

Parallax Occlusion Mapping Final Report

by Gary Keen

FILE	17994_GARY_KEEN_PARALLAX_OCCLUSION_MAPPING_FINAL_REPORT_78196_1919460173.DOCX (2.71M)		
TIME SUBMITTED	21-APR-2016 01:11PM	WORD COUNT	12531
SUBMISSION ID	56197452	CHARACTER COUNT	67178



PRIFYSGOL CYMRU

Y Drindod Dewi Sant

UNIVERSITY OF WALES

Trinity Saint David

SWANSEA - ABERTAWE

**Parallax Occlusion Mapping: A development of the
technique and a study into its feasibility in real-time
applications.**

By

Gary Keen

Supervisor: Andrew Baker

Second Marker: Gordon Dickens

Project submitted as part of the requirements for the award of
BSc(Hons) Computer Game Development

DECLARATION

I, Gary Keen, declare that I am the sole author of this Project; that all references cited have been consulted; that I have conducted all work of which this is a record, and that the finished work lies within the prescribed word limits.

This work has not previously been accepted as part of any other degree submission.

Signed :

Date :

FORM OF CONSENT

I, Gary Keen, hereby consent that my Project, submitted in candidature for the B.Sc.(Hons) Computer Games Development degree, if successful, may be made available for inter-library loan or photocopying (subject to the law of copyright), and that the title and abstract may be made available to outside organisations.

Signed :

Date :

Table of Contents

Abstract	5
Introduction	6
Literature Review	7
Mapping Techniques	7
Bump Mapping	7
Normal Mapping	9
Displacement Mapping	11
Parallax Mapping	13
Relief Mapping	16
Horizon Mapping	17
Rendering Techniques	18
Priority Fill	18
Z-Buffer	19
Deferred Rendering	20
Conclusion	21
Research Methodology	21
Development Methodology	22
Design	24
Software Used	24
Graphics API	24
IDE	25
Windowing and Controls	25
Hardware Requirements	26
Implementation	26
Results	29
FPS Test	30
Acceptance Testing	30
Evaluation	31
Conclusive notes	32
Reflection	34
Future Work	35
Glossary	37
References	38
Bibliography	40
Appendices	41

Abstract

The aim of this development study is to implement and demonstrate an interpretation of the parallax occlusion mapping technique as described by Natalya Tatarchuk to simulate geometry on a flat surface suitable for real time environments. This aim is achieved by incrementally implementing the improvements that eventually came to the desired results and running speed tests to compare and contrast how feasible it is in a real time environment against other common techniques for simulating surface detail.

The parallax occlusion mapping method displaces texture coordinates using a height map based on the view vector to simulate motion parallax as the camera or the object moves around in 3D space. This allows the technique to appear as though it has depth at any angles when it in fact remains a flat surface. Furthermore Blinn-Phong lighting is applied to the effect to show how the technique generates its own occlusions for a realistic result under any lighting conditions.

When testing this shader it was found that against other shaders that simulate surface detail more commonly used in real-time applications such as games the effect was more computationally expensive, running only at half the frames per second as a normal mapping technique. However far more surface detail is simulated in parallax occlusion mapping and does so at around nine times faster than a high polygon mesh with the same surface detail. Nevertheless at 452 frames per second average the technique is still suitable for use in real time applications.

Introduction

When creating real time applications such as games, demonstrations or virtual reality experiences retaining an element of realism in the virtual world is a key factor for immersion, however realism tends to be computationally expensive and slow in performance. This development study aims to understand the feasibility of parallax occlusion mapping, a shader that approximates real surface geometry on flat surfaces by using mapping techniques and how it performs in a real time

environment. This aim is achieved by designing the program and incrementally building each step of the improvements to bump mapping that makes up the final result. To ensure it performs sufficiently in real time an objective is set to test the shader on its frames per second, compared with other common techniques to simulating surface detail on a flat face.

To get a better understanding of parallax occlusion mapping and how it functions research into the literature is done for various methods and improvements of the parallax mapping along with similar shading techniques such as bump mapping, normal mapping, horizon mapping, relief mapping and displacement mapping.

The research and development methodology is explained to lay out the process undertaken in this study to obtain data and build the demonstration. These choices are also contrasted with alternative methods to further explain why the chosen methods are used.

This study will then attempt to create its own parallax occlusion mapping technique based on the existing literature and run performance tests to measure its impact on performance against other commonly used techniques.

After testing the results of the development stage the work is evaluated and concluded. Ideas for further work and improvements are outlined before reflecting on what went wrong and what could have been done better.

[Literature Review](#)

[Mapping Techniques](#)

Bump Mapping

Bump mapping is a very common technique to simulate fine detail on an object without creating additional geometry. This is done by altering the surface normal of an object using a greyscale height map to define levels of intensity and using the altered normal when lighting calculations are applied,

thereby displacing the light to simulate bumps and wrinkles on a flat surface. The technique was first created by James F. Blinn[1] in 1978 as an improvement to an idea put forward by Edwin Catmull in 1973 in which one would use “parameter values of parametrically defined surfaces (in a) texture definition function which scales the intensity of the reflected light”[1]. Blinn acknowledged that this method was good for retaining certain texture detail on an irregular surface, however was not a convincing method for simulating fine detail such as bumps or wrinkles.

In the conclusive notes of his paper, Blinn describes how the initial method of bump mapping was four times as slow as Phong shading and twice as slow as colour texture mapping, with each of his rendered images taking three to seven minutes to produce. Whilst hardware limitation was a factor at the time this shows that the technique does understandably come with a performance hit, though against other core graphical requirements such as the aforementioned Phong shading used to produce smooth lighting and coloured texture mapping the shader itself is not overly computationally expensive.

In 1994, Nelson Max[2] and Barry Becker used the original technique as a displacement function for 3D volume textures to add to the surface position. Examples of the typical use of volumetric textures are particle effects such as smoke, light rays and fog but by utilising bump mapping into the W coordinates of the UV map the information for bump mapping was already existent and read in the pixel shader. Whilst the technique did not take off well with developers the technique removed the need for a separate bump map to be read into the pixel shader as the information is already embedded into the textures.

Mark Kilgard[3] of NVIDIA modernized the original technique hugely in 2000 to work much more efficiently in real-time applications with more accuracy for modern graphics processing units. The improved technique also takes contributions from the diffuse, specular and ambient lighting maps to achieve more accurate bumps and self-shadows based on the geometric normals to give a truer sense of where the light source originates, supporting all types of light sources. Kilgard mentions the

technique is extremely robust though at the time required three rendering passes, however this was reduced by improved hardware and so the technique has since been appropriate for modern real time applications. These rendering passes each handle a different map, the greyscale bump map, the actual texture and the specular. As the specular map is considered optional the method could, in certain situations, be reduced to just two passes. The technique made use of modern graphics processing unit features such as cube mapping for per pixel normalisation and register combiners, an NVIDIA proprietary technology used for computing fragment colours, to calculate the per pixel dot products and other additional mathematical problems in the algorithm. These problems include using tangent space to calculate accurate self-shadowing, making use of the ambient contribution and successive squaring for the specular contribution.

In 2001 Mark Peercy[4] also improved on the original bump mapping algorithm that transforms the lighting calculations to tangent space to avoid potential hardware limitations with rasterization beyond the ability to do Phong shading with the general consensus that Phong shading would likely become a hardware standard due to its large range of application. His technique firstly precomputed a texture of the modified normal in tangent space then transforms all the shading vectors into tangent space for each vertex. Peercy mentions it is done per vertex and not per pixel as for small perturbations per vertex is a sufficient enough for a good approximation of the modified surface. A secondary reason for the choice is that it eliminates the need to rely on per pixel calculations done by the hardware and instead uses per vertex calculations done in a general geometry processor. After the shading vectors are transformed to tangent space they are interpolated and renormalised based on the surface tangent and the texture is normalized. Finally the self-shadowing is applied again on the vertex level.

It is noted that while moving the problem to the vertices increases efficiency of the technique it is only suitable for low polygon models as an increased amount of vertices means more work is required by the algorithm. In conclusion the method was successful in that it is entirely independent

of hardware and their possible limitations though the technique becomes less efficient as more polygons are present. This method could therefore be seen as an extension to Phong shading.

Bump mapping is, in essence, the first mapping technique to effect the final output image specifically by displacing light. This not only gives it purpose in the research for historical reasons but acts as a starting point for the development of a higher level shader. Many iterations and improvements have been made since its conceptualisation to make it more effective in real-time situations and increase the final outputs clarity and so it is a lightweight and effective starting point.

