arm

Improving Visual Rendering Quality in Mobile Virtual Reality on Unity

Developer Guide

Ryan O'Shea
2019-06-01

Table of Contents

Introduction

Virtual reality (VR) is an immersive experience that involves placing a user into a simulated environment using a head mounted display that is placed directly in front of the user's eyes.

VR allows for an entirely new level of immersion when compared to existing gaming experiences as the user is completely immersed into the virtual experience, allowing for more thrilling and engaging content to be delivered.

VR as a technology is growing rapidly, and a recent Newzoo survey of over 2500 people in the US[21] proves this. 28% of people surveyed have used VR in the last six months and of those people, 73% have used VR to play games. Additionally, VR is very popular on mobile, from the people who own a VR headset, 24% of them own a Samsung GearVR headset. Also, with the release of standalone devices such as the Oculus Quest which are also based on mobile technology, this area is set to continue expanding.

Despite this, VR on mobile currently contains a number of limitations, especially when compared with VR on desktop. These limitations are often visible in computer graphics but are amplified in VR by the fact that the rendered image is extremely close to the user's eyes and that the lenses used in VR have a magnifying effect, causing increased annoyance and distraction to the user.

These problems can make mobile VR an unrealistic, unconvincing and at times, possibly even an uncomfortable experience. Therefore, to achieve the true potential of VR these limitations must be either solved or mitigated.

It is fairly simple to diagnose the current limitations of VR on mobile and John Carmack, among others, already has an extensive selection of social media posts on this topic[20]. These posts are aimed at informing developers how best to design and optimize these experiences for mobile hardware as well as suggestions for mitigating the limitations of mobile VR.

Despite this existing material, it seems apparent there is a lack of clear, explicit guidelines to aid developers in knowing which features are best to implement when trying to mitigate these flaws and exactly how best to implement these features.

Article structure

This document is designed to be a guide on how to improve rendering quality in mobile VR, overcoming the most common pitfalls and bad practices with clear explanations and graphical examples before and after the suggested improvement has been implemented.

The article is structured so that each identified problem is described, mitigations are then suggested and detailed with the use of graphics to aid this description where possible. Guidance is also provided for correctly implementing the suggested mitigations in Unity.

This document does not include the best practices for achieving the high performance for VR on mobile, although the techniques suggested have been tested to be suitably performant on mobile. There is a direct correlation between performance and quality which is ever present in graphics and is even more pronounced in VR therefore enabling features such as multiview and maintaining a constant awareness of performance is compulsory when developing for VR.

Factors Impacting Mobile VR Rendering Quality

Aliasing

Description

Aliasing is a consequence of attempting to capture information without using sufficient samples to faithfully recreate it and is an inescapable aspect of signal processing for most practical uses. In audio and video aliasing results in the shape of the produced digital signal not matching the original signal, and is most apparent at high audio frequencies and the finer details of a video. Aliasing begins to occur when the sample rate falls below Nyquist's Frequency[14], which is equal to half of the highest frequency in the content.

The graphics rasterization process also causes aliasing to occur as a direct result of displaying objects seen from a camera in motion against the discrete grid of pixels which form the display. In a VR environment, there are two specific aliasing variants which are especially apparent, these are geometric and 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. An example of this geometric aliasing can be seen on the red edges of the shapes in the image below.

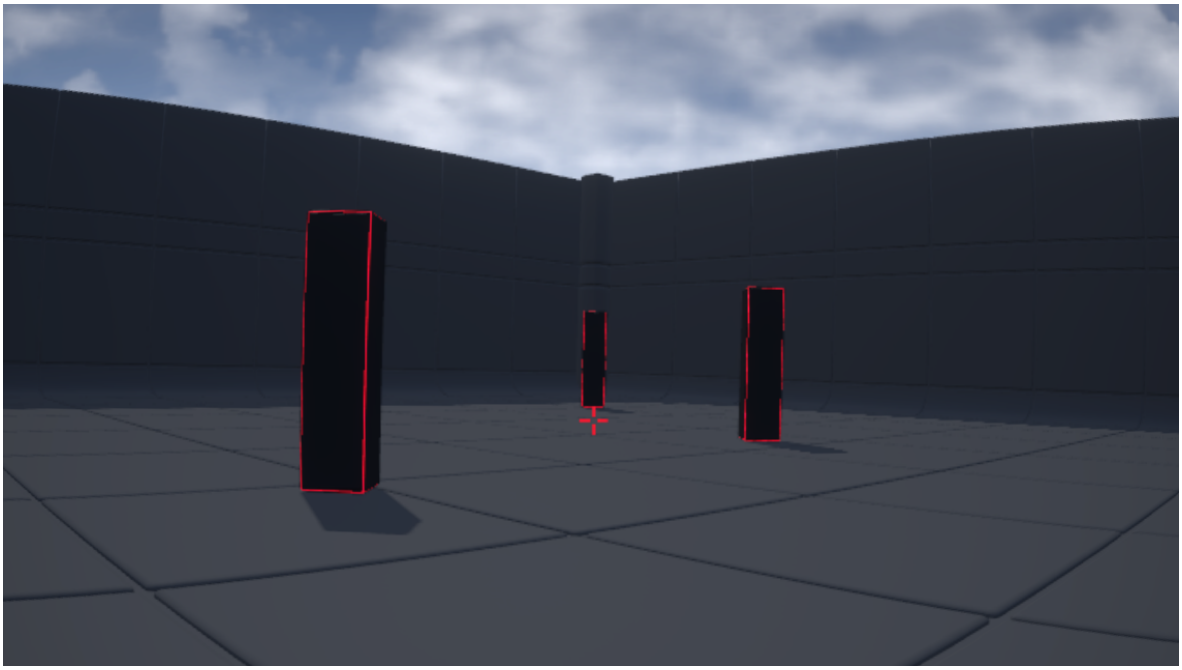


Fig. 1 - Example of geometric aliasing

Geometric aliasing can also occur when polygonally dense meshes are rendered onto a low resolution display, resulting in an extremely high density due to the quantity of polygons within a width of a few pixels. This huge polygonal density results in a large amount of aliasing as it introduces considerably more edges, which in turn results in more aliasing. Additionally, the high density is also extremely inefficient for the rasterization process within the GPU as it 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. This shimmering effect is particularly annoying for the user as it distracts their attention, therefore impacting the sense of immersion and reducing the quality of the user experience. The ridges in the wall on the image below demonstrate specular aliasing.

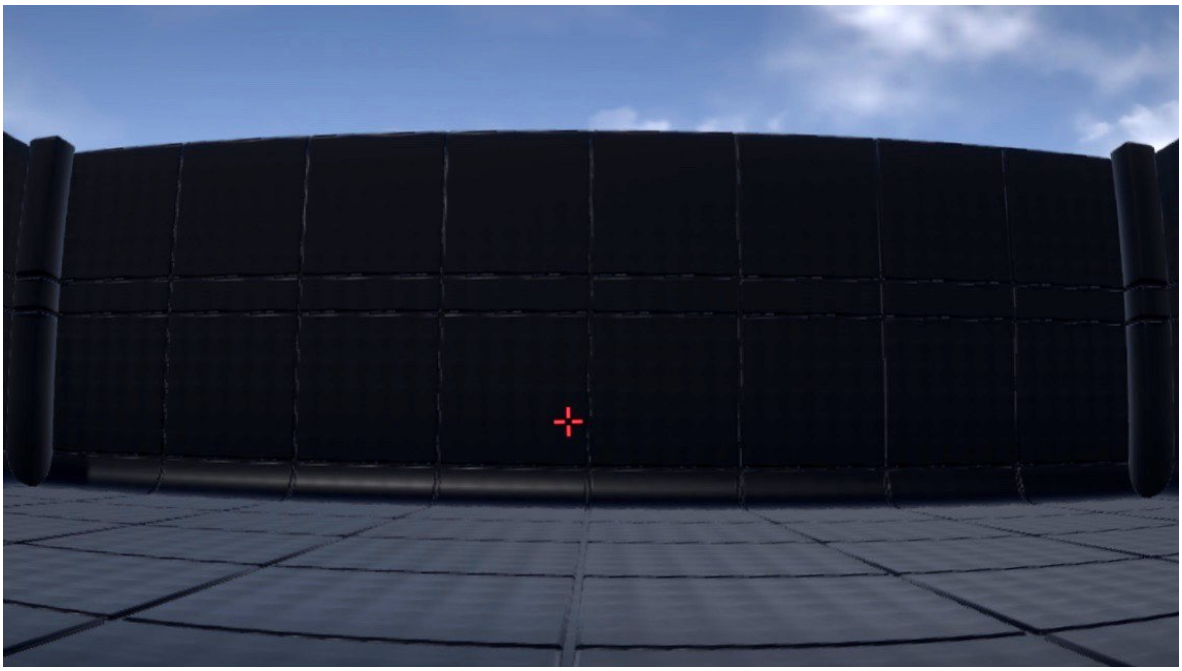


Fig. 2 - Example of specular aliasing

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

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

Mitigations

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, therefore guaranteeing that the texture and all its generated mips will require no more than 1.5 times the original texture size.

