Ambiance
Real-Time Non-Interactive Graphics Demonstration
COMP3901 Special Project A

Xi Chen - xic@cse.unsw.edu.au

Semester 2 2011

Abstract

The focus of this project is to develop a graphics demo, using some of the latest and advanced techniques involved in computer graphics. Each of the scenes in the demo, along with details and other technical components of the demo will be discussed in detail, concluding with specifications, testing results, and possible future improvements.

Introduction

Ambiance is a demo, a non interactive multimedia presentation rendered in real-time using computer graphics techniques. Demos can be used for video card and system benchmarking, but are usually just for artistic purposes. Ambiance draws inspiration from demoscene productions, and video card benchmarking software such as Futuremark's 3DMark^{TM}benchmark products. While most demoscene demos often have extremely small size restrictions, Ambiance will not consider such restrictions.

This project will delve into some of the advanced techniques in the field of computer graphics. Examples include:

- HDR High Dynamic Range rendering
- Bump & Parallax Offset mapping
- Volume rendering using GPU raycasting [1]
- Real-time planetary atmospheric scattering [2]
- HDR Environment cube mapping & HDR Irradiance cube mapping
- Quaternion Julia set fractals using GPU distance raymarching [3] [4]
- Prodecural GPU generated terrain [7]
- Height-based fog [9]
- Subsurface diffusion of skin with physically based diffusion profile [10]
- Skin specular reflection using physically based BDRF [10]
- Texture-based BDRF functions
- Fresnel reflections
- Texture-based GPU vertex displacement
- Post-process bloom effect in HDR
- Post-process colour adjustment, chromatic dispersion & film grain
- Real-time image-based anti-aliasing using Morphological Anti-aliasing
- Spline-interpolated camera movement

Goal

The primary goals of this project are firstly through research & implementation, to gain an understanding of various advanced modern computer graphics algorithms, approaches and techniques, and secondly to develop a graphics demonstration program.

The secondary goal of implementing this demo is to develop a graphics engine framework, which can be used and re-used for any future games and applications.

Technical

Ambiance uses the DirectX 9.0c API, with most shaders using HLSL Shader Model 2.0 and the more complicated ones making use of HLSL 3.0 features.

The DirectX API has been chosen over the alternative OpenGL rendering library for a number of reasons; DirectX has more integrated and matured support for vertex and pixel shaders, which are the main part of this demo. Also, currently DirectX is the most popular graphics library in the games industry, and learning this API could be of future use.

The DirectX 9.0 API has been chosen over the more recent versions DirectX 10 and 11, for a few reasons. Firstly, the newer versions of these APIs are only supported on Microsoft Windows Vista and Windows 7 respectively, and have no support on windows XP. Secondly, these new versions of the API substantially change the syntax, while offering very few extra features over DirectX 9.0. As of writing, many game companies choose to ship their product with two executables, one for DirectX 9 and one for DirectX 11, in order to take advantage of the few new features DirectX 11 offers while keeping compatibility with DirectX 9.

The C++ language has been chosen for its balance of speed, support and features. The language's good support for object-oriented programming and its speed almost being as fast as C makes C++ an ideal choice for this project. C++ was chosen over C for its object-oriented support, and chosen over C# XNA Framework for its speed; since it's a graphics demo with minimal game code, the features in the XNA framework will not come into much help. Higher level languages such as Java and Visual Basic were decided to be not suitable because of the benchmarking computationally expensive nature of the project, and the lack of support they have for DirectX.

The demo's sound will use the FMOD 3.75 library, and the Open-source Asset Importing (Assimp) library will be used for model loading and rendering.

Scene: Solar System

Mercury

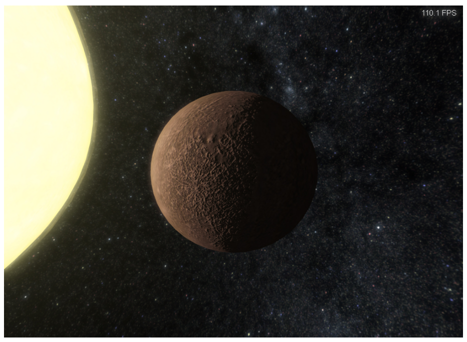


Figure 1: Mercury during development

Mercury is the planet closest to the sun, the first planet developed. The planet's surface colour texture map comes from [11]. The planet's lighting model uses DOT3 bump mapping, along with wrapped diffuse lighting.

Diffuse Wrapping

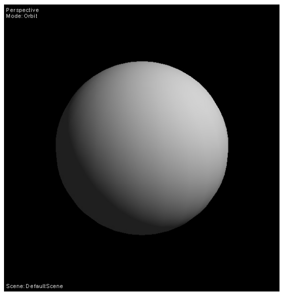


Figure 2: Lambertian diffuse lighting

$$d(\vec{n}, \vec{l}) = \vec{n} \cdot \vec{l}$$

The standard lambertian diffuse lighting model did not work well. This could be because Mercury (along with other inner planets) is so close to the Sun. Even at part of the planet that have normals facing almost right angles to the sun, the size of the sun combined with the planet's close distance to such a large source of light, the lighting is still quite bright compared to the standard lambertian model, which gives a near-zero amount of diffuse light reflection. In addition to this, mercury has an albedo that is relatively constant in brightness, regardless of the angle to the sun.

I tried to increase the ambient light constant, in an attempt to wrap some light around the planet. This worked to some extent, but the side effect is that the dark side of the planet has been significantly brightened from the ambient light. To solve this, a new lighting model was used; wrapped diffuse lighting [12].

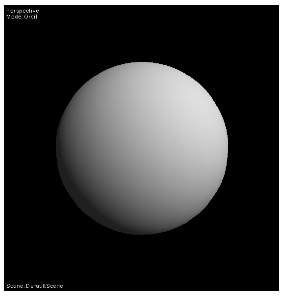


Figure 3: Wrapped diffuse lighting

$$d(\vec{n}, \vec{l}) = \frac{(\vec{n} \cdot \vec{l} + w)}{1 + w}$$

Originally used as a cheap hacky method for subsurface scattering, this method off-sets the diffuse lighting to "wrap" light around the object to places where it would normally be very dark.

To get a more constant albedo and a more sharper transition from the lit side to the unlit side of the sphere, the result of the warpped diffuse lighting model is raised to a power. HLSL shader colors are stored as a floating point value from 0.0 to 1.0, so a power value k of 0.47 was chosen to increase the "contrast" of the diffuse lighting gradient.

This takes us to the lighting model I used for most of my planets:

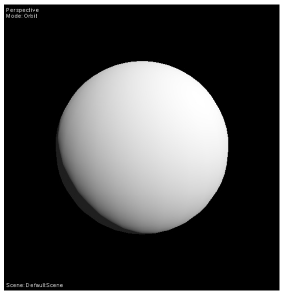


Figure 4: Contrasted wrapped diffuse lighting

$$d(\vec{n},\vec{l}) = \left(\frac{(\vec{n}\cdot\vec{l}+w)}{1+w}\right)^k$$

DOT3 Bump Mapping

To increase the realism of the lighting model, I implemented DOT3 Bump Mapping, commonly referred to as "normal mapping". Tangent space normal data is stored and loaded in the pixel shader as a texture. The vertex shader builds the world-to-tangent space transformation matrix from the normal, tangent and binormal vectors, and then transforms the light and normal vectors before passing them to the pixel shader. The pixel shader recieves the normal and light vector, and calculates the lighting completely in tangent space.

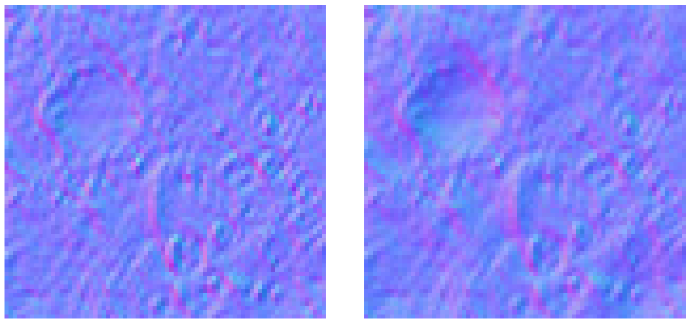

(a) normal (b) normal overlayed

Figure 5: Enhanced normals

The normal map for Mercury was calculated from the diffuse texture, using the NVIDIA Normal Map Filter, from NVIDIA Texture Tools for Adobe Photoshop, which estimates the normal map based on the image RGB intensity. This filter, however, is only able to calculate high-frequency normals, and low-frequency normal detail such as big mountains and canyons do not appear.

To enhance the quality of the normal map, I used the technique employed in [13], which repeatedly overlays blurred versions of the high frequency normal map, similar to the adding of noise function octaves in generating fractional Brownian motion. Then, the new overlayed normals are normalized by applying the "normalize" function in the NVIDIA Normal Map Filter.

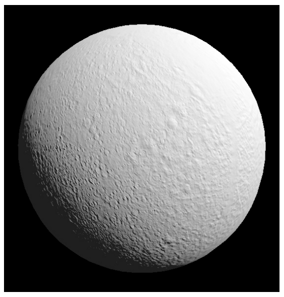
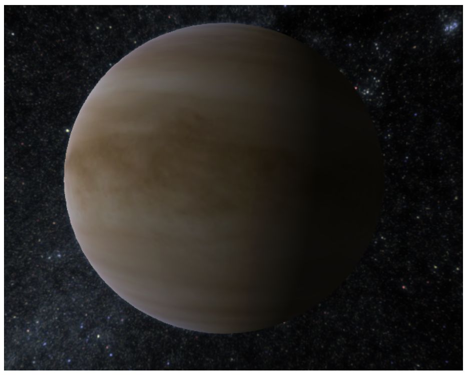


Figure 6: Normal mapped sphere

Venus


(a) Dev. screenshot

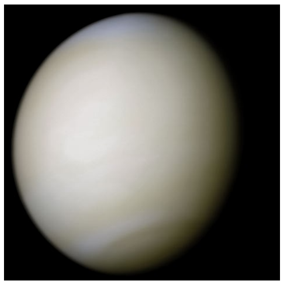

(b) NASA Photo

Figure 7: Venus

Venus is the second most closest planet to the Sun. The planet's surface colour map was made from a combination of two textures. Venus has a relatively constant albedo, but also a gradual fade at the transition from the lit to the unlit side.

A lighting model similar to the diffuse wrapping approach to lighting was used, as was explained in the Mercury section. The justification for diffuse wrapping being used for Venus, is rather that venus has a thick atmosphere which covers the planet's surface when viewd from space, and light scatters in this atmosphere, beyond a perpendicular angle of incidence.

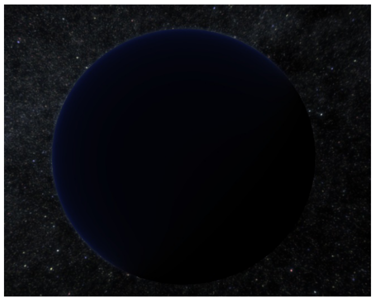


Figure 8: Venus' atmospheric scattering

I also added very slight subtle blue atmospheric scattering around the planet, as seen in some reference photos of Venus. My application's full atmospheric scattering shader comes in two parts; the ground scattering, which calculates the atmopsheric scattering for the actual planet, and the skydome, which is rendered slightly above the planet sphere. Since the visible "surface" of Venus is atmosphere already (the actual rocky ground completely covered by the planet's atmosphere), the skydome was rendered very faint. This is to avoid the default skydome look, which makes the atmosphere look like its above the surface of the planet.

The atmospheric scattering shader will be explained in detail in the Earth section.