Normal Mapping

Over the years bump mapping has been improved in a wide variety of ways to make the method more suitable for use in real time however the most notable of such is the introduction of the normal mapping technique. Similar to bump mapping, normal mapping, also known as Dot3 bump mapping, uses a specific texture map to alter the surface normals of a low polygon model during light calculations however instead of using a greyscale height map to indicate the level of intensity a normal map is used instead. This differs from the traditional greyscale height map as a normal map uses RGB colours to define X, Y and Z coordinates respectively and are generated from a high polygon model directly. This then leads to far more accurate and complex approximations of light displacement as the result is based on a high detail object that has already been created. This idea of using geometric details from higher polygon models was first introduced in 1996 by Venkat Krishnamurthy and Marc Levoy[5] in which they scanned an action figure 75 times to generate a complex 3D model with around 350,000 polygons as an input for their custom pipeline. Throughout the pipeline this model has boundary curves painted on and is outputted as a set of bounded mesh regions called polygonal patches. These patches are then put through an automatic resampler to create a regular grid known as spring meshes. Finally these meshes have surfaces fit to them and output as a set of B-Spline (basis spline) models and displacement maps stored as RGB value images – normal maps. This method was efficient for its time however use of normal mapping to simulate

geometry in real time applications were not available until much later, with its use in the video games industry only being in 2003 with the introduction of programmable shaders.

This paper led to a number of different research specifically focusing on using the generated normal maps to transfer detail from high polygon models to lower polygon models for use in (Dot3) bump mapping and displacement mapping though in 1998 two papers of interest were presented. The first of which by Jonathan Cohen et al.[6] aimed to store surface data within the texture itself, as opposed to displacements, with surface colours being stored within the texture map and surface normals being stored within the normal map. The 3D model goes through a simplification operation and because the attributes such as colours and normals are now separated from the surface position a new texture deviation metric is used which bounds the difference of a mapped attribute value's position from its adjusted position on the original surface. This means the attributes of the high polygon model are mapped correctly in screen space. Conclusively it worked well, using the same action figure model as in Krishnamurthy and Levoy's paper the level of detail was retained from almost 250,000 triangles to merely 975 triangles, however ultimately the method was constrained as the attributes were dependent on the low polygon model being generated from the same specific simplification algorithm meaning the technique could not be applied to a simplified 3D model if a different algorithm was used – a high likely case.

A similar paper released in 1998 by Paolo Cignoni et al.[7] used a simpler method that also stores detail attributes in texture maps for efficiency however unlike the previous paper generalisation was a big factor in this method and for that the attributes are not dependent on a specific simplification process therefore there are no assumptions on how geometry simplification is done and any method becomes compatible. For this benefit the idea of this method is still commonly used.

Normal mapping is a major improvement to the original bump mapping technique. It utilises red, green and blue channels to deliver more accurate representations of fine detail on a mesh (16.8 colours vs 256 colours) and assigns each channel to an axis, X, Y and Z. Because of this, a normal map

can be generated from a high resolution mesh to be used on a more hardware friendly low resolution mesh to produce near identical results. This will be a useful technique during development as it can be used in combination with parallax mapping to tackle both emulated depth and fine detail.

Displacement Mapping

Displacement mapping uses a combination of the vertex shader and the tessellation shader to generate geometry based on data from a height map or normal map. While more efficient than using high polygon models it differs from bump/normal and parallax mapping as it creates physical geometry from the texture map as opposed to simulating an approximation of geometry via texture coordinate offsetting or light displacement. The technique is new to real time applications in comparison to other shading techniques as tessellation shaders weren't implemented until 2008 (with DirectX11) and 2010 (with OpenGL 4.0) however high end rendering software such as RenderMan has been using displacement mapping for a lot longer as time critical rendering was not as much of a concern, as early as the mid-1980's. The reason for real time applications such as video games not having used displacement mapping earlier is because earlier implementations of the technique used adaptive tessellation to create polygons the size of a pixel on a screen, known as micropolygons, to create this geometry. This concept of micropolygons was introduced with the REYES algorithm and the idea of micropolygon displacement mapping via REYES was developed in RenderMan.

The first documented use of displacement mapping was within a 1984 paper released by Robert Cook[8] wherein he used his concept of shade trees, a pipeline concept that makes use of data trees to simplify complex shading instruction sets, to use displacement mapping to bevel a cube as well as perturbing the sides to create the appearance of two intersecting cubes. Cook mentions that displacement mapping was conceived as a solution to the lack of surface features that bump mapping couldn't achieve such as surface detail on an objects silhouette. This point was also

reiterated in GPU Gems 2[9]. This book also mentioned that while per-pixel displacement mapping still exists even in real time, it is done in the fragment shader as opposed to the vertex shader as whilst the vertex shader is more logical three main points are outlined: Firstly GPU's have more pixel processing power than they do vertex processing power as games commonly make use of shaders based off texture maps so the focus of GPU manufacturers has priority there. Secondly pixel shaders are better for accessing texture maps and thirdly pixel processing scales with distance "This means work is concentrated on nearby objects, where it is needed most." [9] The main reason though is that the vertex shader cannot produce more vertices, so this method could also use the geometry shader, where more vertices can be generated.

Lifeng Wang et. al[10] were among a few to improve on displacement mapping to help bring the technique to real time rendering in 2003 with better results not only computationally but also visually. They proposed a per-pixel displacement mapping technique named view-dependent displacement mapping that would model a surfaces displacements along the viewing direction instead of the mesh normal direction. This alternative method allowed for efficient implementation of self-shadowing, occlusions and silhouettes without needing to increase the complexity of the underlying mesh. The traditional displacement method could not support the first two of these effects and were only seen in horizon mapping, whereas horizon mapping could not achieve silhouettes as it did not physically change the geometry of the object however by using the pixel shader all these additional effects were made possible. Additionally by using the pixel shader anti-aliasing through mip-mapping becomes achievable. Conclusively Wang saw a big performance increase in real time applications due to per-pixel processing. As mentioned before there are many benefits outlined in GPU Gems 2[9] to using the pixel shader for displacement mapping and Wang's research is proof of those optimal settings.

Another improvement to displacement mapping came in 1999 by Stefan Gumhold and Tobias Hüttner[11] who used mipmapping techniques to not only optimise displacement mapping but also

to avoid aliasing artefacts caused by high resolution displacement viewed from a distance. Mipmapping is the act of downscaling a resolution over distance as an original amount of pixels/polygons will not be seen the further away from the camera it is, this is also known as level of detail. They achieved this by proposing a view dependent re-meshing technique in which triangles are subdivided based on their position in screen space, where more rough approximations of the displacement were used. The idea was that mipmapping was applied to the actual subdivision of the meshes triangles as opposed to the displacement map itself.

A similar research was also released in 2000 by Michael Doggett and Johannes Hirche[12] which provided multiple additions to the standard displacement mapping technique and pushing more to the graphics pipeline. Similarly to Gumhold and Hüttner's research it takes advantage of a view dependent adaptive re-meshing method to make use of mipmapping techniques to provide more optimal settings for real-time applications as well as removing aliasing caused by high detail models being viewed from distance however it also moves tessellation to the hardware as geometric primitives. This means that the tessellated triangles will not have to be passed into the graphics pipeline from the CPU.

Displacement mapping is the primary rival of the parallax mapping technique in this study. It is important to learn how it works efficiently in real-time situations as it essentially creates thousands more polygons, the opposite aim of parallax mapping. This technique appears to be used very commonly in modern games to increase scene complexity and the aim of this development study is to find out if parallax occlusion mapping is a viable, if not superior, alternative.

Parallax Mapping

Parallax mapping, also known as offset mapping, is an extension of bump/normal mapping that can generate virtual displacement without creating additional geometry by combining the vertex shader and the fragment shader. The technique was first put forward by Tomomichi Kaneko[13] in 2001 with the aim of adding the motion parallax effect to a texture mapped technique wherein objects

that are further away move slower than objects that are closer. Parallax mapping is achieved by storing complex surface detail to a height or normal map then offsetting texture coordinates in screen space to simulate additional geometry. This is done by ray tracing the vector between the actual geometry and the camera viewpoint and extending that vector into the geometry until the offset is found. Whilst Kaneko's conclusion provided a fast and efficient working solution there were problems to the method that prevented the mapping techniques widespread use as artefacts appeared at grazing (steep) angles and there was no self-occlusion, meaning the lighting was flat as it was simulated geometry on a flat polygon. However many of the initial problems have since been alleviated with the introduction of Steep Parallax Mapping and Parallax Occlusion mapping.

