Using Virtual Reality in Unity

Version 1.0

Using Virtual Reality in Unity

Release Information

Document History

Version	Date	Confidentiality	Change
1.0	05 June 2020	Non-confidential	First release.

Non-Confidential Proprietary Notice

Confidentiality Status

Product Status

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

Web Address

http://www.arm.com

Contents

1 Overview

This guide explains how you can use Unity alongside and Virtual Reality (VR). This guide is useful if you want to port a non-VR application into a VR application. The guide also shows the important of reflections, and how to implement reflections into VR.

Most Virtual Reality (VR) hardware has Unity support. Many use OpenVR which in turn requires SteamVR to be installed. All devices need an appropriate runtime, for example Steam or Oculus.

Individual setup varies, consult your product documentation to be sure that all correct parameters are set for the devices that you want to be supported.

1.1. Before you begin

This guide uses the Samsung Gear VR as an example device. Some of the instructions in this guide are specific to the Samsung Gear VR. The principles that are described in this guide apply to any VR device, but you may need to adapt these instructions for your environment.

Throughout the guide, there are references to an Ice Cave demo. There are images from the demo throughout this guide showing you the output from the VR device.

By the end of this guide, you will have a better understanding of the Unity VR porting process and how this is done and how reflections can be implemented.

2 The Unity VR porting process

In this section of the guide, we show you how to port an application or game to native Unity VR. This is a multi-stage process. Follow these steps:

1. Install a recent version of Unity. The most up to date version is best; Unity 5.1 is required for native VR support; 2018.3 adds significant support including when targeting multiple VR platforms.
2. Install the appropriate SDK for each device that you would like to be supported. Devices may also need a signature file that you can download from the relevant website. Place the signature file in the required folder on your device.

   For example, the Samsung Gear VR that runs the Ice Cave demo needs a signature file from the Oculus developer website. The signature file for a Samsung device must go in the `Plugins/Android/assets` folder. Configure the relevant Oculus SDK. Use SDK Suite 1.41 or earlier - Gear support has ended in the most recent Oculus SDK.

Note: From Unity 2018 onwards, you could use the Experimental VR Lightweight Render Pipeline, but more recent versions want the Universal Render Pipeline.

3. Enable Virtual Reality support in **Edit > Project Settings > Player > XR Settings**.

   Note: Currently Unity does not support Vulkan with VR, so set your Graphics API to OpenGL ES3.

   The following screenshot shows the Player Settings window:

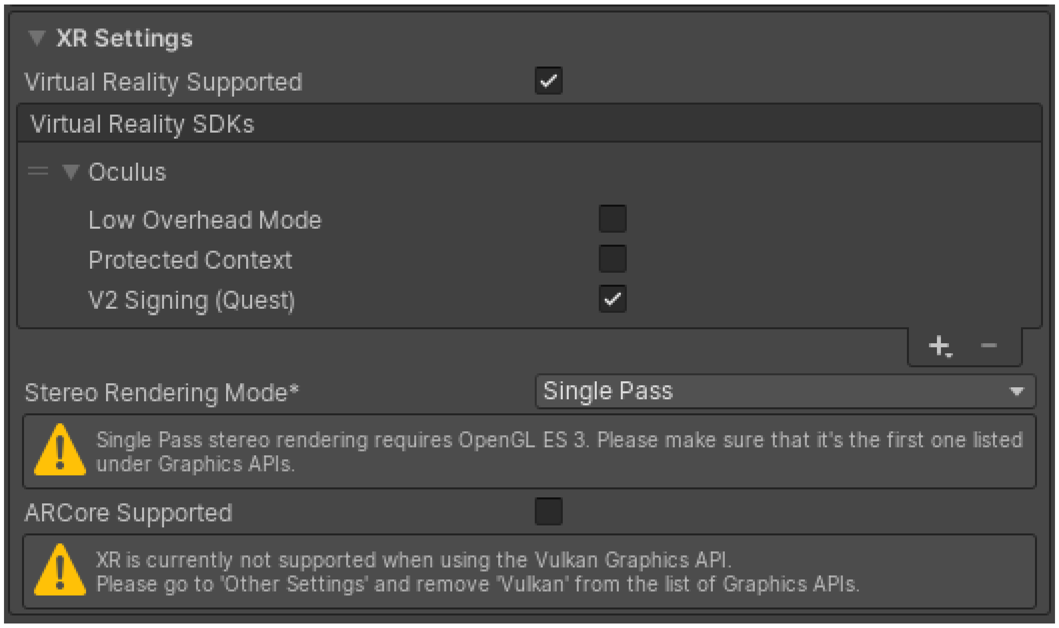
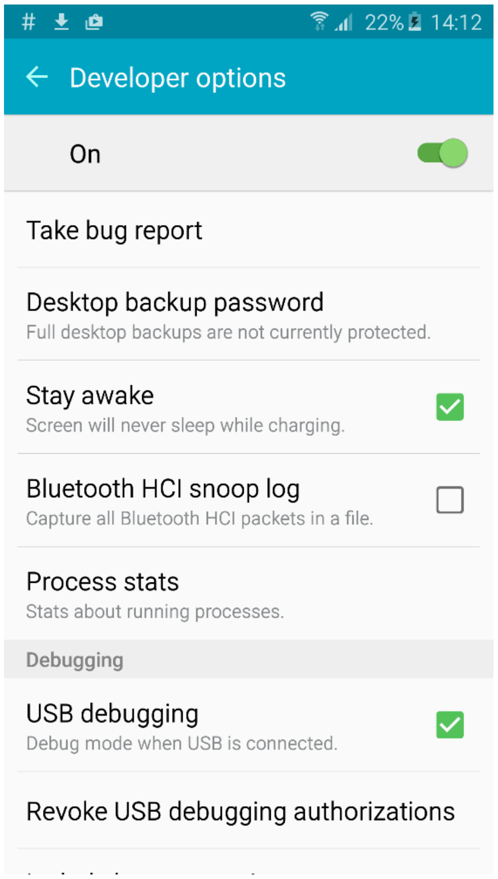