These mipmaps can either be hand generated by an artist or computer generated and are uploaded to the GPU which 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[15] on distant surfaces.

Fig. 3 - 512x512 texture with nine mip levels to 1x1

Caution should be taken when storing multiple successive mipmap levels within a single texture atlas as visual problems can arise when foveated rendering[16] is utilized. These problems arise as a result of one tile being rendered at native resolution but the neighboring tile is rendered at one quarter resolution therefore the texture is sampled from the mipmap two levels lower.

One resulting problem occurs during sampling, 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 will exhibit a color differential. This can be solved by extruding the edge of the texture to create an isolation gap between entries into the atlas.

Unity Implementation

To enable mipmap generation in Unity, select a texture within the A*ssets* section of the *Project* window so that the *Texture Inspector* window opens. Once it is open, enable the *Generate Mip Maps* tick box[1].

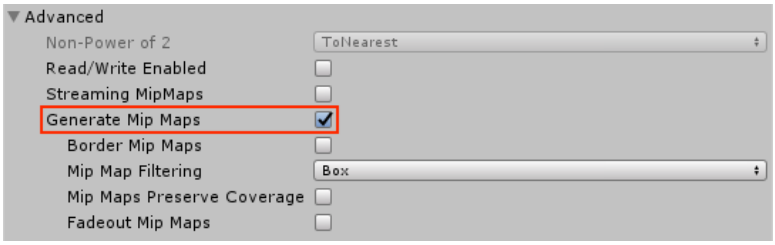


Fig. 4 - Mipmap generation settings in Unity

Level of Detail

Level of Detail (LODs) is the technique of decreasing the detail and complexity of an object as the distance between said object and the viewer increases. By using LODs when the viewer is close to an object it posses a high level of geometric detail so the object appears detailed and accurate yet as the distance between the object and viewer increases the object becomes less geometrically complex.

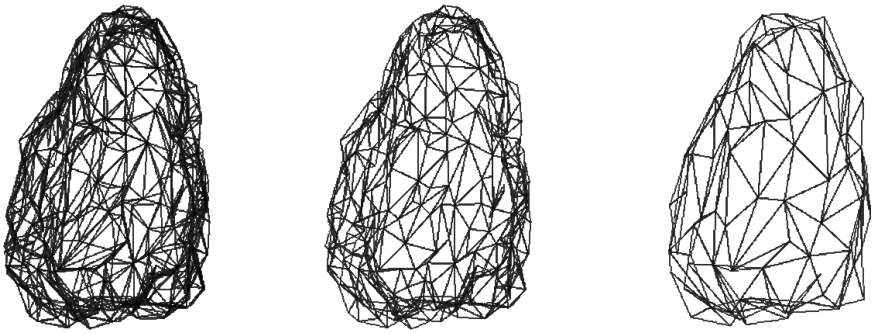


Fig. 5 - Three LODs for a single mesh

The benefits of changing to a model with a lower level of detail as the distance between the viewer and model increase are two fold, not only does this technique reduce aliasing by decreasing the possibility for geometric aliasing to occur but it also increases performance by decreasing the number of vertices that must be shaded by the vertex shader.

LODs should be created with notable differences in the vertex count between each LOD level to avoid subpixel geometry occurring within the mesh, as this will result in unnecessary oversampling which will reduce performance.

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

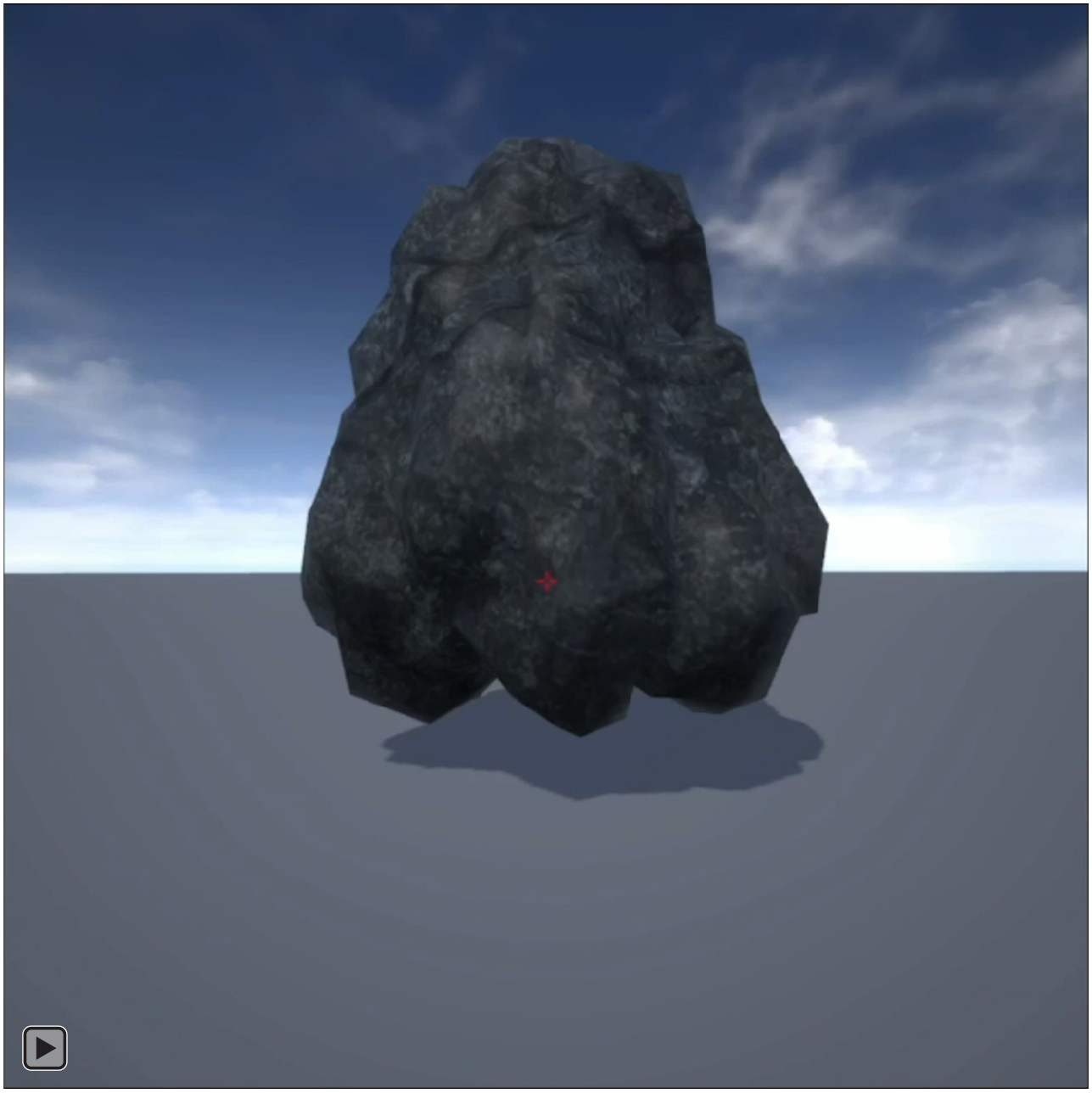


Fig. 6 - Video demonstrating LODs
If this video does not load, please click here: https://www.youtube.com/watch?v=jsLWK6-M1F0

Unity Implementation

To enable LODs for a model in Unity, select a model, then go to *Component*, *Rendering* and select *LOD Group*.