Before these improvements were introduced, however, another enhancement to the original technique was released in 2004 by Terry Welsh[14] which saw the addition of offset limiting in parallax mapping. In the paper Welsh mentions the reason for this addition is because when offsetting a texture coordinate whilst steep viewing angles will be a low to non-existent offset extremely shallow viewing angles may produce an erroneously large to infinite offset, resulting in incorrect pixels being displayed. By adding a pre-defined offset limit this problem is corrected at the trade-off of certain texture offsets not looking as extreme as they did beforehand.

Steep parallax mapping, conceived by Morgan and Max McGuire[15] in 2004, is very similar to parallax occlusion mapping where the primary difference is that steep parallax mapping uses traditional bump maps for efficiency whereas parallax occlusion mapping can use either or both for accuracy but both come with a range of improvements to the initial parallax mapping technique, most notably self-shadowing and the use of multiple sample points. The use of multiple samples is necessary for real-time applications of parallax mapping such as games as without it, using a binary search, shallow viewing angles will take the illusion of depth away leaving only a flat polygon as only one sample has been checked. By stepping through the volume multiple times before the vector hits the intersection more accurate depth can be shown at all angles. An issue with multiple sampling

however is that the more samples you choose to use, the more computationally expensive the technique is going to become, though too few samples and one will be able to see 'layering' artefacts so a balance between how deep the parallax effect is going to be and the number of sample points to alleviate aliasing needs to be found as described in Natalya Tatarchuk's SIGGRAPH06 presentation[16].

In 2006 Natalya Tatarchuk[16], [17] of ATI (AMD) presented the Toyshop Demo [Appendix 1], a two minute tech demo showing off multiple technologies such as various rain rendering techniques and parallax occlusion mapping all with the challenge of being fit into 256MB of dedicated video memory. Whilst the same as parallax mapping at its core, parallax occlusion mapping offers various improvements such as self-occlusion, self-shadowing, adaptive Level-Of-Detail (LOD) and offset limiting[14], similarly to steep parallax mapping. Even with all these additional improvements aliasing can still occur at certain grazing angles, but Tatarchuk shows in multiple examples[16] that most artefacts can be fixed or hidden using various techniques such as depth bias, an increased amount of samples and soft shadowing. With all of this combined an efficient, high fidelity approximation of raised surfaces is created that can also be combined with additional mapping techniques, such as adding fine detail from normal mapping. Parallax occlusion mapping makes use of reverse height maps as the technique extrudes the new offset coordinates into the polygon, making the highest point in the map the lowest point on the offset. This is done primarily as the original polygon is always considered the highest point of the displacement. Reverse height map tracing and the first mention of parallax occlusion mapping was the subject of Natalya Tatarchuk and Zoe Brawley's 2004 paper[18]. When reading from the height map a piecewise linear approximation of the displacement can be used. Whilst this does lead to straighter, more inaccurate edges of the adjusted surface it results in less samples when stepping into the polygon as the surface between two sample points is approximated by straight lines. This, alongside a depth bias, reduces the stair stepping aliasing greatly, meaning less samples can be used which in turn increases the speed of the algorithm[16].

It is important to extract as much information from research on parallax mapping as it is the main focus point of this study. The algorithm uses a vector between the camera and the object to calculate a texture coordinates offset based on a height maps values then draws these offsets to create an illusion of parallax scrolling on a 2D surface. While there were many problems with the initial technique improvements have been made to alleviate most if not all of these.

Relief Mapping

Relief mapping is a technique very similar to parallax occlusion mapping in that it simulates surface details using raytracing within the pixel shader, relying on a height map with the ability to produce self-occlusion and self-shadowing. There are, however, differences that defines the two methods. The primary difference is that parallax occlusion mapping uses a reverse height map tracing whereas relief mapping does not. Another difference between the two algorithms is that parallax occlusion mapping calculates the textures displacement by extending the view vector between the camera and the object into the mesh and using an approximation of the height map to calculate the new offset, subdividing that extended vector to iteratively sample for the adjusted texture coordinate. This subdivision leads to more accuracy for the trade-off of efficiency. This method is based on a piecewise linear approximation of the height map to reduce the number of calls to the pixel shader for the texture. Relief mapping uses depth texture lookups to iteratively add to the normalised vector and compares the Z values until an intersection is found. This method is more susceptible to stair-stepping aliasing unless additional refinement passes are implemented such as interval mapping[19], an efficient per-pixel height map displacement algorithm that can find the texture offset in less samples without aliasing by precomputing intervals and narrowing them down with every iteration. It should be noted however that combining relief mapping and interval mapping is not computationally cheap. Implementing more refinement passes will lower performance too.

Relief mapping was first introduced Manuel Oliveira et. al[20] in 2000 as a method of simulating 3D surfaces with motion parallax. It was presented as a two-step process to applying a texture with

orthogonal displacement per texel. Firstly the map is converted to a texture using a 1D forward transform, then the resulting relief map is applied to the polygon using traditional mapping techniques. Whilst the initial implementation worked well Oliveira noted that a considerable amount of time of the algorithm was taken by the pre-warping and resampling operations at 94.10% of the time taken, stating that a hardware implementation of the algorithm would reduce this bottleneck. Conclusively there were limitations, the biggest being that the relief textures were only suitable for single-layer images due to multiple layers and samples causing uncertainty of which samples should be connected, rendering the 1D transform useless.

In 2005, Fabio Policarpo et. al[21] improved on the method to alleviate these problems, allowing the algorithm to support rendering complex surfaces such as weaves and multi-layering, for example a chain link, without requiring a large amount of memory as other methods at the time did such as using 3D textures or large datasets.

The study acknowledges relief mapping due to its similarities with the main focus point. It should be noted how relief mapping will solve the equation to step through a mesh to find its modified texture coordinate.

Horizon Mapping

Horizon mapping is an extension technique that gives normal mapped surfaces the ability to cast shadows onto the adjusted surface, giving the results of normal mapping or parallax mapping a much more intense and realistic look. Whilst horizon mapping still does not solve the issue of an objects silhouettes remaining in their unadjusted state it improves mapping techniques that make use of dot3 maps greatly with a much higher fidelity. The method was put forward originally by Nelson Max[22] in 1988 as an improvement to Blinn's bump mapping technique and yielded effective results. Horizon mapping is achieved by generating shadow maps from the dot3 map by sampling light angles against the perturbations.

The technique was later improved in 2000 by Peter Sloan and Michael Cohen[23] In which they integrated horizon mapping into a modern graphics API, improving the technique in terms of performance and, more importantly, interactivity. With this research horizon mapping is now far more suitable for real time situations where light sources would change in position and intensity.

Horizon mapping was included as it is a major extension to normal mapping to vastly improve the final image quality of the light displacement. While it may not be used in the development section of this study it is important to note how it improves on the normal mapping technique and how these methods could potentially improve the clarity of self-occlusion in parallax mapping.

Rendering Techniques

Priority Fill

Also known as the painter's algorithm, priority fill is a simple solution to visibility based on depth in 3D graphics. The name is given due to similarities to how painters create depth in scenes in which the polygons are sorted by depth then drawn farthest to closest. In doing so objects are drawn over other objects or parts of objects that would be hidden by it, thereby solving the problem with visibility. This solution however is inefficient as the algorithm will always render objects that are hidden or invisible to the camera and as a scene becomes increasingly complex more and more objects are being rendered and drawn to the screen that would otherwise not need to be drawn, causing frame rate drops in real-time applications. Whilst the algorithm is simple to implement and mostly failsafe it can fail in the instance of a cyclic overlap, when multiple polygons overlap in such a way that the algorithm cannot determine which polygon is above the others, resulting in an infinite draw loop. This can however be avoided by subdividing or cutting the polygons to allow sorting. One such method for cutting these polygons is Newells algorithm[27] which tests for multiple cases, each test in order of computational difficulty. If all of these tests are false the polygons are split across the overlap. These tests are continually performed until every polygon passes the test. A reverse painters algorithm exists in which nearest objects are drawn first and furthest are drawn last and in

theory this should eliminate all problems with the painters algorithm however in practice it suffers from many of the problems present in the original algorithm.

As mentioned before this is the default rendering method and in theory can cause the renderer to perform multiple unnecessary functions on polygons that ultimately won't be seen by the camera. Despite these drawbacks the intended test scene for this project is not at all complex and may suffer (almost) no performance hit.

