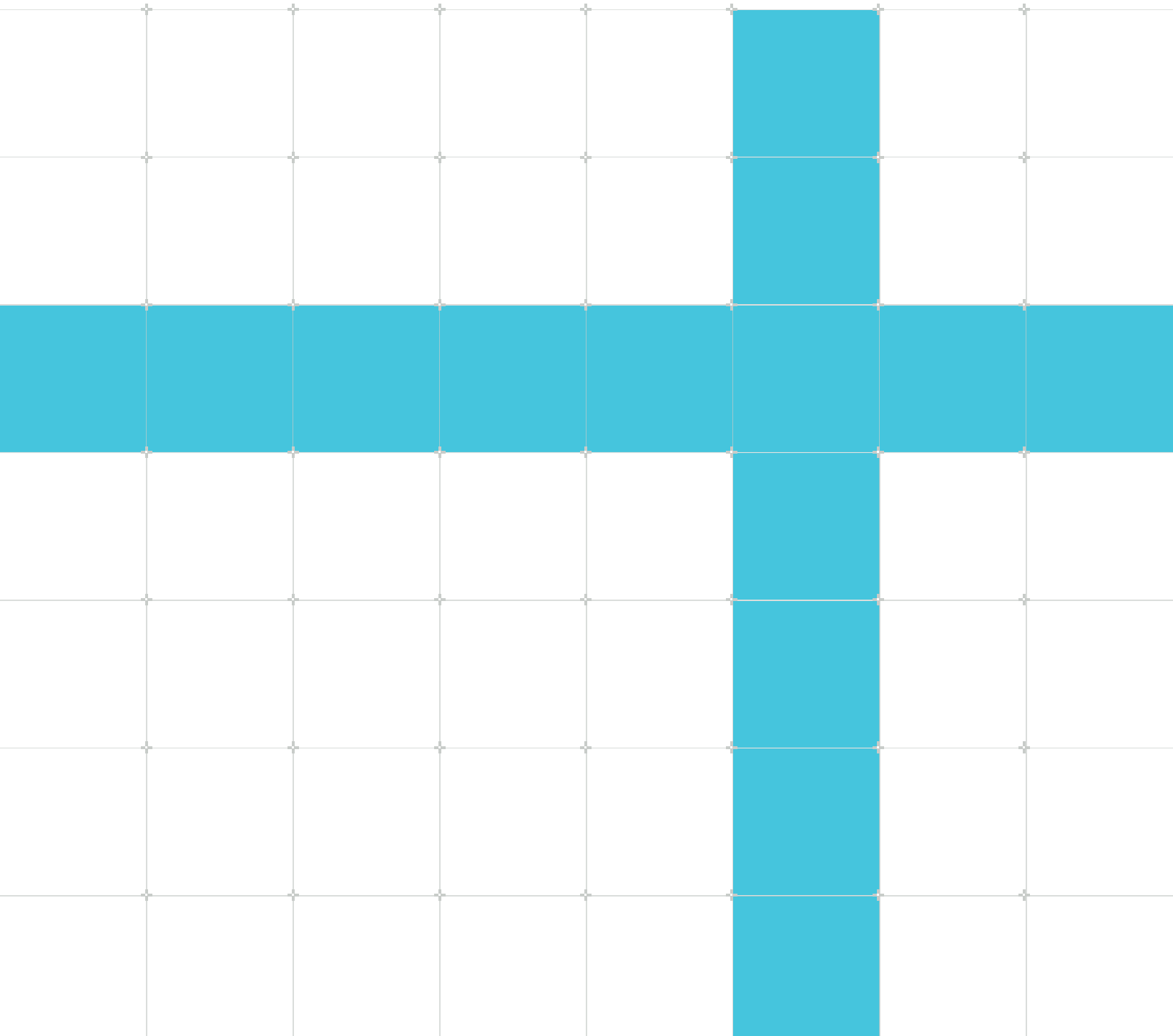
arm

Advanced VR Graphics Techniques

Version 1.0

Issue 02
102073_0100_02_en



Advanced VR Graphics Techniques

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	12 April 2022	Non-Confidential	First release

Proprietary Notice

or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

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 describes various techniques that you can use to improve the graphical performance of your Virtual Reality application.

At the end of this guide, you can Check your knowledge. You will have learned about topics such as aliasing, mipmapping, and bump mapping.

Arm Guide for Unity Developers: Korean

Arm Guide for Unity Developers: Chinese

Arm Guide for Unity Developers: Japanese

2. Aliasing

Aliasing occurs when you try to capture information without using enough samples to faithfully recreate the object. For most practical uses, aliasing is an aspect of signal processing that cannot be ignored.

In audio and video, you get aliasing as you sample the raw data of sound or color to turn it into a discrete digital signal. Aliasing is more obvious at higher audio frequencies and in finer details of a video.

Aliasing occurs when you sample the raw data too infrequently. The threshold for aliasing is when the sampling rate is less than half of the highest frequency in the content. For graphics, aliasing is most obvious in the rasterization process of displaying objects from a camera that is in motion and is going against the discrete grid of pixels which form the display.

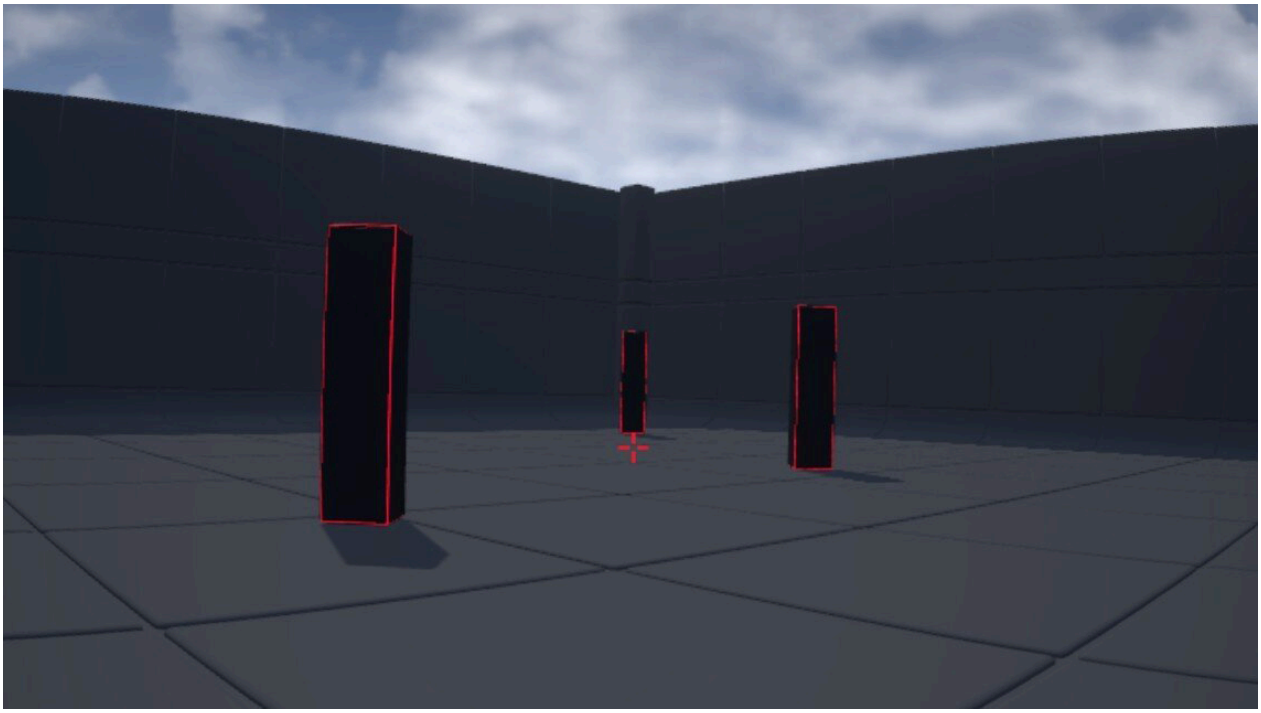
In a VR environment, there are two specific aliasing variants which are especially apparent:

- Geometric aliasing
- Specular aliasing

Geometric aliasing is most visible when a scene contains high-frequency input signals in color space that are displayed on the relatively low frequency of pixel samples, for example, a rapid transition between two contrasting colors.