Earth

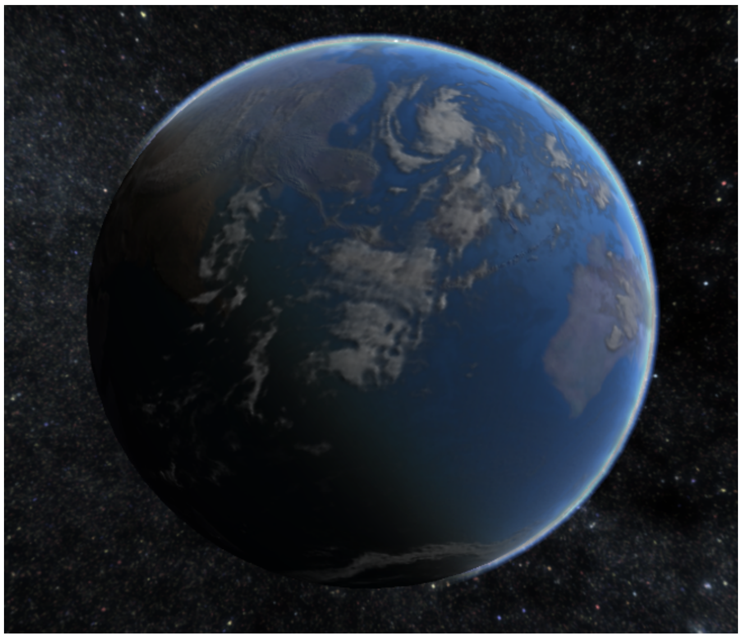


Figure 9: The Earth during development

Earth, our home planet, is the 3rd planet from the Sun. Earth was one of the most challenging planets to render, having a rich atmosphere and moving clouds covering a surface made up of specular reflecting water and land with complex distinguishable features. The city lights from the unlit, night side of the planet are also visible.

Wrapped Diffuse lighting

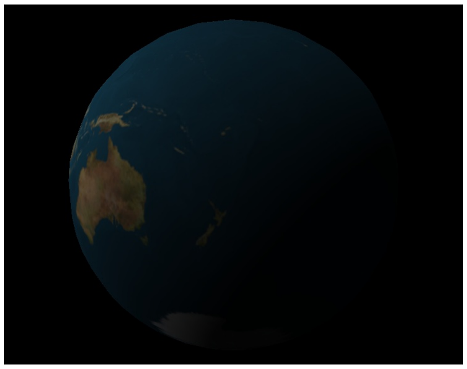


Figure 10: Wrapped diffuse lighting

$$\text{diffuse}(\vec{n}, \vec{l}) = \vec{n} \cdot \vec{l}$$

To start off, the colour map obtained from [11] is multiplied by ordinary wrapped diffuse lighting (diffuse wrapping explained in Mercury section). This produces a nice smooth gradient on the sphere.

Night Lights

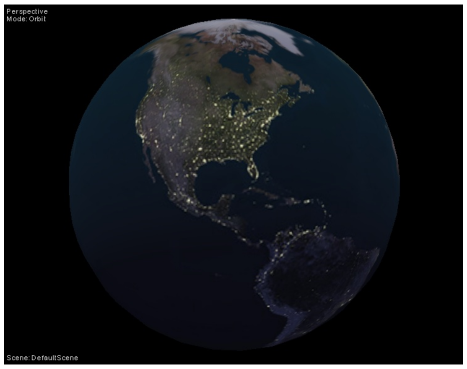


Figure 11: Night side: naive method

$$\text{night}(\vec{n}, \vec{l}) = 1 - \vec{n} \cdot \vec{l}$$

The unlit side of the earth is not completely black, and the light from busy cities are visible from space. My first model for this uses $1 - \text{diffuse}(\vec{n}, \vec{l})$. This however, produces a transitioning problem, where the city lights from the night texture are clearly visible in places where it's not quite night time yet.

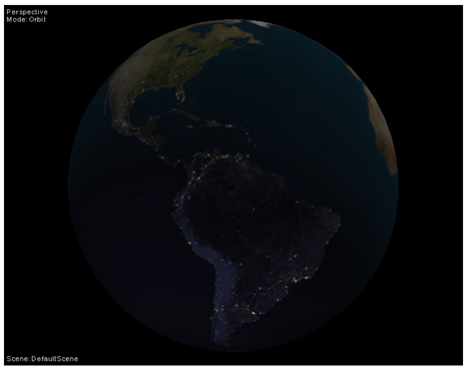


Figure 12: Night side: better method

$$\text{night}(\vec{n}, \vec{l}) = \vec{n} \cdot -\vec{l}$$

$$\text{colour}(\vec{n}, \vec{l}) = \text{diffuse}(\vec{n}, \vec{l}) + \text{night}(\vec{n}, \vec{l})$$

The solution I employed to solve this problem is to use calculate a "backlight", to which the night texture is multiplied. Then, the result is added to the day-time wrapped diffuse lighting. This made sure that the people on earth didn't turn their lights on too early in the day.

Specular gloss mapping

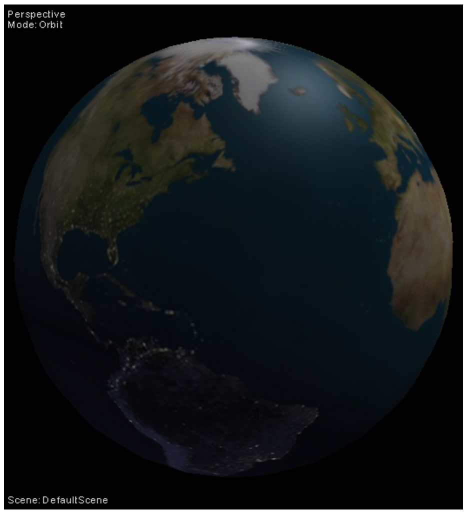
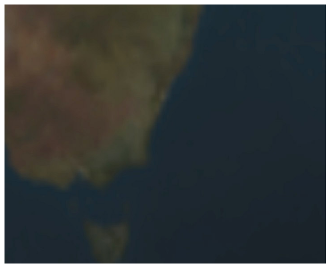


Figure 13: Specular reflection of oceans

The ocean parts of Earth give off specular reflection, while the land does not. In order to effectively render this, a gloss map texture is multiplied with ordinary specular reflection.

Bump Mapping and Parallax Offset Mapping


(a) vertex normals

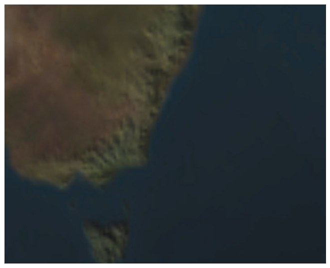

(b) DOT3 bump mapping

Figure 14: Bump mapping

DOT3 bump mapping was used to enhance the detail of the terrain. This technique is explained in detail in the Mercury section.

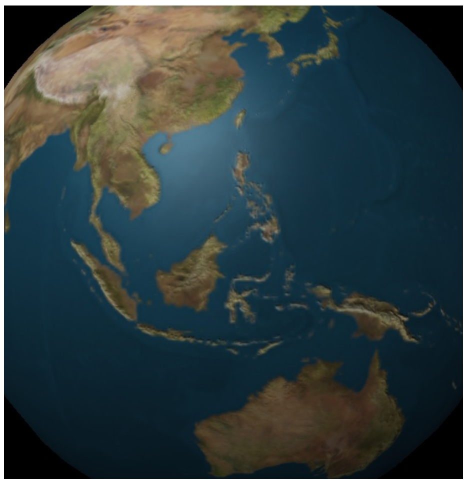


Figure 15: Bump + Parallax offset mapping

On top of this, I implemented parallax offset mapping; A height texture is used, by which the UV co-ordinates were offset in the direction of the camera.

Cloud map

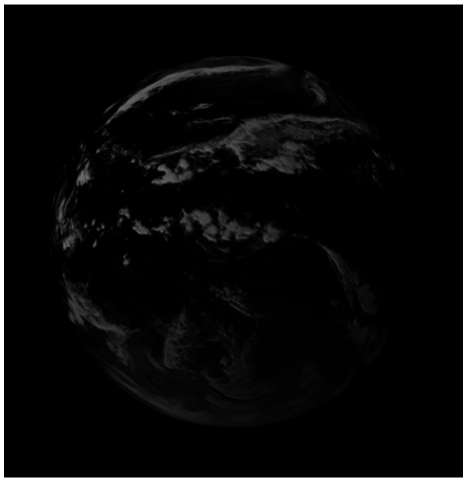


Figure 16: Bump mapped diffuse lighted cloud layer

Cloud is rendered as a slightly larger sphere rendered on top of the earth surface sphere, with alpha-blending enabled. The cloud layer also uses a cloud normal map texture, and is diffuse lighted. The cloud layer alpha value is set using a cloud transparency texture, before being blended with the slightly smaller earth sphere.

Atmosphere Scattering

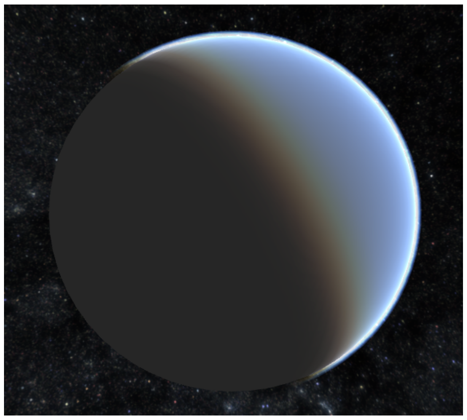


Figure 17: Atmospheric scattering

The Earth is surrounded by a rich atmosphere, clearly visible from space as a bright haze around the planet. My demo uses a real-time GPU-calculated atmospheric scattering algorithm, from [2]. The atmospheric scattering equations involve a double integral over the length of the view ray passing through the atmosphere. The scattering algorithm does this using a 2-D lookup table, indexed by point height and angle from zenith to target object outside of atmosphere.

Scattering Equations

The out-scattering function is defined as

$$t(P_a P_b, \lambda) = 4\pi \cdot K(\lambda) \cdot \int_{P_a}^{P_b} e^{\frac{-h}{H_0}} ds$$

Where λ is the wavelength, P_a and P_b are the entry and end points of the view vector through the atmosphere respectively, $K(\lambda)$ is the scattering constant ($\frac{1}{\lambda^4}$ for Rayleigh scattering), h is the sample point height, and H_0 is the scale height.

Out-scattering defines how much light is lost, scattered out, by particles through the atmosphere for a given interval. The function is a double integral which must be solved on the GPU in real-time. Details on the method used will be explained shortly.

The in-scattering function is defined as

$$I_v(\lambda) = I_s(\lambda) K(\lambda) F(\theta, g) \cdot \int_{P_a}^{P_b} \left(e^{\frac{-h}{H_0}} \cdot e^{-t(PP_c,\lambda)-t(PP_a,\lambda)}\right)$$

Where $I_s(\lambda)$ is the intensity of the sunlight, $K(\lambda)$ is the scattering constant (see above), PP_c is the ray from the point to the sun, PP_a is the ray from the sample point to the camera, and $F(\theta, g)$ is the phase function defined as

$$F(\theta, g) = \frac{3(1-g^2)}{2(2+g^2)} \cdot \frac{1+cos^2(\theta)}{1+g^2-2g \cdot cos(\theta)}$$

Where θ and g are the phase angle between the two rays and the constant of symmetry respectively.

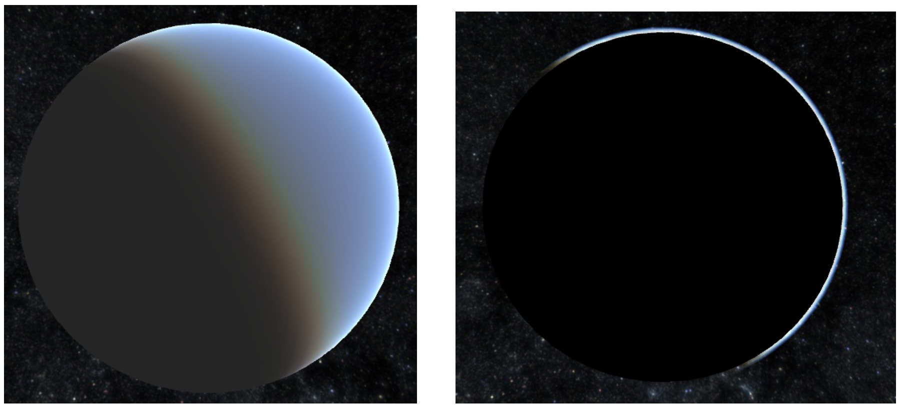
Real-Time GPU Calculation

The atmospheric scattering functions can be estimated real-time on the GPU by estimating the "look-up table" function

$$\text{table}(h, \alpha) = \int_{h_0}^{h_\alpha} e^{\frac{-h}{H_0}} ds$$

Which, given h, the height of the point above the earth's surface, and α, the angle from the point to the object outside of the atmosphere, returns result of the optical depth integral.