XR Settings
Virtual Reality Supported ✓
Virtual Reality SDKs
Oculus
Low Overhead Mode ☐
Protected Context ☐
V2 Signing (Quest) ✓
Stereo Rendering Mode* Single Pass
Single Pass stereo rendering requires OpenGL ES 3. Please make sure that it's the first one listed under Graphics APIs.
ARCore Supported ☐
XR is currently not supported when using the Vulkan Graphics API.
Please go to 'Other Settings' and remove 'Vulkan' from the list of Graphics APIs.

Figure 1. The Player Settings window

4. Set up the camera. Most VR systems have a Prefab set up that you need to add. You also need to remove the default Main Camera.
5. Enable the Developer options menu for Android devices and turn on USB debugging. The following screenshot shows the Developer options menu:


22% 14:12

Developer options

On ✓

Take bug report

Desktop backup password
Full desktop backups are not currently protected.

Stay awake ✓
Screen will never sleep while charging.

Bluetooth HCI snoop log ☐
Capture all Bluetooth HCI packets in a file.

Process stats
Stats about running processes.

Debugging

USB debugging ✓
Debug mode when USB is connected.

Revoke USB debugging authorizations

Figure 2. The Developer options menu

6. Build the application, connect the device, and choose to run on the device in Unity.
7. Launch the application.

2.1. Enable Samsung Gear VR developer mode

When you launch the application on a Samsung device, it prompts you to insert the device into the headset. If the device is not ready for VR, it prompts you to connect to the network to download the Samsung VR software. Samsung Gear VR developer mode can help you to visualize the VR application that is running, without inserting the device into the VR headset.

Samsung Gear VR Developer mode can be enabled if you have already installed a signed VR application. If you have not installed a signed VR application, install a signed VR application so that you can enable this mode. To enable developer mode:

1. On the Samsung device, select Settings > Application Manager > Gear VR Service.
2. Select Manage Storage.
3. Tap on the VR service version six times.
4. Wait for the scan process to complete. A developer mode toggle appears.

Note: Using developer mode reduces the battery life of the phone. This is because it overrides all the settings that turn off the headset when it is not in use.

