NOTTINGHAM
TRENT UNIVERSITY

Hardware Accelerated Computer Graphics Algorithms

Daniel Thomas Rhodes

May 2008

A thesis submitted in partial fulfilment of the requirements of Nottingham Trent University for the degree of Doctor of Philosophy

School of Science and Technology
Nottingham Trent University
Clifton Lane
Nottingham
NG11 8NS

Table of Contents

Abstract

The advent of shaders in the latest generations of graphics hardware, which has made consumer level graphics hardware partially programmable, makes now an ideal time to investigate new graphical techniques and algorithms as well as attempting to improve upon existing ones.

This work looks at areas of current interest within the graphics community such as Texture Filtering, Bump Mapping and Depth of Field simulation. These are all areas which have enjoyed much interest over the history of computer graphics but which provide a great deal of scope for further investigation in the light of recent hardware advances.

A new hardware implementation of a texture filtering technique, aimed at consumer level hardware, is presented. This novel technique utilises Fourier space image filtering to reduce aliasing. Investigation shows that the technique provides reduced levels of aliasing along with comparable levels of detail to currently popular techniques. This adds to the community's knowledge by expanding the range of techniques available, as well as increasing the number of techniques which offer the potential for easy integration with current consumer level graphics hardware along with real-time performance.

Bump mapping is a long-standing and well understood technique. Variations and extensions of it have been popular in real-time 3D computer graphics for many years. A new hardware implementation of a technique termed Super Bump Mapping (SBM) is introduced. Expanding on the work of Cant and Langensiepen [1], the SBM technique adopts the novel approach of using normal maps which supply multiple vectors per texel. This allows the retention of much more detail and overcomes some of the aliasing deficiencies of standard bump mapping caused by the standard single vector approach and the non-linearity of the bump mapping process.

A novel depth of field algorithm is proposed, which is an extension of the authors previous work [2][3][4]. The technique is aimed at consumer level hardware and attempts to raise the bar for realism by providing support for the "see-through" effect. This effect is a vital factor in the realistic appearance of simulated depth of field and has been overlooked in real time computer graphics due to the complexities of an accurate calculation. The implementation of this new algorithm on current consumer level hardware is investigated and it is concluded that while current hardware is not yet capable enough, future iterations will provide the necessary functional and performance increases.

Acknowledgements

Thanks to Richard Cant and Caroline Langensiepen for being always available and always helpful, despite heavy schedules and administrative headaches. Helen Knight, for putting up with this research taking over my life for so long. Al Denby, my desk neighbour, for having suffered similar pains and shown that there is light at the end of the tunnel. Jonathan Townsend, who despite sitting through my stress fuelled rants still has all this to look forward to. The examiners, Andy Day and Peter Fitzgerald. Finally thanks to NVIDIA, ATI and Microsoft for their lack of documentation giving me plenty to do.

Chapter 1: Introduction

Games and simulations form a multi-billion dollar section of the entertainment industry, with 106.37 million GPUs[1] sold worldwide in the final quarter of 2007 [5] and a total of $8.64bn worth of games sold in the US alone during 2007 [6]. Their increasing popularity is partly due to the ever evolving sophistication of computer rendered visual images; which is driven by developments in consumer level graphics hardware in both the PC[2] and console sectors.

Over recent years computer graphics hardware for the consumer market has developed at a pace which is well above that defined by Moore's law [7]. Moore's law is commonly cited with reference to the rapidly continuing advance in computing performance per unit cost, as the increase in transistor count is also a rough measure of processing performance. Therefore according to Moore's law, the performance per unit cost, approximately doubles every two years. With graphics hardware this rapid advancement is largely due to the push towards what NVIDIA has termed as “Cinematic Computing” [8]; essentially by this they mean that they are striving for real-time computer graphics to approach the type of effects which have previously only been available with pre-rendered techniques such as ray-tracing.

In turn, this has made possible recent advances, such as the current generations of

1 **G**raphics / **G**raphical **P**rocessing **U**nit also known as the VPU or **V**isual **P**rocessing **U**nit.
2 **P**ersonal **C**omputer

programmable GPU's. This programmability comes in the form of shaders; which have been prevalent in graphics hardware since 2001, when the NVIDIA GeForce3 was launched. As such shaders are a relatively new development which are still in their infancy; it has yet to be seen just how far they can be pushed and what types of graphical algorithms best suit the new hardware structure they provide.

Shaders provide the programmer with greater flexibility by allowing them to define vertex and / or fragment (pixel) shader programs which can be run on any supporting hardware. For example, they allow the programmer to replace the old style fixed function lighting with shader programs which are able to perform more advanced lighting techniques. So where previously the only options available may have been, for example, Gouraud shading [9], the programmer now has the option to create their own effects such as anisotropic lighting [10] or cell shading[3] [11].

This ongoing "quest for realism" is leading to the requirement for much more advanced and complex techniques in computer graphics systems. As such a great deal of research and development time is invested in researching possible new uses and techniques for shaders by hardware vendors such as NVIDIA and ATI, API maintainers such as Microsoft and SGI, as well as by games and simulation developers. Much research into new and improved techniques is devoted to simulation of real world optical and lighting effects. One such optical effect is the depth of field phenomenon, which has yet to be modelled with any great degree of accuracy in real-time computer graphics. Other desirable techniques include long standing issues within computer graphics such as

3 Also known as cartoon rendering.

texture filtering, which currently offers an unfortunately large trade off between quality and performance on current hardware, and bump mapping, which; while being a long-standing and well established process still has problems caused by previous hardware restrictions and the tendency of developers to use traditional tried and tested methods despite their known defects.

The advent of shaders in the latest generations of GPU's, making consumer level graphics hardware partially programmable, makes now an ideal time to investigate new techniques and attempt to improve upon existing ones. This can be achieved by taking advantage of the new programmable architectures to create new techniques, implementing techniques previously made impossible by hardware restrictions and improving upon existing techniques.

This work looks at several of these current areas of interest within the graphics community including Texture Filtering, Bump Mapping and Depth of Field simulation. These are described below.

A novel depth of field algorithm is presented, which is an extension of the authors previous work [2][3][4]. The technique is aimed at consumer level hardware and attempts to raise the bar for realism by providing support for the "see-through" effect. This effect is a vital factor in the realistic appearance of simulated depth of field and has been overlooked in real time computer graphics due to the complexities of creating a convincing simulation of it.

A new hardware implementation of a texture filtering technique, aimed at consumer level hardware, is also presented. The novelty of this technique lies in the utilisation of programmable consumer level hardware combined with the use of Fourier space to aid filtering, and hence reduce aliasing. The technique provides reduced levels of aliasing compared to many currently popular techniques along with comparable levels of detail retention. This technique adds to the communities knowledge by expanding the range of techniques available which have the potential to offer real-time performance along with easy integration with current and future hardware.

Bump mapping is a long-standing and well understood technique, variations and extensions of it have been popular in real-time 3D computer graphics for many years. A new hardware implementation of a technique termed **S**uper **B**ump **M**apping (SBM) is presented. Expanding on the work of Cant and Langensiepen [1], the SBM technique adopts the novel approach of using normal maps which contain information about multiple vectors per texel. This allows the addition and retention of much more detail into a standard normal map and overcomes some of the aliasing deficiencies which are a common problem with bump mapping. These deficiencies are often caused by the standard single vector approach and the non-linearity of the bump mapping process, more detail is provided on this in Chapter 5.

These techniques are presented in chronological order of investigation, beginning with Depth of Field, then Fourier textures and finally Super Bump Mapping.

Chapter 2: Overview of Graphics Hardware

Graphics hardware provides the means to convert 3D data into 2-dimensions for display on a computer monitor or television. This is done by processing and outputting scene data a pixel at a time. The pixel data is often divided up by polygons[4] and polygon vertices within the hardware for processing convenience and to aid parallelism.

Modern graphics hardware comes in two main variates, graphics cards and on-board graphics. On-board graphics is a term used to describe graphics chips which are integrated into a computers motherboard, this type of graphics hardware is generally considered basic and not up to the demands of modern games and simulations. Graphics cards on the other hand generally provide the higher end of the performance spectrum and can be found mainly in gaming PCs. Although even modern games consoles use graphics processors, which are derived from high-end PC graphics card processors.

There are currently two main methods within most types of consumer level graphics hardware for processing polygon data: the older style fixed function pipeline, as well as the newer and more flexible programmable pipeline. More detail can be found in Sections 2.1 and 2.2, both methods are also discussed by Rhodes et al. [12].

4 Usually triangles in modern systems.

2.1 The Fixed Function Pipeline

The fixed function pipeline is one of two methods available to the API's[5] for handling vertex and pixel processing along with any transformations, lighting and textures that are required by the scene. While the Fixed function pipeline is being rapidly replaced by the programmable pipeline, it remains an important aspect of graphics hardware evolution and for a substantial period of time represented the only option available to games and simulation developers.

Figure 1 [13] illustrates some of the many processes involved throughout the fixed function pipeline. The first task of the fixed function pipeline is to apply any transformations that are required to the geometry. World, View and Projection transformations are applied to the incoming vertices. This positions the geometry appropriately within the co-ordinate system being used. World transformations change the co-ordinates from model space, where points are defined relative to the origin of the model, to world space; where points are defined relative to an origin common to all models (objects) within the scene. View transformations locate the viewer within world space and transform the vertices to camera space; this places the viewer (camera) at the origin of world space. Projection transformations usually handle scaling and perspective translation; however this is not necessarily the case.

Once all the geometry transformations have been performed then clipping and scaling can take place. Clipping essentially removes any excess from the image that exceeds the

5 **A**pplication **P**rogramming **I**nterface.

bounds of the viewport (for example: anything that overlaps the boundaries of the screen) then the results are passed along to the rasteriser[6].

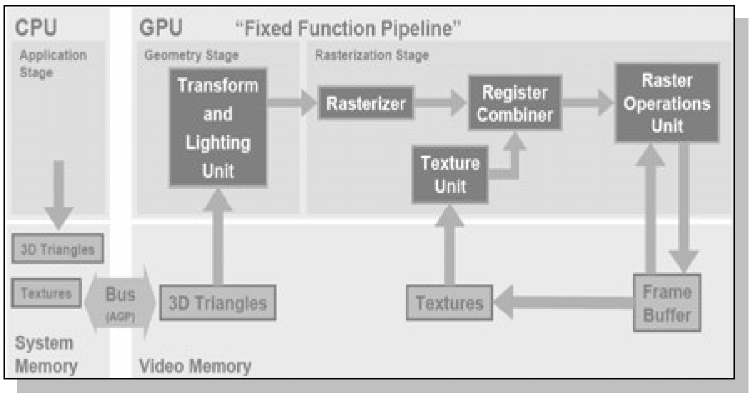


Figure 1: The Fixed Function Pipeline [13]

The fixed function pipeline supports various forms of lighting usually including Diffuse and Specular lighting models along with Flat and Gouraud shading. Fixed function lighting, under DirectX, is discussed further by Dempski, 2002 [15]. The fixed function pipeline also supports basic texture mapping, as well as various features that go hand in hand with texture mapping; including MIP maps and texture blending. Fixed function texturing, under DirectX, is covered in greater depth by Dempski, 2002 [8].

These are all quite efficient at their allotted tasks due to the fact they are designed and optimised specifically for these operations. However it must be remembered that this comes at a cost, they are simply set tasks and pre-defined routines. Outside the realms of 'normal' applications very little in the way of customisation can be done. This makes the fixed function pipeline a very inflexible solution given current demands for varied

6 Rasterisation[14] is the process of taking an image described in a vector graphics format (shapes) and converting it into a raster image (pixels or dots) for output on a video display or printer.

and complex effects asserted by the activities of the games and simulation industries, such as those outlined by the GPU Gems series of books [16].

2.2 The Programmable Pipeline

The programmable pipeline is a relatively new development and is designed to replace the old style fixed function pipeline by providing greater flexibility and programmability through new programmable technology known as shaders. Multiple shader programs can be combined within the programmable pipeline to achieve various effects and perform many different functions. Figure 2 demonstrates how the fixed function and programmable pipelines fit within the whole graphics architecture as defined by Microsoft [17]. As can be deduced from Figure 2 one pipeline can be used as a direct replacement for the other, but the two pipelines cannot be used wholly in conjunction with each other.

The programmable pipeline consists mainly of Vertex and Fragment[7] shaders, and is a direct result of a panel discussion about the future of graphics hardware undertaken as part of SIGRAPH '99 [8]. Until the advent of the programmable pipeline consumer level graphics hardware was restricted to a fixed function-based system. As previously described this meant that the only options available to games and simulation developers were those hard-wired into the graphics hardware by the hardware vendors. This also had the disadvantage that hardware from different vendors could have a completely different set of options and even provide different functionality given the same options

7 Also known as pixel shaders.

and API calls. Obviously this restricted the functions available to whatever the hardware manufacturers and API maintainers deemed necessary and allowed very little freedom for the games and simulation developers.

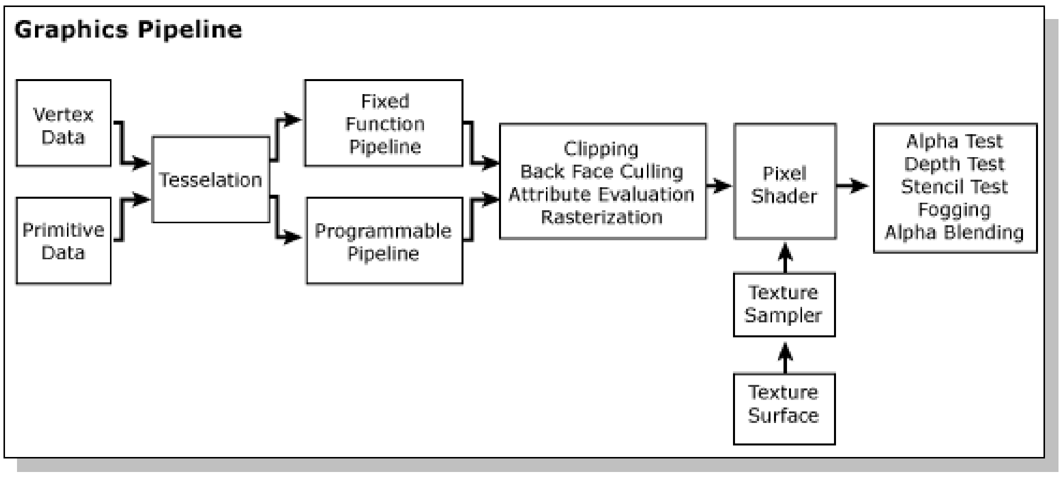


Figure 2: Graphics Pipeline [17]

Shaders present the programmer with greater flexibility by enabling them to define vertex and/or fragment shader programs which can be run on any supporting hardware. For example, they allow the programmer to replace the old style fixed function lighting with shader programs. So where previously the only options available may have been, for example, Gouraud shading [9], the programmer now has the option to create their own effects such as anisotropic lighting [10] or cell shading [11]. Shaders essentially replace the fixed function pipeline of a standard, pre-Geforce3, graphics card by providing programmability. This can be demonstrated by comparing Figure 1 [13] to Figure 2 [17]. However, fixed functions can still be used in conjunction with the majority of, but not all, shader generations. Shader Model 3, for example, cannot be used in conjunction with any fixed function elements [18].

NVIDIA were the first to develop this new direction in graphics on consumer level hardware. This was in the form of the GeForce3 [19]; which supported shader model 1.1 (pixel shader 1.1 and vertex shader 1.1). While this provided a much greater level of control to the programmer than ever before possible on consumer level graphics hardware it did cause the problem of increased development times. This was due mainly to the fact that until relatively recently (the introduction of the first HLSL[8] in 2003 [20]: this is discussed in detail in section 2.2.2) shaders had to be coded directly in shader assembly language. Thus limiting the scope of possible applications due to the specialised and complex nature shader programs and their assembly like language. In turn, this contributed to long development times and other complications inherent with assembly language programming in general. Considered with the faster than Moore's law progression of GPU technology [21] this meant developers simply could not keep pace with the rapid development of the technology. This posed quite a problem, particularly when the difficulties inherent with assembly language programming are compounded by the complexity of many graphical algorithms; yet these shaders were designed to encourage realism on consumer level hardware.

In an attempt to alleviate this problem NVIDIA, Microsoft and SGI[9] all developed their own High Level Shader Languages; Cg, MSHLSL, and GLSL respectively. These HLSL's aim to have the same impact on Shader Assembly language that C and other high-level languages had in replacing assembler as the development tool of choice. HLSL's are designed to make it easier to map algorithms into code, as they provide a much more intuitive way for a human programmer to view the operations of a shader.

8 **H**igh **L**evel **S**hader **L**anguage
9 **S**ilicon **G**raphics, **I**nc.

2.2.1 Shader Model 3

Shader model 3 is the current standard and provides many benefits over previous incarnations, including but not exclusive to more available instructions and greater floating point precision[10].

Shader Model 3 has been required by DirectX since version 9c, and as such any card wishing to claim DirectX compliance will have to support all the features defined in the Shader Model 3 standard. Previous versions of DirectX 9 only required Shader Model 2. NVIDIA [22] provides some comparisons between Vertex Shader 2, Vertex Shader 2a and Vertex Shader 3 as well as providing comparisons between Pixel Shader 2, Pixel Shader 2a, Pixel Shader 2b[11] and Pixel Shader 3.

2.2.2 High Level Shader Languages

Cg (**C** for **g**raphics) is NVIDIA's high level shader language and was developed in partnership with Microsoft; this partnership also produced Microsoft high level shader language (MSHLSL[12]). The two languages are virtually identical; Cg is actually a superset of MSHLSL [23]. The major difference between Cg and MSHLSL is the fact that the MSHLSL is DirectX specific, whereas Cg will happily operate within both DirectX and OpenGL. Both languages are based on C and incorporate some elements of C++, conversely due to the constraints inherent with shader programming much of the

10 See [22] and [18] for a more comprehensive look at shader models.
11 Note there is no Vertex Shader 2b.
12 Although Microsoft simply call it HLSL, it shall be referred to as MSHLSL to avoid confusion.

functionality of C has been lost.

Since OpenGL 1.5 there has been another high level shader language available for use with OpenGL. GLSL[13] was included as part of OpenGL 1.5 in the form of official ARB extensions, where previously it had only been available as unofficial extensions. GLSL is OpenGL's answer to MSHLSL, it is also based on ANSI C and again incorporates elements from C++.

Cg was the first high level shader language to become widely available to developers and was introduced to work with DirectX 8.1 and OpenGL 1.3, Cg is now many iterations on and along with HLSL and GLSL is beginning to reach a greater level of stability in its feature set.

A sample of Cg code and its assembly language equivalent extracted from a Phong shader [25] are shown in Figure 4. As can be seen in this example code, two lines of Cg equates to twenty-five lines of shader assembly. This illustrates the huge difference in development and maintenance time that can be made by switching to a HLSL.

13 Open**GL** **S**hader **L**anguage. See [24] for more detail on GLSL.

Cg

```
…
float3 cSpec = pow(max(0, dot(Nf, H)), phongExp).xxx;
float3 cPlastic = Cd * (cAmbi + cDiff) + Cs * cSpec;
…
```

Assembly

```
…
RSQR R0.x, R0.x;
MULR R0.xyz, R0.xxxx, R4.xyzz;
MOVR R5.xyz, - R0.xyzz;
MOVR R3.xyz, - R3.xyzz;
DP3R R3.x, R0.xyzz, R3.xyzz;
SLTR R4.x, R3.x, {0.000000}.x;
ADDR R3.x, {1.000000}.x, - R4.x;
MULR R3.xyz, R3.xxxx, R5.xyzz;
MULR R0.xyz, R0.xyzz, R4.xxxx;
ADDR R0.xyz, R0.xyzz, R3.xyzz;
DP3R R1.x, R0.xyzz, R1.xyzz;
MAXR R1.x, {0.000000}.x, R1.x;
LG2R R1.x, R1.x;
MULR R1.x, {10.000000}.x, R1.x;
EX2R R1.x, R1.x;
MOVR R1.xyz, R1.xxxx;
MULR R1.xyz, {0.900000, 0.800000, 1.000000}.xyzz,
R1.xyzz;
DP3R R0.x, R0.xyzz, R2.xyzz;
MAXR R0.x, {0.000000}.x, R0.x;
MOVR R0.xyz, R0.xxxx;
ADDR R0.xyz, {0.100000, 0.100000, 0.100000}.xyzz,
R0.xyzz;
MULR R0.xyz, {1.000000, 0.800000,
0.800000}.xyzz, R0.xyzz;
ADDR R1.xyz, R0.xyzz, R1.xyzz;
…
```

Figure 4: Example of the simplifications possible when using a HLSL, such as Cg, compared to Shader Assembler [25]

2.2.2.1 Alternatives

There are alternatives to shader assembly other than programming directly in a HLSL: for example, ATI's RenderMonkey. This is essentially an IDE[14] designed specifically for shaders, it provides methods to edit a shader and view the results within the same workspace. RenderMonkey also provides many artist tools which can minimise the amount of coding required.

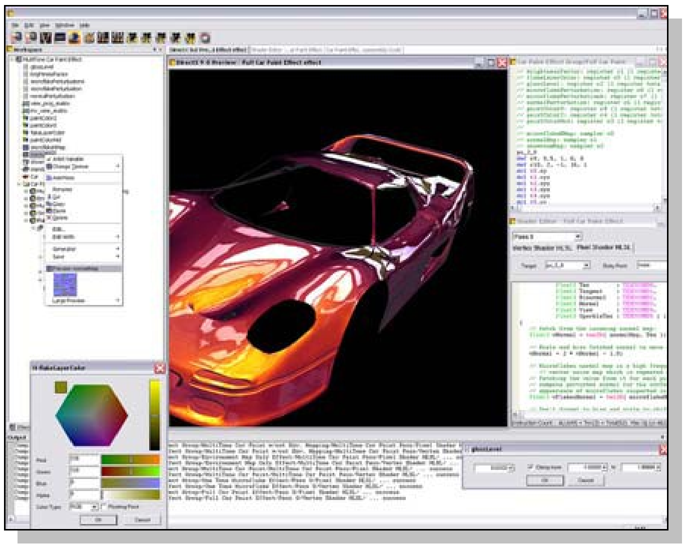


Figure 5: RenderMonkey

Originally developed by 3D Labs and then taken over by ATI, RenderMonkey has been designed to enable both artists and programmers to create and tweak shaders relatively easily; Figure 5 provides an example of a typical RenderMonkey workspace.

While alternatives are now available, RenderMonkey deserves a special mention as it was the first IDE of its type and was ATI's answer to Cg. Where Cg was still simply a

14 **I**ntegrated **D**evelopment **E**nvironment

shader language, RenderMonkey provided an IDE to encompass the whole shader development process as well as a graphical interface onto existing API's.

In its current incarnation, RenderMonkey supports both MSHLSL and GLSL but does not claim to be Cg compatible. This should come as no surprise given that ATI's and NVIDIA's rivalry is very well publicised. As such ATI are concerned about the possibility of Cg becoming a standard, as this would put a significant amount of power in the hands of its rival. This would be a particular problem for ATI given that versions of the Cg compiler have been shown by ATI [26] to perform poorly in comparison to MSHLSL on ATI based systems; when exactly the same code on an NVIDIA system is much more reliable.

After ATI's acquisition of RenderMonkey, NVIDIA produced a response in the form of the NVIDIA FX Composer[15], unfortunately this is specific to MSHLSL development at the time of writing. FX Composer currently provides no support for Cg, which may come as a surprise to many and can perhaps be seen as an indicator of NVIDIA's future direction in HLSL support.

2.2.2.2 Advantages and Disadvantages

All the HLSL's discussed like to claim a certain amount of platform independence. For example MSHLSL should have no problems running on either NVIDIA, ATI or anyone else's hardware (within Microsoft Windows) and Cg will

15 See [27] for more detail on FX Composer.

happily compile up to run on Windows or Linux based systems due to its compatibility with OpenGL and DirectX.

Surely all this so called ‘platform independence’ can only be a good thing? Should it not simply do what it has claimed and allow the same code to run on virtually any hardware you care to throw at it and drastically reduce potential development times?

In reality the answer to the above question is not quite as simple as it may appear at first. HLSL’s hide a large number of the complexities associated with shaders. Fragment shaders in particular carry a lot of restrictions that are not inherently obvious while coding in a HLSL; restrictions that could easily be picked up on while working in shader assembly language. For example, Cg’s loop statements allow the programmer to write code which will be executed a constant number of times on the hardware. This can cause unforeseen results; as in the case of DirectX 8 Vertex Shaders (Vertex Shader 1.1) which do not allow looping. In this case when the code is compiled the Cg compiler unrolls the loop so that the final shader program contains the same set of instructions repeated as many times the loop specified [28]. This means that a loop which contains the equivalent of 2 assembly level instructions, repeated 10 times in Cg will balloon to 20 instructions in the compiled shader assembly. In this manner it would be very easy to exceed the hardware platform's instruction limits (or worse) without the programmer realising, at least until the code is compiled, potentially wasting valuable development time.

Debugging is another interesting topic within the world of HLSL's. Microsoft provide debugging support for MSHLSL in the form of an add-on for Microsoft Visual Studio. Both ATI's RenderMonkey and NVIDIA's FX Composer also provide forms of debug support for MSHLSL including, for example, a disassembler within RenderMonkey and a jump to error feature in FX Composer. Both also provide the advantage of being able to see the results of shader changes instantly within the IDE unlike within more traditional environments such as Microsoft's Visual Studio. For a further discussion of shader related issues including debugging see Rhodes et al. [12][29].

If used carefully alongside the debugging tools available, HLSL's can make the whole process of developing shaders much easier and quicker. HLSL's seem to be the future of shader development and, as intended, have begun to replace shader assembly as the development language of choice for shaders.

2.3 Life at the Bleeding Edge

Unfortunately, using the latest features of newly developed hardware and software comes at a price: unspecified hardware features, unstable drivers and unsupported features are just some of the pitfalls which developers face when working at this level. These issues are often due to the developers unavoidable dependency on the relatively fluid drivers and compilers supplied by the hardware vendors. As discussed by Rhodes et al. [29] this also impacts on teaching methods, objectives and outcomes.

Discrepancies between driver versions are often a particularly difficult problem, as even slight changes between versions can have drastic knock on effects. Especially for finely tuned programs at the level dealt with by the techniques presented here. It can often become impossible to detect whether issues are caused by problems within the code itself, by a hardware feature, or simply by a quirk of the current driver version.

This type of issue has cropped up many times during the undertaking of this research. However, given enough time it is usually possible to work around these problems. Though it is possible to work around such issues, it is rarely possible to ascertain whether an issue is caused by a driver related problem or is an error in the code; unless the issue is subsequently resolved by the vendors. This obviously necessitates the removal of the appropriate work around, which in itself may not be an easy task to isolate and correct due to the nature of not knowing whether the fix was applied to accommodate a driver fault or a genuine error.

One example of an error that was identifiable as a proven driver problem was the failure of the Cg compiler in some versions of the NVIDIA drivers. Cg compiler version 1.3.0001 often failed to create appropriate ASM from a GLSL source, particularly when loops are employed. Unfortunately, the effect this error had on the visual output was severe and noticeably incorrect, thus causing a painstaking re-evaluation of the source material to be carried out before the error was eventually identified as being driver related. This issue is demonstrated by Figure 6 and Figure 7. The examples illustrate the problem by showing the first five lines of output from the same GLSL source, as compiled by two different compiler versions.

```
TEX   R0.xz, c[0].z, texture[0], 1D;
MOVR  R2.w, R0.x;
MULR  R0.x, fragment.texcoord[0], c[0];
MOVR  R3.x, R0.z;
FLRR  R2.x, R0;
```

Figure 6: Cg Compiler Version 1.3.0001

Figure 7 shows the correct result provided by compiler version 1.5, while Figure 6 in comparison demonstrates the missing assembly and misinterpreted instructions which plagued compiler version 1.3. While such errors appear obvious when the correct version is shown next to the incorrect, these issues are in fact rather difficult to diagnose when faced with just the original incorrect output. Often the issues facing the developer are much more subtle than those depicted by the examples chosen for Figure 6 and Figure 7.

```
TEX   R1, c[1].w, texture[0], 1D;
TEX   R0, c[0], texture[0], 1D;
DDXR  R2.zw, fragment.texcoord[0].xyxy;
MULR  R3.xy, R2.zwzw, c[0].y;
DDYR  R2.zw, fragment.texcoord[0].xyxy;
```

Figure 7: Cg Compiler Version 1.5

Other examples of notable issues includes those of Forceware release 70 which suffered from large memory leaks, and inaccurate texture co-ordinates in some Forceware versions prior to 101.34. In some cases processing shader code under Forceware 70 causes over 2 gigabytes of page file usage from an original source file of well under 1 megabyte when compiling and linking Shader model 3 based GLSL code. The texture co-ordinate issue required minor adjustments, in the region of 0.00001%, to be made in order to correct the inaccuracy of texture co-ordinate calculations. This problem also has the effect of disguising similar hardware limited accuracy issues which affect the NVIDIA GeForce 6 and 7 series.

2.3.1 Precision

The main problem encountered was that of calculation precision. For example, while NVIDIA advertise the 6800 platform as supporting full 32-bit precision, the test results presented below show that this cannot be the case. The "full floating point accuracy" advertised should theoretically provide the same results as the same calculations run on a CPU and processed in floating point precision to ensure a fair comparison. From the results it is possible to show that what NVIDIA actually deliver is 32-bit calculations and almost 32-bit floating point precision.

This is likely due to the fact that modern CPUs, such as AMD's[16] Athlon series or Intel's Pentium series, use 80-bit extended precision for all floating point calculations. These extended precision calculations are subsequently rounded to give the final result to whatever precision is requested. The NVIDIA platform on the other hand seemingly

16 **A**dvanced **M**icro **D**evices

does not have this 80-bit extended precision facility. As a consequence the 6800 is slightly less accurate than the average CPU for the same calculations. It is proposed that this is due to the 6800 performing calculations directly in 32-bit floating point precision, and hence some accuracy is lost. See LaMothe 2003 [30] for an in depth discussion of modern CPU FPUs[17].

Consequently, this could prove a problem for the sine and cosine calculations required for many of the more advanced graphical techniques, which may otherwise be suitable for implementation in shaders. Such techniques can be quite sensitive to accuracy problems. Particularly when repeating the same calculations in loops, meaning any errors are compounded.

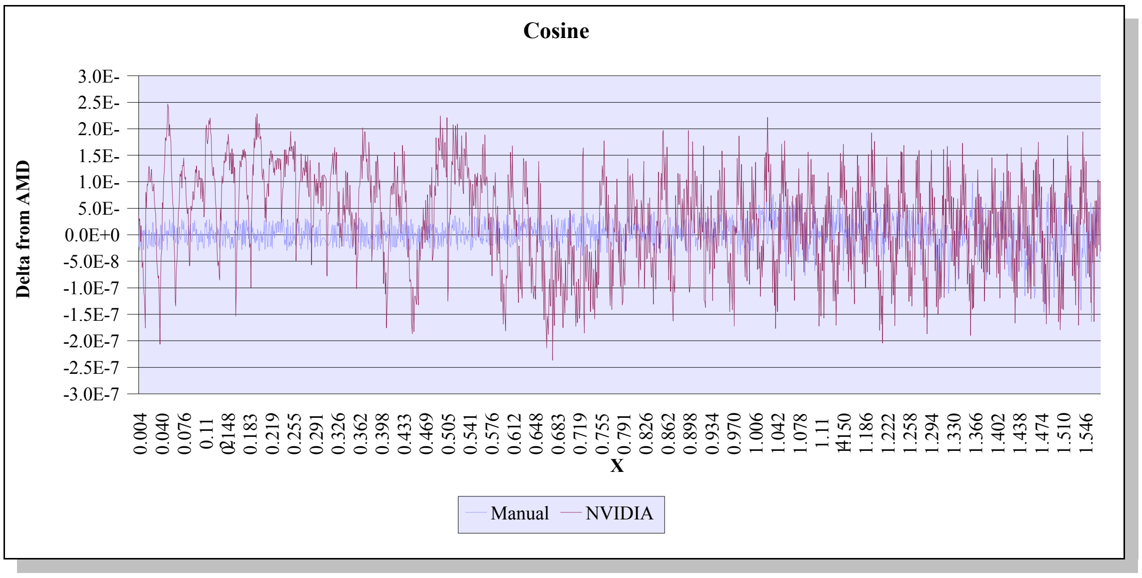


Figure 8: Cosine Inaccuracy

As such it became necessary to run tests to clarify the exact situation with regards to comparative accuracy. A series of tests were run, in both software and hardware, to

17 **F**loating-**P**oint **U**nit

determine exactly how accurate the GPU calculations are. Figure 8 and Figure 9 show the results for this on the test system, consisting of an AMD Athlon CPU and an NVIDIA 6 series GPU.

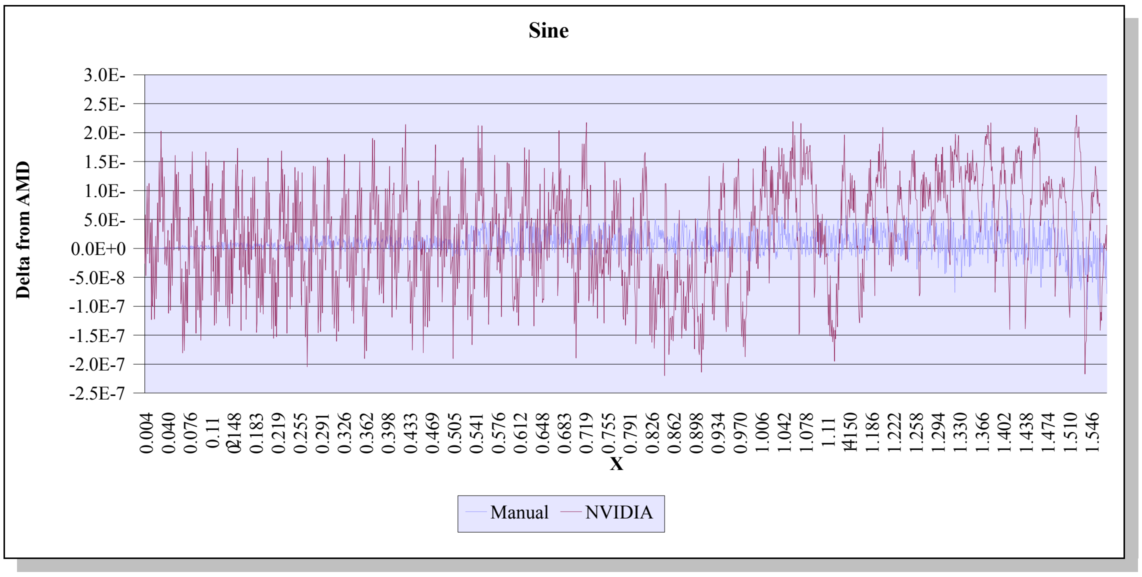


Figure 9: Sine Inaccuracy

Both sine and cosine calculations were performed in software on the AMD platform to determine values which could be used to compare floating point precision against the NVIDIA platform. It was concluded that the AMD produced values to be accurate enough for this task after close comparison with the values provided by Abramowitz and Stegun [31] which are calculated to a precision of twenty three decimal places, and hence are more accurate than most techniques would require. The AMD values are proven to be very accurate for the 32-bit floating point calculations considered. When compared against Abramowitz and Stegun [31] and considered as double precision floating point calculations, they only deviated occasionally on the final decimal place; much more accurate than the NVIDIA hardware allows.

The AMD results for the sine and cosine functions were then compared to the NVIDIA sine and cosine functions as well as a manual method implemented in shaders. Figure 8 and Figure 9 give the deltas of both the NVIDIA and manual shader methods from the AMD values. As can be seen from the graphs the manual method proved more consistently accurate for any given value of x, despite using the same 32-bit floating point hardware.

It can be concluded from this that NVIDIA do not calculate their sine and cosine functions to the full 32-bit floating point precision available. However, it can be conjectured that NVIDIA use look-up tables to improve calculation performance at the expense of this accuracy. This is hinted at by the fact that that the manual version consistently proved more accurate than the built in hardware method when running under the same hardware limitations.

Chapter 3: Depth of Field

The ongoing quest for realism requires increasingly advanced techniques in real-time computer graphics systems, the majority of which aim to simulate or closely approximate real life optical systems. One such optical effect is the depth of field phenomenon, which has yet to be modelled with any great degree of accuracy in real-time computer graphics.

The depth of field phenomenon is an optical effect which is best described in terms of the human eye. When focusing on an object, the rays of light from that object are refracted by the eye's lens directly onto the retina. This provides a sharp in-focus image. Because of the way the eyes lens does this any rays of light from objects sufficiently far away from the in focus plain will be focused either just in-front of or just behind the retina. In turn, this causes blur circles to form of a size proportional to the source 's distance from the point of focus. This is why the further away an object is from the point of focus the more blurred it becomes. Rokita [32] provides a more in-depth look at the physics behind the depth of field phenomenon.

Most computer graphics systems take the easy route out and ignore the depth of field phenomenon altogether, opting instead to use the pinhole camera model, which produces completely sharp images. By ignoring depth of field these systems are limiting how realistic they can be. Without a good approximation of depth of field any computer graphics system, no matter how advanced, will look synthetic. Humans are used to seeing this effect all the time. As the world around us, viewed through our eyes, suffers from depth of field effects. If realistic images are to be obtained in computer graphics, all aspects of the eye need to be taken into account. Even those which may be described as a deficiency of the natural optical system, like depth of field.

Systems that choose to ignore depth of field are starving themselves of an excellent method for providing depth cues and diverting the viewers' attention to areas of importance. Cinematographers, and hence Hollywood films, have utilised the depth of field phenomenon to great effect over the years [33]. It is used to divert the viewers' attention to important aspects of the scene. If done properly these subtle cues are not picked up consciously, and hence provide a very powerful special effect in the Hollywood arsenal.

Possible uses for such an implementation of depth of field include military simulators where depth cues are vitally important. One example of such military uses are flight simulators, where the depth of field effect could provide the pilot with essential information about things like target distance and size. Over the long term this would potentially provide much more realistic simulations and hence better training conditions. Naturally, this would have the effect of lessening the jump between simulator and real life, which would provide numerous benefits. Particularly from a military perspective as complex training scenarios could be worked on from the safety of the base.

Current attempts at implementing real-time depth of field effects suffer from a number of issues. The major problem with most current real-time solutions is a lack of support for the see-through effect. This is the phenomenon of being able to see a sharp object when viewed through a de-focused object. For example, NVIDIA [34] and ATI [35] both provide solutions that do not support the see-through effect. Both these solutions are based around a method similar to that proposed by Potmesil and Chakravarty in 1981 [36].

The lack of support for the see-through effect causes a visible aura around de-focused objects, which results in an unnatural look. In many cases this situation is potentially worse than having no depth of field effect at all, as objects that should be visible become obscured.

Another popular solution is to use multiple discrete samples or multi-sampling, which is discussed in detail in section 3.2.1.1. Microsoft [37] provides an example of one solution which takes this route. The multisampling method has the advantage of supporting the see-through effect; however multisampling throws up several other issues. The most noticeable problem with most multisampling based depth of field techniques is the multi image effect. This occurs when the number of samples is too small, and results in a fuzzy appearance of images rather than actual blurring. Often with multiple images being distinctly visible for lower sample numbers. Unfortunately the only current solution which can provide enough samples to make this unnoticeable is ray-tracing; but as this requires the sample points to be varied per-pixel it is not yet a viable real-time effect.

3.1 What is Depth of Field?

The depth of field phenomena is something experienced, perhaps unknowingly on a conscious level, by everyone on a daily basis. When the eye focuses on a particular object the area and objects around it are perceived as increasingly more blurred the further they are away from the point of focus. This depth of field effect also applies to other lens based optical systems such as photography.

The actual range in which the eye can see completely sharply is relatively small. The eye is rather poor at distinguishing the detail of objects which are not directly on its focal plane. The majority of what the eye sees is non-sharp and a large amount of image enhancement is performed by the brain to get to the final image seen. For an example of how much work the brain does after the eye sends its images for processing examine the human eyes' blind spots, which few people realise exist until shown. Each eye has a blind spot where the optic nerve meets the back of the eyeball. The brain fills in these blind spots utilising image data from the surrounding area. For a practical example see [38]. This emphasises the point that the eye can only see clearly directly in its' plane of view, hence all the surrounding objects will appear blurred. Figure 10 gives an example of Depth of Field in an optical system, the see-through effect is evident and its' impact

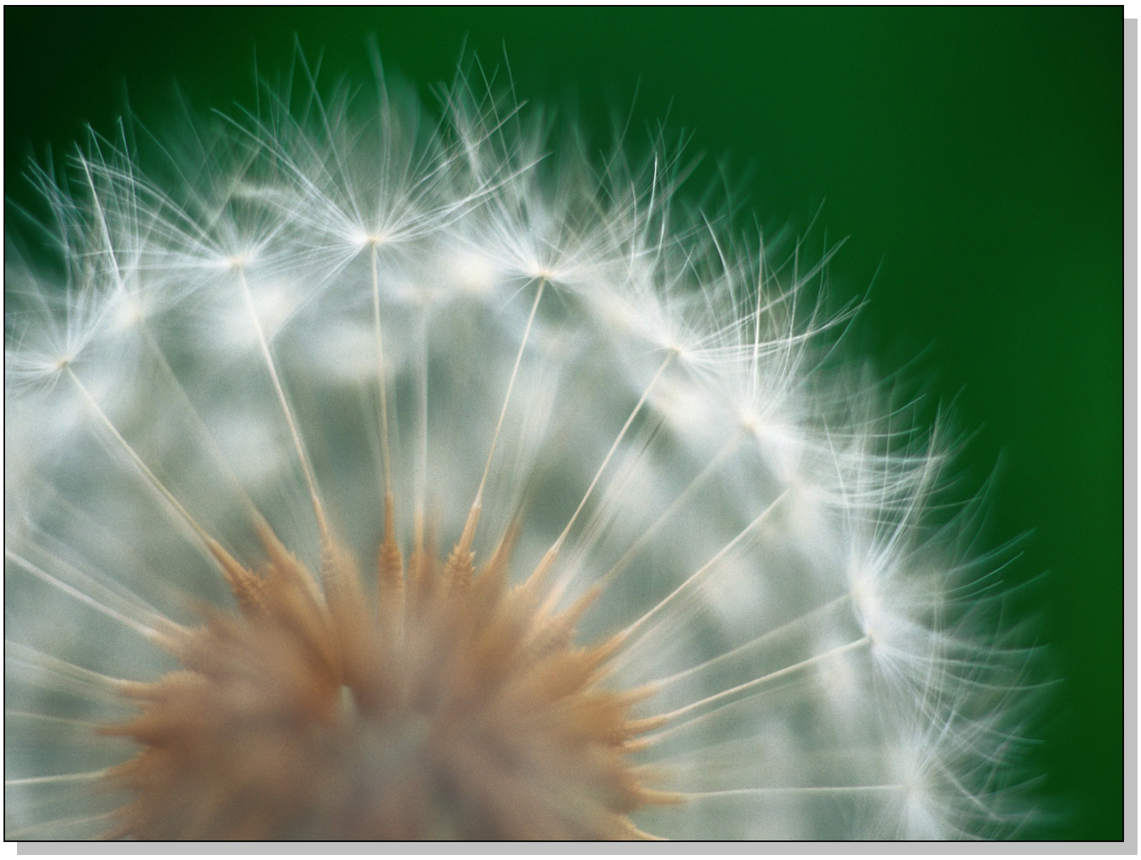


Figure 10: Depth of Field in an optical system

on the final image is plain to see. The in focus plane is clearly distinguishable as the sharp area of the image, with the areas to the foreground and background being increasingly blurred as the distance from the focal plain increases. In this particular example the in focus plain is still visible, due to the see-through effect, despite being potentially obscured by the area of the plant which precedes it in the image.

As observed by Rokita [32], Depth of Field is a direct result of the process of accommodation; which is one of many depth cues that influence the way humans perceive their surroundings. Figure 11 shows a visual example of how the depth of field effect is created. The sharp image is created where the rays of light focus directly onto the photoreceptor; this could be the film (in the case of photography) or the retina (in the case of the human eye). In the eventuality that the rays of light focus in front of or behind the photoreceptors (i.e. the image is de-focused) blur circles are created. Blur circles are best explained by taking into account a single point of light as in Figure 11.

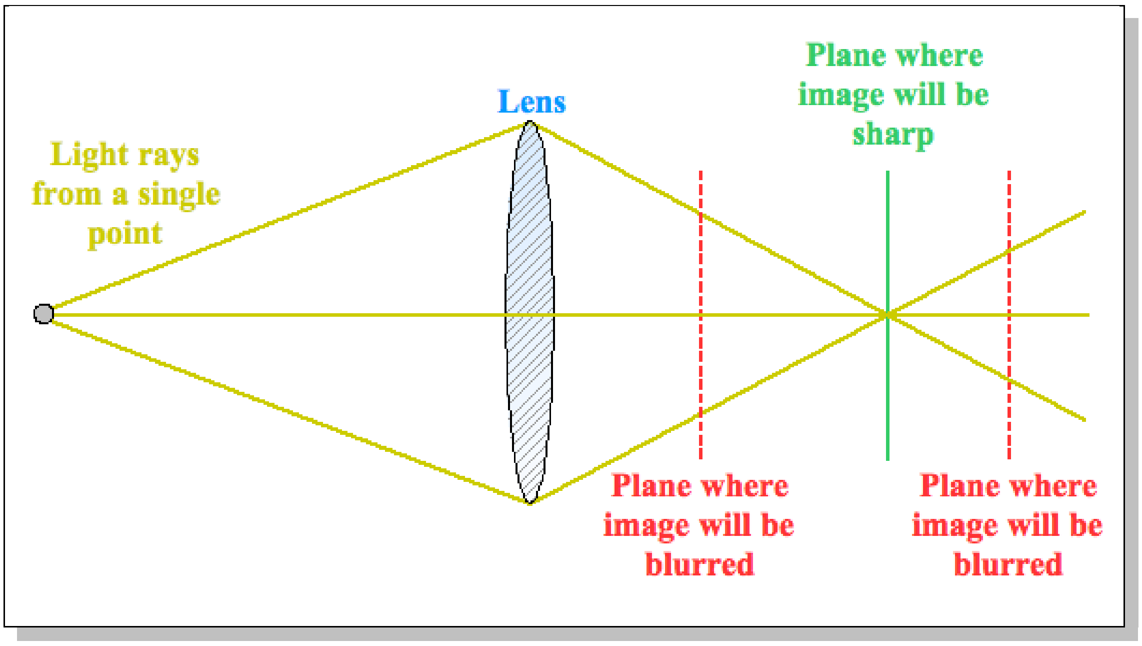

Light rays from a single point
Lens
Plane where image will be sharp
Plane where image will be blurred
Plane where image will be blurred

Figure 11: How the depth of field effect occurs

Taking Figure 11 as an example; if the rays from the light source are in focus then they will converge on the view plane to create a sharp image. For example, in the eye the in-focus rays will converge on the retina. However, if they are not in focus then the rays will converge either in front of or behind the sharp image plane. In the case of the eye, this would mean the defocused light rays are spread over an area which is dependant on the source's distance from the focal plane. The brain fills in any missing information from the de-focused light, which creates a blur circle. Obviously in the real world there is much more than a single point light source, so many thousands of blur circles will appear in any one particular optical image.

This effect is taken advantage of extensively in the worlds of film and photography, for example films such as ‘The Lord of the Rings’ make heavy use of Depth of Field along with various other techniques to draw the viewers’ attention to the important aspects of a scene. This is not a new technique in cinematography however, Depth of Field effects have been utilised over the years by many different directors ranging from Orson Welles to Peter Jackson. Obviously this could also be used to similar effect in areas such as computer games and simulations. However one of the main uses of the depth of field phenomenon for computer graphics images would be to add realism to games and Virtual Reality systems such as flight training simulators.

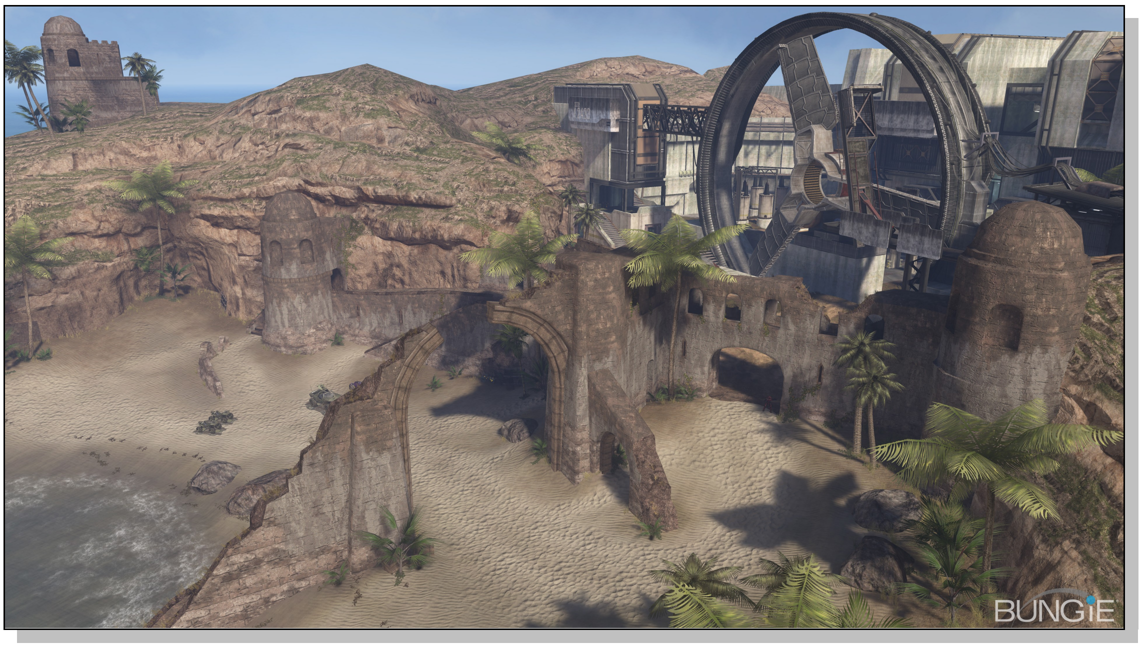


Figure 12: An example of a computer generated image without depth of field

The problem with most current computer rendered images is that they are based on the pin-hole camera model. This means that each image is potentially infinitely sharp, obviously taking into account practical limitations. This is demonstrated by the screenshot shown in Figure 12, taken from the game Halo 3 [39], which does not use any depth of field effects in game [40]. In Figure 12 it can be reasonably assumed that the extreme rear of the image should at least be partially blurred. This is becoming more of an issue as texture detail and draw distances improve, as this only adds to the unnatural feel of a scene when depth of field effects are absent. So some method is needed to create more realistic computer generated images by inclusion of a depth of field effect.

3.2 Existing Solutions

Currently there are a number of solutions to this problem all of which have shortcomings, these methods are discussed by Rhodes et al. [4] and expanded upon in this section. A summary of existing solutions can also be found in GPU Gems [41].

3.2.1 Blurring by Multiple Viewpoints

This is by far the simplest approach to solving the depth of field problem. In addition it is currently the most successful solution for real-time applications, providing moderate results and supporting the see-through effect. The effect is achieved by creating the image from multiple discrete viewpoints as demonstrated in Figure 13. The image from each viewpoint is added to an accumulation buffer where the final image is built up.

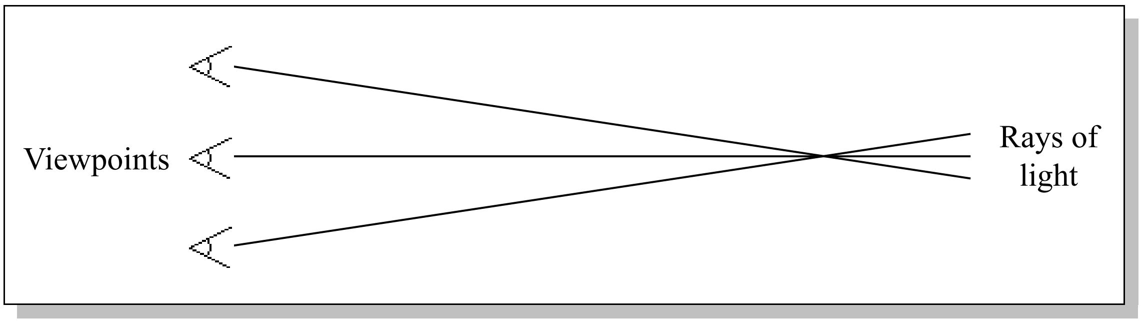


Figure 13: Sampling multiple viewpoints

3.2.1.1 Blurring by Multisampling

The multisampling method suffers a major flaw, the unwanted effect of multiple images being clearly distinguishable. As demonstrated by Figure 14 this is due to the fact that not enough samples have been used. The simple solution to this is to increase the number of samples. However, increasing the level of blurring slightly drastically increases the number of samples that are needed to help disguise this artefact. The number of samples required even for a small level of blurring is potentially very large, particularly with more complex images. Rendering the same scene multiple times is also potentially very time consuming, obviously dependant on the complexity of the scene, so a large number of samples will lead to a big performance hit in your application.

There is another problem with this method that is apparent from Figure 14; particularly when compared with Figure 15. It can be seen that the effect created via multiple images is not actually blurring of the image and is perhaps better described as adding fuzziness to the image.

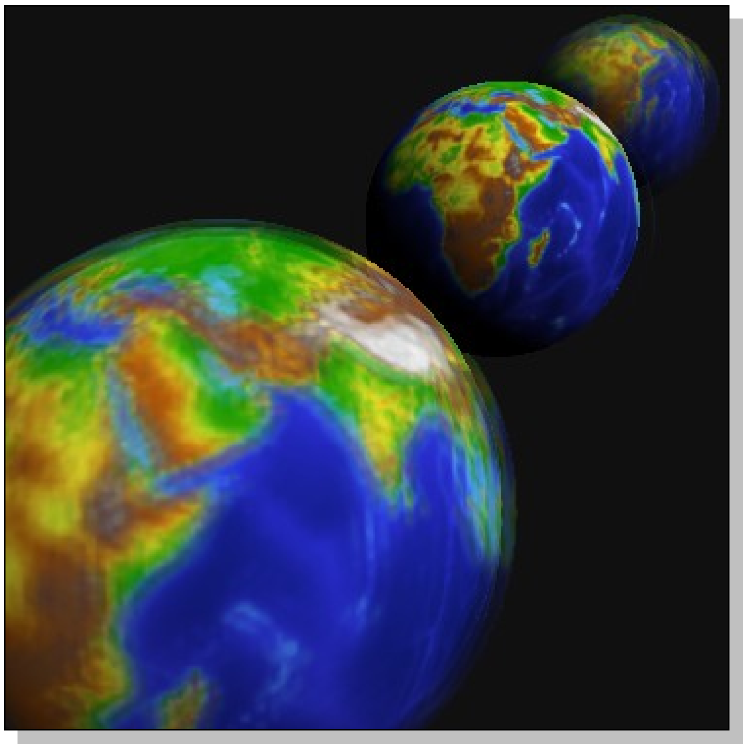


Figure 14: Depth of field by multi-sampling [37]

3.2.1.2 Ray Tracing

Blurring by ray tracing is a form of blurring by multiple viewpoints; the major difference is that ray tracing allows varying of the sample points pixel by pixel. The most realistic depth of field effects can be obtained via ray tracing techniques, where rays of light are traced from the viewpoint to the light source. The calculations involved in ray-tracing make use of the physical properties of light. Figure 15 provides an example of a ray traced image taken from the Pixar film Ratatouille.

Pixar, via their 3D rendering software RenderMan, have been using ray tracing since 2002 [42] and the first feature film made using the technique throughout was Cars [43]. Ray tracing does produce correct results. However, with current technology it is not possible to perform such complex calculations in real-time at the kinds of resolutions

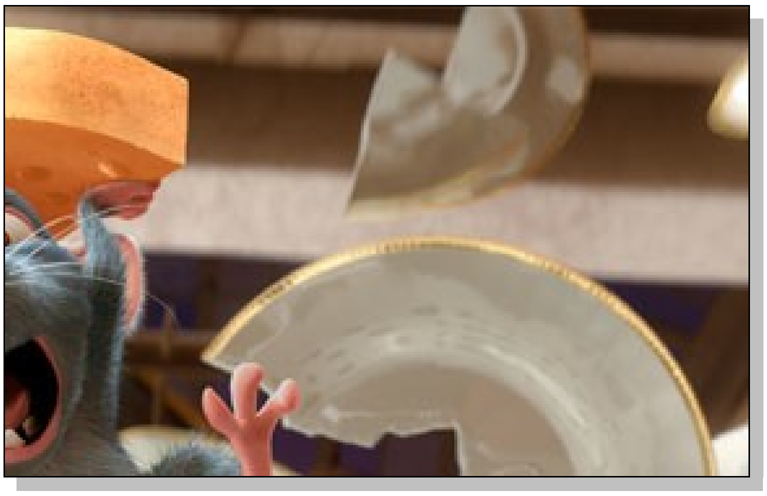


Figure 15: Depth of field by ray tracing

required for games and simulations. As such ray tracing is only of use for pre-processed scenes such as the one shown in Figure 15.

3.2.2 Blurring Dependant on Depth

The basic premise of these systems is to create a blurring effect which is dependant on depth, where the depth is usually taken to be the value of z or $1/z$. Examples of this type of system include work by Snyder and Lengyel. [44], Rokita [32], and Potmesil and Chakravarty [36].

A variation of the Potmesil and Chakravarty method has been implemented by NVIDIA [34], examples of which can be seen in Figures 16 and 17.

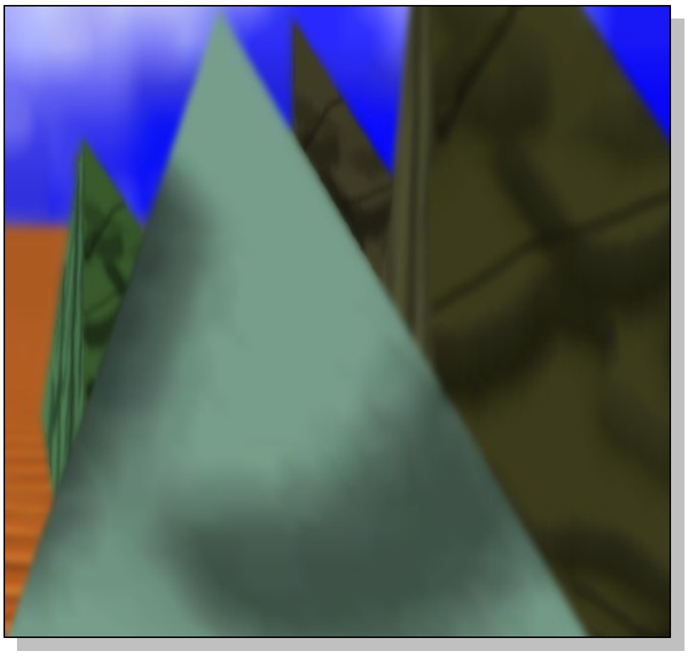


Figure 16: NVIDIA Artefacts [34]

There are several problems with the method employed by NVIDIA. For example, some of the objects within the scene appear to have an aura surrounding them; this is because of the lack of support for the see-through effect. This is a common problem which plagues the Rokita [32], Potmesil and Chakravarty [36], and similar, methods.

The aura artefact can be seen more clearly in ATI's Depth of Field demonstration [35], which is presented in Figure 18 with the focus plane on the chequered wall. This uses virtually the same method as the NVIDIA example, with the exception that pixel shader version 2.0 is preferred over the version 1.1 used by NVIDIA. This means that a much higher level of blurring is achieved due to the extra instructions and resisters available.

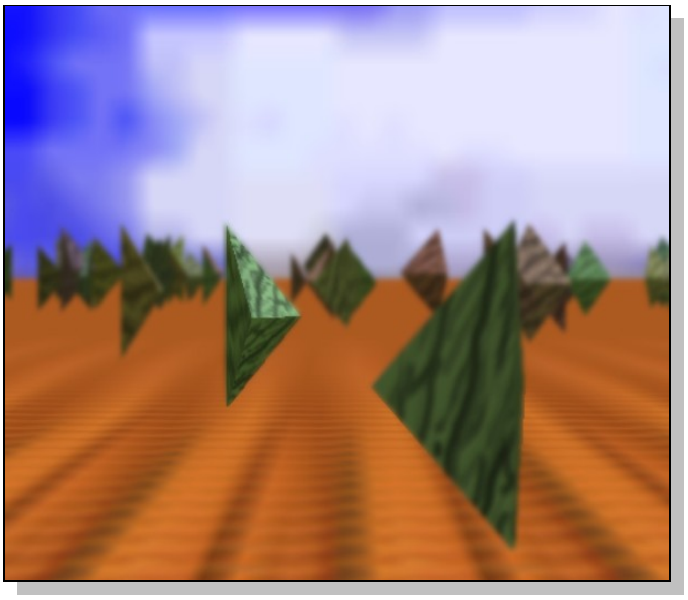


Figure 17: Depth of field by depth and image processing [34]

Snyder and Lengyel [44] get around this problem via the use of a layering system. Assuming that the objects in question truly are on separate layers the see-through effect is supported. However, Snyder and Lengyel's system does have two major drawbacks.

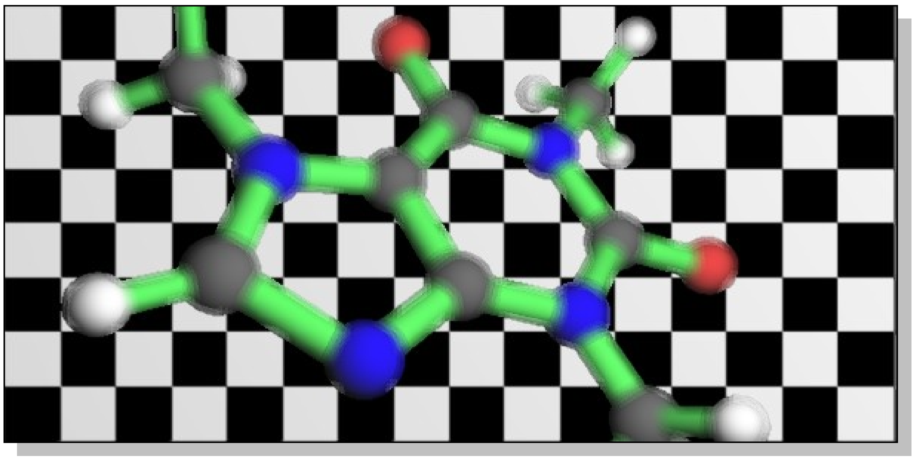


Figure 18: ATI artefacts [35]

Firstly the choice of which objects belong on each layer depends on hidden surface removal considerations rather than depth. Hence correct ordering for use with depth of field cannot be guaranteed. Also in order to be able to use Snyder and Lengyel's system a non-standard method for hidden surface removal must be used. This causes a huge problem as it would require an entirely new type of graphics rendering system and is simply not practical for such a specialised requirement.

3.3 New Depth of Field Implementation

A novel depth of field algorithm is presented, based on consumer level hardware. The technique is designed to run in real time but without many of the major problems identified previously. This provides an extension of previous work by Rhodes et al. [3][2] which was inspired by the work of Chia et al. [45].

The first important aspect is the inclusion of a layering system similar to that proposed by Snyder and Lengyal [44]. However in this case the layers are determined directly by depth and not by hidden surface removal considerations used by Snyder and Lengyal. This method guarantees that two objects rendered on the same level will have a similar level of blurring, which was not the case with Snyder and Lengyal's system.

Each pixel within the system consists of x, y and z values (assumed to be in the form $1/z$) along with their associated colour values as per a standard z-buffer based system. However, unlike a normal z-buffer based system; more than just the winning pixel contributions from the depth test must be retained. This is because these values are necessary if the 'see-through' effect is to be supported. These must be stored in a depth of field A-buffer structure.

Schilling and Staßer [46] provide a description of a suitable A-buffer, however the purpose in that case was quite different. They set out to solve the HSE (**H**idden **S**urface **E**limination) problem on the sub-pixel level. As the A-buffer stores a contribution from each pixel within the viewing frustum, rather than just that of the winning pixel from a standard z comparison, the pixel data may now be read, layer-by-layer, from the front of the A-buffer. Schilling and Staßer [46] state that the major difference between the A-buffer and a traditional z-buffer is that where a z-buffer only retains one item per-pixel the A-buffer retains a list of pixel contributions.

The retention of additional information required by Schilling and Staßer's A-buffer system can cause performance issues. In order to combat this performance impact, culling can be performed to remove redundant data. For example, any winning pixel contribution from behind the focus plane can be ignored. Similarly any contributions at a similar depth to the current winning pixel (but still behind it) can be ignored. These culls should prove beneficial for a hardware based implementation of the algorithm.

The contents of the A-buffer are then blurred to varying degrees dependant on their depth values relative to the focal plane of the system. The in-focus part of the image can be determined by:

$$\frac{1}{f}=\frac{1}{u}+\frac{1}{v}$$

The resultant in-focus image will be at a distance v where the object in question is at a distance u from a lens with a focal length f.

This is fine for the simple case where the system is focused on this object, however if the system is not focused on this object more elements need to be taken into account. The image plane of an out of focus object will be a distance p and the degree of blurring is dependant on the circle of confusion. The circle of confusion can be defined as “The image of a point source that appears as a circle of finite diameter because of defocusing or the aberrations inherent in an optical system” [47]. In terms of the system presented here: the size of the circle of confusion can be determined by the paths of the rays of light which will pass through the edges of the ‘lens’ aperture and converge on the focal point. As shown by the following formula, if the aperture a is taken, then the the size of the circle of confusion C can be calculated:

$$C = \frac{|v-p|}{v} a$$

To illustrate this, Figure 19 provides a visual representation of the two preceding formulae. C and p refer to the case where the image plane is closer to the aperture than the focal plane and where C' and p' refer to the case where the image plane is further away than the focal plane.

This blurring can be achieved via the use of two MIP-map style secondary buffers sets known as the b-buffers. Each set of b-buffers is subdivided into depth levels, where one set is used for the areas of the image in front of the focal plane and one for the areas behind, this can be seen in Figure 20. The b-buffers start at the screen resolution (level

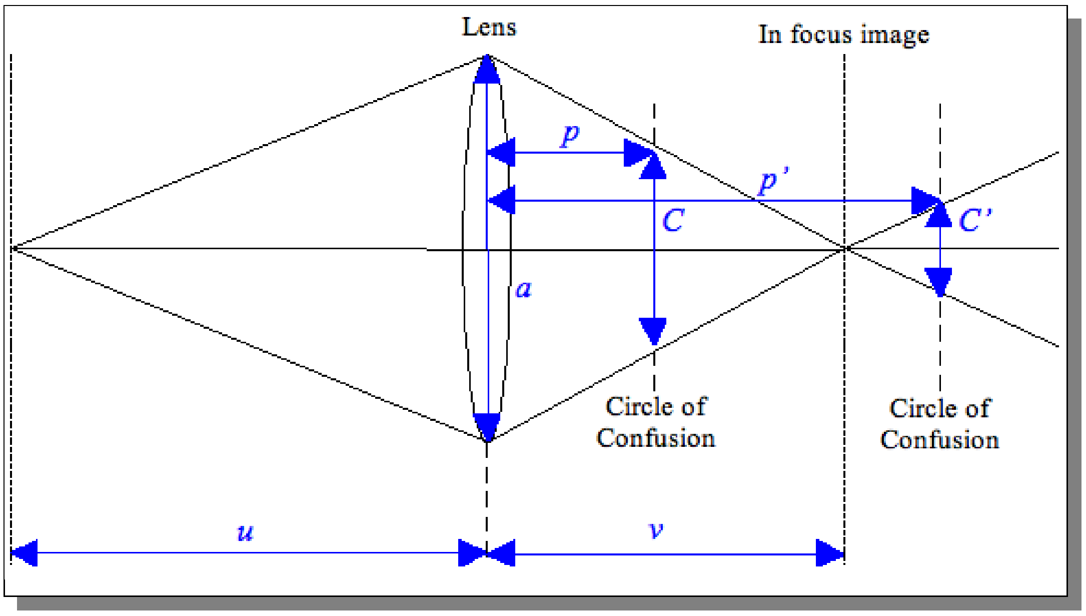


Figure 19: Calculating the Circle of confusion

0) and finish at a resolution consistent with the maximum level of blur required by the scene (level n). Two sets of b-buffers are required because pixel contributions in front of the focal plane will have a different priority to those behind the focal plain when the final image is generated by matting the contents of the b-buffers.

For each A-Buffer layer only the pixels which would usually be visible under standard z-buffering are extracted. These pixels are then removed from consideration for subsequent layers and added into a set of b-buffers. This process is then repeated for each subsequent level of the A-buffer, each time taking only those pixels which win a standard z-test and adding them to the appropriate depth level of the b-buffers.

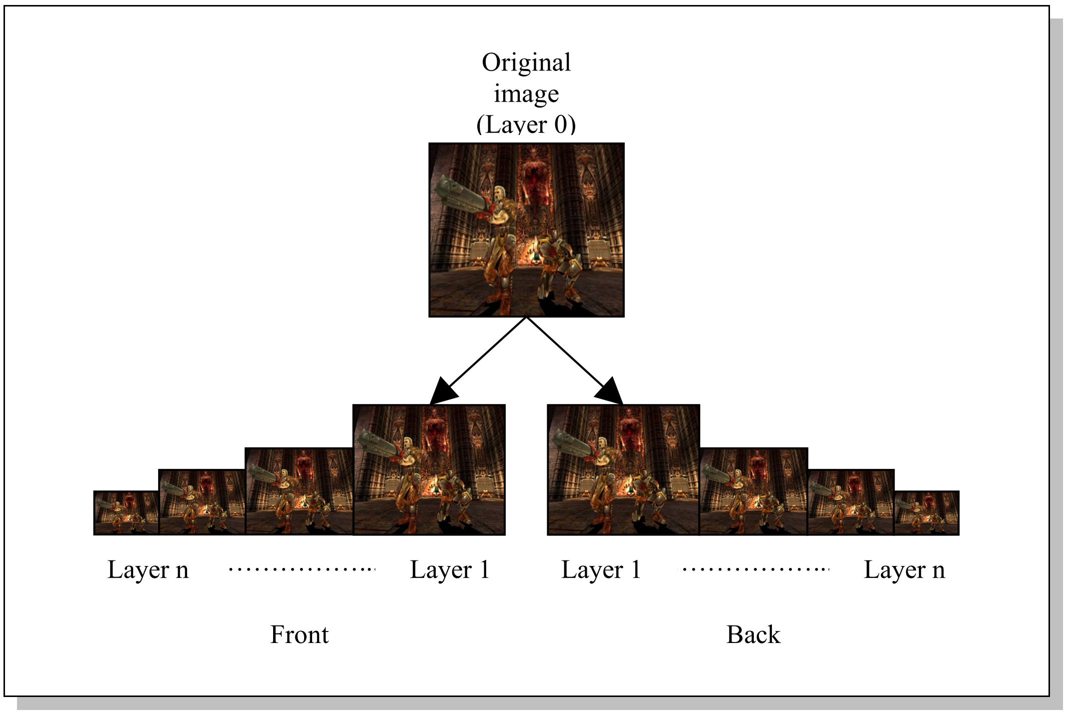


Figure 20: B-buffers

There are multiple b-buffers aligned either side of the pre-computed focus plane which are divided into those sets nearer to the view position and those further away from the view position than the focus plain. The b-buffers reduce in resolution as they move away from the focal point, building up to two sets of MIP map style images such as those in Figure 20. Where a reduction in the dimensions of the pixel data is needed, this is done by means of a Gaussian filter, as this is convenient for reversing with an equivalent Gaussian filter in later stages. Gaussian filters are useful as they are well defined and commonly used as smoothing kernels for generating multi-scale representations in computer vision and image processing [48], so their properties and advantages are well known. A set of b-buffers is created for each layer of pixel contributions extracted from the A-buffer. Despite being supported by Silicon Graphics hardware for many years, the A-buffer must be approximated in this implementation due to a lack of support in modern consumer level hardware. This is done via a process similar to that which NVIDIA term "Depth Peeling" [49], an example of which is presented in Figure 21.

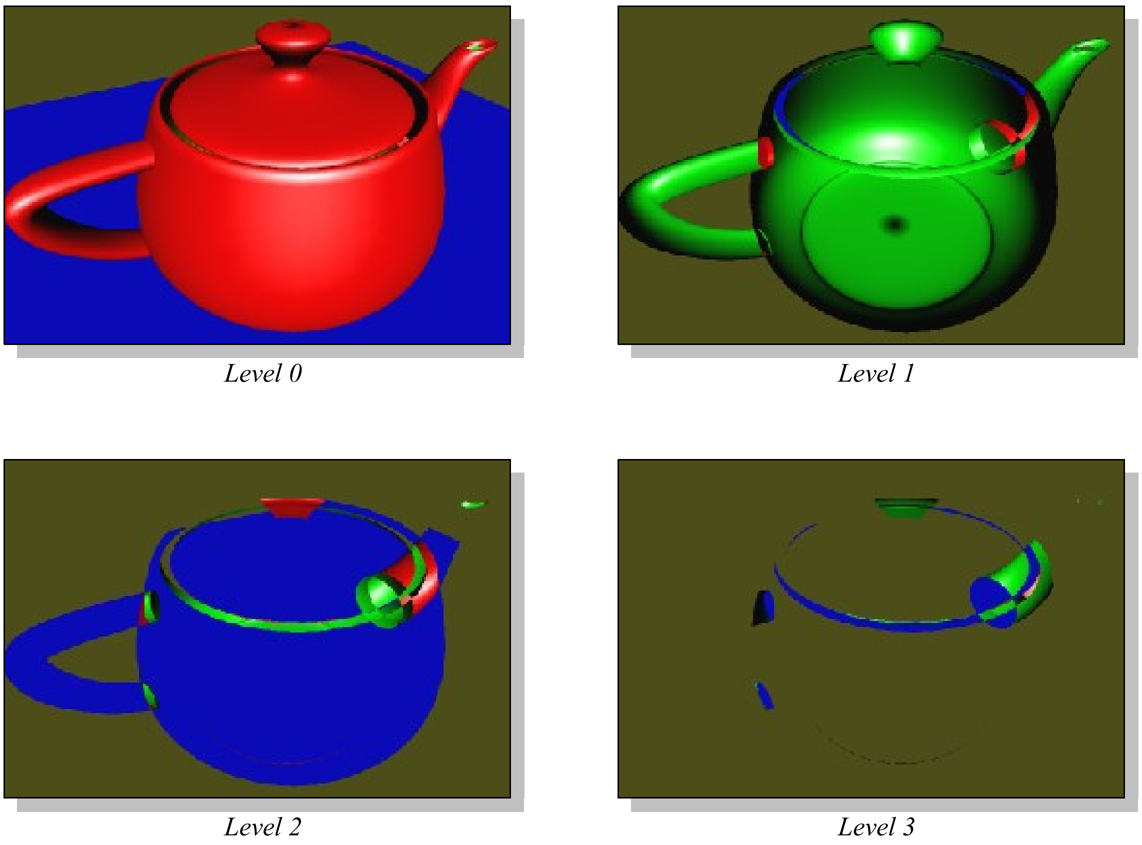


Figure 21: Depth Peeling [49]

The basic premise of Depth Peeling [49] is that each pass across the scene allows us to get a level deeper into the image. For example, three passes will end up with the scene displaying level three and everything in front of that layer, which would normally beat it in a standard depth test, will be ignored.

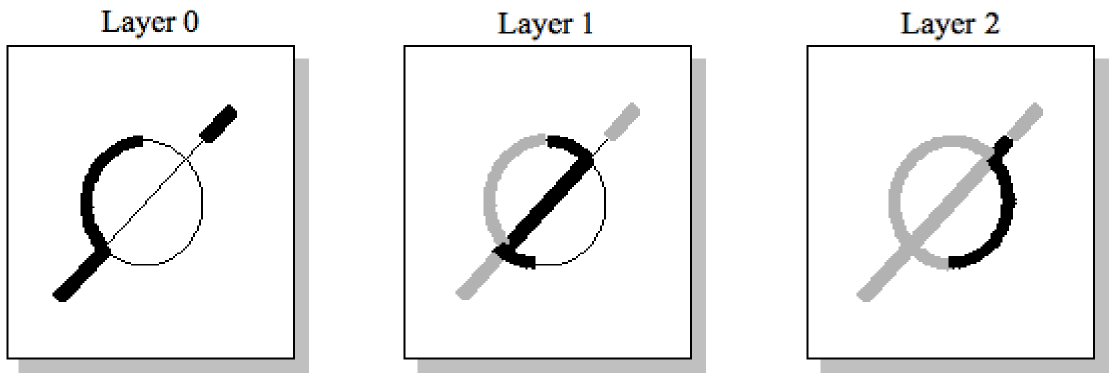
This technique is made possible by the use of multiple depth tests on a single pass. On the first pass z-buffering occurs effectively as normal, however on subsequent passes the winning pixel contributions from the previous pass are used to discard anything from a previous layer by performing the exact inverse of a standard depth test and setting the comparison value in such a way as to remove the previous winning layer. Once previous layers have been discarded by the first test the second test takes over and performs z-buffering as normal.

Within modern graphics API's the programmer generally has three depth testing options z-buffer, w-buffer and no depth testing. Standard depth testing is performed once per frame and is largely managed by the API. The Depth of field problem requires a z-buffer solution which can perform multiple z tests per frame. This is not in-line with the standard single pass approach to z-buffering but instead requires multiple passes to separate each layer.

While there are various settings that can be changed within each API's depth testing configurations (for example to set a z-bias), there is no direct way to allow discarded z values to be retained. This is required to build up the multiple layers in order to approximate the function of the A-buffer.

One possibility explored to enable this layer separation to take place is to attempt to format the output z values in such a way as to allow them to be stored in the alpha channel. This however throws up several issues, firstly the problem of getting the Z values into an appropriate format, secondly the fact that alpha values only have an 8-bit accuracy and finally there is the problem of how to get the values from the alpha channel back into the z-buffer as comparison values for the next pass.

The proposed solution to use “Depth Peeling”, means that each pass across the scene allows the technique to get a level deeper into the image. The levels can be thought of as levels of depth; level 0 being the standard z-buffer test. Level 1 is the result that the same standard z-buffer would provide if level 0 were not part of the image. Level 2 is the result that the z-buffer would provide if level 0 and level 1 were not part of the image and so on. As an example: three passes will result in an image three levels down within the scene, and everything in front of that layer, which would normally defeat it in a standard depth test, will be ignored. Examples of this can be seen in Figure 21 and Figure 22. In the case of Figure 22 the darkest sections represent the winning contributions for each layer.



22: Depth Peeling [49]

This is made possible by the use of multiple depth tests on a single pass. On the first pass z-buffering occurs as normal. However, on subsequent passes the winning pixel contributions from the previous pass are used to discard anything from a previous layer. This is achieved by performing the exact inverse of a standard depth test and setting the comparison value in such a way as to remove the previous winning layer. Once previous layers have been discarded by the first test, the second test takes over and performs z-buffering as normal.

This extra depth test is made possible by exploiting the shadow mapping facilities provided by the NVIDIA 6 series graphics cards. This works because shadow mapping is a form of depth testing; there are in fact very few differences between a standard depth test and shadow mapping. The first difference is that shadow mapping sets colour values rather than discarding pixels. However, this can be worked around by setting the results of the shadow mapping to the alpha channel. The alpha test can then be used to actually discard the relevant pixels. The second difference is that, unlike z-buffering the shadow mapping test is not tied to the camera position. Therefore, it must be explicitly set to the camera position to allow it to be used as a depth test

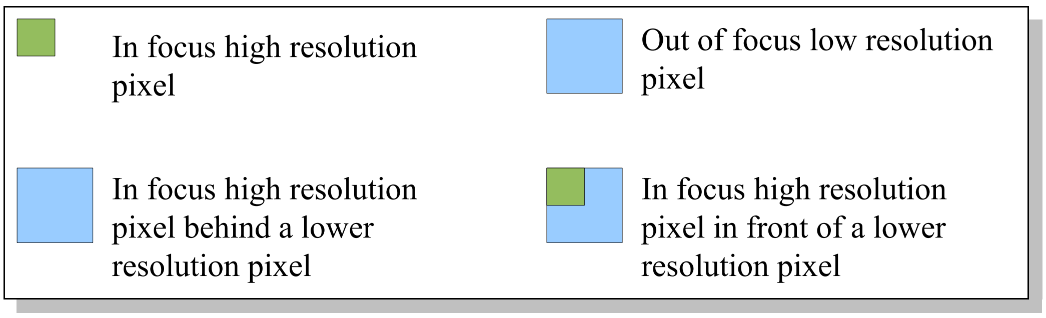

In focus high resolution pixel
Out of focus low resolution pixel
In focus high resolution pixel behind a lower resolution pixel
In focus high resolution pixel in front of a lower resolution pixel

Figure 23: Occupancy and occlusion

Any z information can be discarded at this stage. However, information on pixel occupancy (alpha) is now required. This is because the original high resolution pixels will only partially cover the lower resolution pixels generated in the b-buffers. Figure 23 shows an example of this and similar occlusion problems.

The final stage is to recombine the images into the final image. The amount of processing required for this stage can be greatly reduced by using a hierarchical method such as a Gaussian pyramid. By splatting[18] each layer onto the layer directly above rather than attempting to splat the lowest resolution directly to the highest resolution a lot of processing can be saved. The b-buffers are taken from front to back, for each A-buffer layer in turn, and accumulated to form a final image. The weighting of each pixel contribution added to the final image is decided by the occupancy (or alpha) of that pixel. Contributions from the size-reduced levels are splatted and their footprints are used in the accumulation calculations.

With each b-buffer in this set having been reduced to the appropriate size they must now be read and the contents rendered to the scene. Each pixel in the rendered scene is the accumulated value of corresponding pixels from the complete set of a-buffers.

It is intended that the general filtering and accumulation techniques in the hardware version will remain as close to ideal as circumstances allow, although this may not always be possible due to hardware restrictions. Therefore, as mentioned, a Gaussian reconstruction filter will be employed, and the convolution will be applied to a 2 x 2 set

18 See [50], p389 for more detail on splatting

of pixel samples from each b-buffer set for each displayed pixel, as shown in Figure 24. The b-buffer levels will be accumulated from front to back for each a-buffer layer (i.e. near to far from first layer, then near to far from second layer, etc.), for each pixel with some available occupancy remaining.

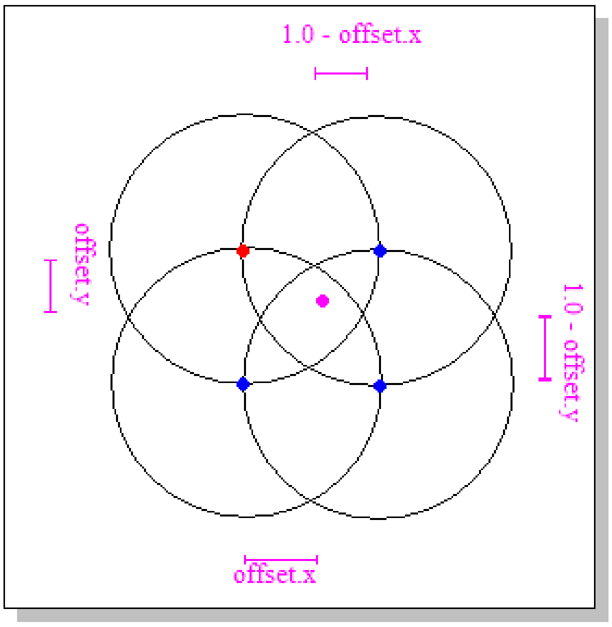


Figure 24: Example sample points

The Gaussian filter is defined by: $e^{-(x^2+y^2)}$.

The filter values may be calculated at run time. However, as the filter function should not be required to vary on a per pixel basis; gains can be made by pre-calculating the values.

The four texels[19] of each image layer nearest to the screen pixel they represent must be

19 Texture map pixels.

accessed, then each is multiplied by the corresponding filter value and their sum taken per level. These sums will then be accumulated until either the occupancy reaches its maximum, or there are no more levels (i.e. the image is complete).

As the currently available test hardware may only access up to 16 texture maps in a single pass, it becomes clear that the hardware implementation shall be required to use one pass for each a-buffer layer. With 14 b-buffer levels in each a-buffer layer, along with one pre-calculated filter and one result from the previous layer, all 16 texture units will be required for each individual stage in the accumulation process.

An occupancy limit of 0.0-1.0 is set on the incoming pixel data from the b-buffers to ensure it is only added into the frame while meaningful occupancy data is available. An occupancy of 1.0 describes a fully opaque pixel, and 0.0 describes a fully transparent pixel.

Once all levels of each A-buffer layer have been added into the final image, located in the frame buffer, the image can be rasterised to the display and the processing of the next frame may begin.

The advantages of this method is that the different resolutions can be easily obtained by splatting the pixels to a lower resolution this also enables a higher level of blurring to be obtained with relatively little processing required. Extra b-buffers cost little extra time and memory so a higher level of blur can be obtained comparatively cheaply. The

- Input standard scene data, including required texturing and lighting details.
- Perform Z-Buffering along with:
    - Multi-pass depth peeling to get A-buffer layers.
    - Retention of extra information in A-Buffer layer pixel data, as described in section 3.3, i.e. multiple contributions from each pixel.
- Perform culls to remove unnecessary data about pixels behind the focus plane.
- Blur A-Buffer levels dependant on distance from plane of focus, this is done by drawing the image data to different sizes in the b-buffers utilising a Gaussian filter, as described in section 3.3.
- Recombine the layers by splatting, front to back taking into account occupancy for weighting of each pixel:
    - The B-buffer levels are accumulated front to back for each A-buffer layer (i.e. near to far from first layer, then near to far from second layer, etc.), for each pixel with some available occupancy remaining.
- Display final recombined image.

Figure 25: Depth of Field simulation hardware process

hardware process is summarised in Figure 25.

3.4 Results

In this section both hardware and software versions of the depth of field algorithm are examined. Success is determined by a number of factors, including: real-time performance, visual quality and compatibility with current hardware.

3.4.1 Software

As a proof of concept, the first stage of development requires the production of a software version of the depth of field algorithm, which runs under a modern graphics API to prove compatibility with current API's and to allow an easier transition to hardware at a later stage.

The basic operation of the software version is as follows:

1. The first task is to set up the film plane along with any variables that require initialising and clear all the buffers ready for processing.
2. Next, a sharp image is drawn, this is simply a standard process to produce a pinhole camera model image. This sharp image can then be fed into the later stages for processing and eventually output as a depth of field image.
3. The processing of the A and b buffers requires that the A-buffer retains a list of pixel contributions, to be blurred dependant on depth, and that the b-buffers contain the varying levels of blurred images required for this task.

4. The final step is to plot the pixels.

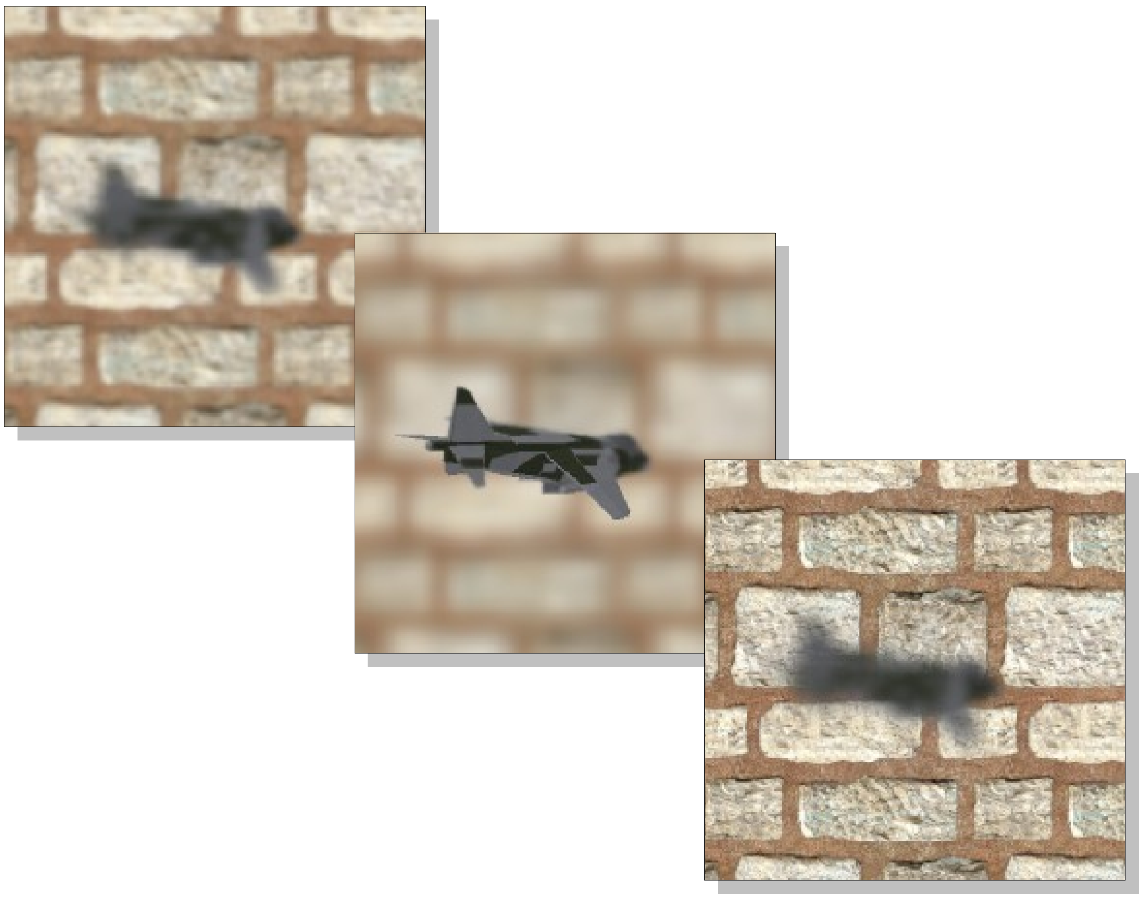


Figure 26: Software Depth of Field Tests

Figure 26 shows some sample visual outputs taken from the software version. The extra level of realism provided by the see-through effect can be seen clearly in these images. Unfortunately, the frame rate is extremely low, taking several minutes to generate a single frame. Too low for any practical use other than generating static or pre-rendered images, for which ray-tracing provides superior image quality.

3.4.2 Hardware

The next step is to attempt to investigate possible implementations of this technique on modern consumer level graphics hardware. Each aspect of the technique is looked at in turn, with the eventual aim of meshing each separate component into a complete hardware-based depth of field simulation.

3.4.2.1 A-Buffers

Depth peeling tests were set-up to show that the process did in fact split the layers as anticipated. Figure 27 demonstrates the depth peeling process and clearly shows that four distinct layers are apparent from the tests.

When considering the frame rate, with the addition of depth peeling, the tests lose around 85% of the frames per second[20] when compared to a simple texturing and lighting example of the same scene. This equates to around 45FPS compared to around 300FPS with basic texturing and lighting (MIP Mapped [51] bilinear filtering [52] and Gouraud shading [9]).

20 FPS

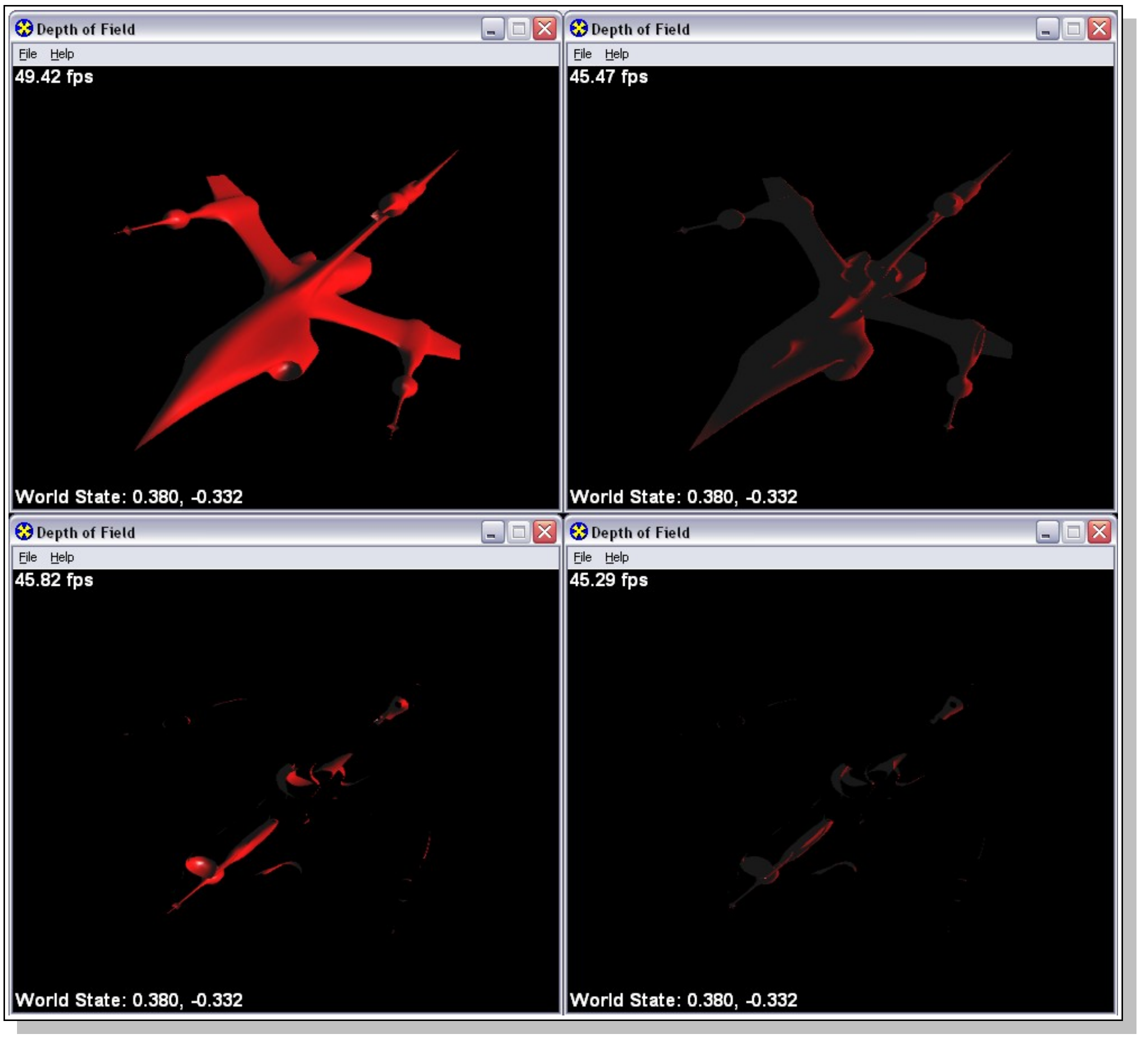
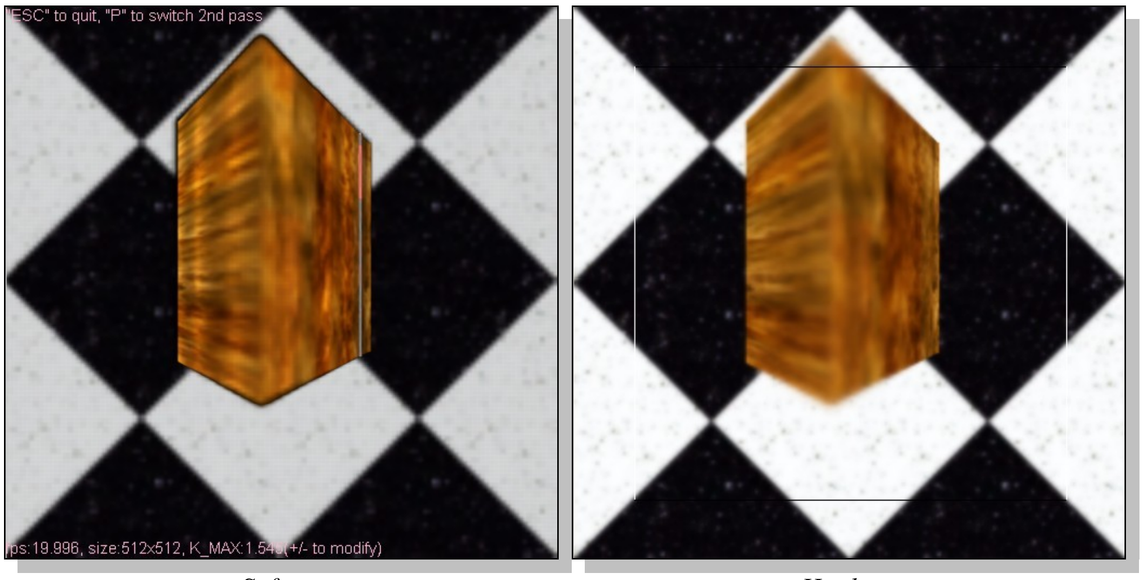


Figure 27: Depth peeling tests

3.4.2.2 B-Buffers

Visual comparisons of the test images from hardware and software versions of the accumulator show slight variations, this is demonstrated by Figure 28.

In Figure 28 some slight discrepancies in colour is observed between versions. This is most likely due to a difference in the rounding of values between the hardware and the software. As shown in section 2.3.1, the GPU supports much lower precision floating point calculations than the 80-bit precision used internally within modern CPUs floating



"ESC" to quit, "P" to switch 2nd pass

fps:19.996, size:512x512, K_MAX:1.54[illegible](+/- to modify)

Software *Hardware*

Figure 28: b-buffer tests

point units. This is accentuated by the fact that the software version takes advantage of double precision floating point numbers rather than the basic floating point available in the GPU, adding much greater precision over that possible in the hardware version even without considering the issues discussed in section 2.3.1. When coupled with the accumulation process for each A and B-buffer layer which will compound any error, this means that the errors shown in 2.3.1 may be exaggerated many times over causing the effect seen in Figure 28.

Although an encouragingly high frame rate is achieved for a single a-buffer layer, a large drop off in throughput is recorded with the addition of extra layers. Some figures for the frame rates are detailed in Table 1.

A-buffer layers	B-buffer layers	FPS
1	6	30.0
	10	30.0
	14	20.0
2	6	20.0
	10	15.0
	14	12.0
3	6	15.0
	10	10.0
	14	8.57
4	6	12.0
	10	7.50
	14	6.67

Table 1: b-buffer performance

3.5 Conclusions

With the aim of implementing a version of the depth of field algorithm, which supports the see-through effect, and investigating the utilisation of modern hardware in an attempt to get it to operate in real-time it can only be concluded that there has been limited success. While each individual component of the system meets the 30fps performance criteria set, a combination of them to create the full depth of field solution is not currently possible while maintaining real-time performance.

With the software version the screenshots show that a convincing effect is created by use of this method. While the frame rate is extremely low, it does prove that the algorithm provides a plausible effect and hence that a full hardware implementation could be beneficial in providing greater realism to games and simulations in the future.

The scope for future work in the field is quite large, as much is currently not possible in real time on existing hardware. Future hardware developments will undoubtedly lead to a solution similar to the one proposed being possible in real time.

This evaluation is appropriate for the GeForce 6 generation of GPUs on which it was examined. Therefore, as with all things reliant on rapidly advancing technology it is subject to change as hardware progresses. The current situation is that the GeForce 8 series and similar GPUs are becoming commonplace, while the GeForce 9 series is approaching release. This makes the next few years, as the new generations of GPUs are released, an ideal time for re-evaluation of this technique.

This is particularly true now. Since the completion of this work Depth of Field has become an extremely popular area of research. Papers such as: Practical post-process Depth of Field [53], Interactive Simulation of the Human Eye Depth of Field and Its Corrections by Spectacle Lenses [54] and Depth-of-Field Rendering by Pyramidal Image Processing [55] all propose real-time solutions to the various issues covered in section 3.2. However, none of these solve all the problems with Depth of Field simulation and all have significant limitations.

Practical post-process Depth of Field [53] takes a standard single layer approach to rendering and completely ignores transparency. Therefore, it cannot support the see through effect. Although the technique presented does allow colour to bleed through from the background in areas where the foreground is blurred. This causes discolouration of the blurred area, as opposed to a see through effect, and is noted as a shortcoming of the technique rather than a feature. The technique also cuts off the blurring in certain regions and at higher levels of blur for “artistic” [53] purposes. In this case, aesthetic considerations caused the need for the author to artificially limit blur levels in order to hide the shortcomings of technique. This is because the technique blurs the “screen” [53] rather than individual objects, thus limiting the level of blur which can be achieved before the effect “breaks down” [53].

Interactive Simulation of the Human Eye Depth of Field and Its Corrections by Spectacle Lenses [54] approaches the problem from the specialist perspective of spectacle lens simulation. It is for this reason that the technique is impractical for the types games and simulations which are discussed at the start of Chapter 3. The technique uses a specialist coordinate system, requires very low polygon models by today's standards[21] and represents blur levels in terms of precomputed voxels. This means that the technique requires prior knowledge of the scene which is simply not practical in most modern interactive games and simulations, and requires prohibitively low polygon counts in order to achieve real time performance.

Depth-of-Field Rendering by Pyramidal Image Processing [55] uses an approximated Gaussian filter for blurring, this causes over blurring which is severe in some cases. The technique also only provides real-time performance for small image sizes and small circles of confusion in static scenes.

21 Kakimoto et al talk in terms of thousands of polygons for the whole scene to achieve real-time performance. Whereas, today's games often use tens of thousands, or even millions of polygons, for single objects.

Chapter 4: Texture Mapping and Aliasing

Traditionally, in its simplest form, texture mapping is performed by taking an image, or texture, and applying it to an object, or objects, in a scene. This is done to add detail, surface texture and/or colour. The process can be thought of as being similar to wallpapering an object or wrapping a gift with paper. However, as computer graphics tend to be displayed on two dimensional screens, and hence only simulate three dimensions, there is often the added complexity of requiring perspective corrections[22] to give a more realistic appearance to the textured scene.

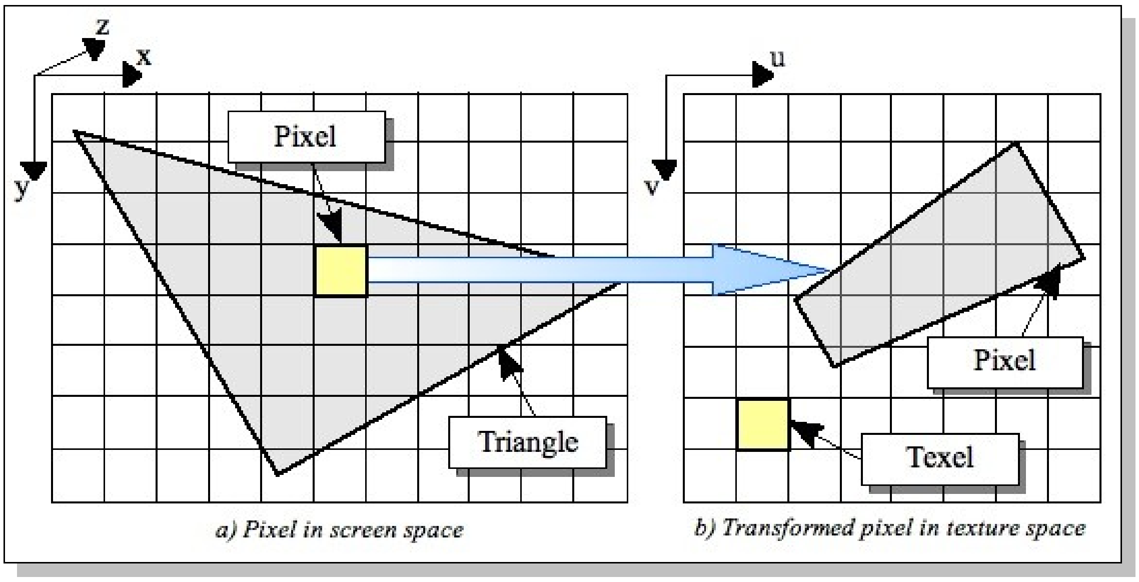


Figure 29: Transforming a pixel into texture space

Hence texture mapping in computer graphics is performed by mapping the texture space

22 An example of perspective correct texture mapping can be seen in Figure 30

co-ordinates (u, v) onto the object in question, which has (x, y, z) co-ordinates. This is done to obtain texture space co-ordinates, which are then used to sample the texture. The results are then applied to the pixel in image space to create the desired textured effect on the object as illustrated by Figure 29.

As can be demonstrated by viewing the texture mapping example in Figure 29 a) the process is quite simple and begins by specifying texture co-ordinates for each corner of the triangle. Traditionally this would be a polygon, however modern graphics hardware will render the scene almost exclusively as a combination of triangles, as seen in Figure 29 a). This is because triangles are the simplest complete shape with which other polygons can be built, with the exception of point-based rendering systems [56], which provide extremely poor performance by comparison.

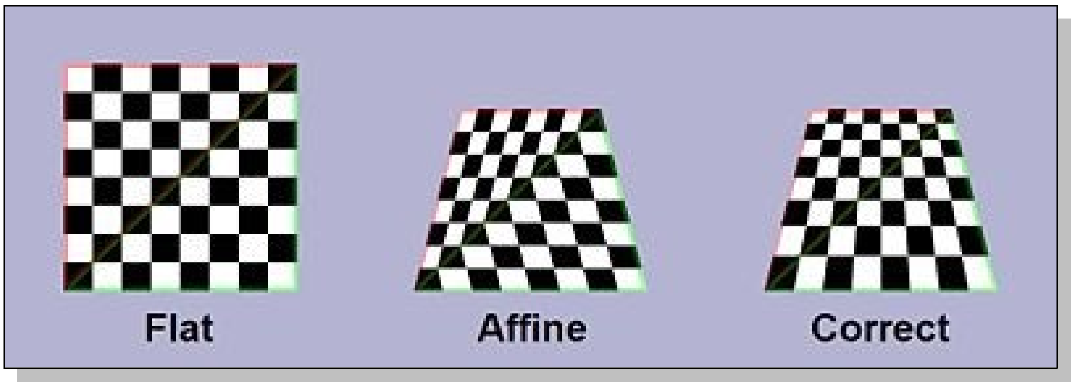


Figure 30: Why perspective correct texturing is required [57]

Triangles also have the advantage of providing a constant scan-line delta (slope gradient), and consistent memory requirements. The consistent memory requirements are due to the fact triangles consist of three sets of (x, y, z) co-ordinates (i.e. always three corners) plus whatever information is required in terms of texturing, lighting etc. It is because of these properties that triangles can provide the building blocks of any larger polygon shape or mesh. Hence their popularity in modern computer graphics as these benefits outweigh those of using a range of different polygons.

The main task involved in texture mapping is to calculate the texture co-ordinates at the current pixel, this is in order to know which points to sample in texture space. The simplest way to do this is via a process known as affine texturing [58], which, while being a simple process lacks perspective correction. Figure 30 [57], demonstrates the shortcomings of affine texture mapping when the surface is angled away from the viewer.

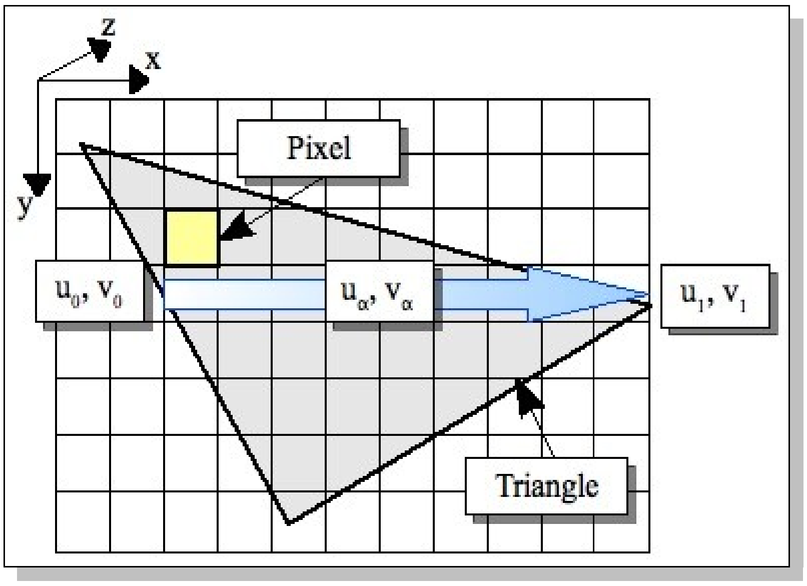


Figure 31: Interpolating texture co-ordinates across a polygon

Affine texture mapping begins with the three triangle corners $(x_0, y_0, z_0, u_0, v_0)$, $(x_1, y_1, z_1, u_1, v_1)$ and $(x_2, y_2, z_2, u_2, v_2)$. u and v are then interpolated across the triangle. As previously mentioned, using triangles, the scan-line delta[23] is constant. This makes the process much quicker, as extra slope calculations are not required. The texture co-ordinates are interpolated across the scan-line, as shown in Figure 31, thus providing the texture co-ordinates at the desired pixel. This is used as an index to sample the texture, the result is then applied to the pixel.

Mathematically the process is as follows:

$$u_\alpha=(1-\alpha)u_0+\alpha u_1 \text{ and } v_\alpha=(1-\alpha)v_0+\alpha v_1$$

where:

$0\leqslant \alpha \leqslant 1$

u_α , v_α = Interpolated texture co-ordinates.

u_0 , v_0 = texture co-ordinates at start point.

u_1 , v_1 = texture co-ordinates at end point.

This technique is extremely simple but suffers with issues associated with a lack of perspective correction as illustrated by Figure 30. Most modern texture mapping techniques are based on a method described by Chris Hecker [59]. Hecker's technique involves dividing throughout by z . Therefore with Hecker style perspective corrected texture mapping instead of interpolating u and v , u/z , v/z and $1/z$[24] are interpolated in the same manner as u and v previously were, down the triangles edge then across the scan-lines. This gives the interpolated values of u/z and v/z , these are

23 Gradient.
24 Z (depth) values are required to enable the perspective correction demonstrated in Figure 30.

then divided by the $1/z$ values, which results in an interpolated and perspective corrected u and v. The perspective corrected values are then used as an index to sample the texture. Thus:

$$u_\alpha = \frac{(1-\alpha)\frac{u_0}{z_0} + \alpha\frac{u_1}{z_1}}{(1-\alpha)\frac{1}{z_0} + \alpha\frac{1}{z_1}} \text{ and } v_\alpha = \frac{(1-\alpha)\frac{v_0}{z_0} + \alpha\frac{v_1}{z_1}}{(1-\alpha)\frac{1}{z_0} + \alpha\frac{1}{z_1}}$$

where:

z_0 = z value at start point.

z_1 = z value at end point.

There are numerous other methods that can be employed to do this, which often depend on the particular hardware or the filtering methods utilised in conjunction with texture mapping. LaMothe 2003 [60] provides a more in-depth discussion of perspective corrected texture mapping. One of the simplest methods, as seen in Figure 29 for demonstration purposes, is to assume the pixel is square and projects into texture space as a trapezium. Once the texture space pixel footprint, and therefore the texture co-ordinates at the pixel, are known; texture samples can be taken which are applied to the pixel's location in screen space. This process is repeated for each pixel of each triangle in the scene producing the final textured polygons.

More recently texturing has often been combined with other techniques such as MIP mapping, bilinear filtering [52] and anisotropic filtering[25] to help reduce artefacts and aliasing. However all these techniques have one common limiting factor: the original image on which the texture is based. The problem with this approach is similar to the problems faced in signal processing, where the sampling rate of the signal, or in this case the image, is too low; causing aliasing. Most of the currently available techniques attempt to compensate for this effect by post-processing the image, treating the symptom and not the cause.

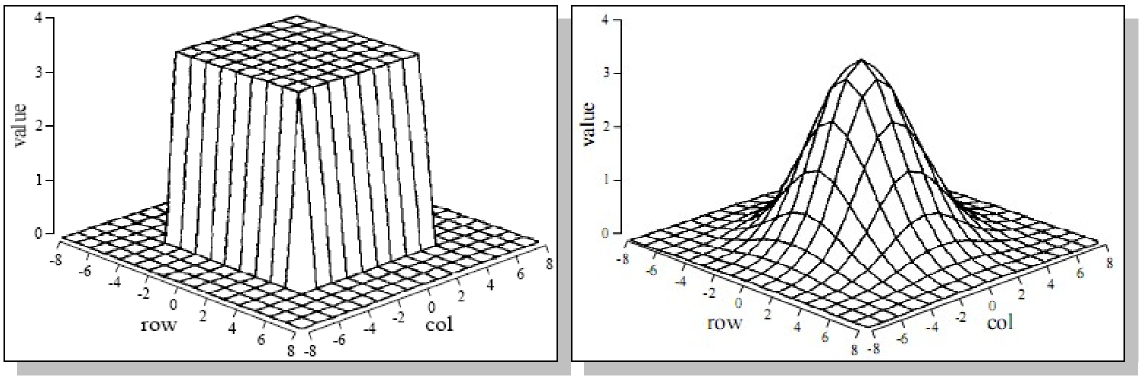
4.1 The Aliasing Problem

Aliasing is an effect that causes different continuous signals to become indistinguishable, or aliases of one another, when sampled. In the signal processing domain, this means that the source signal has not been sampled at a rate high enough to fully reproduce the continuous signal of the original. Therefore it can be said that: aliasing is caused by inappropriate or insufficient discrete samples of continuous data. The average computer monitor and graphics card combination are simply not capable of producing a sufficiently high sample rate, or resolution, to accurately represent many forms of input. So in effect aliasing in image processing and 3D computer graphics is caused by under-sampling of the source image.

Therefore it can be said that the intrinsic way in which monitors and graphics cards work causes aliasing. This is because computer monitors, televisions and similar display devices output pixels. Pixels are often easiest to think of, conceptually, as small squares

25 See section 4.2.6

of colour, although they are in-fact points with a Gaussian like fall off, that represent discrete samples of continuous data. Figure 32 provides an example of a Gaussian and a square function for illustration purposes.



a) Square pixel *b) Point with Gaussian fall off*

Figure 32: Possible Pixel representations [48]

When considering the case of a standard 17” LCD computer monitor, the native resolution will be fixed. LCD monitor resolutions of 1280x1024 are common place in today's market. This equates to approximately 1.3 million pixels on screen, with each pixel being capable of displaying upwards of 65,000 colours[26]. This may appear to be a large number of pixels, however when compared against the average professional print resolution of 300 dpi (dots per inch), or 90,000 dots in each square inch, it can be seen that monitors have a fairly low sample rate by comparison.

An example of the possible output of a monitor is compared to an example of a source image in Figure 33, where each square in the example represents a pixel on screen. Because pixels can only display a single colour at a time in the example output they

26 This equates to 16-bit colour, which is quite low by today's standards.

would have to be either black or white in order to represent the example image. This is problematic, as some of the pixels are naturally sliced by the edge of the object. Therefore, a decision must be made about how to deal with pixels that are not 100% covered by the original image.

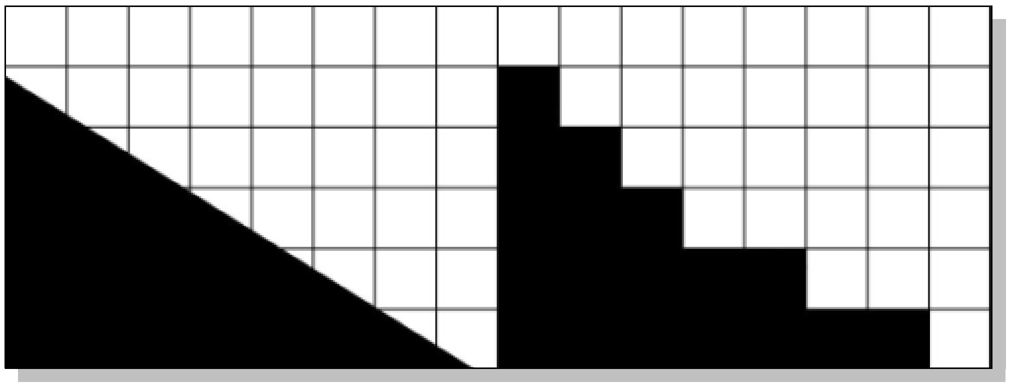


Figure 33: Original source image Vs On-screen representation

Usually, the decision on how to deal with these pixels is a simple one, based on whether or not over 50% of the pixel is covered. If more than 50% of the pixel is covered then the pixel is coloured, otherwise it is not. This means the line will not be straight as intended, but will instead suffer from non uniform edges as seen in Figure 33. While this example deals with edge aliasing the same principals can also apply to texture aliasing.

4.1.1 Types of Aliasing

There are two main types of aliasing that affect 3D computer graphics, edge aliasing[27] and texture aliasing[28]. Existing solutions for each of these are

27 See section 4.1 and Figure 33
28 See section 4.1.1.2

examined and analysed.

4.1.1.1 Edge Aliasing

The problem of edge aliasing can be lessened relatively simply via techniques such as supersampling or multisampling. These types of techniques are often known by the umbrella term of FSAA[29], both involve taking multiple samples per pixel and averaging the results to provide smoother, less jagged, edges.

4.1.1.1.1 Supersampling

Conceptually supersampling [61][62] is simpler than multisampling, but unfortunately it suffers from significantly lower performance.

The basic principal of supersampling involves initially rendering the scene to a resolution much larger than the display is capable of outputting. Multiple samples are then taken for each pixel in the display resolution and then the result is averaged to provide the final colour of each pixel as shown by Figure 34 a).

When using 4x super-sampling on an output device with a resolution of 800x600, you would initially draw the scene to a resolution of 1600x1200 before averaging 4 pixels from the larger image into one pixel of the final 800x600 image. From this example it is simple to deduce that super-sampling quickly becomes unrealistic in real time, unless

29 **F**ull **S**cene **A**nti-**A**liasing.

running directly on optimised hardware. This is because the total number of samples increases rapidly for each extra level of supersampling. Low levels of super-sampling such as 4x times supersampling require 4 times the work of regular rendering, hence the name. Even at that level supersampling can soon become unmanageable in complex scenes where other techniques such as normal mapping and HDR[30] lighting are in operation.

Another advantage of supersampling is that it can improve texture aliasing and image quality as well as the obvious affect is has on edge aliasing. This is due to the extra samples reducing texture aliasing in the same manner to how the extra samples reduce edge aliasing. Therefore, as supersampling does not distinguish between different aspects of a scene the whole scene benefits from the extra samples.

4.1.1.1.2 Multisampling

Multisampling is slightly more complex than supersampling, but at the same time it is a much more efficient version of supersampling. Multisampling does not draw the whole scene at a higher resolution but instead uses sub-pixel samples, or virtual pixels to approximate supersampling.

These virtual pixels correspond to the sample points as shown in Figure 34 b), this achieves a similar effect to supersampling (Figure 34 a)), but at a lower cost. This is because of the use of sub-pixel samples. Texture fetches are kept to a minimum, as the

30 **H**igh **D**ynamic **R**ange.

results of a texture fetch are shared between all the sub-pixels of the main pixel instead of the one texture fetch per sample approach used in supersampling. This means that, where supersampling has a 1 to 1 relationship between texture fetches and samples taken from the higher resolution, multisampling has a 4 to 1 relationship. With every four sub-pixel samples sharing the results of one texture fetch. The main performance difference between supersampling and multisampling however, is that modern GPUs are intelligent enough to only perform multisampling on edge pixels, thus drastically improving rendering speeds. This negates the advantages to image quality and texture aliasing provided internally across the textured object by supersampling and simply provides improvements to the edges. Figure 34 shows an example of both methods, where the yellow dots represent sample points. This clearly demonstrates the fundamental differences between the two techniques, both are taking four samples per-pixel however where supersampling samples four pixels, multisampling samples four subsets of the same pixel.

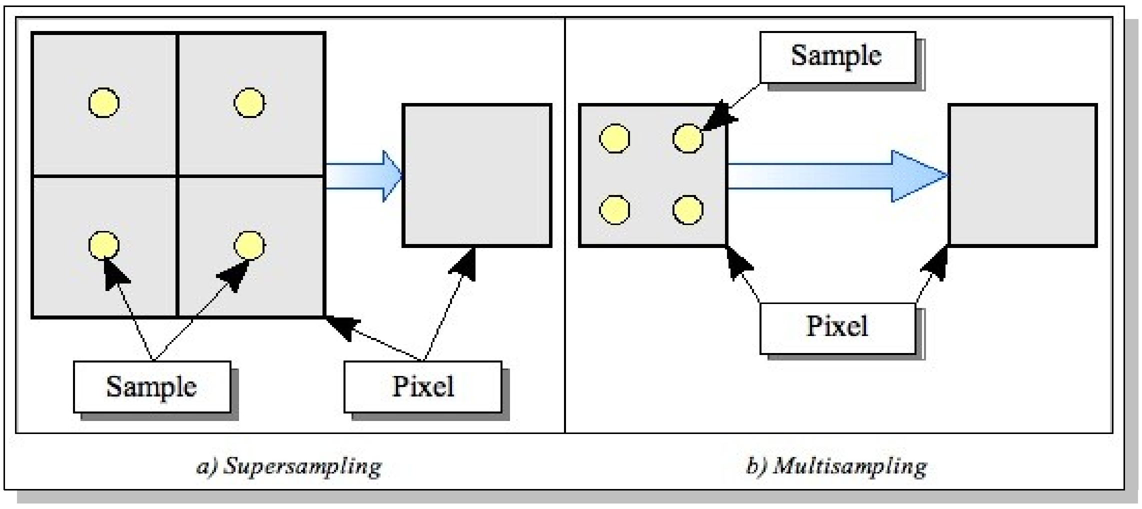
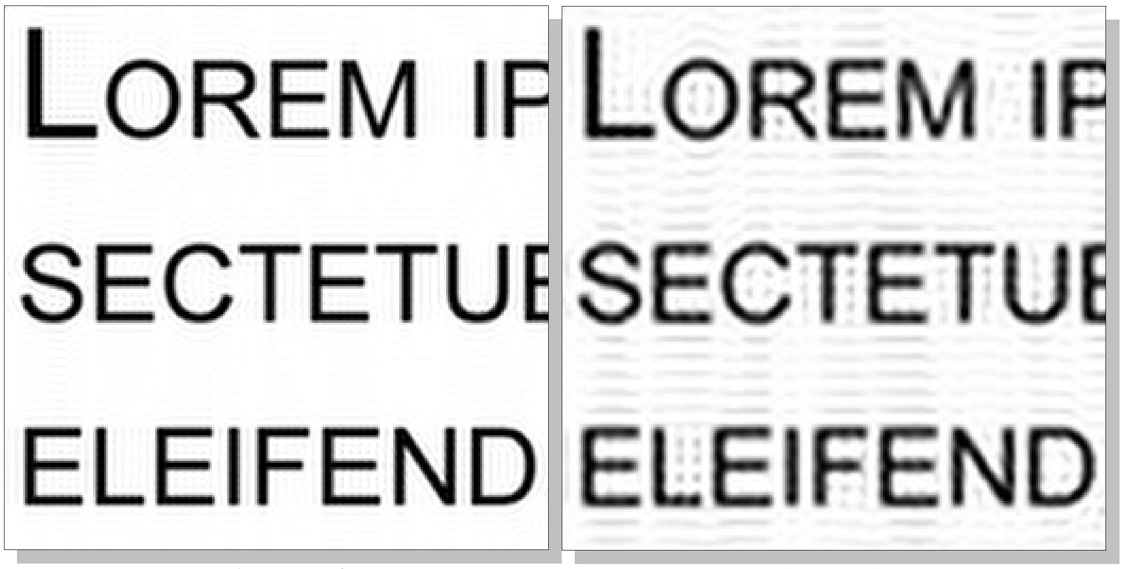


Figure 34: FSAA Comparison

Multisampling is essentially an optimisation of supersampling used by vendors such as NVIDIA and ATI to increase the performance of their chips at the expense of image quality. Supersampling does however provide the better results as far as image quality is concerned and even improves texture quality in some cases within objects, and so it is preferable to multisampling from a purely image quality perspective.

4.1.1.2 Texture Aliasing

Texture aliasing, as shown in Figure 36 a), is a particular problem for 3D graphics. During rasterisation of a textured polygon, screen coordinates are mapped into texture coordinates (u, v) and the texture is sampled using these coordinates. If the polygon is shrunk due to perspective, it may cover only a few pixels in screen space. This will result in only a few possible sampling points spread across the original texture. If the texture contains a lot of fine detail, then the sampling points may not be representative of the original texture. Figure 35 provides an example of these problems for the case of an original 1024x1024 image sampled down to 512x512 as would be done for standard MIP Mapping.



a) Original *b) Sampled*

Figure 35: Re-sampling of an image, resulting in aliasing

Figure 35 b) is the result of performing a simple point sampling operation on Figure 35 a). The banding effects present in Figure 35 b) correspond to high-frequency components of the original image, where large variations of colour are concentrated in small areas (in this case black to white and white to black). Therefore, as already demonstrated for edge aliasing, it can be said that sampling of an image that has high-frequency components with a grid of a lower frequency results in aliasing artefacts, such as those in Figure 35 b) and Figure 36 a).

Unlike edge aliasing, texture aliasing cannot be solved by common and easy to implement full scene anti-aliasing techniques like multisampling. However, the most computationally expensive full scene method; supersampling, does have some benefits for texture aliasing. These are achieved by artificially increasing the sample rate over the whole scene. Object visibility, object shape, perspective, shadowing and texture frequency variance all affect texture aliasing and add to the complexity of addressing the problem.

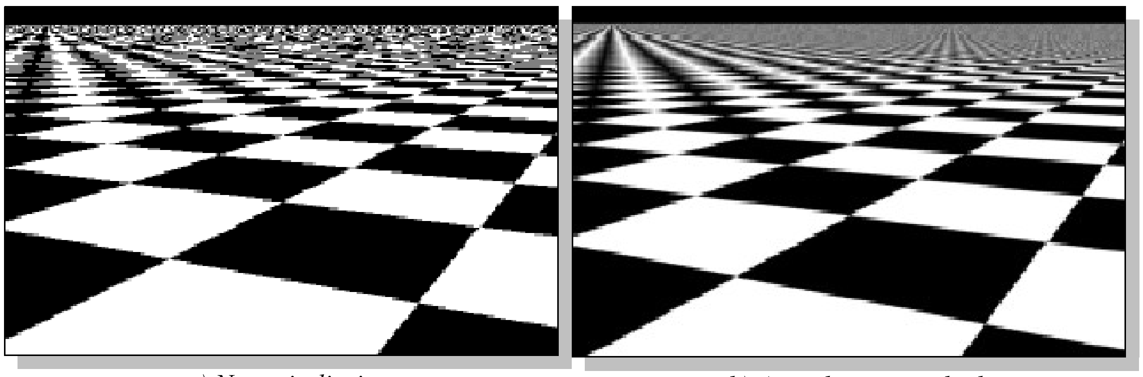


Figure 36: Texture Aliasing

Figure 36 provides an example of texture aliasing effects, as can be seen the under-sampling in Figure 36 a) has caused severe aliasing which is visible even on a static image. The sampling theorem (or Nyquist limit) [63][64] states that the sampling frequency must be at least twice the maximum frequency of the source signal to avoid aliasing. With computer graphics this is often an impossible task to achieve as monitor resolutions are a limiting factor for sampling frequency. Therefore aliasing is to a large extent inevitable and efforts much be made to minimise such effects.

In most real-time systems the approaches used to tackle texture aliasing tend to be variations and extensions of MIP mapping [51], which is discussed in greater detail in section 4.2.1. However, such approaches have a number of shortcomings. Most notably such solutions are usually isotropic[31], such as Bilinear [52] and Trilinear filtering [65].

31 Meaning they often only use square filtering patterns

This is far from ideal as the problem really requires an anisotropic approach, or more specifically a non-square filtering pattern, such as the approach taken by Cant and Shrubsole [66].

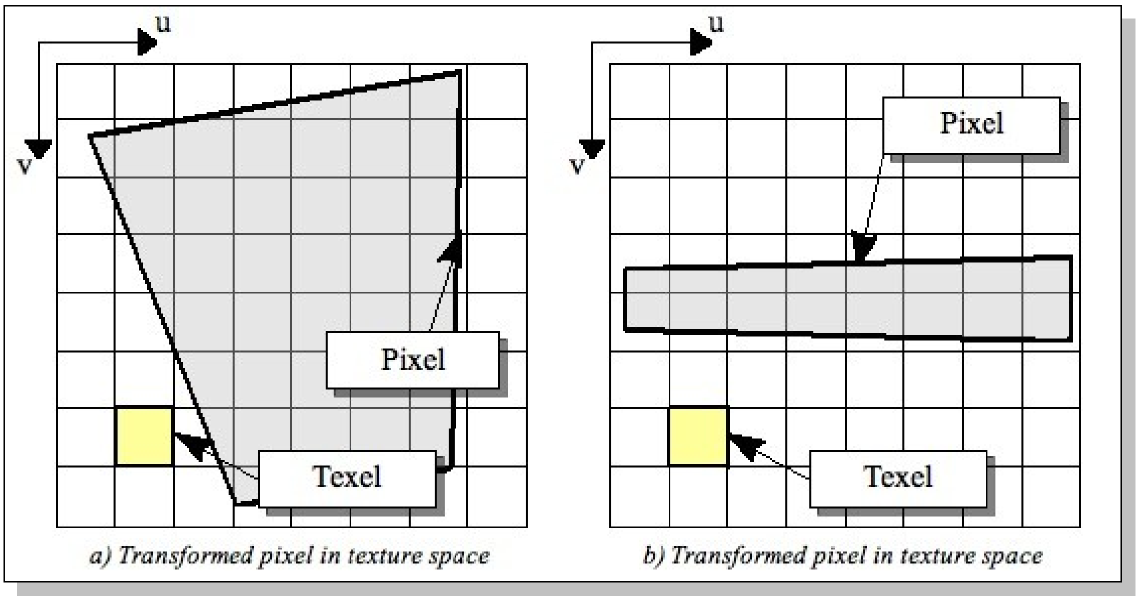

u
v
Pixel
Texel
a) Transformed pixel in texture space
u
v
Pixel
Texel
b) Transformed pixel in texture space

Figure 37: Examples of pixels transformed into texture space

Figure 37 provides some examples of pixels which have been transformed into texture space. As can be seen the resultant patterns are often non-square. In addition, as in Figure 37 b), the patterns can also be long and thin. That is a situation which could cause MIP mapping to suppress the texture, in turn causing fading or blurring. As can be seen from this, an anisotropic approach would seem ideal, however current attempts to do so in real time, while superior to isotropic methods, for the most part still have significant drawbacks.

Many of the best performing techniques are unsuitable for modern shader based graphics hardware, and those that are compatible with modern hardware suffer other shortcomings. With all this in mind much research and development has been undertaken into methods which reduce aliasing. Many of the more popular and efficient methods are discussed here, as well as some higher quality methods which are not currently possible in real-time.

4.2 Existing Solutions

There are many current solutions to the texture aliasing problem. However, all currently popular solutions contain compromises to allow real-time performance and those techniques which provide better anti-aliasing properties suffer with performance.

4.2.1 MIP Mapping

MIP mapping was first introduced by Lance Williams in 1983 under the title "Pyramidal parametrics" [51] and has become one of the most common techniques in computer graphics today. MIP is an acronym of the Latin phrase "Multim Im Parvo" which can be taken to mean "many things in a small space".

Figure 38 shows some sample images taken from Tom's Hardware [67] which clearly shows the benefits of MIP mapping over point sampled texturing. The MIP mapped image shows a substantial reduction in artefacts, although at the expense of image sharpness. It can be argued, at least with the more distant objects, that this case is closer

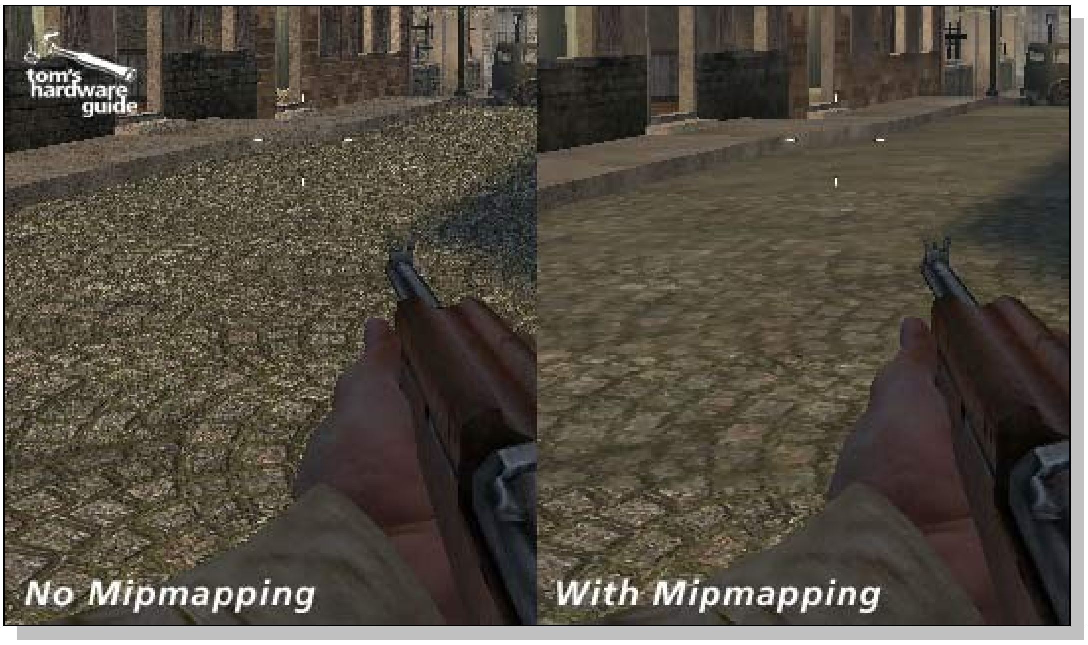


Figure 38: Flat Texturing Vs MIP Mapping [67]

to reality although perhaps not to the extent seen with MIP Mapping. What the image does not show is the severe motion aliasing that is apparent in the non-MIP mapped version, when the scene is in motion. Although, while artefacts are still apparent in the MIP mapped version they will be far less severe.

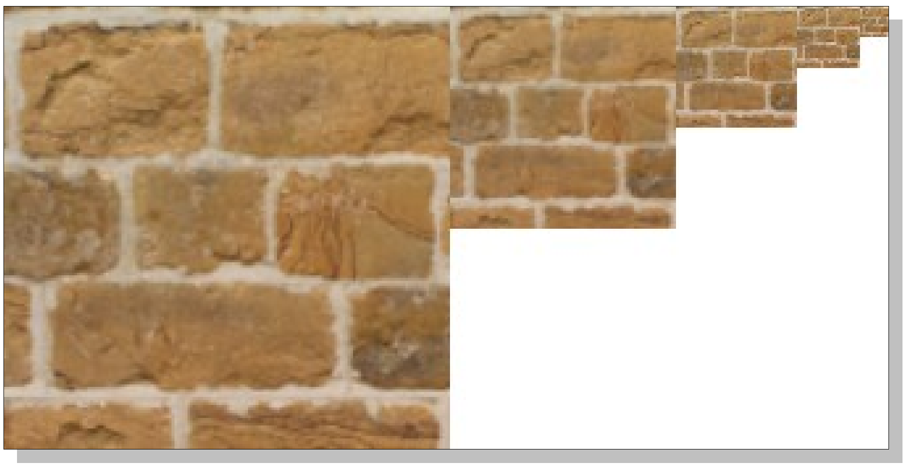


Figure 39: Example of a partial MIP map set

As illustrated by Figure 39, MIP mapping consists of a set of images each of which represents the original image at increasingly reduced levels of detail. Given an original image of 64x64 the MIP map sets would be 32x32, 16x16, 8x8, 4x4, 2x2, 1x1. This enables different MIP map levels to be chosen dependant on the level of detail required. This has the effect of reducing rendering time, because there is less processing to be done on the smaller sets compared to a full size texture. MIP maps also significantly decrease artefacts because the MIP map sets are already partially anti-aliased by the scaling process needed to create them.

Scaling of MIP maps is done by a process of filtering and decimation whereby each MIP map level is derived from the one directly above it, working from the highest resolution of the original image to the lowest resolution of the 1x1 MIP map. This process eliminates a great deal of aliasing. However, because most MIP mapping techniques include frequencies up to the Nyquist limit [63][64] for their resolution, other forms of aliasing are introduced, which creates the need for bilinear filtering. A similar problem also occurs with transitions between MIP map levels, which is where trilinear filtering [65] is useful.

In its simplest form MIP mapping picks the appropriate MIP map level based solely on which of the levels is nearest to the required number of pixels. For example, if a textured object in a scene required a texture of 50x50 then the 64x64 MIP map would be chosen, this is known as nearest neighbour MIP mapping. This does however cause a problem as illustrated by Figure 38, which shows the boundaries of each level of detail becoming visible when MIP mapping is enabled. This is caused by the sharp cut off between levels which occurs with nearest neighbour MIP mapping. As such MIP mapping is much more effective when combined with more sophisticated algorithms for handling level selection, such as trilinear filtering. Trilinear filtering gives a weighted average to MIP maps between levels, taking into account the results from both adjacent MIP maps and smoothing the transition between levels.

There is an increased storage size required for MIP mapping which, assuming texture compression is not used, equates to approximately 1.33 times the original texture size. While this is not a problem for modern consumer level hardware which tend to be designed around MIP maps, this is not an ideal situation in terms of traditional memory management techniques.

Advantages	**Disadvantages**
Speeds up rendering compared to flat texturing	Increased memory requirements
Decreases artefacts	Does not remove all aliasing
	Can over blur image causing loss of detail
	Can under sample the image causing aliasing
	Requires some pre-processing

Table 2: MIP mapping: advantages and disadvantages

4.2.2 Bilinear Filtering

Bilinear filtering [52] or bilinear interpolation is one of the simplest techniques commonly used to improve basic MIP mapping. Bilinear filtering can be used on its own with full detail textures, or as a magnification filter in conjunction with MIP mapping. Calculation of a pixel value is achieved by linearly interpolating the four texel values nearest to the point the pixel represents. This requires linear interpolations in the x and y directions (hence its name) which must then be combined to give the final pixel value.

Mathematically the bilinear filtering pixel value calculations can be expressed as follows:

$$c_{top} = t_{00} \cdot (1 - w_x) + t_{10} \cdot w_x$$
$$c_{bottom} = t_{01} \cdot (1 - w_x) + t_{11} \cdot w_x$$
$$c = c_{top} \cdot (1 - w_y) + c_{bottom} \cdot w_y$$

where:

t_{00} = Top left texel.

t_{10} = Top right texel.

t_{01} = Bottom left texel.

t_{11} = Bottom right texel.

w_x = Weight in x direction.

w_y = Weight in y direction.

c_{top} = Linearly interpolated value for top texels.

c_{bottom} = Linearly interpolated value for bottom texels.

c = Final pixel value.

This derivation implies that bilinear filtering requires four texture fetches per pixel. This makes bilinear filtering theoretically more than 4 times slower than basic nearest neighbour texturing, even before taking into account the three linear interpolations required.

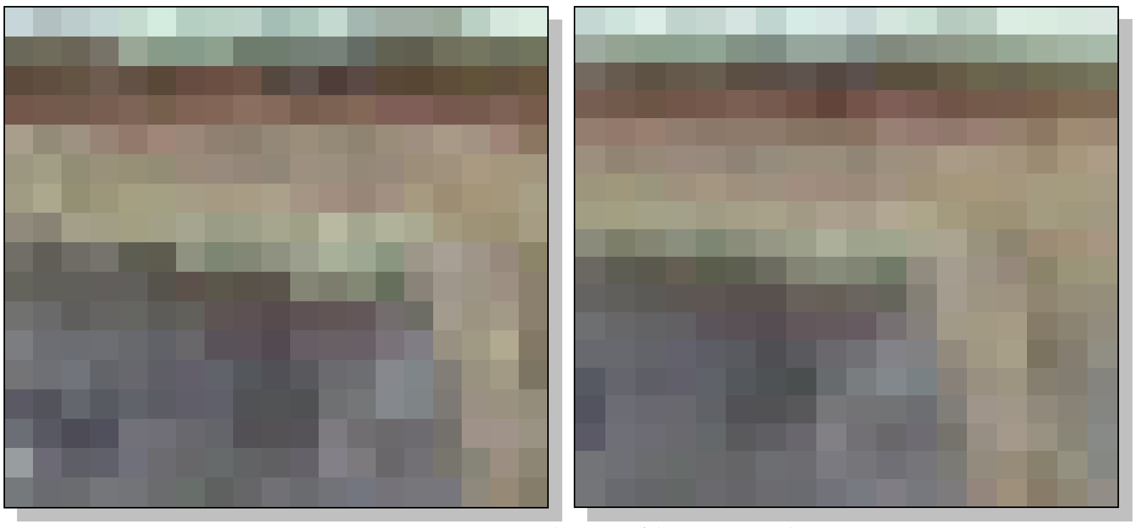


Figure 40: MIP mapping Vs Bilinear filtering with MIP mapping

Figure 40 gives an example of bilinear filtering with MIP mapping alongside basic MIP mapping. This clearly shows that bilinear filtering smooths out the image, resulting in a less pixelated final scene. Bilinear filtering however does not solve the problem of visible layer transitions shown in the earlier MIP mapping examples, as the abrupt change in blurriness is unaffected by this method of filtering. Again referring to Figure 40 it can be seen that bilinear filtering smooths out the texture, this has the effect of removing artefacts at the expense of some detail and sharpness.

Advantages	**Disadvantages**
Removes a great deal of aliasing compared to basic nearest neighbour texel selection	When used on a full detail texture scaled to a small size can cause accuracy problems and loss of detail
Can be used in conjunction with MIP mapping	Can under sample the image causing aliasing.
	Does not remove all artefacts
	Over blurs image
	When used in conjunction with MIP mapping clear distinctions between levels of detail are visible, caused by abrupt changes in blurriness. This can be particularly noticeable at steep angles

Table 3: Bilinear filtering: advantages and disadvantages

4.2.3 Trilinear Filtering

Trilinear filtering [65] is an extension of bilinear filtering into three dimensions to take into account adjacent levels of detail[32]. This implies that trilinear filtering may be used only in conjunction with techniques like MIP mapping, which rely on level of detail calculations. Bilinear filtering is performed for both levels, and the results are linearly interpolated using the fractional part of the level of detail as a weighting factor to produce the final trilinearly filtered pixel value. In other words, trilinear filtering consists of two bilinearly filtered MIP map levels with a linear interpolation between them, to smooth out the transition between the levels of detail.

32 Usually the two MIP map levels nearest to the current LoD, calculated from the Z depth values.

Whilst it is an improvement over bilinear filtering, trilinear filtering still suffers from a number of problems. Most notably trilinear filtering is much slower than bilinear filtering, as it requires twice the number of texture fetches and more than twice the amount of linear interpolations. This means trilinear filtering is at least twice as slow as bilinear filtering.

Trilinear filtering also suffers from the same accuracy problems as bilinear due to the assumption that the pixel occupies a square area on the texture, when in reality it is closer to a trapezium or an ellipse. Consequently this can cause loss of detail in the case where the texture is at a steep angle in relation to the viewpoint, such as that illustrated by Figure 41, and the pixel footprint is narrow and long. This means that in the long direction the pixel gets less detail than it should, as the square representation covers less texels than a trapezium or ellipse would, and in the narrow direction the square pixel gets information from more texels than it should.

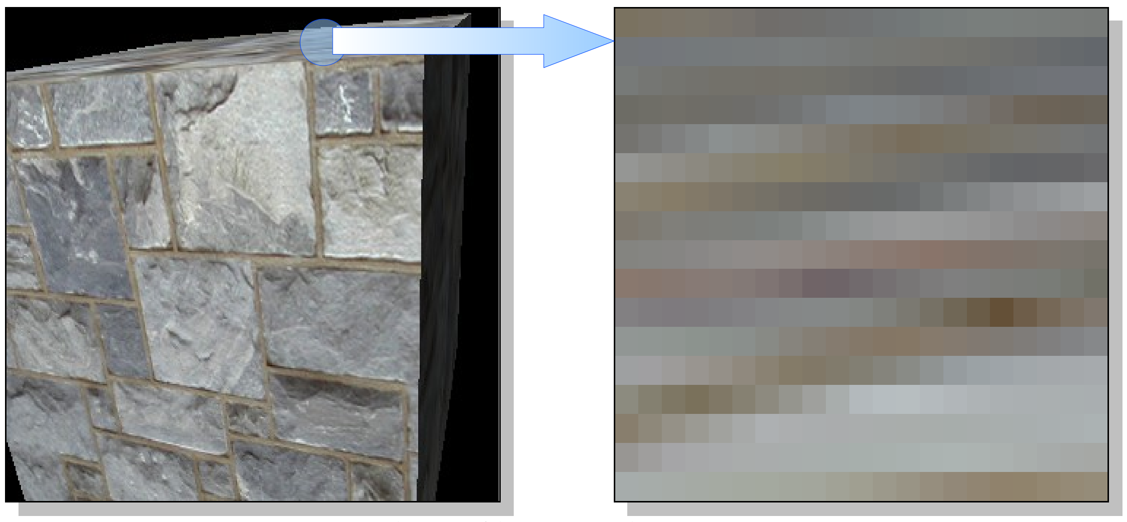

41: Trilinear filtering with MIP mapping

Figure 41 gives examples of trilinear filtering. This shows that trilinear filtering helps greatly with the problem of the visible layer transitions, which constitutes one of the main issues with MIP mapping and bilinear filtering. Figure 41 also demonstrates that while, just as bilinear filtering does, trilinear filtering smoothes out the front face removing much aliasing at the expense of some detail. It also smoothes out the other faces, at the expense of detail and some over-blurring but with the benefit of fewer artefacts. Some detail is lost on the steeper angles compared with pure MIP mapping and bilinear filtering.

Advantages	Disadvantages
Improves over bilinear filtering by making the transitions between levels of detail smoother	Assumes the pixel occupies a square area on the texture. This can cause loss of detail, particularly at steep angles
	Over blurs image
	Does not remove all artefacts

Table 4: Trilinear Filtering: advantages and disadvantages

4.2.4 “Brilinear” and “Trylinear” Filtering

The techniques known as “Brilinear” and “Trylinear” filtering are two different names for the what is effectively the same thing: heavily optimised trilinear filtering. The effect these techniques give is a hybrid of bilinear and trilinear filtering, sacrificing image quality in certain areas for the benefit of greater performance.

These techniques follow a growing trend in consumer level graphics hardware to sacrifice final image quality for improved performance generation on generation. This continues despite large increases in capabilities between hardware generations. This is because, within reason, frame rates in popular game X or benchmark test Y are used to sell graphics cards not overall image quality.

Over the years there has been a great deal of contention between the two main GPU firms, consumers, and the media over the use of such techniques. In the time of the GeForceFX for example NVIDIA advertised with the claims "Cinematic Computing" and "Engineered with passion for perfection" when in actuality the visual quality of the GeForceFX was in many ways inferior to some of its predecessors.

Early GeForce lines such as the 256 used full super-sampling, whereas more recent models like the FX utilise multi-sampling. Which is a more efficient but lower quality process. Similarly, older GeForce lines utilised full trilinear filtering, whereas the FX series did not allow full trilinear filtering. Instead the GeForce FX as well as successive series uses “brilinear” filtering to improve performance.

This “optimisation” was undertaken in the hope of beating rivals in the benchmark tests often found in hardware reviews, and hence selling more GPUs. This is not to say the newer hardware was not at least as capable of trilinear at reasonable speeds than previous models, just that NVIDIA's priority had shifted to speed over visual quality. A similar story is true of ATI who, with the R420 line of graphics chips (the X800 series) introduced facilities to automatically determine whether or not to use full trilinear or optimised trilinear (“Trylinear”) based on the content of the source MIP maps. This trend has continued to the current generation of cards.

Essentially what both techniques do is to decide, based on angle, level of detail and texture content, whether trilinear or bilinear filtering is more appropriate. Based on this, a decision is made to only perform bilinear filtering in the regions where trilinear is not deemed necessary. This results in sharper transitions between levels of detail but can dramatically improve rendering speed.

Much of the reason for the switch to these hybrid methods is to do with the architecture of modern graphics cards, whose texture units supply only one bilinear sample per clock cycle. Thus in order to realise trilinear filtering either two texture units or two clocks are required. Therefore, in theory, trilinear filtering halves the fill rate when compared against bilinear filtering. So by minimising the use of trilinear filtering and maximising the use of bilinear filtering these hybrids can theoretically save a lot of work for the GPU.

An example taken from Tom's Hardware [67], which clearly illustrates the differences between Bilinear, “Brilinear” and Trilinear, is shown in Figure 42. The transition between levels is sharp and obvious in the case of Bilinear, “Brilinear” shows a rapid but smoother transition and Trilinear shows a much more gradual transition between levels. The increased size of the coloured areas when compared to trilinear filtering are symptomatic of the “Brilinear” techniques optimisations, and these are the areas in which only bilinear filtering is performed. This means that with “Brilinear”, much like

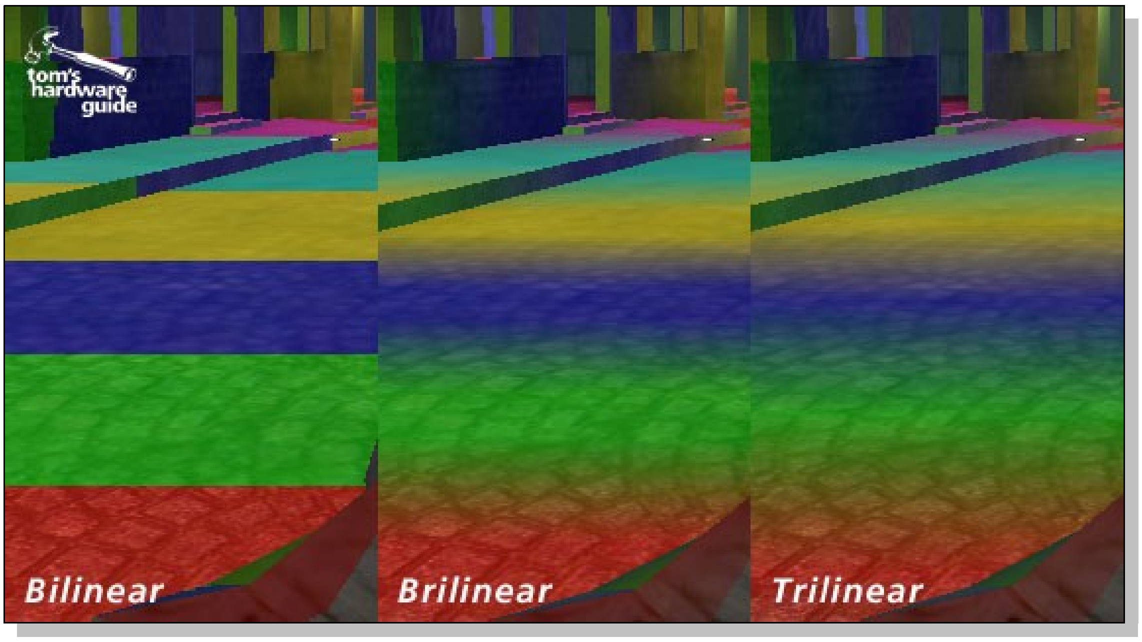


Figure 42: Bilinear, "Brilinear" and Trilinear Filtering [67]

Bilinear, it is easier to spot the level of detail transitions. This can become very noticeable in something like a game or simulation, as one side effect of sharper level transitions is the appearance of bow waves. These bow waves move as the view point changes, making them very noticeable in non-static scenes.

Advantages	**Disadvantages**
Gives significant theoretical performance improvements over full trilinear filtering	Reduced visual quality compared to pure trilinear filtering
Fewer artefacts than bilinear filtering for little extra cost when judged against the comparatively high costs of full trilinear filtering	Often results in a reduced LOD selection (smaller MIP map) to improve performance further, at the expense of greater levels of blurring
	Increased visibility for level of detail transitions over full trilinear filtering

Table 5: "Brilinear" / "Trylinear" filtering: advantages and disadvantages

4.2.5 RIP Mapping

RIP mapping is a basic form of anisotropic filtering[33] which extends MIP mapping to include anisotropically down-sampled images, which can be probed similarly to basic MIP mapping.

RIP maps are essentially non-uniform MIP maps, meaning the extra factor of irregular texture co-ordinates need to be taken into account to calculate the different RIP map selections. Given a 128x128 original image, samples could be taken from many different RIP maps including a 256x128, 128x128, 64x128 or 32x128 RIP map, dependant on each texture axis. Because of this RIP mapping is restricted to axis aligned anisotropic probes and a rectangular filtering pattern. RIP mapping also suffers from much larger memory requirements than the equivalent MIP maps due the the extra maps needed for the non power of two anisotropic maps. This is why RIP mapping remains largely unused on consumer hardware, where texture memory and bandwidth are at a premium and hardware is often optimised solely for power of two textures as commonly found in MIP maps.

Advantages	**Disadvantages**
Fast anisotropic filtering	Only supports axis aligned anisotropy
Similar in principal and execution to the widely adopted MIP mapping	Has massive memory requirements compared to other techniques

Table 6: RIP Mapping: advantages and disadvantages

33 See section 4.2.6

4.2.6 Anisotropic Filtering

Technically speaking there are many varieties of anisotropic filtering, such as RIP Mapping, and EWA[34] [68][69] etc. This is because anisotropic filtering simply refers to any method where the filtering method probes the texture with a non square, or anisotropic, filtering pattern. As illustrated by Figure 37 this can often be narrow and long in texture space. This is to take into account the fact that a single screen pixel may encompass data from multiple texels, often with a greater proportion being in one axis, meaning anisotropic filters allow proper preservation of perspective. This enables greater detail retention in the final screen image, particularly at sharper angles where the problem is accentuated. However, for the purpose of this discussion it will be assumed that anisotropic filtering refers to the variations implemented in current consumer level hardware by NVIDIA and ATI, all other relevant anisotropic methods shall be discussed individually under their respective full titles.

Very little is known publicly about the algorithms used on either ATI or NVIDIA's hardware, as both are extremely secretive about the exact techniques and optimisations they use. However Smith [48] offers a discussion of both companies anisotropic techniques and provides enough information to allow, with some assumptions and deduction, a degree of understanding of both approaches to anisotropic filtering.

The current maximum setting for anisotropic filtering on the main test hardware (NVIDIA GeForce 6800) is 16x, this has remained constant on a number of recent

34 **E**lliptical **W**eighted **A**verage, which is discussed in section 4.2.7.

generations of NVIDIA cards[35]. This means that there are 16 anisotropic samples per pixel, or 128 texture samples per pixel when used in conjunction with trilinear texture filtering. Because trilinear filtering requires two times bilinear filtering, or 2 times 4 samples, giving 8 samples for trilinear, meaning a total of 8 texture samples per anisotropic sample (also known as a tap).

However, both ATI and NVIDIA utilise adaptive anisotropic filtering, meaning that the filtering is applied to varying degrees on different sections of the scene. This is dependant on selections made by their algorithms. This makes it difficult for an outside observer to accurately calculate the potential performance of such an algorithm, although approximations are possible via experimentation. It is also difficult to use the number of samples as a comparison against other techniques, as the total number of samples is dependant on both scene and viewing angle. It can however be reliably said that both implementations, at a setting of 16x, will have a maximum of 128 texture samples per pixel.

As previously mentioned, ATI and NVIDIA both vary the use of anisotropic filtering dependant on the angle of the surface relative to the eye position. This means they use a longer sample area on sharper angles resulting in more samples. However, both companies take different approaches to sample footprint. ATI sample a rectangular area, while NVIDIA on the other hand sample a variable polygonal area. The NVIDIA sample footprint changes shape dependent on the degree of slope related distortion on

35 Although greater levels are possible with Sli (multiple cards running in parallel) or NVIDIA's workstation level Quadro cards.

the x and y axes. This means the NVIDIA method can dynamically vary the number of samples as well as the samples footprint shape and aspect ratio, whereas the ATI hardware can only vary the number of samples and the aspect ratio of the sample area. NVIDIA also utilise a non-linear weighting factor, taking more samples closer to the eye and fewer as the viewing distance increases. This is to improve performance by sacrificing image quality where the differences are less noticeable. This means that the NVIDIA implementation could theoretically provide better performance and/or better image quality dependant on the optimisation settings employed. Thus NVIDIA's technique allows greater flexibility to tailor the filtering to suit the needs of the application.

Neither ATI nor NVIDIA's anisotropic techniques are true implementations of anisotropic filtering, as both use optimisations to increase performance at the expense of visual quality. This almost certainly results in less detail being retained by the final image and will allow aliasing where there would be none in a more complete anisotropic solution.

Figure 43 gives examples, taken from Tom's Hardware [67], of a variety of different filtering techniques which are common on today's consumer level graphics hardware. The advantages of anisotropic filtering are plain to see from the example. Unfortunately there is a significant performance cost involved, even with the heavy optimisations used by NVIDIA and ATI.

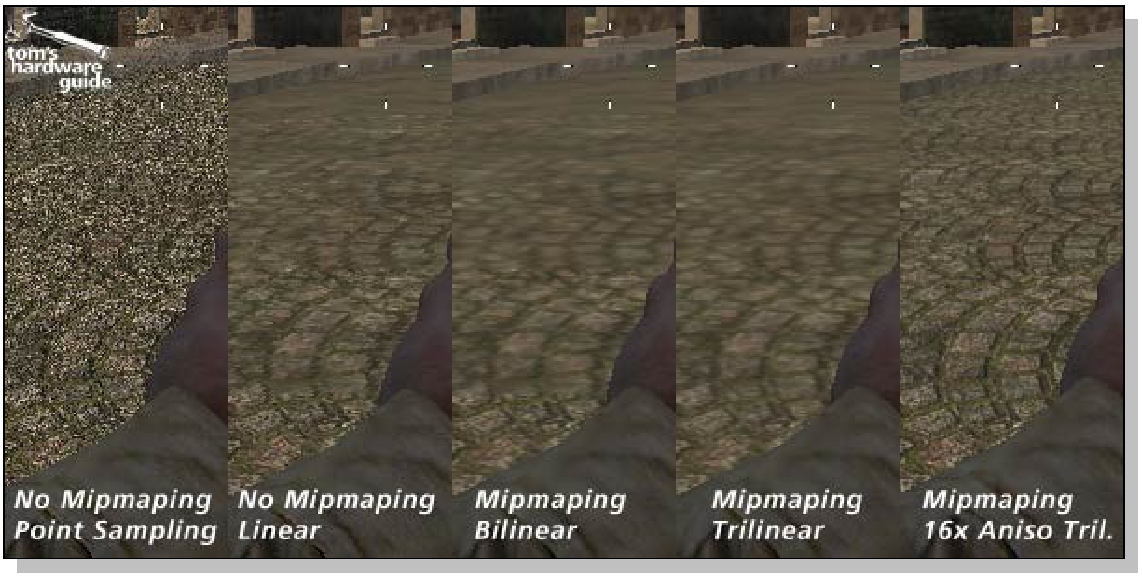


Figure 43: The Anisotropic Advantage [67]

What is not clear from the images in Figure 43 is that aliasing still occurs, even at 16x anisotropic filtering, when the scene is in motion. It is worth noting that even without the optimisations this would still be the case unless many more samples are taken. This is due to the inaccurate representation of a pixel as a rectangle or four sided polygon at the sampling stage.

Advantages	**Disadvantages**
The techniques fit in well on current consumer level hardware and can be used in conjunction with either bilinear or trilinear filtering to improve the final image. Although trilinear should be the preferred choice to minimise artefacts and aliasing	Full implementation details are not known and so the most used methods on consumer hardware are difficult to compare like for like with other techniques, for example slight differences in texel size and orientation calculations could make massive differences to the final image
Optimisations provide better performance than using the full range of samples for every pixel	Optimisations make it difficult to properly compare performance against other techniques
	Optimisations have a negative impact on aliasing and image detail
	Neither implementations representation of a pixel as a rectangle or four sided polygon are strictly accurate

Table 7: Anisotropic filtering: advantages and disadvantages

4.2.7 Elliptical Weighted Average

Elliptical Weighted Average or EWA [68][69] was first proposed by Heckbert and Greene in 1986. It is a method of anisotropic texture filtering whose central premise is that pixels are not square but circular, or more precisely they are points with a fall off similar to the Gaussian function illustrated in Figure 32 b) on page 64. Therefore this leads to the conclusion that the pixel footprint in texture space is not rectangular or polygonal, as in ATI and NVIDIA's anisotropic filtering techniques, but is in-fact an ellipse. The EWA ellipse is formulated as:

$$d^2(u,v)=Au^2+Buv+Cv^2$$

Where the biquadric co-efficient's for computing d^2 are:

$$A_{nn}=\left(\frac{dv}{dx}\right)^2+\left(\frac{dv}{dy}\right)^2$$
$$B_{nn}=-2\times\left(\frac{du}{dx}\times\frac{dv}{dx}+\frac{du}{dy}\times\frac{dv}{dy}\right)$$
$$C_{nn}=\left(\frac{du}{dx}\right)^2+\left(\frac{du}{dy}\right)^2$$

and

$$F=A_{nn}\times C_{nn}-\frac{B_{nn}^2}{4}$$
$$A=\frac{A_{nn}}{F}$$
$$B=\frac{B_{nn}}{F}$$
$$C=\frac{C_{nn}}{F}$$

[68][69][70][71][72]

The value of d^2 represents the distance from the centre of the pixel squared, when the texel position is mapped back to screen space [70][71]. This means d^2 may be used as an index to a table of Gaussian weights, unrelated to the affine projection but dependant on the pixel filter.

EWA calculates d^2 for every texel in or near the elliptical footprint, where texels with a value of $d^2 \leqslant 1$ are considered to be within the pixel footprint. Hence they are sampled, weighted, and accumulated, while those with a value of $d^2 > 1$ are discarded. The result is then divided by the sum of the weights which gives the elliptical filters volume in texture space.

McCormick et al [70] describe EWA as “the best software anisotropic texture filtering algorithm known to date”, “the most efficient direct convolution method known for computing a textured pixel” and say that the technique “provides a quality benchmark against which to compare other techniques”. Shin et al [71] say that EWA “generates the very high quality images, but requires the intensive computation power and texel values. This method provides a quality benchmark used when to compare various filtering techniques”. Therefore it can be concluded that EWA may be used as a quality benchmark for comparison with other techniques, and can be held up as an exemplar of quality for anisotropic techniques.

EWA has two main drawbacks: firstly it could be argued that EWA's visual quality could be sharper than its Gaussian filter allows. It is argued that other filters are able to produce sharper images without introducing significantly more aliasing artefacts [73]. However, as pointed out by Lansdale [74] and McCormick et al [70] none of these filters are as easily handled, mathematically, as the Gaussian for unifying the reconstruction filter and projected pixel filter. EWA's second major drawback is the sheer number of operations it requires, which to date have precluded a real-time hardware implementation.

Advantages	**Disadvantages**
Widely regarded as the pinnacle of anisotropic techniques for visual quality	Currently impractical for real-time use due to the large number of operations required to implement
	Images could be shaper than the Gaussian filter allows

Table 8: EWA: advantages and disadvantages

4.2.8 Texram

Texram is a variety of footprint assembly which achieves high performance via a logic-embedded, highly-integrated, dynamic memory device [75]. Providing a higher visual quality than trilinear texture filtering but without the computational expense of methods like EWA. As Texram requires specialist hardware that does not fit in with the current standard graphics pipeline it does not make an appropriate comparison to many of the other techniques discussed, however it deserves a special mention due to the technique's influence on Feline[36].

McCormack et al [70] state that Texram uses a series of trilinear filter probes along a line, which approximates the length and slope of the major axis of EWA's elliptical footprint. It was considered too costly to implement the computation of the full ellipse parameters in hardware, so simplified approximations were proposed. The approximations actually underestimate the length of the ellipses major axis which introduces blurring, they also deviate slightly from the slope of the major axis, again causing blurring along with some aliasing. These errors are particularly evident when Texram is used in conjunction with environment mapping, but are however “visually insignificant under typical perspective projections” [70].

36 Discussed in section 4.2.9.

Advantages	Disadvantages
Provides comparable results to EWA under typical perspective projections for significantly lower computational cost	Requires specialist hardware not commonly found on consumer level graphics cards
	Approximations of the ellipses major axis can introduce blurring and aliasing

Table 9: Texram: advantages and disadvantages

4.2.9 Feline

Feline is a hybrid method which attempts to bring together the best elements that Texram and EWA have to offer. Feline stands for Fast Elliptical Lines and was proposed by McCormack et al [70] in 1999. Unlike EWA, Feline is an isotropic technique which utilises multiple isotropic probes to mimic the shape of the EWA filter. Feline can also be simplified to use axes approximations, similar to those employed by Texram, to further enhance performance.

McCormick et al [70] claim that Feline compares well against EWA which they term “the best software anisotropic texture filtering algorithm known to date”. This is because Feline provides similar levels of image quality to EWA with performance closer to that of Texram. The main advantage that Feline holds over EWA is that it requires significantly less set-up computation and fewer cycles for texel fetches. Feline improves over Texram as it uses standard MIP-maps, therefore it does not require specialist hardware and minimal extensions are required to standard 3D interfaces, such as OpenGL, to implement it [70].

Advantages	**Disadvantages**
Achieves higher visual quality than Texram with little additional logic	Has substantially higher computational set up costs than Texram
Images generated by Feline filtering, are much sharper and exhibit fewer Moire artefacts than images which have been developed using trilinear filtering	Requires more advanced filters, such as Lanczos, to match the image quality of EWA which impacts heavily on Feline's cost
By using a better filter, such as Lanczos, Feline is able to create mip-maps which display fewer artefacts than EWA	
The associated set up costs are substantially lower than mip-mapped EWA	

Table 10: Feline: advantages and disadvantages

4.2.10 Clamping

Clamping is a method proposed in 1982 by Norton, Rockwood & Skolmoski [76]. This technique is based on bandwidth limiting in object space, it selectively suppresses frequencies using a power series approximation of a box filter.

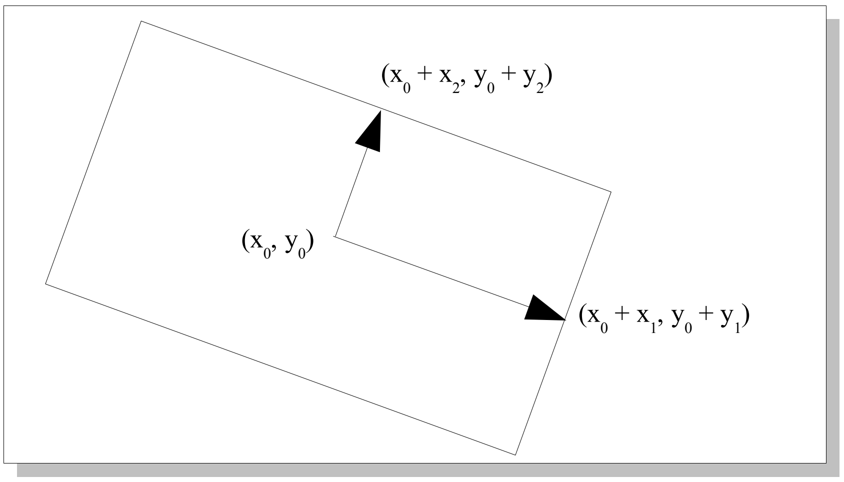


Figure 44: "Parallelogram filter" in object space [76]

A simple box filter is used to suppress high frequencies, Norton et al. note that because of this the perspective transformation to object space becomes trivial. This perspective transformation is logically approximated by a linear transformation, meaning that the box filter translates to the parallelogram in object space as illustrated in Figure 44. Taking Norton's mathematics verbatim, this means that the points within the parallelogram can be parametrised by:

$$(x_{0,} y_0)+s(x_{1,} y_1)+t(x_{2,} y_2)$$

where

$$-1<s,t<+1$$

and the sides of the parallelogram have length and direction which can be calculated by:

$$2\cdot(x_{1,} y_1) \text{ and } 2\cdot(x_{2,} y_2)$$

The average value of $I(x,y)$ is calculated by:

$$1/4\int_{-1}^{+1}\int_{-1}^{+1} e^{(ik(x_0+sx_1+tx_2)+il(y_0+sy_1+ty_2))}dsdt$$

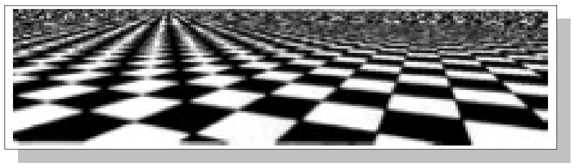
This allows the clamping function (C) to be defined by:

$$C=\left\{\begin{matrix} e^{(ikx_0+ily_0)}(1-r\cdot((kx_1+iy_1)^2-(kx_2+iy_2)^2)) \\ \ldots if\ r\cdot((kx_1+ly_1)^2+(kx_2+ly_2)^2)<1 \\ 0\ \ otherwise \end{matrix}\right\}$$

Which amounts to multiplying $e^{(ikx_0+ily_0)}$ by

$$C(x_1,y_1,x_2,y_2,k,l)=max(0,1-r((kx_1+iy_1)^2+(kx_2+iy_2)^2))$$

To aid computation Norton et al. initially use an approximation to the sinc function from


a) Point Sampling

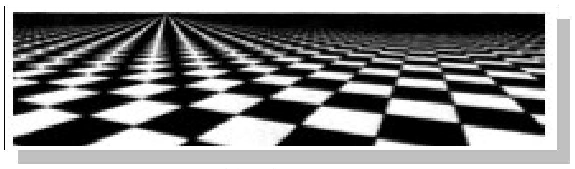

b) Clamping

Figure 45: Clamping comparison images

its low power series. They show that this is suitable for parallelograms which cover a small region of the texture. For larger regions, it becomes necessary for the frequency to be multiplied by a suppression factor which “clamps” the result of the sinc function between the values of 0 and 1.

Figure 45 shows a comparison between a chequerboard texture which has been point sampled and one which has had its high frequency terms clamped. The clamped image shows that high frequencies have been suppressed, thus reducing aliasing when compared to the point sampled image.

There is one major drawback to the clamping method, it only works for textures which are defined as a set of Fourier terms. These terms can be obtained either by Fourier analysis or synthesised directly by sums of sine and cosine waves. This restricts the variety of textures which can be antialiased by this method.

Advantages	**Disadvantages**
Significantly reduces aliasing	Clamping only works for textures which are defined as a set of Fourier terms

Table 11: Clamping: advantages and disadvantages

4.2.11 Texture Potential MIP Mapping

Texture Potential MIP Mapping, or TPM presents a hybrid approach combining Cant and Shrubsole's Texture Potential Mapping [66] with industry standard MIP mapping techniques in an effort to aid efficiency and performance.

Cant and Shrubsole [66] propose a system which follows the form of each projected pixel faithfully. Antialiased textures are produced by taking an average intensity over all the texels that lie within the pixel footprint. This is achieved by integrating the texture within the pixel footprint by taking into account only those values that lie around the edges of the pixel footprint, in a similar manner to Gauss' theorem in physics. Standard MIP mapping takes over in the non-summed direction to reduce overheads at the expense of some loss in quality.

Advantages	**Disadvantages**
Compares favourably against competing algorithms such as footprint assembly [75]	Typically requires 1.5 times the memory of the original texture pattern, compared to 1.33 for conventional MIP mapping alone

Table 12: TPM: advantages and disadvantages

4.2.12 Conclusions

As can be seen from the above much research and development has been expended in the area of texture filtering since the advent of texture mapping. The current best practice, widely used and seminal techniques have been investigated, resulting in the conclusion that no one technique currently provides a complete solution to the texture aliasing problem.

It is generally accepted that EWA is the current best practice technique for overall texture quality, but this has to date precluded a real-time implementation. The GPU vendors "Anisotropic" filtering techniques are the most widely used currently, but these provide very little impact on aliasing when compared to more advanced techniques like EWA and Feline.

As such it must be concluded that there is much work still to do in this area and that any research aiming to improve texture quality in real-time systems can only add to the knowledge of the community.

4.3 Fourier Textures

As previously discussed, to date the majority of texture filtering techniques deal solely with treating the symptom of aliasing and artefacts. Therefore, most techniques largely ignore the root cause of these effects, which are commonly a product of under-sampling.

Under-sampling can be described as: the sampling of frequencies above the image's Nyquist limit, which is a limit of at least twice the maximum frequency of the source signal. This is as defined by the sampling theorem [63][64]. An ideal solution would be to simply provide enough samples to overcome aliasing, however in computer graphics this is often an impossible task to achieve, as monitor resolutions are a limiting factor for sampling frequency, and so some degree of aliasing is, to a large extent inevitable.

It is proposed that filtering should be carried out at the pre-processing stage to remove the troublesome frequencies which fall above the range of the original image's Nyquist limit [63][64].

Fourier Transforms [77] are used to convert data into frequencies. This can be used to convert image space data into Fourier space, giving the image in terms of frequencies rather than colour data. This should not be done for all types of data but only for data where an analysis of frequencies is appropriate and meaningful.

It can be shown that, by filtering out weaker frequencies and retaining only a set of more dominant frequencies, an image can be recreated to a sufficient visual quality by a reverse Fourier transform on this sub-set of image data. In addition it is suggested that the frequencies-based representation in Fourier space will allow much easier and more effective filtering of an image, thus allowing frequencies which fall above the range of the original image's Nyquist limit to be easily identified and removed, and hence reducing aliasing.

4.3.1 Fourier Textures Technique

In order to assess the validity of the Fourier texture filtering theory, the practicalities of representing images using only a subset of their frequencies need to be established. Thus, a software based testing environment is required for this purpose. Written using C++ and Windows GDI+, the testing environment initially performs a Discrete Fourier Transform (DFT) on an image to obtain its representation in Fourier space.

Originally, Fast Fourier Transforms (FFT) were investigated for this task, but after this consideration it was decided that they are too costly in terms of implementation time and flexibility. There are many variations of FFT, such as those described by Smith [48], each of which has specialist applications, advantages and drawbacks. For example, some are limited to power of two values for N. While FFT's do provide considerably better performance[37] they are much more complicated to implement, and suffer from a number of limitations when compared to DFT's. DFT's are also less constricted, simpler and easier to control, this is particularly important here given the need to control frequency content, in order to allow proper evaluation of the technique.

The next stage is to order the frequencies by magnitude and to retain only the top *n* frequencies, while filtering out those frequencies which fall above the range of the

37 (N^2) compared to $O(N.logN)$

original image's Nyquist limit. These filtered frequency sets are then supplied as parameters to a reverse DFT resulting in the set of frequencies being transformed back into image space. This proves useful in attempting to discover how many frequencies (n) it takes to represent a texture to a standard either indistinguishable from the original to the naked eye, when displayed on a computer monitor, or at least to a level that makes a reasonable approximation of the original image.

The most difficult part of this process is the reconstruction and filtering of an image from Fourier space, which it is intended will eventually be performed via hardware accelerated vertex and fragment shaders.

Initially, as a proof of concept, the process shall be investigated via the use of procedurally generated textures to attempt to ascertain whether or not the technique reduces aliasing as anticipated. This approach is an extension of the work by Cant and Shrubsole [66] and is based on the mathematical work of Richard Cant, this is as follows:

Let the original texture pattern be defined as:

$$\tau^{A}(x_T)$$

Let the original pixel filter be:

$$\rho(x_S)$$

Where x_S is a position vector relative to the centre of the pixel.

The sampled value for a particular point is now defined as:

$$\phi(x_0)=\int d^2 x_S \rho(x_S-x_0)\tau(x_T(x_S))$$

For a particular pixel with the screen co-ordinates i, j this becomes:

$$\phi_{ij}=\int d^2 x_S \rho(x_S-x_{ij})\tau(x_T(x_S))$$

If it is assumed that:

$$x_R=x_S-x_{ij}$$

Therefore, it must follow that:

$$\phi(x_{ij})=\int d^2 x_R \rho(x_R)\tau(x_T(x_R+x_{ij}))$$

So assuming that, in the region where ρ is significantly larger than zero, the transform between the screen coordinates and texture coordinates can be adequately approximated by a linear transform x_T, with some matrix M describing it:

$$x_T(x_R+x_{ij})=x_T(x_{ij})+M\,x_R$$

The variation in perspective at this point is minimal so it is safe to assume linearity for the transform. This transformation is built from the derivatives of the texture co-ordinates with respect to the screen co-ordinates, which are directly obtainable under the fragment shader and hence should allow implementation in hardware.

This gives:

$$\phi(x_{ij})=\int d^2 x_R \rho(x_R)(\tau(x_T(x_{ij})+M\,x_R))$$

Next formulate the texture in Fourier transform form:

$$\tau(x)=\int d^2p\,\tilde{\tau}(p)\mathrm{e}^{ip\cdot x}$$

$$\phi_{ij}=\int d^2p\int d^2x_R\,\rho(x_R)\,\tilde{\tau}(p)\mathrm{e}^{ip\cdot(x_T(x_{ij})+M\,x_R)}$$

Thus:

$$\phi_{ij}=\int d^2p\,\tilde{\tau}(p)\mathrm{e}^{ip\cdot(x_T(x_{ij}))}\int d^2x_R\,\rho(x_R)\,\mathrm{e}^{ip\cdot M\,x_R}$$

Let:

1. $$\rho(x_R)=\frac{1}{N}\mathrm{e}^{-\frac{x_R\cdot x_R}{2\sigma^2}}$$

Which is a Gaussian, including the normalisation (N) factor to ensure that:

$$\int d^2x_R\,\rho(x_R)=1$$

So now including the frequency space variable p , this gives:

$$\phi_{ij}=\frac{1}{N}\int d^2p\,\tilde{\tau}(p)\,\mathrm{e}^{ip\cdot(x_T(x_{ij}))}\int d^2x_R\,\mathrm{e}^{ip\cdot M\circ x_R-\frac{x_R\cdot x_R}{2\sigma^2}}$$

It is noted that:

$$-\frac{1}{2\sigma^2}(\sigma^2 ip\cdot M-x_R)^2=\frac{\sigma^2}{2}(p.M)^2+ip\cdot M\cdot x_R-\frac{x_R\cdot x_R}{2\sigma^2}$$

Completing the square in the exponent will give:

$$\phi_{ij}=\frac{1}{N}\int d^2p\,\tilde{\tau}(p)\,\mathrm{e}^{ip\cdot(x_T(x_{ij}))-\frac{\sigma^2}{2}(p\cdot M)^2}\int d^2x_R\,\mathrm{e}^{-\frac{1}{2\sigma^2}(\sigma^2 ip\cdot M-x_R)^2}$$

As the integration over x is infinite the “centre” of the integration can be shifted by changing variables to:

$$y=(\sigma^2 ip\cdot M - x_R)$$

Without changing the value of the integral.

The integration over y (formerly x) will cancel N to leave:

$$\phi_{ij}=\int d^2 p\,\tilde{\tau}(p)\,e^{ip\cdot(x_T(x_{ij}))-\frac{\sigma^2}{2}(p\cdot M)^2}$$

Now if the texture consists of a discrete set of frequencies p_k each with an amplitude τ_k this reduces to a discrete sum:

2. $$\phi_{ij}=\sum \tau_k e^{ip_k\cdot(x_T(x_{ij}))}e^{-\frac{\sigma^2}{2}(p_k\cdot M)^2}$$ [38]

Note that the suppression factor[38] "belongs" to the individual component in the sum and must be applied individually.

Everything thus far has been done in complex number form, as it is more convenient for calculation. However, it is now required to take the real part of everything so:

$$e^{ip_k\cdot(x_T(x_{ij}))}$$

becomes:

$$\cos(p_k\cdot(x_T(x_{oj})))$$

It is noted that the suppression factor will not change if the phase of the sine wave changes:

$$\cos(p_k\cdot(x_T(x_{oj})))\rightarrow\cos(p_k\cdot(x_T(x_{oj}))+\phi)$$

38 Suppression factor $e^{-\frac{\sigma^2}{2}(p_k\cdot M)^2}$ relates to the anti aliasing factor applied in section 4.3.2 onwards.

If the frequency variations are small then the change will also be small. So a slow variation of phase across the texture will not affect things too much as it will only introduce frequencies near the base frequency. This effect has been exploited previously by things such as the Narrow Band Noise Model [78]. In fact it should be possible to get away with a slow variation of frequency and amplitude too, as is common practice when dealing with FM and AM radio signals. This can be exploited to produce more "interesting" textures without needing a lot of extra frequency components.

Now split the p integral into two with the dividing point p_0 where:

$$p_0 M x_R \approx 0$$

For all values of x_R such that:

$$\rho(x_R) \neq 0$$

p_0 is such that it does not change significantly over the pixel footprint, which is defined by $\rho(x_R)$. It is thus defined by the "large" axis of $\rho(x_R)$.

$$\phi_{ij} = \int_{p<p_0} d^2 p \tilde{\tau}(p) e^{ip\cdot(x_T(x_{ij}))} + \int_{p<p_0} d^2 p \tilde{\tau}(p) e^{ip\cdot(x_T(x_{ij}))} \int d^2 x_R \rho(x_R) e^{ip\cdot M x_R}$$

The first term on the right hand side gives the traditional MIP Map contribution.

Now define p_1 such that:

$$|p_1 M| x_{RM} > \pi$$

Where x_{RM} is the radius of the circle within which ρ [39] is greater than zero. It follows

39 See equation 1.

that the x_R integral will be zero for values of p greater than p_1 because the integrand is oscillating over more than one cycle in that region. p_1 is thus defined by the "small" axis of $\rho(x_R)$.

This leaves the following:

$$\phi_{ij} = \int_{p<p_0} d^2p\,\tilde{\tau}(p)\,\mathrm{e}^{ip\cdot(x_T(x_{ij}))} + \int_{p_1>p>p_0} d^2p\,\tilde{\tau}(p)\,\mathrm{e}^{ip\cdot(x_T(x_{ij}))} \int d^2x_R\,\rho(x_R)\,\mathrm{e}^{ip\cdot M x_R}$$

Now write the integral:

$$\tilde{\rho}(p\cdot M) = \int d^2x_R\,\rho(x_R)\,\mathrm{e}^{ip\cdot M x_R}$$

And so:

$$\phi_{ij} = \int_{p<p_0} d^2p\,\tilde{\tau}(p)\,\mathrm{e}^{ip\cdot(x_T(x_{ij}))} + \int_{p_1>p>p_0} d^2p\,\tilde{\tau}(p)\,\mathrm{e}^{ip\cdot(x_T(x_{ij}))}\tilde{\rho}(p\cdot M)$$

Unfortunately this list of frequencies from p_1 to p_0 is likely to be quite long so a bit more work in x space is required first, so go back to the equation:

$$\phi_{ij} = \int d^2x_R\,\rho(x_R)\,\tau(x_T(x_{ij}) + M x_R)$$

And expanding the texture function locally around each pixel:

3. $$\tau(x_T(x_{ij}) + M x_R) = \tau(x_{Tmn} + (x_T(x_{ij}) - x_{Tmn}) + M x_R)$$

4. $$\tau(x_T(x_{ij}) + M x_R) = \tau_{mn} + \tau_{Rmn}(x_T(x_{ij}) - x_{Tmn}) + M x_R$$

Substituting equation 3 into the integral gives:

$$\phi_{ij} = \int d^2x_R\,\rho(x_R)\,\tau_{mn} + \int d^2x_R\,\rho(x_R)\,\tau_{Rmn}(x_T(x_{ij}) - x_{Tmn}) + M x_R$$

Now concentrate on the second term in this equation, since the first is just the appropriate MIP Map entry.

Moving to Fourier transform representation gives:

$$\delta \phi_{ij} = \int d^2 p \int d^2 x_R \rho(x_R) \tilde{\tau}_{Rmn}(p) \mathrm{e}^{ip\cdot(x_T(x_{ij}) - x_{Tmn} + M x_R)}$$

or

$$\delta \phi_{ij} = \int d^2 p \tilde{\tau}_{Rmn}(p) \mathrm{e}^{ip\cdot(x_T(x_{ij}) - x_{Tmn})} \int d^2 x_R \rho(x_R) \mathrm{e}^{ip\cdot M x_R}$$

Now $\tilde{\tau}_{Rmn}(p)$ will only contain wavelengths shorter than the spacing between points in the MIP Map, so there is no need to set a lower bound p_0. Choosing p_1 as before:

$$\delta \phi_{ij} = \int d^2 p \tilde{\tau}_{Rmn}(p) \mathrm{e}^{ip\cdot(x_T(x_{ij}) - x_{Tmn})} \tilde{\rho}(p\cdot M)$$

This way the technique can either be applied to an existing MIP mapping system or used in conjunction with its own procedurally generated textures such as those shown in Figure 47. Figure 46 shows the HLSL shader code used to create the images in Figure 47, the variable names relate to the mathematics shown above. The shader code takes in a number of parameters, including: the colours to use for the generated texture, x and y frequencies, phase, offsets, an amplitude and a filter width. These parameters are used to adjust the texture pattern created as well as the antialiasing factors.

```
float4 ps_main(PS_INPUT Input) : COLOR0 {
        const float PI  = radians(180);
        float2 Texcoord = (256 * PI) * Input.Texcoord;
        int k;
        float x= Texcoord.x + xoffset;
        float y = Texcoord.y + yoffset;
        float fT, fHk, fGk = 0.0f;
        float2 dtx = ddx(Texcoord);
        float2 dty = ddy(Texcoord);
        float2x2 mM = float2x2(dtx.x, dty.x, dtx.y, dty.y);
        float2 vP0qk, fpdotM0qk, vPpk0, fpdotMpk0, vPpkqk, fpdotMpkqk, vPpkminusqk,
        fpdotMpkminusqk;
        float fS0qk, fSpk0, fSpkqk, fSpkminusqk;
        // loop through the number of terms
        for(k = 0; k < NTERMS; k++) {
                vP0qk = float2(0, yfreq);
                fpdotM0qk = mul(vP0qk, mM);
                fS0qk = exp(-((fFilterWidth / 2) * dot(fpdotM0qk, fpdotM0qk)));
                vPpk0 = float2(xfreq, 0);
                fpdotMpk0 = mul(vPpk0, mM);
                fSpk0 = exp(-((fFilterWidth / 2) * dot(fpdotMpk0, fpdotMpk0)));
                vPpkqk = float2(xfreq, yfreq);
                fpdotMpkqk = mul(vPpkqk, mM);
                fSpkqk = exp(-((fFilterWidth / 2) *dot(fpdotMpkqk, fpdotMpkqk)))
                vPpkminusqk = float2(xfreq, -yfreq);
                fpdotMpkminusq = mul(vPpkminusqk, mM);
                fSpkminusqk = exp(-((fFilterWidth / 2) * dot(fpdotMpkminusqk, fpdotMpkminusqk)));
                fHk = 0.5 * (cos((xfreq * x) + (xfreq * xphase) - (yfreq * y) - (yfreq * yphase))) *
                fSpkminusqk + (cos((xfreq * x) + (xfreq * xphase) + (yfreq * y) + (yfreq * yphase)) *
                fSpkqk);
                fGk = (cos((xfreq * x) + (xfreq * xphase)) * fSpk0 + cos((yfreq * y) + (yfreq * yphase)) *
                fS0qk);
                fT += (pow(amplitude, 2)) * (pow( offset, 2) + offset * fGk + fHk);
                xfreq = 0.9 * xfreq;
                yfreq = 0.9 * yfreq;
        }
        fT = saturate(fT);
        float4 vTexColour = lerp(backColour, spectralColour, fT);
        return (vTexColour);
}
```

Figure 46: Rendermonkey shader code

4.3.2 Results

At this stage it is only the visual results which are important, this is in order to ascertain that aliasing is suppressed as expected. Figure 47 shows the results of pairing the technique with procedurally generated textures as described mathematically above. The technique shows promising results as, with an appropriate filter width set, the technique removes all visible aliasing under test conditions. This includes motion aliasing, which is difficult to demonstrate with the static images presented in Figure 47.

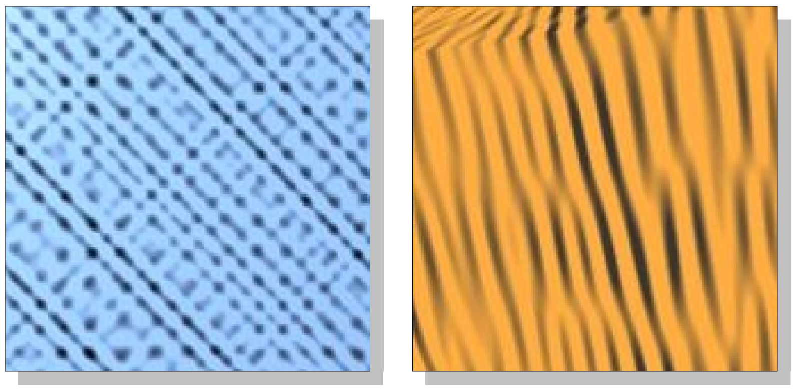


Figure 47: Anti-aliased procedurally generated textures

Although difficult to demonstrate with the static images of Figure 47, the technique does indeed have the desired anti-aliasing effect for procedurally generated textures. Next it is required to establish that a subset of frequencies can provide a reasonable approximation of an image, that is acceptable to the human eye. Both Figure 48 and Appendix A provide demonstrations of what it is possible to recreate using only a subset of frequencies.

A wide variety of textures were tested in order to ascertain whether different types of texture are more or less suitable for this technique[40]. The original texture size for these tests was 128x128. These dimensions were chosen to give the technique a reasonably sized pool of frequencies to work from while minimising the time taken to test each set. At the chosen resolution the original images will consist of 16384 individual pixels, this equates to 16384 frequencies in Fourier space.

40 See Appendix A.

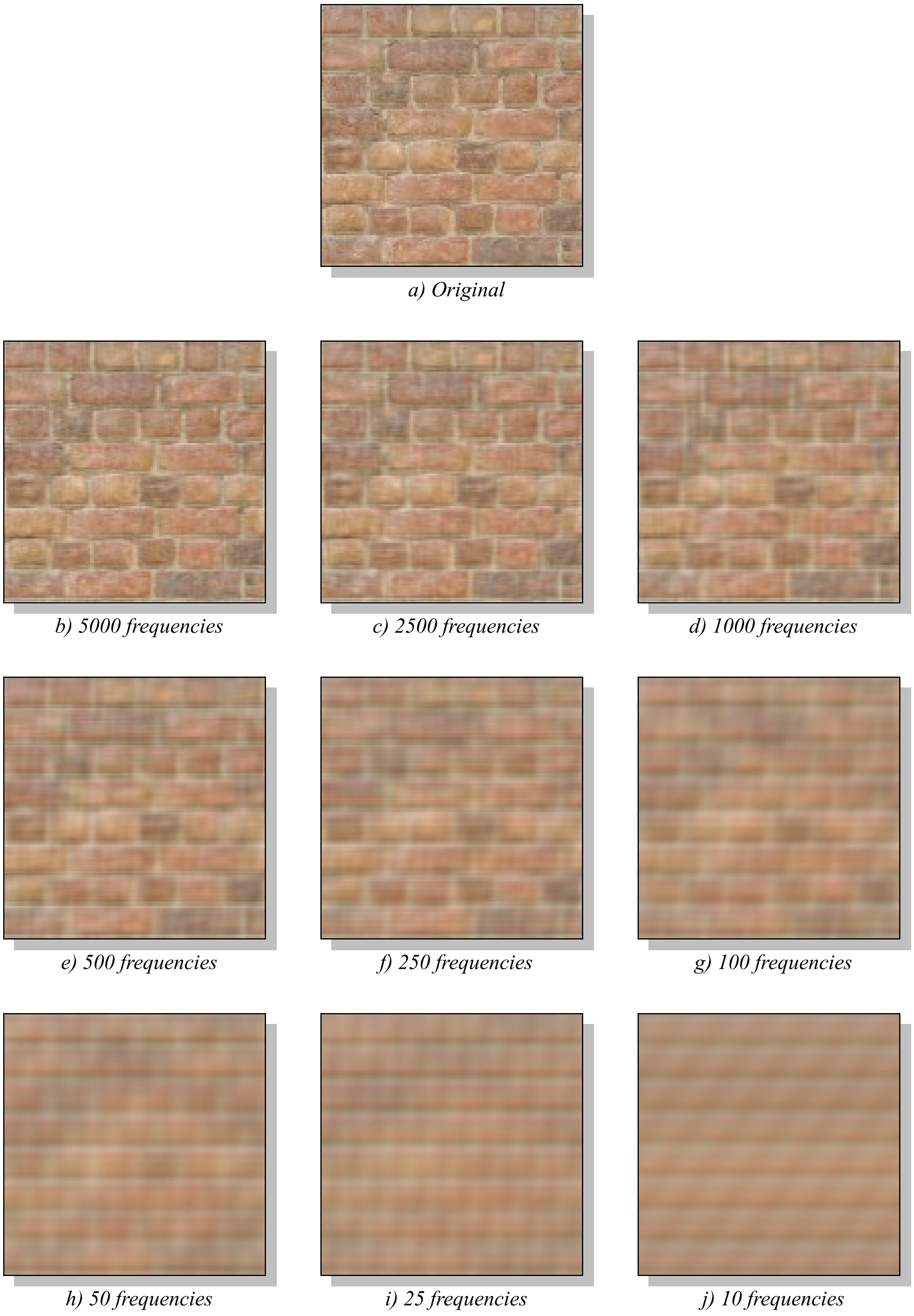
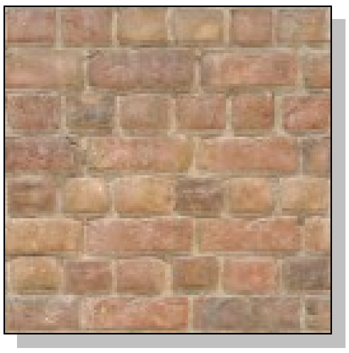


Figure 48: Software Frequency Tests: Brick Wall

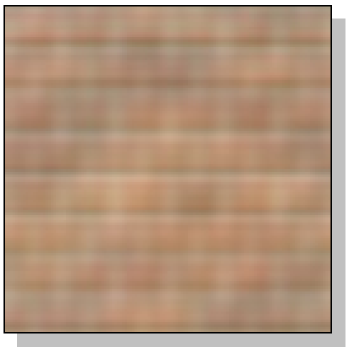
From the example test images it can be seen, that 5000 frequencies provides an approximation almost indistinguishable from the original for most types of texture tested, with most still having easily recognisable features of the original right down to 250 or even 100 frequencies in some cases. This shows us that a texture can be accurately represented by around 30% of its original frequency content. Similarly it shows us that textures can still be distinguishable with as little as 0.006% of the texture's original frequency content.

As these tests were performed as a software process, at this stage the performance is fairly slow. Thus, the next step is to ascertain how well the same tests perform when the technique is implemented in hardware vertex and fragment shaders. Doing this will enable us to evaluate the practicalities of running such a technique on a real-time system alongside other existing techniques such as bump mapping, in order to create a whole scene.

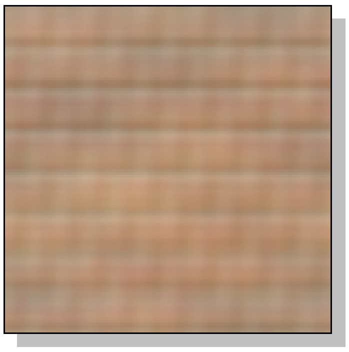
Figure 49 and Appendix B show comparisons between the software and hardware (shader) based versions of the Fourier textures technique. At first glance both versions appear to give similar results. However, on closer inspection it becomes apparent that there are slight differences between the two. This was an extremely difficult problem to diagnose given that the hardware version was expected to yield identical results but to process the data much quicker. After careful analysis it became clear that this was in fact an issue caused by deficiencies within the test hardware itself, due to the inherent inaccuracies in the calculations of the NVIDIA 6800 series GPU; on which the test



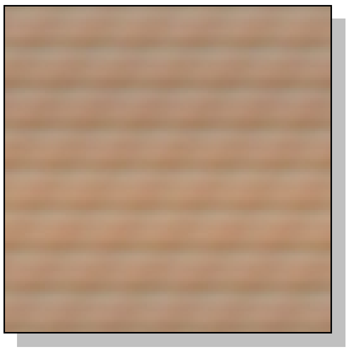
a) Original



b1) 50 frequencies

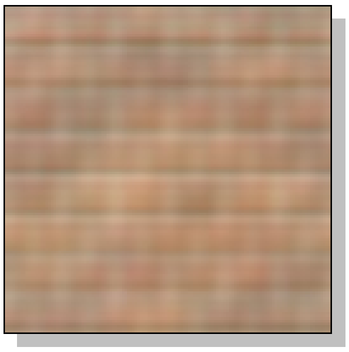


b2) 25 frequencies

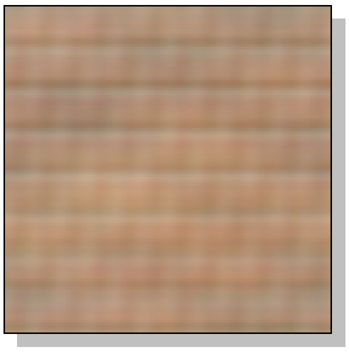


b3) 10 frequencies

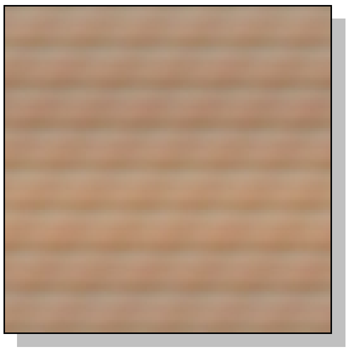
b) Software



c1) 50 frequencies



c2) 25 frequencies



c3) 10 frequencies

c) Hardware

Figure 49: Hardware Frequency Tests: Brick Wall

system is based. This meant that the slightly poorer visual quality in the hardware version's results were simply down to less accurate calculations being available on the hardware platform.

Another limitation imposed by the hardware platform is that, due to the way shaders perform loop operations and the requirements of calculating multiple frequencies per pixel, it is only possible to achieve a maximum of 256 frequencies per pixel. However, this in itself is not considered a major issue for the current hardware generation as real-time performance is lost long before reaching that level.

4.3.3 Conclusions

Despite the accuracy problems, and due to the small visible differences in quality between the hardware and software versions, it is still considered worthwhile pursuing this technique further. However, given the large number of frequencies required to represent an image at a reasonable visual quality and the performance implications of that (Performance is covered in greater detail in section 4.4.3), it will be necessary to reappraise the use of the technique. This may be achieved by combining the technique with currently known and widely used techniques such as MIP mapping.

4.4 Fourier Texture Filtering

Given the performance issues involved with representing images purely in terms of a subset of their frequencies, it is considered necessary to attempt to combine the technique with MIP mapping.

When done efficiently, standard MIP map tables include frequencies right up to the

Nyquist limit for their resolution. However, by harshly filtering the MIP maps and using a variation of the Fourier Textures technique to add filtered frequencies back to the filtered MIP maps before they are rendered, it should be possible to remove aliasing in a similar manner to that already shown. This will also have the effect of adding back some detail thus potentially preventing over blurring.

This variation of the technique should be able to achieve real time performance while not sacrificing too much in terms of image quality. It should also maintain low levels of aliasing. The process of harshly filtering the MIP maps to below their Nyquist limits should remove many potential causes of aliasing, although this will be at the expense of texture detail. This detail however will be added back into the final image one frequency at a time via the Fourier technique discussed earlier.

4.4.1 Fourier Texture Filtering Technique

As previously discussed the Fourier textures technique consists of two steps. Firstly there is the pre-processing software stage, as shown by Figure 50 and Figure 51, which consists of MIP map and Fourier frequency set generation. Then there is the hardware shader stage, as illustrated in Figure 52, which brings all that data together to create the final image.

The key feature of this technique is that it's built upon standard MIP mapping techniques whereby the standard MIP mapping system is supplied with pre-filtered MIP

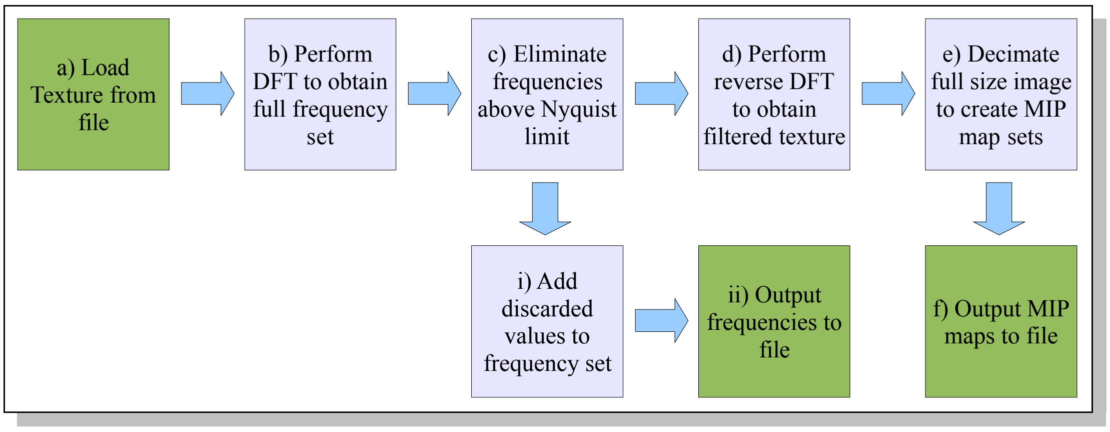

a) Load Texture from file
b) Perform DFT to obtain full frequency set
c) Eliminate frequencies above Nyquist limit
d) Perform reverse DFT to obtain filtered texture
e) Decimate full size image to create MIP map sets
i) Add discarded values to frequency set
ii) Output frequencies to file
f) Output MIP maps to file

Figure 50: MIP Map and Frequency set creation

maps. These maps have been harshly filtered to below the image's Nyquist limit in order to remove as much aliasing as possible.

1.43 times the texture storage is required over and above that needed for standard texture filtering to hold the Fourier information. As the currently available texturing facilities are not ideally suited to storing this data the Fourier information must be organised and retrieved manually. The layout of this structure is sequential, whereby an appropriate sized section of texture memory must be allocated to hold the required number of frequencies. The frequencies are then simply fed into the structure by order of magnitude. Considering there is a maximum of 256 frequencies possible on the test hardware, the requirements of this are fairly simple.

Taking an 128x128 source texture as an example:

a) Load texture data from a bitmap file.

b) Perform a discrete Fourier transform on each pixel of the input image to give a full set of Fourier frequencies.

c) Filter out all frequencies above the Nyquist limit (those beyond the quadrant between (0,0), (0,63), (63,63) and (63,0) in Frequency space) and:

   i. Retain the Zero (x = 0, y = 0) frequency for all levels. Add the remaining discarded values (frequencies above the Nyquist limit) to frequency sets. Which set a frequency is added to depends one of four pre-selected angle ranges (22.5°, 67.5°, 112.5°, 157.5°) and level of detail considerations. This is because the most "important" frequencies for any given texel will depend on sample footprint, level of detail, and angle to the viewer.

   ii. Output the frequency sets to file.

a) Perform reverse DFT to obtain filtered texture.

b) Decimate full size image to create MIP map sets. For example to obtain the 64x64 MIP map every 1 pixel in 4 is taken from the original 128x128 texture.

c) Output MIP maps to file in an appropriate image format, such as a bitmap.

Figure 51: Software stage

There is one complication however. Each Fourier frequency consists of 8 components[41] as opposed to the usual 4 components per pixel (RGBA). This requires the manual calculation of texture co-ordinates within the shader and leads to the requirement of two texture fetches per Fourier frequency. Whilst obviously not ideal this would be easily solved by the additional hardware functionality of an 8 component per pixel texture type or the ability to create custom texture fetch functions.

41 The real and imaginary components of the frequency contribution

```
Initialise OpenGL
Load texture data from the MIP map files and Fourier text files
    For each MIP map level
        Load image data into appropriate OpenGL texture MIP Map
        level
    Set Texture parameters
Load frequency set data
    While not end of file
        read frequency
        add frequency to frequency array
    Load frequency array into OpenGL texture
Initialise shaders
Render scene
    For each pixel
        Calculate level of detail
        Calculate texture coordinates
        For each frequency
            Sample texture to fetch frequency
            Calculate suppression factor (see section 4.3.1, equation 2)
            Calculate contribution from frequency content (see section
            4.3.1 and Appendix I)
            Apply suppression factor
        Perform standard bilinear/trilinear filtering on MIP map texture
        Combine results to final pixel colour using weighting factor
```

Figure 52: Hardware stage

4.4.2 Texture Selection

In order to properly compare the algorithm against others of its type, tests must be run for many different textures and resolutions. This is in order to ascertain what effect, if any, the differing patterns and resolutions have on final image quality and performance. The choice of textures and resolutions, since it cannot be made arbitrarily or exhaustive, must therefore be carefully selected and justified.

The test textures are selected to include types of textures which are commonly found within games and simulations, textures commonly used for testing texture filtering algorithms and textures explicitly chosen for their Fourier footprints. Textures chosen on the basis of their Fourier footprints are selected if it is considered that they may cause issues with the Fourier technique, this is in order to properly test the technique.

Figure 53 and Appendix C show screen shots from a selection of games which have been released since the year 2000. All the games displayed are either award winners or topped the sales charts for PC games at some point in the year of their original release. Therefore, this sample can be considered representative of popular computer games of their time and hence the texture content they used may also be considered representative of popular video games of their time.

The games sampled show large variations in both visual style and content. However, there are common themes running through many of the textures seen in each. This helps demonstrate that common textures used in games since the year 2000 include things such as grass, brick, stone, metals, wood and cloth. There is a fairly obvious pattern amongst these textures, as they are all things commonly seen in the real world on a day to day basis. This pattern is evident because games companies often strive to simulate what is, what was and what may be, as realistically as current hardware allows. It can therefore be concluded then that common textures used in games are simplified or re-imagined versions of the same images seen every day in the real world, meaning things such as lawns, brick walls, clothes and roads are all commonly simulated in games.

Figure 53: Half-Life 2: Episode Two, 2007

Given the types of textures commonly used in games, the images in Figure 54 were selected as candidates for testing and provide a representative set of the kinds of texture commonly found in games. However, for the purposes of testing the Fourier technique, it is not enough to be representative of game textures; the test images must also provide a suitably wide variety of Fourier footprints in order for the system to be properly tested.

After examination it is expected that this set will give a suitably wide spectrum of Fourier footprints, with textures such as Figure 54 a) Brick giving fairly even axis aligned horizontal and vertical frequency distributions, textures such as Figure 54 d) grass giving a fairly random distribution of frequencies, and textures like Figure 54 b) Fence and Figure 54 h) stripes showing a strong frequency bias in one direction.

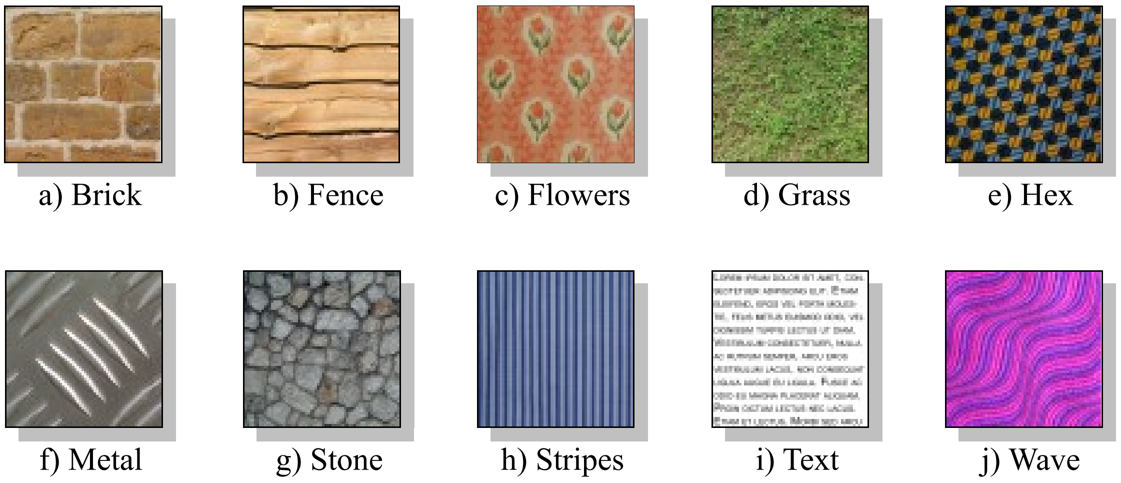
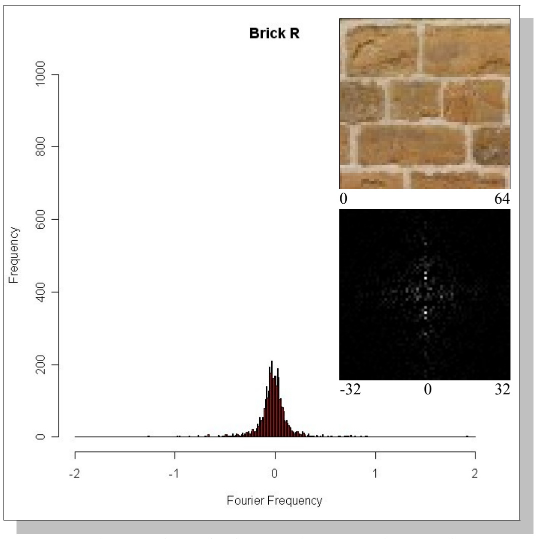


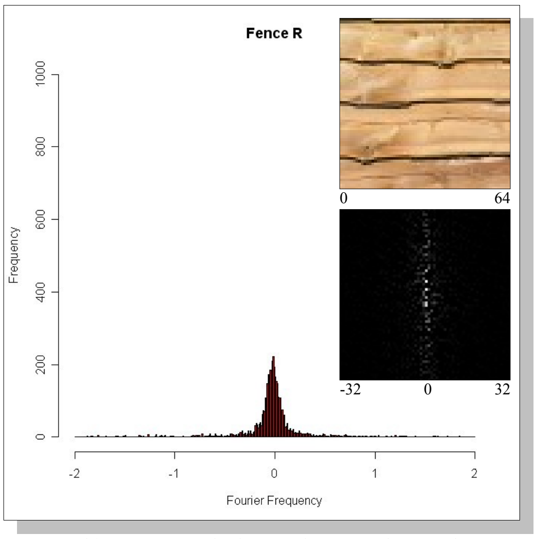
Figure 54: Textures for Testing

In order to confirm these assumptions are correct, it is necessary to transform each image into Fourier space. This will give us an indication of the overall frequency content, in particular the dominant frequencies direction and bias. It is necessary to produce histograms from the Fourier space frequencies, this allows a more accurate gauge of the frequency spread for each texture, and hence the impact particular frequencies and frequency ranges should have in image space. The resultant histograms and Fourier space representations are presented in Figures 55 and 56.

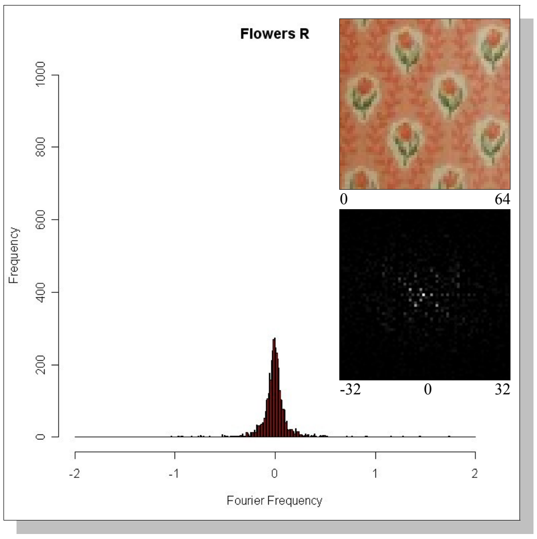
Given the requirements, Figure 55 and Figure 56 show the image space, Fourier space and Fourier space histogram representations of the images for comparison. Only the red channel is compared here. As the analysis is required to differentiate between the images and not the different colour channels, for which the red channel alone is



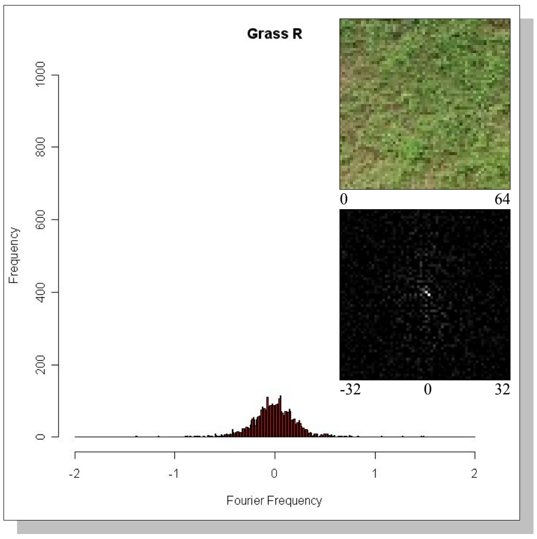
a) Brick red channel range limited



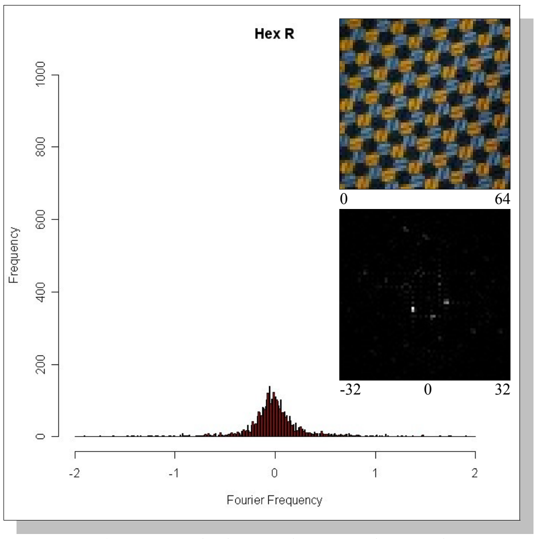
b) Fence red channel range limited



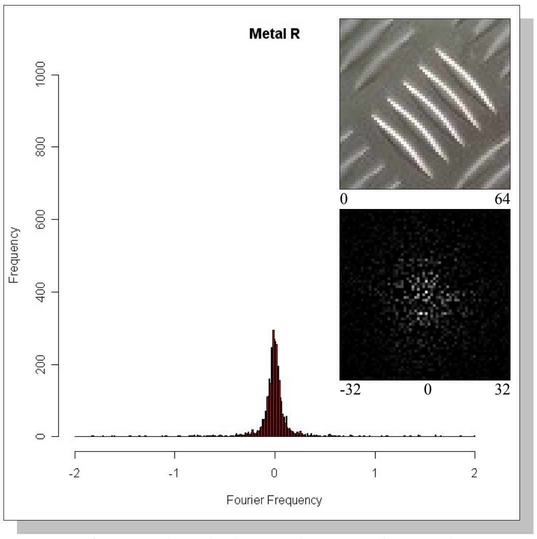
c) Flowers red channel range limited



d) Grass red channel range limited



e) Hex red channel range limited



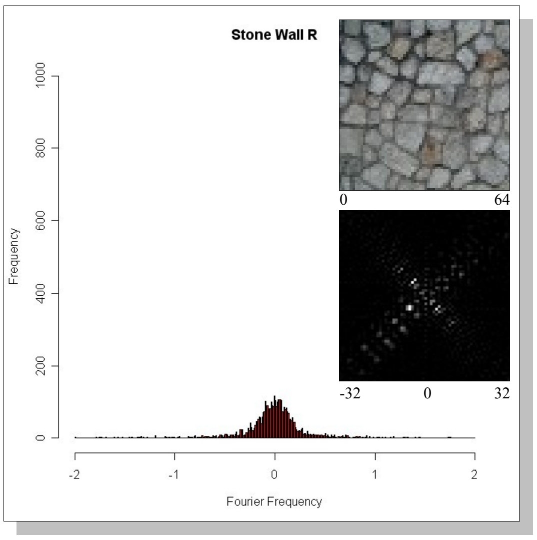
f) Metal red channel range limited

Figure 55: Red Channel Histograms

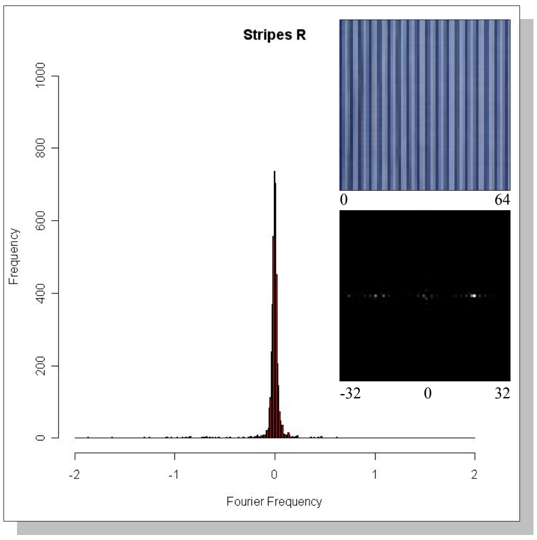
adequate. A full listing of red, green and blue histograms is available in Appendix D.

The Fourier space representations depict the distribution of the frequencies which represent each image. The brighter white areas of the Fourier space representations indicate a higher concentration of frequencies, and the histograms show the frequency of each of those Fourier frequencies. These are presented on a scale of -32 to 32, which corresponds to the 0 to 64 pixels of the image space representation.

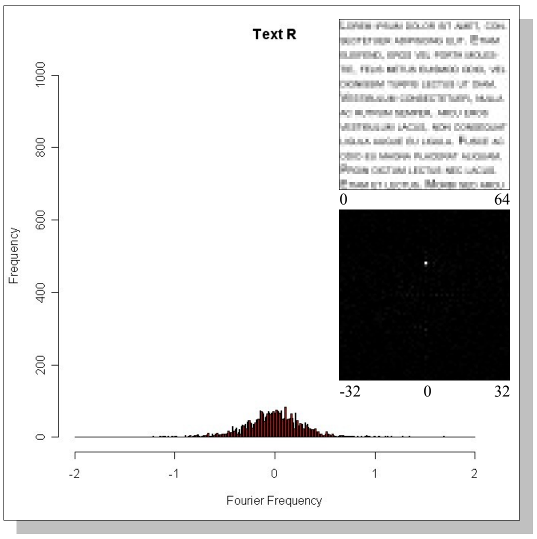
The brick texture is analysed in Figure 55 a), it shows that the brick pattern gives a fairly even distribution of frequencies in Fourier space. As expected, it is aligned on the horizontal and vertical axes, which represents the mortar lines in image space, although it does display a slight bias towards the vertical (the horizontal mortar lines in image space). The histogram confirms this with the separate peaks representing areas of denser frequency population in the Fourier space illustration, with a gradual tail off towards the lower frequencies. The weaker and slightly more randomly positioned frequencies observed represent the pattern of the brick surface. It is these fine details of the texture which will prove most difficult for the Fourier technique to recreate. This is because the nature of the technique relies on strong dominant frequencies being able to represent the majority of the image. The dominant mortar lines of the wall should be the easiest part of this pattern for the Fourier technique to reproduce, using a fairly low number of frequencies. However, the presence of two dominant frequency ranges will push the required number of frequencies up and this may result in the mortar pattern being lost with lower numbers of frequencies.



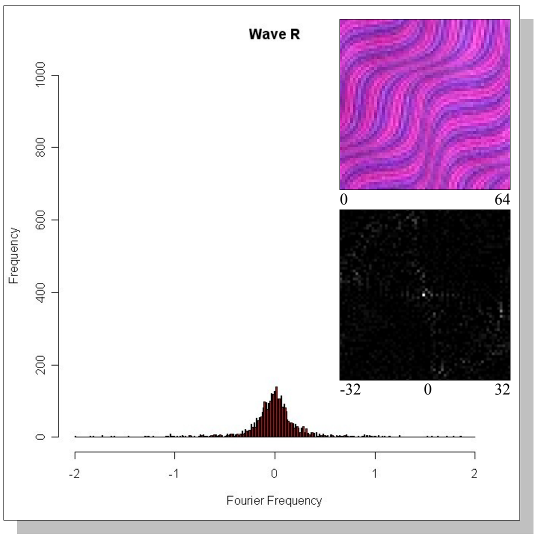
a) Stone red channel range limited



b) Stripes red channel range limited



c) Text red channel range limited



d) Wave red channel range limited

Figure 56: Red Channel Histograms

Figure 55 b) displays a very strong vertical frequency component in Fourier space. This represents the horizontal fence slats and wood grain in image space. The histogram and Fourier space representations demonstrate a higher concentration of high frequencies than in texture a), as well as a higher rate of fall off towards the lower frequencies. Again, some more random patterns are visible from the wood grain, but the majority of these are still fairly well aligned with the general vertical layout of frequencies. This indicates that the fence texture should provide better results for lower number of frequencies than many of the other patterns. However, the fine detail of the wood grain will only be reintroduced by the technique at much higher frequency levels, most likely beyond the realms of acceptable performance levels.

Figure 55 c) again shows mostly higher frequencies which is to be expected of such a regular pattern repeated in both the x and y directions. This repetition also gives a much more even spread in Fourier space. Unlike the previous two examples no particular orientation is dominant. This should enable the Fourier technique to easily pick out the major features of the pattern even for lower number of frequencies, meaning this pattern should give favourable results at steeper angles.

Figure 55 d) is a much more random pattern in image space, and as such gives a much more random spread of frequencies in Fourier space. While the higher frequencies are again dominant in the histogram, there is a lower peak along with a more gradual and even fall off than in the previous examples. The randomness of the image gives a fairly even spread of frequencies in Fourier space with no particular orientation being dominant. Consequently it will be difficult for the Fourier technique to accurately represent the pattern with low numbers of frequencies, and will likely result in a fairly solid green at steep angles. This is not necessarily a problem, as this kind of fine random pattern will tend to do the same, to a lesser extent, in optical systems at steep angles.

Figure 55 e) provides a much wider spread of frequencies than the previous examples.

This is displayed in all directions, with a slight dominance amongst the higher frequencies. The lack of a set of any obviously dominant frequencies here is a factor of the pattern having fine detail on the weave of the hexagon shapes. Again, this is a feature that is likely to be lost without the use of very high numbers of frequencies. precluding real-time operation. However, this is not necessarily a problem as, at steep angles, some of this detail would also be lost in an optical system.

Figure 55 f) like Figure 55 e) gives a much wider spread of frequencies than the previous examples. This is displayed in all directions, but with a greater dominance amongst the higher frequencies. This is caused by the grain of the metal, with the brightest spots in Fourier space being representative of the diagonal patterns. These dominant diagonal patterns should prove easy to represent with relatively few frequencies, but again the fine grain of the detail will be lost with all but the highest frequency numbers. Again, an optical system would also lose some of the fine detail at sharper angles.

Figure 56 a) shows an interesting dominance of frequencies along the the diagonals in Fourier space. This shows that despite the uniform appearance you may expect from the pattern in image space, with a more or less equal balance due to the stones all being multi sided and non uniform, there is actually a dominance of diagonal frequencies. Again the lack of any particularly dominant frequencies may cause issues with the Fourier method at lower numbers of frequencies.

Figure 56 b) illustrates the clear dominance of a narrow range of frequencies in one plain. Because of this, the stripe texture should give the best results with fewest frequencies, however the orientation of the texture when viewing may adversely effect this as some of the frequencies sets will contain relatively few of these dominant frequencies.

Figure 56 c) may well prove the most difficult of the textures to represent without an almost full set of frequencies. This is because, as you would expect with text, there are no real dominant frequencies or patterns within the image. Text patterns are a widely known problem for anti-aliasing techniques and prove to be the downfall of many otherwise promising techniques. This shall be the biggest test of the Fourier method.

Figure 56 d) gives a much wider spread of frequencies than the previous examples, even the text, but with a greater dominance among the higher frequencies this pattern should prove slightly more favourable for the Fourier technique.

It has been shown that the selected group of textures have a suitably wide variety of Fourier footprints, while sticking with the mandate of examining textures used commonly in games. These common game textures include those such as Figure 56 a), and Figure 56 c) provides an example of a texture commonly used for testing texture filtering algorithms. The variety of Fourier footprints, as shown in Figure 55 and Figure 56, should aid us in determining which types of texture give favourable results with the Fourier system, which cause issues, and whether or not the above expectations are borne out.

Given the processes involved in filtering the textures and generating the frequency sets, it would be expected that the technique should perform well on those textures with little variation in Fourier space, or where the majority of the frequencies lie on the same plain. This would indicate regular patterns like Figure 56 b) should be reproduced well via the Fourier method.

These textures are to be tested at resolutions of 64x64 to 1024x1024. This is in order to show both resolutions representative of textures used in games at the higher end of the scale, and at the lower end it shall be shown that lower resolution textures need not necessarily mean sub-standard detail. This will enable us to demonstrate whether the technique will allow the use of lower resolution textures, compared to those which are currently standard within the industry, to be feasibly used within games. This has the advantage of requiring less processing and will therefore potentially improve performance. It should be noted however that while the higher size textures chosen are representative of the games industry at the time of writing the average size of textures used has been steadily increasing and may continue to do so for the foreseeable future.

4.4.3 Results

In this section, the Fourier Texture Filtering (henceforth Fourier) algorithms' performance is analysed after translation into hardware shaders. The results will be compared against exemplars of current best and accepted standard practice within the games and simulation industries. Comparisons against real world images taken by an optical system shall also be made.

The comparisons with real world images of the textures will provide an important gauge of success for all the techniques observed, not only the Fourier techniques. Through comparisons of the textures with real world images, it shall be possible to see how close these techniques come to being able to simulate real world situations, and which, if any, of the visual defects shown by the techniques are naturally occurring within optical systems.

It is important to note when examining real optical images that optical systems suffer from defects such as depth of field, as such it is necessary to use a very high quality source for the comparison images. This is in order to rule out any interference from such defects in the comparisons, as none of the techniques presented in this section attempt to simulate depth of field.

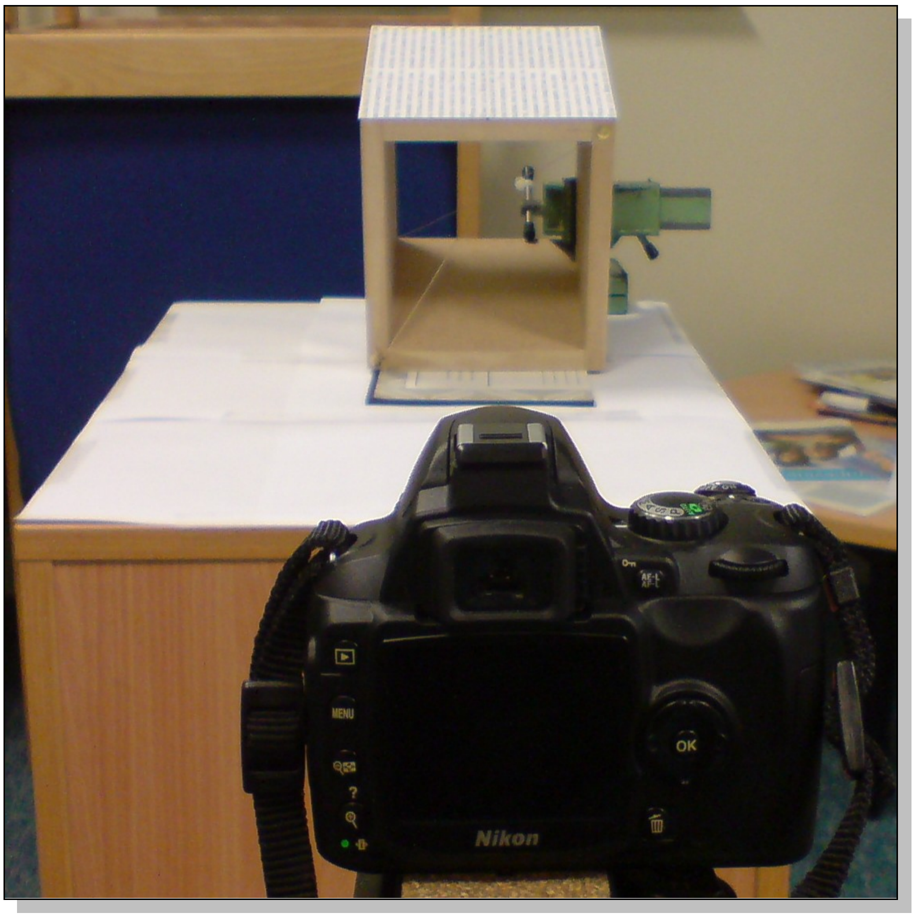


Figure 57: Set-up for producing the optical comparison images

With this in mind high-resolution printed versions of the textures were created and applied to a real world object. This object is sized in proportion with the simulated cube. The real-world object was positioned in front of a digital still camera in such a way as to mirror the simulated environment. Figure 57 shows the arrangement used for obtaining the optical images, as the parameters of the simulation are known, the organisation and set-up of the optical scene was simply a matter of matching these parameters in the real scene.

In order to avoid the undesirable effect of depth of field the system was set-up to produce images which are far too sharp and high in resolution to provide fair comparison against the simulated textures. Because the simulated textures are limited by the output resolution of the hardware, it is required that the resultant optical images be re-filtered to a resolution which makes the desired comparisons appropriate and fair.

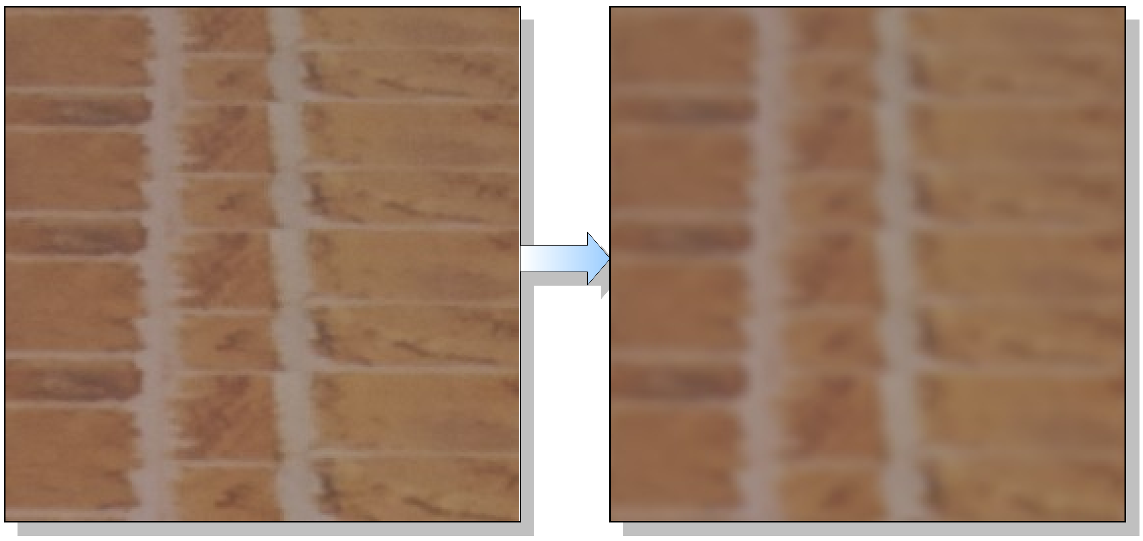


Figure 58: Original Vs Re-filtered version of the Brick texture

The results of the re-filtering process are shown in Figure 58. The process involves first filtering down the original high resolution image by approximately a factor of four and then filtering the resultant image back up to the original image size via the same filtering method. The well defined and well understood nature of optical images allows the use of a high quality isotropic filter[42], such as that proposed by Mitchell and Netravali [79], without needing to worry about many of the issues discussed earlier which plague filtering techniques in the 3D graphics domain.

The final filtered image shown in Figure 58 b) gives an example of the appropriate level

42 Isotropic filters are ideal here as they are independent of directional constraints.

of detail for proper comparison with the presented techniques. As mentioned, to directly take an optical image with these parameters would result in image blurring due to depth of field rather than purely by the resolution constraints required here. However, it should be noted that even with these precautions in place, various other potential issues will still effect the final images. This is because colour reproduction, lighting and other environmental conditions must be taken into account. These issues should not affect the results however, as it is only image detail comparisons that are required.

The results of tests for all pertinent aspects of each algorithms stated visual goals are discussed below. Once the success or otherwise of each algorithm is established they are then performance tested. This is to establish whether or not they are appropriate for real-time use as is, with optimisation, or not at all on current hardware.

Visual quality will be closely examined and compared to existing similar techniques. The existing techniques used for comparison are chosen based upon several criteria. Firstly, current widely used techniques are chosen and any expected and actual differences in visual quality noted. Secondly, if appropriate, the recognised best case techniques are used to show how Fourier texture filtering, which is designed for real time use, compares against techniques with theoretically superior visual quality but non real-time performance.

It is often noted that 30 Frames Per Second (FPS) is an appropriate goal as a minimum average frame rate within a game or simulation to “avoid jerky motion” [80], and is in fact the absolute frame rate set by many games, such as the Grand Theft Auto series. This gives the same result as activating vertical synchronisation or v-sync in your graphics driver settings, it helps prevent tearing artefacts caused by uneven frame rates. Using this figure of 30 FPS as a goal for the Fourier algorithm when fully optimised for the target system and integrated into a real-time application, such as a game, the un-optimised algorithms should display performance relative to the performance of un-optimised versions of bilinear and trilinear filtering, for which fully optimised hardware implementations exist for comparison.

As well as visual quality and performance other more subtle elements are discussed, such as the pre-processing software stage, which some of the techniques require. These are elements which may not be quantifiable in terms of frames per second or visual accuracy but still require analysis, as such they will be considered in the context of use within a game industry setting. Various factors are recorded and analysed, particularly processing time and man hours taken to generate the data used in each test case, as both are important considerations for practical use.

Presented below are the testing results for both the hardware and software processes required for the use of the Fourier techniques. The software side is examined first; this can be easily done as a batch process, so performance is not necessarily an issue, although the pre-processing required should be comparable with other techniques. The testing results for the hardware side are then presented, in which performance and visual quality are the most important aspects for comparison against the other techniques examined.

4.4.3.1 Software

Evaluating the success of the software stage is relatively straightforward as it simply needs to produce two things, a set of MIP maps with a reduced frequency content and a set of the top n frequencies which are to be used to add detail back into the final image.

As this part of the technique is an off-line software process which can be easily done in batch the performance of this is largely irrelevant. As the hardware side of the technique relies on the software, the success of this stage of the technique can be determined by the success of the hardware shader process.

4.4.3.2 Hardware

Analysing the hardware shader section is much more complex. Success is measured in a number of ways, either by achieving a better quality final image, and thus reducing artefacts and aliasing compared to current techniques, or by showing better performance for a similar level of image quality. Even a slight reduction of aliasing can be considered a success, as long as not too much image detail is lost in the process.

In order to test the Fourier algorithm it needs to be compared against suitable and currently available techniques. Those which represent both the current de-facto standard techniques within the industry and also the highest quality techniques available must be selected. Therefore, “Anisotropic filtering”, as utilised on current generation graphics hardware, and EWA have been chosen for comparison purposes.

EWA has been selected as a suitable benchmark for image quality because, as McCormick et al [70] state: EWA “provides a quality benchmark against which to compare other techniques” and is “the best software anisotropic texture filtering algorithm known to date”. Shin et al [71] also argue that EWA “generates the very high quality images, but requires the intensive computation power and texel values. This method provides a quality benchmark used when to compare various filtering techniques”. The EWA technique provides superior image quality to other current techniques, and hence shows the least artefacts or aliasing. Visually EWA should compare favourably to any technique considered here, theoretically showing the least artefacts or aliasing. However, EWA suffers badly in performance compared to less computationally expensive techniques. Thus, it can be concluded that EWA may be used as a quality benchmark when considering the success of texture filtering techniques and for comparison against other techniques. EWA can therefore be held up as an exemplar of quality for anisotropic techniques.

As previously mentioned, “Anisotropic filtering” describes the current most widely utilised technique in real-time graphics systems, including games and simulations. At the time of writing it is used, in slightly differing forms, on both NVIDIA and ATI hardware. These two companies constitute the vast majority of the graphics card market. Anisotropic filtering is a much higher performance algorithm than less performance oriented techniques like EWA, and hence it suffers many more artefacts. It is chosen as an appropriate comparison algorithm because it is the current de-facto industry standard, this is despite heavy optimisation often reducing visual quality as discussed previously. The particular implementation of Anisotropic filtering chosen is that used on NVIDIA's line of GeForce graphics cards. This is because NVIDIA are the market leaders, hence NVIDIA's implementation of anisotropic filtering is the most widely utilised at present.

In the case of the Fourier textures technique, it is expected that a slight loss in image detail will be observed when compared to NVIDIA's overly sharp anisotropic implementation, but with considerably less aliasing. When comparing to EWA it is expected that similar levels of detail and aliasing are displayed in the case of lower resolution images. Although it is expected that EWA will provide the best visual results overall, particularly at higher resolutions.

The optimum settings for the Fourier technique, given each particular set of test images, also need to be discovered. For each particular input image this requires ascertaining which settings of filter width and number of frequencies give the optimum balance between final image quality against performance. Perfect image quality is defined here as: a complete lack of artefacts while retaining all of the source textures sharpness. Therefore the optimum image quality for any given scene is achieved by balancing the number of on screen artefacts against the level of blurring. In some situations sharpness is preferential. However in any real time simulation, where motion of the scene is required, blurring is infinitely preferable to aliasing artefacts. This is because aliasing artefacts are greatly accentuated by motion and hence increase in significance with the animated scenes required by most modern games and simulations.

It is expected that the Fourier technique will perform well at the lower resolutions, this is because fewer frequencies will be required to accurately represent the original image. This also means that a greater frame rate will be achievable due to the fact that fewer frequencies will be necessary. It is also expected that the Fourier technique shall perform better on textures with a narrower concentration of frequencies, as shown in the histograms earlier, and hence a smaller number of dominant frequencies contributing to the final image. In these cases it becomes possible to represent the original texture well with fewer frequencies and may hence allow better performance while still maintaining acceptable levels of image quality.

With regards to the performance of each algorithm in a real world scenario, one important factor to consider is memory utilisation. This is because even cutting edge graphics cards have a limited amount of memory, particularly when coupled with the usual requirement that games and simulations, and hence the texture filtering techniques they utilise, need to be made to take into account less capable systems.

4.4.3.2.1 Theoretical Performance

There are several important indicators for theoretical performance, the most cited of which are usually memory requirements and sample counts.

Memory requirements give an indication of how well a technique will fit into existing hardware, where memory is at a premium. While sample counts can provide a rough indicator of processing requirements, with higher sample counts potentially prohibiting real-time performance.

4.4.3.2.2 Memory utilisation

Techniques such as EWA and Anisotropic filtering are generally used in conjunction with standard MIP mapping, this obviously has extra memory requirements over and above that of regular texturing. Unfortunately the memory footprint of MIP Maps are not ideal for standard memory management techniques. This is due to the requirement of MIP map level dimensions being power of twos and each level being exactly half the size of the previous level, culminating at 1x1. This leads to an odd number of pixels, which can be awkward in terms of memory management.

Looking at the memory requirements of common techniques it is possible to establish that the Fourier technique compares well against standard MIP mapping.

MIP mapping requires:

$$m_t \approx T_n + T_{n-1} \ldots T_{n-m}$$

where:

- m_t = total memory used
- T_n = memory required for texture at level of detail n. i.e. texture size = $width \times height$ at n multiplied by the number of bytes per pixel.

So for a 64×64 texture, assuming a floating point texture with 32-bits (4 bytes) per component in RGBA pixel format:

$$\begin{aligned}
m_t &= ((64 \times 64) + (32 \times 32) + \ldots + (2 \times 2) + (1 \times 1)) \times (4 \times 4) \\
m_t &= (4096 + 1024 + 256 + 64 + 16 + 4 + 1) \times (4 \times 4) \\
m_t &= 5461 \times 16 \\
m_t &= 87376\, Bytes
\end{aligned}$$

That means 87376 Bytes are required for MIP Mapping alone. This roughly equates to a memory requirement of $m_t \approx 1.33\mathrm{m}_o$ where m_o is the memory footprint of the original texture

Next this is considered alongside the Fourier technique, which is calculated similarly but must take into account several extra factors:

$$m_t \approx (T_n) + (T_{n-1}) \ldots (T_{n-m}) + F$$

where:

- F equals $2 \times L \times S \times A$.
- L is the number of levels of detail.
- S is the number of sub-levels.
- A equals the number of angles.

A is typically 4, where S and L are dependant on the original texture size:

- $L = n+1$ where $T_w = 2^n$ and T_w is the original texture width.
- $S = L$.
- A is selected as 4 to balance visual quality and performance.

Taking the same 64×64 floating point texture as an example, the Fourier system requires:

$$\begin{aligned}
&L=7, S=7, A=4.\\
&m_t=(((64\times64)+(32\times32)+\ldots+(2\times2)+(1\times1))\times(4\times4)+((2\times7\times7\times4)\times(4\times4)))\\
&m_t=(4096+1024+256+64+16+4+1)\times(4\times4)+392\times(4\times4)\\
&m_t=5853\times16\\
&m_t=93648\,\mathit{Bytes}
\end{aligned}$$

Which still equates to less than two full size textures (i.e. less than two times the size of the original texture or $m_t < 2\mathrm{T}_w$) to cover all levels including MIP maps and frequency sets. More accurately the Fourier method requires $m_t \approx 1.43\mathrm{m}_o$ where m_o[43] is the memory footprint of the original texture. This compares to $m_t \approx 1.33\mathrm{m}_o$ for standard

43 In the 64x64 texture example this would be 65536 Bytes or 64x64x16

MIP mapping.

These figures are a little more tricky to calculate for some versions of anisotropic filtering, as the memory footprint depends on the particular implementation used. RIP-Mapping for example uses a great deal of texture memory, where as Summed Area tables and Footprint assembly[44] are much more comparable to regular MIP Mapping in terms of memory footprint.

The NVIDIA implementation of anisotropic filtering requires the same memory footprint as standard MIP Mapping ($m_t \approx 1.33\mathrm{m}_o$), as it builds on the basis of trilinear filtering. However, NVIDIA's anisotropic filtering requires many more samples, or texture fetches, than standard MIP Mapping.

The NVIDIA GeForce 6800, on which most of the testing is performed, has a maximum 2D texture size of 4096x4096. This is a common limitation and when combined with texture memory limits, 256Mb[45] on many current cards[46], enables the calculation of the number of textures which can be stored in texture memory at any one time.

Given an original texture size of just 64x64, each MIP Mapped texture will occupy approximately 87376 bytes; meaning a total of around 3072 64x64 textures will be available on a 256Mb GeForce 6800 in a typical scene which requires MIP mapping.

44 See section 4.2.8
45 268435456 Bytes
46 The 6800 used for testing is limited to 128Mb

Closer examination of this figure reveals that using larger and larger textures rapidly decreases the number of possible textures per scene. The addition of more and increasingly complex techniques such as Normal Mapping only adds to this requirement. This makes the slight increase in memory footprint an important factor in analysing the success of the Fourier technique, and at only 1.43 times the size of a regular texture it compares favourably with many of the other anti-aliasing techniques available and only suffers a slight increase in memory requirements over the current industry standard basic MIP mapping technique.

4.4.3.2.3 Samples and bandwidth

The number of samples, or texture fetches, is an important measure of potential performance which avoids the difficulties of comparing optimised and un-optimised techniques as well as the constraints or deficiencies of particular implementations.

Basic texture mapping uses one sample per pixel, or one texture fetch per pixel, which means that for a 64x64 textured image there will be a total of 4096 samples taken per frame. This is calculated as follows:

$$
\begin{aligned}
&s_p = 1\\
&s_t = s_p \times height \times width\\
&s_t = s_p \times 64 \times 64\\
&s_t = 1 \times 4096\\
&s_t = 4096
\end{aligned}
$$

where s_t is the total number of samples required for the image and s_p is the number of samples per pixel.

Standard bilinear filtering utilises 4 samples per pixel, so that equates to 4 texture samples per pixel or a total of 16384 samples for a 64x64 texture image. So for bilinear filtering:

$$\begin{aligned}
&s_p = 4\\
&s_t = s_p \times height \times width\\
&s_t = s_p \times 64 \times 64\\
&s_t = 4 \times 4096\\
&s_t = 16384
\end{aligned}$$

Trilinear filtering uses an extra four samples per pixel over and above that of bilinear due to the fact it takes two bilinear samples, so that equates to a total of 32768 samples in a 64x64 texture image:

$$\begin{aligned}
&s_p = 8\\
&s_t = s_p \times height \times width\\
&s_t = s_p \times 64 \times 64\\
&s_t = 8 \times 4096\\
&s_t = 32768
\end{aligned}$$

Both the Fourier technique and the NVIDIA anisotropic technique build on bilinear and/ or trilinear filtering so these sample numbers provide the basis on which both techniques build.

As discussed previously, NVIDIA's anisotropic technique theoretically builds on trilinear filtering, with each anisotropic sample equating to one trilinear sample. Therefore 16x anisotropic filtering means 16 trilinear samples per pixel, this equates to 128 samples per pixel or 524288 samples in a 64x64 texture image:

$$\begin{aligned}
s_p &= 8\times 16\\
s_t &= s_p \times height \times width\\
s_t &= s_p \times 64\times 64\\
s_t &= 128\times 4096\\
s_t &= 524288
\end{aligned}$$

Similarly 8x anisotropic filtering requires:

$$\begin{aligned}
s_p &= 8\times 8\\
s_t &= s_p \times height \times width\\
s_t &= s_p \times 64\times 64\\
s_t &= 64\times 4096\\
s_t &= 262144
\end{aligned}$$

4x anisotropic filtering requires:

$$\begin{aligned}
s_p &= 8\times 4\\
s_t &= s_p \times height \times width\\
s_t &= s_p \times 64\times 64\\
s_t &= 32\times 4096\\
s_t &= 131072
\end{aligned}$$

2x anisotropic filtering requires:

$$s_p = 8 \times 2$$
$$s_{2x} = s_p \times height \times width$$
$$s_{2x} = s_p \times 64 \times 64$$
$$s_{2x} = 16 \times 4096$$
$$s_{2x} = 65536$$

However, the actual number of samples taken by this technique is most likely limited by optimisations such as brilinear filtering and optimisations based on viewing angle, so the true number is unlikely to reach these maximum values.

The hardware implementation of the Fourier technique requires two bilinear or trilinear samples per frequency per pixel. For example, in the case of the bilinear Fourier technique, if 10 frequencies are used then 20 bilinear samples are required per pixel. Hence the technique requires a theoretical maximum of 80 samples per pixel. Therefore when used in conjunction with bilinear filtering, the Fourier technique requires 16 frequencies to match the requirement of NVIDIA's 16x anisotropic filtering at 128 samples per pixel. Thus, if the technique can obtain similar levels of detail within that range of frequencies then it can be considered a real competitor for anisotropic filtering in regards to performance.

In the case of trilinear Fourier, 10 frequencies equates to a theoretical maximum of 160 samples per pixel. Hence, when used with trilinear filtering, the Fourier technique requires just 8 frequencies to match the requirement of NVIDIA's anisotropic filtering at 128 samples per pixel. A fuller list of sample requirements can be found in Table 13 for comparison.

As with the NVIDIA anisotropic filtering, some optimisations can be made to improve performance, so the numbers are not necessarily definitive. Obviously more is known about the implementations of the optimisations in this case. The Fourier technique uses the angle of the surface to limit the number of frequencies used, as certain angles require much less added detail that others, so the true number of samples per pixel varies based on angle and is therefore scene dependant. The figures given in Table 13 represent the absolute maximum number of samples in all cases.

Frequencies	**Bilinear** (Samples per pixel)	**Trilinear** (Samples per pixel)
0	0	0
10	80	160
20	160	320
30	240	480
40	320	640
50	400	800
60	480	960
70	560	1120
80	650	1280
90	720	1440
100	800	1600
110	880	1760
120	960	1920
130	1040	2080
140	1120	2240
150	1200	2400
160	1280	2560
170	1360	2720
180	1440	2880
190	1520	3040
200	1600	3200
210	1680	3360
220	1760	3520
230	1840	3680
240	1920	3840
250	2000	4000

Table 13: Fourier texture filtering sample requirements

Figure 59 and Figure 60 show graphical representations of Table 13, with the results for NVIDIA's anisotropic filtering superimposed over the top. It can be seen from Figure 60 particularly that the maximum 16x setting for NVIDIA filtering is roughly equivalent to 16 Fourier frequencies in the current system.

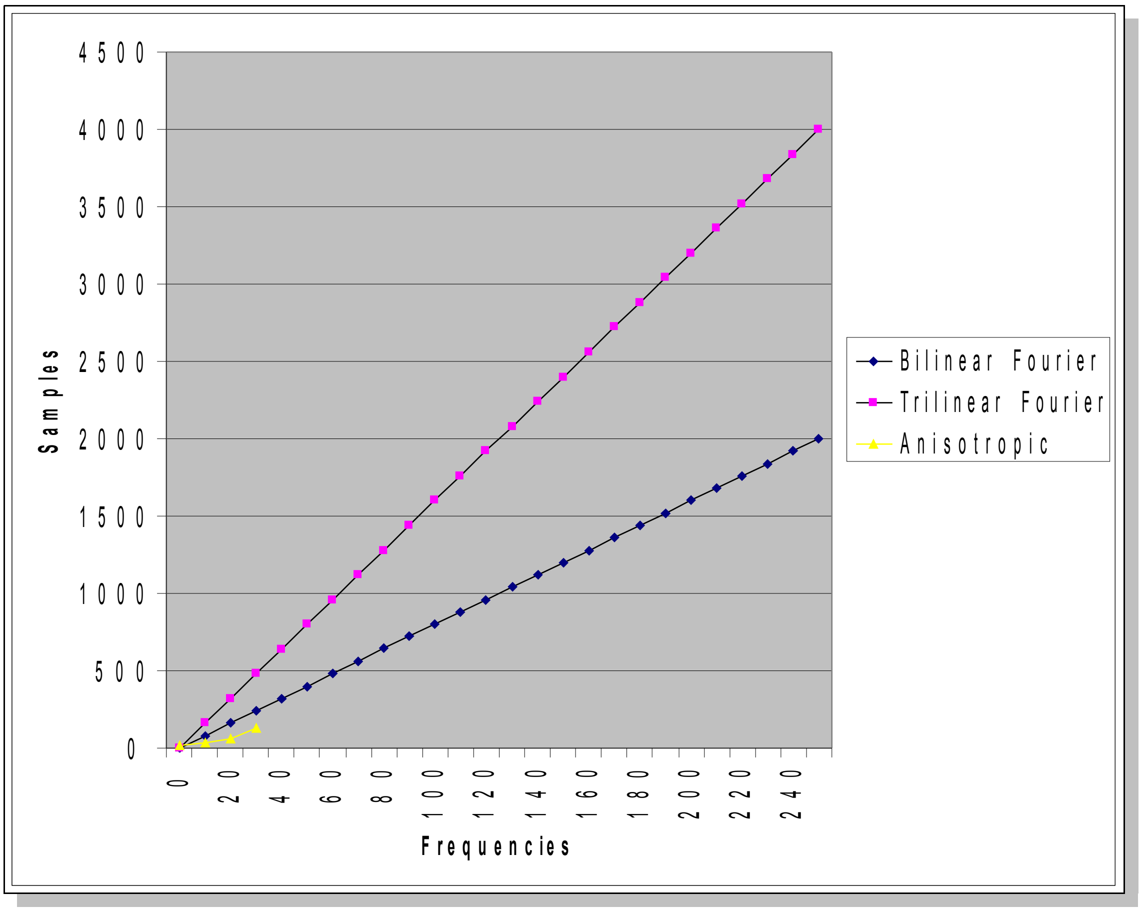


Figure 59: Samples - full graph

It should be noted that the number of samples taken by the presented implementations of the Fourier technique is artificially increased by the unorthodox methods used to store and retrieve the texture. With only slight adjustments to OpenGL or any other 3D API, this number can easily be halved. The basic problem lies with the fact that the Fourier techniques require the retrieval of 8 values per pixel, that is twice the usual four values retrieved per pixel. This is an issue because the standard texturing pipeline is set up to cope with only the four standard RGBA values. A slight alteration to the API's or hardware could facilitate the halving of the number of samples required and allow one texture structure per Fourier texture. Without these adjustments the Fourier techniques therefore require two texture fetches from two separate structures. Although, even with these adjustments the Fourier technique would still require twice the bandwidth of standard MIP Mapping.

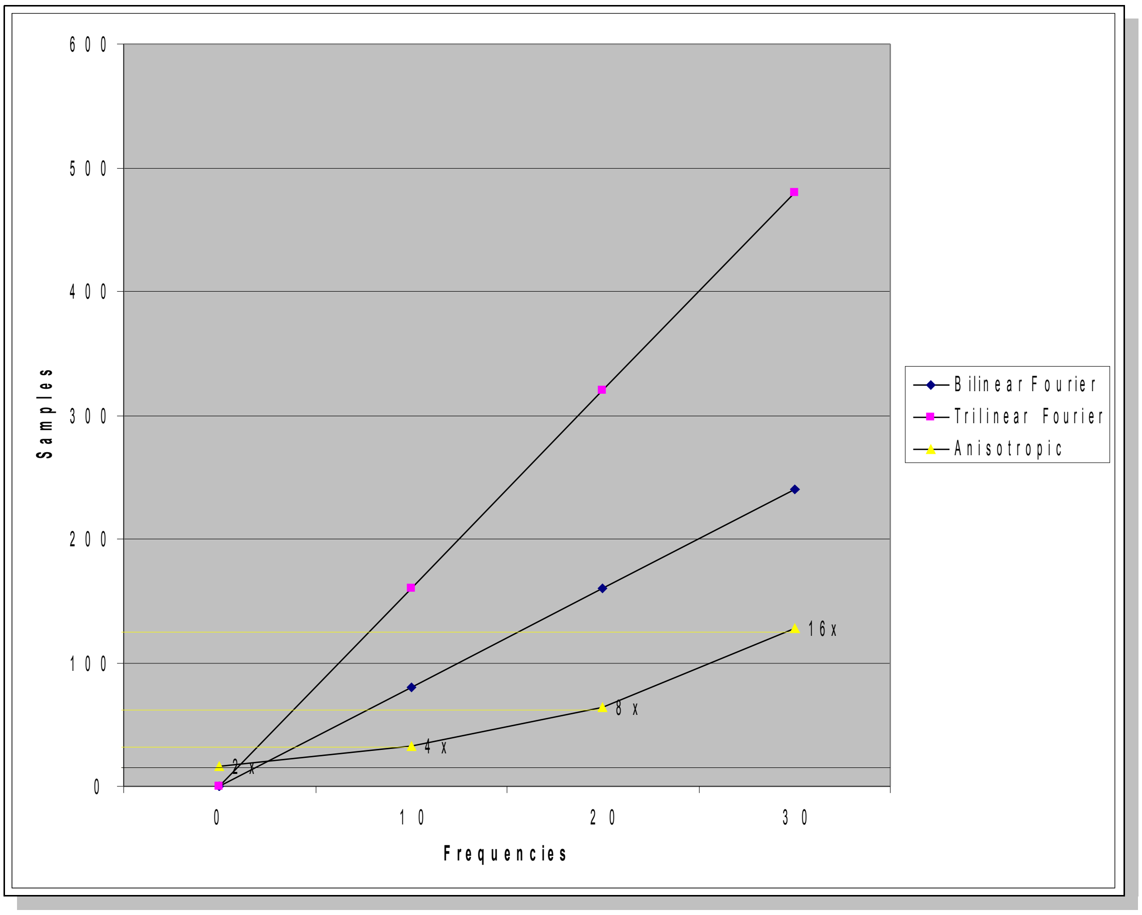


Figure 60: Samples - zoomed

This means that, if properly integrated into a 3D API, the Fourier techniques would provide a much more viable alternative to the NVIDIA anisotropic filtering on a theoretical performance basis. However, to properly compete, it must also show comparable or better image quality, as well as similar or lower aliasing levels for comparable sample counts. Taking into account the compromises imposed by current API's this point comes at 32 frequencies when paired with bilinear filtering and at 16 frequencies when paired with trilinear filtering. This is double the number possible for comparable sample numbers in the current implementation.

4.4.3.2.4 Observed Performance

This section investigates the observed performance of each algorithm on the test system. It is expected that the heavily optimised and hardware accelerated NVIDIA anisotropic filtering shall provide many times the performance of the Fourier techniques, which should in-turn provide similar performance advantages over the computationally expensive EWA.

In order to accurately analyse the performance of each technique, the frame rates achieved must be examined while varying conditions such as resolution and input texture. It is expected that increases in texture resolution shall adversely affect the performance of all the techniques, as they all utilise per-pixel operations. Neither the Fourier or EWA techniques take any account of the content of the textures at runtime and so should, within the realms of experimental error, provide identical results for each texture. It is assumed that the same shall be true of the NVIDIA anisotropic technique. However, as little is known about the exact implementation of the NVIDIA technique, it is conceivable that there are optimisations in place which take into account texture content. One possible optimisation here would be to take samples of the texture at load time and to analyse the differences in contrast between the texels. This is done in order to determine possible pattern variations within the texture and hence the minimum/maximum level of anisotropy that may be necessary.

Perhaps most importantly, the effect of varying the number of samples needs to be ascertained for the Fourier and NVIDIA Anisotropic techniques. Usually it can be expected that an increase in the number of samples taken will have a direct effect on frame rates, or more precisely a doubling of the number of samples will lead to a halving of the frame rate. However, both techniques apply some optimisations. It is known that the Fourier techniques use angle based optimisation to reduce the number of samples (frequencies) required, so it is expected that this shall be reflected in the performance graphs. Perhaps providing a similar curve to what could be expected from an un-optimised version, with the curve raised along the y axis by the optimisation.

As previously discussed, the input texture should make no difference to the performance of the techniques. This theory is examined here, also the affect of extra samples on each technique is investigated. Figure 61[47] and Figure 62 show graphical representations of the test data for the Fourier and NVIDIA Anisotropic techniques from the 64x64 brick input textures. Unfortunately EWA proved too slow to obtain any meaningful performance data, showing a consistent 2 frames per second on the test hardware regardless of the software settings.

Figure 61 shows that increasing the number of samples (hence frequencies) does indeed have a direct effect on the frame rates. A fairly steep fall off is observed at lower numbers of frequencies easing off towards the top end. This shows around a 50% initial performance hit per 10 frequencies which drops to less than 0.01% per 10 frequencies at the top end. This illustrates that optimisations are successfully restricting the number of

47 In Figure 61 the abbreviations BF and TF stand for Bilinear Fourier and Trilinear Fourier respectively.

extra frequencies used at the top end of the range.

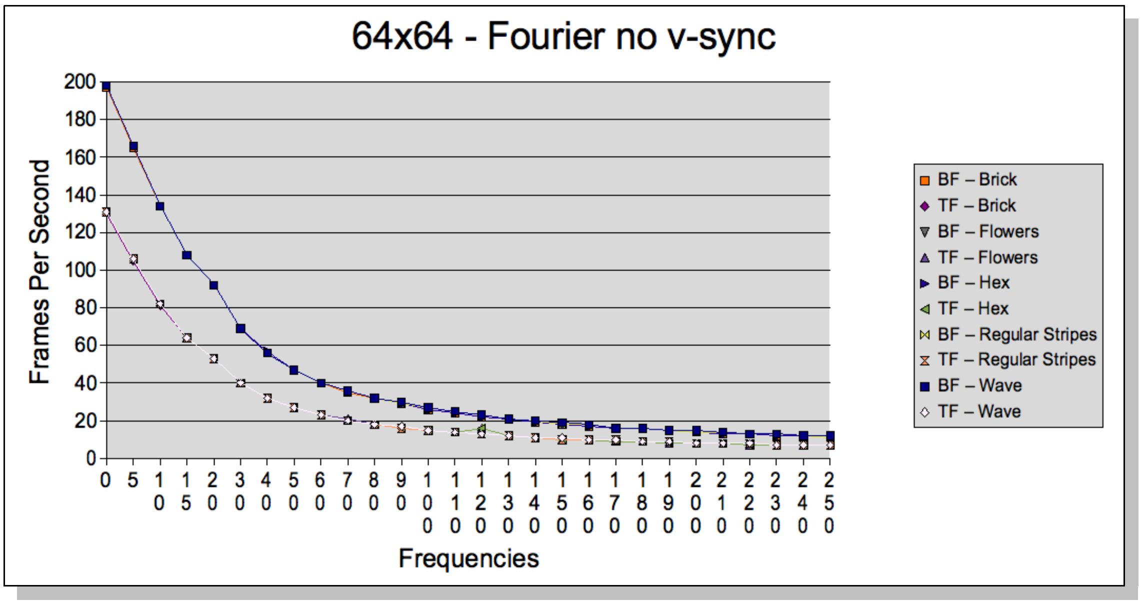


Figure 61: Fourier 64x64 Performance Graph

It can also be shown from Figure 61 that, as hypothesised, the input texture makes no difference to performance. Apart from a few small fluctuations, the performance for each texture is identical. These fluctuations can easily be explained by experimental error given the nature of the experiment and the scale of the differences observed.

What can be observed from Figure 61 and Figure 62 is that the NVIDIA Anisotropic technique, when run unconstrained, easily outpaces the Fourier technique. On examination of the data it can be shown that the Fourier technique shows between 1% and 15% of the performance of the NVIDIA anisotropic technique and that EWA offers between 0.13% and 0.15% of the performance of the NVIDIA anisotropic.

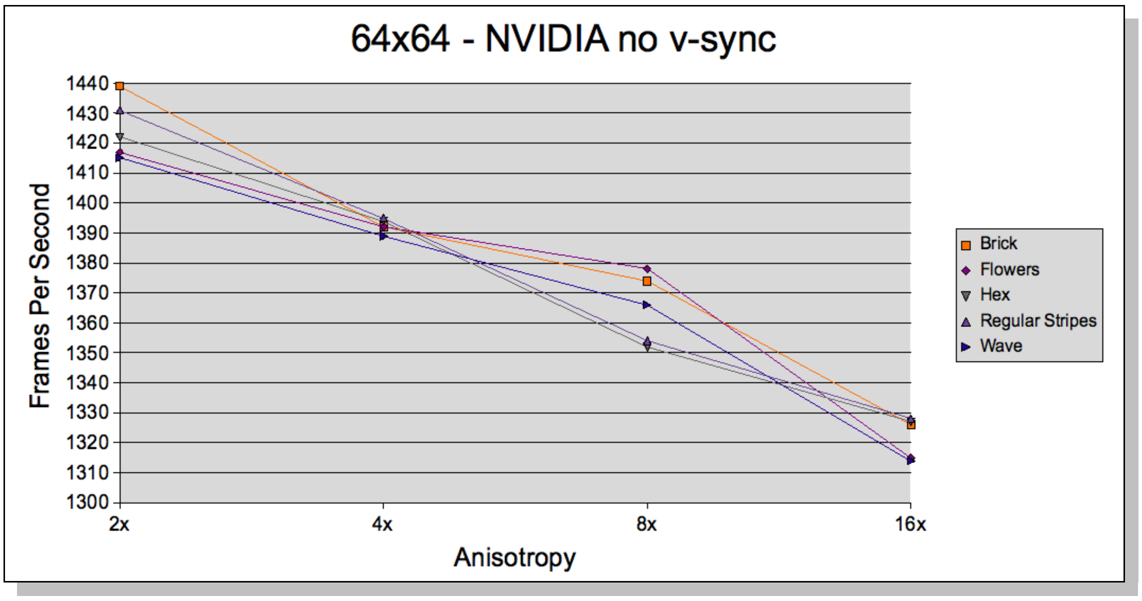


Figure 62: Anisotropic 64x64 Performance Graph

However, it should be noted that there is another factor that needs to be considered in terms of actual rather than theoretical performance, that is the matter of v-sync[48]. V-sync is a common graphics driver setting, which is often activated by default. This is because it prevents tearing artefacts in the on screen images, caused by timing issues. Activation of v-sync limits the maximum frame rate to that of the monitors synchronisation rate. This would mean that both the NVIDIA anisotropic and Fourier techniques would be limited to the refresh rate of the monitor. For example, if the refresh rate were 85Hz then this translates to a performance cap of 85 frames per second. Although for these purposes it is better to look at the unlimited performance of each technique, in order to identify feasibility for real time applications by examining the maximum frame rates.

While Figure 62 shows much more variation in frame rates between textures than Figure 61 the differences still only represent less than 2% of the maximum frame rate.

48 Vertical Synchronisation, which is the synchronisation of the frame rate with the monitors vertical refresh rate

Therefore these differences are most likely within the realm of experimental error rather than evidence of any texture content optimisations, although this should not be ruled out.

4.4.3.2.4.1 Brick Wall

Having firmly established that the texture content makes no difference to any of the techniques performance, the next step is to concentrate on one of the textures, to further investigate performance levels.

As can been seen in Figure 63, when using the Bilinear technique, the maximum

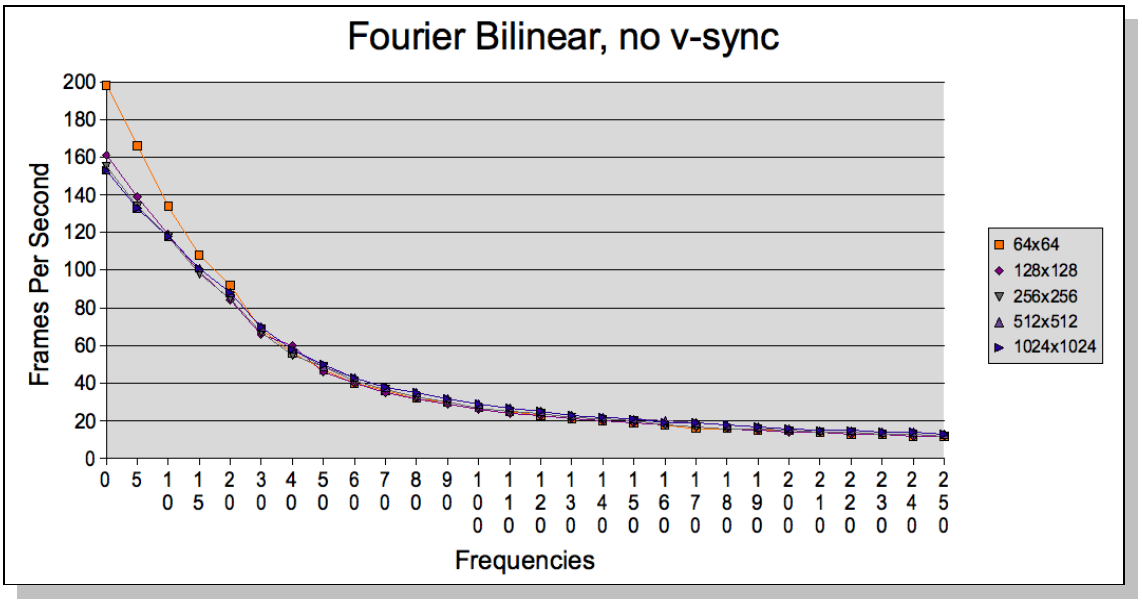


Figure 63: Brick Wall - bilinear performance

difference between frame rates for the different texture sizes is less than 8% of the maximum frame rate achieved with the 64x64 texture. However, this difference is using the same number of frequencies for each size of texture. As already established, the number of frequencies required to accurately represent an image increases dramatically as texture size increases. This is due to the percentage of frequencies required remaining around the same for each size increase.

Figure 64 demonstrates a similar pattern for Trilinear filtering with the maximum difference between frame rates being again around 8% of the maximum frame rate. Both Figure 63 and Figure 64 show a similar drop off in performance. The graphs demonstrate that the Bilinear based Fourier performs around 5-6% better than the Trilinear based Fourier technique. While this is not a huge performance difference, trilinear filtering is perhaps an unnecessary extra computational expense in this case as the Fourier process helps reduce the visibility of the MIP map layer transitions that trilinear filtering is usually tasked with solving.

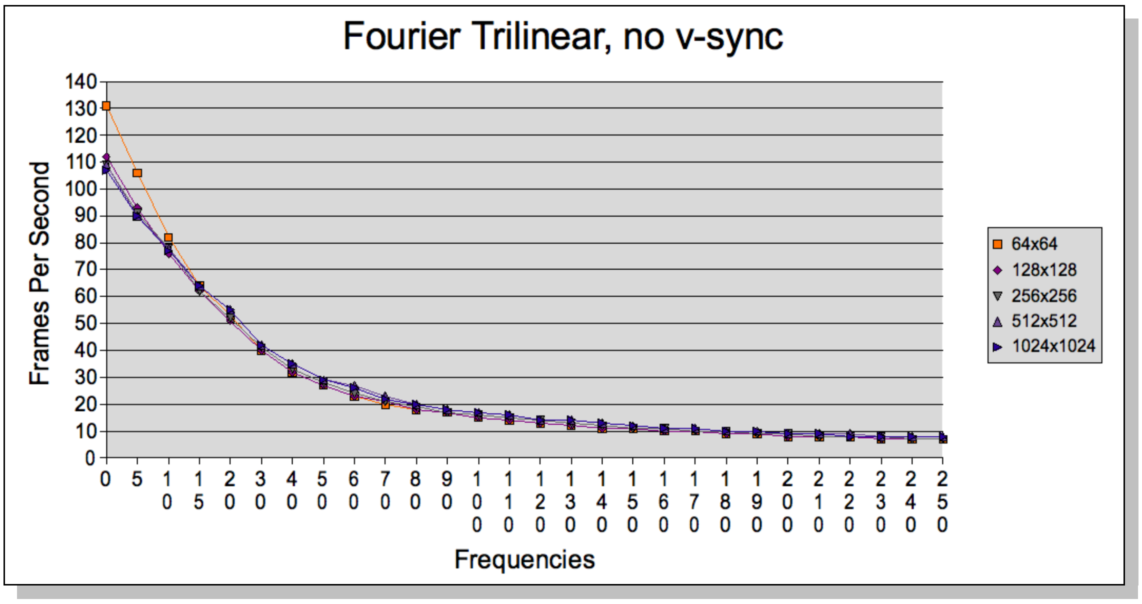


Figure 64: Brick Wall - Trilinear performance

Figures 65 and 66 compare the performance of the NVIDIA technique on two different platforms, the NVIDIA 6800 and newer 8600 graphics cards which became available late into the testing process. What can be seen from these graphs is that the performance

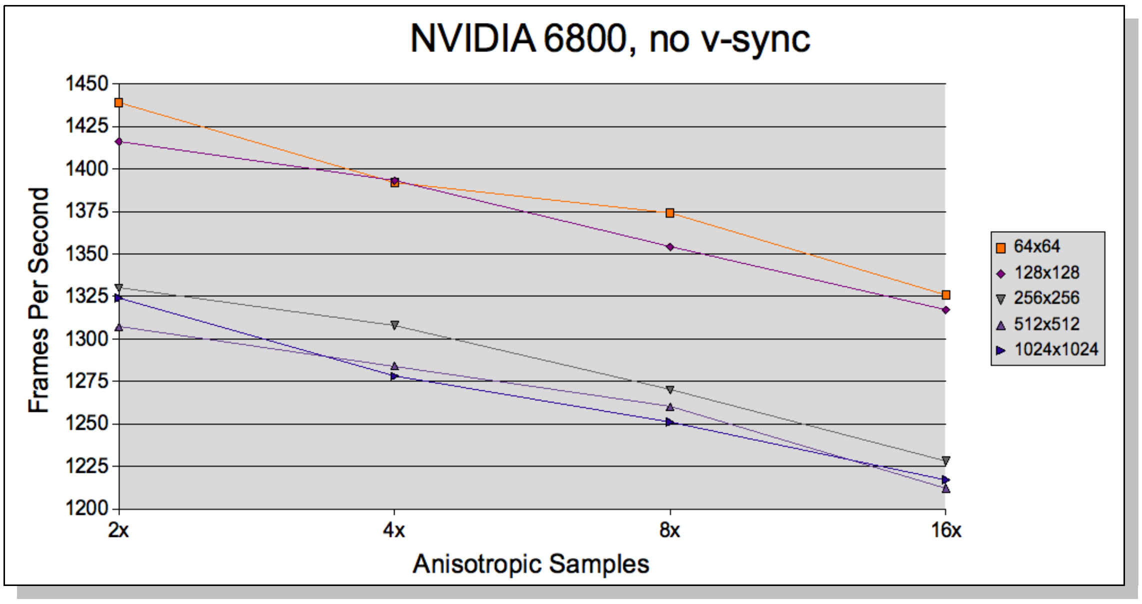


Figure 65: Brick Wall – NVIDIA 6800 performance

of anisotropic filtering has actually decreased despite a significant increase in shader performance between the hardware generations.

This may be construed as the signalling of a shift of focus by NVIDIA away from fixed function techniques such as this to the more general shaders. With many old fixed function techniques now being implemented exclusively in shaders this is unsurprising, but it does demonstrate the differences in performance between shader programs and fixed function techniques. These performance graphs also show a shift of optimisation for the technique, with the 6800 performing best at a resolution of 64x64 while the 8600 version of the same technique has seemingly been optimised for larger textures.

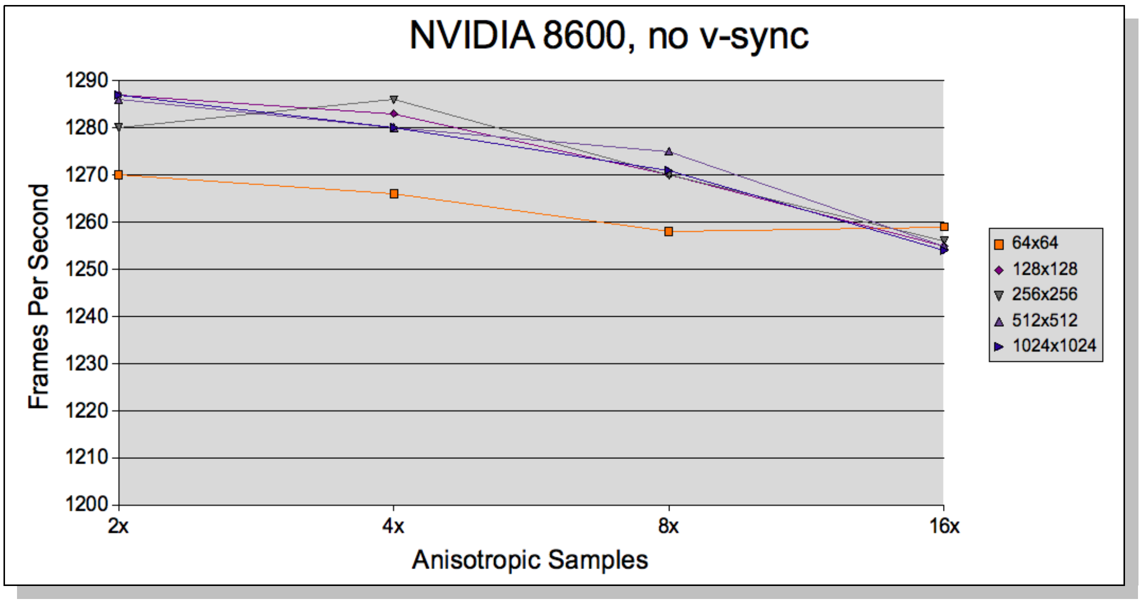


Figure 66: Brick Wall – NVIDIA 8600 performance

4.4.3.2.5 Visual Results

Having established the performance of each technique the next task is to examine the visual output of each.

4.4.3.2.5.1 Brick Wall

The first texture examined, the brick wall pattern, is one example of the kind of textures often found in video games. Scenes involving brick walls are extremely popular in video games and are more often than not required to be viewed from many different angles both sharp and shallow, making this brick texture a prime candidate for testing the Fourier technique.

Taking a closer look at the exact texture chosen. Figure 67 a) shows the original brick wall texture for comparative purposes, and Figure 67 b) shows a Fourier analysis of the texture. The Fourier space representation indicates that the majority of the frequencies which make up the image are around 50% below maximum frequency, along with fairly low frequencies along x and y although they are slightly higher in the y direction.

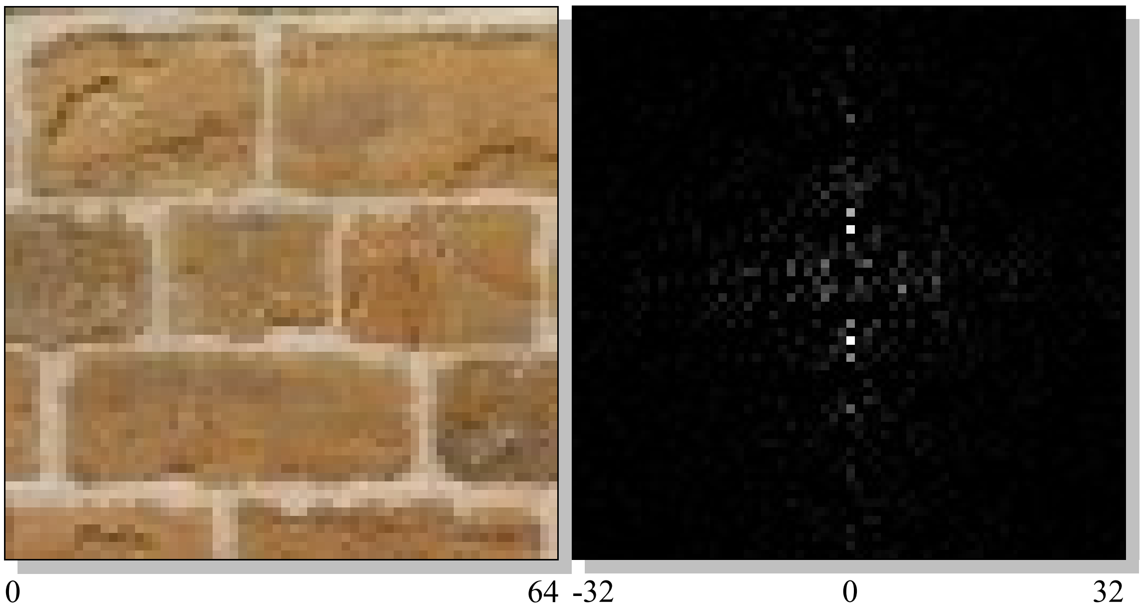


Figure 67: Brick Wall

Figures 68 - 71 display the visual results of testing the 64x64 brick texture at the optimum number of frequencies. In this case it is determined that 74 frequencies provides a comparative level of quality with the other techniques. Comparing the generated images to the optical system in Figure 72 it can be seen that the NVIDIA technique is overly sharp for the horizontal mortar lines, this results in motion aliasing which was not observed with the other techniques.

64x64

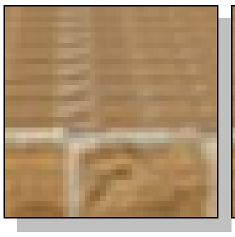


Figure 68: Bilinear

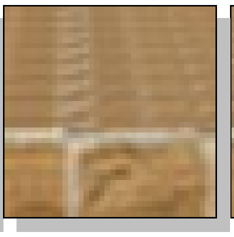


Figure 69: Trilinear

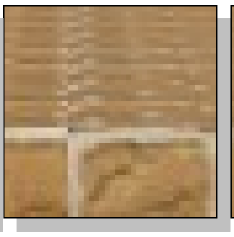


Figure 70: NVIDIA

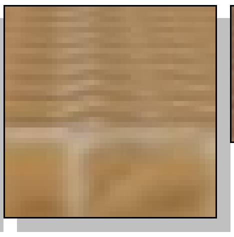


Figure 71: EWA

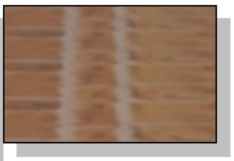


Figure 72: Optical

It is also noted that no technique retains the detail of the vertical mortar lines as shown in the optical image. In this case EWA shows an over blurred image, this can be lessened by adjustments to the technique but at the expense of aliasing. However, this is expected as the EWA implementation is optimised for larger textures.

The next texture resolution examined is 128x128. Figures 73 - 76 show the visual results for this test. At this resolution the Fourier techniques cannot compete on detail retention even with a full 256 frequencies. However, it should be noted that neither Fourier technique show any signs of the aliasing which plagues the NVIDIA anisotropic filtering. As expected EWA retains much more detail at this resolution, although it still lacks sharpness on the front face.

128x128

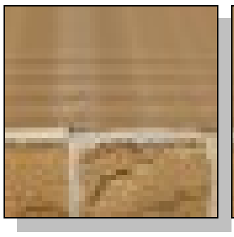


Figure 73: Bilinear

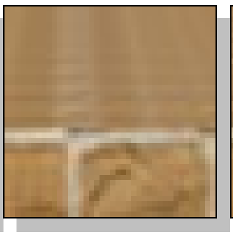


Figure 74: Trilinear

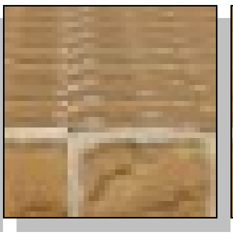


Figure 75: NVIDIA

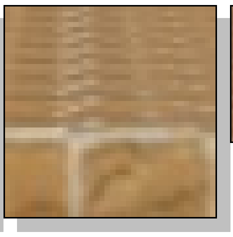


Figure 76: EWA

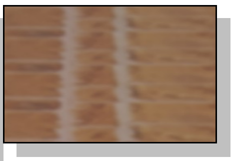


Figure 77: Optical

At a resolution of 256x256 the Fourier techniques again suffer with detail retention compared to the other techniques, this can be seen in Figures 78 - 81.

256x256

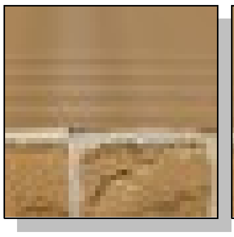


Figure 78: Bilinear

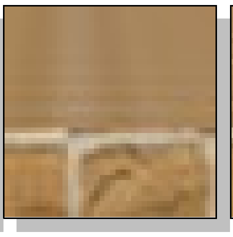


Figure 79: Trilinear

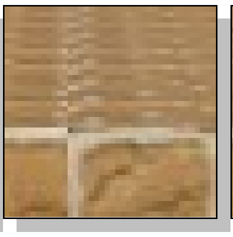


Figure 80: NVIDIA

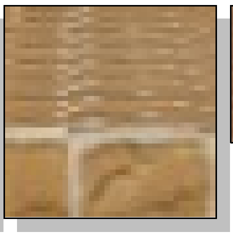


Figure 81: EWA

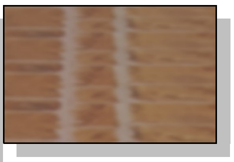


Figure 82: Optical

The EWA technique is beginning to show its true abilities at this resolution, demonstrating slightly greater detail retention than the NVIDIA technique at distance with significantly less aliasing.

512x512

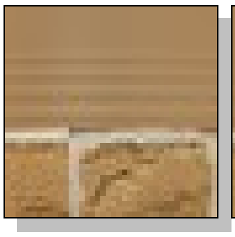


Figure 83: Bilinear

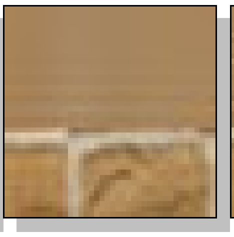


Figure 84: Trilinear

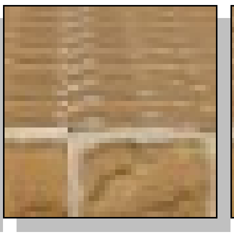


Figure 85: NVIDIA

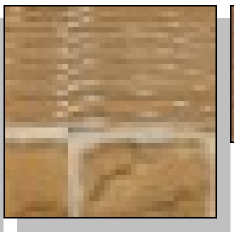


Figure 86: EWA

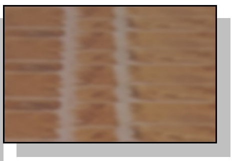


Figure 87: Optical

Figures 83 and 84 demonstrate an almost complete loss of texture detail for the Fourier techniques at a resolution of 512x512. EWA (Figure 86) again adds more detail over the previous lower resolution, and now shows a noticeable improvement over the NVIDIA technique (Figure 85), particularly towards the top of the slope.

The final resolution tested is 1024x1024. Again the Fourier techniques (Figures 88 and 89) show a lack of detail retention. As expected EWA displays slightly more detail at this resolution, showing by far the best detail retention of any technique examined.

1024x1024

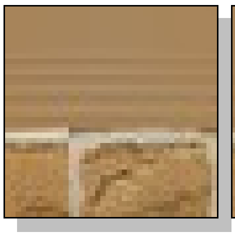


Figure 88: Bilinear

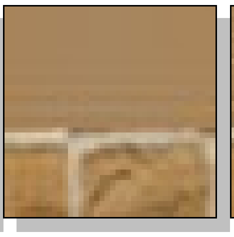


Figure 89: Trilinear

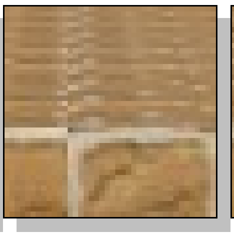


Figure 90: NVIDIA

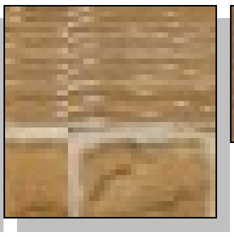


Figure 91: EWA

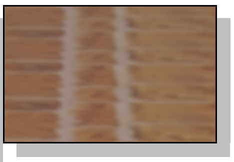


Figure 92: Optical

At this point it is noted that the NVIDIA technique provides visually indistinguishable results for all resolutions. This is most likely an indicator that the NVIDIA technique performs resolution independent operations which remain the same regardless of texture size and also explains the poorer performance for higher resolution textures on the 6800 test platform. EWA shows noticeable improvements in detail retention as the texture resolution increases. EWA's increasing detail is in direct opposition to the Fourier techniques which lose detail as texture resolution increases, this is due to the 256 texture limit imposed by the hardware.

Despite the 256 frequencies limitation and given the results at a resolution of 64x64, it is possible to estimate the number of frequencies this texture would require at higher resolution. These estimates are presented below in Table 14.

Texture	Height	Width	Texels	Frequencies	% of total frequencies
Brick	64	64	4096	74	1.81%
	128	128	16384	>256	>1.56%
	256	256	65536	983-1311	1.5%-2.0%
	512	512	262144	3932-5242	1.5%-2.0%
	1024	1024	1048576	15729-20972	1.5%-2.0%

Table 14: Frequency requirements for the brick wall texture

Table 14 displays the frequency requirements for each texture resolution, where red values are estimates based on previous results. These estimates are necessary as the test hardware does not allow greater than 256 frequencies, therefore the exact values cannot be obtained experimentally. As can be seen, the requirement of 74 frequencies for competitive representation at a resolution of 64x64 equates to 1.81% of the total frequencies. It is also known that at a resolution of 128x128, more than 1.56% of the total frequencies are required for an accurate representation of the original image. This allows us to estimate the remaining values as they are likely to lie within 1.5% and 2% of the total number of frequencies for each texture size. These estimates are reasonable, given it is expected that the percentage of frequencies required should remain fairly constant regardless of texture size.

4.4.3.2.5.2 Fence

The next texture examined is another example of the kind of textures which are often found in video games; a fence pattern. Scenes involving fences, similarly to brick walls, are popular in video games. They are often required to be viewed from many different angles both sharp and shallow, making the fence texture a prime candidate for testing the Fourier technique.

Figure 93 takes a closer look at the exact texture chosen. Figure 93 a) shows the original fence texture for comparative purposes, and Figure 93 b) shows a Fourier analysis of that same fence texture. The Fourier space representation shows a strong vertical frequency component, representing the horizontal fence slats and wood grain. Some more random patterns are also visible, which are likely to be representative of the wood grain, but these are still fairly well aligned with the general layout of frequencies.

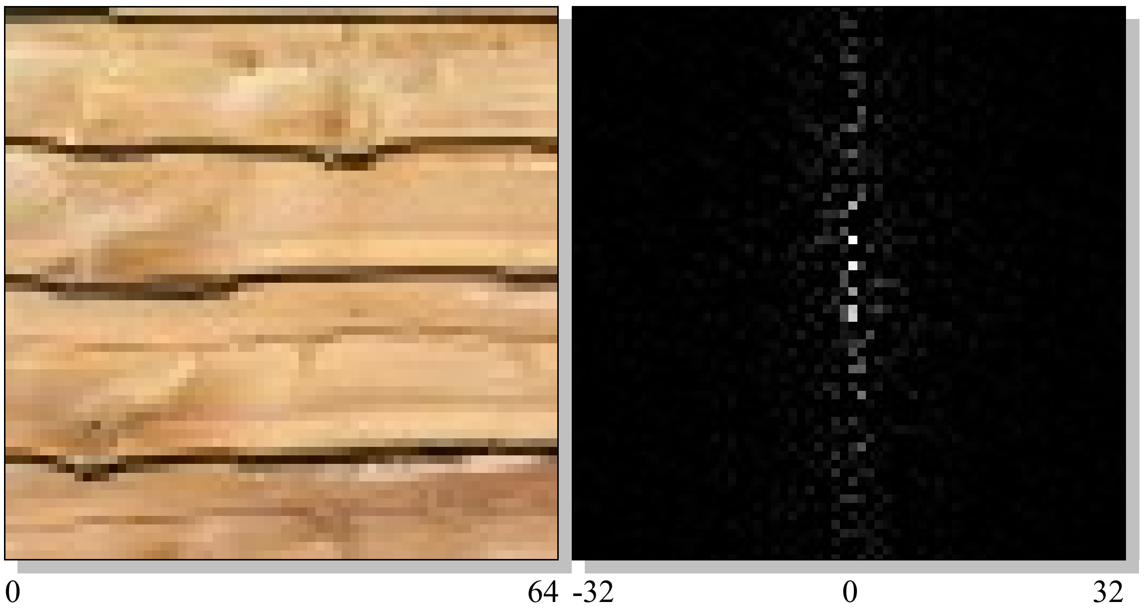


Figure 93: fence

This means that the fence texture should provide good results for a lower number of frequencies than many of the other patterns, although this will depend on texture orientation. The fine detail of the wood grain is likely only to be reintroduced by the technique at much higher frequency levels, well beyond the realms of acceptable performance.

Figures 94 - 97 show the visual results for a resolution of 64x64. In this case, all techniques provide almost indistinguishable levels of detail on the angled surface. The Fourier techniques achieve this level of detail at a frequency level of 74. The NVIDIA technique is the only tested algorithm which displays aliasing, whereas both the EWA and Fourier techniques show strong anti-aliasing properties.

64x64

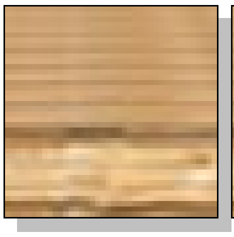


Figure 94: Bilinear

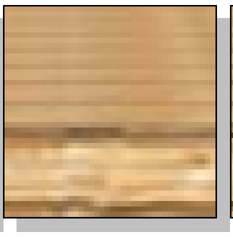


Figure 95: Trilinear

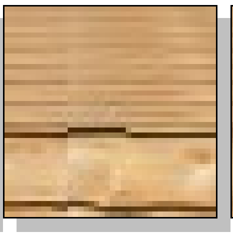


Figure 96: NVIDIA

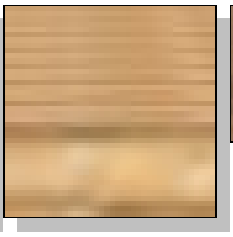


Figure 97: EWA

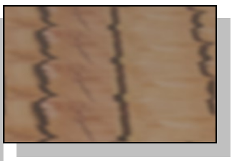


Figure 98: Optical

The same tests were performed at a resolution of 128x128 (Figures 99 - 102) and show that the Fourier techniques loses much of the distant detail, even with the full 256 frequencies.

128x128

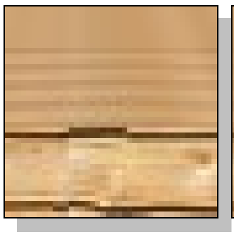


Figure 99: Bilinear

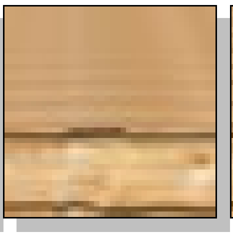


Figure 100: Trilinear

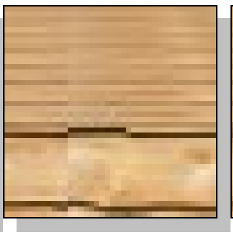


Figure 101: NVIDIA

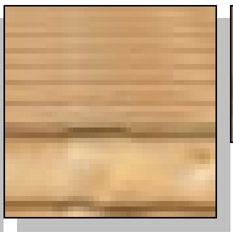


Figure 102: EWA

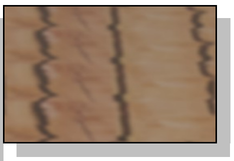


Figure 103: Optical

It must, therefore, be concluded that the fence texture requires more than 256 frequencies for accurate representation at this resolution and above. There is no way of discovering the exact number of frequencies required, hence estimates are presented in Table 15. A resolution of 64x64 requires 1.81% of the total frequencies for an accurate representation and a resolution of 128x128 requires >1.56% of the total frequencies. Therefore, it can be estimated that the other resolutions shall require somewhere in the region of 1.5%-2.0% of the total frequencies available for that resolution.

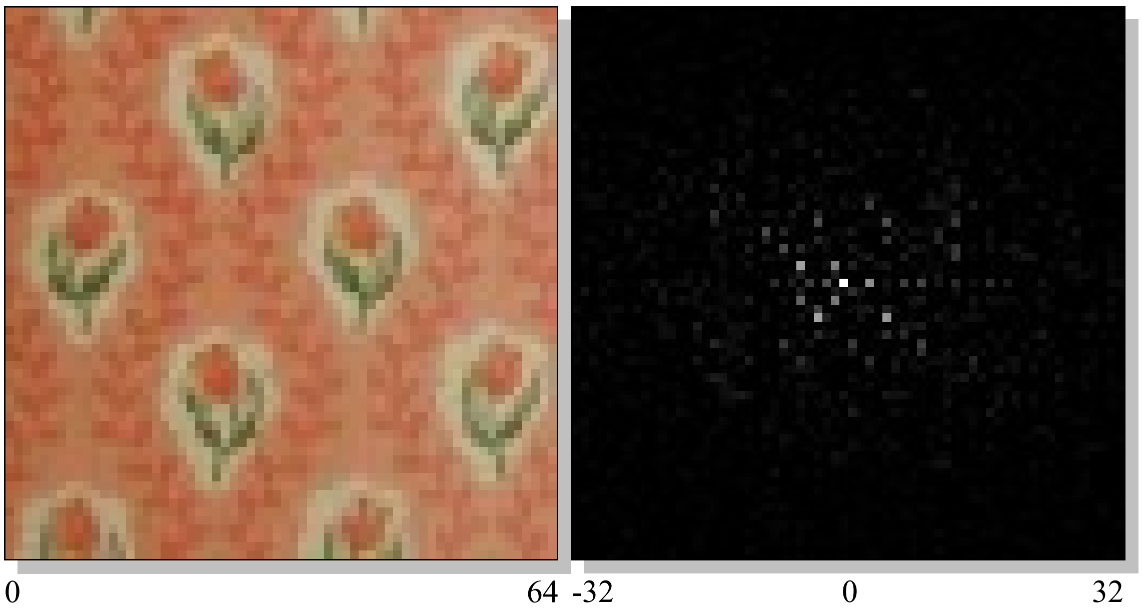
Texture	Height	Width	Texels	Frequencies	% of total frequencies
Fence	64	64	4096	74	1.81%
	128	128	16384	>256	>1.56%
	256	256	65536	983-1311	1.5%-2.0%
	512	512	262144	3932-5242	1.5%-2.0%
	1024	1024	1048576	15729-20972	1.5%-2.0%

Table 15: Frequency requirements for the fence texture

4.4.3.2.5.3 Flowers

The next texture examined is one example of the kind of textures included to thoroughly test the Fourier technique, a flower pattern. The wide scattering of frequencies shown in Figure 104 b) should prove a challenge for the Fourier technique, which relies on a few dominant frequencies being able to sufficiently represent an image.

Figure 104 allows closer inspection of the flowers texture. Figure 104 a) shows the original texture for comparative purposes, and Figure 104 b) shows a Fourier analysis of the texture. The Fourier space representation demonstrates that the majority of the dominant frequencies appear to be around 50% below the maximum frequency. These frequencies are scattered in a non-uniform pattern which may prove difficult for the Fourier technique.



a) Image Space *b) Fourier Space*

Figure 104: Flowers 64x64

As expected, Figures 105 and 106 show that the Fourier techniques struggle with this kind of texture. This is due to the 256 frequencies limit the hardware imposes. The differences in detail retention between the Fourier and NVIDIA techniques demonstrate that an insufficient number of frequencies can be used in the current hardware system, to accurately represent this texture. It also demonstrates that, due to the scale of the difference in detail, that a relatively small number of extra frequencies could provide more competitive results.

Estimating the number of frequencies required is difficult in this case, as even the lowest resolution tests did not provide enough detail to record results on which to base these estimations. Table 16 illustrates known frequency requirements for this texture.

64x64

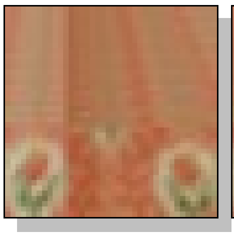


Figure 105: Bilinear

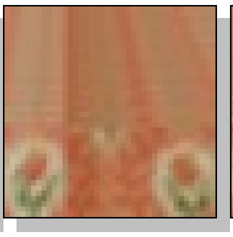


Figure 106: Trilinear

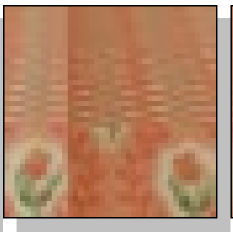


Figure 107: NVIDIA

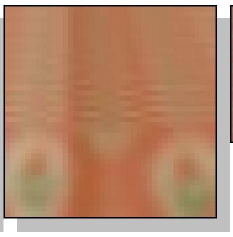


Figure 108: EWA

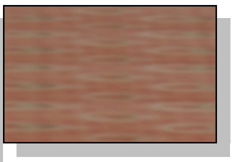


Figure 109: Optical

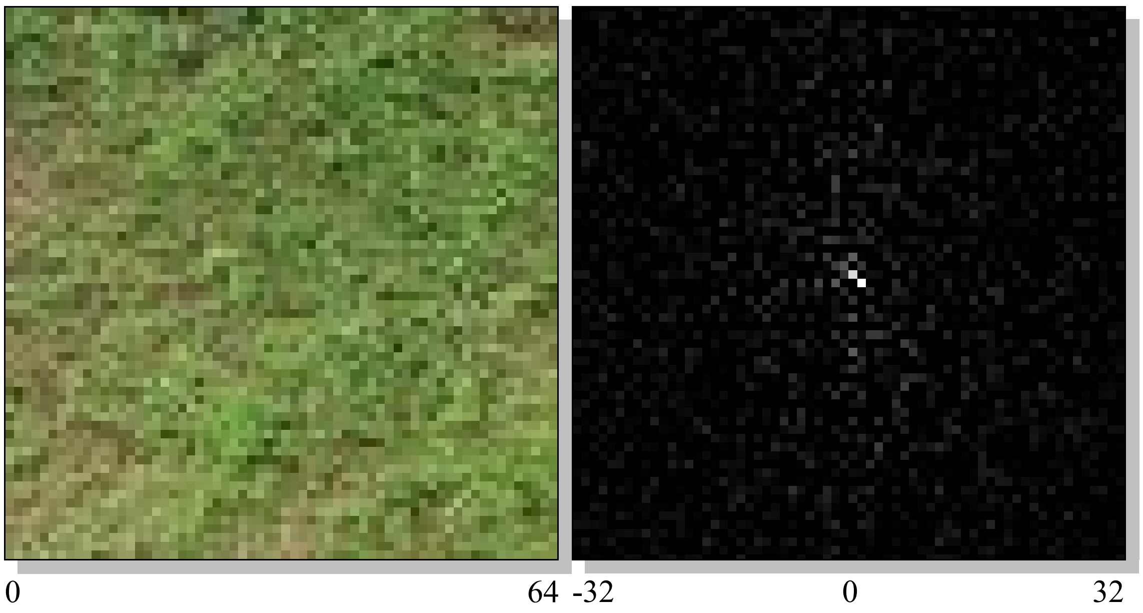
Texture	**Height**	**Width**	**Texels**	**Frequencies**	**% of total frequencies**
Flowers	64	64	4096	>256	>6.25%
	128	128	16384	>256	>6.25%
	256	256	65536	>256	>6.25%
	512	512	262144	>256	>6.25%
	1024	1024	1048576	>256	>6.25%

Table 16: Frequency requirements for the flowers texture

4.4.3.2.5.4 Grass

The grass pattern is one example of the kind of textures which are often found in video games. Taking a closer look at the exact texture chosen. Figure 110 a) shows the original grass texture for comparative purposes, and Figure 110 b) shows a Fourier analysis of that same grass texture.

The Fourier space representation illustrates that, in this pattern the majority of the frequencies making up the image form a random pattern. This will be very difficult for the Fourier technique to reproduce accurately. The fine detail in the source image will also prove problematic for the other techniques.



a) Image Space *b) Fourier Space*

Figure 110: Grass

Figures 111 - 114 show that the Fourier techniques cannot quite match the detail of the NVIDIA technique for this texture. This could be rectified by raising the 256 frequencies limit imposed by the hardware, particularly as the differences are minimal in this case. Table 17 demonstrates that the frequency requirements for this texture are at least 6.25% of the total frequencies available.

64x64

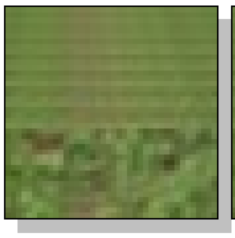


Figure 111: Bilinear

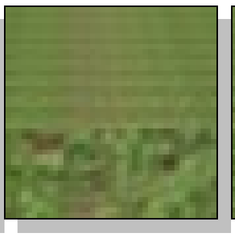


Figure 112: Trilinear

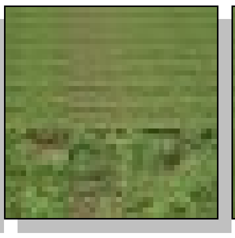


Figure 113: NVIDIA

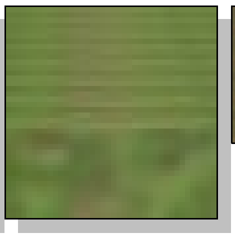


Figure 114: EWA

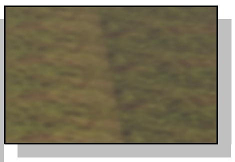


Figure 115: Optical

Texture	Height	Width	Texels	Frequencies	% of total frequencies
Grass	64	64	4096	>256	>6.25%
	128	128	16384	>256	>6.25%
	256	256	65536	>256	>6.25%
	512	512	262144	>256	>6.25%
	1024	1024	1048576	>256	>6.25%

Table 17: Frequency requirements for the grass texture

4.4.3.2.5.5 Hex

The next texture examined is another example of the kind of textures included to thoroughly test the Fourier technique; a Hexagon pattern. The fabric elements of the pattern are also often seen in games. The lack of any particularly dominant frequencies as shown by Figure 116 b) should prove a challenge for the Fourier technique, which relies on a few dominant frequencies being able to sufficiently represent an image.

Taking a closer look at the exact texture chosen: Figure 116 a) shows the original Hex texture for comparative purposes, and Figure 116 b) shows a Fourier analysis of that same Hex texture. The Fourier space representation illustrates that the majority of the dominant frequencies are around 25% below maximum frequency, but the these frequencies are scattered in a non-uniform pattern which may prove difficult for the Fourier technique.

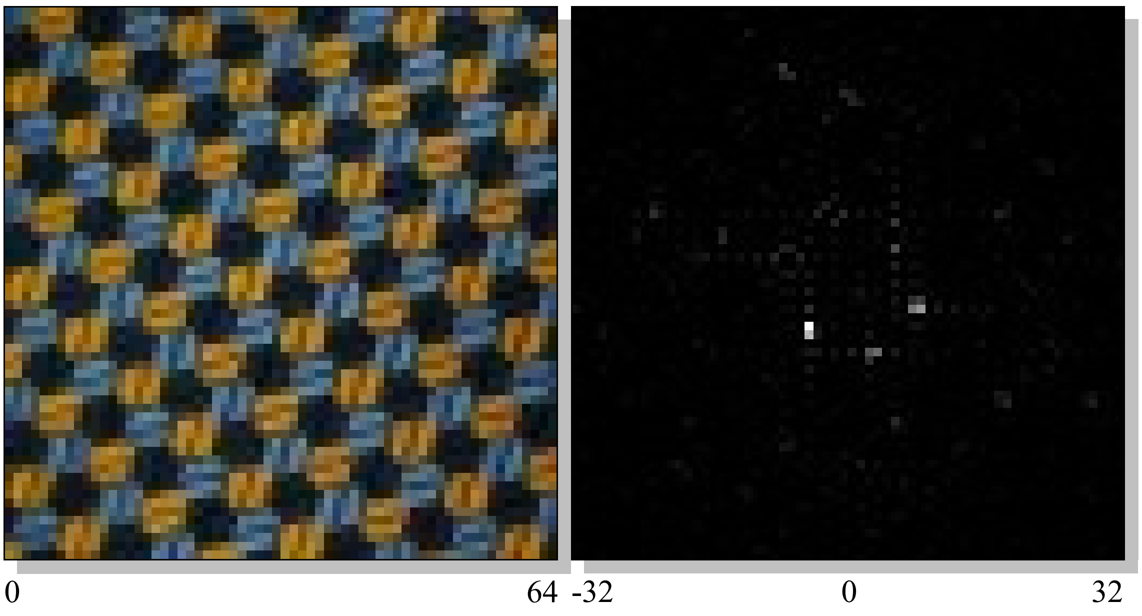


Figure 116: Hex 64x64

The test results illustrated in Figures 117 - 120 show an interesting loss of detail by the NVIDIA technique at the top of the image. This could be an indicator of the NIVIDA techniques optimisations, cutting the workload by lessening the processing for smaller MIP map levels. The NVIDIA technique does however still show better detail retention than the other techniques for this texture size.

64x64

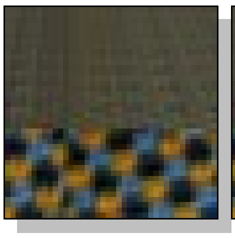


Figure 117: Bilinear

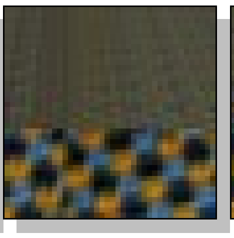


Figure 118: Trilinear

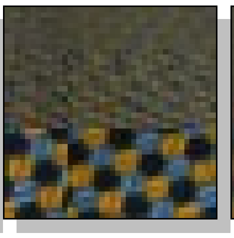


Figure 119: NVIDIA

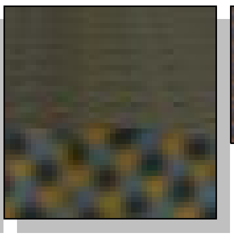


Figure 120: EWA

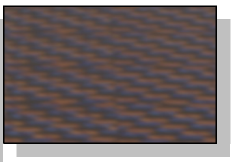


Figure 121: Optical

This texture requires more than the 256 frequencies the hardware is restricted to. As such Table 18 shows only the known frequency requirements of this texture.

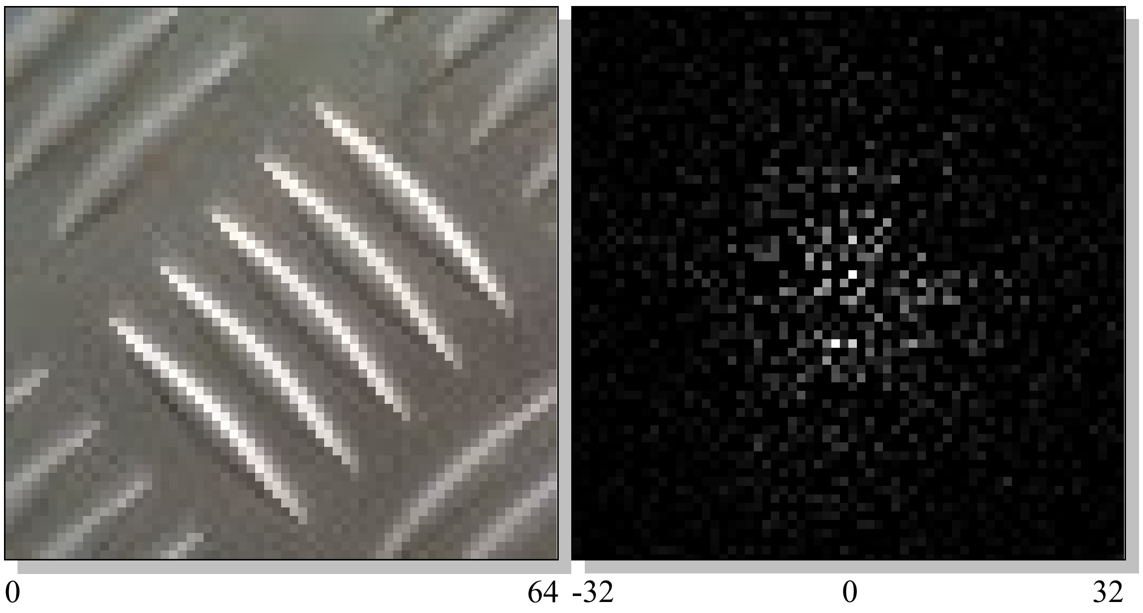
Texture	Height	Width	Texels	Frequencies	% of total frequencies
Hex	64	64	4096	>256	>6.25%
	128	128	16384	>256	>6.25%
	256	256	65536	>256	>6.25%
	512	512	262144	>256	>6.25%
	1024	1024	1048576	>256	>6.25%

Table 18: Frequency requirements for the hex texture

4.4.3.2.5.6 Metal

The metal texture shown in Figure 122 is the next texture to be analysed. This texture is another example of the type of textures that are commonly found in video games.

For comparative purposes the original texture is shown in Figure 122 a), while Figure 122 b) shows a Fourier analysis of that same metal texture. The Fourier space representation shows that, similarly to the grass texture, there is a wide spread of frequencies. Although in this case there are signs of some more dominant frequencies, which should enable the Fourier techniques to cope slightly better with this texture than with the grass.



a) Image Space *b) Fourier Space*

Figure 122: Metal

64x64

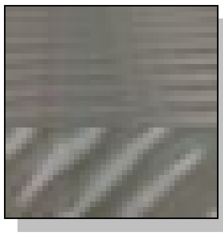


Figure 123: Bilinear

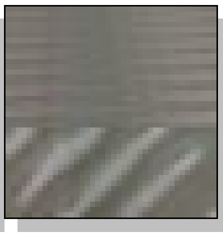


Figure 124: Trilinear

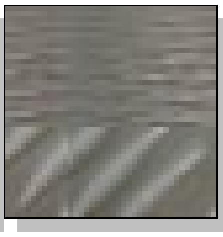


Figure 125: NVIDIA

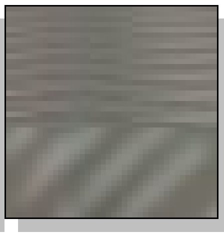


Figure 126: EWA

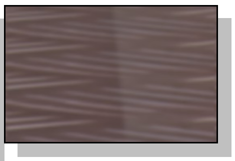


Figure 127: Optical

None of the techniques cope particularly well with the detail of this texture. The NVIDIA technique again provides the most detail in the test images. This illustrates that more than 256 frequencies are required for the Fourier technique to match this level of detail, as shown by Table 19.

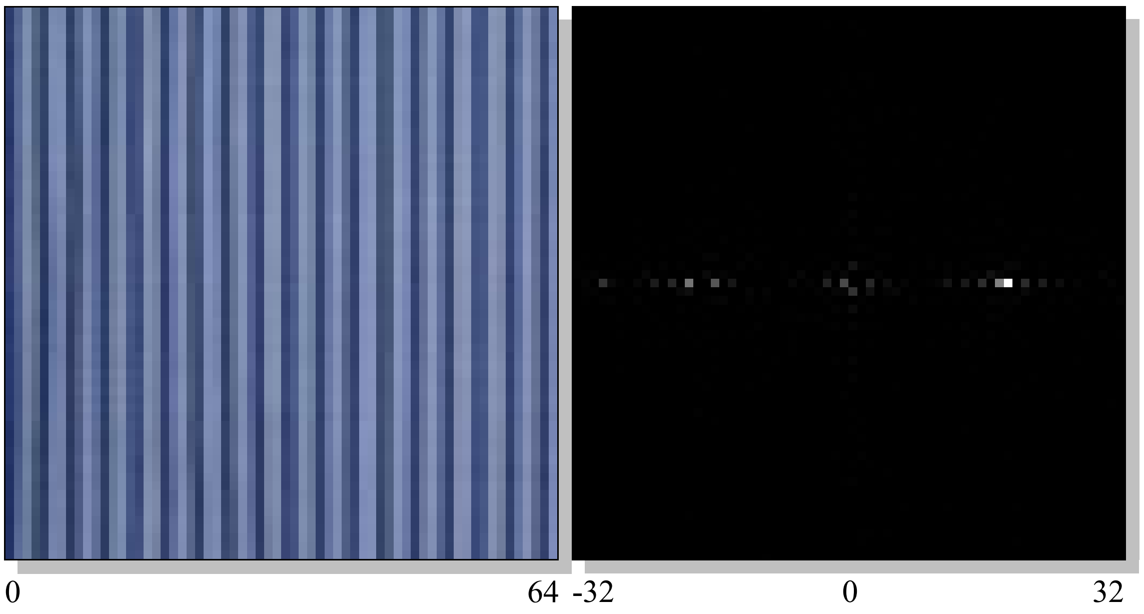
Texture	Height	Width	Texels	Frequencies	% of total frequencies
Hex	64	64	4096	>256	>6.25%
	128	128	16384	>256	>6.25%
	256	256	65536	>256	>6.25%
	512	512	262144	>256	>6.25%
	1024	1024	1048576	>256	>6.25%

Table 19: Frequency requirements for the metal texture

4.4.3.2.5.7 Stripes

The next texture investigated is another example of the kind of textures included to thoroughly test the Fourier technique, a Stripes pattern. The narrow band of frequencies shown in Figure 128 should prove ideal for the Fourier technique, which relies on a few dominant frequencies to enable an accurate representation of the source image.

Figure 128 takes a closer look at the exact texture chosen. Figure 128 a) shows the original Stripes texture for comparative purposes, and Figure 128 b) shows a Fourier analysis of the texture. The Fourier space representation illustrates that the majority of the dominant frequencies are on the horizontal plain in Fourier space, this should prove advantageous to the Fourier techniques, although the texture orientation may determine the extent of any benefits gained. The single plain of frequencies may prove a disadvantage to the Fourier technique for orientations where the dominant frequencies are at odds with the texture orientation.



a) Image Space *b) Fourier Space*

Figure 128: Vertical Stripes 64x64

This texture displays good results under the Fourier techniques, this is demonstrated by Figures 129 - 132. Figure 131 shows a clearly visible level of detail transition in the case of the NVIDIA technique, this is likely caused by an inappropriate optimisation choice creating an overly sharp foreground. All techniques show some degree of motion aliasing on this texture, this is minimal with the EWA and Fourier techniques. The NVIDIA technique however suffers from severe aliasing.

64x64

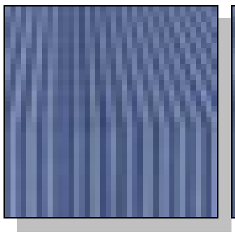


Figure 129: Bilinear

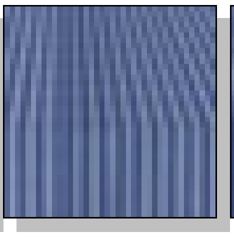


Figure 130: Trilinear

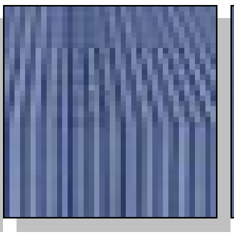


Figure 131: NVIDIA

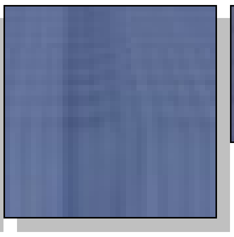


Figure 132: EWA

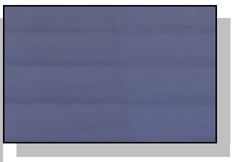


Figure 133: Optical

At a resolution of 128x128 the Fourier technique begins to struggle under the restrictions imposed by the test hardware. Figures 134 and 135 illustrate the need for more frequencies than the 256 frequency restriction allows.

128x128

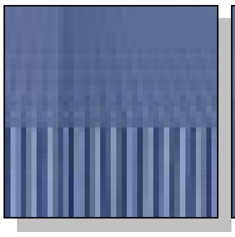


Figure 134: Bilinear

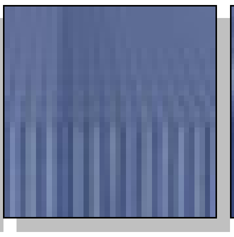


Figure 135: Trilinear

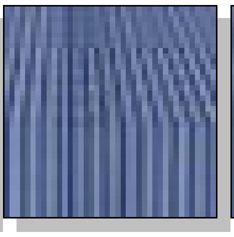


Figure 136: NVIDIA

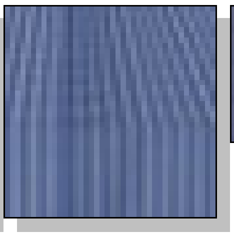


Figure 137: EWA

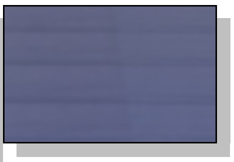


Figure 138: Optical

Table 20 shows the frequency requirements for this texture. Only 20 frequencies are required for a reasonable image quality at a resolution of 64x64. This equates to 160 samples in the case of bilinear filtering. While this is still more than the 128 samples required by the NVIDIA technique, the lack of aliasing under test conditions and relatively small difference in sample requirements makes this a desirable trade off. This is particularly true when the hardware restrictions artificially doubling the sample count are taken into account.

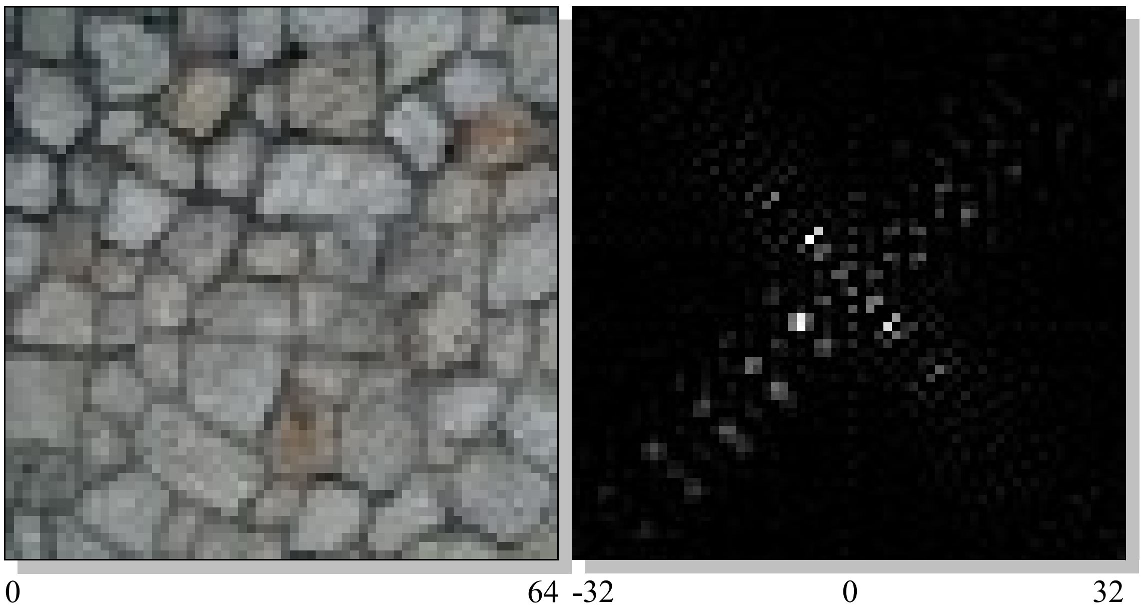
Texture	Height	Width	Texels	Frequencies	% of total frequencies
Stripes	64	64	4096	20	0.49%
	128	128	16384	>256	>1.56%
	256	256	65536	321 - 1022	0.49% - 1.56%
	512	512	262144	1285 - 4089	0.49% - 1.56%
	1024	1024	1048576	5138 - 16358	0.49% - 1.56%

Table 20: Frequency requirements for the stripes texture

4.4.3.2.5.8 Wall

The stone wall pattern is another example of the type of texture that is extremely popular in video games.

Figure 139 a) shows the original stone texture for comparative purposes, and Figure 139 b) shows a Fourier analysis of that same stone texture. This texture shows an interesting dominance of frequencies along the diagonals in Fourier space. This illustrates that there is a dominance of diagonal frequencies. This is despite the uniform appearance which may be expected by examining the pattern in image space, due to the stones all being multi-sided and non-uniform. The lack of any particularly dominant frequencies may cause issues with the Fourier method at lower numbers of frequencies.



a) Image Space *b) Fourier Space*

Figure 139: stone

Figures 140 - 143 displays the results for this texture at a resolution of 64x64. The Fourier techniques display the need for a slightly higher level of frequencies that the hardware allows. However, the Fourier techniques do come close to matching the image detail of the NVIDIA technique. The NVIDIA technique itself is overly sharp, this lessens the actual difference between it and the Fourier techniques, and also results in aliasing.

64x64

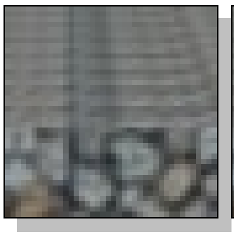


Figure 140: Bilinear

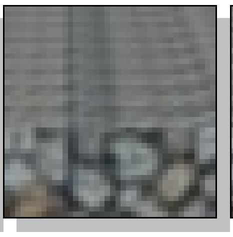


Figure 141: Trilinear

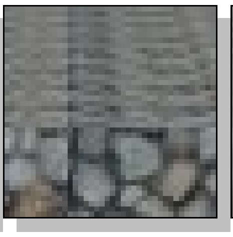


Figure 142: NVIDIA

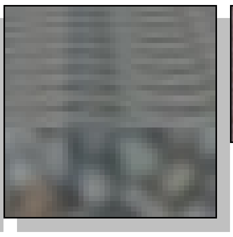


Figure 143: EWA

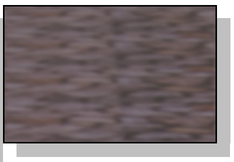


Figure 144: Optical

The frequency requirement estimates for this texture are shown in Table 21.

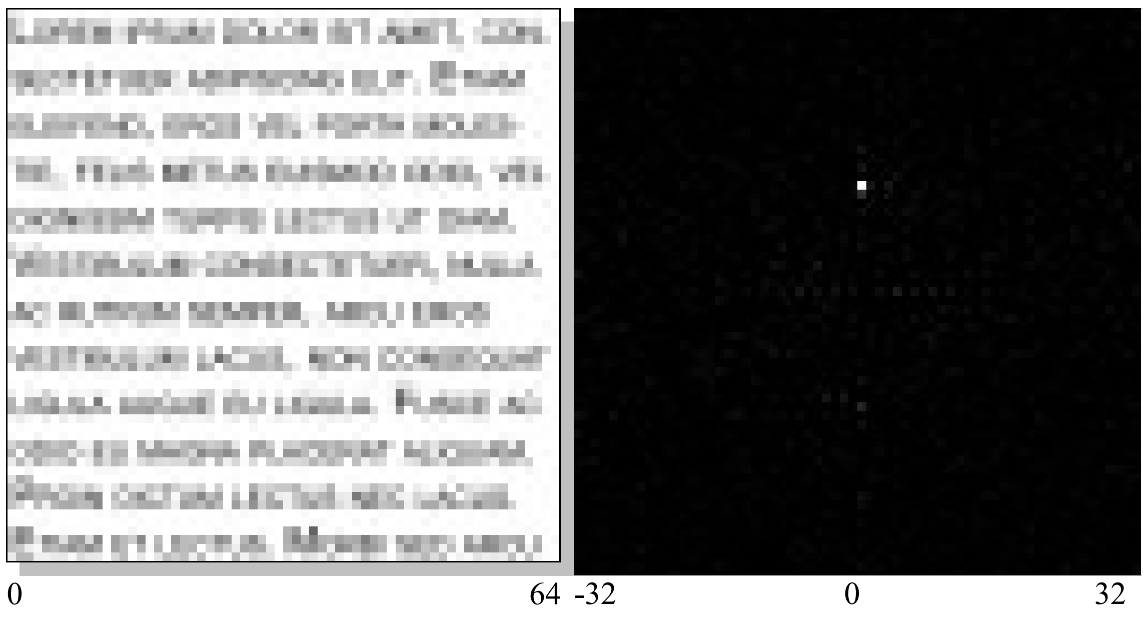
Texture	Height	Width	Texels	Frequencies	% of total frequencies
Wall	64	64	4096	>256	>6.25%
	128	128	16384	>256	>6.25%
	256	256	65536	>256	>6.25%
	512	512	262144	>256	>6.25%
	1024	1024	1048576	>256	>6.25%

Table 21: Frequency requirements for the wall texture

4.4.3.2.5.9 Text

The next texture examined is another example of the kind of textures often used for testing texture filtering algorithms, a text pattern. This is because the fine detail of text is very difficult to reproduce effectively in 3D computer graphics.

Taking a closer look at this texture: Figure 145 a) provides the original Text texture for comparative purposes, and Figure 145 b) gives a Fourier analysis of the texture. The Fourier space representation illustrates that, as you would expect with text, there are no



a) Image Space *b) Fourier Space*

Figure 145: Text

real dominant frequencies or patterns within the image. Text is a widely known problem for anti-aliasing techniques and proves the downfall of many otherwise promising techniques. Hence, this may be the biggest test of the Fourier method.

As predicted all of the techniques struggle with the text texture. At this resolution the NVIDIA technique shows slightly greater detail than the Fourier techniques, although this could easily be rectified by the removal of the 256 frequency hardware limitation. The NVIDIA technique again displays severe aliasing, which is absent from the other techniques under test conditions.

64x64

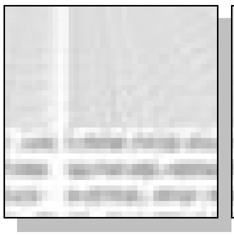


Figure 146: Bilinear

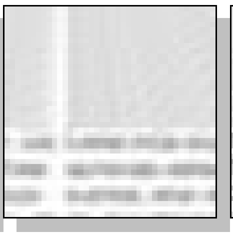


Figure 147: Trilinear

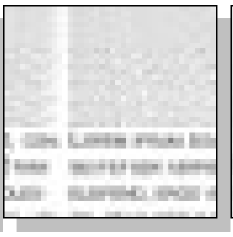


Figure 148: NVIDIA

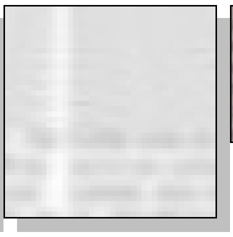


Figure 149: EWA

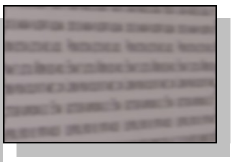


Figure 150: Optical

The estimates of frequency requirements for this texture are shown in Table 22.

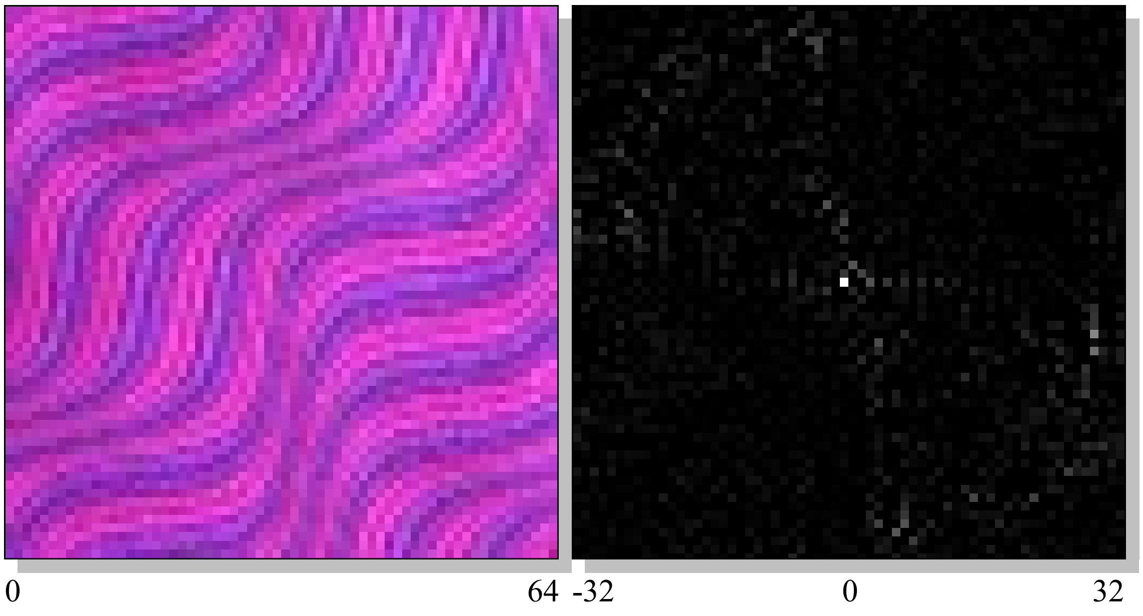
Texture	Height	Width	Texels	Frequencies	% of total frequencies
Text	64	64	4096	>256	>6.25%
	128	128	16384	>256	>6.25%
	256	256	65536	>256	>6.25%
	512	512	262144	>256	>6.25%
	1024	1024	1048576	>256	>6.25%

Table 22: Frequency requirements for the text texture

4.4.3.2.5.10 Wave

The wide scattering of frequencies provided by this texture, and illustrated in Figure 151, should prove a challenge for the Fourier technique.

Similarly to some of the other textures tested, the Fourier technique shows the requirement of a greater number of frequencies than the hardware allows (Figures 152 and 153). This makes comparisons difficult, however the complete lack of aliasing is ample compensation for the slight loss of image detail when compared to the NVIDIA



a) Image Space *b) Fourier Space*

Figure 151: Wave 64x64

technique (Figure 154).

64x64

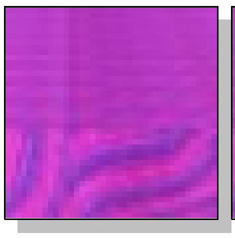


Figure 152: Bilinear

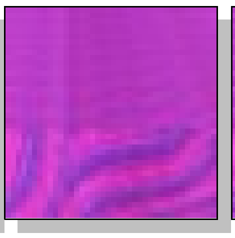


Figure 153: Trilinear

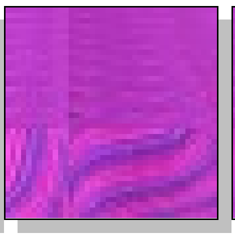


Figure 154: NVIDIA

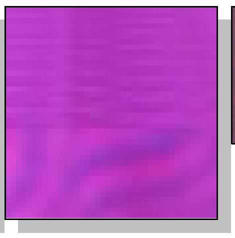


Figure 155: EWA

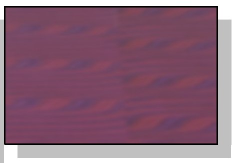


Figure 156: Optical

Table 23 displays the estimates of frequency requirements for this texture.

Texture	Height	Width	Texels	Frequencies	% of total frequencies
Wave	64	64	4096	>256	>6.25%
	128	128	16384	>256	>6.25%
	256	256	65536	>256	>6.25%
	512	512	262144	>256	>6.25%
	1024	1024	1048576	>256	>6.25%

Table 23: Frequency requirements for the wave texture

4.5 Conclusions

The Fourier techniques show between 1% and 15% of the performance of the fixed function NVIDIA anisotropic filtering on the 6800 platform, and EWA shows between 0.13% and 0.15% the performance of the NVIDIA technique. Upon initial inspection this may appear like a large difference. However, it must be remembered that the NVIDIA technique is fully hardware optimised. As such, sample counts and texture memory bandwidth usage make a much fairer comparison. Although, it should be noted that the later tests carried out on the 8600 platform show a drastic decrease in this gap. It can therefore be theorised that, even without this optimisation being applied to the Fourier techniques, hardware is progressing at a pace that will allow these techniques to run at comparable speeds within a few generations.

As illustrated by the results from the 8600 tests[49] this performance gap shall decrease with each GPU iteration, as shaders are optimised and fixed functionality is gradually reduced. It must also be remembered that both the Fourier and the EWA implementations in these tests are simply a proof of concept and have not been

49 Section 4.4.3.2.4

optimised and refined in the same way that the NVIDIA technique has.

With subtle changes to existing API's and some work on code optimisation, the performance of both EWA and the Fourier techniques could be improved significantly. This is a particularly important point because, as illustrated by the performance section, the NVIDIA anisotropic technique has already peaked in terms of optimisation. This indicates that any future performance improvements shall be linked almost totally to hardware progression or compromises in quality.

When comparing performance between these techniques it must also be noted that the NVIDIA anisotropic technique is known to use optimised versions of trilinear filtering whereas the other trilinear based techniques examined use full trilinear filtering. It can be deduced therefore that some of the increased sharpness demonstrated by the NVIDIA technique is provided by this optimisation, along with a little extra performance. The presence of these optimisations also explains the visible layer transitions on some textures as well as some of the aliasing issues suffered by the NVIDIA technique.

Given the continually increasing performance of shaders, optimised versions of the Fourier techniques discussed here could be viable for real-time use within the next few hardware iterations. EWA in real-time is likely to be only a few generations behind. However, neither techniques are currently suitable for real-time without optimisations to the shader code and API's.

As the test results indicate, each technique has advantages and drawbacks. The main advantage of the Fourier technique is the ability to use significantly lower texture resolutions. The Fourier techniques could therefore be used as a means of reducing overall texture size within an application, and therefore significantly reducing texture bandwidth requirements, by providing smaller textures with more detail than would otherwise be possible.

The Fourier techniques achieve similar quality results at low resolutions to higher resolution source textures with other techniques, without introducing aliasing. Figures 157 - 160 demonstrate that, by using a suitable number of frequencies, the Fourier techniques can match anisotropic filtering for quality. This is done using only a 64x64 original texture, compared to 1024x1024 for the NVIDIA technique. Perhaps the most important observation is that the Fourier techniques do this with no recorded aliasing under test conditions, on all but the stripes texture. The stripes texture did however cause issues with all the techniques examined. The NVIDIA technique displayed some aliasing on all techniques, which is in many cases severe particularly in the case of the stripes texture.

The reduction in texture sizes possible with the Fourier technique would be advantageous for systems where memory sizes are limited, where low level caching is available or where memory footprint is an issue. Alternatively the technique could be reserved for use only on lower MIP map levels where it is most effective, while other techniques better suited to higher resolutions are used for the larger levels.

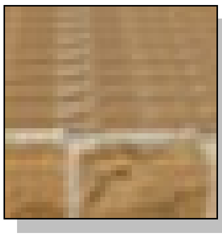


Figure 157: Bilinear 64x64

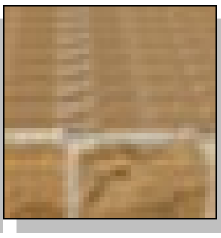


Figure 158: Trilinear 64x64

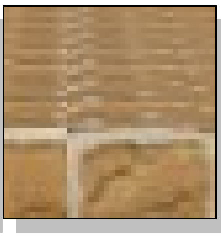


Figure 159: NVIDIA 1024x1024

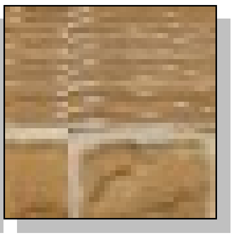


Figure 160: EWA 1024x1024

Another important observation is the differences between the bilinear and trilinear versions of the Fourier technique demonstrated by the tests. The addition of trilinear filtering removes a slight degree of sharpness in all cases. The bilinear version does not suffer from the usual visible MIP map level transitions, these have been suppressed by the technique in the same manner as the aliasing. It can be concluded therefore that the addition of the Fourier technique to a scene removes the need to perform trilinear filtering to hide level of detail transitions. This will provide sharper results and save many samples per cycle, as illustrated by Table 13.

Taking the stripes texture as an example: the NVIDIA anisotropic technique requires 128 samples per pixel. The Fourier technique with bilinear filtering requires 20 frequencies or 160 samples. This indicates that the NVIDIA technique should provide better performance, however this does not take into account the hardware and API limitations which cause a doubling of the sample count for the Fourier techniques. If these could be worked around, perhaps by the provision of customisable texture fetch routines or the ability to increase the number of components per texel, then this sample count could be halved. In the example above, this would mean a new sample count of

80, which is a significant improvement over the 128 samples required by the NVIDIA technique. Although this does still leave the extra bandwidth requirements of the Fourier technique, these are negated by the ability to greatly reduce texture sizes.

The current hardware imposed limitations, particularly on the number of frequencies, indicates that the development of hybrid methods may be preferable in the short term. These hybrid methods could intelligently choose the appropriate filtering method based on texture content and angle.

While EWA represents the ultimate eventual aim for any games or hardware company looking to minimise aliasing and maximise texture quality, the Fourier techniques presented here when used in conjunction with existing techniques could prove an effective real-time solution. Given the kinds of optimisations already in use within these techniques a hybrid method which combines the sharpness of the NVIDIA techniques images with the lack of aliasing shown by the Fourier technique could provide a noticeable improvement over current techniques for performance, memory footprint and image quality.

One drawback of the Fourier techniques is the large amount of pre-processing required to generate the Fourier sets. This is not deemed to be a major failing of the technique as look-up tables and similar techniques, which also require heavy pre-processing of data, are commonly used and this type of work is easily done in batch.

Chapter 5: Bump Mapping

Bump mapping is a technique which was first introduced in 1978 by James F. Blinn [81]. It was designed to simulate wrinkled or bumped surfaces in two dimensional computer graphics systems. Bump mapping, in more general terms, is a method that is used to imply more detail and depth in a 3D models geometry than actually exists. This eliminates the need for extra polygons which would otherwise be required to achieve the same level of detail, thus potentially saving a great deal of storage space and processing time.

Most bump mapping techniques work by perturbing the illumination of a surface on a per-pixel basis, hence creating the illusion of bumps. This means the underlying geometry is not altered in any way[50].

There have been many variations of Blinn's work used, some of which are more successful than others, see below for a discussion of many of them. Real bump mapping (as defined by Blinn [81]) uses per-pixel lighting with a lighting calculation at each pixel based on perturbed normal vectors, but this is computationally expensive compared to some of the other variations.

50 See [82] for an example of something that does alter the underlying geometry.

Bump mapping and its many variations have, in recent years, become a commonplace feature in computer games. This has come via fixed functionality and shaders on modern consumer level graphics hardware. Bump mapping and many of its derivatives are now considered to be relatively common effects utilised to add realism to a scene. This has of course triggered much interest in the area for games and hardware companies, it has however, at the same time, meant less interest in more academic fields. This has arguably had the effect of enabling less innovation in new techniques while at the same time allowing more constant improvements to existing methods.

5.1 Existing Solutions

Many techniques exist to tackle the problem of bump mapping, a selection of the best and most influential are presented and examined here.

5.1.1 Emboss Bump Mapping

Emboss bump mapping is a variety of Fake Bump Mapping, which is an umbrella term to describe a number of techniques used to approximate Blinn's real bump mapping.

Emboss bump mapping involves using a monochrome version of the original texture map, shifting it in a particular direction[51], subtracting it from the original texture, shifting it back then blending it with the original texture giving the illusion of shadow

51 Determined by the direction and intensity of the light source.

on one side and highlights on the other. This means that Emboss bump mapping is similar in principal to the shift and subtract operation in image processing [48]. This technique relies heavily on the concept of tangent space, as do many other visual effect techniques such as Anisotropic lighting and Normal Mapping.

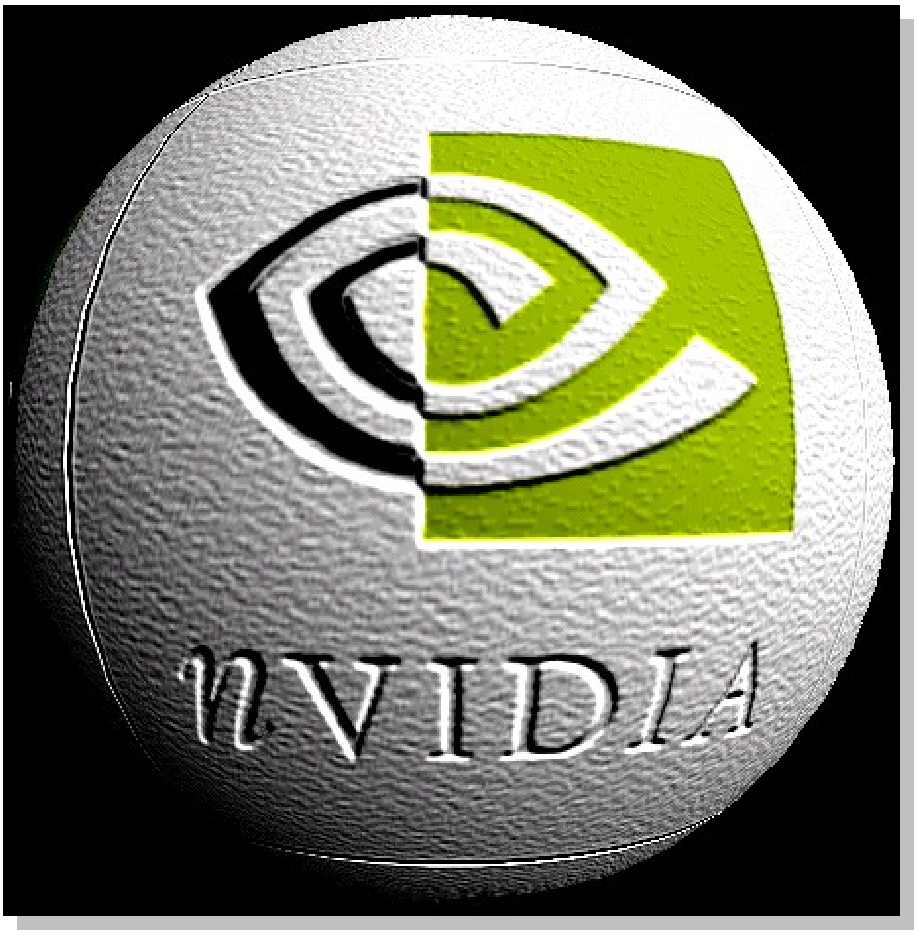


Figure 161: Emboss Bump Mapping [83]

The mathematics behind Emboss bump mapping is as follows:

$$C=(L\cdot N)D_l D_m$$

where:

C is the final colour

L is the light vector

N is the normal vector

D_l is the light diffuse colour

D_m is the material diffuse colour

As indicated by Gold [83], Emboss bump mapping approximates $(L \cdot N)$

This technique relies on the observation that subtracting a grey-scale height map from a slightly shifted version of itself gives the illusion of an embossed surface, where the direction of the shift gives the appearance of illumination. "Mathematically, this subtraction of the height map is an approximation of the derivative of the height map in the direction of the shift" [84].

Emboss bump mapping is a GeForce2 era technique which uses texel index alteration to simulate bumps and is a form of embossing. This technique was the first to be widely utilised in games, with early hardware support provided to simulate embossing via multiple alpha-blending passes. Despite its early popularity the technique quickly fell out of favour with developers and as such has been rarely used in games since. This decline in popularity was largely down to the fact it was difficult to achieve good results due to emboss bump mapping being a per-polygon technique, which produces artefacts and is limited to monochrome lighting systems. The technique was incompatible with multi-coloured lighting meaning emboss bump mapping could not be used if multi-coloured lighting was required. The increasing capacity and capabilities of the hardware meant that other techniques, which provide much better results, were quickly available for little extra relative cost.

Although having the advantage of being a simple technique, emboss bump mapping only produces correct results on still objects “If you're trying to bump the ripples and waves in a water to reflect light realistically, it cannot be done because the texture changes sporadically with the dynamic polygons waving as water. This is where surface curvature becomes much of a problem. Simulating environmental effects is not possible with embossing.” [85].

Advantages	**Disadvantages**
Very simple technique	Diffuse lighting only, no specular component
	Causes under-sampling artefacts
	Cannot simulate objects that need to be transformed
	Limited to monochrome lighting

Table 24: Emboss bump mapping: advantages and disadvantages

5.1.2 Normal Mapping

Normal mapping[52] is one of the techniques widely used in the fixed function pipeline. Of the techniques in common use today, normal mapping is the one which most closely resembles Blinn's original method [81]. Cook 1984 [86] first described the normal map format based on Blinn's Bump Mapping technique.

52 Also known as DOT3 Bump Mapping or DOTPRODUCT3 Bump Mapping.

While other bump mapping methods perturb the existing normal of a model, normal mapping replaces the normal entirely via the use of a 'normal map'. A normal map is an image/texture which is used to represent the direction of the normals on a given surface.

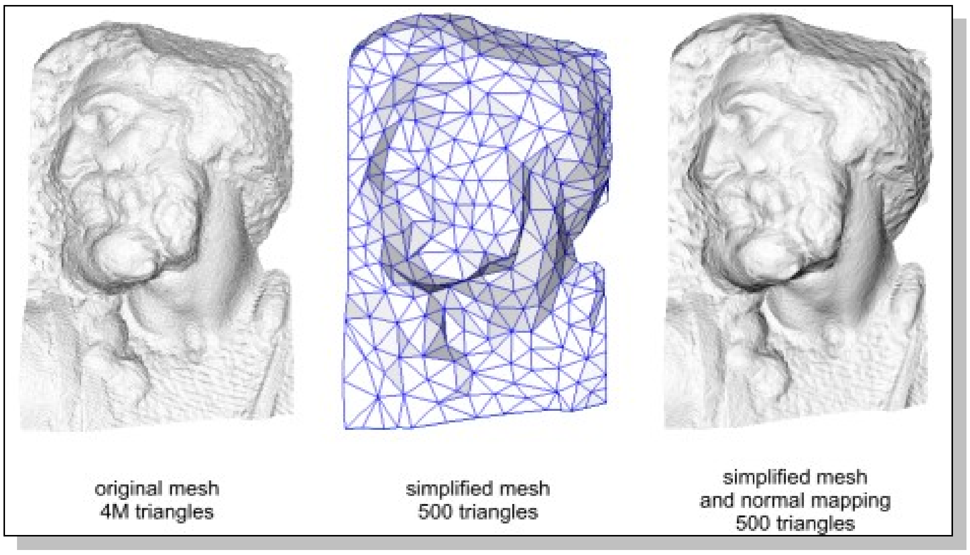


Figure 162: Normal Mapping [87]

There are two varieties of normal mapping: object space and tangent space. These differ only in the co-ordinate systems used to measure and store the normals. Tangent space is the most popular, at the time of writing, as many other current techniques such as Anisotropic lighting rely on tangent space co-ordinates.

This technique can greatly enhance the appearance of a model with a low polygon count, by exploiting a normal map generated from a higher resolution model. While this idea of taking geometric details from a high resolution model had been introduced by Krishnamurthy and Levoy [88], where this approach was used for creating displacement maps over nurbs, its application to the more common triangle meshes came later. The year 1998 saw two papers presented which put forward the idea of transferring details as normal maps from high to low polygon meshes. Cohen et al. [89] presented a constrained simplification algorithm that tracks how the lost details should be mapped over the simplified mesh. Cignoni et al. [90] presented a slightly simpler approach that de-couples the high and low polygonal mesh. This allows the recreation of lost details independent of the low polygon models creation method. The latter of the two approaches is the one used by most of the currently available tools, albeit with some minor variations.

The normal map is used to represent "per-pixel 3-space normals (N) to a surface" [84]. Therefore, by using an extra RGB bitmap (normal map) textured across the model, more detailed normal vector information can be encoded. This is achieved by mapping each colour channel in the bitmap (red, green and blue) to a spatial dimension (X, Y and Z). In the case of object-space normal maps these spatial dimensions are relative to a constant co-ordinate system. However, for the more popular tangent space normal maps they are relative to a smoothly varying co-ordinate system. This system is based on the derivatives of position with respect to texture co-ordinates. This enables much more detail to be added to the surface of a model, especially in conjunction with advanced lighting techniques such as the Phong illumination [91].

For example, this information can be used to calculate the Lambertian (diffuse) lighting intensity of a surface (I), where the unit vector from the shading point to the light source (L) is dotted[53] with the unit normal (N) as shown by the formula:

53 By taking the dot product, also known as the scalar product or inner product.

$$I = N \cdot L$$

The most common implementation of normal maps, is used by Valve's Source engine and implemented by hardware in NVIDIA GPUs: The red channel provides the relief of the material when lit from the right, the green is the relief of the material when lit from below, and the blue channel should be the relief of the material when lit from the front. In other words, the XYZ co-ordinates of the face normals are placed in the RGB values of the normal map. Dempski, 2002 [87] provides more detail on Normal Mapping.

Extra performance can be achieved with normal mapping by combining with the technique known as MIP mapping. This technique allows the reduction of the normal map for more distant surfaces, thus lowering the processing burden and allowing the selection of a more appropriate level of detail.

Advantages	**Disadvantages**
Lower processing requirements compared to other techniques which produce similar levels of image quality, when used in conjunction with MIP mapping	Like its predecessors normal mapping is only suitable for simulating small perturbations and is hence inappropriate for very bumpy surfaces
High quality images	
Has enjoyed direct hardware support in home consoles since the X-Box and in home PC's since the GeForce2 era [92]	
Widely used well known technique	

Table 25: Normal mapping: advantages and disadvantages

5.1.3 Environment Mapped Bump Mapping

Environment mapped bump mapping (EMBM) was not implemented in consumer level hardware until the Matrox G400 and first generation ATI Radeon cards. Figure 163 shows an example of EMBM as implemented by ATI [93]. See Dempski, 2002 [94] for more information on Environment Mapping.

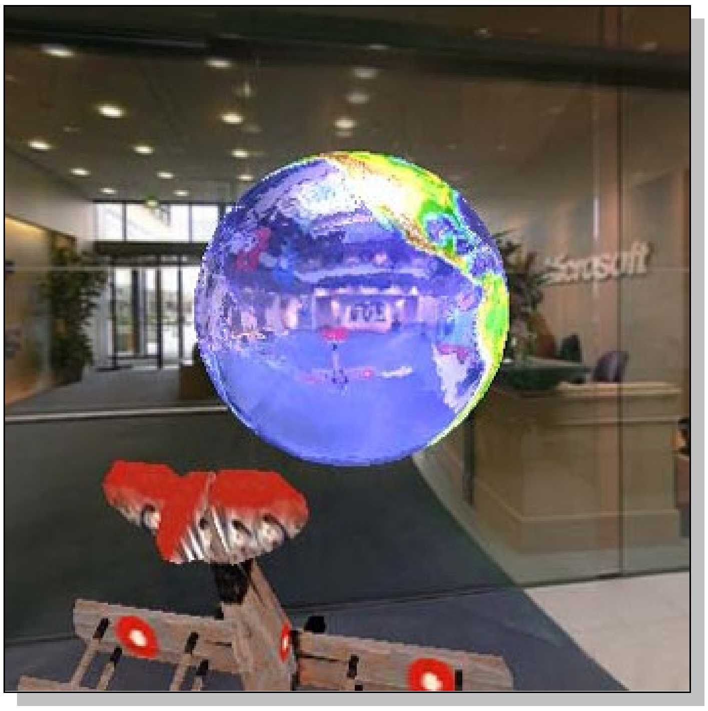


Figure 163: Environment Mapped Bump Mapping [93]

Nuydens [95] discusses a particular type of EMBM, that was fairly common amongst fixed function based graphics hardware. More generally, EMBM can be considered as any bump mapping implementation that is combined with environment mapping. This is a fairly natural combination of techniques as the reflected ray calculated by bump mapping can be used to feed the environment mapping calculations. Many techniques, such as that proposed by Schilling [96], combine these methods.

EMBM is particularly popular for simulating bumpy reflective surfaces via environment reflections; which is the typical use of environment maps.

Advantages	**Disadvantages**
Can be either view-dependant or view-independent	Often suffers from the inappropriate use of antialiasing techniques
Can be used with both diffuse and specular lighting models	
Good effect for reflective bumpy surfaces	

Table 26: EMBM: advantages and disadvantages

5.1.4 Parallax Mapping

Parallax Mapping, was first introduced by Kaneko et al. [97] and inspired by Oliveira and Bishop's work on Relief Textures [98]. Parallax mapping is also known as Offset Mapping or Virtual Displacement Mapping [95]. It is perhaps the most advanced bump mapping technique currently utilised by the games industry, as it offers many of the benefits of displacement mapping for significantly lower cost.

Parallax mapping attempts to overcome some of the shortcomings of Blinn's original Bump Mapping [81] and Cook's Normal Mapping [86] by addressing the issues of motion parallax, or “the apparent displacement of the object due to viewpoint change” [99], and dynamic occlusion.

Parallax Mapping is achieved by offsetting each fragment's texture co-ordinates towards the eye, by a distance which is dependent on the height map value at that location. As such parallax mapping requires that the texture co-ordinates at each pixel are corrected by a given offset to approximate the parallax effect when moving the viewpoint relative to an uneven surface. As stated by Welsh [100], three components are required to compute this offset, the original texture co-ordinate, a value for the surface height and a tangent space vector from the pixel to the viewpoint. The texture co-ordinates are supplied through standard methods, often built into consumer level hardware. The surface height can be drawn from a height map, which correlates to the regular texture map by storing one height value per texel. The tangent space vector from the pixel to the viewpoint is obtained by sourcing the view vector in global co-ordinates by subtracting a surface position from the eye (view) position and transforming the resulting vector into tangent space [100].

While parallax mapping, of all the bump mapping techniques, offers the closest results to displacement mapping, it does still have issues. Steep bumps, for example, are rendered incorrectly due to sampling issues. This means they can appear as parallel sheets of texture rather than actual bumps, this is one of the shortcomings of current techniques which this work looks to overcome. Parallax Mapping also has an inherent problem in accurately depicting shallow angles, such angles can cause severe aliasing. Welsh proposes a solution to this problem [100], however the solution introduces other issues. Welsh suggests the removal of a factor $1/N \cdot V$ for shallow angles, effectively assuming the dot product is zero for those regions. This in turn causes other problems in the form of texture “swim” [101].

As shown in Figure 164 Parallax mapping can be used in conjunction with corrected Z values to create “Z-correct bump mapping” [95], this enables better simulation of displacement mapping. Particularly on the intersection of two objects, where the intersection is transformed from a straight line to one that appears to follow the contours of the bumps. Unfortunately Z-correct bump mapping does have some issues; it can cause severe problems with hardware depth testing, which when implemented in shaders can result in completely disabling the early hardware depth culls common on today's consumer hardware. This can cause large drops in performance and the opportunity for early fragments culls is lost.

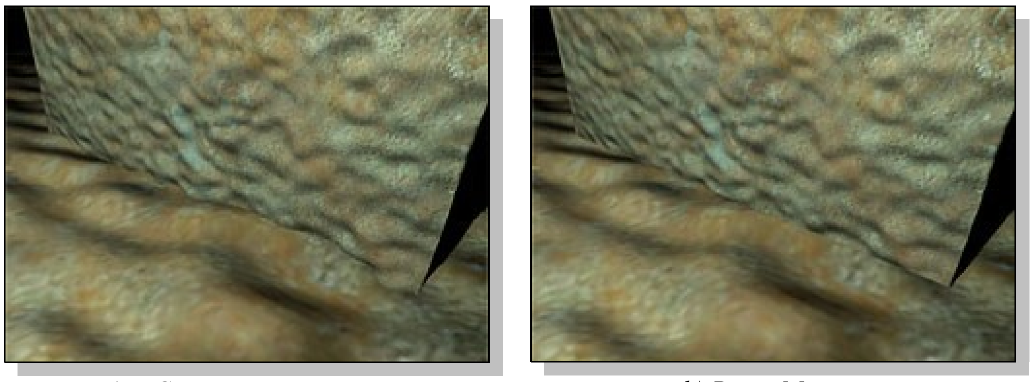


Figure 164: Z-Correct Bump Mapping Vs Bump Mapping [95]

Dynamic parallax occlusion mapping, as introduced at SIGRAPH 2005 and then expanded on in 2006 by Tatarchuk ([99] and [102] respectively), adds another layer of realism to parallax mapping by allowing the addition of soft shadows and uses an adaptive level of detail system to maximise performance. Figure 165 shows Dynamic parallax occlusion mapping compared against standard normal mapping, the visual advantages of this method are clear to see.

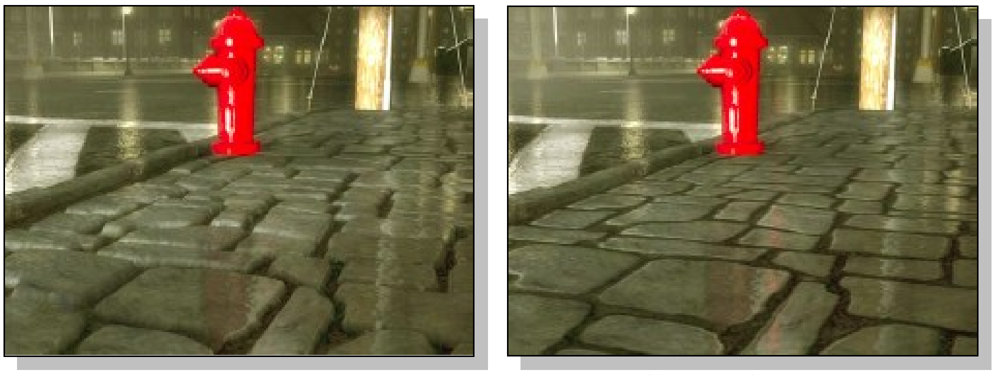

a) Parallax Occlusion Mapping b) Normal Mapping

Figure 165: Parallax Occlusion Mapping Vs Normal Mapping [102]

Advantages	**Disadvantages**
Closest bump mapping method to displacement mapping	Steep bumps are rendered incorrectly, due to sampling issues they can appear as parallel sheets of texture
Better performance than displacement mapping	When used as Z-correct bump mapping can cause issues with hardware depth testing
Can be extended to simulate soft shadows and include z-correct bump mapping	Parallax Mapping has an inherent problem in accurately depicting shallow angles, and can cause severe aliasing. However the solution causes other problems in the form of texture "swim"

Table 27: Parallax mapping: advantages and disadvantages

5.1.5 Displacement Mapping

Displacement mapping, while not technically a form of bump mapping as defined by Blinn [81], is related and belongs to the same family of effects. Generally, bump mapping techniques perturb the surface normal. Displacement mapping, on the other hand, perturbs the actual geometric points of a surface creating real bumps. These bumps, just like real geometry, can cast shadows, occlude other objects and help simulate much more complex geometry than is possible with Blinn style bump mapping.

Displacement mapping first became well known over 20 years ago when it was implemented by the SGI[54] tool RenderMan, which is now synonymous with Disney Pixar. However, Displacement mapping is only now beginning to become viable in real time applications at the consumer level.

Displacement mapping uses height maps to model surface perturbations and modify the surface geometry, meaning displacement mapping actually adds geometric detail to a mesh.

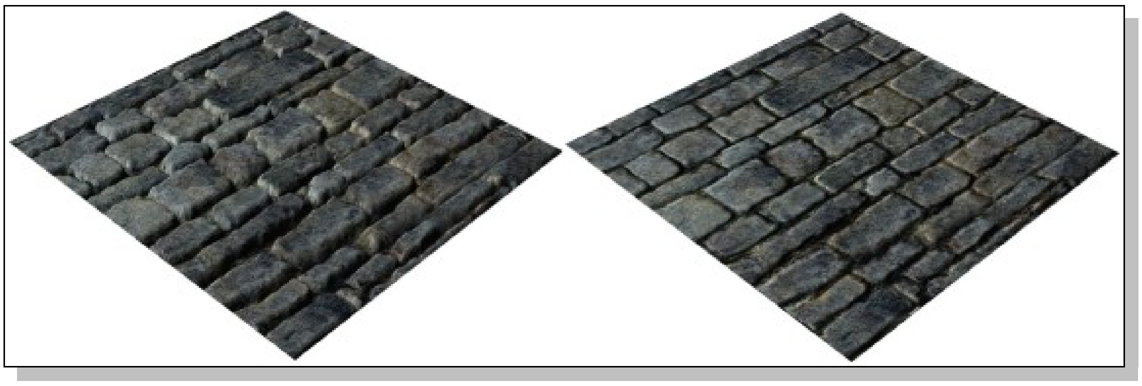


Figure 166: Displacement Mapping Vs Bump Mapping [82]

Fairly high polygon counts are required to achieve a good quality effect from displacement mapping, which leads to poor performance compared to other techniques.

54 Silicon Graphics Inc.

However, with use of an adaptive tessellation scheme [95] this processing burden can be drastically reduced while still achieving a high quality effect.

Advantages	Disadvantages
High quality effect that can simulate much greater depth than bump mapping	Slow due to the requirements of processing higher polygon counts
Hardware acceleration becoming much more commonplace, particularly with the advent of shaders	Results can be poor for lower polygon counts
Can be applied offline	

Table 28: Displacement mapping: advantages and disadvantages

5.2 Anti-Aliased Bump Mapping

While many techniques exist to approach the modelling of wrinkled surfaces, not much progress has been made in handling the aliasing problems that ensue from the use of such techniques. Standard anti-aliasing techniques tend to be designed with texture aliasing in mind, where the texture in question is applied to a flat surface. Often these techniques, such as MIP mapping, are naively applied to bump mapped surfaces. This creates theoretically incorrect results due the the non-linearity of the bump mapping process.

Current systems which employ bump mapping techniques commonly attempt to solve texture aliasing via MIP mapping, perhaps in conjunction with other techniques such as trilinear filtering. However MIP mapping proves inadequate for even simple textures at shallow angles, and suffers similar issues even when the texture is face on, where bump mapping is employed. This is because often the pixel footprint will contain many different normal vectors and using MIP mapping reduces that to a single averaged normal. This becomes a problem when used in conjunction with common techniques such as Phong [103] style lighting (where "Phong style" describes a multitude of lighting techniques currently in common use which are based wholly or partially on Phong's model) and environment mapping. Both these, and similar, techniques rely on a non-linear relationship between the normal vector and final pixel value. Therefore because averaging does not commute with non-linear transformations the results shall be theoretically incorrect. This is demonstrated by Figure 167 which has been taken from Cant and Langensiepen's work on "efficient anti-aliased bump mapping" [1].

Figure 167 a) provides a reference image showing the texture at full scale. Figure 167 b) shows a theoretically correct representation produced by the averaging of shades. Figure 167 c) show the deficiencies of the MIP mapping (normal averaging) techniques causing an overly smooth surface and therefore a glossy final image. Figure 167 d) shows Cant's technique which takes into account the spread of vectors within each pixel, this is much closer to the theoretically correct Figure 167 b).

While previous investigations have looked into this problem, each has defects or poses significant barriers to practical integration with current hardware and/or real-time performance. Becker and Max [104] present a technique which uses a pre-computed shade value stored over a variety of lighting and viewing angles. Therefore, their technique requires a bi-directional reflection distribution function (BRDF) to be stored

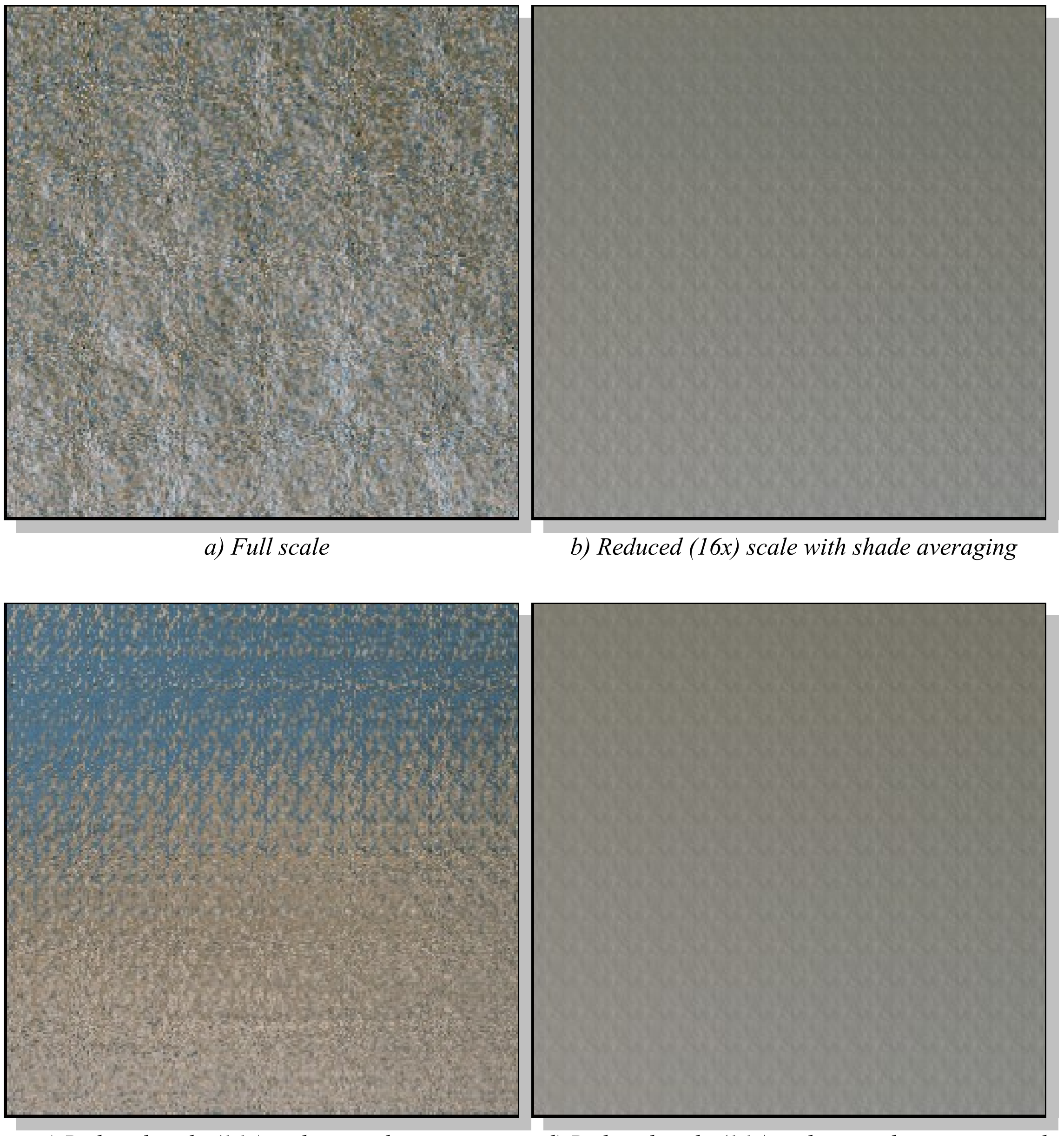

a) Full scale
b) Reduced (16x) scale with shade averaging
c) Reduced scale (16x) with normal averaging
d) Reduced scale (16x) with normal averaging and modified spread

Figure 167: Comparison of anti-aliasing techniques for normal mapped images: Environment mapped fractal frosted glass texture [1]

per texel. Naturally this requires a large amount of storage space which can be prohibitive to any practical use of the technique. Their proposed solution to the high storage requirements is to switch to standard normal mapping at larger scales. However this proved impractical for real time implementation. Further work has been done on reducing the storage requirements (e.g. Sloan et al. [105]), however all have significant drawbacks which either preclude practical implementation on current consumer grade hardware or do not solve the problem.

Fournier [106] proposed another solution; by representing the effect of a large number of evenly distributed vectors by a single vector. By allowing more than one vector to be used for each texel he solved the problem of bunching, however the technique is limited to Phong style lighting models. The least squares fitting method was used to find the set of vectors, however this required a large amount of offline processing.

Olano and North [107] proposed an alternative solution based on Gaussian distributions, which can be linearly combined when texels are averaged. This has the advantage of removing the need for a filtering stage, thus providing performance advantages of the techniques mentioned previously. However this technique suffers from a lack of support for Fournier's [106] concept of a “multiple surface”. This is due to the fact that a combination of Gaussian's always combine to a single Gaussian centred around the average direction. Essentially this precludes the use of Olano and North's technique for certain types of normal map where more than one normal vector is required per texel, as demonstrated by Figure 168.

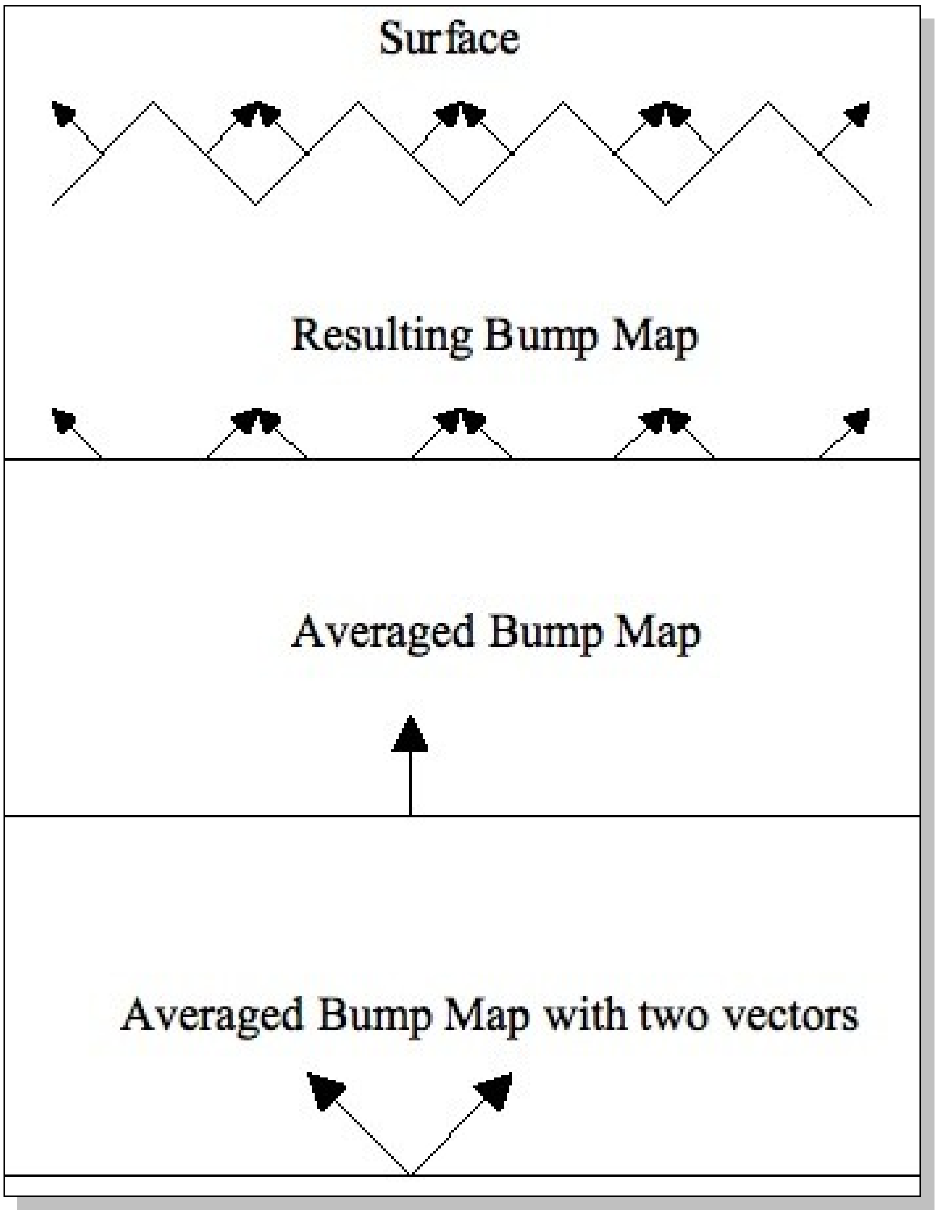


Figure 168: Example of a surface which requires multiple vectors [1]

As an alternative to this Schilling [108] proposed the concept of a roughness matrix to define an anisotropic distribution in place of the Phong peak. However, while this does provide improved rendering of certain types of bump map, it has limited flexibility compared to other work in the area. Schilling [96] later extended his work on anti-aliasing bump maps to environment map based systems. However, as noted above, his method lacks “multiple surface” capability.

As observed by Cant and Langensiepen [1], no previous technique which takes into account “multiple surfaces” provides a suitable candidate for real-time implementation on consumer level graphics hardware. It is from this point that they introduce their multi-vector “efficient anti-aliased bump mapping” technique.

5.3 Super Bump Mapping

With Super Bump Mapping (SBM) the issue of efficient anti-aliased bump mapping, using multiple vectors, is addressed. Currently single vector techniques are commonplace, these techniques suffer various deficiencies including the inability to correctly represent certain types of bump map as discussed previously. This new work is based on the methods of Cant and Langensiepen [1], who introduce a technique based on multiple vectors where the vectors are derived through competitive learning techniques.

5.3.1 Efficient anti-aliased bump mapping

A variation of Cant and Langensiepen's multiple vector technique (“Efficient anti-aliased bump mapping” [1]) is implemented in shader based graphics hardware for the first time. Previously only a two vector approximation, with significant compromises limiting flexibility, has been shown to be possible by Cant and Langensiepen [1].

Cant and Langensiepen's [1] approach bases each vector in the multi-vector set on an original group of vectors within the source normal map. These vectors are computed via competitive learning techniques based on the work of Kohonen [109]. As defined by Cant and Langensiepen, for each texel averaged a set of exemplar vectors is generated. These exemplars are initially generated with random values before they are trained to represent groups of vectors from the original set. Cant and Langensiepen provide a structured English description of the process, this is shown in Figure 169.

While a random seed is used measures are employed to avoid falling into local minima and to speed convergence. Modifying all exemplars as opposed to just the winning one avoids the problem of random seed choice which causes long convergence times due to distant starting points. Exemplars that never win are replaced by actual vectors, these are the least well represented in the exemplar set, thus avoiding falling into local minima.

Cant and Langensiepen also employ the concept of a “neighbourhood” to provide greater “organisation” to the results, this means that the exemplars which fall closest to the “winner” are modified more strongly that those further away. This enables better results to be obtained from more structured source normal maps.

The vectors are then grouped by dot product values, and weightings are calculated by dividing the number of exemplars within a group by the total population in a particular texel. Empty groups are ignored by being assigned a weighting of zero.

Reduction of the average Q values in successive generations is monitored to detect

```
For each texel
    For each learning cycle
        For each original vector
            Compare with exemplar vectors
            Make closest ("winning") exemplar closer to original vector
            Make all other exemplars slightly closer to original vector*
        If any exemplar not closest to any vector in this cycle
            Replace by least well-matched original vector
    For each original vector
        Place in group associated with closest matching (final) exemplar
    For each group
        Calculate average vector, weight and Q.
        If using environment maps generate MIP (env) map level from Q
        Else if using Phong model generate index "n" from Q

*Optionally a "neighbourhood" mechanism is used here with the adjustment
to the non-winning exemplars falling away in inverse proportion to their
distance (within the set) from the winning exemplar.
```

Figure 169: Structured English description of the competitive learning process

convergence, once detected the learning rate is reduced. The learning cycles are finished when group membership remains constant. Cant and Langensiepen note that very few cycles are needed for convergence to occur and that entire MIP map sets can be created in a few minutes on modest hardware.

5.3.2 Super Bump Mapping Technique

This technique builds upon the foundations of Cant and Langensiepen's competitive learning technique along with standard normal mapping style bump mapping. As such it should lend itself well to integration with modern graphics hardware, as well as providing compatibility with other common techniques such as Parallax Mapping [99] and Phong [103] style lighting techniques.

A Cant and Langensiepen style competitive learning system is used to output multiple vector sets to text files, which are termed SBM files. The system is designed to output differing numbers of vectors depending on the MIP map level, as demonstrated by Table 29 when taking a 256x256 texture as an example. This is because lower MIP map levels will require greater assistance from the multiple vectors in retaining detail, while, as shown by Cant and Langensiepen, lower numbers of vectors are sufficient for the higher levels.

Level	Size	Vectors
0	256x256	1
1	128x128	2
2	64x64	3
3	32x32	4
4	16x16	5
5	8x8	6
6	4x4	7
7	2x2	8
8	1x1	9

Table 29: Vectors per level

These files are read into the OpenGL system where the sets are split between two OpenGL textures (texture2D). This is organised as shown by Figure 170, where each level is subdivided into an appropriate number of vectors. Level 0 is the same size as a single full sized 2D texture, each subsequent level (1, 2, 3, etc.) is exactly half the size of its' preceding level. Therefore, level 2 is exactly half the size of level 1, level 3 is exactly half the size of level 2 and so on. This is because the size of each level is the number of vectors multiplied by the total number of pixels in the MIP map level.

Once loaded into OpenGL textures, the SBM information can be passed to the shaders, where standard environment mapping is performed with the addition of a few considerations required for Super Bump Mapping.

The texture structure used to squeeze the system into standard OpenGL textures requires some manipulation of texture co-ordinates in order to properly access the SBM data.

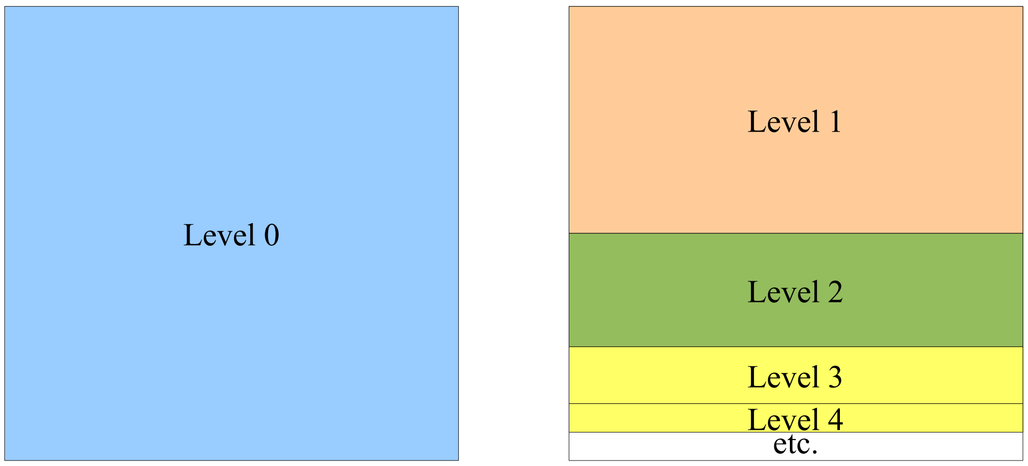


Figure 170: Super Bump Mapping memory organisation

Level 0 is accessed as per a standard 2D texture, the other levels require the calculation of co-ordinate offsets which allow direct access to individual vectors. Another issue which needed to be worked around was the problem of squeezing the 5 required values of x, y, z, weight and level of detail into the four component texels of an OpenGL texture. This was done by calculating z on the fly, in the shader. It is possible to calculate z at run time because the vectors are all normalised in the pre-processing stage, so the vectors' z component can be calculated by:

$$v_z = \sqrt{(1 - v_x^2 - v_y^2)}$$

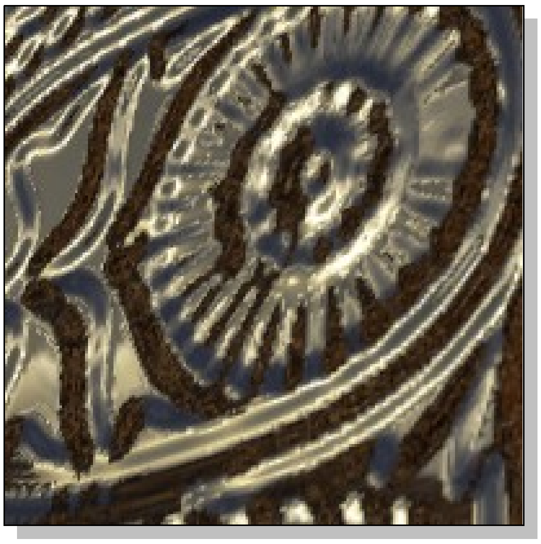
The reflection vector is calculated and the reflected pixel is fetched from the environment map once per vector. This is done using the pre-computed weight value to scale each contribution appropriately based on the spread of the vectors. The final stage is to take the final surface colour and bilinearly or trilinearly interpolate the results as required. Trilinear interpolation is preferred in the tests, in order to remove the sharp transitions between MIP map levels which are often associated with non-trilinearly interpolated scenes. This also has the effect of allowing the estimation of realistic performance values for the SBM process when combined with other common techniques.

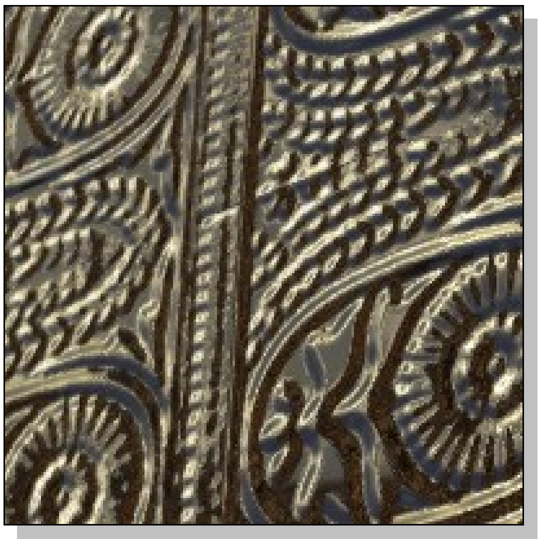
5.4 Results

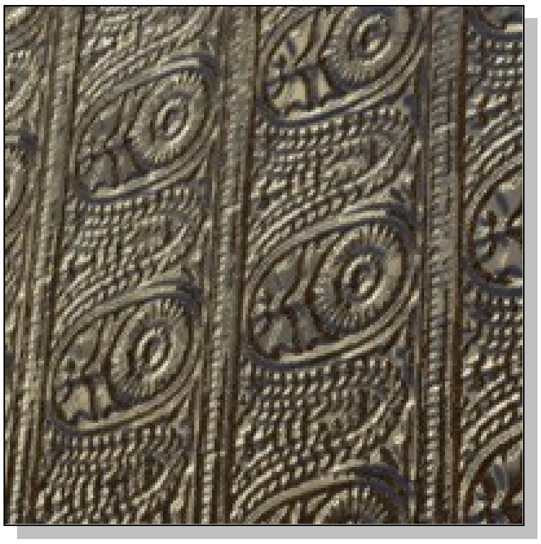
The analysis of the SBM technique is approached from two perspectives. Firstly, the final visual appearance of the technique should provide superior results to standard MIP mapped normal mapping. This is approximated here using the single vector version of SBM, which is simply standard normal mapping with the addition of the spread calculation. The SBM technique is therefore compared to both an improved version of standard normal mapping and the original Cant and Langensiepen technique [1]. Secondly, the technique shall be analysed in terms of performance in comparison to single vector techniques.

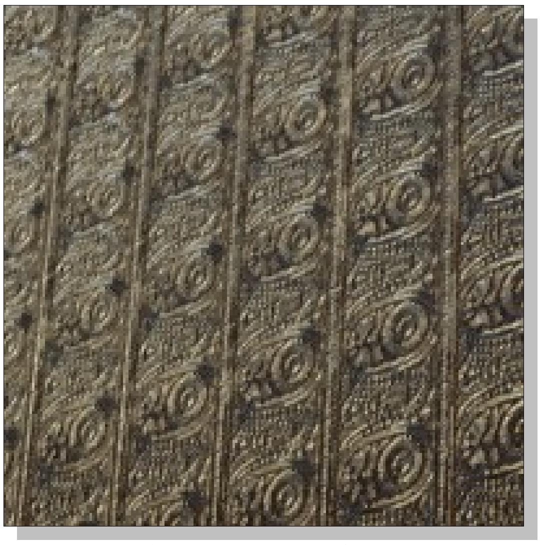
5.4.1 Visual Results

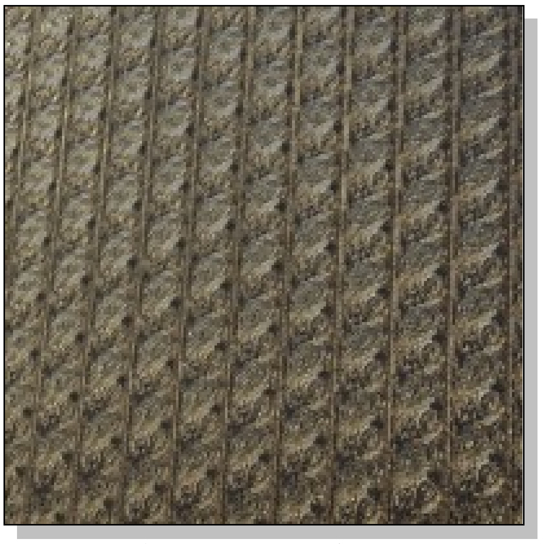
It can be seen from Figure 171 and Figure 172 that the hardware version of the technique compares well visually to both Cant and Langensiepen's technique, as well as the other techniques they focus on in their analysis of the “Efficient anti-aliased bump mapping” technique. Figure 171 a) and e) provide the best images for comparison with Figure 172 as these are test run at the same scales as those presented by Cant and Langensiepen [1]. The hardware based images are presented in an environment mapped scene reflecting a mountain range environment map. This is to proves it is possible to


a) Vectors = 2, Scale = 1


b) Vectors = 2, Scale = 2


c) Vectors = 2, Scale = 4


c) Vectors = 2, Scale = 8


e) Vectors = 2, Scale = 16

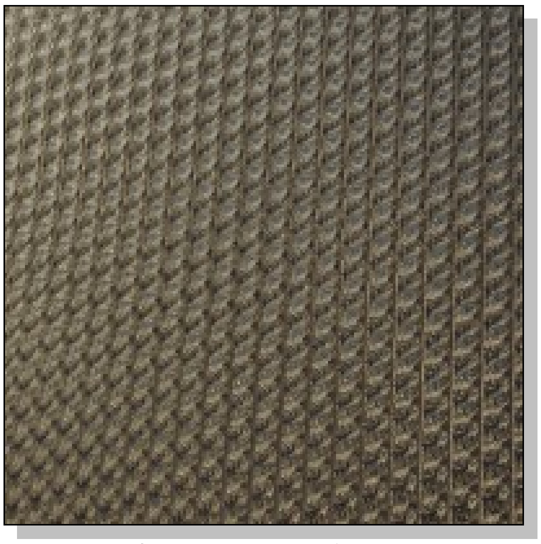

f) Vectors = 2, Scale = 32

Figure 171: Flower normal map

integrate the Super Bump Mapping with existing techniques and also provides a good example of typical uses for such techniques.

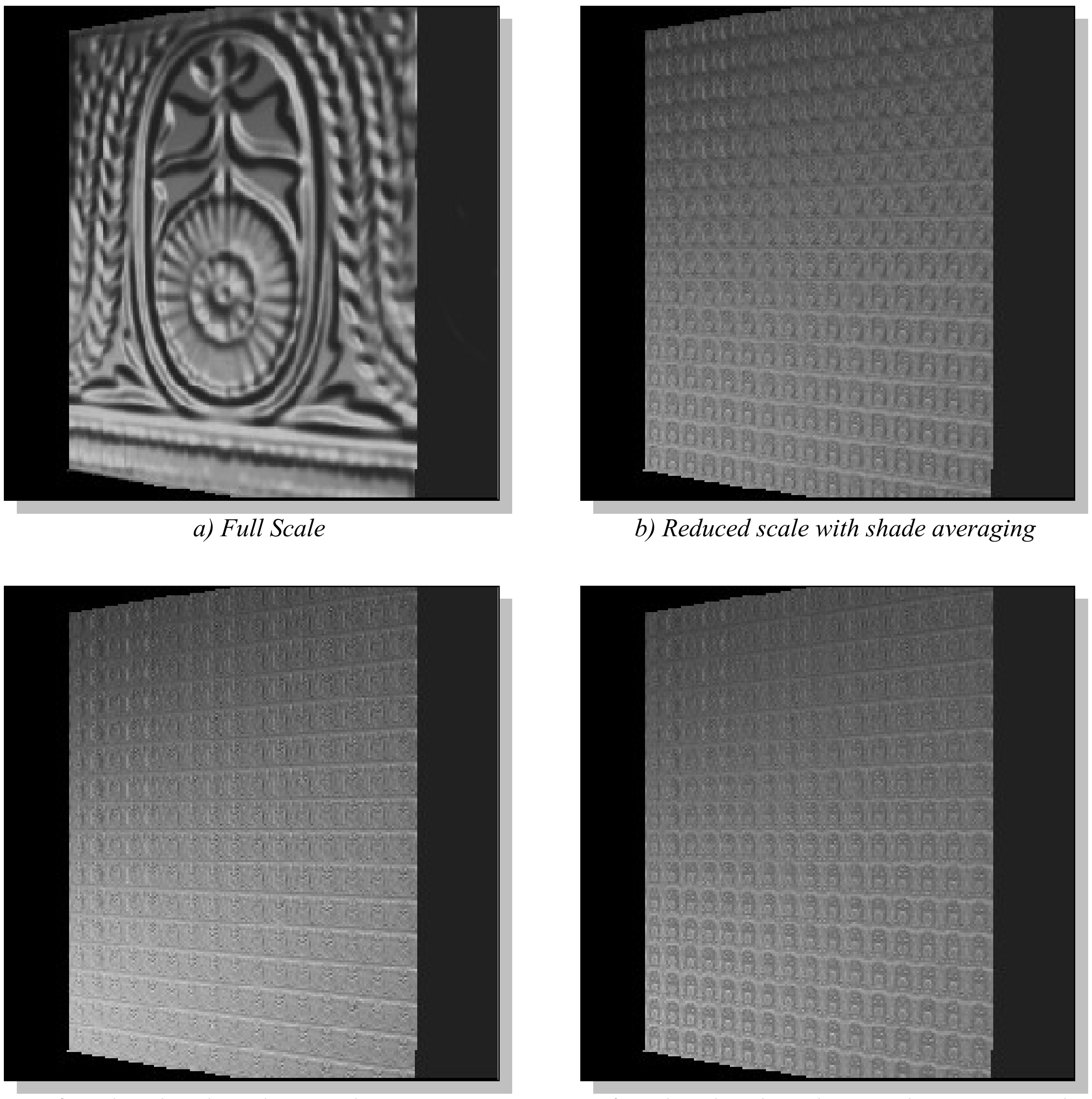

a) Full Scale
b) Reduced scale with shade averaging
f) Reduced scale with Normal averaging
f) Reduced scale with Normal averaging and variable spread

Figure 172: Cant & Langensiepen Flower normal map [1]

Figure 173 shows the full scale version of the image for comparative purposes, this shows clearly what the pattern should look like in the environment mapped scene.

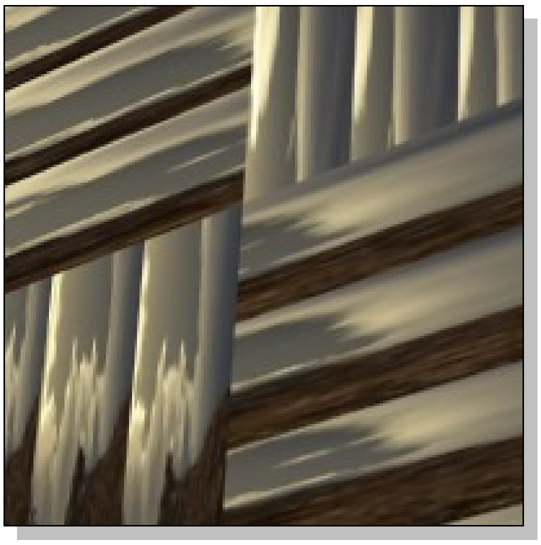
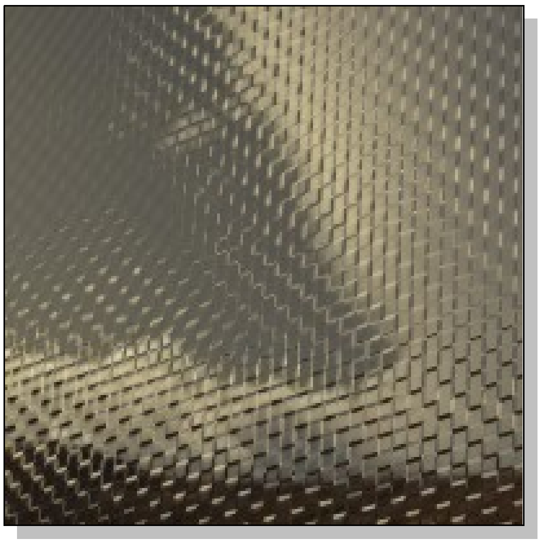
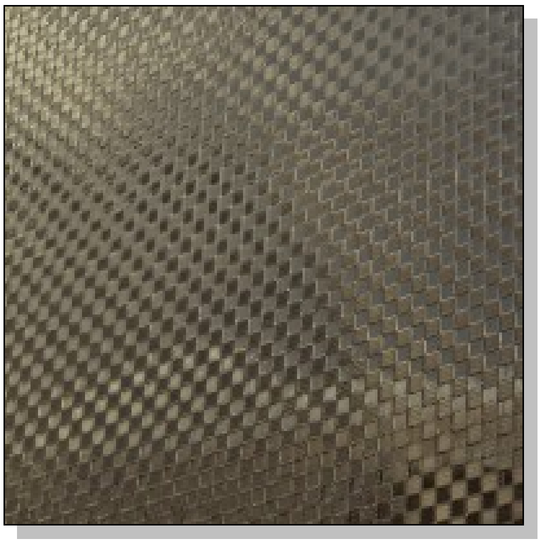


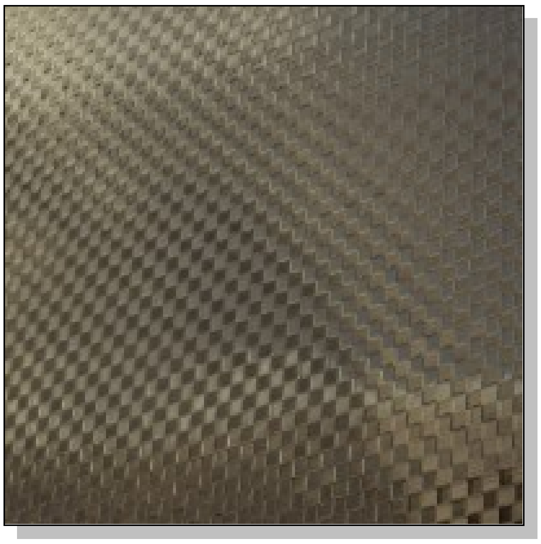
Figure 173: Grid normal map, full scale

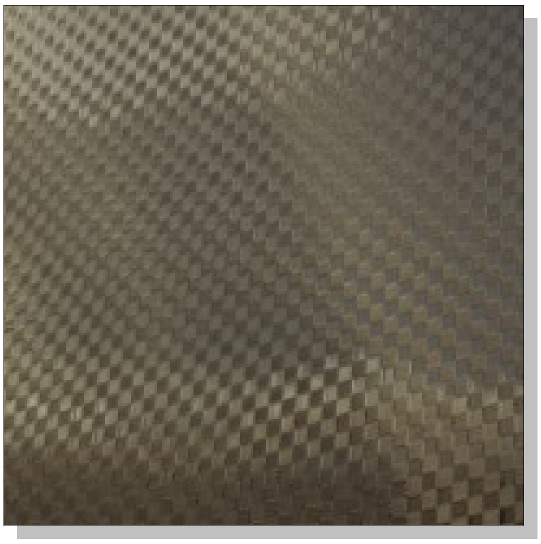
Figure 174 shows the same grid like pattern at a much more difficult 32x scale, Figure 174 a) – f) highlight the difference in final image detail caused by extra vectors. Figure 174 a) illustrates the expected overly glossy image provided by only a single vector system. This image also shows a lack of detail causing the individual grids to appear larger in the vertical direction than they are in reality.

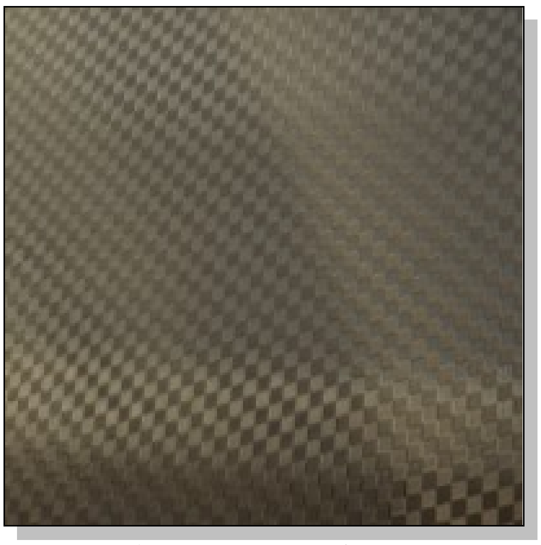
Figure 174 a) also demonstrates that too much of the environment map detail remains clearly visible, given the nature of the original normal map and the scale viewed at this should not be the case. At the opposite end of the scale Figure 174 f) provides the best representation of the original by providing the greatest level of detail, to the point where much of the internal grid detail is still visible. However for this particular pattern the differences between 16 and 32 vectors will be to a large extent negligible in real world applications such as games, where the detail would not be scrutinised to such a high degree as it is here.


a) Vectors = 1, Scale = 32


b) Vectors = 2, Scale = 32


c) Vectors = 4, Scale = 32


c) Vectors = 8, Scale = 32


e) Vectors = 16, Scale = 32

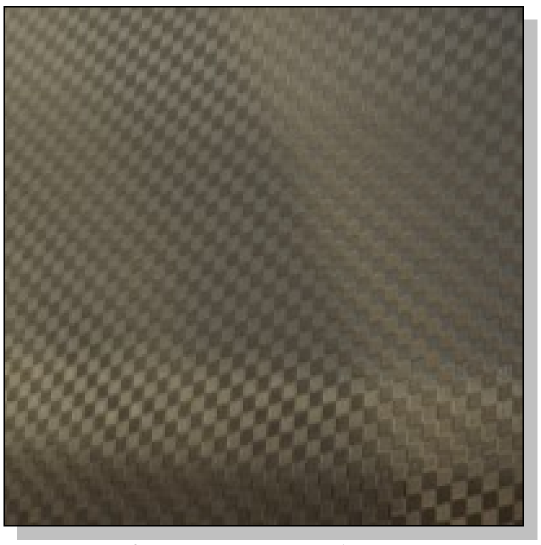

f) Vectors = 32, Scale = 32

Figure 174: Grid normal map reduced scale

Figure 175 shows a full scale reeded normal map, which is examined at a much more severe scale in Figure 176. With this pattern it is expected that severe aliasing will be observed at higher scales. It is for this reason the pattern was chosen, to highlight the differences the SBM technique can make even under extreme conditions.

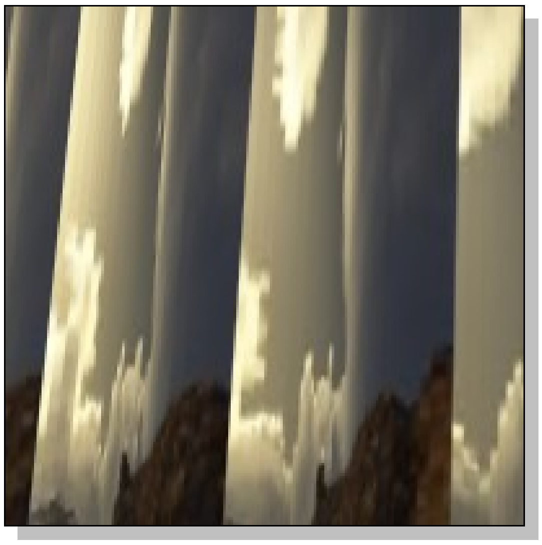


Figure 175: Reeded normal map, full scale

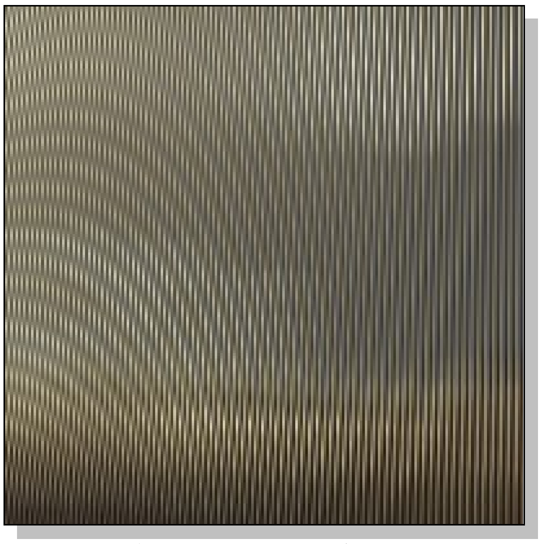
Figure 176 a) shows the expected over glossiness and lack of normal map detail retention witnessed in previous examples. However, it also displays a great deal of aliasing caused by the nature of the pattern. While this aliasing appears through all the test images shown here, the increase in the number of vectors has a noticeable effect on the visibility of the aliasing. As can be seen by examining Figure 176 a) through f), the aliasing gradually fades with each increase in the number of vectors.

a) Vectors = 1, Scale = 32

b) Vectors = 2, Scale = 32

c) Vectors = 4, Scale = 32

c) Vectors = 8, Scale = 32


e) Vectors = 16, Scale = 32

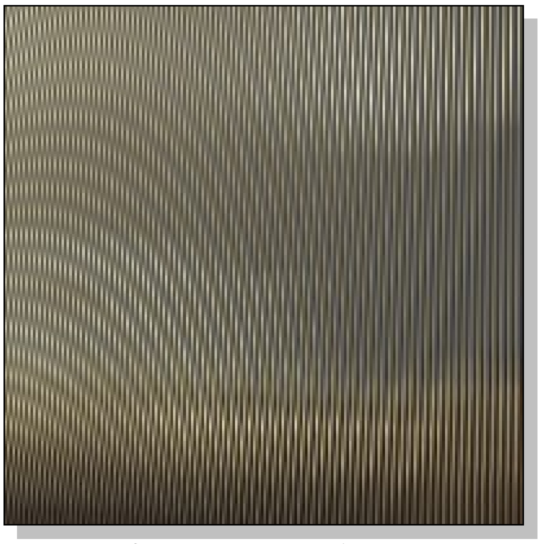

f) Vectors = 32, Scale = 32

Figure 176: Grid normal map reduced scale

5.4.2 Performance

The performance of this technique is examined on the NVIDIA 8600 platform. Figure 177 and Appendix G show how scale effects performance when the number of vectors is kept constant. It is apparent from these graphs that the performance (measured in frames per second[55]) falls as scale increases, the number of frames per second only slip below the target figure of 30fps only at fairly low scales.

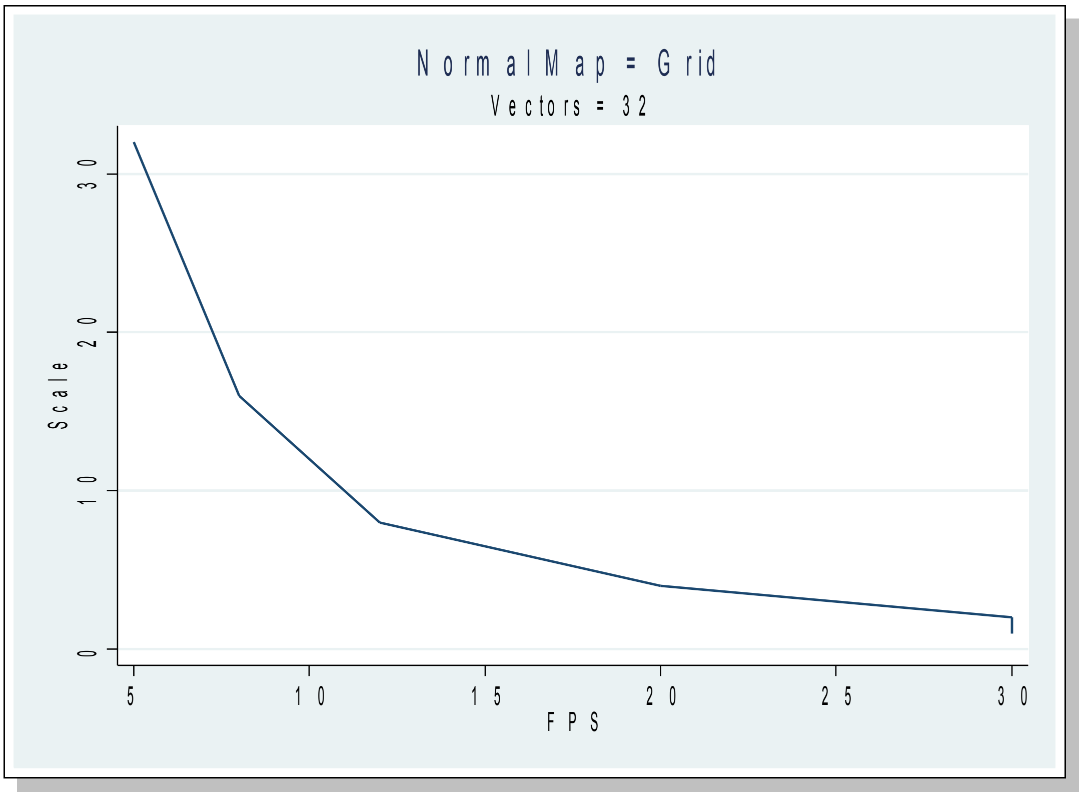


Figure 177: How scale effect FPS

However, it should be noted that the technique is doing twice as much work as would normally be required. This is due to the necessity of performing multiple texture fetches

55 FPS.

and calculating the z normal values on the fly, in order to fit within current hardware restrictions. It should also be noted that this technique is run in conjunction with trilinear filtering, which more than doubles the required workload at each pixel.

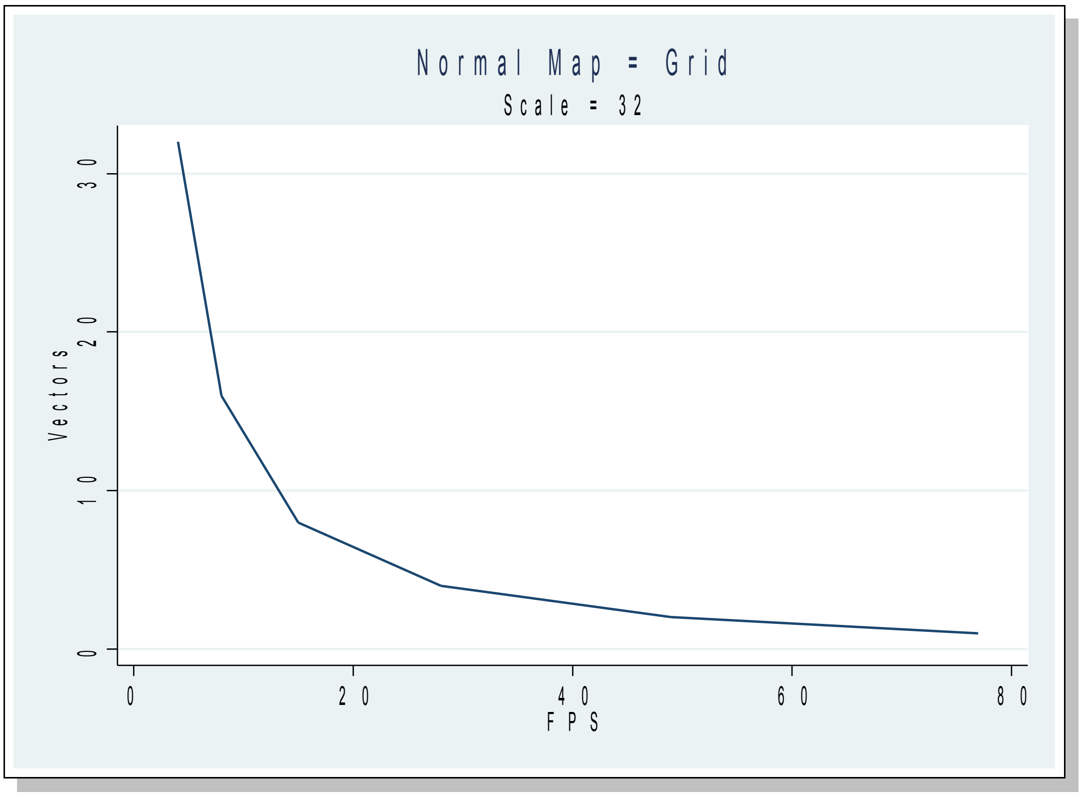


Figure 178: How the number of vectors effect FPS

Figure 178 and Appendix G demonstrate the effect that varying the number of vectors has on the techniques performance. Varying the number of vectors has a similar, but greater, effect on the performance of the system compared to the scale variations. As can be seen, standard single vector normal mapping achieves around 80 frames per second, and each additional vector reduces the frame rate further.

5.5 Conclusions

It has been shown that Cant and Langensiepen's technique can be implemented on current consumer level hardware[56] and that the results are favourable compared to their software version presented in [1].

The technique displays performance of up-to 80 frames per second, this could easily be improved up with some simple optimisations and better integration with the hardware. Some fairly basic adjustments such as being able to include an extra component per texel, to remove the need to calculate z on the fly, would yield notable performance improvements. While it was deemed necessary to include trilinear filtering in tests for reasons of image quality and comparisons to Cant and Langensiepen's images, this is not a necessity and would also provide performance increases if omitted.

Visually, the technique as implemented here performs well when compared to the single vector technique and provides comparable visual results to Cant and Langensiepen's original. The technique is also a prime candidate for combination with other techniques such as Parallax mapping, as the strengths of both techniques are in different, but complimentary, areas.

56 An NVIDIA 8600 in testing

Chapter 6: Conclusions and Future Work

All of the techniques implemented show a great deal of potential for consumer level hardware, although some are nearer to plausible real-time use than others.

The depth of field technique presented shows that current consumer level hardware, while still advancing, is not quite ready for such a complex effect. Although the software version presented in section 3.4.1 proves the theoretical capabilities of the algorithm, the current capabilities of consumer level graphics hardware precludes a full hardware accelerated implementation.

With the aim of implementing a version of the depth of field algorithm, which supports the see-through effect, and investigating the utilisation of modern hardware in an attempt to get it to operate in real-time it can only be concluded that there has been limited success. While each individual component of the system meets the 30fps performance criteria set, a combination of them to create the full depth of field solution is not currently possible while maintaining real-time performance.

The software version presented in section 3.4.1 shows that a convincing effect is created by use of this method and demonstrates the advantages of the see through effect. While the frame rate is extremely low, it does prove that the algorithm provides a plausible effect and hence that a full hardware implementation should be beneficial in providing greater realism to games and simulations in the future.

The scope for future work in the field is quite large, as much is currently not possible in real time on existing hardware. Future hardware developments will undoubtedly lead to a solution similar to the one proposed being possible in real time.

This is particularly true now. Since the completion of this work Depth of Field has become an extremely popular area of research. Papers such as: Practical post-process Depth of Field [53], Interactive Simulation of the Human Eye Depth of Field and Its Corrections by Spectacle Lenses [54] and Depth-of-Field Rendering by Pyramidal Image Processing [55] all propose real-time solutions to the various issues covered in section 3.2. However, none of these solve all the problems with Depth of Field simulation and all have significant limitations.

The Fourier texture filtering technique presented provides an interesting alternative to currently popular techniques, such as NVIDIA's anisotropic filtering. The technique proves that a lower resolution source image does not necessarily mean lower quality output. When coupled with the sizeable performance and memory footprint improvements, which are possible due to the techniques ability to allow reduced texture sizes, the Fourier technique becomes an attractive proposition for certain texture types. Possible future work includes further optimisation of the technique for consumer level hardware. This includes the addition to current API's and hardware, of the ability to customise texture fetch routines. The technique would also benefit from investigations into selective use of Fourier texture filtering. By combining the technique with other existing texture filtering methods, and intelligently selecting the appropriate technique based on texture content, it should be possible to maximise the benefits of the technique while using the strengths of other techniques to maximise quality in other areas.

As shown in section 4.4.3, the Fourier techniques show between 1% and 15% of the performance of the NVIDIA anisotropic filtering technique when tested on the 6800 platform. While this may appear like a large difference, it must be remembered that the NVIDIA technique is fully hardware optimised. As such, the sample counts and texture memory bandwidth usage analysed in section 4.4.3 provide a much fairer comparison. The later tests carried out on the 8600 platform show a drastic decrease in this gap. It can therefore be said that, should this trend continue, hardware is progressing at a pace that will allow these techniques to run at comparable speeds within a few generations. It should also be noted that both the Fourier implementations in the tests are simply a proof of concept and have not been optimised and refined over a period of years in the same way that the NVIDIA technique has.

With subtle changes to existing API's and some work on code optimisation, the performance of both EWA and the Fourier techniques could be improved significantly. This is a particularly important point because, as illustrated by section 4.4.3, the NVIDIA anisotropic technique has already peaked in terms of optimisation. This indicates that any future performance improvements for the NVIDIA technique shall be linked almost totally to hardware progression or compromises in quality.

When comparing performance between these techniques it must also be noted that the NVIDIA anisotropic technique is known to use optimised versions of trilinear filtering (see section 4.2.4), whereas the other trilinear based techniques examined use full trilinear filtering (4.2.3). It can be deduced therefore that some of the increased sharpness demonstrated by the NVIDIA technique is provided by this optimisation, along with a little extra performance. The presence of these optimisations also explains the visible layer transitions on some textures as well as some of the aliasing issues suffered by the NVIDIA technique, as shown in section 4.4.3.

The steadily increasing performance of shaders generation on generation should make optimised versions of the Fourier techniques, as discussed in section 4.4.1, viable for real-time use within the next few hardware generations. EWA in real-time is likely to be only a few generations behind. However, neither techniques are currently suitable for real-time without significant optimisations to the shader code and minor alterations to existing API's.

As the test results indicate, each texture filtering technique examined has advantages and drawbacks. The main advantage of the Fourier technique being the ability to use significantly lower texture resolutions. The Fourier techniques could therefore be used as a means of reducing overall texture size within an application, and hence significantly reduce texture bandwidth requirements, by providing smaller textures with more detail than would otherwise be possible. Because the Fourier techniques achieve similar quality results at low resolutions to higher resolution source textures with other techniques, without introducing aliasing. By using a suitable number of frequencies, the Fourier techniques can match anisotropic filtering for quality. Perhaps the most important observation is that the Fourier techniques do this with no recorded aliasing under test conditions, on all but one texture. This is the stripes texture, which did cause issues with all of the techniques examined.

The reduction in texture sizes possible with the Fourier technique would be advantageous for systems where memory sizes are limited, where low level caching is available, or where memory footprint is an issue. The technique may also be used only on lower MIP map levels where it is most effective, while other techniques better suited to higher resolutions could be used for the larger levels.

Other important observations made are the differences between the bilinear and trilinear versions of the Fourier technique as demonstrated by the tests in section 4.4.3. The addition of trilinear filtering removes a slight degree of sharpness in all cases. However, the bilinear version does not suffer from the usual visible MIP map level transitions. These have been suppressed by the technique in the same manner as the aliasing. It can be concluded therefore that the addition of the Fourier technique to a scene removes the need to perform trilinear filtering to hide level of detail transitions. This will provide sharper results and save many samples per cycle.

The current hardware imposed limitations, particularly on the number of frequencies, indicates that the development of hybrid methods may be preferable in the short term. These hybrid methods could intelligently choose the appropriate filtering method based on texture content and angle.

While EWA represents the ultimate eventual aim for anybody looking to minimise aliasing and maximise texture quality, the Fourier techniques presented here when used in conjunction with existing techniques could prove an effective real-time solution in the meantime. Given the kinds of optimisations already in use within these techniques a hybrid method which combines the sharpness of the NVIDIA techniques images with the lack of aliasing shown by the Fourier technique could provide a noticeable improvement over current techniques for performance, memory footprint and image quality. However, one drawback of the Fourier techniques is the large amount of pre-processing required to generate the Fourier sets. This is not deemed to be a major failing of the technique as look-up tables and similar techniques, which also require heavy pre-processing of data, are commonly used and this type of work is easily done in batch.

The Super Bump Mapping technique demonstrates the advantages of multi-vector bump mapping, both for detail and anti-aliasing properties. It has been shown, in section 5.4, that Cant and Langensiepen's technique can be implemented on current consumer level hardware and that the results are favourable compared to their software version presented in [1]. While the technique performs well currently, there exists a great deal more potential for the technique if properly integrated with consumer level hardware.

For full hardware integration this technique requires either adjustments to the number of components allowed per-pixel or the addition of programmable texture fetch routines to API's and hardware in future generations.

The technique displays performance of up-to 80 frames per second. As discussed in section 5.4, this could easily be improved up with some simple optimisations and better integration with the hardware. Some simple adjustments, such as being able to include an extra component per texel and the ability to remove the need to calculate z on the fly, could yield notable performance improvements. While it was deemed necessary to include trilinear filtering in tests for reasons of image quality comparisons to Cant and Langensiepen's images, this is not a necessity and would also provide performance increases if omitted.

Visually, the technique, as shown in section 5.4, performs well when compared to single vector techniques and provides comparable visual results to Cant and Langensiepen's original. The technique is also a prime candidate for combination with other techniques such as Parallax mapping, as the strengths of both techniques are in different, but complimentary, areas.

References

[1] Cant R & Langensiepen C, 2003. **Efficient anti-aliased bump mapping**. Computers & Graphics, Volume 30, Issue 4. pp. 561-580

[2] Rhodes D, Cant R & Al-Dabass D, 2004. **Depth Of field algorithms for more realistic simulations**. Proceedings of UKSIM2004, the UK Simulation Society, St Catherine's College, Oxford, 29 - 31 March 2004. pp. 162-168

[3] Rhodes D, Cant R & Al-Dabass D, 2003. **A new depth of field algorithm with applications to games**. Proceedings of Game-on 2003, 4th International Conference on Intelligent Games and Simulation, IEE Savoy Place, London, November 19-21, 2003. pp. 157-161

[4] Rhodes D, Cant R & Al-Dabass D, 2003. **Current depth of field algorithms & techniques for games**. Proceedings of Game-on 2003, 4th International Conference on Intelligent Games and Simulation, IEE Savoy Place, London, November 19-21, 2003. pp. 19-21

[5] Galistel A, 2008. **GPU sales figures for Q4 2007 [online]**. NordicHardware. Available at: [URL:http://www.nordichardware.com/news,7290.html]

[6] BBC News, 2008. **US video games sales hit record [online]**. BBC. Available at:

[URL:http://news.bbc.co.uk/1/hi/business/7195511.stm]

[7] Intel ®, 2006. **Moore's Law [online]**. Intel ®. Available at: [URL:http://www.intel.com/technology/mooreslaw/index.htm]

[8] Dempski K, 2002. **Introduction to Textures**. Real-time rendering tips and techniques in DirectX, 1st ed. Premier Press. ISBN: 1931841276/978-1931841276. pp. 187-212

[9] Gouraud H, 1973. **Continuous shading of curved surfaces**. IEEE Transactions on Computers, Volume C-20, Issue 6. pp. 623-628

[10]Kilgard M, 1999. **Hardware Accelerated Anisotropic Lighting [online]**. NVIDIA Coporation, Presented at GDC '99. Available at: [URL:http://developer.nvidia.com/attach/6820]

[11]Dempski K, 2002. **Cartoon Shading**. Real-time rendering tips and techniques in DirectX, 1st ed. Premier Press. ISBN: 1931841276/978-1931841276. pp. 465-477

[12]Rhodes D, Cant R, Langensiepen C & Al-Dabass D, 2004. **Programmable GPUs and shading languages: past, present and future**. Proceedings of CGAIDE2004, 5th Game-on International Conference, Microsoft HQ, Reading, UK. pp. 66-70

[13]NVIDIA®, 2004. **Programming Graphics Hardware [online]**. NVIDIA®

Corporation. Available at: [URL:ftp://download.nvidia.com/developer/presentations/2004/Eurographics/EG_04_IntroductionToGPU.pdf]

[14]LaMothe A, 2003. **Polygon Rasterization Review**. Tricks of the 3D Game Programming Gurus, 1st ed. Sams Publishing. ISBN: 0672318350. pp. 928-948

[15]Dempski K, 2002. **Fixed Function Lighting**. Real-time rendering tips and techniques in DirectX, 1st ed. Premier Press. ISBN: 1931841276/978-1931841276. pp. 161-185

[16]Fernando R, 2004. **GPU Gems**. GPU Gems. NVIDIA. ISBN: 0321228324.

[17]Microsoft®, 2003. **Architectural Overview for Direct3D [online]**. Microsoft® Corpiration. Available at: [URL:http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/programmingguide/gettingstarted/architecture/overview.asp]

[18]Microsoft®, 2004. **Shader Model 3 [online]**. Microsoft® Corporation. Available at: [URL:http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/ProgrammingGuide/ProgrammablePipeline/HLSL/ShaderModel3/ShaderModel3.asp]

[19]NVIDIA®, 2001. **The Infinite Effects GPUs [online]**. NVIDIA® Corporation.

Available at:

[URL:http://www.nvidia.co.uk/docs/lo/1050/SUPP/gf3ti_overview.pdf]

[20]Mark W, Glanville R, Akeley K & Kilgard M, 2003. **Cg: a system for programming graphics hardware in a C-like language**. ACM SIGGRAPH '03 Papers. pp. 896-907

[21]HEXUS, 2002. **Cg [online]**. HEXUS.NET. Available at: [URL:http://www.hexus.net/content/reviews/review.php?dXJsX3Jldmlld19JRD0zNzkmdXJsX3BhZ2U9M]

[22]NVIDIA®, 2004. **Shader Model 3 Unleashed [online]**. NVIDIA® Corporation. Available at: [URL:ftp://download.nvidia.com/developer/presentations/2004/SIGGRAPH/Shader_Model_3_Unleashed.pdf]

[23]Case L, 2002. **Making the Right Graphic Choice [online]**. ExtremeTech. Available at: [URL:http://www.extremetech.com/article2/0,1558,727608,00.asp]

[24]Kessenich J, Baldwin D & Rost R, 2006. **The OpenGL® Shading Language [online]**. SGI. Available at: [URL:http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf]

[25]NVIDIA®, 2002. **Getting Started with Cg [online]**. NVIDIA® Corporation.

Available at: [URL:http://developer.nvidia.com/docs/IO/3974/ATT/Getting Started.pdf]

[26]ATI, 2004. **D3D Tutorial Optimisations [online]**. ATI. Available at: [URL:http://www.ati.com/developer/gdc/D3DTutorial_Optimisations.pdf]

[27]NVIDIA®, 2004. **FX Composer [online]**. NVIDIA® Corporation. Available at: [URL:http://developer.nvidia.com/object/fx_composer_home.html]

[28]Penfold D, 2002. **HLSL's, Cg and the RenderMonkey [online]**. Tom's Hardware Guide. Available at: [URL:http://graphics.tomshardware.com/graphic/20021004/]

[29]Rhodes D, Cant R, Langensiepen C & Al-Dabass D, 2006. **Teaching Shader Programming for Games**. Proceedings of ICCMS/UKSim 2006, 9th Int. Conference on Computer Modeling & