POWERVR

Shader Based Water Effects

Filename	:	POWERVR.Shader Based Water Effects.1.10f.OGLES2External.doc
Version	:	1.10f External Issue (Package: POWERVR SDK 2.06.26.0716)
Issue Date	:	26 Feb 2010
Author	:	Imagination Technologies

Contents

List of Figures

1. Introduction

Generating efficient and detailed water effects can add a great deal of realism to programs that utilise real-time graphics. The purpose of this document is to highlight techniques that can be used in software running on POWERVR SGX platforms to render high quality water effects at a relatively low computational cost.

The effect can be achieved in a variety of ways, but this document will focus on the use of vertex and fragment shaders to alter the appearance of a given plane to simulate a water effect.

2. Water Demo

Although there are many examples of water effects using shaders that are readily available, they are all devised around the availability of high performance graphics chips on desktop platforms. The following sections of this document describe how the general concepts presented in these desktop implementations can be tailored to run at an interactive frame rate on the lower end of PowerVR SGX platforms, including the optimisations that were made to achieve the required performance.

2.1. Geometry

In the demonstration, the plane that the water effect is applied to has been calculated using the POWERVR Shell infinite plane equation, `PVRTMiscCalculateInfinitePlane()`. This function takes a plane equation (in the form (A,B,C,D)), the current camera position and the far clip plane distance as inputs and gives an output of four or five points that can be used to render the plane in OGLES2.0. As the effect does not require a highly tessellated surface, the calculated plane is adequate for the effect. To reduce the number of calculations that are required, the program assumes that the plane will always lie along the y-axis.

The skybox used in the demo has been calculated using the Shell's `PVRTCreateSkybox()` method. The skybox texture is stored in the PVRTC 4bpp format and uses bilinear filtering with nearest MIP-mapping to provide a good balance between performance and quality.

2.2. Bump Mapping

To simulate the perturbation of the water's surface, a normal map is used to bump the plane. The normal map used in the demo is y-axis major (as opposed to many other bump maps that are z-axis major). The texture coordinate for each vertex is calculated in the vertex shader as the x and z values of the position attribute that has been passed into the shader, `inVertex`. To animate the surface of the water, a bump texture coordinate for the vertex is calculated by scaling the original texture coordinate and adding an offset to it (the bump's translation). Once this value has been calculated, it is passed to the fragment shader as a varying so that the bump coordinate of each fragment will be interpolated automatically. The translation is calculated on the CPU each frame by adding the multiplication of the bump's velocity and delta time value since the last frame to the current translation value. Only the fractional component of the translation is given to the shader, which allows sufficient accuracy to be maintained. In the fragment shader, the bump coordinate is used to sample the normal map, the result of which is converted from texture-space into normal-space so that it can be used in further calculations. All calculations in the fragment shader are done in model-space as this is native to the object and saves redundant transformations into world, view space or any other representations of the object's orientation.

Using a single bump layer can make it very apparent that the perturbation is occurring along a linear path, and for this reason it is suggested that at least two scaled and translated bump layers are applied to the plane to make the surface perturbations look much more natural (Figure 1).