The following figure shows an example Ice Cave screenshot from the Samsung Gear VR developer mode view:

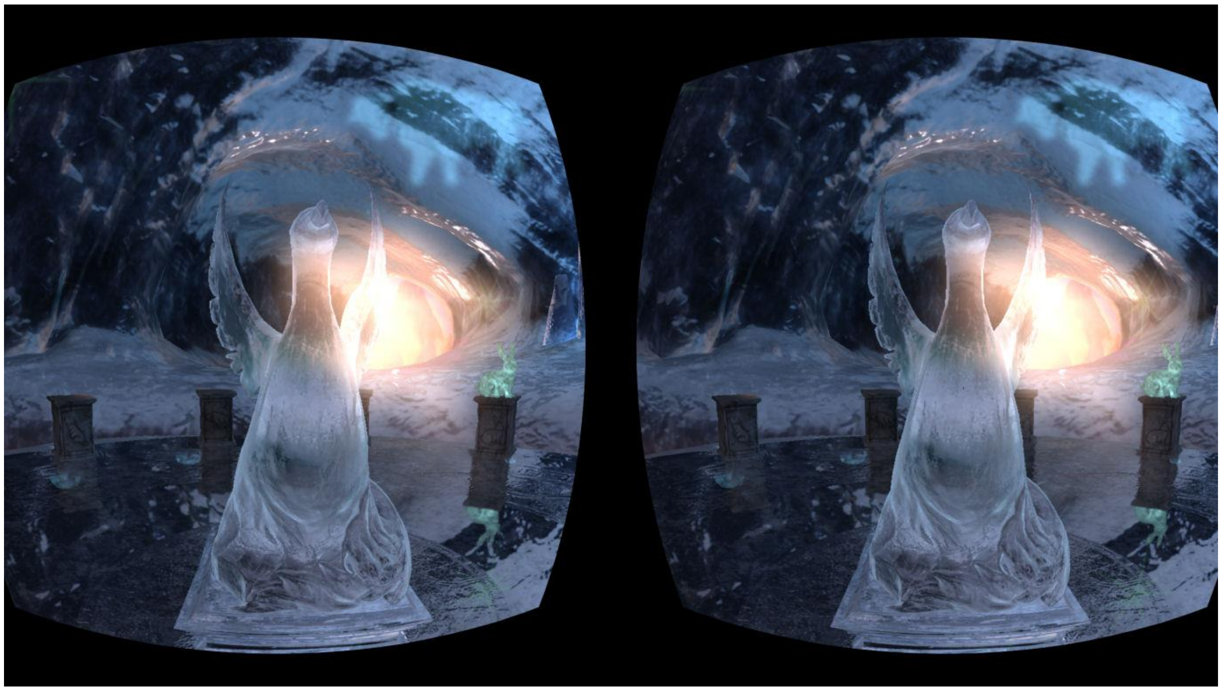


Figure 3. An example screenshot from the VR application running in Samsung Gear VR developer mode

3 What to consider when porting to VR

In this section of the guide, we describe what you must consider when determining whether you should port a game or an application to VR.

VR creates a very different user experience compared to other application types. This means that some factors that work for non-VR applications or games do not work for VR. One of ways that you could see this difference is to test the application on a few different users to determine their comfort levels. Then adjust the code so that they find the experience comfortable.

For example, camera animations that are comfortable in a non-VR game might be uncomfortable in a VR version of the game. For example, the Ice Cave demo without VR has an animation mode for the camera. Some users find this uncomfortable and suffer motion sickness, especially when the camera moves backwards. Removing this mode prevents this unsettling experience.

Another example of what works in a non-VR application, but do not work in a VR application is visual effects. This can be seen in the non-VR Ice Cave demo which uses a dirty lens effect that changes its intensity based on the camera alignment with the sun. Users testing this effect in VR found that it looked wrong, so it was removed.

3.1. Controllers

Users experience differences when controlling an application on a VR or a non-VR application, there are differences. This is because a non-VR application can be controlled using the touch screen of the phone or by tilting the phone.

These control mechanisms might not be possible in a VR application. For example, the non-VR Ice Cave demo uses two virtual joysticks to control the camera. This does not work on a VR device, because the touchscreen is not accessible. The Ice Cave VR demo is designed to run on Samsung Gear VR, which has a touchpad on the side of the headset. Using the touchpad instead of the touch screen solves this problem.

The following figure shows the touchpad on a Samsung Gear VR headset:

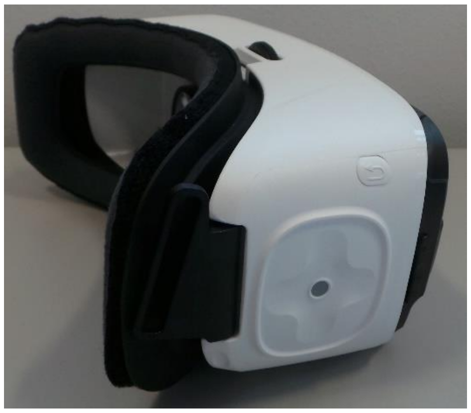


Figure 4. A Samsung Gear VR headset showing the touchpad

VR can benefit from control methods that are not normally associated with phones and mobile devices. Some of these methods are worth considering, depending on the target audience for the application.

