firefly generator prefab is composed of a C# script for the update, and a box collider that encloses the volume that each firefly can move in.

The script takes the number of fireflies to be generated as a parameter, and the prefab to instantiate the firefly object. This is because the movement of the fireflies is random within the bounding box, it limits changes to a specific angle, away from the direction that the firefly is moving. This ensures that the firefly does not make sudden changes of direction.

To generate a random movement, a piecewise cubic Hermite interpolation is used to create the control points. The Hermite interpolation provides a smooth continuous function that behaves correctly even when different paths are connected. The first derivative at the end points are also continuous, so that there are no sudden velocity changes.

This interpolation requires one control point each for the start and the end, and two tangents for each of the control points. This is because the control point and the tangents are generated randomly, the script stores three control points and two tangents. The interpolation uses the position of the first and second control points to define the first point tangent. The second and third control points are used to define the tangent of the second control point.

At loading time, the script generates the following for each firefly:

- An initial position
- An initial direction that uses the Unity function `Random.onUnitSphere`

The following code shows how the control points are initialized:

```
_fireflySpline[i*_controlPoints] = initialPosition;
Vector3 randDirection = Random.onUnitSphere;
_fireflySpline[i*_controlPoints+1] = initialPosition + randDirection;
_fireflySpline[i*_controlPoints+2] = initialPosition + randDirection * 2.0f;
```

The initial control points lie on a straight line. The tangents are generated from the following control points:

```
//The tangent for the first point is in the same direction as the initial direction
 vector
_fireflyTangents[i*_controlPoints] = randDirection;
//This code computes the tangent from the control point positions. It is shown here
 for
// reference because it can be set to randDirection at initialization.
_fireflyTangents[i*_controlPoints+1] = (_fireflySpline[i*_controlPoints+2] -
_fireflySpline[i*_controlPoints+1])/2 + (_fireflySpline[i*_controlPoints+1] -
_fireflySpline[i*_controlPoints])/2;
```

To complete a firefly initialization, you must set the color of the firefly and the duration of the current path interval.

On each frame, the script updates the position of each firefly that uses the following code to compute the Hermite interpolation:

```
// t is the parameter that defines where in the curve the firefly is placed. It
 represents
// the ratio of the time the firefly has traveled along the path to the total time.
float t = _fireflyLifetime[i].y / _fireflyLifetime[i].x;
//Hermite interpolation parameters
Vector3 A = _fireflySpline[i*_controlPoints];
Vector3 B = _fireflySpline[i*_controlPoints+1];
float h00 = 2*Mathf.Pow(t,3) - 3*Mathf.Pow(t,2) + 1;
float h10 = Mathf.Pow(t,3) - 2*Mathf.Pow(t,2) + t;
```

```
float h01 = -2*Mathf.Pow(t,3) + 3*Mathf.Pow(t,2);
float h11 = Mathf.Pow(t,3) - Mathf.Pow(t,2);
//Firefly updated position
_fireflyObjects[i].transform.position = h00 * A + h10
 * _fireflyTangents[i*_controlPoints] + h01 * B + h11 *
  _fireflyTangents[i*_controlPoints+1];
```

If the firefly completed the whole piece of randomly generated path, the script creates a new random piece that starts from the end of the current one. This is shown in the following code:

```
// t > 1.0 indicates the end of the current path
if (t >= 1.0)
{
        //Update the new position
        //Shift the second point to the first as well as the tangent
        _fireflySpline[i*_controlPoints] = _fireflySpline[i*_controlPoints+1];
        _fireflyTangents[i*_controlPoints] = _fireflyTangents[i*_controlPoints+1];
        //Shift the third point to the second, this point doesn't have a tangent
        _fireflySpline[i*_controlPoints+1] = _fireflySpline[i*_controlPoints+2];
        //Get new random control point within a certain angle from the current fly
 direction
        _fireflySpline[i*_controlPoints+2] = GetNewRandomControlPoint();
        //Compute the tangent for the central point
        _fireflyTangents[i*_controlPoints+1] = (_fireflySpline[i*_controlPoints+2] -
        _fireflySpline[i*_controlPoints+1])/2 + (_fireflySpline[i*_controlPoints+1]
 -
         _fireflySpline[i*_controlPoints])/2;
        //Set how long should take to navigate this part of path
        _fireflyLifetime[i].x = _fireflyMinLifetime;
        //Timer used to check how much we traveled along the path
        _fireflyLifetime[i].y = 0.0f;
}
```

10. Related Information

Here are some resources related to material in this guide:

- Arm architecture and reference manuals
- Arm based demos made with Unity
- Arm Community - Ask development questions and find articles and blogs on specific topics from Arm experts.
- Arm Guide for Unity developers
- Advanced graphic techniques - Getting started
- Local cubemap rendering
- Virtual Reality: The Ice Cave demo

11. Next steps

This guide has introduced you to some special effects that you can implement in your Unity programs. Some of the special effects that we have looked at include the dirty lens effect, fog effects, and icy wall effect.

After reading this guide, you will be ready to implement some of the techniques into your own Unity programs. If you want to learn more about Unity, you can read our Arm guides for Unity developers.