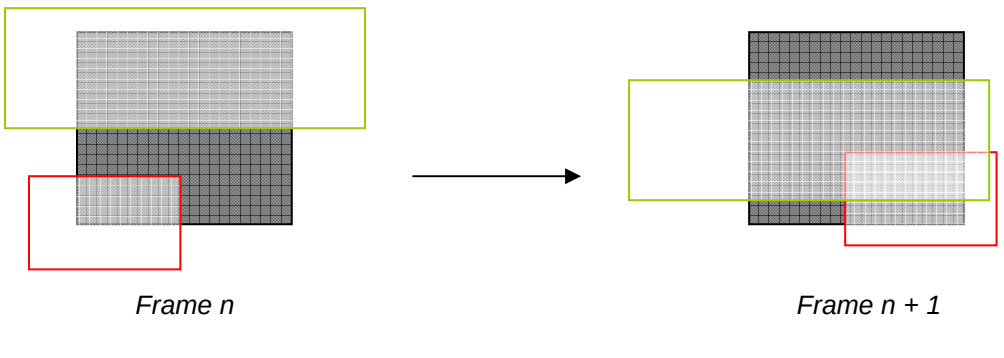

Figure 1 Bump map animation over two frames

2.3. Reflection render pass

Reflection in the program is achieved through an additional render pass before the main render loop. The purpose of this is to render all of the geometry that needs to be reflected during the frame and storing the information as a texture that can easily be applied to the water's surface during the main render. Before rendering the scene, the camera needs to be mirrored about the plane to give the correct view for the reflection. To do this, the inverse of the plane's direction is saved as the second row of a mirror matrix (where the forth element is set to double the w value of the plane so scaling occurs correctly), which is then multiplied by the current view matrix to give the new view matrix required for the render. As the diagram in Figure 2 shows, only mirroring the camera will result in the inclusion of objects below the water's surface that need to be ignored for the reflection to work correctly. This issue can be avoided by utilising a user defined clip plane along the surface of the water to remove all objects below the water from the render (See section 3.1 for information on how this can be achieved in OGLES2.0) . Figure 3 shows the clipped reflection scene stored as a texture in the demo.

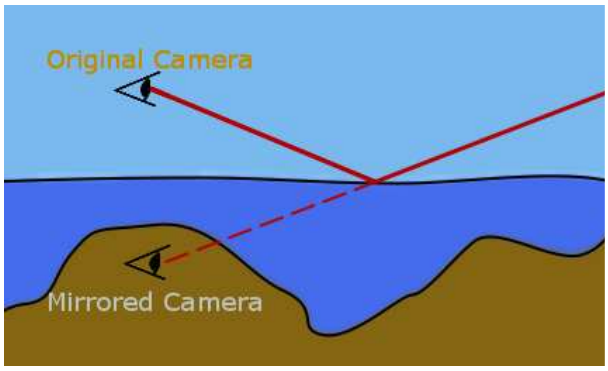

Figure 2 Mirrored camera for reflection

Figure 3 Reflection stored as a texture

The texture coordinate of each fragment is computed by dividing the fragment's coordinate on the screen (retrieved using the `gl_FragCoord` command) by the dimension of the window. To save a few cycles of computation in the fragment shader, the divide is removed and the fragment's coordinate is instead multiplied by the reciprocal of the window's dimension. Once the texture coordinate for the fragment has been found, it is then offset by subtracting the normal computed during the bump mapping process scaled by the `WaveDistortion` uniform. The scaling factor used for the normal determines how severely the water is perturbed. The newly calculated texture coordinate is then used to sample the appropriate reflection texel for the fragment. The colour value of the fragment can then be output by the shader, resulting in the application of the perturbed reflection texture to the surface of the water, as shown in Figure 4.

Figure 4 Water effect using only a permutated reflection texture

Figure 5 Water effect using the reflection and a constant mix with the water colour

Figure 6 Water effect using a permutated reflection texture and alpha blending

Used alone, this reflection technique results in unrealistically reflective water, as objects lying beneath the surface or colouring caused by dirt within the body of water are not considered. There are several additional steps that can be taken at this stage to improve the quality of the effect depending on the end result you are hoping to achieve and the performance you require:

- perform another render pass to create a refraction texture (Section 2.4) (Expensive – reduced the performance of the effect on the development hardware by 60%)
- mix the colour value of the fragment with a water colour (Figure 5) (Good for simulating very murky water – reduced the performance by 9%)
- alpha blend the water so objects below the water can be seen (Figure 6) (Reduces the realism of the effect when rendering deep water because the edges of submerged objects will still appear sharp – reduced the performance by 11%)

It may be worth opting for one of the less intensive solutions if the water effect you require will be quite shallow as there may be little point applying refraction to this type of water effect. If you have chosen to mix the texel colour with a water colour, it can be done in one of the two ways discussed in section 2.4.