Z-Buffer

Though the painter's algorithm is inefficient it is typically the default for drawing objects to a screen. Alternatives exist such as the previously mentioned Z-buffer in which a first pass of the render is saved as a 2D greyscale depth map of all the objects from the perspective of the camera which is then used to determine how close or far an object is to the camera. Whilst Z-buffering is typically much more efficient as it does not need to render objects or parts of objects that will ultimately be hidden to the camera the very nature of the 2D depth map means that transparency will be eliminated as, for example, a window will be taken at face value due to the depth of the polygons that will make the window, meaning the polygons behind the window will not be rendered. There are techniques around this phenomenon though they delay the Z-buffer and add to the computational cost of the method. The first of these techniques is to draw all the visible opaque polygons before drawing the transparent ones, forcing the Z-buffer to consider the objects behind the transparent polygons. This is mostly a rough, quick technique that does work however falls over when attempting to render one transparent object behind another such as looking through one window of a car and out the window on the other side. The second technique is to alpha blend the transparent objects (add transparency to the texture using 0 alpha values). This is a good method for getting around the problem however one thing to note is that Z-buffering by default does not take into account alpha textures and only looks at the polygons, though OpenGL and DirectX has

functions to get around this problem by preventing pixels with a defined alpha value from writing to the Z-buffers though again this method is not perfect and will only reduce the problems.

Deferred Rendering

Deferred rendering is a technique used to improve efficiency and speed of shader and lighting algorithms. If the rendering is deferred all lighting calculations and shaders are not performed until a second pass, similar to post-processing effects. On the first pass only the attributes needed to perform the shaders are stored. These attributes are important information such as object positions, normals and texture maps. The advantage of deferring the shaders and lighting is that only the visible geometry or pixels have the effects applied to them. This means time and resources are not wasted applying the effects to objects or parts of objects that would never be seen to begin with. By deferring the shaders the complexity of managing the shaders resources is also reduced[24] and the graphics pipeline is simplified as everything is broken up into two passes.

Deferred rendering does not come without its disadvantages however as by deferring the calculations until the second pass the renderer will not be able to handle transparency easily and more information will need to be stored in the G-buffer which may cause video memory issues if there are too many texture maps to store. The reason it may not handle transparency is because deferred shading utilises a Z-buffer which stores depth information of the pixels in the form of a greyscale image after the objects in the scene have been rendered. As the information stored in the Z-buffer is a 2D representation it will always bias towards the closer of two or more overlapping objects, effectively eliminating the information of the hidden object. The Z-buffer is an alternative solution to priority fill.

Deferred shading was first introduced by Michael Deering et. al[25] in 1988 as part of a solution to the memory access bottleneck between the processor and the frame buffer due to the floating point chip running faster than the polygon fill times other than very-large-scale-integration systems by using a pipeline of powerful "triangle processors". Though the concept of deferred rendering was

introduced in this paper its first documented use of the name “deferred shading” was not until 1992 when Steven Molnar et. al[26] published a paper outlining the architecture of PixelFlow, a shading language and parallel multi-computer system that aimed to perform high speed image generation whilst avoiding hardware bottlenecks that would otherwise delay the rendering.

The benefits deferred shading offers makes it a better option over the default technique of priority fill as it reduces the amount of polygons needed to have lighting and shaders applied to it only to what is needed. This leads to a faster render which is what this study is attempting. Its immediate drawbacks do not affect the intended test scene of the development study however having to manually delay the shaders until a second pass will be harder to implement.

Conclusion

In conclusion, there have been many rendering techniques to create additional fine detail to (low polygonal) 3D meshes to cut down processing time and allow for real-time rendering of high quality scenery and objects. From the beginning which saw bump mapping by James F. Blinn which used a greyscale texture map to displace light, giving objects the illusion of fine detail where none existed all the way to modern techniques such as parallax occlusion mapping and displacement mapping which not only creates the illusion of fine detail, but also of big detail and shapes thereby alleviating the silhouetting problem and also adding extra effects that will increase the fidelity of the object in real-time such as self-shadowing. However despite looking into what these techniques are and how they work the question still remains on whether parallax occlusion mapping is a feasible option in a real-time scenario as opposed to other commonly used techniques. The study aims to answer this question in the development section of this paper.

Research Methodology

The purpose of research is to provide a better understanding of the subject area and how each different mapping technique differs and what the advantages and disadvantages of each technique

is. Descriptive research is undertaken to go into further detail as to why parallax occlusion mapping is selected as the subject technique for this study instead of other alternative shaders and also clarifies what problems the technique aims to solve.

Parallax occlusion mapping is an existing technique that is used in a variety of different games and is also present in technology demonstrations for many different reasons. This project makes use of that existing information and research to create an interpretation of the technique that can be used in real time applications. With this information kept in mind this study is secondary research as while the methods used to create the final product are original the technique in question is not. A deductive approach is taken to the literature, utilising this existing research to form a hypothesis for this study. As research has already been done on this topic, previous findings and conclusive notes can be used to outline expected results, pinpoint what data is needed to be collected and compare and contrast the actual results.

Three main strategies are taken to obtain the necessary information for this study. The literature review, which is extensively researched above, offers valuable information that forms a theoretical idea to be turned into actual data. Software development, the second strategy, requires a development of the software to collect data from; designing, building and extensively testing the technique to obtain accurate results. Finally an experiment is performed to test the execution times and FPS count of the parallax occlusion mapping shader to ensure the results of the final product meet the standards for use in real time applications and also to compare and contrast against existing results.

Development Methodology

To obtain the results of the project an evolutionary prototyping software development methodology was used. This methodology was chosen over other alternatives as parallax occlusion mapping can

be seen as a series of improvements to the bump mapping technique and so the logical and most efficient way to achieve the results would be to follow these improvements from step one.

The evolutionary prototyping methodology starts with a design before building a rough prototype of the design. The client then tests the prototype and offers feedback for refinement. These last three steps are repeated until the prototype can be labelled as a final product. The product thereby evolves during the process and can end up either being slightly, or vastly different from the original design specification.

Advantages of using evolutionary prototyping is that the final product is more likely to match the clients requirements as they offer input at each stage of evolution to ensure the final product is right for them. This constant communication with the client is also a benefit as they are able to see the product build up and a more of an understanding on how the system works. However a common problem with prototyping models is that it is hard to tell when to finalize the product and stop making small refinements to the current prototype which could cause time issues. In the case of this project this problem could occur when optimising the code for faster execution or when handling small details such as self-occlusion however due to the nature of the technique as previously mentioned it is suitable for efficient building of the shader.

Alternative models to prototyping are agile and waterfall models and both of these models could also have worked well in this project. Waterfall is typically risky as all stages of the project from conceptualisation until maintenance are done chronologically without revisiting certain stages. This means that each stage must be perfected before deciding to move on. While this may increase the time taken to finish the project it does give reliable and high quality results. Because of its one-way nature waterfall models are only suitable for projects that have a defined set of requirements that are not likely to change at all such as this project. A waterfall model was not chosen due to possible time constraints.

Agile models are more accurate and adaptable in terms of an ever changing set of requirements given by the projects clients. The software is programmed in small bits at a time with client input at each iteration in case of a change in requirements until the software is perfected for the user's needs. Agile models are gaining increasing popularity amongst development teams as it offers flexibility for changes. An agile model was not chosen for this project as the project requirements were not likely to change during the development phase. It was also not selected as agile models focus on teamwork between members of a group and/or multiple groups whilst this project is developed by a one man team.

Design

Software Used

Graphics API

For this project OpenGL will be used as the graphics API to write the program. Alternatives include DirectX however OpenGL is chosen as it offers cross-platform support, a non-proprietary (open source) library and personal familiarity.

OpenGL offers many versions and typically the rule of thumb for developers is to use the earliest version possible so that as much hardware is supported as possible. With this kept in mind OpenGL 3.2 is chosen as the version to use for this project. The reason behind this choice (as opposed to selecting an earlier version) is that OpenGL 3 introduced the deprecated function list which ends support for the fixed function pipeline functionality, therefore forcing this project to be completely custom coded in terms of a matrix stack, matrix math functions and a vertex buffer object to load vertex information into the shader. Furthermore, OpenGL 3.2 extends the deprecated function list making it a suitable candidate for this project.