This estimation is done by using the exponentiation function for the α axis, and a pre-calculated curve-fitted polynomial function for the other h axis.



(a) Ground sphere (b) Skydome

Figure 18: The two parts of the atmosphere

Surface Scattering
The surface scattering equation used is

$$I'_v(\lambda) = I_v(\lambda) + I_e(\lambda) \cdot e^{-t(P_a P_b, \lambda)}$$

Where λ is the wavelength, $I_v(\lambda)$ is the in-scattering function, $I_e(\lambda)$ is the light reflected off the surface, and $t(P_a P_b, \lambda)$ is the out-scattering equation.

The surface scattering is rendered as two alpha-blended spheres same size as the planet, rendered with depth testing off. One sphere is used for attenuation, which uses multiplicative blending. The other sphere is used for the scattering, which uses additive blending.

The Skydome
The skydome is rendered as a alpha-blended sphere slightly larger than the planet, with front-face culling and depth-test enabled. This will render an "inside out" hemisphere that is behind the actual planet, except for a ring around the planet (since the sphere is slightly larger). Since only the back-face is drawn, the z-value of the skydome sphere effectively tells us exactly where the camera view ray leaves from the atmosphere.

Cloud Sphere Interference Problem

Since the cloud sphere is bigger than the planet, and only the back-side of the skydome is rendered with depth testing, at the edges of the sphere the cloud layer can interfere with the skydome.

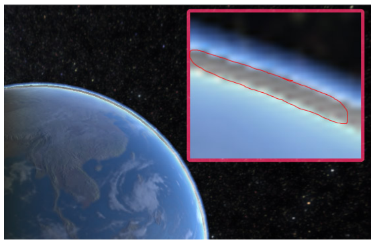
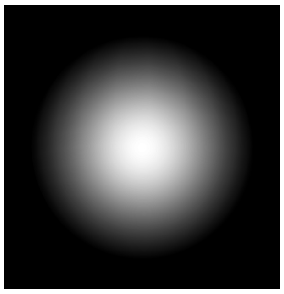
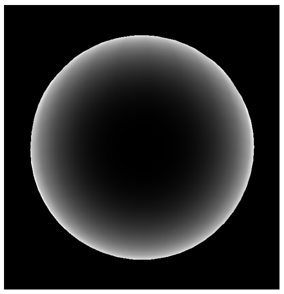


Figure 19: The cloud sphere interferes with the skydome

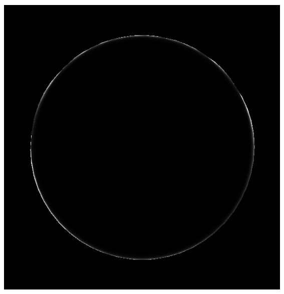
To solve this, I obtained a camera-aligned sharp white "ring" around the sphere by calculating one minus the dot product of of the view vector and the sphere normal vector, then raising it to a power. I then subtracted this "ring" from the alpha value of the cloud sphere.



(a) $\vec{v} \cdot \vec{l}$



(b) $1 - (\vec{v} \cdot \vec{l})$



(c) $(1 - (\vec{v} \cdot \vec{l}))^k$

Figure 20: The cloud alpha falloff ring

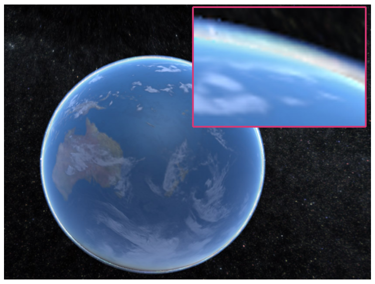


Figure 21: The cloud sphere with alpha fadeout ring

This view-aligned ring technique was later used on gas giant lighting models.

Mars

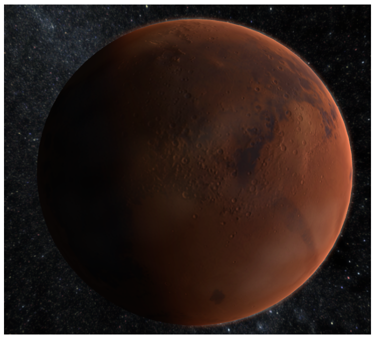


Figure 22: Mars during development

Mars is the 4th planet from the sun, and is known for its large geographical features and its distinctive red colour. The shader techniques used to render mars were very similar to the techniques used to render the Earth; paramax bump mapping for the land and real-time atmospheric scattering for the atmosphere. To model the red scattering, the colour of the incoming sunlight in the atmospheric scattering shader calculations was modified from white to a suitable reddish colour. The brightness of the skydome was also significantly reduced.

Jupiter

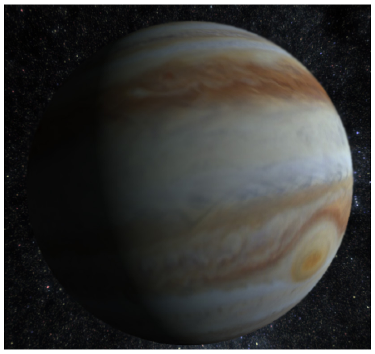


Figure 23: Jupiter during development

Jupiter is the largest planet in the solar system. Rendering of jupiter has two new techniques. The lighting model of Jupiter uses an extension of diffuse wrap lighting [12], which gives a slight tint of colour at the wrapped diffuse areas using the smoothstep function. The diffuse texture of Jupiter is animated using NASA video footage of Jupiter's surface map.

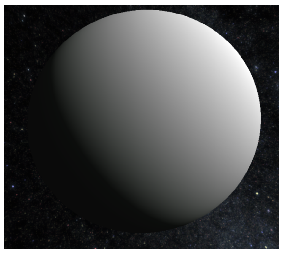


Figure 24: Lighting of Jupiter showing slight green tint

For the animated diffuse texture, the 5 video keyframes of the NASA video footage were extracted using VLC Media Player's file output option. The current the previous video frame are then calculated based on time on the CPU, and those two frames sent to the GPU to be interpolated.

For the GPU inter-frame interpolation, first a naive solution was implemented; linearly interpolate between the current and next video frame. While this was a good starting base and gave reasonably good results if the time-scale was set to very high, at slower video playback speeds the "ghosting" of features from one frame fading into another was clearly visible.

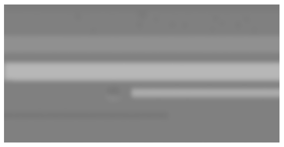


Figure 25: UV Displacement Velocity Map

In order to minimise this "ghosting" effect and for video frame to seamlessly transition onto the next, a UV-displacement velocity map was used. A texture was hand crated, where a pixel value of 128 meant the feature at this place does not move, a pixel value of higher than 128 meant the feature moves to the right between this frame and next frame, and vice versa. The pixel shader looks up this value, then displaces the UV accordingly. This gives the effect of the video frame picture moving in the direction and with the speed specified by the displacement texture.

In order to blend the current video frame into the next, the UV of the current frame is displaced by time t, while the video UV texture of the next frame is displaced by $1 - t$. The two is then linearly interpolated between the two video frames.

This works well for the Jupiter surface video, as the features are nearly always moving horizontally, and although different rings move at different speeds, the features of Jupiter's surface spin around the planet at a reasonably constant rate.

Saturn

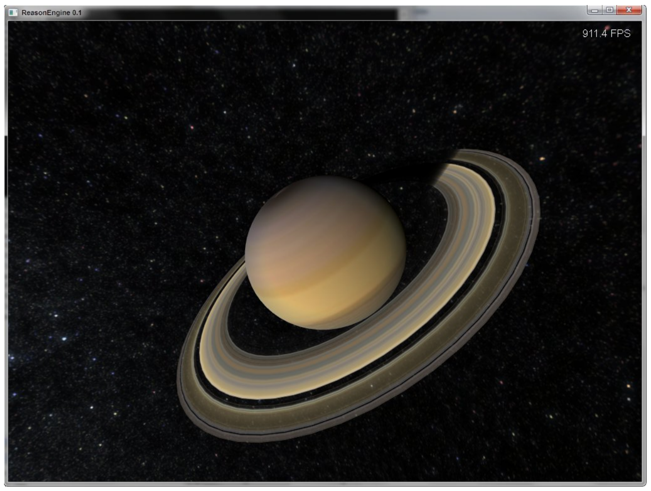


Figure 26: Saturn during development

Saturn is the second closest gas giant from the sun, and is well known for its rings and visibly oblong shape. Both the planet and its rings present challenges to the rendering of this planet.

Lighting model - planet body

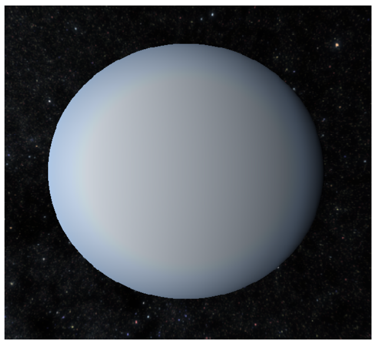


Figure 27: Saturn lighting

Because Saturn, like Jupiter, is a gas giant with no hard surface, reference satellite photos has shown the planet having a slight blue-green tint of colour from subsurface diffusion around the edges. Even bright lit side of the planet is very soft at the edges. The lighting model used for saturn uses colour-tinted wrapped diffuse lighting, same as Jupiter. A very slightly dark "fade-out" ring around the edges of the sphere was added, by multiply-ing the view vector with the surface normal of the planet and exponentiating that (see "Cloud Sphere Interference Problem" section from Earth). These two techniques result in a softer appearance than straight out Lambertian lighting.

Lighting model - rings

The rings of Saturn were rendered using a texture-lookup based BRDF function. Three lighting textures were used, giving three values for every point on the ring; the backscattered light intensity value, the forwardscattered light intensity value, and an unlit scattering value. These three values are then linearly interpolated depending on phase angle and the side of the ring plane, giving our BRDF function:

$$f_{\text{lit}}(\vec{w_i}, \vec{w_o}) = \text{lerp}(L_f, L_b, \vec{w_i} \cdot \vec{w_o})$$

$$f_{\text{unlit}}(\vec{w_i}, \vec{w_o}) = \text{lerp}(L_u, L_b, \vec{w_i} \cdot \vec{w_o})$$

where $\vec{w_i}$ and $\vec{w_o}$ are in light and view directions respectively, L_f, L_b and L_u are the forwardscatter, backscatter and unlitscatter intensity values given by the three textures respectively, and lerp is a simple linear interpolation function.

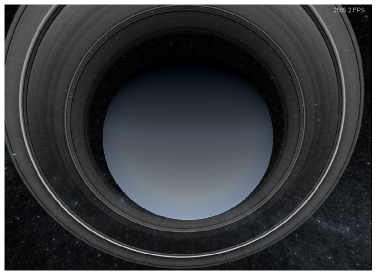


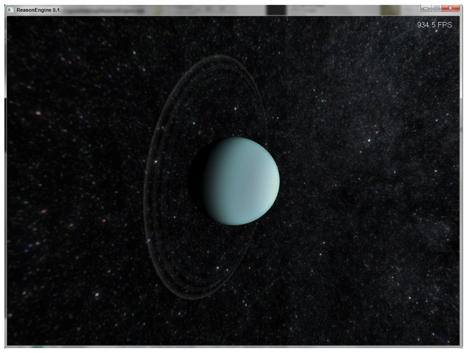
Figure 28: Ring lighting showing unlit side

In addition to the three lighting textures, two additional colour and transparency alpha maps were also used.