As introducing a new render pass can be very expensive, there are a few steps that can be taken to improve the efficiency of this process:

1. **Render as little as possible:** The CPU should be used to determine which objects are above the water plane and, if possible, which objects are intersecting the plane as these are the only objects that will be needed during the render. If this is still proving to be too expensive, the pass can be reduced to just draw the key objects in the scene, such as a skybox and terrain.
2. **Favour FBO use to fetching from the frame buffer:** Rather than creating textures from the contents of the frame buffer, a frame buffer object with a texture bound to it should be used to save the output of the render pass to a texture (See the “OpenGL ES 2.0 Application Development Recommendations” document available on the PowerVR Developer Relations website for more information).
3. **Render to texture at the lowest comfortable resolution:** As the reflection texture is going to be distorted anyway, you can get away with using a fairly low resolution. A texture 256x256 has proven effective in the demo, but depending on the maximum resolution of the screen on the platform you are developing for, this resolution could be reduced further. Keep in mind that a drop from 256x256 to 128x128 will result in a 75% lower resolution.
4. **Avoid using discard to perform clipping:** Although using the discard keyword works for clipping techniques, its use negates performance advantages that the POWERVR architecture offers (See section 3.1 for more information).

2.4. Refraction render pass

In a case where the rendered water should appear to be fairly deep, adding refraction to the simulation can vastly improve the quality for the effect. To do this, an approach similar to that taken during the reflection render pass should be used, where all objects below the water are rendered (through a mix of rough clipping on the CPU and using the inverse of the water plane to clip during the render) and then saved to a texture (Figure 8). If the effect you require should produce very clear water, you will want to render all elements of the scene below the water, including the skybox (or similar object if your application uses one) so that all of these items are rendered to a texture (Figure 9). If you want to produce murky water, you can use a fogging effect to fade out objects as they submerge further (See section 2.4.1 for more information).

Once the scene has been rendered to a texture, it can then be utilised by the water's fragment shader. The texture coordinate found using the process discussed in section 2.3 is also used to sample the refraction texture. Once the texel for the fragment has been determined, it needs to be mixed with the reflection texel that has been sampled. This can either be done by mixing the textures together by a constant amount (e.g. 50/50), or using an equation such as the Fresnel term to provide a dynamic mixing of colour based on the current viewing angle. Figure 7 shows the full effect of the water, where reflection and refraction textures are mixed using the Fresnel term.

Figure 7 Full water effect

2.4.1. Fogging

A fogging effect to simulate depth can be accomplished by setting a maximum fogging distance (beyond which objects are no longer visible) and using the value to perform a linear fade between the object's original colour and the fogging colour based on the object's depth. To do this, the reflection pass must use the water's colour as the clearing colour and objects such as the skybox that should appear infinite, and therefore past the maximum fogging distance, should be omitted from the render. As the program assumes the water will always lie on the y-axis, the w component of the water plane is negated and passed into the shader as the water height. The reason for the negation is that it allows the value to refer to the plane's displacement from the centre of the world instead of specifying how far along the plane's normal (positive y-axis) it is displaced. The depth of the vertex is then calculated as the water height minus the y-axis value of the vertex in world-space, the result of which is then passed to the fragment shader as a varying float. The fragment shader then calculates the mix of the fogging colour and the object's colour as the fragment's depth divided by the maximum fogging depth (which is implemented as the fragment's depth multiplied by the reciprocal of the maximum water depth to save a few cycles). This mixing value can then be used to determine how much of the fogging colour is applied to the fragment. The maximum fogging value can also be used on the CPU to only render objects that are below the surface and above the maximum fogging distance. This is a good way to cut down the number of objects that need to be rendered during the refraction pass. To perform a murky render pass, the clear colour should be set to the water colour (Figure 10). On the development hardware, disabling the fogging effect gave a 3.5% increase in performance.