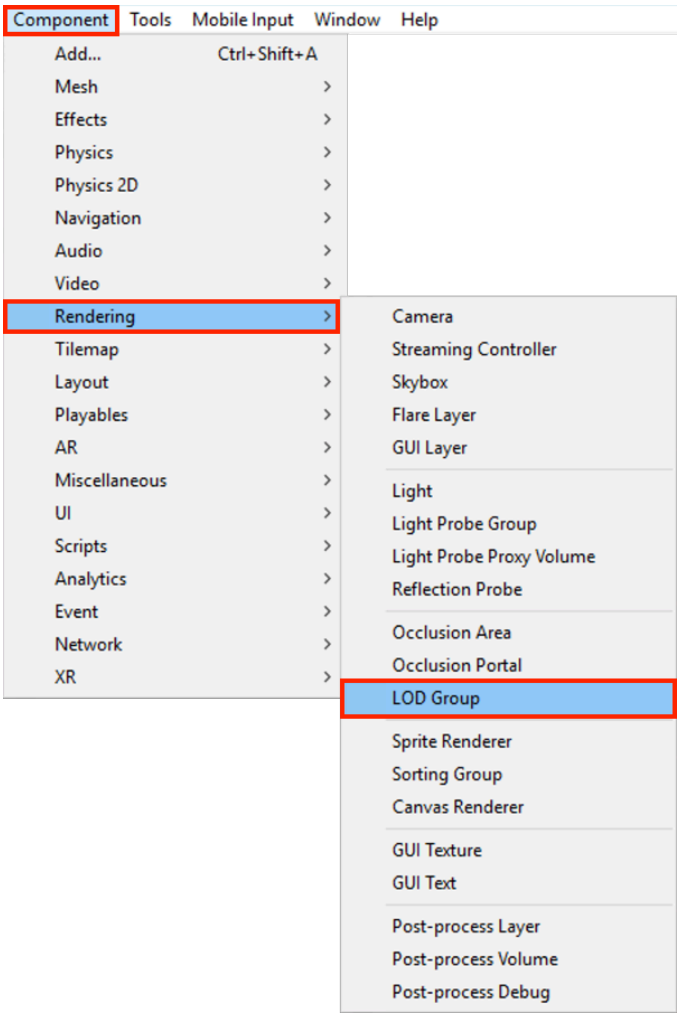


Fig. 7 - Enabling LOD Groups for an object in Unity

This will display the *LOD Group* window in the *Object Inspector Panel*, from selecting each LOD level can be selected to expose the *Renderers* panel, in this panel the meshes with the desired level of complexity can be chosen. These LOD levels can be resized depending on the point at which to move between LODs.

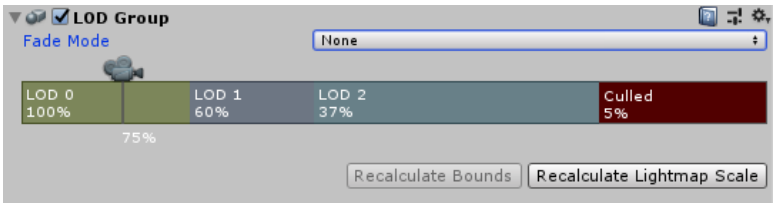


Fig. 8 - LOD settings in Unity

If desired, the Fade Mode can be set to either Cross Fade or SpeedTree depending on the application of the object, this exposes a *blend factor* that can be accessed within a shader to allow for the implementation of smooth blending between LODs[5].

Unity also supports shader LODs, allowing for different shaders or shader passes to be applied depending on the current value of `Shader.globalMaximumLOD` or `Shader.maximumLOD`. Utilizing this allows for more performant and less complex shaders to be applied when the benefit these shaders brings would be too costly for the hardware[3].

Multisample Anti-Aliasing

Multisample anti-aliasing (MSAA) is an anti-aliasing technique which is a more efficient variation of supersampling. Supersampling renders the image at a higher resolution before downscaling to the display resolution, therefore 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, reducing the "jaggies" along the edges of triangles, far outweigh the cost and it should be used whenever possible.

Mali GPUs are designed for full fragment throughput when using 4x MSAA so it can be used with only a minor hit to performance, which is caused by additional fragments that are generated along the edges of triangles.

Unity Implementation

To enable MSAA in Unity, go to *Edit*, *Project Settings* and select *Quality*, this will open the *Project Quality Settings Panel.*

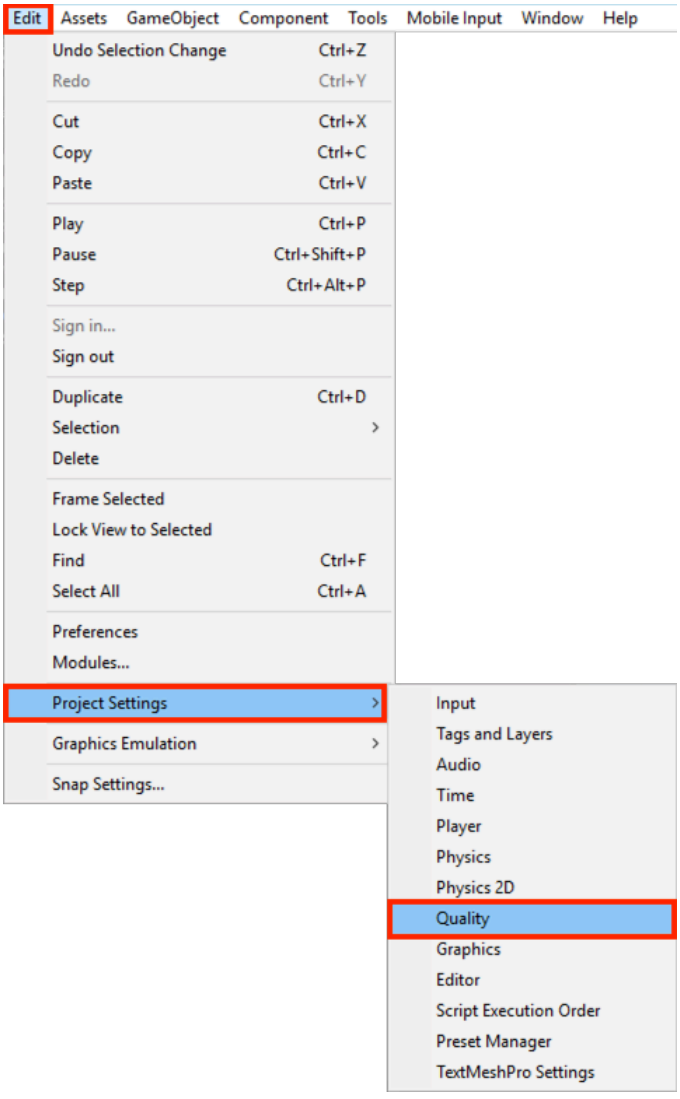


Fig. 9 - Opening MSAA settings in Unity

In this window select the *Anti Aliasing* dropdown and choose the appropriate setting, preferably 4*x* *Multi Sampling* where possible, making sure to set this for all quality levels[4].

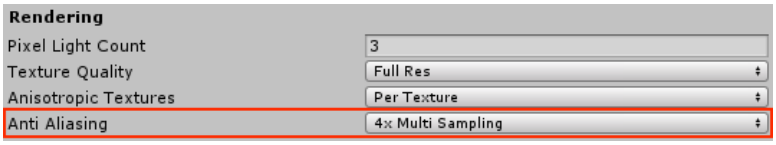


Fig. 10 - MSAA settings in Unity

Color Space

Color space is a model which represents the numerical values which each color is assigned, it also describes the amount of area that each color will be allocated in the space therefore defining how much variation is available within that color.

Originally, rendering was performed in the gamma color space[17] 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)[18] there has been a shift towards rendering in the linear color space[17] as this allows for the values of multiple light sources to easily and accurately be accumulated within shaders whereas 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 to aliasing 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 become increasingly white which can cause specular aliasing effects due to their white appearance. In a linear color space the object will brighten linearly with the increase in brightness which stops the object becoming white so rapidly and therefore reduces the risk of specular aliasing.