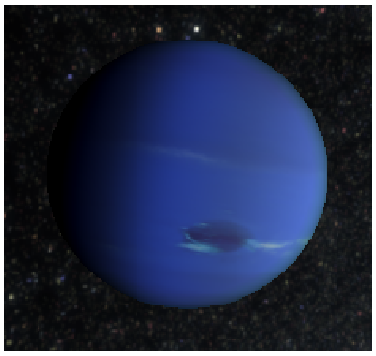
Shadow - rings

Saturn casts a shadow on its rings, which was calculated analytically on the GPU from the pixel shader of the ring. The ray from the pixel position to the light is collided with the planet sphere using point-line distance. If the distance from the ray to the origin of the planet was lower than the planet's radius, then that part of the ring is covered by shadow,, and vice versa. Soft shadow is estimated by taking the absolute difference of the distance from the ray to the planet origin from the planet's radius, and exponentiating the resulting value, before adding it to the "hard shadow". This results in a softer fade-out around the shadow.

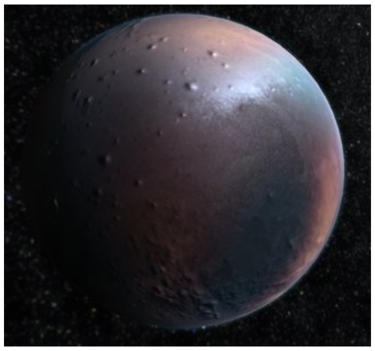
Uranus, Neptune & Pluto



(a) Uranus



(b) Neptune



(c) Pluto

Figure 29: The outer 3 planets during development

The techniques used for Uranus were similar to Saturn. since the rings of Uranus are much less visible than Saturn's, simple flat shading is used instead of the texture-lookup BDRF lighting function used for Saturn's rings.

The rendering techniques used for neptune were also similar to Saturn; The colour-tinted diffuse wrap lighting and darkened edge were used to give a soft appearance.

Pluto's rendering techniques are similar to that of Earth and Mars. Since no one knows yet what Pluto looks like in high resolution, the diffuse colour map was made from NASA low-resolution maps and rocky planet texture maps. I wanted pluto to look icey, so I added gloss-mapped specular lighting to the planet's surface. The planet also has a faint icey-coloured atmosphere.

Scene: Elevated

This scene is inspired by the Breakpoint 2009 winning PC 4k demo, elevated by Rgba. The main features of this scene are completely GPU-generated procedural terrain[7], and Preetham's sky [14].

Terrain

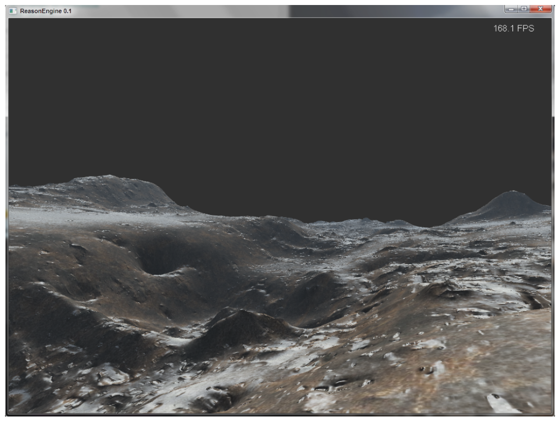


Figure 30: Procedurally GPU generated terrain

The scene's terrain is generated completely on the GPU, from height geometry to lighting, texturing, and fog.

Terrain - Height function and Geometry

The height function, which is used to displace the terrain geometry, is calculated using a modified fractal Brownian motion noise function. An ordinary fractal Brownian motion noise function, or perlin noise, adds multiple octaves of white noise with different frequencies, with the most popular version doubling the frequency each octave:

$$Z_{\text{fbm}}(x, y) = \sum_{i=0}^{n} w_i f(2^i x, 2^i y), w_i = \frac{1}{2^i}$$

The height function used for ambiance adds a modification on top of this function, which attempts to model features of erosion, resulting in much more realistic looking terrain. It weights the normal FBM by the squared length of the unweighted sum of the derivatives from all the octaves below the current octave.

$$Z(x, y) = \sum_{i=0}^{n} \frac{1}{2^i} \frac{f(2^i x, 2^i y)}{1 + \left(\sum_{j=0}^{i} \frac{\partial f}{\partial x}(2^j x, 2^j y)\right)^2 + \left(\sum_{j=0}^{i} \frac{\partial f}{\partial y}(2^j x, 2^j y)\right)^2}$$

Where $f(x, y)$ is a 2-dimensional quintic interpolated white noise function in both equations, which takes in x, y as integers and returns a random floating point number from -1.0 to 1.0. The white noise is implemented on the GPU using lookups based on the given x,y co-ordinates into a noise texture, which was made using the noise filter in Adobe® Photoshop®.

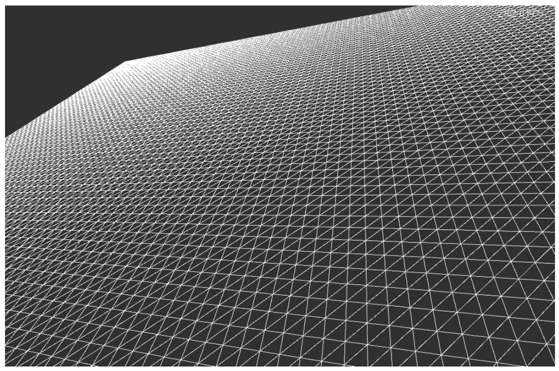


Figure 31: Terrain - a flat tesselated quad

The terrain is sent to the DirectX API from the CPU as a flat tesselated square quad. The vertex shader then evaluates the height function (mentioned above) using the (x,z) position of the quad vertex, and displaces the y (height) position of the vertex with the value of the terrain height function. Since the vertex shader needs to evaluate the height function, which needs to evaluate the white noise function multiple times, which requires a texture lookup, the vertex shader texture look-up capabilities of vertex shader version 3.0 is required. Vertex shader 2.0 does not allow for texture taps from within the vertex shader.

Terrain - Lighting - Diffuse

Lighting is calculated completely in the pixel shader. In order to calculate Lambertian diffuse lighting, the normal to the height function needs to be calculated. In order to calculate the normal to the surface, the derivatives with respect to x and y of the surface must be calculated. I have implemented two methods of doing this (although the demo only uses one, of course): central differences and analytical derivatives.

The central differences method takes 4 extra samples of the height function, up, down, left right. It uses the differences between the up/down and left/right samples to estimate the two derivatives of the height function respectively. The sampling distance ε is given as a parameter.

The analytical method mathematically calculates the derivatives of the height function. The white noise function $f_w(x, y)$ is interpolated in the interpolated noise function $f(x, y)$ using the polynomial:

$$k(x) = 6x^5 - 15x^4 + 10x^3, k(y) = 6y^5 - 15y^4 + 10y^3$$

Which can be rather easily derived:

$$\frac{\partial k}{\partial x} = 30x^4 - 60x^3 + 30x^2, \frac{\partial k}{\partial y} = 30y^4 - 60y^3 + 30y^2$$

$$\frac{\partial^2 k}{\partial x^2} = 120x^3 - 180x^2 + 60x, \frac{\partial^2 k}{\partial y^2} = 120y^3 - 180y^2 + 60y$$

And interpolating the white noise function with these polynomials gives:

let a, b, c, d be $f_w(x, y), f_w(x+1, y), f_w(x, y+1), f_w(x+1, y+1)$ respectively:

$$k_0 = a$$
$$k_1 = b - a$$
$$k_2 = c - a$$
$$k_3 = a - b - c + d$$
$$f(x, y) = k_0 + k_1 u + k_2 v + k_3 uv$$
$$\frac{\partial f}{\partial x} = \frac{\partial k}{\partial x}(k_1 + k_3 v), \frac{\partial f}{\partial y} = \frac{\partial k}{\partial y}(k_2 + k_3 u)$$
$$\frac{\partial^2 f}{\partial x^2} = \frac{\partial^2 f}{\partial x^2}(k_1 + k_3 v), \frac{\partial^2 f}{\partial y^2} = \frac{\partial^2 f}{\partial y^2}(k_2 + k_3 u)$$
$$\frac{\partial^2 f}{\partial xy} = \frac{\partial k}{\partial x}\frac{\partial k}{\partial y}k_3, \frac{\partial^2 f}{\partial yx} = \frac{\partial k}{\partial y}\frac{\partial k}{\partial x}k_3,$$

More details given in [14][16]. Thus, this gives us the first and second derivatives when we evaluate the interpolated white noise function $f(x, y)$. In order to get the derivative of the full modified FBM (fractal Brownian motion) perlin noise height function, we must use the known first and second derivatives of the white noise function and calculate mathematically the derivative of the modified FBM function Z(x, y):

$$Z(x, y) = \sum_{i=0}^{n} \frac{1}{2^i} Z'(i, x, y) = \sum_{i=0}^{n} \frac{1}{2^i} \frac{f(2^i x, 2^i y)}{1 + \left(\sum_{j=0}^{i} \frac{\partial f}{\partial x}(2^j x, 2^j y)\right)^2 + \left(\sum_{j=0}^{i} \frac{\partial f}{\partial y}(2^j x, 2^j y)\right)^2}$$

Although it seems quite messy and complicated, one can see that deriving this is just applications of the quotient rule and chain rule.

$$\text{let } u(x, y) = f(2^i x, 2^i y)$$
$$\frac{\partial u}{\partial x} = \frac{\partial f}{\partial x}(2^i x, 2^i y) \cdot 2^i$$
$$\frac{\partial u}{\partial y} = \frac{\partial f}{\partial y}(2^i x, 2^i y) \cdot 2^i$$

$$\text{let } hx(x,y) = \sum_{j=0}^{i} \frac{\partial f}{\partial x}(2^j x, 2^j y)$$

$$\text{let } hy(x,y) = \sum_{j=0}^{i} \frac{\partial f}{\partial y}(2^j x, 2^j y)$$

$$\text{let } v(x,y) = 1 + hx(x,y)^2 + hy(x,y)^2$$

Deriving these two functions $u(x,y)$ and $v(x,y)$ gives us:

$$\frac{\partial v}{\partial x} = 1 + 2hx(x,y)\frac{\partial hx}{\partial x} + 2hy(x,y)\frac{\partial hy}{\partial x}$$
$$\frac{\partial v}{\partial y} = 1 + 2hx(x,y)\frac{\partial hx}{\partial y} + 2hy(x,y)\frac{\partial hy}{\partial y}$$

$$\frac{\partial hx}{\partial x} = \sum_{j=0}^{i} \big(\frac{\partial^2 f}{\partial x^2}(2^j x, 2^j y) \cdot 2^j\big)$$

$$\frac{\partial hy}{\partial x} = \sum_{j=0}^{i} \big(\frac{\partial^2 f}{\partial yx}(2^j x, 2^j y) \cdot 2^j\big)$$

$$\frac{\partial hx}{\partial y} = \sum_{j=0}^{i} \big(\frac{\partial^2 f}{\partial xy}(2^j x, 2^j y) \cdot 2^j\big)$$

$$\frac{\partial hy}{\partial y} = \sum_{j=0}^{i} \big(\frac{\partial^2 f}{\partial y^2}(2^j x, 2^j y) \cdot 2^j\big)$$

And finally, we have all the parts we need to differentiate our final height function $Z(x,y)$ by quotient rule:

$$\frac{\partial Z}{\partial x} = \sum_{i=0}^{n} \frac{1}{2^i}\frac{\partial Z'}{\partial x}, \frac{\partial Z}{\partial y} = \sum_{i=0}^{n} \frac{1}{2^i}\frac{\partial Z'}{\partial y}$$

$$\frac{\partial Z'}{\partial x} = \frac{v\frac{\partial u}{\partial x} - u\frac{\partial v}{\partial x}}{v^2}, \frac{\partial Z'}{\partial y} = \frac{v\frac{\partial u}{\partial y} - u\frac{\partial v}{\partial y}}{v^2}$$

And thus we have our two derivatives, from which the surface normal can be calculated.

