EG UK Computer Graphics & Visual Computing (2019)
G. K. L. Tam and J. C. Roberts (Editors)

# Optimising Underwater Environments for Mobile VR

L. ap Cenydd[1] & C. Headleand[2]

[1]School of Computer Science & Electronic Engineering, Bangor University, UK
[2]School of Computer Science, Lincoln University, UK

**Abstract**

*Mobile Virtual Reality (VR) has advanced considerably in the last few years, driven by advances in smartphone technology. There are now a number of commercial offerings available, from smartphone powered headsets to standalone units with full positional tracking. Similarly best practices in VR have matured quickly, facilitating comfortable and immersive VR experiences. There remains however many optimisation challenges when working with these devices, as while the need to render at high frame rates is universal, the hardware is limited by both computational power and battery capacity. There is also often a requirement that apps run smoothly across a wide variety of headsets. In this paper, we describe lessons learned in rendering and optimising underwater environments for mobile VR, based on our experience developing the popular aquatic safari application 'Ocean Rift'. We start by analyzing essential best practices for mobile app development, before describing low-cost techniques for creating immersive underwater environments. While some techniques discussed are universal to modern mobile VR development, we also consider issues that are unique to underwater applications.*

**CCS Concepts**
• ***Human-centered computing*** → ***Virtual reality;*** • ***Computing methodologies*** → *Procedural animation;*

## 1. Introduction

The revival and popularity of consumer VR in recent years has required a new set of standards and best practices be developed. An essential requirement is to design and optimise apps so that they maintain a steady frame rate of either 60 or 72 frames per second (fps) on mobile hardware, allowing for a comfortable and immersive experience. Indeed not adhering to these frame rates is often a blocker to publishing on app stores. However maintaining frame rates on mobile hardware is only one consideration. An app that runs smoothly can still benefit greatly from further optimisation. Similar to all mobile apps, reducing the load on the CPU or GPU can extend battery life. Conversely, if conserving battery is of less concern, knowing how to effectively optimise mobile VR applications can facilitate more advanced rendering, and more complex environments and game play mechanics.

In this viewpoint paper we will discuss how evolving optimisation strategies were used to construct an immersive underwater environment for Ocean Rift, one of the most popular applications in mobile VR over the past 5 years [Oce19], Figure 1. We will concentrate on aspects of rendering performance, including lighting, shading, meshes, batching, culling and level of detail. We will also explain how the nature of an underwater world presents challenges that are not necessarily a concern in applications where the user is grounded.

**Figure 1:** *Screenshot of the Ocean Rift title screen.*

## 2. Virtual Reality Best Practices

While VR as a popular consumer technology is a relatively new concept, there has been a lot of advancement in terms of industry best practices. This development has been informed by decades of VR research, and proactive innovation by companies like Valve, Sony, and Oculus [YHD*17]. The development and subsequent promotion of best practices, especially as a way of mitigating cybersickness in users [MS92,PMK17], has been a major focus of the VR development community. Cybersickness is related to simulator

DOI: 10.2312/cgvc.20191254

and motion sickness, and can result in symptoms like headaches, nausea and dizziness [Oma90].

Many VR best practices are universal across high performance PC, mid and low powered consoles, and mobile hardware. For example techniques for minimising aliasing and chromatic aberration, and the use of appropriate shaders are universal considerations. Similarly how a user moves through the virtual world is vitality important across all platforms. We have previously discussed our experiences in creating novel and comfortable movement modalities for underwater VR [aH19]. However there are some issues that are specific or amplified in mobile VR hardware, and it is vital not only be aware of these throughout development but to actively design with them in mind.

A set of best practices for mobile development is maintained by Oculus [ocu19a]. Rendering specific advice include being batch friendly, using baked lightmaps and static geometry, no shadow buffers, reflections or multiple cameras, minimise rendering passes, keep transparency to minimum, avoid alpha-testing and use texture compression. There are also CPU optimisations to consider including being mindful of the number of objects in a scene, use physical simulations sparingly and using object pooling to recycle objects instead of allocating new ones at runtime.

There is also a requirement to design and optimise around the lowest common denominator hardware. For example if you are developing an application destined for both mobile and PC markets, the base design would need to target the lower powered platform. Similarly within mobile there is a wide variety of VR hardware available, from a Samsung Galaxy S6 powered Gear VR [Gea19] to an Oculus Quest [Que19] which can render around four times as many polygons.

## 3. Targeting Frame Rate

All VR applications should aim to render at the native frame rate of the device. While it is usually fine to have occasional short drops in performance, perhaps due to a background process, prolonged drops in frame rate should be avoided at all costs. There is significant evidence that drops in frame rate can lead to discomfort and cybersickness. Indeed on curated asset stores like the Oculus Store this is a threshold requirement for passing quality assurance. Similarly pushing the boundary of what is possible on the device is also not a good idea for a commercial application. Developers need to be aware of every scene's worst case scenario in terms of camera angle, asset visibility, and extent of user interaction.

Techniques like Oculus' Asynchronous TimeWarp (ATW) [vW16] transform stereoscopic images based on the latest head tracking information every frame. The idea is to eliminate or reduce the judder in rendering that is prevalent on mobile phone applications due to temporary drops in frame rate. ATW works by reading the updated orientation information from the headset just prior to displaying the last rendered image to the user. Using this information, it constructs a transformation matrix that warps the eye textures from where they were when the frame was rendered, to where they should be at this exact moment. This not only helps to have as short a motion-to-photon latency as possible, but smooth over issues due to hitches in frame rate.

Mobile VR headsets up until very recently have only been capable of tracking 3 degrees of freedom (DOF) - that is they could only track head rotation, and not the user's position. However newer headsets like the Oculus Quest are capable of positional tracking using inbuilt cameras. While 6 DOF tracking is a massive leap forward in mobile VR technology and potential experiences, it does lead to positional tracking judder from incorrect projection, which is not managed by ATW. However, it is possible to warp the depth buffer similarly to perform a positional re-projection and alleviate some juddering artifacts.

Recently Oculus introduced another tool for automatic optimisation, called Fixed Foveated Rendering (FFV). FFV takes advantage of how the human eye works by rendering the periphery at a lower resolution to the centre. With devices like the Oculus Quest and Go rendering at a 1440 x 1600 per eye resolution, such techniques can be a powerful way of conserving performance with minimal artifacts.

While techniques like ATW and FFV have been transformative in allowing comfortable VR experiences on mobile hardware, it is not something that should be relied on to smooth over design or performance limitations. For example with ATW if head rotation is fast, the user will see black pulling into their peripheral vision, as the time warped render texture has not rendered that part of the world yet. Similar issues are found at the periphery of positional re-projection. Furthermore positional time warp in mobile VR for the foreseeable future is limited to maintaining smooth head tracking only, and will not affect artifacts in the virtual scene.

## 4. Underwater Environments in Virtual Reality

In Ocean Rift, users can swim around and explore 14 different habitats, ranging from coral reefs and shipwrecks to the Arctic and Atlantis. There are over 50 animals in the app, including dolphins, sharks, whales, sea lions, manatees and prehistoric creatures. The creatures are animated using advanced procedural animation techniques, which we discuss in [HTC17], [HTC18]. The goal is to make the user feel like they are underwater, using techniques which both increase immersion, and match the potential perceived realism of the virtual creature motion and behaviour. In this section we will look at a variety of rendering and optimisation techniques we have used to create these vivid underwater environments, especially focusing on mobile VR optimisation. We will take specific notice of polygonal and draw call budgets, as they are so critical in yielding viable performance.

Ocean Rift was developed in the Unity Game Engine [Uni19b] using the Oculus Mobile SDK for Android [Ocu19b] and OpenGLES 3.

### 4.1. Draw Calls and Geometric Complexity

Two of the most critical values to monitor when designing, developing and optimising for mobile VR are the draw call and geometric complexity of a scene. While most 3D projects will have a polygon or draw call budget, in mobile VR it is important to be conservative with these and not overshoot especially for long periods of time. Conservative estimates for draw calls on mobile hardware are between 100-300 calls, with vertices and triangles similarly anything between 50-300k, though this can vary greatly depending on what is being rendered. While a scene can potentially have millions of polygons and objects, it is essential that we do not render more than the aforementioned budget at any one time. Draw calls in particular are often resource-intensive, with the graphics API doing significant work for each call.

The two main techniques available when managing draw calls is static and dynamic batching. Static batching aims to combine static objects into larger meshes, and tries to render them in as few draw calls as possible. Similarly dynamic batching groups small non-static meshes together and aims to transform vertices on the CPU, and draw them in a single call. There are many engine specific rules for both static and dynamic batching [Uni19a]. For our underwater environments we mostly use static batching, which provided that objects are static and share a material can usually be batched in a single draw call. Typical examples of this are rocks, plants and bushes, where we can render hundreds in a single draw call with only a modest performance impact. We do however use dynamic batching occasionally when rendering schools of fish, explained in Section 4.10.

Level of detail systems (LoD) are often used in real-time rendering to reduce the polygonal count of a scene, usually by transitioning models into simpler representations based on distance from camera. We don't do this often in Ocean Rift, as we are largely CPU bound due to draw calls. Furthermore the abrupt transitions between LoDs (commonly known as popping) can also be very pronounced and immersion breaking in VR. However we do use LoD techniques when rendering animals, as discussed in sections 4.10 and 4.11.

## 4.2. Lighting

Most habitats in Ocean Rift are rendered using a mixture of baked and real-time lights. We use a bright directional light as the main light source of the scene. This is accompanied by an ambient light set to a similar colour as the water, which simulates bouncing ambient light. Finally a second directional light is used to light the water surface, and occasionally provide a more subtle second light source for large animals. While this does double the draw call on these animals, it can be advantageous especially if in open water where draw calls are naturally lower.

Figure 2 shows a typical example of a lightmapped environment from Ocean Rift. We use lightmapping for all static parts of the environment, which include the sea bed, rocks, plants and objects like shipwrecks and reefs. This is standard practice in most mobile VR applications, where lighting is baked rather than calculated. This allows for static but realistic shadowing, global illumination and ambient occlusion.

Dynamic shadows are currently prohibitively expensive on mobile hardware, however shadows can be very effective in grounding dynamic objects in a scene. We use a technique called blob shadowing for animals close to the ground, including crabs and other crustaceans. Blob shadows consist of a quad that is raycasted to the surface, acting like a crude dynamic shadow that follows the creature around, rendered using a basic transparent multiply shader.

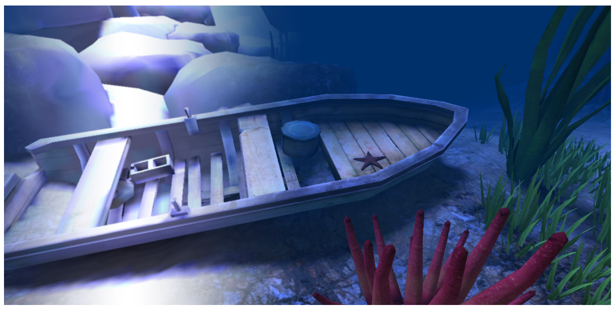
**Figure 2:** *Image of baked lightmap applied to sea bed environment and final screenshot.*

While this can result in artifacts as the quad intersects terrain polygons, provided that the ground is relatively flat this can help better ground the creatures in the surrounding lightmapped environment.

## 4.3. Fog

Forward rendered vertex fog is effective at giving the impression of a murky underwater environment. In Ocean Rift, exp2 vertex fog effect is used with an appropriate colour. The camera's clear colour is also set to this colour value, and the density of the fog is set so that the fog fades to a vanishing point just before the far clip plane, so that the water effect fades entirely to the base fog colour before clipping can be seen. This is important to ensure that the water plane fades completely before the far clip plane, and that animals or objects close to the far clip plane do likewise. Different habitats can have varying visibility (and associated fog density and clip plane distance). The trade off is largely the denser the scene, with lots of objects and animals, the thicker the fog needs to be in order to pull the far clip plane closer and ensure we don't overstep the polygon and draw call budget.

The worst case scenario in an underwater environment is that the user swims upwards until the sea bed is only just in view, and then look directly downwards. This represents the largest and busiest camera frustrum, whilst paradoxically being quite boring as most objects are barely visible due to fog. It is therefore critical to ensure that the user cannot overwhelm rendering by doing this. This is a challenge that can largely be overcome through proper consideration during the design of the environment.