Users can use controllers to connect to the VR device using Bluetooth. In this situation, the Ice Cave demo uses a custom plug-in that extends the Unity functionality to interpret the Android Bluetooth events. These events trigger movement of the camera.

The following image shows a person holding the Bluetooth controller that works with the Ice Cave demo:

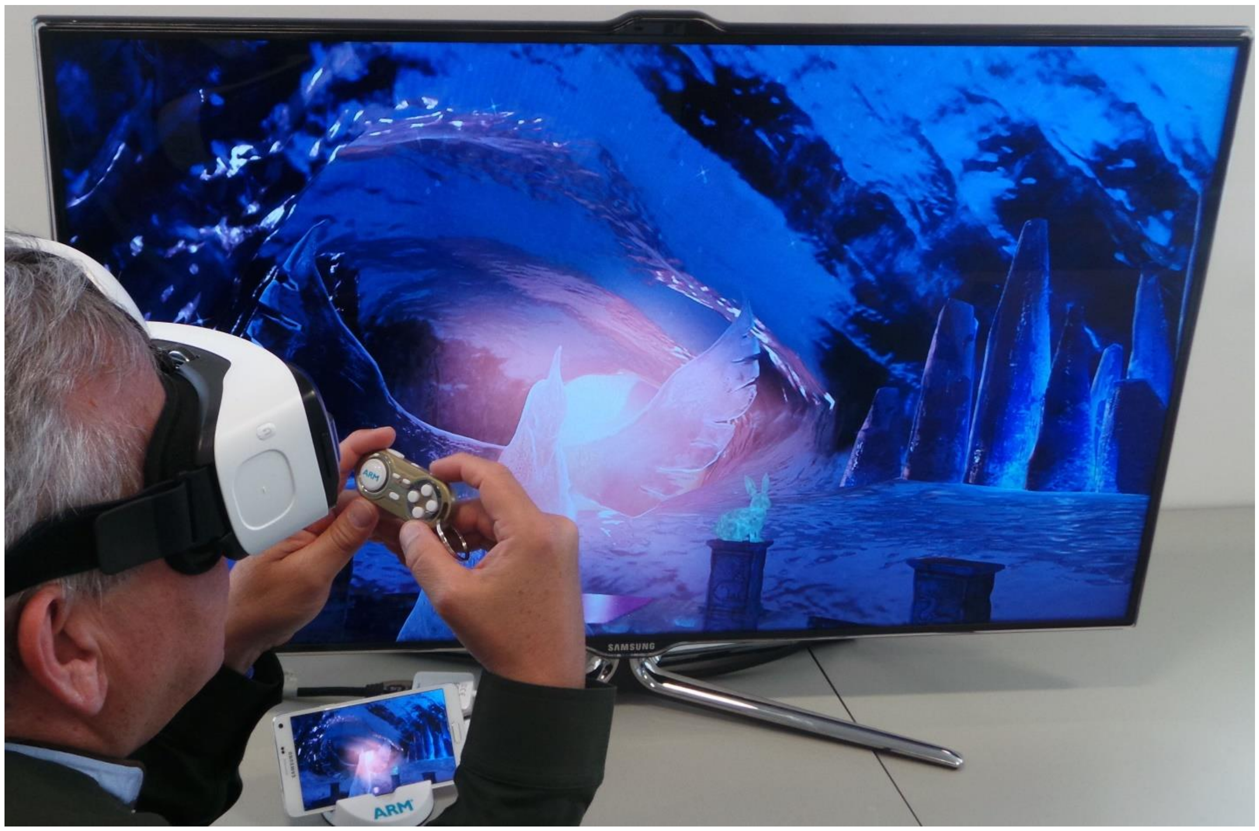


Figure 5. Navigating the Ice Cave demo with a Bluetooth controller.

4 Reflections in VR

Reflections are important in any VR application. The real world contains many reflections, so people notice when they are missing from a game or application. In this section of the guide, we focus on why reflections are important in VR and the different ways that these reflections can be implemented.

Reflections in VR can use the same techniques that traditional games use. However, the reflections must be modified to work with the stereo visual output that a user sees.