Out of the two methods, the method that the demo makes use of is the analytical method, because it gives the highest resolution and most accurate derivatives without taking 4 extra samples of the noise function.

Terrain - Lighting - Ambient & Ambient Occlusion

Ambient occlusion is estimated by using the y component of the normal vector:

$$\text{ao} = (n_y)^k + a$$

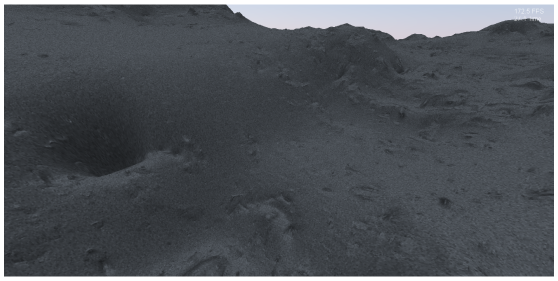


Figure 32: Terrain - ambient lighting

Total ambient lighting is calculated using a constant factor, with noise added for some detail, and a small subtle blue back-light to show some detail, all combined with the ambient occlusion term above.

Terrain - Lighting - Shadow

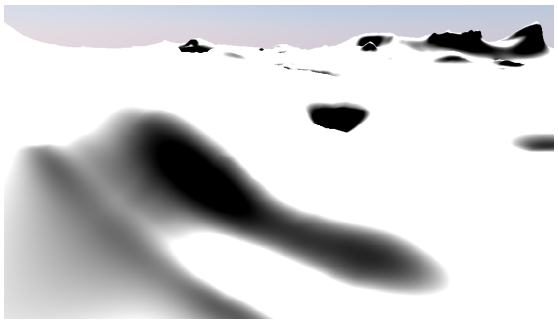


Figure 33: Terrain - shadow estimation

Terrain shadow is estimated by evaluating diffuse lighting of the terrain at a lower frequency (details in [8]).

Terrain - Texturing

The terrain uses procedural texture mapping, although the textures themselves are not procedural. The terrain procedurally fades between two textures; a "snow" texture and a "land" texture. The mix factor between these two textures are determined by:

$$f_{\text{mix}} = \text{smoothstep}(A - B \cdot n_y, C - D \cdot n_y, \text{noiseFBM} \cdot E)$$

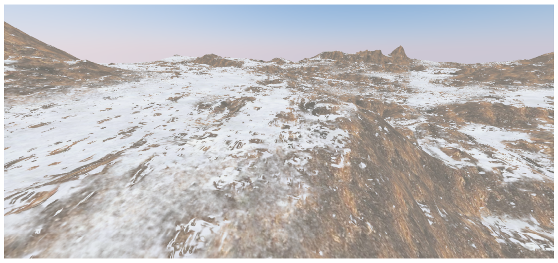


Figure 34: Terrain - texture mixing

Where A, B, C, D, E are parameters that control the results of the texturing, and noiseFBM is a regular interpolated FBM noise function f_{fbm}, and n_y is the y (height) component of the terrain normal.

Terrain - Fog

The fog used for the terrain is an analytically integrated exponential height-based fog [9]. The density of the fog falls off exponentially with regard to height:

$$d(y) = a \cdot e^{-by}$$

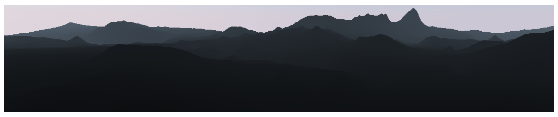


Figure 35: Terrain - fog

Plugging in the equation for the camera view ray, we get our integrated fog:

$$D = \int_{o}^{T} d(o_y + t \cdot k_y)dt = a \cdot e^{-b \cdot o_y} \frac{1 - e^{-b \cdot k_y \cdot T}}{b \cdot k_y}$$

Sky

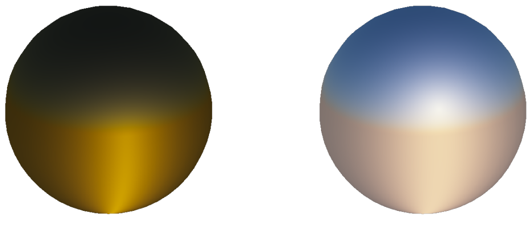

(a) Dawn (b) Day

Figure 36: Results of Preetham's Sky Model

I found an implementation [15] of Preetham's Sky Model [14] and ported it from GLSL to HLSL. It is an extension of Perez all-weather sky model from luminance only into xyY colour space. It is a simple estimation algorithm, which mathematically estimates sky colour, and involves no numerical integration. The resulting colour in xyY space is then converted into RGB space, which is done on the GPU in the pixel shader.

Flythrough

Since the terrain is not stored in a texture, but rather a 2-D mathematical function, one can add and modify off-sets to the X,Y co-ordinates used to evaluate the function. This, however can be used in a more creative way; moving X and Y every frame can give the illusion of the terrain moving, while the actual rendered terrain and camera both stay in the same spot. Furthermore, since this is a mathematical noise function, one can endlessly "move" down endless amounts of terrain using this method, while the actual rendered terrain quad and the camera remain stationary.

Scene: Chronosphere

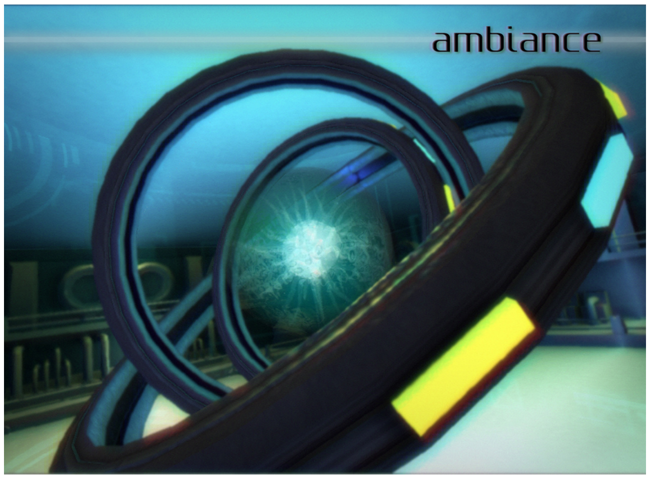


Figure 37: Chronosphere scene during development

The main feature of this scene is the GPU raycasted volume rendering of a 3-dimensional volume texture. It also features model loading using the Assimp library, and environment bump mapping with animated normal perturbation using an interpolated 4-dimensional noise function (x, y, z, time).

Volume rendering

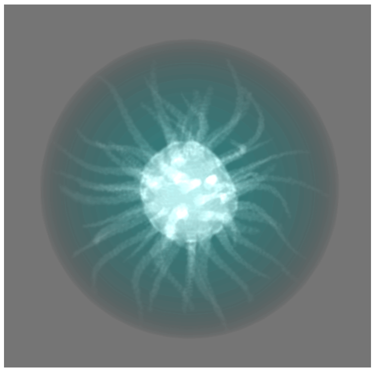


Figure 38: Realtime GPU raycasted volume rendering

The volume data is stored in a 256x256x256 voxel volume texture, which is tri-linearly interpolated by the graphics hardware through DirectX. In order to render this volume, the back-face of a sphere is rendered using front-face culling. The pixel shader program then uses the interpolated world space pixel position as the ray exit point, and then analytically calculates using sphere-ray intersection the ray entry point. This gives a ray in world-space, which the pixel shader program divides into equal segments and for each segment, takes samples of the volume data. The pixel shader program then estimates the integration of the volume density over the ray by adding the segmented samples together, multiplied by the segment length (regular trapezium integration).

A simple waving animation is added to the static volume by offsetting the sample positions by a 4-dimensional perturbation function, made using a sum of various sine / cosine functions varying over time.

Reflections

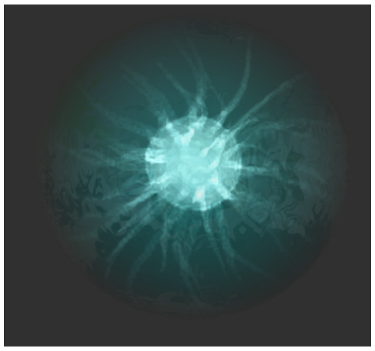


Figure 39: sphere with environment bump mapped reflection

The view and normal vectors are calculated and interpolated to the pixel shader program in world space. The pixel shader program then offsets the x, y and z components of this vector by a linearly interpolated 4-D noise. This noise function takes 16 samples of a 4-dimensional integer white noise function (implemented using a noise texture) and linearly interpolates between them. The world space x, y and z position is used to evaluate this function, and the 4th dimension used being time. This gives an animated "wavey" perturbation of the sphere normal.

This offset normal is then reflected about the sphere surface view vector, and then used for looking up into a cube environment map texture. This results in a bumpy "wavey" animated cube mapped reflective sphere appearance, and is then alpha-blended on top of the results of the volmetric raycasting.

Rings

The rings were modelled in Autodesk® 3DS Max®, manually texture mapped and textured, and ambient occlusion baked using xNormal. The rings are lit by 4 specular-diffuse point lights, positioned in various places in the scene. The ring lighting shader uses several different textures; diffuse colour texture, baked ambient occlusion texture, light exemption texture (to make sure the bright yellow/cyan squares on the outside of the ring are exempt from any lighting), and coloured specular (gloss) map texture.

The models were loaded using the Open Asset Import Library (Assimp). For a more detailed explanation of the frontend getting Assimp to work with DirectX, refer to the Model loading section.

Background

The background model was found as a free model, made by Alexandre Genovese. They were processed in 3DS Max, and ambient occlusion baked using xNormal.

Scene: Julia Sets

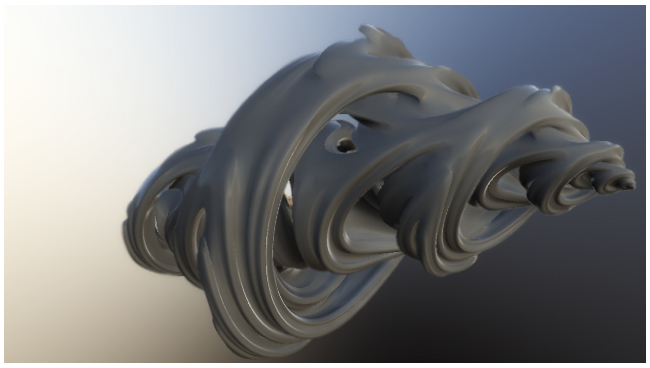


Figure 40: Realtime GPU raymarched Julia sets

This scene was inspired by the demoscene demo ”kindernoiser” by Rgba [3] (1st place combined 4k bcnparty 2007). It features a realtime distanced raymarched Julia set fractal, HDR irradiance maps, and HDR cube environment mapped fresnel reflection.

Quaternion Julia sets

Julia set fractals are calculated by finding the limit of the function given at a point, where the function repeatly displaces the point via the polynomial:

$$Z = Z^2 + C$$

For some constant vector $\vec{C}$ which defines the resulting shape of the fractal. Points which have a limit are defined to be ”inside” the factal shape, and the points which diverge to infinity are outside. Varying the constant factor $\vec{C}$ results in interesting, morphing shapes.

For Julia set images in 2 dimensions, one can calculate the above equation in 2D using the complex number system. For Julia sets to be calculated in 3 dimensions, one must calculate the above equation in a numbering system that is 3 dimensions or higher. One possible approach is to define a new numbering system with defined multiplication and addition operations. I chose to use the quaternion 4 dimensional numbering system, just setting the 4th term to 0.

This results in a 3D volume, for which there are various ways to render, from point clouds, raycasting, raymarching, polygonisation (marching cubes algorithm), voxels..etc. I chose the implement the most easiest and elegant method, distance raymarching.