The shader itself will also use GLSL version 330 to be in line with the OpenGL version. Unlike the version of OpenGL, each version of GLSL does not offer many differences in performance however

the primary difference is syntax. This version of GLSL will require the use of in and out prefixes to define the inputs and outputs of the vertex and the fragment shader and also removes the ability to use built-in variables that would otherwise handle the model view projection (MVP) Matrix and vertex information. This will not detriment the project though as the model, view and projection matrices will all be calculated in the CPU, and vertex information will be passed into the shader via a vertex buffer object.

While having to program the MVP matrix and VBO from scratch will be harder in the short run as more is needed to be learnt and debugged – taking up time and resources – it will offer vast improvements to the immediate mode alternatives as more of the program is controlled by the developer and these custom functions will be executed faster and supported by more hardware as they are not deprecated functions like their immediate mode counterparts.

IDE

The integrated development environment (IDE) used will be Visual Studio 2013. Though the year of the brand does not matter (with the only affecting difference being the build tools used to compile the program) it was decided to use Visual Studio over other IDE's as it offered a comprehensive built in library of useful functions, including OpenGL integration, a compiler, easy to use GUI and an integrated debugger. This is everything that will be needed to run and test the project.

Other IDE's include qt creator however this was not selected as while it does build and compile, there is no integrated OpenGL support. Moreover it has windowing functions in the form of drag and drop however it is better if these functions were instead programmed.

Windowing and Controls

The extension used to handle windowing and keyboard functions will be freeglut. Freeglut is a supported and extended open source alternative to glut and was chosen over other windowing API's such as qt creator or VB as it requires the developer to program the windowing functions as opposed to simply using drag and drop features. With that in mind the windowing features in qt creator are

far more comprehensive and easy to use. The keyboard controls are present for demo purposes only to control the positioning of the light source and the rotation of the cube.

Hardware Requirements

The only hardware required for this project to work is a graphics processing unit that supports OpenGL 3.2 or higher. Aside from this specification any hardware is compatible and eligible to host the demonstration. As previously mentioned OpenGL was chosen in part because of its cross platform support so any operating system will be able to run the demonstration.

Implementation

The project requirements were to create an effect that can simulate geometry where none exists without the use of tessellation. The effect must also be able to self-shadow and self-occlude to retain realism under any condition of varying lighting levels and direction of the camera or light in relation to the surface.

During the development stage multiple small prototypes were created at each evolutionary stage that would build up to the final product. Once the object was read from a wavefront obj file and stored into a vertex buffer object a simple Dot3 mapping technique was developed which applies a colour map to the object as well as using a normal map to modify the surface normals at a per pixel level which would displace light to simulate fine detail on a flat surface. The main focus of this step to take into the next step was the TBN matrix which takes the tangent, bitangent and normals of each vertex to convert the object into tangent space instead of world space where the bitangent is the line between two points, in this case, the remaining axis in relation to the X axis (tangents) and Z axis (normals). This was an important first step as when using a normal map in world space all modified surface normals use world coordinates meaning no matter which direction each triangle was facing the new normals would always face positive X in the world. Tangent space however

modifies the normals in relation to the direction of each triangle so that on a cube the bumps would always face outwards in any direction instead of just positive X.

Initially a rough approximation was used to calculate tangents by taking the cross product of the surface normals and either the Z coordinate (0.0, 0.0, 1.0) or the Y coordinate (0.0, 1.0, 0.0) and using the result with the biggest value as the tangents. This caused many issues with the final representation of surface geometry at this method of calculating tangents caused the positive and negative Y faces to be inverted and also experience bad texture swim. This was due to the approximation only assuming the face is on the X axis and was not set to handle Y axis faces so the displacement was effectively clamped to a 2D plane which caused the texture swim. As well as this erroneous appearance the tangents were not consistent with each other, as any vector could have been chosen as the tangent. To overcome this the tangents were calculated in the CPU and passed into the GPU, a less computationally expensive method than doing the calculations in the GPU, by taking the edge of each triangle and the edge of each texture coordinate and applying a coefficient to obtain the final value. This gave accurate results on all faces that were consistent as they took into account information such as texture coordinates and the edges of the triangles surface. The determinant value can be found below.

```
float determinant = 1.0 / ((texEdge1[0] * texEdge2[1]) - (texEdge1[1] * texEdge2[0]));
```

The second step introduced the initial parallax mapping technique as described by Tomomichi Kaneko where texture coordinates were displaced based on the values of a height map to simulate motion parallax. The technique worked well to a certain extent however as the viewing angle became more shallow the effect began to fall over as the texture began to ripple. This was caused by the fact that only one sample to find the displaced texture coordinate is taken between the meshes actual surface and the lowest possible value. The best way to describe this phenomenon would be to imagine it as binary. If the displaced texture is on the meshes surface (0) then the coordinate is not changed and remains where it is, but when the texture coordinate is displaced (1) it is moved to the

lowest value. This was not good enough for a game or demo environment and multiple samples would be needed to accurately find each fragments depth value.

Step three emulated steep parallax mapping based on the descriptions given by Morgan and Max McGuire where the eye vector steps through the mesh multiple times to find the correct displacement amount. There were two ways of doing this: using a binary search to half the distance between each sample, thereby increasing precision twofold each step, or equally distancing each sample to offer more consistent results. The binary search tends to be more accurate and is used by the relief mapping technique, an extremely similar alternative to parallax occlusion mapping, however due to its precision, performance is reduced as more samples are required for a good result. By equally distancing each sample an approximation is found fairly quickly but too few samples result in a noticeable stair stepping aliasing as seen in Appendix 1. Despite this drawback it was selected for the project as parallax occlusion mapping improves the search.

Step four modified the steep parallax mapping technique to include piecewise linear interpolation and self-occlusion. The linear interpolation was a simple modification to the existing code in which the previous sample was stored and the surfaces displacement depth was interpolated between the two points. This offers an optimisation of the technique as the interpolation allowed for acceptable visual results at a lower number of samples as the stair stepping aliasing is being fixed by the interpolation between the two points. This does not offer an ultimate solution to optimisation though as an extremely low number of samples will either still cause the aliased effect or the distance between two samples will be too great that the linear interpolation will be noticeable. Self-occlusion/shadowing was achieved by essentially doing the same loop for the light vector as was done with the eye vector. A set number of samples was used to step into the mesh with the light vector. For every step into the mesh a counter would rise, increasing a modifier that affects the final light level of that fragment. This increased the sense of depth as deeper fragments inherently became darker. This modifier was then multiplied with the mean average of all channels of the

normal map and the height map to add a bias to bumps and dips, increasing the effect of self-occlusion as these small details were further darkened. While the use of the averaged value worked well to give the effect of self-occlusion there is a problem of shadow bias towards the lower and left side of each bump as these are higher values in the red and green channels of the normal map however this erroneous lighting is only noticeable on bigger surface details such as cobblestone.

Many other small adjustments were made during this step to improve the overall quality of this effect. A simple if statement was used to check if the current texture coordinates extended below zero or above one and, if so, deleted them. Doing so gave the mesh a silhouette which further gave the illusion of high surface detail on the simple cube. This check was necessary as when the viewing angle reaches extremely shallow angles the texture coordinate displacements reaches infinity giving the illusion that, within the confines of the cubes mesh, the texture repeated infinitely. By clamping the texture coordinates to a value between zero and one these repeating fragments were deleted leaving only the fragments that exist on the texture map as is which thereby gave a silhouette. Another adjustment was removing the use of depth bias to improve the quality of the approximated surface at all angles. When a depth bias is applied to the parallax mapping loop the surface would appear to flatten as the normals become more parallel with the view vector. Removing the depth bias allowed the depth to stay consistent at all angles, giving a perspective correct sense of depth. This effect can be seen in Appendix 2.

A visualisation of the evolution of the prototype can be found in Appendix 1. Note that the steep parallax mapping step is not included as the result is extremely similar to the parallax occlusion mapping step. As mentioned above the differences between these two steps are more focused on optimisation and occlusion.

Results

All tests were performed with an NVidia GTX 960 graphics processing unit containing 2GB of on-board memory. Additionally an Intel i7-6700k was used to handle the CPU side of the project. The texture maps loaded and applied to the cube were one 24-bit bitmap and one 32-bit bitmap, both at a size of 2048x2048 pixels. All of the above factors affect performance and results may (drastically) differ if these change.

FPS Test

Results from this test were taken by measuring the frames per second (FPS) once per second for one minute. The results were then averaged by mean to obtain an accurate final number. The test compared results between no active shaders, a simple Blinn-Phong lighting implementation, the final parallax occlusion implementation and bump mapping. It is worth noting that this test is performed simply to measure the impact of frames per second this implementation has against other common shaders and many other variables affect the final FPS count that are not directly linked with the shader including, but not limited to; optimisation of CPU side code, communication times between the CPU and GPU, texture resolution, polygon count, the hardware used and even the print function used to display the FPS. However while these factors affect the final FPS count of the demonstration they all remain constant between each test so the results are still valid.

Blinn-Phong Lighting	Blinn-Phong + Mesh at 480,000 Triangles	Blinn-Phong + Parallax Mapping	Blinn-Phong + Parallax Occlusion Mapping*
1285FPS	49FPS	903FPS	452FPS

*Parallax Occlusion Mapping was performed using 30 offset samples and 10 occlusion samples. This sample count was selected based on the results of the acceptance testing below when finding a minimum acceptable sample count.

Acceptance

Testing

To test whether the results of this project were convincing ten subjects were given the project demonstration and asked whether what they were seeing was in fact a simple cube. Using the brickwork texture, out of the ten subjects, nine were convinced at thirty samples or more that the demonstration contained more than six polygons. However when using the cobblestone texture all ten subjects admitted that while the effect still gave a convincing sense of depth the edges of the mesh gave the effect away. These results conclude that for non-seamless or large detail textures a border is required to retain the quality of the effect at all angles. For large detail textures a cube with chamfered edges may also fix the erroneous edge artefact.

Evaluation

While this data does not inform whether this technique is a suitable replacement for alternative techniques such as tessellation and displacement mapping, it does show that the performance hit caused by the shader is roughly 2.84 times slower than when using a shader that simply calculates accurate lighting and around two times slower than when using bump mapping. However visually the results of the shader offer far more surface detail than bump mapping. Therefore while the performance hit can be seen as significant at two times slower than bump mapping, it is still in relative suitable for use in real time applications. The choice must be made whether the extra visual clarity is indeed worth this performance hit.

Furthermore the shader is 9.2 times faster than rendering a high polygon mesh with the same level of surface detail however results are still pending for a low polygon mesh containing large surface details in combination with a bump mapping shader for finer details. This shows that to some extent using the parallax occlusion mapping is much more suitable for real time environments than the use of high polygon meshes.

The result of this project did support the research done in the literature review as the technique offered a vastly superior alternative to bump mapping however some issues with the technique that were mentioned were also revealed such as the erroneous edges on large displacements caused by the displacement being limited by the meshes geometry. The results of this testing were expected in comparison to the research previously undertaken in the literature review. The aims and objectives were met as sufficient testing was done to compare and contrast the shader with other commonly used techniques and the shader itself worked as intended.

Conclusive notes

The literature review was important as it gave some background information on the project on what the technique is and how it works, and also what tools were needed to develop the shader. Throughout the literature review many different techniques for simulating geometry on a flat surface was researched such as bump mapping, horizon mapping, parallax mapping and displacement mapping. Reading this existing material not only gave an idea into the problems of each techniques but also the issues they remedy and why.

Bump mapping was created as a method of simulating fine detail on a mesh by displacing the light using a height map. The technique worked well and was very optimal for use in real time however was not always accurate on moving objects. Normal mapping improved on this technique by converting the surface normals of a high polygon model into a 2D texture map, storing the X, Y and Z values in the R, G and B channels respectively. This vastly improved on the accuracy of the light displacement as it modified the surface normals in tangent space as opposed to setting an intensity value for light. Parallax mapping was created to simulate motion parallax on flat surfaces by using a height map to displace texture coordinates based on the view vector. This method worked well but only for certain angles as only a single sample was taken and so parallax occlusion mapping improved this technique in a number of ways by increasing the number of samples to check for the height field intersection and add self-occlusion by taking into account the light vector as well as the

view vector. Displacement mapping however subdivides the mesh and displaces the mesh based on the height map, offering a high polygonal alternative.

The research methodology undertaken was secondary research as the technique already exists and this study aimed to develop an interpretation of parallax occlusion mapping. A deductive approach was taken to the literature to develop an understanding of how the technique works and testing this information in the development. The hypothesis formed from this research approach is that the parallax occlusion mapping shader would be slower than traditional techniques used in real time applications such as bump mapping however will offer far more advanced visual clarity than the aforementioned technique and will still be vastly less computationally expensive than rendering a high polygon mesh with the same level of detail as the parallax occlusion mapping technique.

Evolutionary prototyping was chosen as the software development methodology as the literature suggested that the parallax occlusion mapping technique was in fact a series of improvements to the normal mapping technique. By using evolutionary programming this series of improvements could be followed from step one to not only achieve the desired results but also to gain a better understanding of the problems and solutions to each of the techniques.

The design section outlined the hardware requirements and the software choices for this development project. There were few hardware requirements to run this project as the only limitation of the project was the fact that it was written in OpenGL 3.3. This meant that the graphics card of the machine hosting the demonstration must be able to support at least OpenGL 3.3. Many factors were put into the selection of the various software used to develop the project and ultimately these choices offered good support for the techniques requirements.

When implementing the program evolutionary programming was followed closely. Firstly a normal mapping technique was created to simulate geometry by displacing surface normals in tangent space. It was this TBN matrix that converts coordinates from world space to tangent space that was taken from this step as when displacing normals and texture coordinates the displacement must be

done on the correct axis in relation to the triangles tangents. Secondly the original parallax mapping technique was implemented to improve on the normal mapping technique which displaced texture coordinates well but fell over on shallow angles. To improve on this steep parallax mapping was implemented to find the correct height field intersection point using multiple samples. This method vastly improved the visual aspect at a much greater performance hit as shown in the FPS test. Finally the self-occlusion loop was implemented to offer more realistic lighting on the simulated geometry and a piecewise linear interpolation function was added to the displacement loop to optimise the effect by reducing the amount of samples needed.

When the implementation was completed testing was done in two ways to ensure the project met the initial aims. An FPS test was conducted to measure the average frames per second for the parallax occlusion mapping implementation which was then compared and contrasted against the previous evolutionary steps to get a better understanding of the performance hit of the technique. The results confirmed the hypothesis formed during research that the final implementation was visually more accurate than more common techniques such as bump mapping whilst being more computationally expensive at around two times less frames per second. A test was also done for a high polygon mesh with the same level of surface detail created using displacement mapping however the parallax occlusion mapping appeared to be 9.2 times quicker than this method of surface detail.

Acceptance testing was also performed to an audience of 10 subjects to test how convincing the technique was at simulating real geometry. The results concluded that larger or more rounded displacements required chamfered edges or a border object to remain completely convincing.

Reflection

Many changes needed to be made from the initial design specification which was expected as per the prototyping methodology however this caused problems that were primarily time consuming.

Whilst focusing on writing the shader an immediate mode renderer was used (`glBegin/glEnd`) to store and display vertex information. Because this method of rendering did not support the storing of surface tangents an approximated calculation was used in the vertex shader. Firstly doing these calculations in the graphics processing unit were inherently slow and wasted GPU cycles and secondly these approximated tangents caused many issues that affected the final result of the shader. When calculating tangents in this way it assumes the vertices normals are always on the X axis and so for the top and bottom polygons of the cube the texture coordinate displacement direction was incorrect, causing an inverted appearance and bad texture swim as the displacement was clamped to the X axis. The approximated tangents also appeared to be inverted which meant an inversion on the calculation needed to be done. To fix this error the tangents had to be calculated properly for every triangle. This meant the immediate mode renderer had to be replaced with a vertex buffer object (VBO) where three vertices were read from the wavefront obj file and stored at a time and tangents were calculated for each triangle. Many issues were encountered when attempting to get the VBO to render which caused delays on finalisation of the code such as clean-up, commenting and optimisation.

Other problems occurred throughout the development process that did not cause much of a delay but were problematic nonetheless. Many of these were small yet elusive mistakes such as typos.

Whilst a Gantt chart was created and followed relatively accurately time management was somewhat of an issue as the Gantt chart did not take into account error checking or problem solving. This caused a bit of a rush to get the final report written as more work than anticipated was put on fixing the demonstration.

Future Work

Improvements can be made to the current state of this technique as it was set out to be a core parallax occlusion mapping showcase. Parallax mapping is very flexible in that it can be used in

combination with other mapping techniques such as bump mapping for finer details, horizon mapping, specular mapping and occlusion mapping and in doing so can make the effect seem more realistic and potentially even quicker in the case of bump mapping where less height information is stored in the height map and finer detail is instead done only on the lighting calculations.

Other performance improvements to the technique can also be made: Instead of using a set number of samples for the eye and light vectors to step into the mesh a range of samples can be used depending on the dot product between, for example, surface normals and the camera direction where a parallel angle will use more samples to retain detail at shallow angles whilst a more perpendicular angle will use less samples as the texture coordinate displacement is almost or exactly the same as the non-displaced counterpart and therefore does not need more samples. Usually the disadvantage of doing this would be that the audience will be able to see the amount of samples grow as the angle changes in the form of stair stepping aliasing growing or shrinking however as parallax occlusion mapping uses piecewise linear interpolation the stair stepping effect is effectively eliminated.

Parallax occlusion mapping is not limited to a single pass and can be deferred to a second pass within the graphics buffer. Further research can be done to measure the difference in terms of performance and visual fidelity when using this method.

The effect could be optimised further in terms of sample count as further work can be done to experiment with the use of non-linear interpolation or approximation. This should allow for even less samples to be used whilst retaining a good visual quality as each sample is interpolated using curves, removing the downside of linear interpolation where too few samples causes the displacement to be too straight. Additional testing may need to be done to see whether this method is more optimal and, if not, whether the visual difference is worth the extra performance hit.

Glossary

Chamfer – A sloped surface on an edge or corner. Chamfering is a method used to soften or round off the sharp edges of meshes in 3D modelling.

Fragment Shader – A type of shader that outputs colour information for each pixel.

Grazing Angle – The angle between the camera vector and the surface normal.

Immediate Mode Rendering - A method of rendering in which functions on the CPU directly cause the rendering of graphics.

Micropolygon – A polygon the size of (or less than) a single pixel.

Mipmapping – A technique in which a high resolution image is scaled into multiple levels of detail.

Motion Parallax – The effect in which a closer object will move quicker than an object further away.

Normalisation – Outputting a vector of the same direction but with a length of one.

Shader – A program that can perform rendering effects on the graphics processing unit.

Texture Swim – An effect in which the texture on a surface appears to move or swim across the mesh as it rotates.

Wavefront obj – an object file that stores three indices per vertex, ordered: X/X/X Y/Y/Y Z/Z/Z (Vertex/Texture/Normals)

Vertex Buffer Object – A non-immediate mode rendering method for uploading vertex data such as position, normals and texture coordinates.

Vertex Shader – A type of shader that runs once every vertex with the purpose of converting the vertices positions into 2D space for the screen. Each value between the vertices are interpolated for the fragment shader.

References

- [1] J. F. Blinn, "Simulation of wrinkled surfaces," *ACM SIGGRAPH Comput. Graph.*, vol. 12, no. 3, pp. 286–292, Aug. 1978.
- [2] N. L. Max and B. G. Becker, "Bump shading for volume textures," *IEEE Comput. Graph. Appl.*, vol. 14, no. 4, pp. 18–20, Jul. 1994.
- [3] M. J. Kilgard, "A Practical and Robust Bump-mapping Technique for Today's GPUs," *Game Dev. Conf. 2000*, pp. 1–39, 2000.
- [4] M. Peercy, J. Airey, and B. Cabral, "Efficient bump mapping hardware," *Proc. 24th Annu. Conf. Comput. Graph. Interact. Tech. - SIGGRAPH '97*, pp. 303–306, 1997.
- [5] V. Krishnamurthy and M. Levoy, "Fitting smooth surfaces to dense polygon meshes," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques - SIGGRAPH '96*, 1996, pp. 313–324.
- [6] J. Cohen, M. Olano, and D. Manocha, "Appearance-preserving simplification," in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques - SIGGRAPH*

'98, 1998, pp. 115–122.

- [7] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno, "A general method for preserving attribute values on simplified meshes," in *Proceedings Visualization '98 (Cat. No.98CB36276)*, pp. 59–66, 1998.
- [8] R. L. Cook, "Shade trees," *ACM SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 223–231, Jul. 1984.
- [9] M. Pharr and R. Fernando, "Chapter 8. Per-Pixel Displacement Mapping with Distance Functions," in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, 2005, p. ???
- [10] L. Wang, X. Wang, X. Tong, S. Lin, S. Hu, B. Guo, and H.-Y. Shum, "View-dependent displacement mapping," *ACM Trans. Graph.*, vol. 22, no. 3, p. 334, Jul. 2003.
- [11] S. Gumhold and H. Tobias, "Multiresolution Rendering With Displacement Mapping," in *Proceedings of the ACM SIGGRAPH 2006 Courses on - SIGGRAPH '06*, 2006, p. 81.
- [12] M. Doggett, J. Hirche, and W. S. I. Gris, "Adaptive View Dependent Tessellation of Displacement Maps @ A," vol. 59, no. c, 2000.
- [13] T. Kaneko, T. Takahei, M. Inami, N. Kawakami, Y. Yanagida, T. Maeda, and S. Tachi, "Detailed Shape Representation with Parallax Mapping," *Proc. ICAT 2001*, pp. 205–208, 2001.
- [14] T. Welsh, "Parallax mapping with offset limiting: A per-pixel approximation of uneven surfaces," *Infiscape Corp.*, pp. 1–9, 2004.
- [15] M. McGuire and M. McGuire, "Steep Parallax Mapping," *I3D 2005 Poster*, pp. 2005–2005, 2005.
- [16] N. Tatarchuk, "Practical parallax occlusion mapping with approximate soft shadows for detailed surface rendering," in *ACM SIGGRAPH 2006 Courses on - SIGGRAPH '06*, 2006, p. 81.
- [17] N. Tatarchuk, "Dynamic parallax occlusion mapping with approximate soft shadows," in *Proceedings of the 2006 symposium on Interactive 3D graphics and games - SI3D '06*, 2006, p.

63.*

- [18] Z. Brawley and N. Tatarchuk, "Practical parallax occlusion mapping with approximate soft shadows for detailed surface rendering," in *ShaderX3: Advanced Rendering with DirectX and OpenGL*, 2004, pp. 135–154.
- [19] I. B. Borecki and G. P. Vogler, "Interval mapping of quantitative trait loci using a sib-pair linkage method," *Genet. Epidemiol.*, vol. 12, pp. 723–728, 1995.
- [20] M. M. Oliveira, G. Bishop, and D. McAllister, "Relief texture mapping," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques - SIGGRAPH '00*, 2000, pp. 359–368.
- [21] F. Policarpo and M. M. Oliveira, "Relief mapping of non-height-field surface details," in *Proceedings of the 2006 symposium on Interactive 3D graphics and games - SI3D '06*, 2006, p. 55.
- [22] N. L. Max, "Horizon mapping: shadows for bump-mapped surfaces," *Vis. Comput.*, vol. 4, no. 2, pp. 109–117, Mar. 1988.
- [23] P.-P. J. Sloan and M. F. Cohen, "Interactive Horizon Mapping," 2000, pp. 281–286.
- [24] S. Hargreaves and M. Harris, "Deferred Shading," *Present*.
http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf Hoppe, H, pp. 59–66, 2004.
- [25] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt, "The triangle processor and normal vector shader," in *Proceedings of the 15th annual conference on Computer graphics and interactive techniques - SIGGRAPH '88*, 1988, pp. 21–30.
- [26] S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: high-speed rendering using image composition," *ACM SIGGRAPH Comput. Graph.*, vol. 26, pp. 231–240, 1992.
- [27] M. E. Newell, R. G. Newell, and T. L. Sancha, "A new approach to the shaded picture problem," *Proc. ACM Natl. Conf.*, pp. 443–450, 1972.

*Please note that the title of this journal may also refer to a presentation of the same title. The referenced material can be found here:

<http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Tatarchuk-POM-SI3D06.pdf>

Bibliography

D. Shreiner, G. Sellers, J. Kessenich, B. Licea-Kane (2013). *OpenGL Programming Guide*. Boston: Addison Wesley.

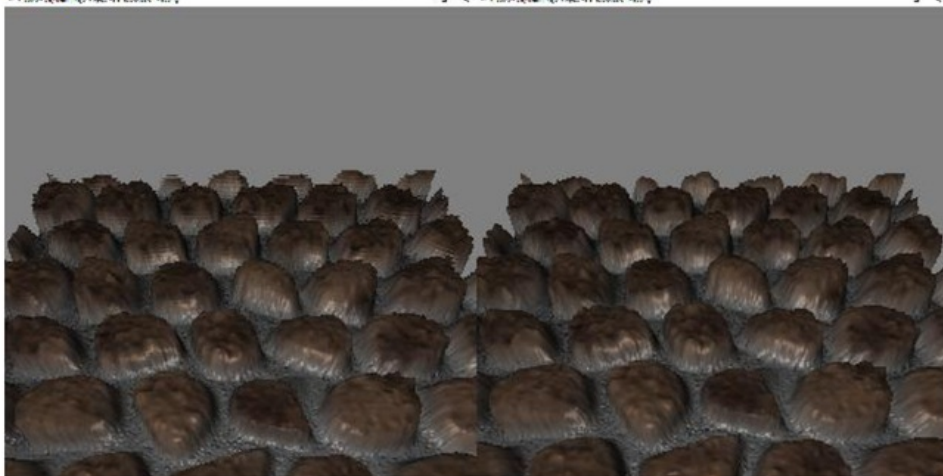
R. Rost, B. Licea-Kane, D. Ginsburg, J. Kessenich, B. Lichtenbelt, H. Malan, M. Weiblen (2009). *OpenGL Shading Language*. 3rd ed. Boston: Addison Wesley.

Appendices

Appendix 1

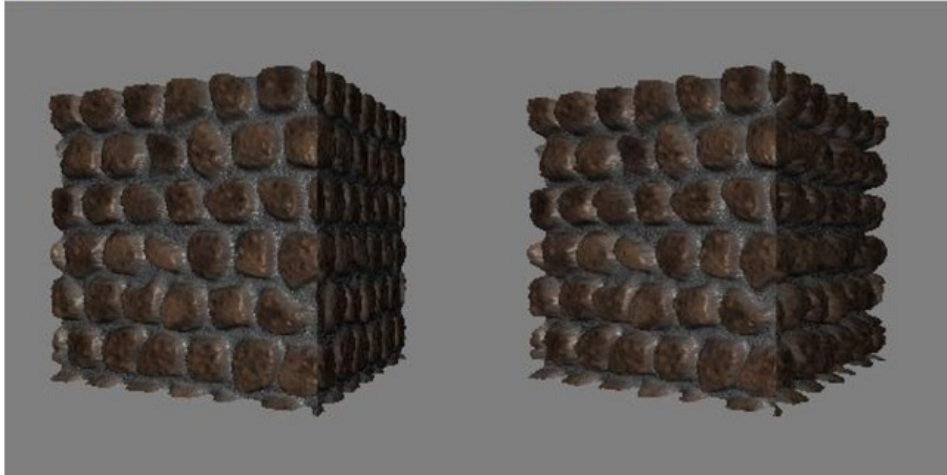
AMDBoard, *ATI Toyshop Demo*. 2007 [Online]. Available:
<https://www.youtube.com/watch?v=LtxvpS5AYHQ>. [Accessed: 6-November-2015]

Appendix 2

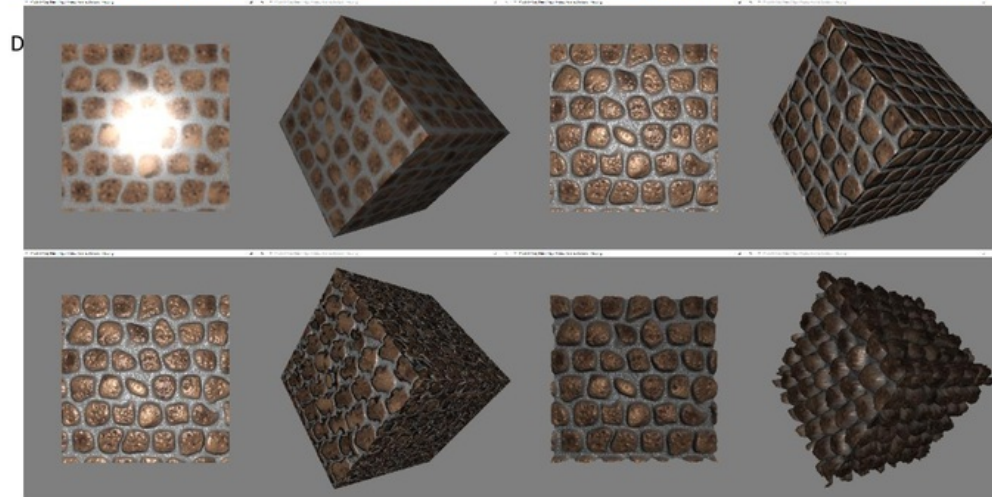


This image highlights the stair stepping artefacts seen when using a low number of samples when stepping through the mesh. **Left:** ten offset samples are used. Stair stepping is clearly visible. **Right:** thirty samples are used. Stair stepping is barely noticeable on first glance.

Appendix 3



Left: Depth bias is applied. As the surface normals reach grazing angles the offset depth is reduced. Right:



Left to right, top to bottom: 29166706427971 A textured cube with Blinn-Phong illumination. Blinn-Phong with normal mapping. The original parallax mapping technique, showing the visual error seen at shallow angles. Parallax occlusion mapping using thirty offset samples and ten occlusion samples.

Appendix 5



The texture maps used in the demo. **From left to right:** colour map, normal map, height map. The height map was stored inside the normal maps alpha channel to save on initial loading times. Note that the height map is inverted to what is typically used. This is because inside the parallax occlusion mapping loop zero is treated at the meshes surface and one is treated as the lowest displacement point, so the value of zero becomes the highest point in the surface.

Appendix 6

```

//////////Calculate the tangents using the edge of the triangle and the edge of the texture coordinates//////////
float edge1[3];
edge1[0] = obj[j - 1].vertices[0] - obj[j - 2].vertices[0];
edge1[1] = obj[j - 1].vertices[1] - obj[j - 2].vertices[1];
edge1[2] = obj[j - 1].vertices[2] - obj[j - 2].vertices[2];

float edge2[3];
edge2[0] = obj[j].vertices[0] - obj[j - 2].vertices[0];
edge2[1] = obj[j].vertices[1] - obj[j - 2].vertices[1];
edge2[2] = obj[j].vertices[2] - obj[j - 2].vertices[2];

float texEdge1[2];
texEdge1[0] = obj[j - 1].texcoords[0] - obj[j - 2].texcoords[0];
texEdge1[1] = obj[j - 1].texcoords[1] - obj[j - 2].texcoords[1];

float texEdge2[2];
texEdge2[0] = obj[j].texcoords[0] - obj[j - 2].texcoords[0];
texEdge2[1] = obj[j].texcoords[1] - obj[j - 2].texcoords[1];

float determinant = 1.0 / ((texEdge1[0] * texEdge2[1]) - (texEdge1[1] * texEdge2[0]));

obj[j - 2].tangents[0] = (edge1[0] * texEdge2[1] - edge2[0] * texEdge1[1]) * determinant;
obj[j - 2].tangents[1] = (edge1[1] * texEdge2[1] - edge2[1] * texEdge1[1]) * determinant;
obj[j - 2].tangents[2] = (edge1[2] * texEdge2[1] - edge2[2] * texEdge1[1]) * determinant;

obj[j - 1].tangents[0] = (edge1[0] * texEdge2[1] - edge2[0] * texEdge1[1]) * determinant;
obj[j - 1].tangents[1] = (edge1[1] * texEdge2[1] - edge2[1] * texEdge1[1]) * determinant;
obj[j - 1].tangents[2] = (edge1[2] * texEdge2[1] - edge2[2] * texEdge1[1]) * determinant;

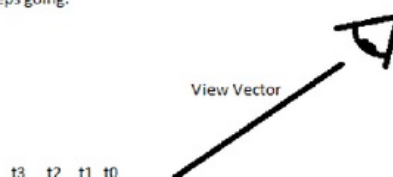
obj[j].tangents[0] = (edge1[0] * texEdge2[1] - edge2[0] * texEdge1[1]) * determinant;
obj[j].tangents[1] = (edge1[1] * texEdge2[1] - edge2[1] * texEdge1[1]) * determinant;
obj[j].tangents[2] = (edge1[2] * texEdge2[1] - edge2[2] * texEdge1[1]) * determinant;

```

A screenshot of the code used to calculate the surface tangents per triangle. The edge between the second and first vertex is found as well as the edge between the third and first vertex. The same is done with the texture coordinates before the tangents are calculated using a determinant.

Appendix 7

t0 - Original Texture Coordinate.
t1 - First Sample. Offset is not correct so algorithm keeps going.
t2 - Second Sample. Offset is not correct so algorithm keeps going.
t3 - Modified Texture Coordinate.



A simple diagram illustrating how steep parallax mapping works. The view vector steps into the mesh at intervals until the height map surface is found.