A good implementation of reflections can make an application or game feel more realistic and immersive. However, there are a few extra issues that you must consider when you implement reflections in VR. We explain these issues in this section of the guide.

4.1. Reflections using local cubemaps

One of the ways that you can create reflections is to use cube mapping. In a cubemap, the six faces of a cube are used as the map shape. The six sides of the cubemap have the environment projected onto them. The cubemap is stored as either six square textures, or is unfolded into six regions of a single texture. The cubemap is generated by rendering the scene from a given position with six different camera orientations. A 90-degree view frustum represents each cube face.

The use of local cubemaps avoids creating the reflection texture for each frame. Instead, this method fetches the reflection texture from a pre-rendered cubemap. This method applies a local correction to the reflection vector, based on where the cubemap was generated. The scene bounding box uses that information to fetch the correct texture.

Reflections that are generated using local cubemaps do not suffer from pixel instability or pixel shimmering. Pixel instability and pixel shimmering can occur when a reflection is generated at runtime for each frame. For more information, see Implementing reflections with a local cubemap.

4.2. Combine different types of reflection

Different reflection generation techniques are required to achieve the best performance and effect. Which technique is required depends on the shape of the reflective surface, and whether the reflective surface and the object being reflected are static or dynamic? The result from the different reflection generation techniques must be combined to produce the result that the user sees. For example, in the Ice Cave demo, some reflections use a static cubemap, but the dynamic objects are rendered using a mirrored camera. These reflections are combined in a single shader. For more information, see Combining reflections.

5 Stereo reflections

This section of the guide shows you how to implement stereo planar reflections in Unity VR. To do this in your VR game, you must make a few adjustments to the non-VR game code.

In a non-VR game, there is only one camera viewpoint. In VR, there is one camera viewpoint for each eye. This means that the reflection must be computed individually for each eye.

If both eyes are shown the same reflection, users quickly notice that there is no depth in the reflections. this lack of depth is inconsistent with their expectations and can break their sense of immersion, negatively affecting the quality of the VR experience.

To correct this problem, two reflections must be calculated. These two reflections must be shown with the correct adjustment for the position of each eye while the user looks around in the game.

To implement stereo reflections the Ice Cave demo uses:

- Two reflection textures for planar reflections from dynamic objects
- Two different local corrected reflection vectors, to fetch the texture from a single local cubemap for static object reflections

Reflections can be for either dynamic objects or static objects. Each type of reflection requires a different set of changes to work in VR.

5.1. Implement stereo planar reflections in Unity VR

Before following the code in this section, you must ensure that you have enabled support for virtual reality in Unity. To check this, follow these steps:

1. Select **Edit.**
2. Select **Project Settings.**
3. Select **Player.**
4. Select **XR Settings.**
5. Select the checkbox for **Virtual Reality Supported.**

5.2. Dynamic stereo planar reflections

Dynamic reflections require some changes to produce a correct result for two eyes.

The following code shows how to set up two new cameras that both have a target texture to render to. You must disable both cameras so that their rendering is executed programmatically:

```
void OnPreRender(){
        SetUpReflectionCamera();
        // Invert winding
        GL.invertCulling = true;
}
void OnPostRender(){
        // Restore winding
        GL.invertCulling = false;
}
```

This script places and orients the reflection camera using the position and orientation of the main camera. To do this, the code calls the `SetUpReflectionCamera()` function just before the left and right reflection cameras render. The following code shows how `SetUpReflectionCamera()` is implemented:

```
public GameObject reflCam;
public float clipPlaneOffset ;
…
private void SetUpReflectionCamera(){
        // Find out the reflection plane: position and normal in world space
        Vector3 pos = gameObject.transform.position;

        // Reflection plane normal in the direction of Y axis
        Vector3 normal = Vector3.up;
        float d = -Vector3.Dot(normal, pos) - clipPlaneOffset;
        Vector4 reflPlane = new Vector4(normal.x, normal.y, normal.z, d);
        Matrix4x4 reflection = Matrix4x4.zero;
        CalculateReflectionMatrix(ref reflection, reflPlane);

        // Update reflection camera considering main camera position and orientation
        // Set view matrix
        Matrix4x4 m = Camera.main.worldToCameraMatrix * reflection;
        reflCam.GetComponent().worldToCameraMatrix = m;

        // Set projection matrix
        reflCam.GetComponent().projectionMatrix = Camera.main.projectionMatrix;
}
```

`SetUpReflectionCamera()` calculates the view and projection matrices of the reflection camera. `SetUpReflectionCamera()` determines that the reflection transformation `worldToCameraMatrix` must be applied to the view matrix of the main camera.