Unity Implementation

To enable Linear color space in Unity, go to *Edit*, *Project Settings* and select *Player*, this will open the *Project Player Settings Panel.*

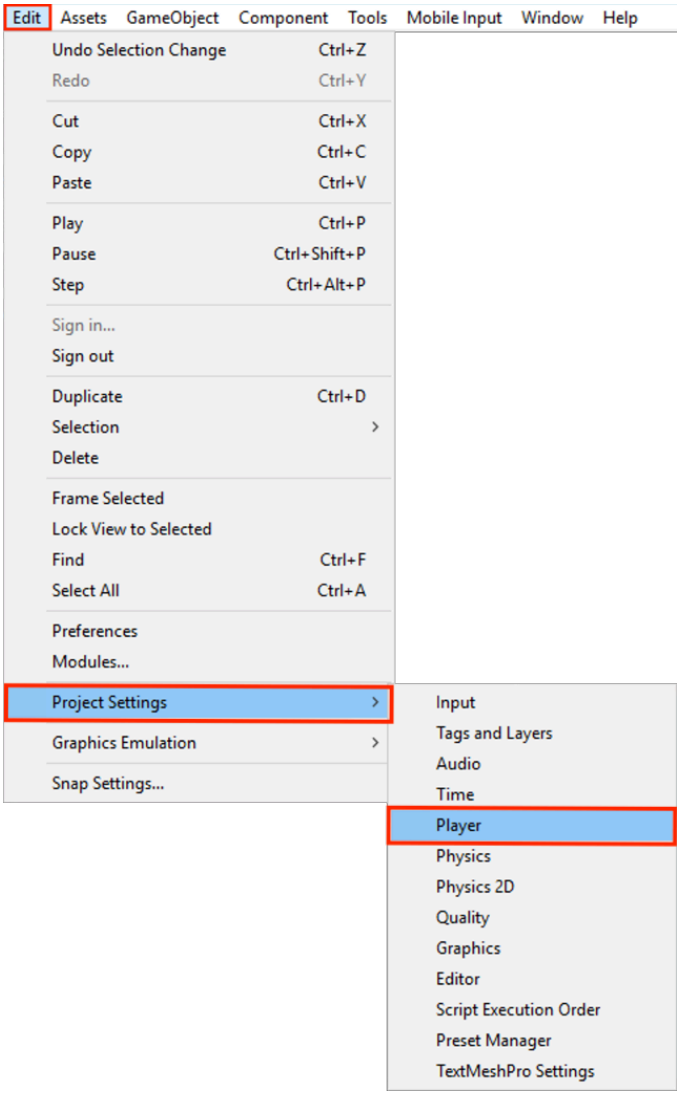


Fig. 11 - Opening player settings in Unity

Inside of the *Player Settings Panel* within the *Other Settings* tab, underneath the *Rendering* section, open the dropdown option named named *Color Space* and choose *Linear*.

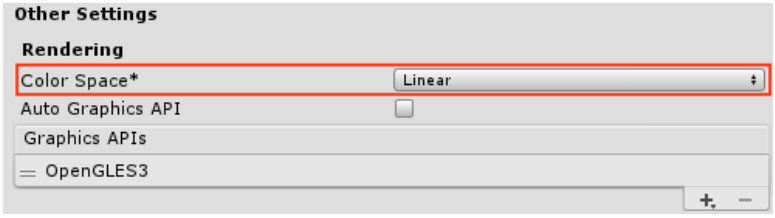


Fig. 12 - Color space settings in Unity

Once changing to Linear color space the Graphics API must be set to either at least OpenGLES 3.0 or Vulkan, to set this uncheck the *Auto Graphics API* option and remove *OpenGLES2* from the list named *Graphics APIs*, the *Minimum API Level* setting within the *Identification* section must also be at least *Android 4.3 'Jelly Bean' (API Level 18)*[5].

Texture Filtering

Texture filtering is a technique to reduce the aliasing that occurs when sampling from textures, this aliasing results from textures being mapped onto objects where the pixel on the surface does not lie exactly on the grid of pixels but rather is offset to some degree due to the object the texture is being mapped to an arbitrary distance and orientation to the viewer.

There are two problematic situations that can occur when mapping a texture pixel (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.

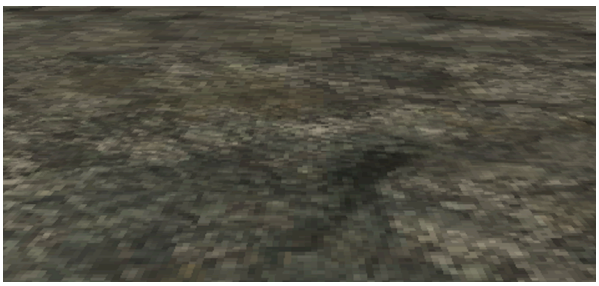
Texture filtering relies heavily on mipmapping, this is because during magnification the number of texels to be fetched never exceeds four but during minification as the textured object moves further away the entire texture may fit within one pixel. In this case all the texels would have to be fetched and merged for an accurate result, which would be too expensive to implement for a GPU. Instead 4 samples are selected, resulting in 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 together and whether mip mapping is utilized in the filtering process. The performance cost of each filtering method varies so selecting the type of filtering to be used should be a case by case process where the performance cost of the filtering method is be weighed against the visual benefits it provides.

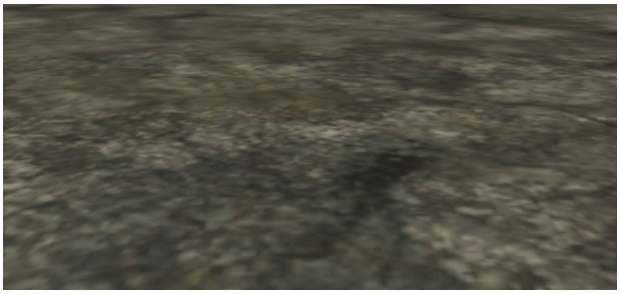
For example, trilinear filtering is twice the cost of bilinear filtering yet the visual advantages are not always apparent especially on textures being applied to distant objects. Instead, the use of 2x anisotropic filtering is recommended as it often gives better image quality as well as increased performance. When using anisotropic filtering the maximum number of samples must be set but it is recommended to exercise caution when using higher numbers of samples, such as 8 samples or higher, as the performance impact makes it inappropriate for mobile. This techniques is best suited for textures on slanted surfaces, such as ground in the distance.

Below is a set of images displaying the difference between the different types of texture filtering. Of particular note, is the indiscernible difference between bilinear and trilinear filtering when 2 sample anisotropic filtering is enabled, despite the additional performance cost incurred.

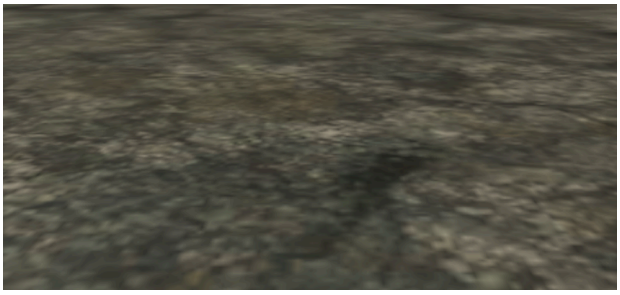
No Filtering



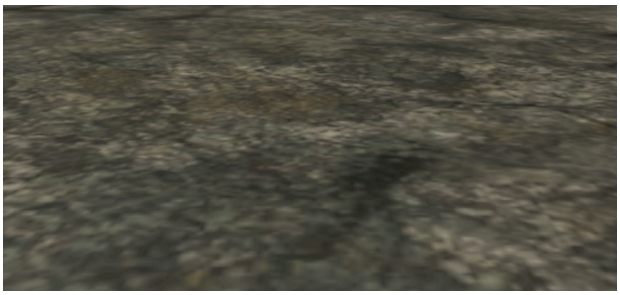
Bilinear Filtering



Trilinear Filtering



Bilinear and 2x Aniso Filtering



Trilinear and 2x Anisotropic Filtering

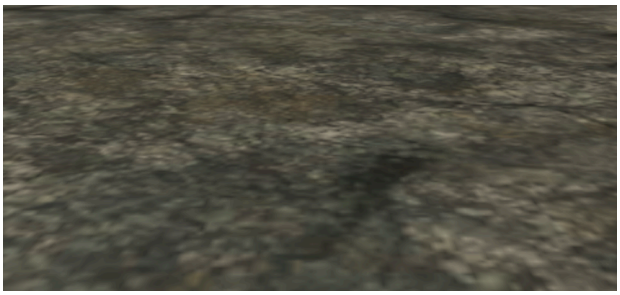


Fig. 13 - Comparison of different texture filtering methods

Unity Implementation

To set the texture filtering settings for a texture in Unity, go to *Edit*, *Project Settings* and select *Quality*, this will open the *Project Quality Settings Panel,* see Fig. 9[4].

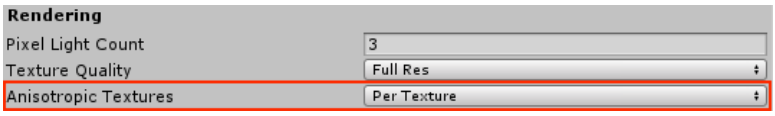


Fig. 14 - Setting texture filtering settings to per texture in Unity

In this window select the *Anisotropic Textures* dropdown and choose *Per Texture*.

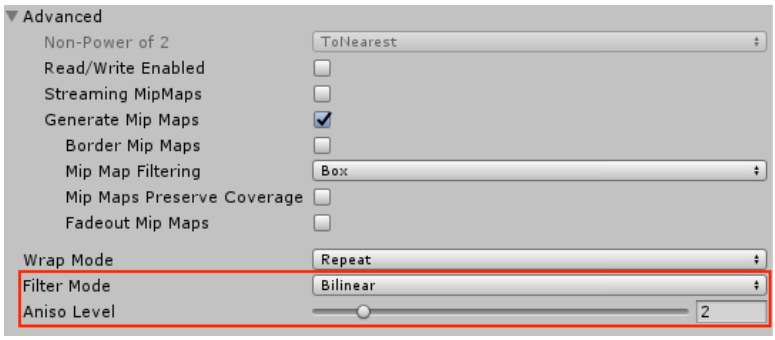


Fig. 15 - Choosing texture filter settings for a texture in Unity

Then for each texture, select it within the A*ssets* section of the *Project* window so that the *Texture Inspector* window opens. Once it is open, set the *Filter Mode* and *Aniso Level* setting appropriately for that texture, considering the guidance above.

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 but 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 so 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 yet enabling sorting is an expensive process and substantially diminishes performance.

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

The video below demonstrates the difference between a basic alpha test implementation, on the left, and an alpha to coverage implementation, on the right. In the alpha to coverage implementation there is considerably less aliasing and reduced flickering.

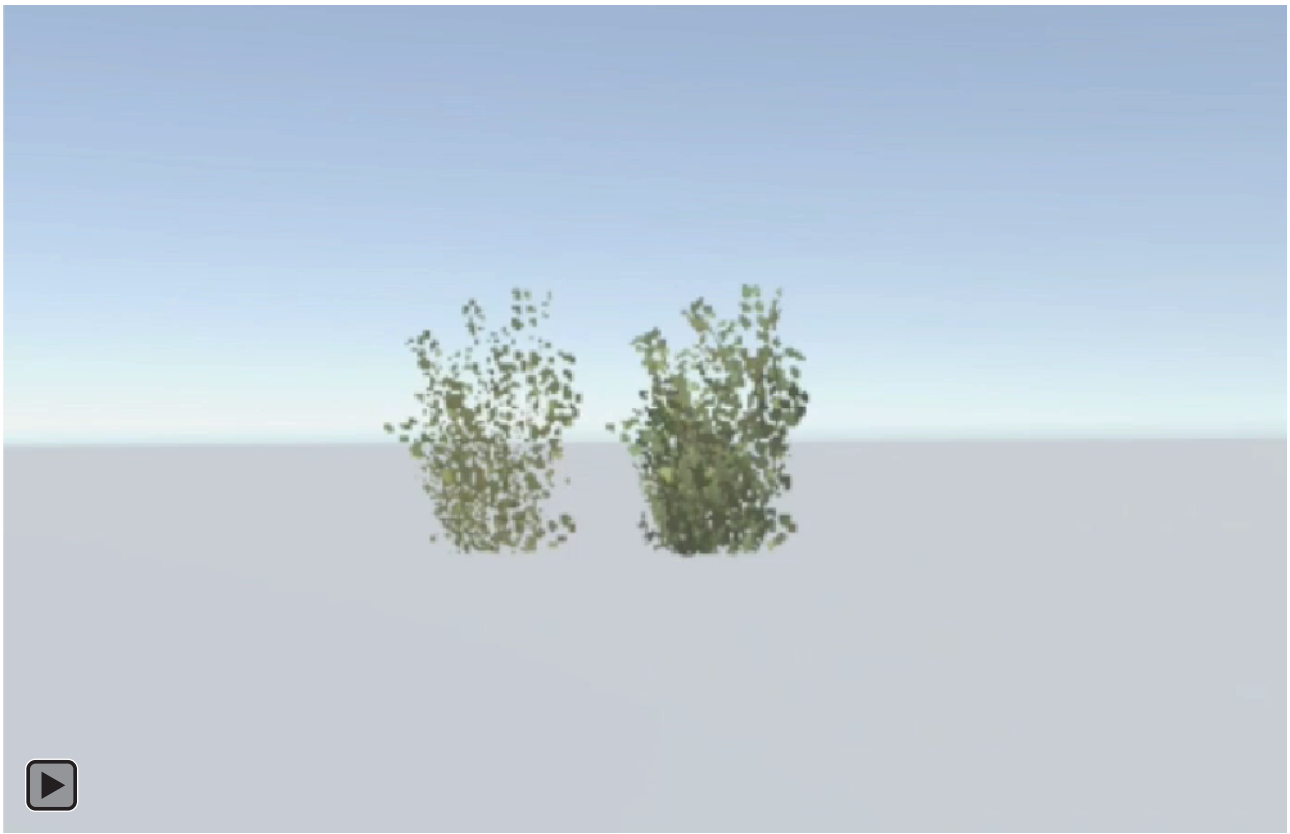


Fig. 16 - Video demonstrating alpha testing (left) and alpha to coverage (right)
If this video does not load, please click here: https://www.youtube.com/watch?v=9nBprtPhXEQ

Unity Implementation

To enable ATOC in Unity create a new shader in the *Project* window, then insert the following shader code[6].

```
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
        }
    }
}
```

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

Level Design

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

When creating geometry for a scene care should be taken when modeling the edges of meshes to avoid sharp or sudden edges, for example, a staircase has flat edges on each individual step each which can create aliasing if the viewer rotates their head but the individual steps themselves can also be the cause of aliasing when a staircase is viewed from a large distance each step will become a thin object in the distance which can flicker as the viewer moves.

An example of replacing a set of stairs where the edge of each stair produces aliasing with a ramp that produces significantly less aliasing is shown below.

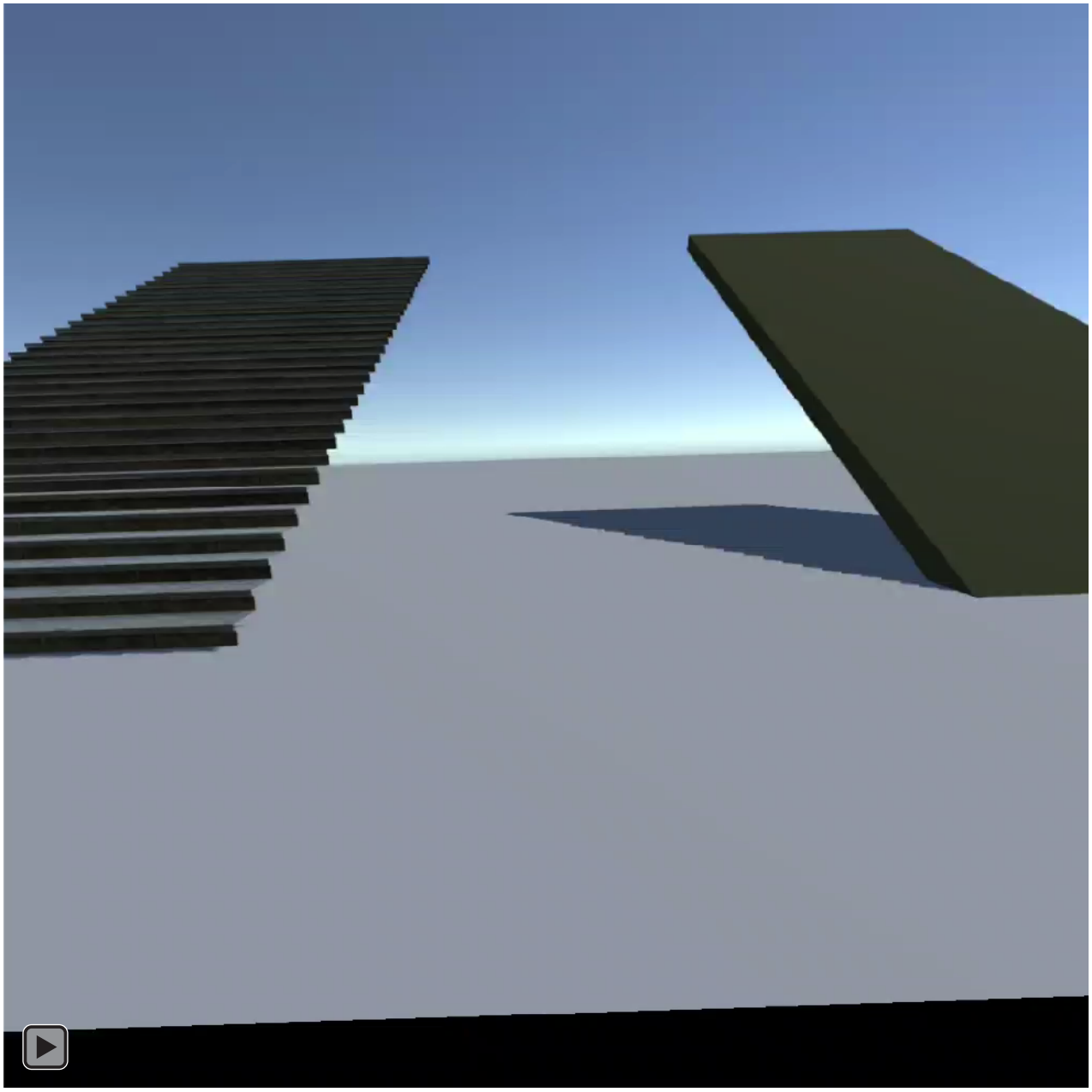


Fig. 17 - Video comparing aliasing on stairs and ramps
If this video does not load, please click here: https://www.youtube.com/watch?v=zqhSDKHuXYk

Whenever possible smooth, round shapes should be used in place of those with hard edges, bevels and thin objects such as wires or cables, have a propensity to cause substantial aliasing when they are viewed from a distance as they will be rendered effectively as lines, therefore causing aliasing.

Careful consideration should made on the use of metallic materials which should be minimized where possible as metallic objects produce specular effects when lit, these specular effects will flicker as the viewer moves resulting in aliasing, therefore matte materials should be utilized at length.

A comparison between the aliasing caused by metallic and matte materials is below, the top image shows a metallic material while the lower image shows a matte equivalent.

Fig. 18 - Comparison of metallic (top) and matte (bottom) materials

Lighting for a scene should be implemented with caution as bright, shiny lights will result in specular effects appearing on the lit objects which can produce aliasing but more importantly, aliasing produced from specular effects is particularly noticeable as the human eye will be drawn to the ‘flashing’ of the pixel as the specular effect appears and disappears between frames.

All lighting should be prebaked when the scene is built to avoid expensive real time lighting calculations and reduces the presence of lighting artifacts such as banding. To do this scenes should be designed so that they are appropriate for prebaking which requires few moving objects and no live day/night cycle. Utilizing light probes can help to minimize the quantity of prebaking required and they are especially advantageous in 6 degrees of freedom VR.

Reflections require more caution in VR than in typical uses as the complexity required for a technique such as screen space reflections is too extreme for VR therefore other techniques should be employed for reflections. Possible techniques include, reflection probes, cube maps[19] or for objects that require high quality reflections as they can be rendered inverted within the reflective surface.

To help conceal aliasing particle effects should be deployed such as, fog or smoke, techniques which have long been used to conceal short render distances, can also be used to conceal aliasing caused by objects in the distance.

An example of a scene before, on the left, and after, on the right, fog has been introduced is shown below to demonstrate the reduction of aliasing following the addition of a particle effect.

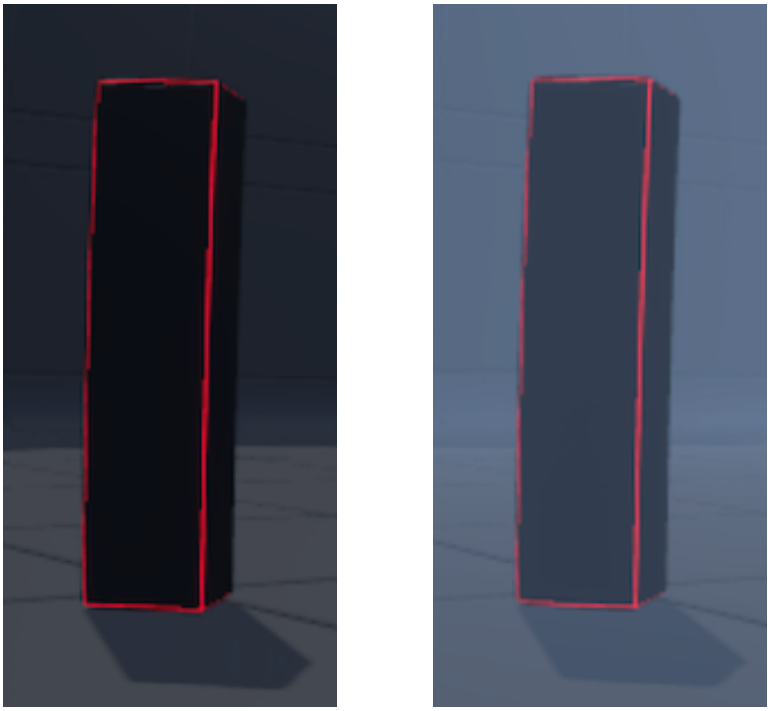


Fig. 19 - Comparison of before (left) and after particle effects (right)

Banding

Description

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

These bands are most apparent within VR as the user is immersed within the scene therefore their eyes adjust to the light levels within it, once this has occurred they the bands will be even more apparent. If there are many bands the user's eyes will be constantly adjusting to the changing light levels which can also be physically tiring.

Mitigations

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 will be broken up and disrupted therefore they will no longer be 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.

Unity Implementation

To enable dithering in Unity install the Unity Post Processing Stack v2[8] in your project, once this has been done create a new *Post Processing Profile* in the *Project Window*, by right clicking in the *Project Window* and under *Create*, choose *Post-processing Profile*.

Then, 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, then within the *Post Volume Volume*, set the created *Post Processing Profile* as the *Profile*.

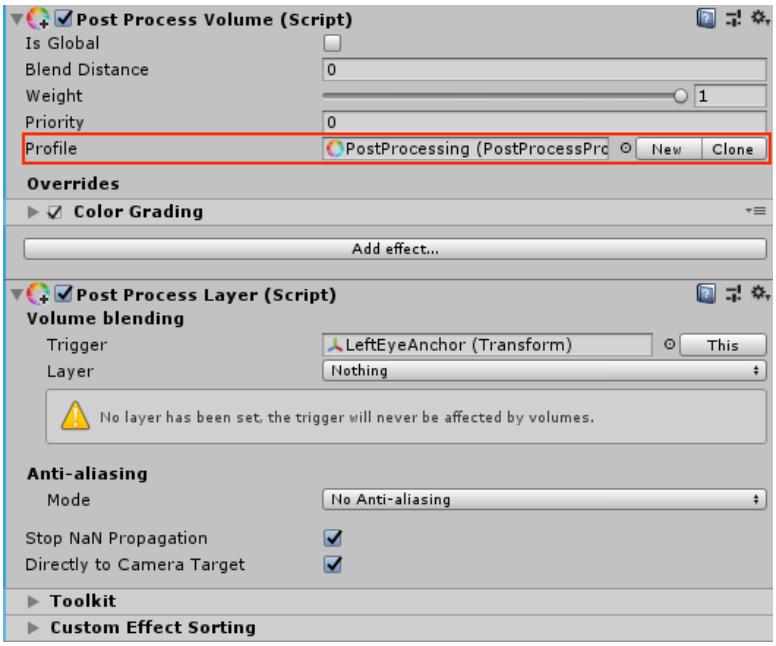


Fig. 20 - Setting a Post Processing Profile in Unity

Tone Mapping

Tone mapping 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 screen. This function uses a lookup table (LUT) through which each color is mapped to the appropriate corresponding color for the new tone.

By applying Tone Mapping to the scene banding caused by low lighting levels or sharp gradients in textures, which should be avoided where possible, can be reduced.

Unity Implementation

To enable tone mapping in Unity install the Unity Post Processing Stack v2[8] in your project, once this has been done create a new *Post Processing Profile* in the *Project Window*, by right clicking in the *Project Window* and under *Create*, choose *Post-processing Profile*.

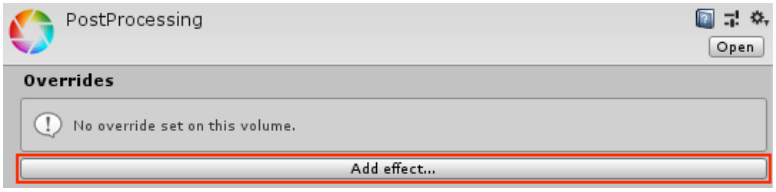


Fig. 21 - Adding a post processing effect in Unity

Then select this *Post-processing Profile* and in the *Inspector* select *Add effect…,* under *Unity* choose *Unity Color Grading* then set *Mode* to *High Definition Range* and set the *Mode* within the *Tonemapping* section to *ACES*[9].

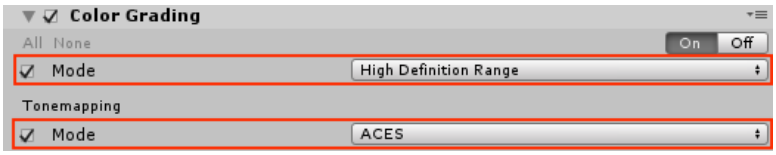


Fig. 22 - Adding a post processing effect in Unity

Then, 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 tone mapping, finally, within the *Post Volume Volume*, set the created *Post Processing Profile* as the *Profile*, see Fig. 20.

Bump Mapping

Description

Bump mapping is a widely adopted technique for reducing the vertex count of a mesh, this technique involves simulating the finer details such as bumps on the surface of an object. This simulation is performed by manipulating the normals of the object before using them for the lighting calculations.

While normal mapping is a highly robust technique for typical uses, it is not as effective in virtual reality as the user can easily change their viewing angle of a normal mapped texture. As this change of perspective is not accounted for within the normal mapping technique the illusion of depth is broken.

Additionally, normal mapping cannot account for the use of stereoscopic lenses used in virtual reality headsets as the normal is only generated from one viewpoint therefore each eye receives the same normal which looks incorrect to the human eye.

Below is a video demonstrating the appearance of a standard material, a normal mapped material and a material with parallax occlusion mapping.

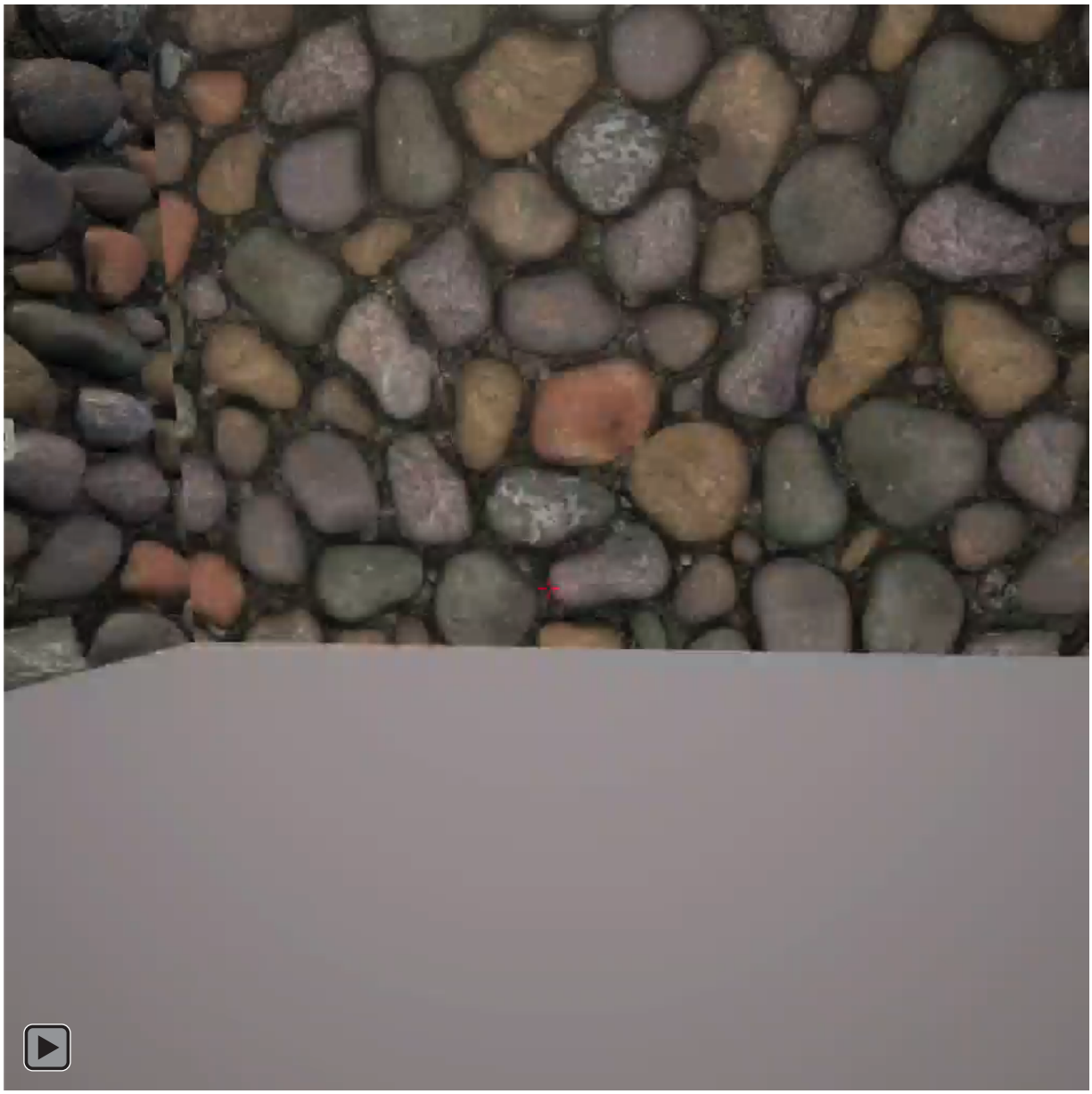


Fig. 23 - Video comparing different bump mapping methods
If this video does not load, please click here: https://www.youtube.com/watch?v=IQvNLr2Eqoc

Mitigations

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 the high polygon count version with the normals of the finer details being stored in the normal map texture.

During rendering the fragment shader samples from the normal map, generating normals from the sampled values. These generated normals are combined with the surface normals of the low polygon version before being used in lighting calculations. The lighting then shows the finer surface details without needing to render the individual vertices of theses details.

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

Unity Implementation

To add Normal Mapping to a material in Unity, select the material in the *Project Window* then open the *Material Inspector Panel* and then select a shader that includes support for normal mapping, such as *Mobile/Bumped Diffuse*, then set the required *Normalmap* texture[10].

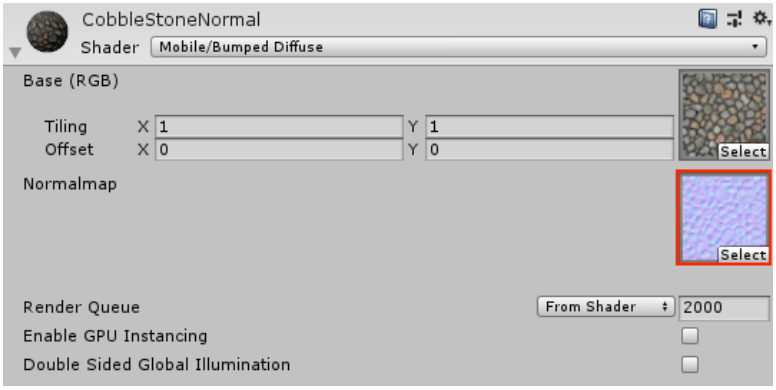


Fig. 24 - Adding a normal map texture in Unity

Parallax Occlusion Mapping

Parallax occlusion mapping is a technique similar to normal mapping but differs as 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, therefore maintaining the illusion of depth.

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

Unity Implementation

To add Parallax Occlusion Mapping to a material in Unity, select the material in the *Project Window* then open the *Material Inspector Panel* and select a shader that supports Parallax Diffuse mapping such as the *Standard* shader[11].

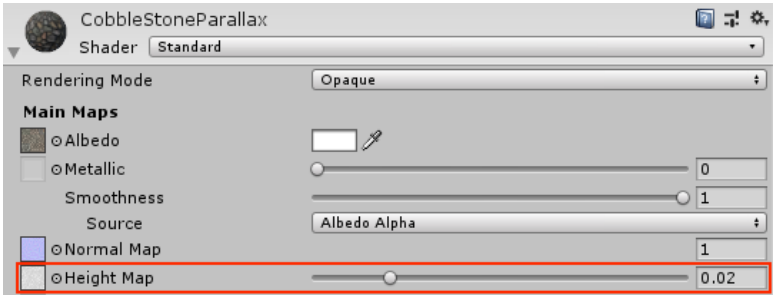


Fig. 25 - Adding a normal map texture in Unity

Then set the required *Albedo*, *Normal Map* and *Height Map* texture.

Shadows

Description

Traditional methods of shadow mapping are computationally intensive on mobile as they require additional framebuffers and render passes which causes a serious decrease in performance.

Therefore shadow buffers on mobile are often low resolution and do not have any filtering implemented, as a result of this the shadows often introduce large amounts of aliasing and damage the realism of the simulation due to artifacts such as shadow acne and hard shadows.

Below is a screenshot demonstrating the comparison between typical shadows and a blob shadows.

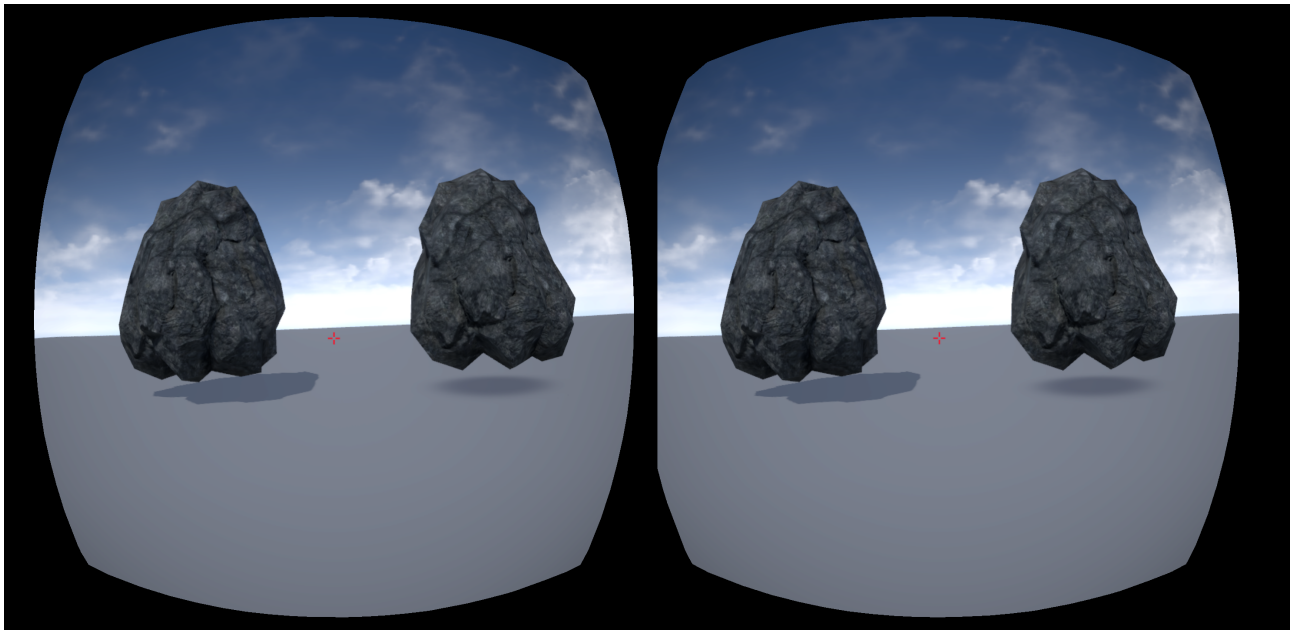


Fig. 26 - Comparison of shadow mapping (left) and blob shadows (right)

Mitigations

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, blob shadows should be rendered below objects. These blob shadows produce considerably less aliasing than using typical shadow mapping techniques and also give a performance improvement.

Unity Implementation

To implement blob shadows, first import the *Unity Standard Assets* from the *Unity Asset Store*[13], then navigate to the *BlobShadowProject* prefab within *Assets/StandardAssets/Effects/Projects/Prefabs* and drag this prefab into the *Hierarchy* as a child of the object that requires a blob shadow[12].

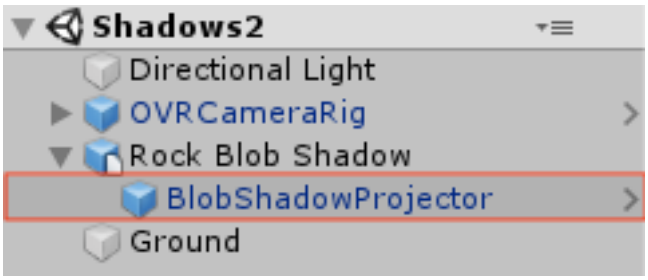


Fig. 27 - Setting a child object in Unity

Once this is done a blob shadow will appear, to customize its appearance edit the prefab options including its location relative to the object that should be casting the shadow.
Finally, select the object that is casting the shadow and set the *Cast Shadows* option to *Off*.

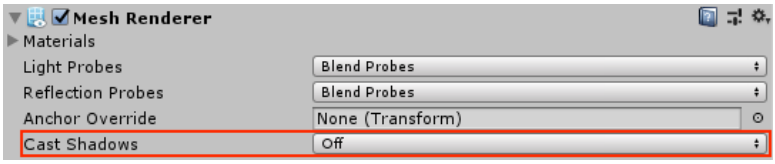


Fig. 28 - Disabling *Cast Shadows* for an object in Unity

Summary

This document has been produced in the hope that the techniques and methods described will aid developers in the process of improving and refining the visual fidelity of mobile virtual reality games.

Some techniques, for example multisample anti-aliasing and texture filtering are easy to implement and have little to no impact on performance making them vital in mobile virtual reality. While other techniques, such as bump shadows may not be as useful for all developers due to their impact on the art style they should still be considered given the benefits they provide.

To provide any feedback or suggestions on this document, please email: developer@arm.com

Acknowledgements

This document was produced from a collaboration between Arm, Unity and Bangor University.

Firstly, thank you to Roberto Lopez Mendez for the invaluable guidance and feedback he provided during the creation of this document and for his effort in creating and proposing this topic.

Thank you to AC Mahendran and Brandon Fogerty from Unity for reviewing the document and the suggestions they gave.

Thank you also to Sam Martin, Pete Harris, Andreas Loeve Selvik and Christian Forfang from Arm for taking the time to review and provide feedback on the document.

Finally, thank you to my supervisor Llyr ap Cenydd from Bangor University for his advice during the creation of this document.

References

[1] - https://docs.unity3d.com/540/Documentation/Manual/class-TextureImporter.html
[2] - https://docs.unity3d.com/530/Documentation/Manual/class-LODGroup.html
[3] - https://docs.unity3d.com/Manual/SL-ShaderLOD.html
[4] - https://docs.unity3d.com/Manual/class-QualitySettings.html
[5] - https://docs.unity3d.com/Manual/LinearRendering-LinearOrGammaWorkflow.html
[6] - https://medium.com/@bgolus/anti-aliased-alpha-test-the-esoteric-alpha-to-coverage-8b177335ae4f
[7] - https://developer.oculus.com/blog/tech-note-shader-snippets-for-efficient-2d-dithering/
[8] - https://github.com/Unity-Technologies/PostProcessing/wiki
[9] - https://github.com/Unity-Technologies/PostProcessing/wiki/Color-Grading
[10] - https://docs.unity3d.com/Manual/StandardShaderMaterialParameterNormalMap.html
[11] - https://docs.unity3d.com/Manual/StandardShaderMaterialParameterHeightMap.html
[12] - https://docs.unity3d.com/Manual/class-Projector.html
[13] - https://assetstore.unity.com/packages/essentials/asset-packs/standard-assets-32351
[14] - http://www2.egr.uh.edu/~glover/applets/Sampling/Sampling.html
[15] - http://mathworld.wolfram.com/MoirePattern.html
[16] - https://community.arm.com/developer/tools-software/graphics/b/blog/posts/white-paper-foveated-rendering
[17] - https://learnopengl.com/Advanced-Lighting/Gamma-Correction
[18] - https://learnopengl.com/PBR/Theory
[19] - https://community.arm.com/developer/tools-software/graphics/b/blog/posts/achieving-high-quality-mobile-vr-games
[20] - https://www.facebook.com/permalink.php?story_fbid=1818885715012604&id=100006735798590
[21] - https://pages.arm.com/arvr-report.html