This aliasing is especially visible in straight lines that appear to crawl when moving the camera. The following image shows an example of this geometric aliasing on the red edges of the shapes:

Figure 2-1: Example of geometric aliasing

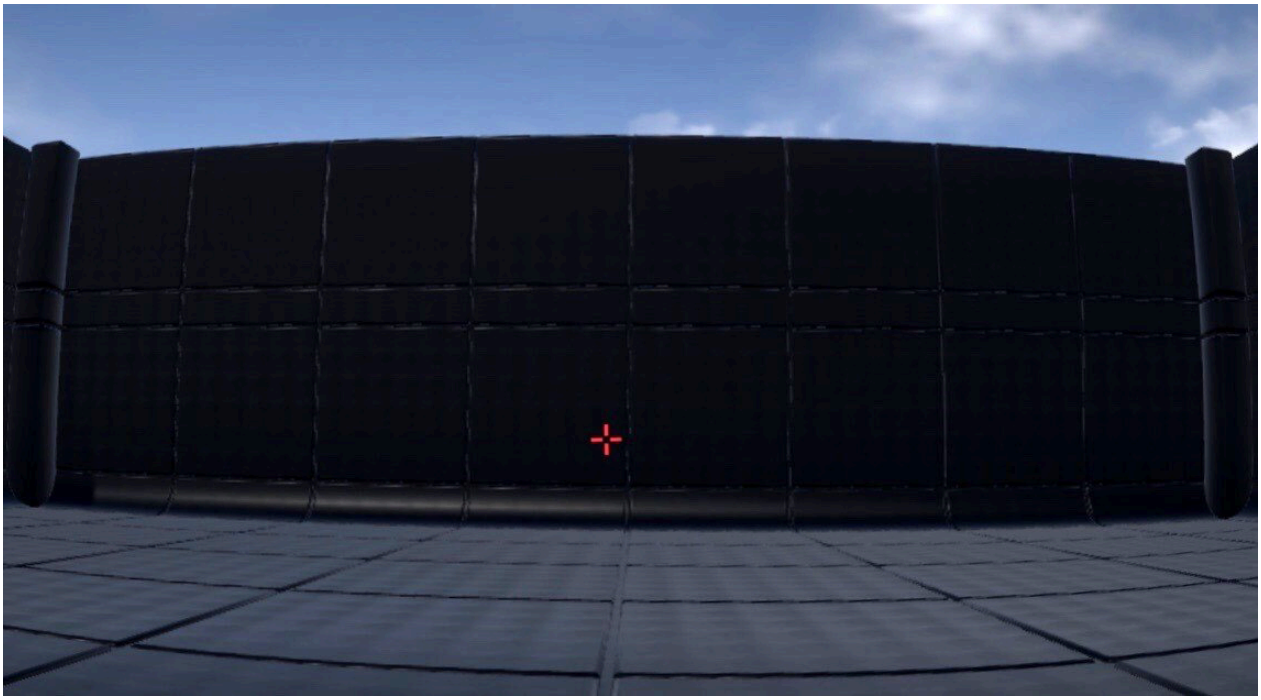


Geometric aliasing can also occur when meshes with high polygon counts are rendered onto a low-resolution display, resulting in too much detail due to the quantity of polygons with the width of a few pixels. Lots of polygons means lots of edges, and with few pixels to render them in, this results in more aliasing.

Also, a high density of triangles is inefficient for rasterizing in the GPU, with the GPU spending many cycles on geometry that is too small to make any significant difference to the final picture. Additionally, the excess cycles the GPU spends on rasterizing geometry means that there are fewer cycles left to execute fragment shaders. This is because the GPU has to spend many cycles rasterizing the geometry rather than on executing the fragment shaders.

Specular aliasing results from objects with sharp highlights which 'pop' in and out of existence as either the camera or the objects move. This manifests itself as pixels which appear to shimmer across frames as a result of the specular effect being present in one frame but not in the next. The shimmering effect is annoying for the user as it distracts their attention, impacting the sense of immersion and reducing the quality of the user experience. The ridges in the wall on the following image demonstrate specular aliasing.

Figure 2-2: Example of specular aliasing



Common anti-aliasing techniques such as supersampling are not feasible on mobile due to the computational complexity. While Multisample Anti-Aliasing (MSAA) is both effective and performant, it only acts on geometric aliasing. MSAA is therefore ineffective against under-sampling artifacts that occur within shaders such as specular aliasing.

Various techniques are required to mitigate each variation of aliasing, due to the many types of aliasing that occur within computer graphics, and more specifically VR.

3. Multisample Anti-Aliasing

Multisample Anti-Aliasing (MSAA) is an anti-aliasing technique that is a more efficient variation of supersampling. Supersampling renders the image at a higher internal resolution before downscaling to the display resolution by performing fragment shading for every pixel at that higher resolution.

MSAA performs the vertex shading normally but then each pixel is divided into subsamples which are tested using a subsample bitwise coverage mask. If any subsamples pass this coverage test, fragment shading is then performed, and the result of the fragment shader is stored in each subsample which passed the coverage test.

By only executing the fragment shader once per pixel, MSAA is substantially more efficient than supersampling although it only mitigates geometric aliasing at the intersection of two triangles.

For VR applications the quality benefits from 4x MSAA far outweigh the cost and it should be used whenever possible. In particular, 4x MSAA reduces the “jaggies” caused by additional fragments that are generated along the edges of triangles.

Mali GPUs are designed for full fragment throughput when using 4x MSAA so it can be used with only a minor effect on performance.

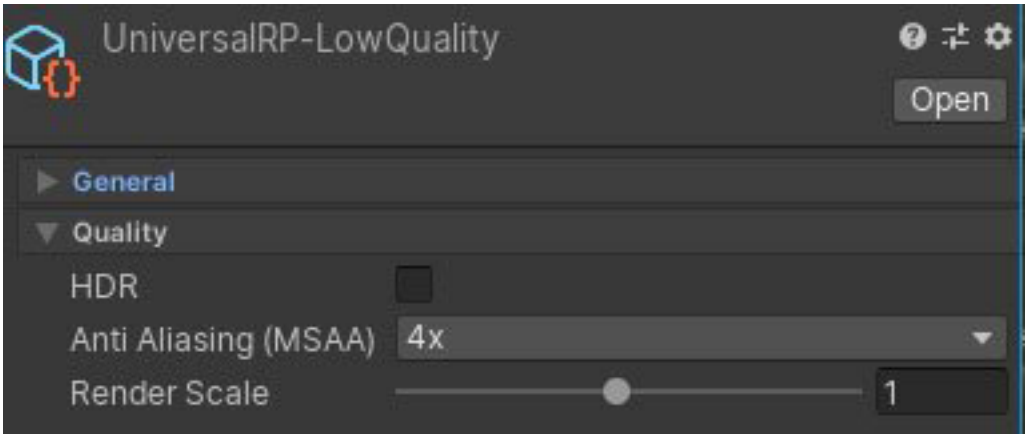
Implementing Multisample Anti-Aliasing in Unity

In Unity, Multisample Anti-Aliasing is set up in the Universal Render Pipeline (URP) settings if you are using URP. Otherwise, it is set up in the Project Quality Settings panel.

To enable MSAA by using the URP panel:

1. Go to the Assets window.
2. In the Inspector window, go into the Quality drop-down.
3. Choose the required MSAA level from the drop-down menu here.

Figure 3-1: URP MSAA settings panel

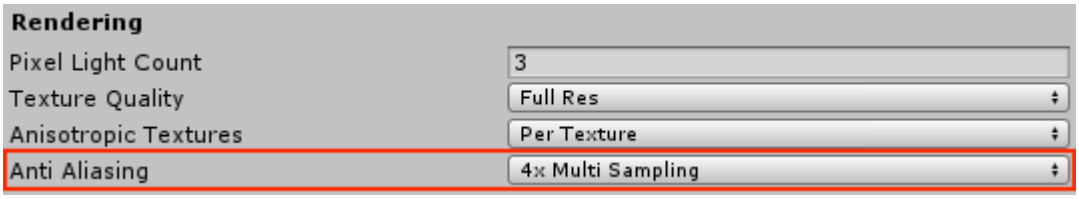


To enable MSAA by using the Project Quality Settings panel:

1. Go to Edit, then Project Settings.
2. Selecting Quality opens the Project Quality Settings panel.
3. Now choose the Anti Aliasing drop-down, and then choose the appropriate setting. Preferably 4x Multi Sampling where possible.

Be sure to set the anti-aliasing value for all quality levels.

Figure 3-2: msaa settings



4. Mipmapping

Mipmapping is a technique where a high-resolution texture is downscaled and filtered so that each subsequent mip level is a quarter of the area of the previous level. This means that the texture and all of its generated mips requires no more than 1.5 times the original texture size.

Mipmaps can either be hand-generated by an artist or computer-generated, and are uploaded to the GPU. The GPU then selects the optimal mip for the sampling being performed. Sampling from the smaller mip level helps to minimize texture aliasing, maintain the definition of textures on surfaces, and prevent the formation of moiré patterns on distant surfaces.

Figure 4-1: Nine mipmap levels

Caution must be taken when storing multiple successive mipmap levels within a single texture atlas, as visual problems can arise when foveated rendering is used.

Foveated rendering uses eye-tracking to display higher-quality graphics where the user is currently looking. Image quality can be greatly reduced outside the gaze of the fovea.

Problems arise when one tile is rendered at the native resolution, but the neighboring tile is rendered at one-quarter resolution. This means that the neighboring tile samples the texture from the mipmap two levels lower.

One resulting problem occurs during sampling. For example, when a texel is blurred with its neighboring texels during texture filtering and can wrongly bleed in color from the neighboring levels in the texture atlas.

As a result, two neighboring pixels which are in separate tiles, that lie in two differing foveated regions, exhibit a color differential. Extending the edge of the texture to create an isolation gap between entries into the atlas solves this problem.

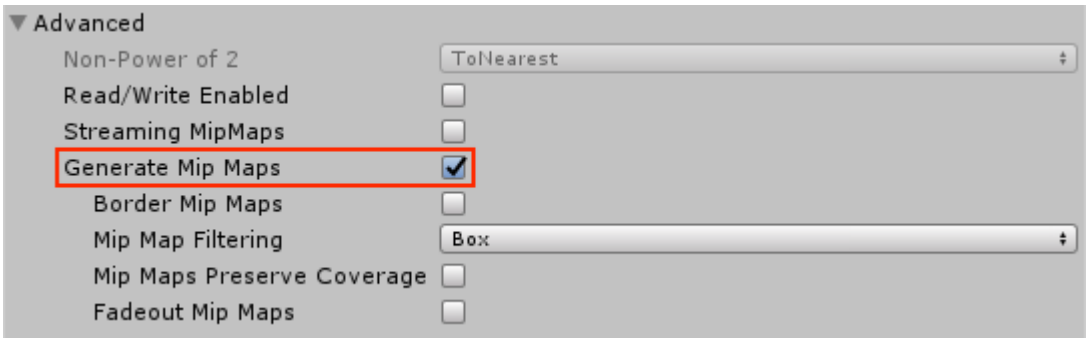
Implementing mipmapping in Unity

The automatic generation of mipmaps is turned on for textures in the Inspector window.

To enable mipmap generation:

1. Select a texture within the Assets section of the Project window to open the Texture Inspector window.
2. Enable the Generate Mip Maps option.

Figure 4-2: Generate mipmaps

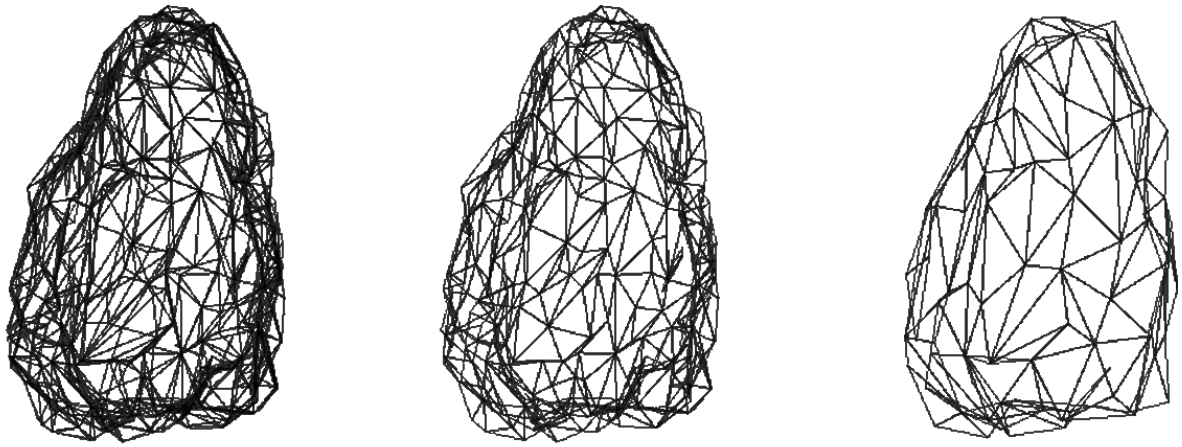


5. Level of Detail

Level of Detail (LOD) is the technique of decreasing the detail and complexity of an object as the distance increases between the object and the viewer.

By using LODs, when the viewer is close to an object, the object has a high level of geometric detail. Therefore, the object appears detailed and accurate. When an object moves further away from the screen it automatically switches to a lower detail model, showing a less detailed model to the player.

Figure 5-1: Three lods



There are two key benefits of changing to a model to one with a lower level of detail:

- A reduced level of artifacts due to the decreasing chances of geometric aliasing occurring
- The number of vertices that require shading are reduced, resulting in an increase to performance.

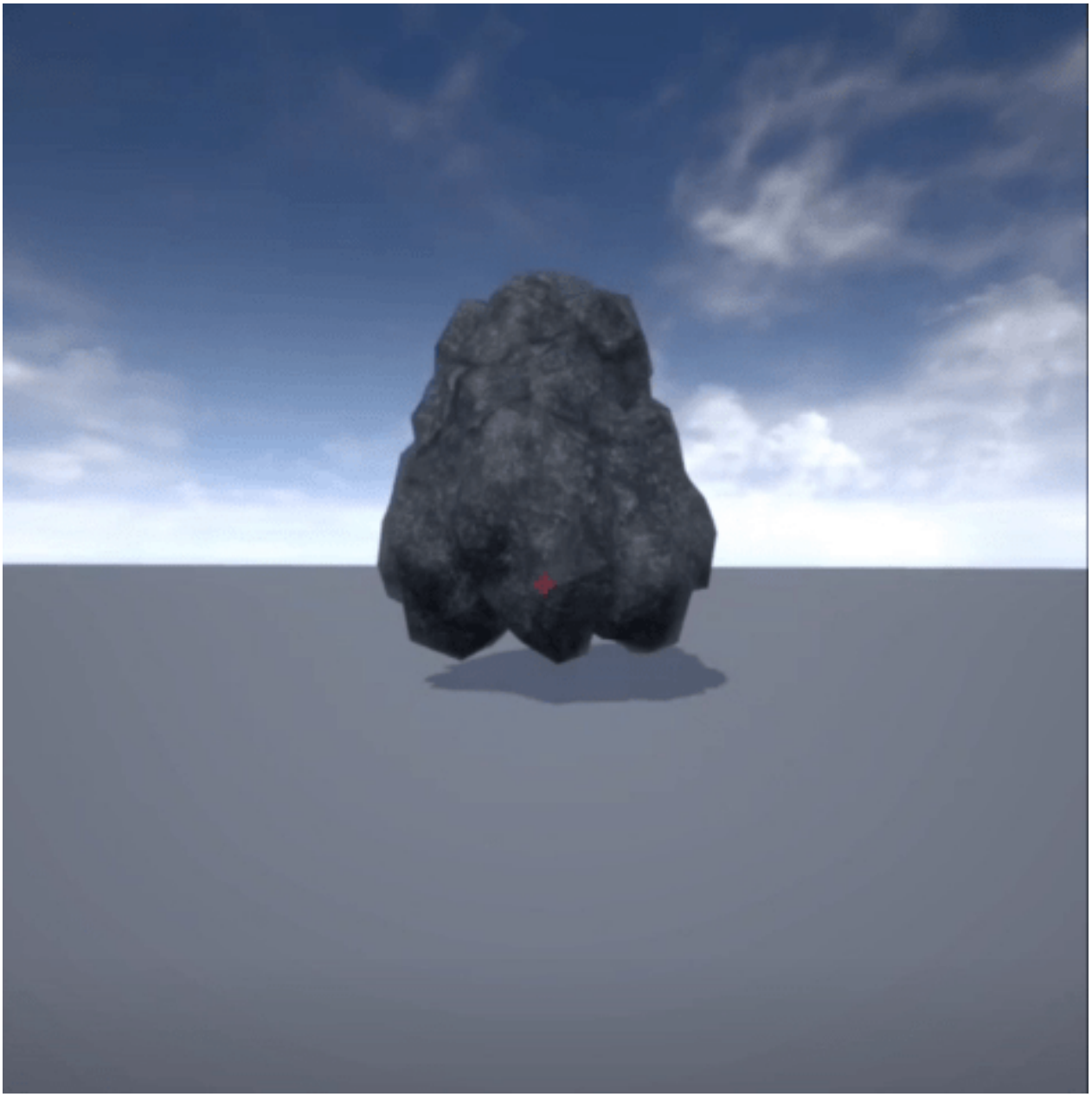
To prevent unnecessary oversampling from occurring, due to geometry that is tiny, LODs must be created with notable differences in the vertex count between each LOD level.

Further information on LODs can be found in the LOD section of the Real-Time 3D Art Best Practices: Geometry guide.

Visual examples demonstrating how LODs can look

The following image and video shows an example of how LODS work as the object gets closer or further away from the camera:

Figure 5-2: Final lod render



The following video shows an example of a mesh passing through three separate LODs as the camera moves closer to the object. The video shows that the finer details, that become apparent when the camera is closer to the object, are not visible from a distance. However, up close the finer details aid in improving the realism of the object.

Implementing Level of Detail in Unity

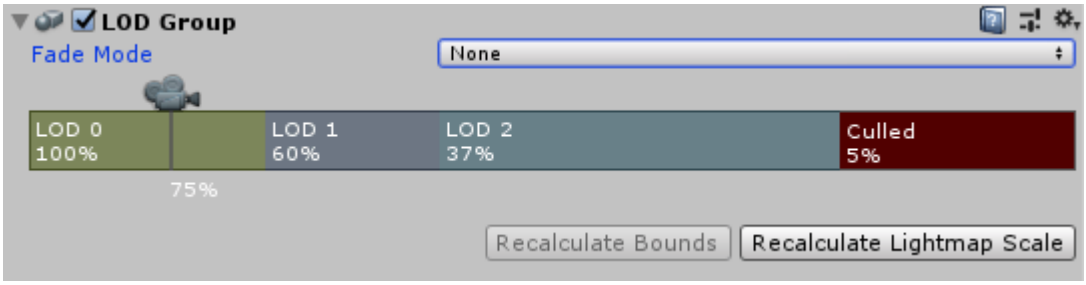
Implementing Level of Detail (LODs) in Unity is based on using the Component menu option.

To enable LODs for a model:

1. Select the required model.
2. Go to Component in the Object Inspector Panel.
3. Choose Rendering.
4. Select LOD Group.

Selecting LOD Group displays the LOD Group window in the Object Inspector Panel. Different LOD levels can be selected to expose the Renderers panel, where the meshes with the required level of complexity can be chosen. The available LOD levels can be resized, depending on the point where you want the application to move between LODs.

Figure 5-3: LOD settings



If necessary, the Fade Mode can be set to either Cross Fade or SpeedTree. Depending on the application of the object, the Fade Mode exposes a blend factor that can be accessed within a shader. The blend factor allows for the implementation of smooth blending between LODs.

Unity also supports shader LODs, allowing for different shaders, or shader passes, to be applied depending on the current value of either `Shader.globalMaximumLOD` or `Shader.maximumLOD`.

Utilizing shader LODs allows for more efficient shaders to be applied when the benefit these shaders brings would be too costly for the hardware.

6. Color space

Color space prescribes the numerical values that each color is assigned. Color space also defines the area each color is allocated in the space, and therefore how much variation is available within that color.

Originally, rendering was performed in the gamma color space. Otherwise, gamma correction would be required on the final image before it can be displayed on the monitor because monitors are designed to display gamma color space images.

With the advent of Physically Based Rendering (PBR), there has been a shift towards rendering in the linear color space. Doing so allows for the values of multiple light sources to easily and accurately be accumulated within shaders. In the gamma color space, this addition would not be physically accurate due to the curve inherent to rendering in the gamma color space.

Rendering in the linear color space brings further benefits as it can help to reduce specular aliasing. This benefit occurs as increasing the brightness within a scene when rendering in the gamma color space causes objects to rapidly become white which can cause specular aliasing effects. In a linear color space, the object brightens linearly which stops the object becoming white so quickly and therefore reduces the risk of specular aliasing.

How to choose the color space in Unity

In Unity, you choose the color space in Project Player Settings.

To enable linear color space:

1. Go to Edit, then Project Settings.
2. Select Player to open the Project Player Settings panel.
3. Open the Color Space drop-down option that is located inside the Player Settings, Other Settings option.
4. From the Rendering section, choose the Linear option from the drop-down menu.

Figure 6-1: Color space settings

The Graphics API must be set to OpenGLES 3.0 when you set your application to the linear color space. To choose the correct API, uncheck the Auto Graphics API option and then remove OpenGLES2 from the list of named Graphics APIs. The Minimum API Level setting from within the Identification section must also be set to at least Android 4.3 'Jelly Bean', API Level 18.

7. Texture filtering

Texture filtering is a technique that is used to reduce the aliasing that occurs when sampling from textures.

Aliasing occurs where the pixel you are rendering on the screen does not lie exactly on the grid of pixels within the texture that is being mapped onto the object that you are rendering. Instead, it is offset to some degree as the object the texture is mapped to is at an arbitrary distance and orientation to the viewer.

There are two problematic situations that can occur when mapping a texture pixel, or texel, to a screen pixel, either the texel is larger than the screen pixel, in this case it must be minified to fit the screen pixel. Alternatively, a situation can occur where the texel is smaller than the screen pixel, therefore multiple texels must be combined to fit the screen pixel.

Texture filtering relies heavily on mipmapping. This is because that, during magnification, the number of texels to be fetched never exceeds four. However, during minification, as the textured object moves further away, the entire texture can fit within one pixel. In this case, every texel would have to be fetched and then merged for an accurate result, which would be too expensive to implement for a GPU.

Instead, four samples are selected, resulting in an unpredictable under-sampling of the data. Mipmapping overcomes this by pre-filtering the texture at different sizes so that as the object moves away a smaller texture size is applied instead.

Widely implemented methods of filtering include bilinear, trilinear, and anisotropic filtering, with the difference between them being how many texels are sampled, how these texels are combined, and whether mipmapping is utilized in the filtering process.

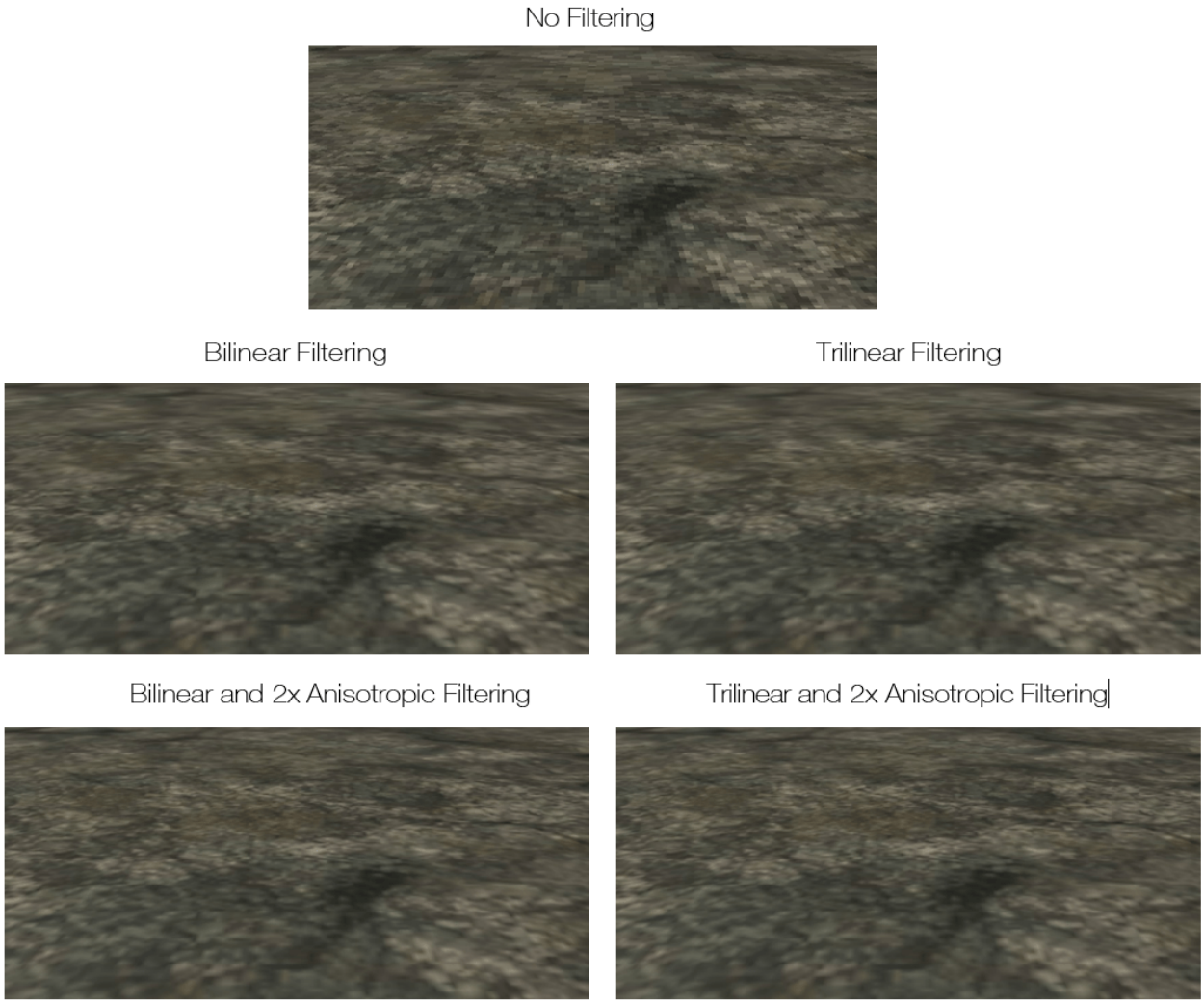
The performance cost of each filtering method varies. So, selecting the type of filtering to be used must be on a case-by-case process where the performance cost of the filtering method is weighed against the visual benefits that it provides.

For example, trilinear filtering is twice the cost of bilinear filtering yet the visual advantages are not always apparent especially on textures being applied to distant objects. Instead, the use of 2x anisotropic filtering is recommended as it often gives better image quality and increased performance.

When using anisotropic filtering, the maximum number of samples must be set. However, due to the performance impact incurred on mobile, we recommend that you exercise caution when using higher numbers of samples, especially eight samples or higher. This technique is best suited for textures on slanted surfaces, such as ground in the distance.

The following set of images displays the difference between the different types of texture filtering. Of particular note, is the indiscernible difference between bilinear and trilinear filtering when 2x sample anisotropic filtering is enabled. Even though trilinear filtering costs more resources to use.

Figure 7-1: Texture filtering methods



Further information on texture filtering can be found in the Real-Time 3D Art Best Practices: Texturing guide.

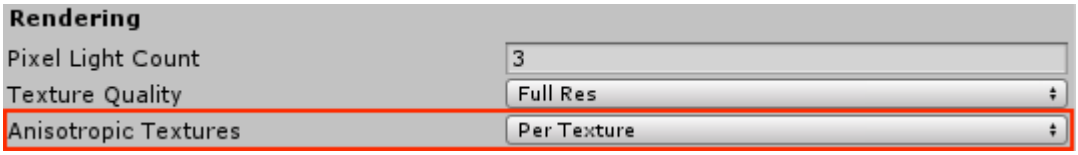
Implementing texture filtering in Unity

In Unity, texture filtering is set in the Project Quality Settings.

To set the texture filtering settings for a texture in Unit:

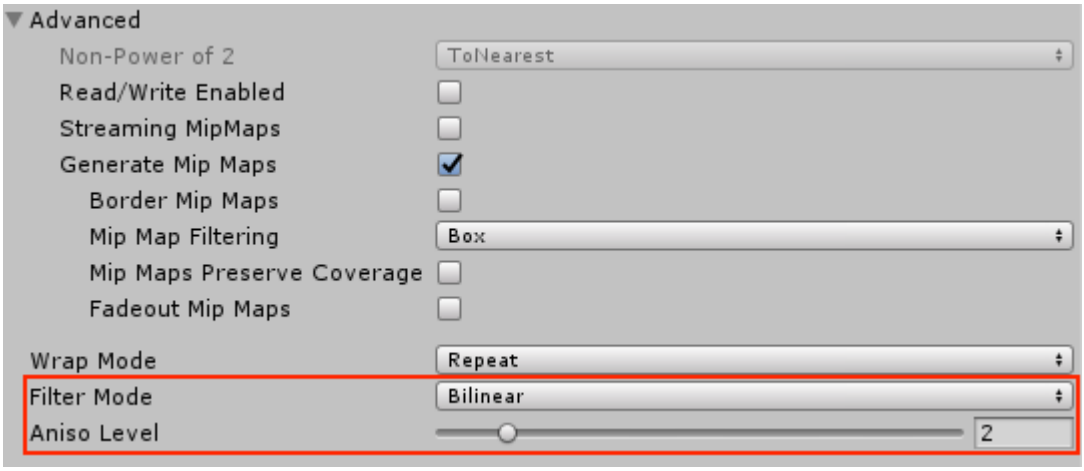
1. Go to Edit.
2. Choose Project Settings.
3. Select Quality to open the Project Quality Settings panel.
4. Choose the Anisotropic Textures drop down from this window.
5. Choose the Per Texture option.

Figure 7-2: Per texture setting



Then for each texture, select it within the Assets section of the Project window so that the Texture Inspector window opens. When the window is open, set both the Filter Mode and the Aniso Level settings for that texture.

Figure 7-3: Texture filter settings



8. Alpha compositing

Alpha compositing is the technique of combining an image with a background image to produce a composite image that has the appearance of transparency.

Alpha testing is a widely implemented form of alpha compositing. However, it can produce severe aliasing effects at the edges of objects as the alpha channel is bitwise, so there is no blending between edges.

Multisampling has no impact in this case as the shader is run only once for each pixel. Each subsample returns the same alpha value, leaving the edges aliased.

Alpha blending is an alternative solution. But, without polygonal sorting, the blending fails and objects are rendered incorrectly. However, enabling sorting is an expensive process and diminishes performance.

Alpha to Coverage (ATOC) is a different method of alpha compositing which can help reduce aliasing. ATOC transforms the alpha component output of the fragment shader into a coverage mask and combines this with the multisampling mask. ATOC then uses an AND operator, only rendering pixels that pass the operation.

Visual example demonstrating how alpha testing and alpha coverage looks

Both the image and the video at the following link demonstrates the difference between a basic alpha test implementation, which is on the left, and an alpha to coverage implementation, which is on the right:

Figure 8-1: Alpha compositing example

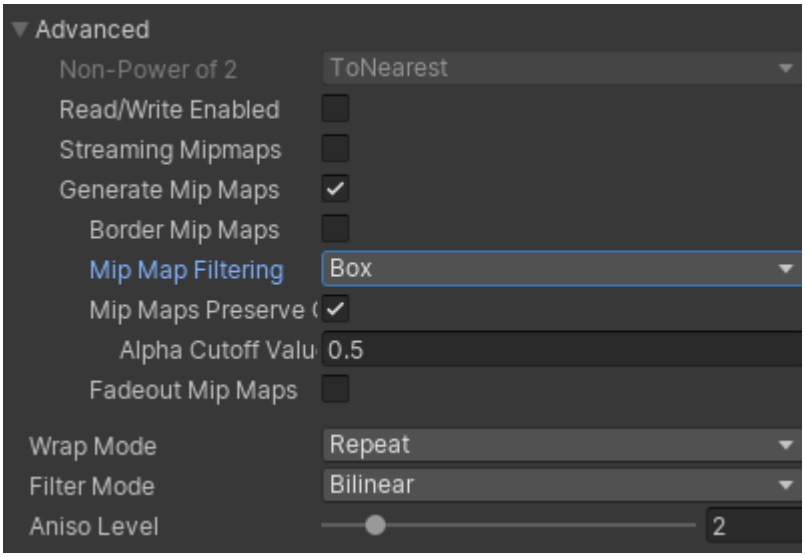
Implementing alpha compositing in Unity

Alpha to Coverage is implemented within Unity shaders.

Mip Maps Preserve Coverage option - Unity 2017 onwards

The Mip Maps Preserve Coverage option removes the need for the mip map level calculation that is required in earlier versions. It can be enabled in the Texture Inspector.

Figure 8-2: Mip Maps preserve coverage



To enable ATOC in Unity from 2017 or later, you must first create a new shader in the Project window, then insert the following shader code:


Note

Apply this shader to the material and set that material to the object which requires alpha compositing.

```
AlphaToMask On
                struct frag_in
                {
                float4 pos : SV_POSITION;
                float2 uv : TEXCOORD0;
                half3 worldNormal : NORMAL;
                };

                fixed4 frag(frag_in i, fixed facing : VFACE) : SV_Target
                {
                /* Sample diffuse texture */
                fixed4 col = tex2D(_MainTex, i.uv);

                /* Sharpen texture alpha to the width of a pixel */
                col.a = (col.a - 0.5) / max(fwidth(col.a), 0.0001) + 0.5;
                clip(col.a - 0.5);

                /* Lighting calculations */
                half3 worldNormal = normalize(i.worldNormal * facing);
                fixed ndotl = saturate(dot(worldNormal,
 normalize(_WorldSpaceLightPos0.xyz)));
                col.rgb *= ndotl * _LightColor0;
                return col;
                }
```

For versions of Unity before 2017

To enable ATOC in pre-2017 versions of Unity, you must also add a mipmap level calculation to your shader code. As with 2017 and newer versions of Unity, you must first create a new shader in the Project Window. Then insert the following shader code:

Apply this shader to the material and set that material to the object which requires alpha compositing.

```
Shader "Custom/Alpha To Coverage"
        {
          Properties
          {
            MainTex("Texture", 2D) = "white" {}
          }
            SubShader
          {
            Tags { "Queue" = "AlphaTest" "RenderType" = "TransparentCutout" } Cull
 Off
            Pass
            {
              Tags { "LightMode" = "ForwardBase" }
              CGPROGRAM
              #pragma vertex vert #pragma fragment frag

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

              struct appdata
              {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
                half3 normal : NORMAL;
              };

              struct v2f
              {
                float4 pos : SV_POSITION;
                float2 uv : TEXCOORD0;
                half3 worldNormal : NORMAL;
              };

              sampler2D _MainTex; float4 _MainTex_ST;
              float4 _MainTex_TexelSize;

              v2f vert(appdata v)
              {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                o.uv = TRANSFORM_TEX(v.uv, _MainTex);
                o.worldNormal = UnityObjectToWorldNormal(v.normal);
                return o;
              }

              fixed4 frag(v2f i, fixed facing : VFACE) : SV_Target
              {
                fixed4 col = tex2D(_MainTex, i.uv);
                float2 texture_coord = i.uv * _MainTex_TexelSize.zw; float2 dx =
 ddx(texture_coord);
                float2 dy = ddy(texture_coord);
                float MipLevel = max(0.0, 0.5 * log2(max(dot(dx, dx), dot(dy,
 dy))));
                col.a *= 1 + max(0, MipLevel) * 0.25; clip(col.a - 0.5);
```

```
                half3 worldNormal = normalize(i.worldNormal * facing);
                fixed ndotl = saturate(dot(worldNormal,
normalize(_WorldSpaceLightPos0.xyz)));
                fixed3 lighting = ndotl * _LightColor0; lighting +=
ShadeSH9(half4(worldNormal, 1.0));
                col.rgb *= lighting; return col;
              }
              ENDCG
            }
          }
        }
```

9. Level design

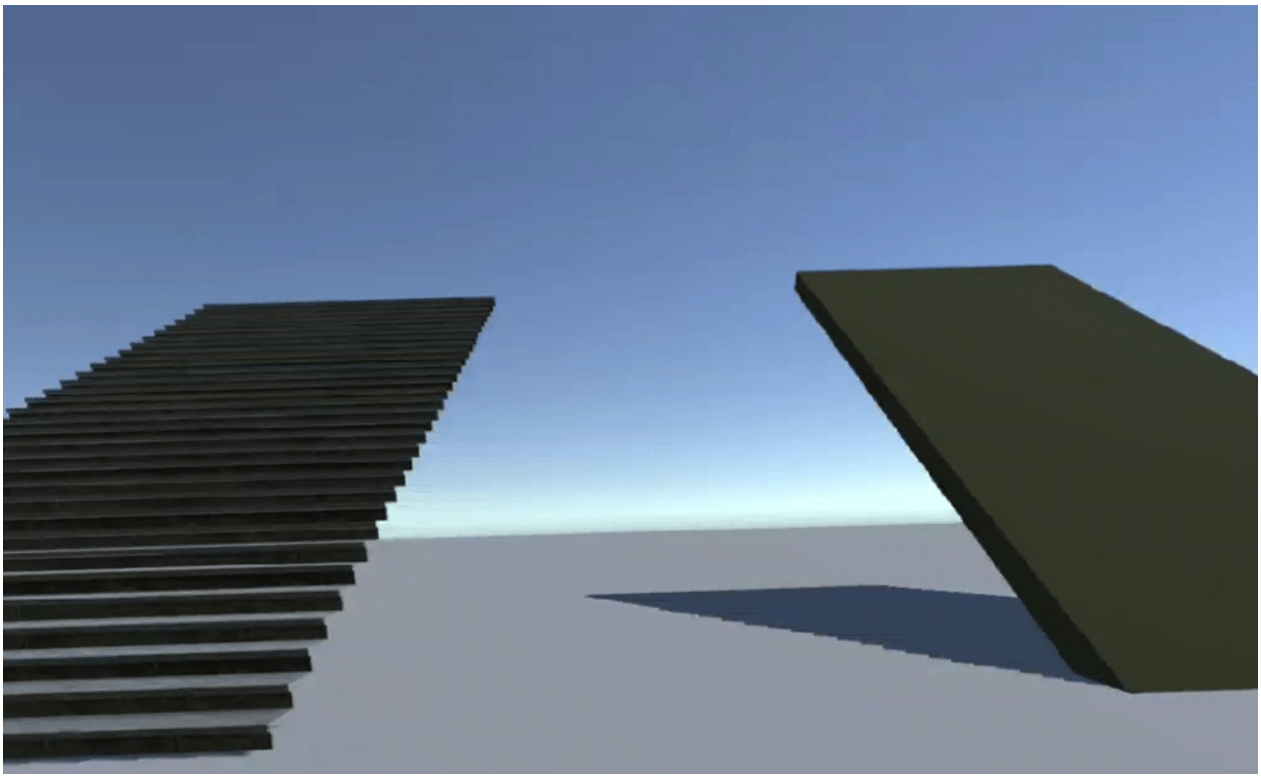
Despite all of the technical options available to help minimize aliasing, careful level design is still vital in helping reduce aliasing artifacts. Poor choices in level design can make every technical solution redundant, so wise decisions in level design still play a critical role in minimizing aliasing.

When creating geometry for a scene, care must be taken when modeling the edges of meshes to avoid sharp or sudden edges. For example, a staircase that has flat edges on each individual step can create aliasing if the viewer rotates their head.

However, the individual steps themselves can also be the cause of aliasing when a staircase is viewed from a large distance. Each step becomes a thin object in the distance that can flicker as the viewer moves around.

The following image shows an example of the different levels of aliasing that is produced when you use stairs, versus a ramp:

Figure 9-1: Stairs vs ramp



Video comparing aliasing on stairs and ramps

The following video shows an example of the different levels of aliasing that is produced when you use stairs, versus a ramp:

Whenever possible, use smooth, round shapes in place of those shapes with hard edges or bevels. Thin objects, such as wires or cables, have a propensity to cause substantial aliasing when they are viewed from a distance. This is because thin objects are rendered as lines, causing aliasing to occur.

Careful consideration must be made on the use of metallic materials. These should be minimized where possible, as metallic objects produce specular effects when lit. These specular effects flicker as the viewer moves, resulting in aliasing. Therefore, the use of matte materials is preferred.

The following image shows a comparison between the aliasing that is caused by metallic and matte materials. The top-half of the image shows a metallic material, while the lower image shows the matte equivalent.

Figure 9-2: Metallic matte objects

Implement scene lighting with caution, as bright shiny lights result in specular effects appearing on the lit objects. Doing so can cause specular aliasing which is very noticeable. The eye is drawn to the ‘flashing’ of the pixel as the specular effect appears and disappears between frames.

To avoid expensive real time lighting calculations and reduce the presence of lighting artifacts such as banding, lighting can be prebaked when the scene is built. To do this, scenes must be designed

so that they are appropriate for prebaking which requires few moving objects and no live day or night cycle.

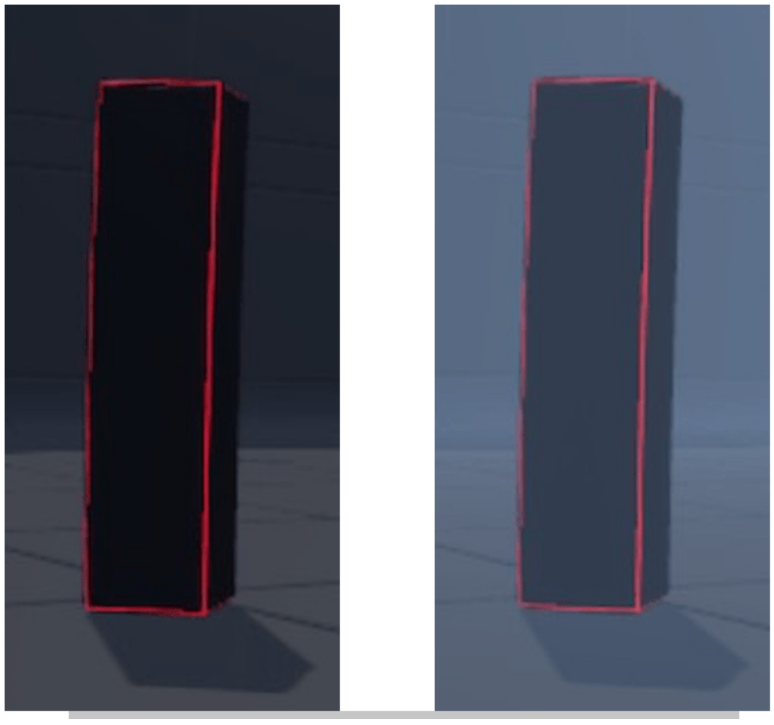
Utilizing light probes can help to minimize the quantity of prebaking required and they are especially advantageous in VR with 6 degrees of freedom.

Reflections require more caution in VR than is typical than in more common use-cases. Techniques like screen space reflections are too computationally expensive for VR. Therefore, other techniques must be employed for reflections. Possible techniques for objects that require high-quality reflections include reflection probes and cube maps, as reflections can be pre-rendered for use at runtime.

To help hide aliasing, particle effects can be deployed, such as fog or smoke. These techniques, that have long been used to hide short render distances, can also be used to cover aliasing that is caused by objects in the distance.

In the following figure, the right-hand image demonstrates how aliasing is reduced after the addition of a particle effect:

Figure 9-3: Particle effects



10. Banding

Banding occurs as a result of the inability to accurately represent the required colors within the given number of bits per pixel. Within VR, banding manifests itself as distinct bands of colors that are marked by abrupt changes between each band.

The bands are most apparent within VR as the user is immersed within the scene. Therefore, their eyes adjust to the light levels within the scene. When the eyes have become adjusted, the bands are even more apparent. If there are many bands, then the eyes are constantly adjusting to the changing light levels, which can also become physically tiring.

Mitigation technique - dithering

Dithering is the process of introducing noise either to the banding material or to the viewer. Through this process, the distinct bands of color are broken up and disrupted therefore they are no longer so distinct.

There are many variations of dithering which can be introduced, each of which either apply a different form of noise, or take a different approach to gathering the noise which is applied. For example, generating noise in real time, while others sample noise from a texture crafted to contain random noise.

Mitigation technique - tonemapping

Tonemapping is a subset of color grading that transforms High Dynamic Range (HDR) colors so that they fit within the Low Dynamic Range (LDR) that is suitable for displaying on non HDR-capable screens. Tonemapping uses a Look Up Table (LUT) so that each color is mapped to the appropriate corresponding color for the new tone.

Any low lighting levels or sharp gradients in textures that are causing banding can be reduced by applying tonemapping to the scene.

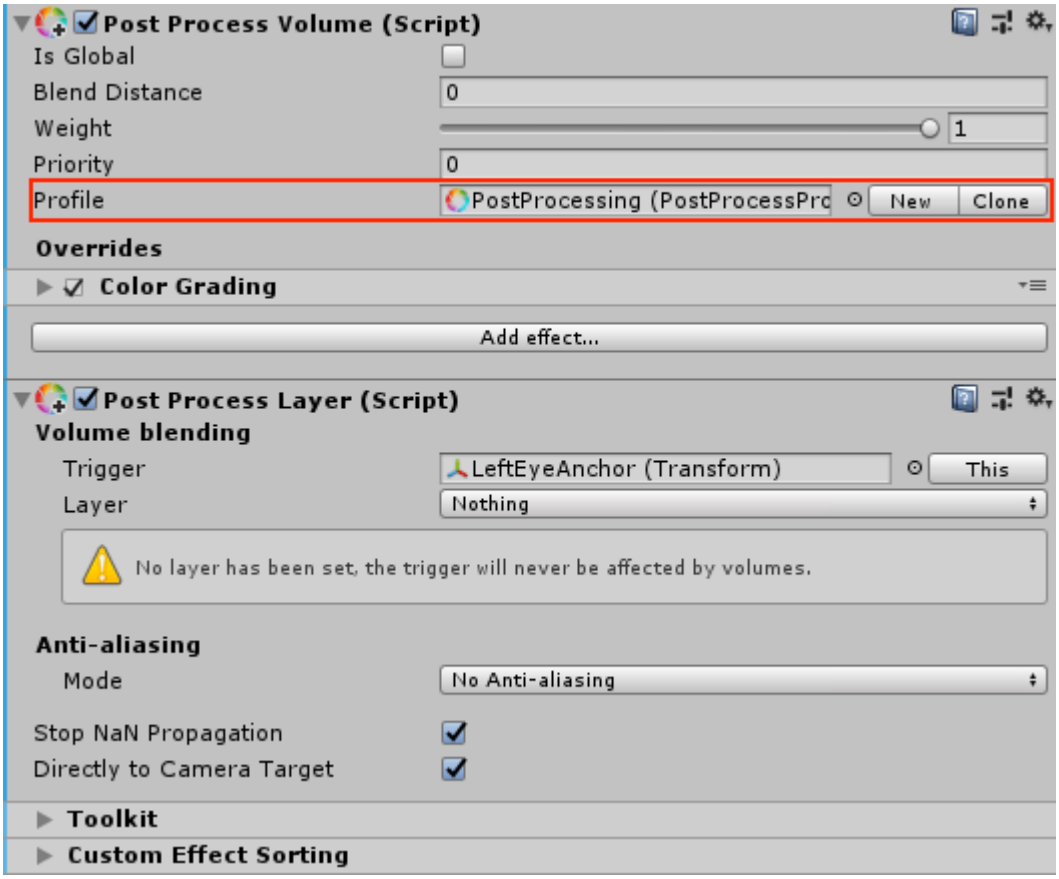
Implementing dithering in Unity

Dithering in Unity is often done through the Post Processing package, which must be added and setup. Alternatively, there is an effect that is built into the camera in the Universal Render Pipeline.

To enable dithering:

1. In Window, then Package Manager, search for Post Processing and install the package. Make sure that it is version 2.x that you are using.
2. Create a Post Processing Profile in the Project Window, by right-clicking in the Project Window.
3. Under Create, choose Post-processing Profile.
4. From within the Inspector for a camera, add a Post Process Volume and a Post Process Layer component to each camera in the scene that requires dithering.
5. Finally, from within the Post Processing Volume, set the created Post Processing Profile as the Profile.

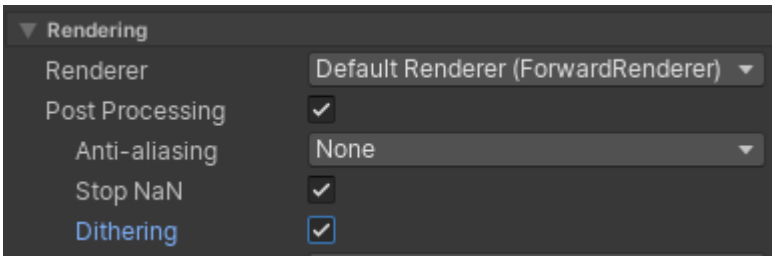
Figure 10-1: Post processing profile



As shown in the following screenshot, in Universal Render Pipeline in Unity, you can enable dithering in a camera:

1. In the Inspector window for the Camera in the rendering section, tick the Post-Processing box.
2. Tick the Dithering box below it.

Figure 10-2: Enable dithering



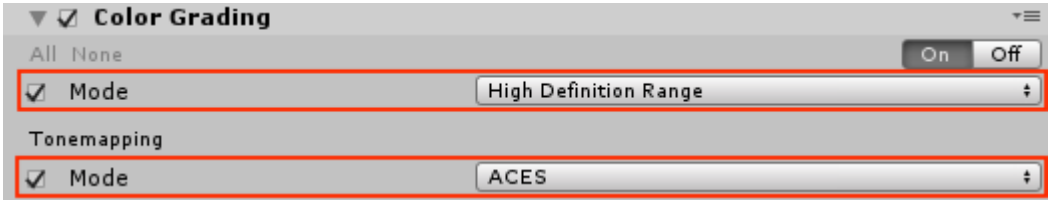
Implementing tonemapping in Unity

Tonemapping in Unity requires the installation and setup of the Post Processing package.

To enable tonemapping:

1. In Window then Package Manager, search for Post Processing, and install the package. Make sure that it is version 2.x that you are using.
2. Next, create a Post Processing Profile in the Project Window, by right-clicking in the Project Window.
3. Under Create, select Post-processing Profile.
4. Now select this Post-processing Profile.
5. In the Inspector, select Add effect….
6. Under Unity, choose Unity Color Grading, then set the Mode option to High Definition Range.
7. Tick the box next to the Mode for tonemapping and choose ACES in the combo-box.
8. From within the Inspector for a camera section, add a Post Process Volume and a Post Process Layer component to each camera in the scene that requires tone mapping.
9. From within the Post Process Volume, set the pre-created Post Processing Profile as the Profile.

Figure 10-3: Setting tone mapping



11. Bump mapping

Bump mapping is a technique that is used for reducing the vertex count of a mesh and involves simulating the finer details, such as bumps, on the surface of an object. The required simulation is performed by manipulating the normals of the object before being used for the lighting calculations.

While normal mapping is a suitable technique for most uses, it is not as effective in VR as the player can easily change their viewing angle of a normal mapped texture. Therefore, the illusion of depth is broken as this change of perspective is not accounted for within the normal mapping technique.

Also, normal mapping cannot account for the use of stereoscopic lenses that are used in VR headsets as the normal is only generated from one viewpoint. Therefore, each eye receives the same normal which looks incorrect to the human eye.

Video comparing different methods of bump mapping

The following video shows an example of both normal mapping, and parallax occlusion mapping:

Mitigation technique - Normal mapping

Normal mapping is the most common implementation of bump mapping and involves creating both a high and low polygon version of a mesh during the modeling process. A normal map is then created by exporting from the high polygon version, and the normals of the finer details are then stored in the normal map texture.

When rendering, the fragment shader samples from the normal map and normals are generated from the sampled values. These generated normals are then combined with the surface normals of the low polygon version before being used when calculating lighting. The lighting then shows the finer surface details, without the need to render the individual vertices of theses details.

While normal mapping is, typically, not as effective in VR, normal maps are still more effective than a flat material. Especially when careful consideration is given to the positioning of the normal maps when lighting a scene.

Mitigation technique - Parallax occlusion mapping

Parallax occlusion mapping is a technique similar to normal mapping. However, it accounts for the angle of the viewer, relative to the surface normal, when displacing the texture coordinates.

Therefore, at steeper viewing angles, the texture coordinates are displaced by a higher degree. This maintains the illusion of depth.

Parallax occlusion mapping is a computationally expensive process. So only use this technique on smaller materials that the viewer can get close to. Textures that are further away gain little from parallax occlusion mapping because the viewing angle cannot change considerably.

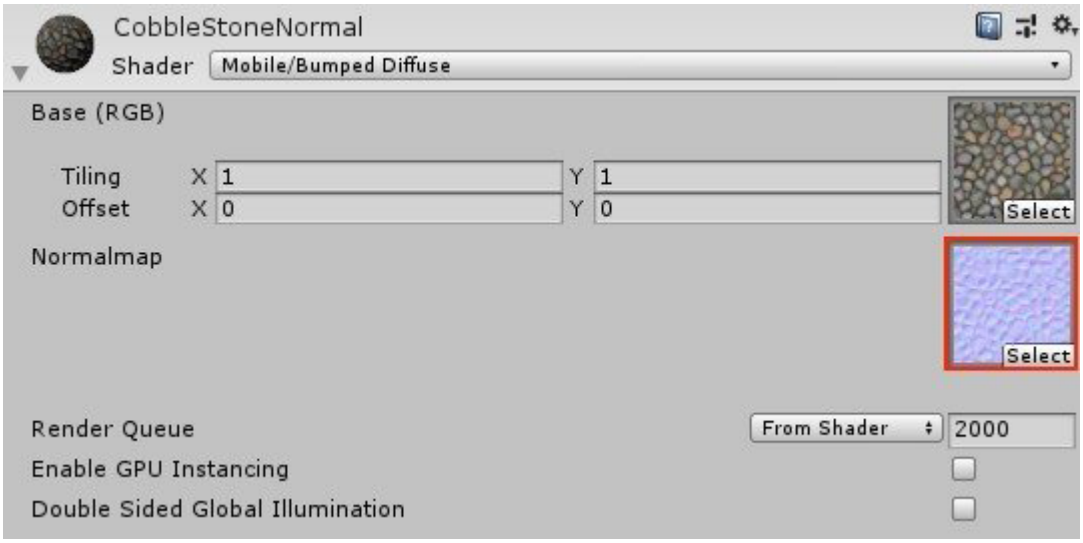
Implementing normal mapping in Unity

Normal mapping is available in many built-in shaders within Unity.

To add normal mapping to a material:

1. Select the material in the Project Window.
2. Open the Material Inspector Panel and select a shader that includes support for normal mapping, such as Standard, Universal Render Pipeline, or Bumped Diffuse for mobile.
3. Finally, set the required Normal Map texture.

Figure 11-1: Normal map texture



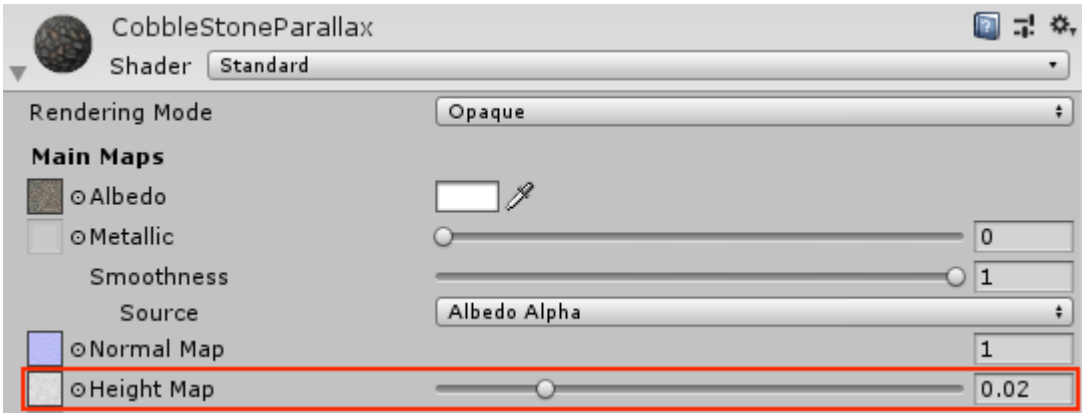
Implementing parallax occlusion mapping in Unity

Parallax occlusion mapping can be done using the built-in shaders in Unity.

To add parallax occlusion mapping to a material:

1. Select the material in the Project Window.
2. Open the Material Inspector Panel and select a shader that supports parallax diffuse mapping. For example, the Standard shader. Finally, set the required Albedo, Normal Map, and Height Map texture.

Figure 11-2: Parallax occlusion mapping



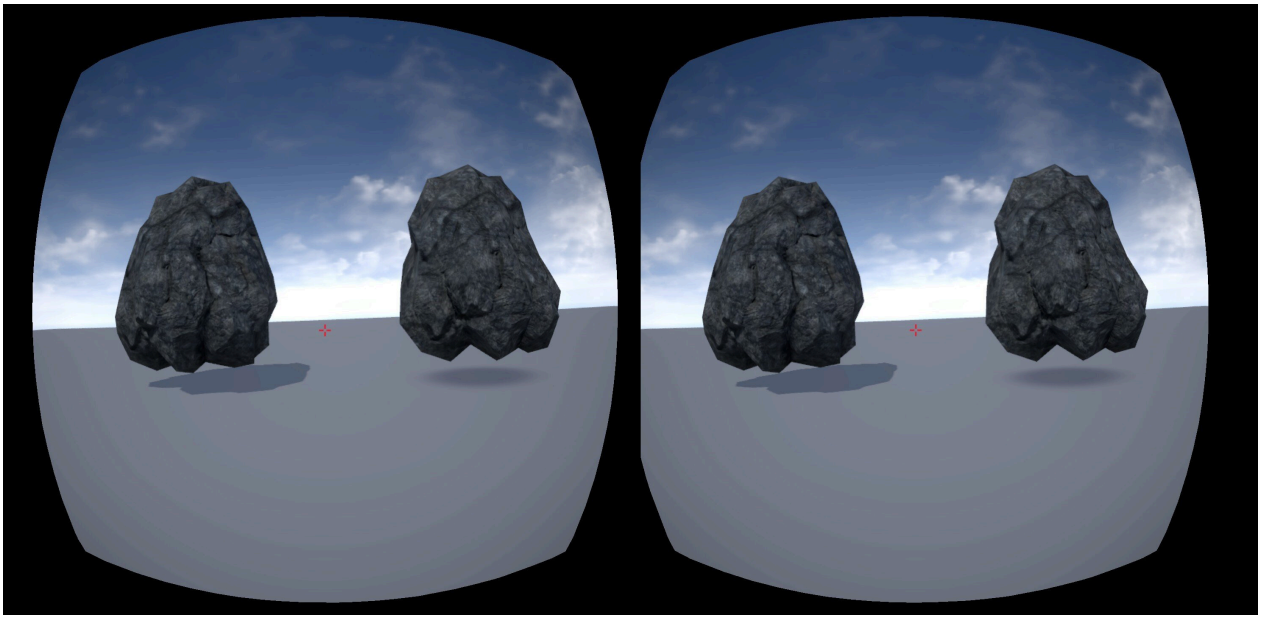
12. Shadows

Traditional methods of shadow mapping are computationally intensive on mobile device. This causes a serious decrease in performance, as shadow maps require extra frame buffers and render passes.

Therefore, shadow buffers on mobile devices are often low resolution and do not use any filtering. In turn, the shadows often introduce large amounts of aliasing, damaging the realism of the simulation due to artifacts such as shadow acne and hard shadows.

The following screenshot demonstrates the comparison between a typical shadow, on the left, and a blob shadow, on the right:

Figure 12-1: Comparison of shadow mapping



Another technique worth considering is the use of baked shadows. Unlike with shadow mapping, shadow baking does not require any additional render passes. Instead, high quality shadows are baked into the textures that they cast shadows on.

Mitigation technique - Blob shadows

The recommended practice in VR is to avoid rendering shadows where possible. But, if a shadow is required, such as underneath a player, then blob shadows can be rendered underneath objects. Blob shadows produce considerably less aliasing than using typical shadow mapping techniques while providing an improvement to performance.

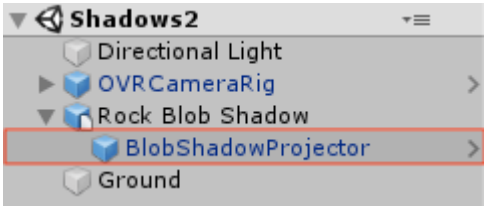
Implementing blob shadows in Unity

Blob shadows are available as a scene component in Unity for free from the Asset Store.

To implement blob shadows:

1. Import the Unity Standard Assets from the Unity asset store.
2. Navigate to the BlobShadowProject prefab within Assets > StandardAssets > Effects Projects > Prefabs.
3. Drag the BlobShadowProject prefab into the Hierarchy as a child of the object that requires a blob shadow.

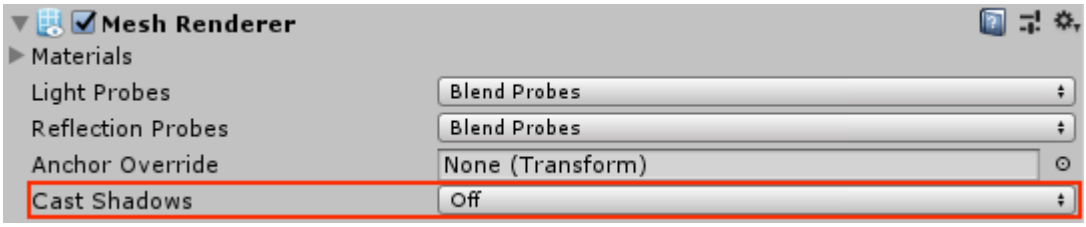
Figure 12-2: Setting a child object



When completed, a blob shadow appears. To finish setting up blob shadows:

1. Edit the prefab options. Make sure that you include the location of the blob shadow that is relative to the object that must be casting the shadow.
2. Select the object that is casting the shadow and then set the Cast Shadows option to Off.

Figure 12-3: Disabling cast shadows



13. Check your knowledge

The following questions will help you test you knowledge:

Which two aliasing variants are especially apparent in a VR environment?

Geometric and specular aliasing

Which term describes the process of using eye-tracking to display higher quality graphics where the user is currently looking?

Foveated rendering

What name is given to the process of intentionally introducing noise into an image in order to mitigate the problem of color banding?

Dithering

14. Related information

Here are some resources related to material in this guide:

- Arm Developer: Virtual Reality Presentations
- Unity at Arm
- Arm Developer: Graphics and Gaming Development
- Arm Developer: Virtual Reality Optimization with Arm Mobile Studio