To set the position of the cameras for each eye, add the following code after the line `Matrix4x4 m = Camera.main.worldToCameraMatrix * reflection:`

Left eye	Right eye
`m[12] += stereoSeparation;`	`m[12] -= stereoSeparation;`

The shift value `stereoSeparation` is 0.011. The `stereoSeparation` value is half the eye separation value.

Attach another script to the main camera, to control the rendering of the left and right reflection cameras. The following code shows the Ice Cave implementation of this script:

```
public class RenderStereoReflections : MonoBehaviour
{
     public GameObject reflectiveObj;
     public GameObject leftReflCamera;
     public GameObject rightReflCamera;
     int eyeIndex = 0;

     void OnPreRender(){
     if (eyeIndex == 0){
            // Render Left camera
            leftReflCamera.GetComponent().Render();
            reflectiveObj.GetComponent().material.SetTexture(
            "_DynReflTex",leftReflCamera.GetComponent().targetTexture);
     }
     else{
            // Render right camera
            rightReflCamera.GetComponent().Render();
            reflectiveObj.GetComponent().material.SetTexture( "_DynReflTex",
            rightReflCamera.GetComponent().targetTexture);
     }
     eyeIndex = 1 - eyeIndex;
   }
}
```

This script handles the rendering of the left and right reflection cameras in the `OnPreRender()` callback function of the main camera. This script is called once for the left eye and once for the right eye. The `eyeIndex` variable assigns the correct render order for each reflection camera and applies the correct reflection to each eye of the main camera. The first time the callback function is called, `eyeIndex` specifies that rendering is performed for the left eye. This matches the order that Unity calls the `OnPreRender()` method.

5.3. Check that different textures are in use for each eye

Checking whether the script is correctly producing a different render texture for each eye is important.

To test whether the correct texture is being shown for each eye, follow these steps:

1. Change the script so that it passes the `eyeIndex` value to the shader as a uniform.
2. Use two colors for the reflection textures, one for each `eyeIndex` value.

If your script is working correctly, the output should look similar to the following screenshot. This screenshot shows two different well-defined left and right textures on the platform. This means that, when the shader is used to render with the left camera, the correct left reflection texture is used, this is also the case when the shader is used to render the right camera.