Distance Raymarching

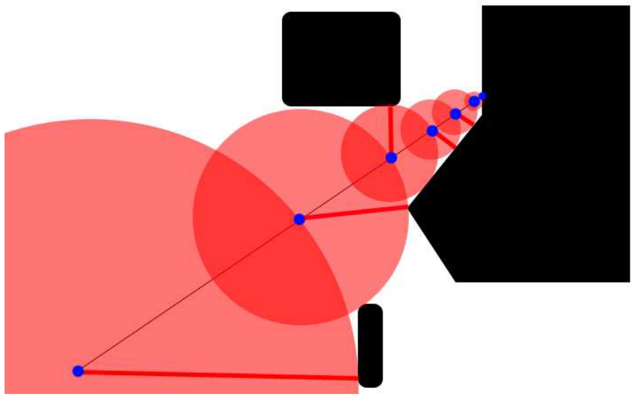


Figure 41: Distance raymarching

To raytrace into the 3D Julia set fractal volume, the simplest method would be to use brute force; a constant step length. However, to achieve better results faster, a more smarter approach could be taken. Say there exists a function which gives a underestimation of the distance to the surface of the volume given a point; since this is the smallest distance from the given point, travelling along our raytracing ray for this distance will guarantee that we will not ever hit or go past the surface. We can then evaluate the distance using new point we arrive at after we have travelled, and then travel along the ray this much again, and repeat again, and so on, until we get close enough to the surface.

This significantly reduces the number of samples required to achieve the same quality, and as sampling a fractal julia set involves repeatly applying quaternion addition and multiplication potentially hundreds of times, this is a very significant optimisation. For details see [3]

Distance Estimation

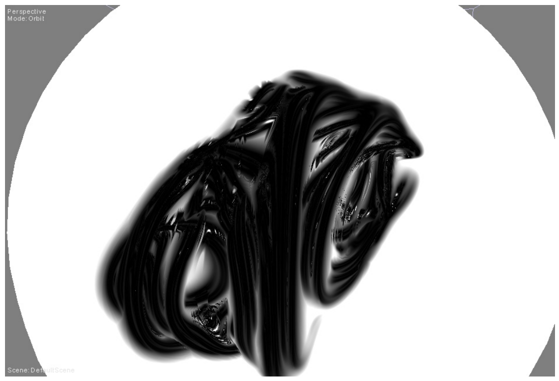


Figure 42: Distance function

A suitable distance estimation function needs to be developed in order to use the distance raymarching algorithm on the Julia set volume. I used the function mentioned in [5] and [6], which defines the distance to a fractal set as:

$$d = \lim_{n \to \infty} \frac{|z_n| \cdot \log|z_n|}{|z'_n|}$$

where

$$Z_{n+1} = Z_n^2 + C$$

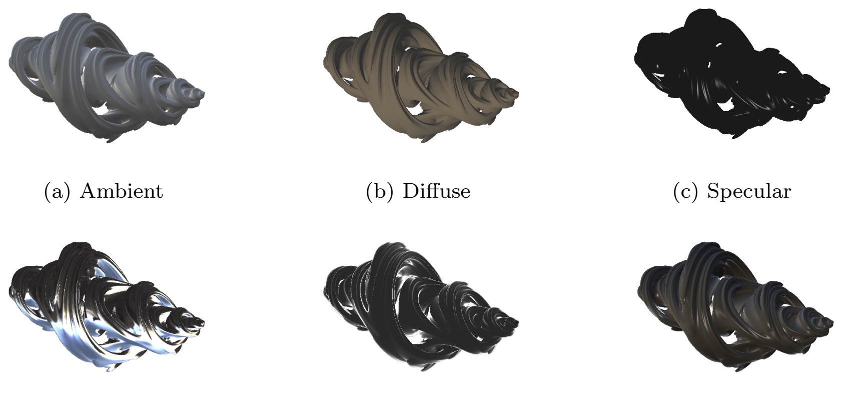
$$Z'_{n+1} = 2Z_n Z'_n + 1, Z'_0 = 1$$

Lighting

Ambient lighting is done use a pre-calculated irradiance stored, stored in a HDR cube environment map. The normal of the fragment is used to look into this HDR cubemap, and the result is added to a constant ambient color, multiplied by am ambient amount factor. This HDR irradiance cubemap data comes from [17], and irradiance is calculated and the format converted using a modified version of ATI®'s CubeMapGen software.

Diffuse lighting is done using Lambertian lighting, with the normals calculated per-pixel. Specular lighting uses ordinary Phong specular model, also with normals calculated per-pixel.

Reflection uses a separate reflection environment map, which is also an HDR cubemap. The view vector is reflected about the surface normal, and the reflected vector is used for looking up the cubemap. This reflection is multiplied by a Fresnel term, calculated as $1 - (\vec{V} \cdot \vec{N})^k$.



(a) Ambient (b) Diffuse (c) Specular

(d) Reflection (e) Fresnel term (f) Final composition

Figure 43: Lighting components of the Julia set scene

Scene: Human Head

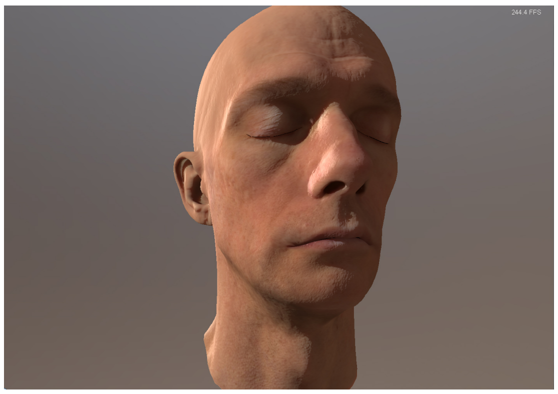
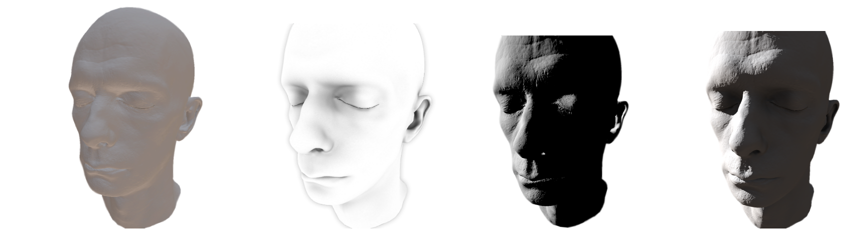


Figure 44: Human Head scene during development

This scene draws heavy inspiration from NVIDIA® Human Head Demo. Following the chapter explaining their techniques from [10], I have managed to develop an implementation of the NVIDIA® skin shader. The techniques are roughly divided into 4 components: diffuse irradiance, subsurface diffusion, and specular reflection, and final composition.

Diffuse Irradiance

The first step is to calculate the amount of diffuse surface irradiance at each point on the surface, disregarding any subsurface scattering. This component is divided into 3 steps: ambient, diffuse and shadow lighting.



(a) Ambient irradiance cubemap (b) Ambient occlusion (c) Diffuse (d) Final irradiance with shadow

Figure 45: Irradiance lighting steps of human head

For ambient lighting, a pre-calculated irradiance environment cubemap is used. The surface normal is used to look-up into this cubemap. This is then multiplied by an ambient occlusion term, which has been baked and stored in the alpha channel of the diffuse colour texture.

Diffuse lighting is then added, a simple $\vec{N} \cdot \vec{L}$ Lambertian diffuse lighting model. This diffuse lighting is multipled by a shadow factor, which is 0 if the fragment is shadowed and 1 other wise. Shadowing is implemented using a high-resolution floating-point render target shadow map, rendered from the light's perspective.

Subsurface Diffusion

The main feature that gives human skin its appearance is due to a mechanism of light referred to as subsurface scattering. Light penetrates the translucent surface of human skin, and is scattered around beneath the skin before exiting at some point. Subsurface scattering, much like how normal lighting can be modelled by separation into ambient, diffuse and specular components, can also be modelled by dividing it into directional scattering and diffuse scattering components.

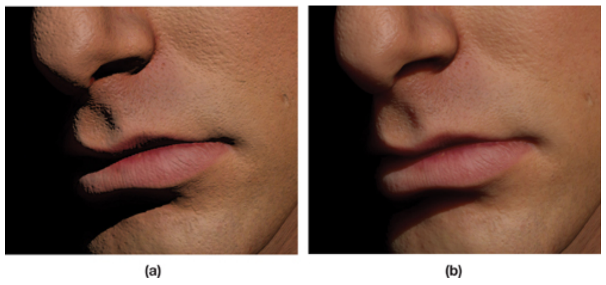


Figure 46: Subsurface scattering is crucial to creating a realistic appearance of skin. Image Source: [10]

Directional scattering referrs to single-scattering, and while modelling this is crucial to creating a realistic appearance of some materials such as marble, jade and smoke, for the human skin material it is general acceptable to omit modelling this [10].

Thus, that leaves us with diffuse scattering, or subsurface diffusion. This component is modelled by convolving the irradiance at each point of the surface by a diffusion profile [10]. However, properly convolving the diffusion profile requires a non-separable 3-dimensional kernel to be convolved over every point on the surface, and this cannot be done at the time of writing efficiently.

(a) Irradiance Map

(b) Blur 1

(c) Blur 2

(d) Blur 3

(e) Blur 4

(f) Blur 5

(g) Blur 6

Figure 47: Texture space Gaussian diffused irradiance textures

In order to convolve the diffusion profile efficiently, the texture-space diffusion [10] technique is modified and extended. diffuse irradiance is rendered in texture space into an off-screen irradiance map texture, using the texture UV co-ordinates of the vertex as the vertex's position in the vertex shader. The RGB diffusion profile curves are then curve-fitted and approximated using a sum of 6 gaussian curve functions using optimization tools such Mathematica or MATLAB. Then, the irradiance map texture is then blurred (separately in the U and V direction, since gaussian diffusion is separable) according to the curve-fitted parameters, and then summed and weighted according to the curve-fitted weighting parameters.

To correct for inconsistent texture-stretching across polygons, a UV stretch map is used to modify the radius of the gaussian blur according to the texture stretching.

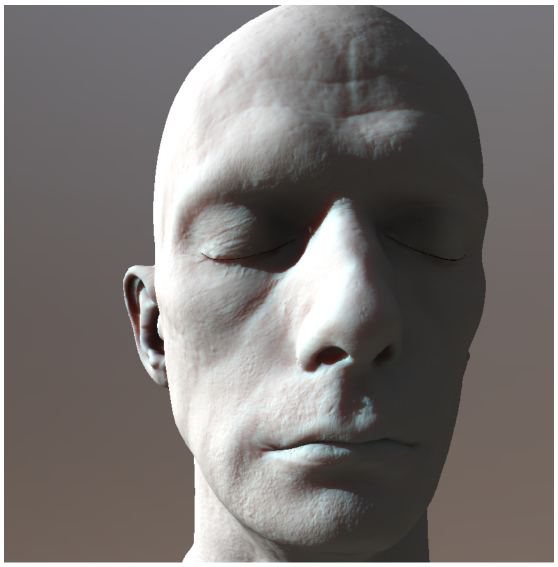


Figure 48: Result of subsurface diffusion

Specular Reflection

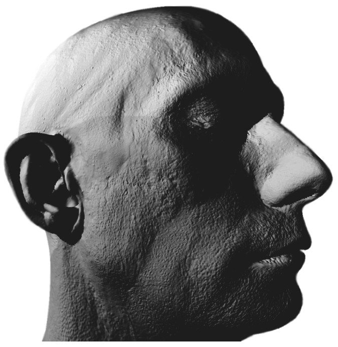


Figure 49: Specular lighting of skin

While the commonly used Phong specular model is sufficient for many materials, a realistic skin appearance will require a more complicated physically-based specular reflection BDRF model. The specular reflection can be commonly represented with a specular BDRF function as:

$$L_{\text{spec}} = C_s \cdot S \cdot \rho_s \cdot \text{specBDRF}(\vec{N}, \vec{V}, \vec{L_s}, \eta, m) \cdot \max(\vec{N} \cdot \vec{L}, 0)$$

where C_s is the specular colour, S is the shadow factor, ρ_s is a specular scaling factor, specBDRF is the specular BDRF function, $\vec{N}$, $\vec{V}$, $\vec{L_s}$ are the normal, view and light vectors respectively, and m is a roughness factor.