On desktop and console a more realistic underwater atmosphere could be created using advanced techniques like global vertical fog. This would allow the water to get darker and murkier with depth. However while this cannot be replicated on mobile for any one viewpoint, it is possible to interpolate between values for fog density and colour based on distance from the surface. We also influence the directional and ambient light with depth, so that as the user swims deeper the entire scene changes atmosphere. It is also possible to interpolate in 3D, allowing for separate biomes inside an underwater habitat. For example clear blue water at the surface, can transition to dark green murky water at the sea floor, and a separate orange-tinted quadrant to give the impression of pollution or silt.

### 4.4. Water Surface

The water surface in Ocean Rift is largely cosmetic, and acts as a boundary at the top of each habitat. While it is possible for animals like dolphins to jump in and out of the water, and for the user to trigger rings and food to drop into the water, there is no camera interaction. However graphically the surface does have an important role in terms of immersing the user in the underwater environment. It is not possible to simulate fluid surface mechanics on mobile VR hardware yet, though such techniques are very effective on more powerful hardware. However it is possible to create a fairly realistic illusion of a water surface using simple shader techniques.

Examples of the water surface can be seen in Figures 1, 3 and 5. The surface mesh consists of a flat plane with typically a 100x100 vertex dimension, though larger habitats require a higher resolution. During run-time a shader modifies the vertex positions of the plane to produce a subtle wave effect. Two normal maps are also combined and scrolled (the first on the u axis, the second on the v axis) in the fragment shader, to simulate the finer details on the water surface. The shader also applies specular highlighting and diffuse noise to the final colour of the water surface. A variant of the shader also allows for a second diffuse texture to crudely simulate reflection from the sea bed for shallow water environments. This results in a shader that is effective at representing a wide variety of water surfaces, from still shallow waters to more open turbulent seas.

The water surface shader is one of the most advanced used in the application, consisting of real-time lighting calculations, three or four texture samples and multiple normal map calculations in the fragment shader. However simulating a pseudo-realistic water surface is integral to providing the feeling of being underwater in VR, and in most environments if the player is close to the water surface it is unlikely that there will be many creatures, vegetation or a cluttered environment to draw at the same time.

### 4.5. Light Shafts

It would be possible on more powerful hardware to simulate volumetric lighting or sub-surface scattering effects. However these require post-processing which are prohibitively expensive on mobile hardware due to per-pixel calculations, the use of render textures, and deferred rendering. However to break up the ubiquitous colour of the fog, and to give environment a greater sense of depth, a crude approximation of light shafts can be added to the scene. These consist of a quad mesh with a light shaft texture, shaded with a simple additive transparency shader. Large transparent objects are a big performance hit on mobile hardware due to overdraw - that is, overlapped parts of the image must be re-rendered for each object. However large transparent quads in the distance are possible provided that we minimise overdraw.

Light shaft quads are spawned randomly around the user, within a range of about 6-12m, though this is scaled with fog density. The origin of the quad is offset so that the light shafts appear to start at the height of the water surface. Light shafts are made to fade in and out after 15-20 seconds, or when the user swims too close, by altering the alpha channel, and their positions are randomly updated between cycles. While there are more physically accurate methods of simulating light shafts (such as volumetric lighting) the effect given by a quad with additive shader is very effective for the purpose of adding atmosphere to an underwater scene. Providing that the light shafts are far enough from the camera, their lack of volume is not apparent (due to user perspective), which is especially important in VR. The illusion can be broken if a creature swims through a light shaft, as the clipping edge will clearly show that the light shaft is a 2d plane. However, this is only obvious with slow-moving or large creatures (such as whales), and can be mitigated by detecting and fading out and cycling light shaft positions when creatures are nearby. Similarly for shallow water a simple check to ensure there is no collision between the light shaft and the static environment can be performed before placing. The light shaft effect can be seen in Figures 3 and 5.

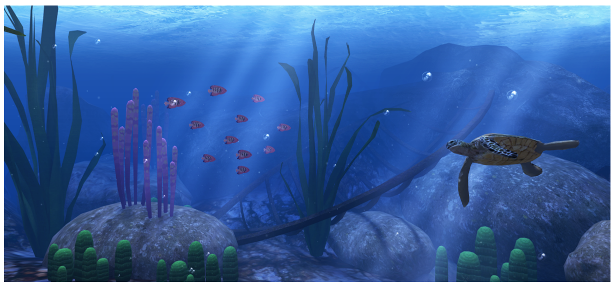

**Figure 3:** *Screenshot of shallow water environment, with water surface, fish school and two transparent quad-based light shafts.*

### 4.6. Particle Systems

Clouds of small particles such as dust motes have long been used in VR to provide a subtle sense of depth. While the technique is now largely ubiquitous, Ocean Rift was one of the first applications to use a particle system to provide this extra volume. In the underwater environment a particle system can be used to create a cloud of detritus billboard particles which float in the water. The particle system volume follows the camera around, and is offset based on the user's velocity. This ensures that particles are always being instantiated around the user as they swim around. A wind modifier is also used with a low frequency and amplitude in order to simulate turbulence and currents in the water.

We also use particle systems extensively for bubble effects, including emissions from cetacean blowholes, simulation of the user's scuba breathing, and hidden emitters in the environment to enhance atmosphere. We use simple billboard particles for the bubbles rather than spherical meshes. While this does mean we lose geometrical accuracy in VR, billboards allow us to render a lot more bubbles, and the trade-off isn't particularly noticeable due to their round shape and the fact that billboards always directly face the camera.

As the user is able to freely move around, it is necessary to fade particles that are near the camera. The reason for this is that particles passing through the eyes as the user swims forward can be quite uncomfortable. While it is possible to modify the near clipping plane to mitigate this, this in itself can be detrimental to the experience due to particles disappearing instantly, or when rendering near object interactions like the user's hands. An alternative is to use the depth buffer in a custom additive shader, which acts to fade particles as they get near to the camera. A similar shader can be applied to other small particle systems like bubbles, plankton and floating detritus.

### 4.7. Caustics

Caustics effects aim to simulate how light appears to dance on the surface of underwater objects due to refraction at the water surface. This can have a dramatic effect on the perceived realism and dynamicity of a scene. As caustics are quite transient and chaotic, it is possible to crudely approximate caustics on mobile hardware without having to perform any simulation or adhere to physical accuracy. Caustics are especially useful at enhancing the underwater effect on the surface of large aquatic creatures like sharks, dolphins and whales. Dancing light on the skin can help break up the silhouette, texture and material of the creature's skin and make a big difference to the perceived realism. We do not apply the caustics shader to static terrain and objects due to the performance cost, though we do apply the effect in special enclosed environments like a shark cage.

There are two main techniques for achieving this effect on mobile hardware, depending on what is being affected and the composition of the scene. One approach is to use a projector or equivalent, which acts to orthographically project a tiled and looping animated caustics texture downwards onto anything in the scene. The disadvantage of this technique is that everything that is projected onto will need to be rendered twice, which doubles the number of passes and draw calls.

The alternative technique is to apply the animated caustics texture directly in the fragment shader for specific objects, masking the effect so that it only affects polygons which are being lit by the main directional light. The strength of the mask can also be affected by the angle of light, so that the caustics effect fades with glancing angles. This technique also have disadvantages, for example the tongue of a blue whale would receive caustics even if it shouldn't due to occlusion. However this can be mitigated by a second texture which masks the inside of the creatures mouth. An example screenshot of sea lions lit using a caustics shader can be seen in Figure 5.

While cycling through 32 caustic textures produces a typical fast dancing light effect, it can be inadequate for some environments and water depths. By interpolating between frames in the shader it is possible to produce slower caustics. Similarly it is possible to fade the effect with distance from the surface, so that caustics are bright at the water surface and fade as the user swims into deeper waters.

### 4.8. Terrain

There are many techniques for rendering large terrains for real-time applications, including complex LoD systems with dynamic polygonal resolution and advanced vegetation systems. The Unity engine has its own in-build terrain system for example. However as previously mentioned we are largely CPU-bound in mobile VR, which requires that we minimise the number of draw calls as much as possible. While splitting the terrain into segments and performing culling and LoD optimisation in real-time has many advantages, we instead draw the entire terrain mesh in a single draw call. This does put a limit on how complex the terrain can be, and the largest terrains in our application consists of a grid of 100x100 quads (10,000 vertices) for a $1km^2$ habitat. However with a relatively smooth topology and well placed assets this is more than adequate.

Terrain texturing is performed using a splat shader, a common technique where a color encoded texture map is used to blend various tiled textures together. The effect is to have a complex surface texture on simple geometry, rendered in a single draw call with baked lighting. Figure 4 shows an example of the terrain rendered using a splat map. The terrain topology and splat map were hand crafted for each habitat inside the Unity editor, using custom editor extensions developed for the project.

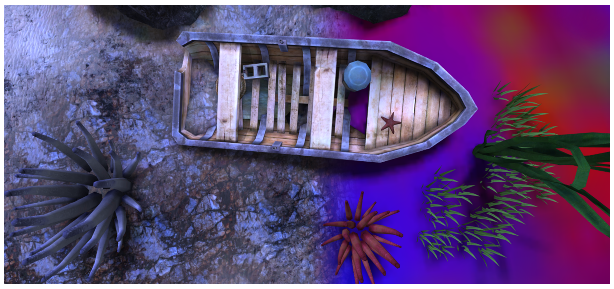

**Figure 4:** *Screenshot showing textured sea bed (left) and splat map (right). The terrain shader uses this map to blend four seamless tiled sand and rock textures together to produce the final surface texture.*

### 4.9. Vegetation

The sea bed is often full of aquatic vegetation, especially in shallow waters. We render vegetation using a custom shader which combines light-mapping with vertex animation. Most plants are modelled using polygons only, rather than billboards or alpha blended quads. The main reason for this is to mitigate overdraw, which is very expensive on mobile chipsets.

Using a modelling package we provide each vertex of the plant mesh with a colour. Each channel represents a different aspect of motion - red controls vertical movement, blue horizontal and green high frequency. The vertices are black at the base of the plant (to curb any movement at the terrain surface), and gradually get brighter with height. A vertex shader at run-time applies three separate sinusoidal waves at various frequencies and amplitudes in order to simulate floating and bending in the water current. In order to stop all plants from swaying in unison, the effect is phased based on position in 3d space.

The effect of this is that we can have veritable forests of statically batched animated plants in a scene, with each type rendered in a single draw call.

### 4.10. Fish Schools

Fish can commonly be seen schooling in large groups, which can be particularly impressive when seen in VR. We simulate schooling using the standard Boids algorithm, pioneered by Craig Reynolds [Rey87]. Here three steering forces are combined to produce schooling behaviour - 'Cohere' which makes fish swim towards their neighbours, 'Separate' which ensures they don't get too close to one another, and 'Align' which allows the fish to swim in a common direction.

We combine the Boids algorithm with a wandering steering behaviour, which moves fish towards an ever changing target position within a set boundary. This boundary allows us to place many schools within a habitat, while ensuring they don't get too close to one another. We also scale the number of fish in each school based on VR platform.

By default, each fish will require their own draw call, which would be prohibitively expensive even for small schools of fish. However dynamic batching has limitations - for example the Unity engine will currently dynamically batch moving meshes containing no more than 900 vertex attributes, with no more than 300 vertices. Dynamic batching also does not work on skinned meshes - such as ones with skeletal animations.

Our solution currently is to have a level of detail system that swaps each school fish from an animated skinned mesh to a sub-300 vertex mesh based on distance. This means that only fish that are near to the camera require their own draw call, while fish further away are rendered in a single batched draw call. While this means that draw calls will go up significantly if the user swims close or into a school of fish, it does allow us to optimise at quite a short distance, and allows schools to be much closer together and give the impression that the sea is full of life.

While Boids is a relatively cheap algorithm, it does have CPU overhead especially when evaluating neighbourhoods and calculating forces between neighbours. In order to optimize performance, we disable schools that are just outside the camera frustrum. We perform similar optimisation on all small singular animals in the app. This does somewhat reduce the dynamicity of the environment, but is an important trade-off to preserve CPU bandwidth.

While we have concentrated on skinned mesh animation for schooling fish in Ocean Rift, it would also be possible to animate schools using a custom animation shader, similar to the technique developed for the video game ABZU [Abz19]. This would work in a similar way to our vegetation shader, where vertices are directly animated in the shader to produce animation. While this technique would only be suitable for smaller fish or large fish schools, it would allow for low-cost animated static batching, potentially allowing us to render thousands of fish in a single draw call. Furthermore such a technique could be extended to accomodate the rendering and animation of more amorphous and complex morphologies, such as jellyfish, octopus and squid.

### 4.11. Star Animals

Star animals like dolphins, sharks and whales are the main attraction. It is therefore necessary to spend as much processing power as possible on rendering and animating these creatures in order to produce the most vivid experience. VR can give users an enormous sense of presence in a virtual environment, and the requirement to have quality models, shaders, animation and behaviour are all heightened due to this increased immersion.

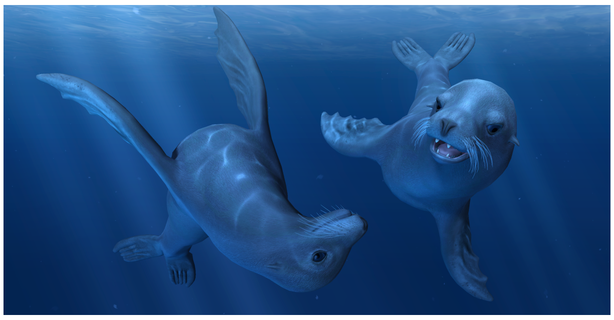
**Figure 5:** *Screenshot of two sea lions with caustic shader effect.*

There are many rendering issues which could potentially pull users out of the experience, such as visible polygon defects on the mesh, unrealistic shaders and materials, and texture stretching due to inadequate mesh skinning. Similarly, sub-par animation, glitching or unrealistic behaviour are also amplified in VR, due to the user's attention being focused intimately on the animal in front of them. The added perceptual volume that a player has in VR due to the immersive experience can make them more critical of unrealistic behaviour.

A lot of these issues can be alleviated by having good quality models, textures and shaders. For mobile VR this can present a challenge, as you cannot simply throw polygons and complex multi-pass shaders at the problem. For example, a common technique in video game development is to generate a low poly model from a high poly one, and bake the extra surface information into normal maps. While perfectly adequate on a flat screen, normal maps do not account for stereoscopy, so the effect can break down in VR, especially up close. However it is possible to mitigate this issue by modelling specifically with low poly in mind, and normal maps can still be very effective at producing high frequency information like scales, pores, scratches and scars in VR. Similarly baking an ambient occlusion map into the diffuse texture can also help give extra detail to a creature or environment in some circumstances.

Figure 5 shows a screenshot of two Sea Lions from Ocean Rift. Each model consists of around 7500 polygons and 15000 faces in total, split into three sub-meshes - the body, eyes and whiskers, teeth and tongue collectively. This is also our typical setup for most large animals like whales, sharks and prehistoric creatures.

The facial area is very important in VR, especially for large intelligent creatures like sea lions. This is where the user will tend to look most often, and their focal point when tracking the creature as it moves. It is therefore important that there is a higher frequency of polygons around the face, both to provide extra geometric detail and facilitate facial animation.

We control the jaws and eyelids using bones which allows for expression, micro movements and blinking. We also rotate the eye spheroids at runtime so that the creatures are looking at something - usually towards the camera, with some micro random adjustments to simulate saccadic eye movements and give the impression that the animal is studying the user. These micro movements of the eyes and eyelids are a very important feature for selling the realism of close encounters, as dull lifeless eyes can be very detrimental to immersion.

Level of detail techniques are commonly used in 3d applications to render the environment or characters at lower detail based on distance (less polygons, simpler shading or animation). For large animals we tend to keep the overall body mesh the same number of polygons with distance, due to the fact that underwater the animal will be quite close when visible, and transitions between LoD levels are quite apparent in VR. However with smaller animals it is possible to switch between several LoD mesh levels, especially with exp2 fog where visibility reduces quite quickly.

Our sea lions are rendered in three draw calls when up close, and in a single draw call when further away. While we render the eyes, mouth parts and body up close, we only show the latter when animals are distant. The exact crossover point is a matter of tweaking and varies based on fog density and animal in question.

Creature optimisation is usually about balancing the requirements of each scene. If for example there are five sea lions in a habitat, expensive draw calls are multiplied by five if all are in the closest LOD (perhaps due to items of interest like a ring or food being dropped into the water). In this case, we mitigate the chances of this happening by only allowing a few sea lions to show interest in an area near the player at any one time. This does vary based on platform however - on older Samsung Gear VR hardware we might limit this to two at a time, while on PC we can remove this limitation altogether.

## 5. Conclusion

Developing mobile VR apps is a significant challenge, and adhering to best practices is essential in creating compelling and comfortable VR experiences. Developers need to maintain a high quality player experience, while being mindful of the limitations of a low-power platform. This is not a problem that is likely to be resolved in the near future, despite the rapid development and introduction of more powerful devices. Publishing platforms still serve older devices, such as the Samsung Gear VR. Furthermore, while computational power has increased so have the demands on power. Unfortunately, battery technology has not improved significantly so this will continue to be a bottleneck for developers for some time.

In this paper we have presented a number of optimisation techniques used in the popular VR experience 'Ocean Rift'. We have explained how we follow universal best practices, and given examples of how we have applied these when creating underwater environments. We present this case study from our commercial experience in the hopes that it will support the community in developing increasingly visually impressive and interactive experiences.

## References

[Abz19] Creating the Art of ABZU, 2019. URL: https://www.gdcvault.com/play/1024409/Creating-the-Art-of-ABZU. 6

[aH19] AP CENYDD L., HEADLEAND C. J.: Movement modalities in virtual reality: A case study from Ocean Rift examining the best practices in accessibility, comfort, and immersion. *IEEE Consumer Electronics Magazine 8*, 1 (Jan 2019), 30–35. doi:10.1109/MCE.2018.2867971. 2

[Gea19] Samsung Gear VR website, 2019. URL: https://www.samsung.com/global/galaxy/gear-vr/. 2

[HTC17] HENSHALL G. I., TEAHAN W. J., CENYDD L. A.: Crowd-sourced procedural animation optimisation: Comparing desktop and vr behaviour. In *2017 International Conference on Cyberworlds (CW)* (Sep. 2017), pp. 48–55. doi:10.1109/CW.2017.52. 2

[HTC18] HENSHALL G. I., TEAHAN W. J., CENYDD L. A.: Virtual reality's effect on parameter optimisation for crowd-sourced procedural animation. *The Visual Computer 34*, 9 (Sep 2018), 1255–1268. doi:10.1007/s00371-018-1501-2. 2

[MS92] MCCAULEY M. E., SHARKEY T. J.: Cybersickness: Perception of self-motion in virtual environments. *Presence: Teleoperators & Virtual Environments 1*, 3 (1992), 311–318. 1

[Oce19] Ocean Rift Gear VR Store webpage, 2019. URL: https://www.oculus.com/experiences/gear-vr/1249878741704255. 1

[ocu19a] Oculus Best Practices for Mobile Development, 2019. URL: https://developer.oculus.com/documentation/unity/latest/concepts/unity-mobile-performance-intro/. 2

[Ocu19b] Oculus Mobile SDK, 2019. URL: https://developer.oculus.com/downloads/package/oculus-mobile-sdk/. 2

[Oma90] OMAN C. M.: Motion sickness: a synthesis and evaluation of the sensory conflict theory. *Canadian Journal of Physiology and Pharmacology 68*, 2 (1990), 294–303. PMID: 2178753. doi:10.1139/y90-044. 2

[PMK17] PALMISANO S., MURSIC R., KIM J.: Vection and cybersickness generated by head-and-display motion in the oculus rift. *Displays 46* (2017), 1–8. 1

[Que19] Oculus Quest website, 2019. URL: https://www.oculus.com/quest/. 2

[Rey87] REYNOLDS C. W.: Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph. 21*, 4 (Aug. 1987), 25–34. doi:10.1145/37402.37406. 6

[Uni19a] Unity Documentation - Draw Call Batching, 2019. URL: https://docs.unity3d.com/Manual/DrawCallBatching.html. 3

[Uni19b] Unity Game Engine, 2019. URL: https://unity3d.com/. 2

[vW16] VAN WAVEREN J. M. P.: The Asynchronous Time Warp for virtual reality on consumer hardware. In *Proceedings of the 22Nd ACM Conference on Virtual Reality Software and Technology* (New York, NY, USA, 2016), VRST '16, ACM, pp. 37–46. doi:10.1145/2993369.2993375. 2

[YHD*17] YAO R., HEATH T., DAVIES A., FORSYTH T., MITCHELL N., HOBERMAN P.: Oculus best practices. *Oculus VR* (2017). 1