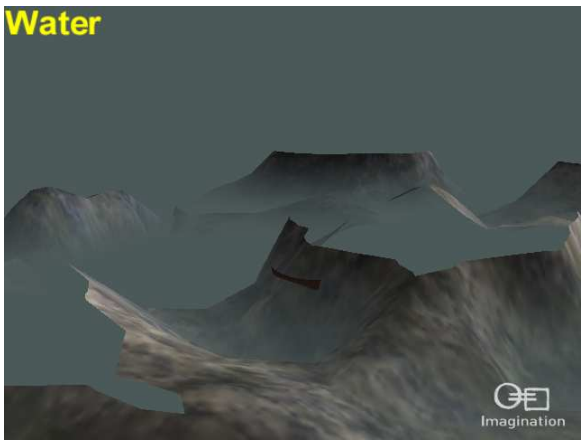

Figure 8 Refraction stored as a texture

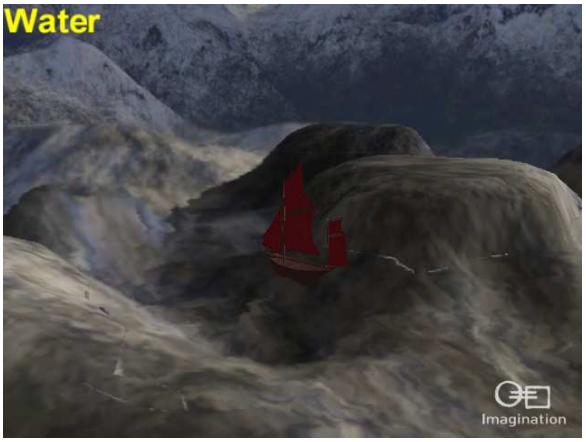

Figure 9 Water effect using only the refraction texture (without fogging)

Figure 10 Water effect using only the refraction texture (with fogging)

2.4.2. Fresnel term

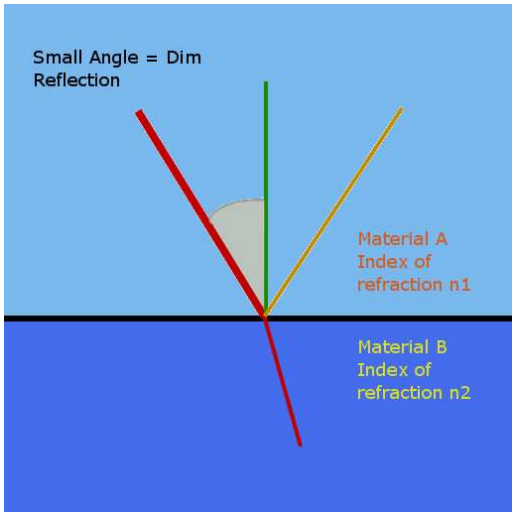

Figure 11 Small angle of Fresnel reflection

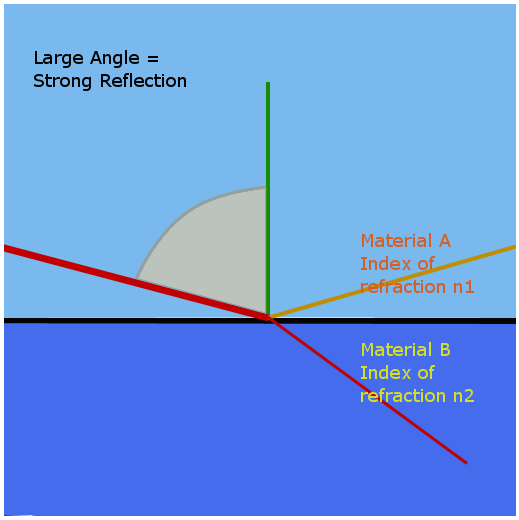

Figure 12 Large angle of Fresnel reflection

The Fresnel term is used to determine how much light is reflected at the boundaries of two semi-transparent materials (the rest of which is absorbed through refraction into the second material). The strongest reflection occurs when the angle of incidence of the light ray is large, and, adversely, reflection decreases as the angle of incidence reduces. To clarify, the light direction used in these calculations is the light that would be seen by the eye, not the direction of any lights declared in the world (calculated in the vertex shader as the position of the eye in model space minus the position of the vertex). The approximation of the Fresnel term used in the demo is determined using the following formula:

```
 R(0) = pow((n1 – n2),2) / (n1 + n2)^2
R(alpha) = (1 – R(0)) * (1 – cos(alpha))^5 + R(0)
```

To save computation time, the result of the equation above is calculated outside of the application using the values in the tables below:

Material	Index of refraction
Air	1.000293
Water (At room temperature)	1.333333

	Fresnel approximation
R(0)	0.02037
1 – R(0)	0.97963

The angle between the normal of the water and the water-to-eye direction is found by calculating the dot product of the two values after they have both been normalised. During implementation, the assumption is made that the accumulated normal found during bump mapping is already normalised (although this is not always the case) and the normalised value of the water-to-eye vector is found by performing a texture lookup of a normalisation cube map to save cycles (See source code for more information). Once the angle has been found, the following calculation is performed to determine the mix between reflection and refraction for the fragment:

```
airWaterFresnel = 1.0 – fEyeToNormalAngle
airWaterFresnel = airWaterFresnel^5;
airWaterFresnel = ((1-R(0)) * fAirWaterFresnel) + R(0)
```

The resultant value of this calculation is used to linearly interpolate between the two texture colours using the GLSL ES `mix()` method. Using the Fresnel calculation instead of a constant mix on the development hardware reduces the performance by 22%, but gives a much more realistic output.

3. Optimisations

3.1. User defined clip planes in OpenGLES 2.0

Although OpenGLES 2.0 allows the use of a programmable graphics pipeline, which allows this water effect to work so well, there are some drawbacks that occur due to the reduced functionality compared to its desktop counterpart, OpenGL. One of the features that are not available, but required to produce a good quality reflection and refraction, are user defined clip planes. Many OpenGLES 2.0 text books suggest performing a render pass that uses the discard keyword in the fragment shader so that fragments beyond the user defined clip plane will be ignored. Although this method works and will produce the required output, using the discard keyword is very expensive because the program has to reach the fragment shader before it realises that the fragment is not required, which is a significant waste of processing time. On devices such as those utilising the PowerVR SGX platform, this wasted computation time can significantly reduce performance and its use negates performance advantages that the POWERVR architecture offers.

To solve this problem, a projection matrix modifying technique can be used. The projection matrix is typically used to convert all of the objects in the scene from view space coordinates into clip space, and part of this process is to clip objects that do not fall between the near, far, left, right, top and bottom planes of the projection-space. If you consider the function of the projection matrix this way, it becomes apparent that the program already has a built in mechanism for clipping. For this reason, the projection matrix can be altered to set the near clipping plane as a plane that the user has defined (Figure 13).

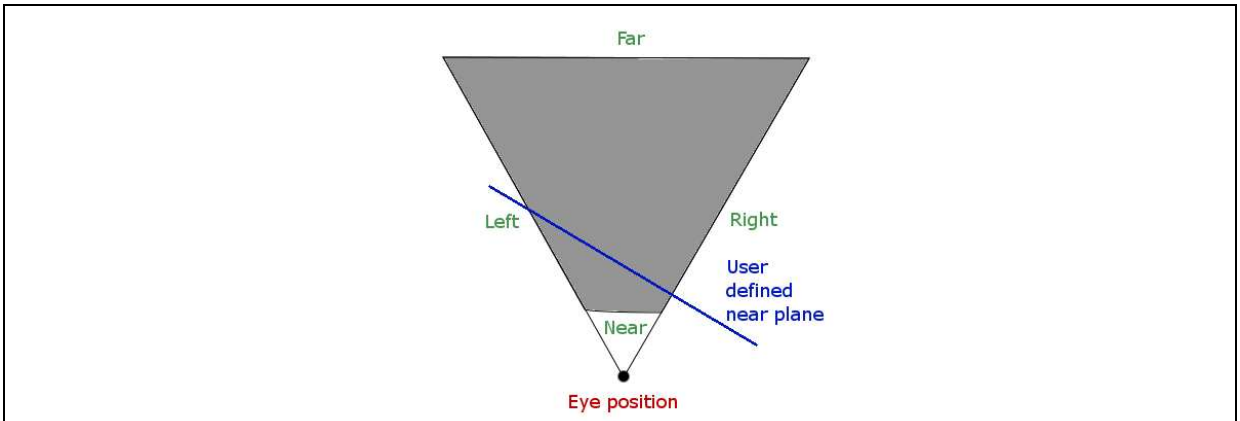

Figure 13 Birds-eye-view of a projection matrix with a proposed user defined clip plane

For this technique to work, the user defined clipping plane has to be converted into view space. As the plane only has direction (as it should be in the form (A, B, C, D)), the plane must be multiplied by the inverse of the view matrix to convert it into view space. For this technique to work, it is assumed that the w component of the view space plane is negative so that the clipping plane is facing away from the camera (as the near clipping plane of the projection matrix would normally). This does restrict the flexibility of the clipping method, but does not pose a problem for the clipping required in the demo.

The near clipping plane is defined in the projection matrix as the third and fourth rows, so these are the values that need to be altered. For perspective correction to work, the fourth row must keep the values (0,0,-1,0). For this reason, the third row has to be set to the following:

```
row3 = (clipPlane.x, clipPlane.y, clipPlane.z + 1, clipPlane.w)
```

Unfortunately, the projection matrix modifying technique is not quite this straight forward. By changing the position of the near plane, the far plane is skewed and no longer remains parallel with the near plane (Figure 14). Although this problem cannot be fixed fully, the effect can be minimised. This is done by scaling the clip plane before the third row is set, which causes the orientation of the far clipping plane to change (Figure 15). To do this, the point that lies furthest from the near plane in NDC (Normalised Device Coordinates) must be calculated, using the following equation:

```
clipSpaceCorner = (sgn(clipPlane.x), sgn(clipPlane.y), 1.0, 1.0)
clipSpaceCorner = clipSpaceCorner * projection.inverse()
```

Using the newly calculated clip space corner, the plane can be scaled using this equation:

```
clipPlane = clipPlane * (2.0 / clipSpaceCorner . clipPlane)
```

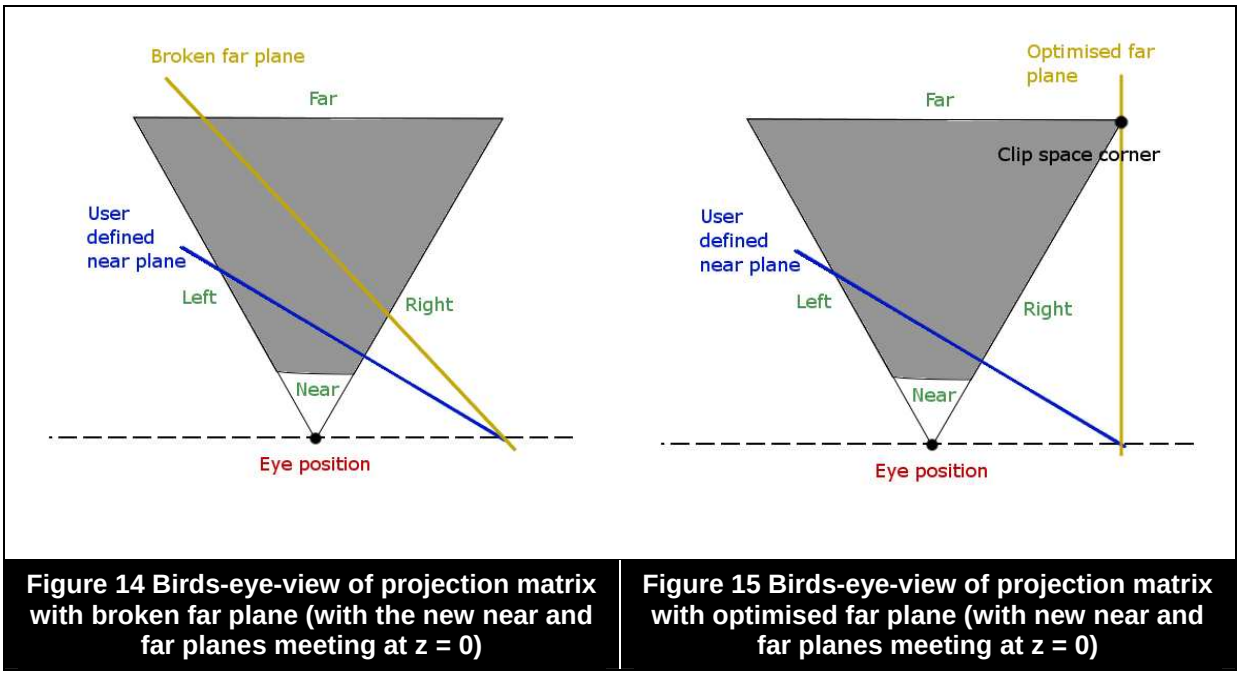

Figure 14 Birds-eye-view of projection matrix with broken far plane (with the new near and far planes meeting at z = 0)

Figure 15 Birds-eye-view of projection matrix with optimised far plane (with new near and far planes meeting at z = 0)

Although this technique may seem harder to understand than the discard method of clipping, it is significantly faster as it allows the graphics hardware to perform clipping for almost no additional cost.

3.2. #define in GLSL

During production of this demo, it became apparent that using Booleans for if and if-else statements in vertex and fragment shaders resulted in at least an extra cycle for each decision point. This quickly became an issue as simple shaders that were covering large portions of the screen, such as the skybox, were processing needless cycles. Rolling out the code into a number of nearly identical shader files would provide a way around this issue, but would cause a lot of code duplication. To allow single shader files to be used repeatedly for multiple code paths, the #define pre-processor directive was used with #ifdef and #else in shader files so that code paths could be specified during compilation-time instead of at run-time. The SDK allows this when using the method `PVRTShaderLoadFromFile()` by providing inputs for the additional parameters `aszDefineArray` and `uiDefArraySize`, where `aszDefineArray` is a pointer to an array of strings holding defines the user wants to append and `uiDefArraySize` is the number of defines in the array. This method automatically provides the #define and new-line character that need to be appended to the source code for each define, so the user only needs to provide an array of strings for the names i.e. A_USEFUL_DEFINE.

Once a number of shaders have been loaded in this way, the shader used for an object can be changed during run-time to a shader with a different code path. Although this method creates a shader for each decision path (which uses more memory, but would have also been done if the code was rolled out in different files), it allows the performance of shaders to be improved by removing redundant cycles.

3.3. Further Optimisations/Improvements

3.3.1. Normalisation cube map

As texture lookups are cheaper than performing normalisation operations on three dimensional vectors, normalisation in the demo is performed using a lookup to a normalisation cube map. For demonstration purposes the method of generating the normal map has been left in the code, but time spent initialising the application could be saved by loading a pre-existing normalisation map instead. The theory behind this method is simple; take a directional vector as the texture coordinate of the lookup and return the colour at that position. This colour represents the normalised value of any directional vector that points to its position. As the value retrieved using this technique is in texture-space (0,1), a conversion into normal-space (-1,1) needs to be used to allow the value to be used in further calculations. Using this method can half the number of cycles required to normalise a vector.

3.3.2. Scale water distortion

Without scaling the amount of distortion that is applied to each fragment, water in the distance can end up sampling the reflection and refraction textures at too big an offset, which gives water in the distance an unrealistic degree of distortion. Additionally, the bigger offset for distant fragments results in a higher amount of texture read cache misses. By scaling the amount of distortion that is applied to a given fragment, the visual quality of the effect can be improved and the number of stall cycles caused by texture cache misses can be reduced. This is done in the demo by dividing the wave's distortion value by the distance between the camera and the fragment's position (so fragments further from the camera are distorted less). Implementing this change reduces the performance of the app <1% as most of the additional maths in the fragment shader is covered up by the stall cycles caused by the texture reads and its use reduces the number of stall cycles caused by the reads (which reduces the texture read bottleneck).

3.3.3. Render the water effect to a texture

Due to the heavy use of the fragment shader to produce the effect, the demo is fragment shader limited. To reduce the overhead of this bottle neck, the water effect can be rendered to a texture at a lower resolution and then applied to the water plane during the final render pass. This technique benefits the speed of the demonstration by reducing the number of fragments that are rendered using the water effect. This can be further reduced by rendering objects that will obscure areas of the water in the final render pass, such as the demo's terrain and boat. Although the introduction of more objects to the render can improve the speed of the water effect, the inaccuracies caused by mapping the texture to the final water plane can result in artefacts around the edges of models that were used during the low resolution pass. This is generally not that noticeable, providing the shaders used for the additional objects in the low resolution pass are the same as the shaders used in the final render (e.g. rendering models without lighting during the low resolution pass will cause highlights around dark edges of models in the final pass, so this should be avoided). One of the best ways to steer clear of the problems caused by the scaling is to avoid drawing objects that are very detailed around their edges that overlap the water as this reduces the likelihood of artefacts occurring. In the demo, the boat is omitted from the water's render to texture pass as it is too detailed to be rendered without causing artefacts and does not give as big a benefit as the terrain does when covering areas of the water (Figure 16 & Figure 17).

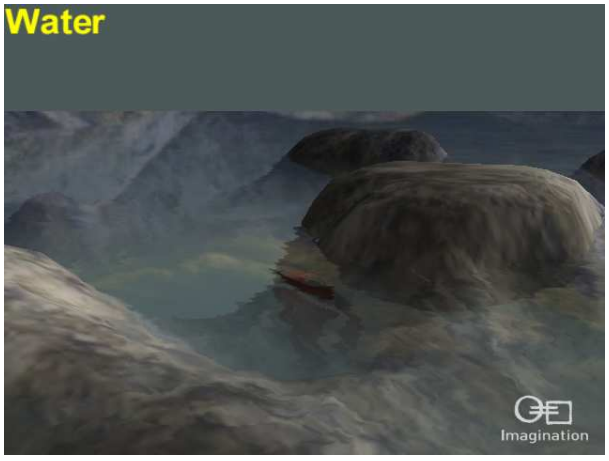

Figure 16 Low resolution water render

Figure 17 Low resolution render applied to the final plane

When rendering to a texture at a 256x256 resolution and performing the final render pass to a 640x480 screen, the reduction in quality is only slightly noticeable, but the performance is increased by ~18%.

3.3.4. Removing artefacts at the water's edge

One of the biggest problems with shader effects that perturb texture coordinates is the lack of control over the end texel that is chosen. Due to the clipping that is implemented in the reflection and refraction render passes, it is very easy for artefacts to appear along the edges of objects intersecting the water plane when the sampled texel is taken from inside the object. The colour of the texel inside the object is usually either the colour of an object that lies behind the object, but has not been clipped, or the clearing colour. The simplest workaround to this problem is to offset the clipping plane that is being used for the render pass further back along the plane's direction (in the demo's case, along the positive y-axis by default). In the case of the refraction render pass, this will cause some of the geometry above the water to be included in the rendered image, which, although not an accurate solution, means that perturbed texture coordinates into the object will cause the colour from just above the surface to be used (Figure 18). Despite being inaccurate, this fix allows the majority of the artefacts to be removed from the effect for very little additional computation. The downside of this method is that the reflection and refraction textures will include some fragments that should have been omitted (e.g. along the left and right sides of objects facing the camera there may be reflected/refracted pixels of objects that are not actually there).

Figure 18 Full effect using artefact fix

Another way to compensate for the artefacts, and improve the aesthetics of the effect, is to use fins or particle effects along the edges of objects intersecting the water to give the appearance of a wake where the water is colliding with the objects. The drawback of these techniques is that they both require the program to know where in the scene objects are intersecting the water, which can be very expensive if the water height is changing or objects in the water are moving dynamically.

4. Reference Material & Contact Details

POWERVR Public SDKs can be found on the Imagination Technologies website:

http://www.imgtec.com

Additional OpenGL-ES Programming information can be found on the Khronos Website:

http://www.khronos.org/

Developer Community Forums are available:

http://www.khronos.org/message_boards/

Additional information and Technical Support is available from POWERVR Technical Support who can be reached on the following email address:

devtech@imgtec.com