For a realistic model of human skin specular reflection, the Kelemen/Szirmay-Kalos specular BDRF model [10] is calculated:

$$\text{specBDRF}(\vec{N}, \vec{V}, \vec{L_s}, \eta, m) = max\left(\frac{\text{PH}_{\text{Beckmann}}(\vec{N} \cdot \vec{H}, m) \cdot F(\vec{H}, \vec{V}, F_0)}{(\vec{L} + \vec{V}) \cdot (\vec{L} + \vec{V})}, 0\right)$$

Where $\text{PH}_{\text{Beckmann}}(\vec{N} \cdot \vec{H}, m)$ is the Beckmann distribution function, pre-calculated into a texture using the function [10]:

$$\text{PH}_{\text{Beckmann}}(\vec{N}, \vec{H}, m) = \frac{1}{m^2 \cdot \text{NdotH}^4 \cdot e^{-\frac{(\tan(\cos-1(\vec{N} \cdot \vec{H})))^2}{m^2}}}$$

And $F(\vec{H}, \vec{V}, F_0)$ is a Fresnel term function, also pre-computed into a texture, given by:

$$F(\vec{H}, \vec{V}, F_0) = (1 - \vec{V} \cdot \vec{H})^5 + F_0(1 - (1 - \vec{V} \cdot \vec{H})^5)$$

The parameters m and ρ_s are varied over the mesh by a specular parameter texture, and $F_0 = 0.0028$ for skin.

Final composition

The summing the gaussian blurred irradiance maps according to the parameters of the diffusion profile, the specular calculation, and diffuse colour mixing are all done by the pixel shader program on the final composition pass. There are two ways diffuse colour could be mixed: post-mixing or pre-mixing.

Post-mixing means that the irradiance map is calculated off a purely white skin material, and the resulting scattering is multiplied by the diffuse color. This has the advantage of retaining the full default of the 4098x4098 resolution diffuse texture map (creating and blurring irradiance maps this large would have too large of an impact on performance). However, this can lead to inaccurate colouring, due to the fact that colour of the skin is not taken into account when calculating subsurface diffusion.

Pre-mixing, on the other hand, multiplies the irradiance map itself by the diffuse colour texture map. This has the advantage of much more accurate and realistic looking skin scattering, while having the disadvantage of losign the high resolution detail of the original diffuse texture map.

I used the approach recommended in [10]: do both. However, when both post-mixing and pre-mixing are used, the diffuse texture map could be multiplied twice, therefore the colour of the mesh would be multiplied by the diffuse map squared. So, we come up with a new term $0 < s < 1$ which controls the "mix" between pre and post scattering; the irradiance map pre-mixing is multipled by diffuse^s while the post-mixing is multiplied by diffuse^{1-s}. For the parameter s, I used the NVIDIA® recommended value, 0.5.

Scene: City

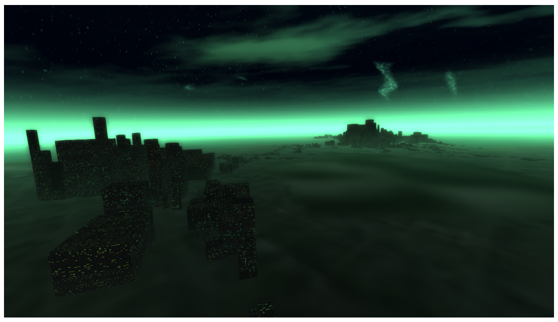


Figure 50: City scene

This scene features reflective water, and height based fog. The city model was generated using a procedural city generator plugin for Blender, and exported as a model. The height based fog is calculatd using analytical integration, more details about this in the Evelated scene section. The land plane features reflection mapped water reflections with surface normals perturbed using animated sine and cosine waves. The reflections are done using an environment cubemap.

Scene: Lucy

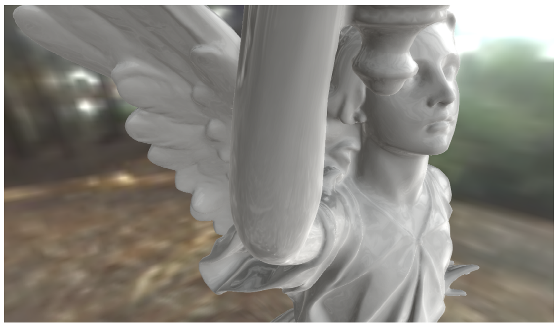


Figure 51: Lucy scene

A simple scene featuring fresnel cubemapped reflections and cubemapped irradiance maps. The model comes from the Stanford model repository [18].

Scene: Car

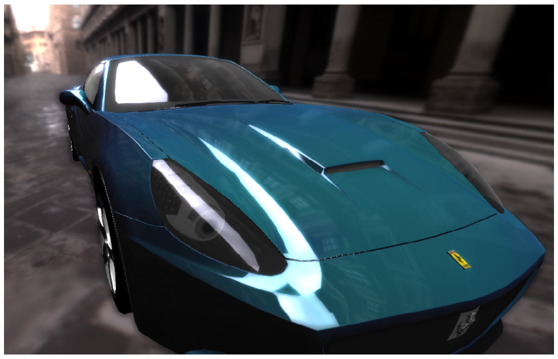


Figure 52: Car scene

This scene features a texture-based BDRF function shader, HDR irradiance maps, HDR fresnel reflections, and material-based model rendering. The diffuse BRDF function is stored in a texture lookup table based on light and view angles. The initial data for this BDRF came from Ford Motor Company, who directly measured their car paints in their lab, and then this was hand-enhanced by shader artists at NVIDIA. The car tyres and interiors are bump mapped to enhance detail.

Scene: Sphere

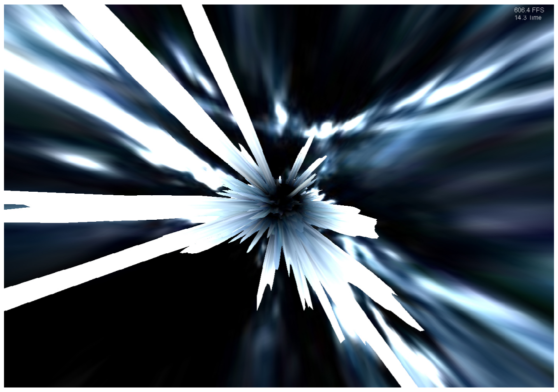


Figure 53: Sphere scene

The sphere scene features GPU texture-based vertex displacement, and music spectrum synchronization. The vertex displacement is done in the vertex shader program by using the vertex shader 3.0 texture lookup feature. The vertices are displaced along the normal according to the brightness level of the cube mapped environment texture. The audio synching is done using FMOD's in-built FFT module, which is then binned into low, mid and high bands, for easier analysis.

Post Processing: High Dynamic Range

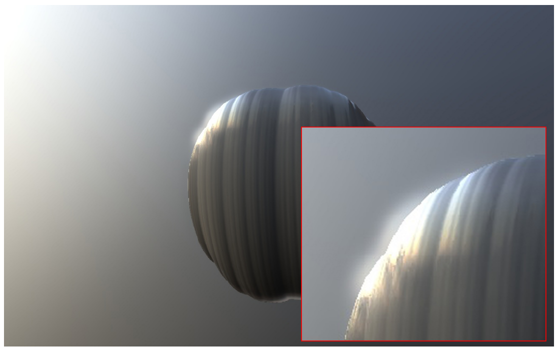


Figure 54: HDR Bloom

The entire demo is rendered in full 16-bit floating point high dynamic range (HDR). Bloom is then calculated on the HDR image, by additively blending a thresholded blurred screen image on top of the original. The bloom effect in HDR has a much more realistic appearance compared to bloom in low dynamic range, because HDR is able to represent the same colour in varying levels of intensity, and these varying levels of intensity can be used to compute the amount of bleeding in the bloom effect. For example, LDR is unable to distinguish the sun from white brick, as both are white. HDR, however, will not propagate or bloom the white brick, as it has a low intensity level, while HDR will give the bright sun a strong bleeding bloom effect. The bloom is calculated using a 3-pass 7-sample gaussian blur, also in HDR.

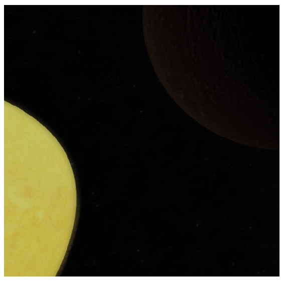


Figure 55: The yellow colour and texture of the sun during fade to black.

Additionally, rendering in HDR will also give much more realistic fade effects;

as evident in the space scenes, during the fade-out of the scene to black, yellow colour and texture of the sun can be seen, rather than the usual white.

Post Processing: Colour Adjustment

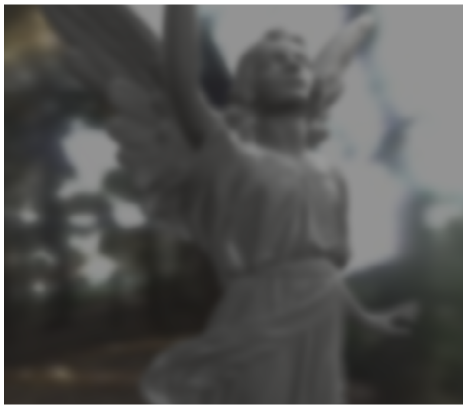


Figure 56: Saturation, brightness, blur controls

The various effects seen in the demo including fade to/from black, to/from white, blur, saturation, brightness controls are done in the colour adjustment post processing shader. The blur uses the same 3-pass gaussian blurred image from the HDR bloom. Saturation, fade, contrast...etc are calculated per-pixel in the pixel shader program.

Post Processing: Chromatic dispersion & Film Grain

The demo also supports post-process chromatic dispersion and film grain. Chromatic dispersion is implemented using 3 noise-displaced texture lookups, for R, G and B. Film grain is implemented using a noise texture. These effects, like colour adjustment, are faded in/out as needed during the demo.

Post Processing: Morphological Anti-aliasing

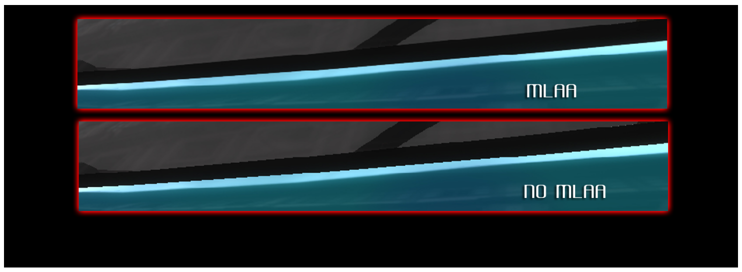


Figure 57: MLAA comparison

Morphological anti-aliasing is a recent (SIGGRAPH'11)[19] image-based anti-aliasing approach. Traditional methods of image-based anti-aliasing approaches use a cross bilateral blur filter with an edge detection algorithm, so the detected edges are blurred. MLAA is able to produce comparible results with traditional multi-sampled anti-aliasing (MSAA), producing about the same results as 4xMSAA while being faster.

With the recent introduction and increasing in popularity of image-space lighting technqiues such as screen space ambient occlusion (SSAO), deferred lighting, and various screen space global illumination techniques, aliasing has become a big problem as these techniques are incompatible with traditional hardware MSAA. Thus, image-based post-processing anti-aliasing approaches such as MLAA has seen an explosion in popularity, with implementations in recent (as of writing) game titles such as Deus Ex Human Revolution® and God of War 3®, and AMD Catalyst™ drivers version 11.2 implementing MLAA for the ATI Radeon HD 5000 series and above[20].

MLAA is divided into 4 steps: discontinuity(edge) detection, distance propagation, blend weights calculation and final blurring.