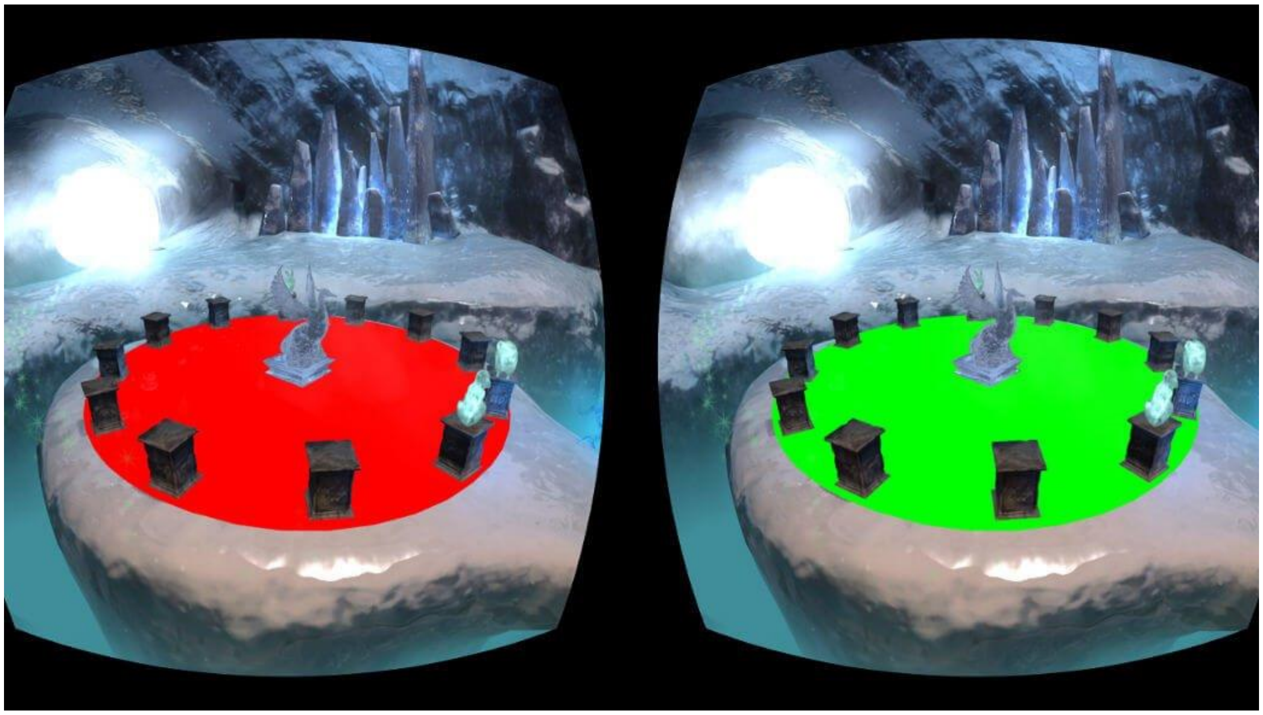


Figure 6. An example of a correct reflection texture output check

5.4. Static stereo reflections

You can use cubemaps to efficiently create stereo reflections from static objects. In this case, you must use two different reflection vectors to fetch the texels from the cubemap, one for the left camera and one for the right camera. Unity provides a built-in value to access the camera position in world coordinates, in the shader: `_WorldSpaceCameraPos`.

However, in VR, the position of the left and right cameras is required. `_WorldSpaceCameraPos` cannot provide the positions of the left and right cameras. This means that you must use a script to calculate the position of the left and right cameras, and to pass the results to the shader as a single uniform.

The following code shows how to declare a new uniform in the shader that can pass the information for the camera positions:

```
uniform float3 _StereoCamPosWorld;
```

The best place to calculate the left and right camera positions is in the script that is attached to the main camera. This script gives easy access to the main camera view matrix. The following code shows how to do this for the `eyeIndex = 0` case.

The code modifies the view matrix of the main camera to set the position of the left eye in local coordinates. The left eye position is required in world coordinates, so that the inverse matrix is found. The left eye camera position is passed to the shader through the uniform `_StereoCamPosWorld`.:

```
Matrix4x4 mWorldToCamera = gameObject.GetComponent().worldToCameraMatrix;
mWorldToCamera[12] += stereoSeparation;
Matrix4x4 mCameraToWorld = mWorldToCamera.inverse;
```

```
Vector3 mainStereoCamPos = new Vector3(mCameraToWorld[12], mCameraToWorld[13],
      mCameraToWorld[14]);
reflectiveObj.GetComponent().material.SetVector("_StereoCamPosWorld", new Vector3
      (mainStereoCamPos.x, mainStereoCamPos.y, mainStereoCamPos.z));
```

The code is the same for the right eye, except that the stereo separation is subtracted from `mWorldToCamera[12]`, instead of being added to `mWorldToCamera[12]`.

In the vertex shader, you must find the following line of code, which is responsible for calculating the view vector:

```
output.viewDirInWorld = vertexWorld.xyz - _WorldSpaceCameraPos;
```

Replace the preceding line of code with the following line of code. The following line of code uses the new left and right eye camera positions in world coordinates:

```
 output.viewDirInWorld = vertexWorld.xyz - _ StereoCamPosWorld;
```

When the stereo reflection is implemented, the stereo reflection is visible when the application runs in editor mode. This is because the reflection texture flickers as it repeatedly changes from the left eye to the right eye. This flickering is not visible in the VR device, because a different texture is used for each eye.

5.5. Optimize stereo reflections

Without further optimizations, the stereo reflection implementations run all the time. This means that processing time is wasted on reflections when they are not visible.

The following code checks whether a reflective surface is visible, before any work is performed on the reflections themselves:

```
public class IsReflectiveObjectVisible : MonoBehaviour
{
      public bool reflObjIsVisible;

      void Start(){
             reflObjIsVisible = false;
      }

      void OnBecameVisible(){
             reflObjIsVisible = true;
      }

      void OnBecameInvisible(){
             reflObjIsVisible = false;
      }
}
```

After defining the **`IsReflectiveObjectVisible`** class, use the following if statement in the script that is attached to the main camera. This statement allows the calculations for stereo reflections to only execute when the reflective object is visible:

```
void OnPreRender(){
      if (reflectiveObjetc.GetComponent().reflObjIsVisible){
      …
      }
}
```

The rest of the code goes inside the if statement. The if statement uses the class `IsReflectiveObjectVisible` to check whether the reflective object is visible. If it is not visible, then the reflection is not calculated.

6 Outcome

Using the information that we have gathered in The Unity VR porting process, What to consider when porting to VR, Reflections in VR and Implementing stereo planar reflections in Unity VR, we can create a VR version of your game that implements stereo reflections. These reflections contribute to the sense of immersion, which improves the VR user experience. When stereo reflections are not implemented, people notice that depth is missing from the reflections.

The following screenshot from the Ice Cave demo, running in developer mode, shows the stereo reflections that are implemented. You can see these reflections in the pool in the foreground:

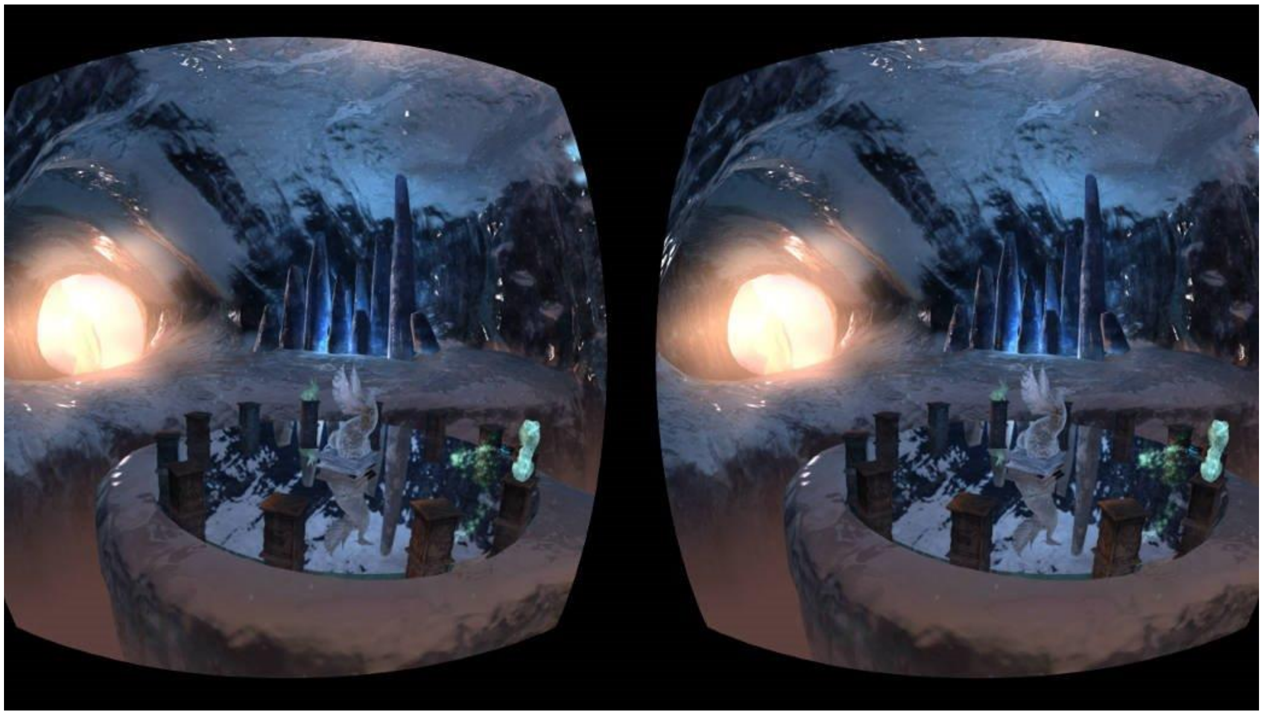


Figure 7. A screenshot from the Ice Cave demo

7 Related information

Here are some resources related to material in this guide:

- Arm architecture and reference manuals
- Arm Community - Ask development questions and find articles and blogs on specific topics from Arm experts.

Here are some resources related to topics in this guide:

- Arm Guide for Unity Developers Optimizing Mobile Gaming Graphics
- Advanced graphics techniques
- Oculus developer website
- Understanding Frustum
- Unity at Arm
- 2018.3 adds significant support

8 Next steps

After reading this guide, you should understand how to use different reflections in Unity VR. We introduced the porting process of an application to native VR and showed how to implement reflections in Unity VR. We explained stereo reflections, how to optimize them, and how they might look like in the Ice Cave demo.

To keep learning about VR, see more of our VR tutorials.