MLAA Discontinuity Detection

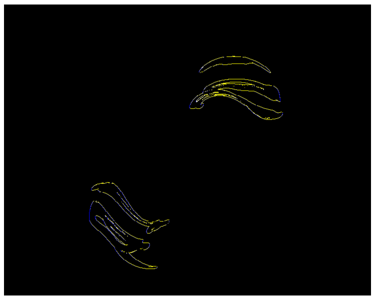


Figure 58: Discontinuity Detection of Julia Set Scene

The aliased edges are found by taking right and down samples for every pixel, and seeing if there is a discontinuity in the vertical / horizontal direction at this pixel respectively. Discontinuities are detected by thresholding the distance of the two pixels in *Lab* colour space; distance above the threshold means there is a discontinuity here. From this information, the distance table is initalised; r and g components of the table texture are set pixel value 1 if there is a horizontal discontinuity detected, and b and a set to 1 if there is a vertical discontinuity.

MLAA Distance Propagation

Line distance is propagated from the extremities. The distance is propagated using a recursive doubling approach, with 4 samples taken at each pixel for 4 separate passes, given a maximum distance of 255 pixels. The distance propagation happens separately for left/right/up/down, and adding the left distance of every pixel to the right distance gives the total length of the horizontal line, similarly for vertical lines. For details on the recursive doubling algorithm, refer to [19].

MLAA Blend Weights Calculation

After the distance has been propagated, the correct L-shape cases must be detected, and from that, the blending weights must be calculated. The orientation and shape of the L-shape is determined by taking samples at corresponding up/left pixels, and then taking further samples at the extremeties of the relevant horizontal/vertical lines. For further details, refer to [19].

Figure 59: Trapezium table texture : $A = \frac{1}{2}\left(1 - \frac{2p+1}{L}\right)$

Once the correct L shape is detected, the blend weights, which depend on the area of the subpixel triangle/trapezium, is determined by using a texture pre-calculated look-up table. The blending weights, output of this stage, are stored in a blend weights texture, with r, g, b, a of the texture storing blending factor in the up, down, left and right directions respectively.

MLAA Blurring

The last step of the algorithm is to blur the final scene according to the blend weights texture. This is done using a simple 4-sample blur in each direction up, down left and right.

MLAA Performance

These tests were done on at 1440x1080 resolution:

Scene	MLAA (FPS)	no MLAA (FPS)	Cost (ms)
City	250	375	1.333
Lucy	210	240	0.680
Ion	272	400	1.176
Terrain	95	107	1.180
Julia	83	90	1.084
Head	132	145	0.679
Car	243	340	1.174

The Lucy and Head scenes were very fast because these scenes are simple, single-object scenes and therefore do not contain a lot of lines; Early exit branches within the shader code therefore comes into effect, and much of the scene is not calculated.

Technical: Models

3D models are loaded using the Open Asset Import Library (Assimp). Written in C++, designed for use with OpenGL, and released under the BSD license, Assimp is an independant popular portable model import and export library that features support for a wide range of formats.

In order to render the model from Assimp, however, proved to be a slightly non-trivial task. The scene graph has to be manually traversed using a recursive depth-first search function, which recursively pushes the new node's translation matrix onto the matrix stack. In OpenGL, this is quite trivially implemented, as openGL handles the building and handling of vertex buffers in the background and the programmer only has to send in the vertex co-ordinates every frame or compile a buffer. However, DirectX requires that polygons be sent to the library in static vertex buffers, "batches" of triangles.

There are several ways to implement this traversing of scene graph while only restricted to static vertex buffers. It is possible to create and lock a vertex buffer every frame, but this will make model rendering very very slow. It is also possible to "flatten" the scene graph manually, calculating on the CPU all the vertices of every node of the scene graph by multiplying the vertex positions by their transform matices. However, this approach is not compatible with any future implementations of bone animation. The third and best approach to this problem is to build a separate vertex buffer for every node of the scene graph, as each node is just a collection of triangles and is static.

This is implemented in my engine using a hash table. The a scene path string which stores the current path from the scene root node of the depth-first traversal through the scene graph is kept, and when a node is drawn, the scene path string is hashed into a hash table which stores a pointer to a DirectX vertex buffer as its value. The first time any node is rendered, the hash table look-up will fail, and a new vertex buffer will be created and stored in the hash table. The next time the same node is rendered, the hash table will find the pre-built vertex buffer and render that straight away.

For comparison, loading an MD5 model from the game Doom 3 without using the hash table gives about 30 FPS on an intel® i5 760 and NVIDIA® GeForce GTX570. After the vertex buffer hash table optimisation, the same scene with the same Doom 3 model renders at 1500 FPS.

Technical: Camera and Animation

Camera movement is done with Catmull-Rom splines. These splines are especially suitable for camera point interpolation for their simplicity, easiness to compute, and guaranteeing to pass through each keyframe, and also guaranteeing continuous tangents. 3 seperate splines are kept for each camera movement path, one for the camera position, one for the target position, and one for the up vector.

Time-based animation is analytically integrated, and is calculated as a function of time. The demo is designed so that one can set the global ”time” value to any point in the demo, and it will automatically resume the demo from that point.

Specifications

- Project Name: Ambiance: Real-time Graphics Demo
- Version: 1.0
- Recommended System Requirements:
  - Intel Core 2 Duo or AMD Athlon X2 at 2Ghz x86/x64 CPU or higher
  - 2GB+ System RAM
  - Microsoft Windows XP or newer, 32-bit or 64-bit.
  - DirectX 9.0c Hardware T&L with Pixel Shader 3.0 support
  - DirectX 9.0c compatible sound card
  - NVIDIA® GeForce 9600GT 512MB or better
  - 350MB Hard disk space

Results

Performance results:

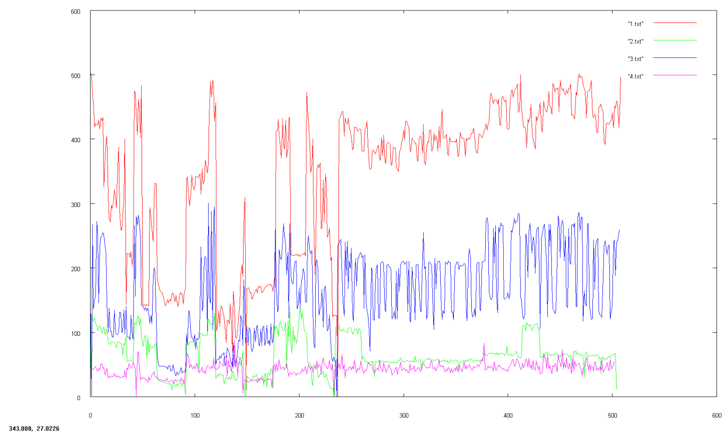


Figure 60: Performance results

System	Average FPS
1	322.935504
2	60.728505
3	152.324419
4	39.847584

System 1 had Intel i5 760 Quad core 2.8Ghz, NVIDIA GeForce GTX570 1GB, 4GB RAM, Windows 7 64-bit

System 2 had Intel Core Duo 2.4Ghz, NVIDIA GeForce 9600GT 512MB, 2GB RAM, Windows 7 32-bit

System 3 had Intel i3 530 Duo core 2.93Ghz, NVIDIA GeForce GTX260SO 1GB, 4GB RAM, Windows 7 64-bit

System 4 had Intel i7 Q720 1.6Ghz, ATI Mobility Radeon HD 5730 1 GB, 4GB RAM, , Windows 7 64-bit

Conclusion

The outcomes were successful; A deeper understanding and experience were gained from working with DirectX and HLSL shaders; a full understanding was gained during the development of the project of how the rendering pipeline works, shader optimisations, shader branching optimisations, various computer graphics techniques, screen space techniques, GPU raytracing...etc.

The final product met my original expectations and requirements. An effective advanced graphics demonstration program was developed, suitable for both artistic display and system benchmarking.

Future Expansions

One possible direction of future expansion would be to implement more deferred lighting techniques; SSAO, SSDO, SSGI, irradiance volumes ...etc. The demo code could be re-factored for DirectX 11, and making use of the tesselation feature of DirectX 11, or even ported to OpenGL & GLSL for more cross-platform compatibility. Another possible direction would be to use the engine framework built to create more complicated multimedia software, such as games.

References

[1] inigo quilez, 2002-2007. *Terrain Raymarching* [online] Available at `http://www.inf.ufrgs.br/~oliveira/pubs_files/inpainting.pdf` [Accessed 15/11/2011]

[2] Sean O'Neil, 2005. *Accurate Atmospheric Scattering* In: M. Pharr, R. Fernando, ed. 2005. *GPU Gems 2* Pearson Education/NVIDIA. Ch.16. Available at `http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter16.html`

[3] inigo quilez, 2001-2007. *3D Julia Sets* [online] Available at `http://www.iquilezles.org/www/articles/juliasets3d/juliasets3d.htm` [Accessed 15/11/2011]

[4] Keenan Crane, 2004-2005 *Ray Tracing Quaternion Julia Sets on the GPU* [online] Available at `http://www.cs.caltech.edu/~keenan/project_qjulia.html` [Accessed 15/11/2011]

[5] inigo quilez, 2004 *Distance Rendering For Fractals* [online] Available at `http://www.iquilezles.org/www/articles/distancefractals/distancefractals.htm` [Accessed 15/11/2011]

[6] John C. Hart , Daniel J. S , Louis H. Kauffman T *Ray tracing deterministic 3-D fractals*, SIGGRAPH '89

[7] inigo quilez, 2008. *Advanced Perlin Noise* [online] Available at `http://www.iquilezles.org/www/articles/morenoise/morenoise.htm` [Accessed 15/11/2011]

[8] inigo quilez, 2008. *Behind Elevated* [online] Available at `http://www.iquilezles.org/www/material/function2009/function2009.htm` [Accessed 15/11/2011]

[9] inigo quilez, 2010. *Better Fog* [online] Available at `http://www.iquilezles.org/www/articles/fog/fog.htm` [Accessed 15/11/2011]

[10] Eugene d'Eon, David Luebke, 2007. *Advanced Techniques for Realistic Real-Time Skin Rendering* In: H. Nguyen, ed. 2007. *GPU Gems 3* Pearson Education/NVIDIA. Ch.14. Available at `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch14.html`

[11] James Hastings-Trew, 2007. *JHT's Planetary Pixel Emporium* [online] Available at `http://planetpixelemporium.com` [Accessed 15/11/2011]

[12] Simon Green, 2004. *Real-Time Approximations to Subsurface Scattering* In: *GPU Gems* Pearson Education/NVIDIA. Ch.16. Available at `http://http.developer.nvidia.com/GPUGems/gpugems_ch16.html`

[13] Scott Warren , 2007. *Recovering detailed normals from photo textures.* [online] Available at `http://www.cgtextures.com/content.php?action=tutorial&name=normalmap` [Accessed 15/11/2011]

[14] A. J. Preetham, Peter Shirley, Brian Smits, *A Practical Analytic Model for Daylight*, SIGGRAPH '99 Proceedings of the 26th annual conference on Computer graphics and interactive techniques

[15] greeneggs, gamedev.net , 2002. [online] Available at `http://www.gamedev.net/topic/100346-having-trouble-implementing-a-sky-colouring-algorithm/` [Accessed 15/11/2011]

[16] Ilya Zaytsev, 2010. *Beneath a Preetham's sky* [online] Available at `http://r2vb.com/index.php?id=9` [Accessed 15/11/2011]

[17] Paul Debevec, 2004. *Light Probe Image Gallery* [online] Available at `http://ict.debevec.org/~debevec/Probes/` [Accessed 15/11/2011]

[18] Various authors, 2011. *The Stanford 3D Scanning Repository* [online] Available at `http://graphics.stanford.edu/data/3Dscanrep/` [Accessed 15/11/2011]

[19] Venceslas Biri, Adrien Herubel, Stephan Deverly *Morphological Antialiasing And Topological Reconstruction*, SIGGRAPH '11

[20] AMD/ATI, 2011. *Morphological Anti-Aliasing* [online] Available at `http://sites.amd.com/us/game/technology/Pages/morphological-aa.aspx` [Accessed 15/11/2011]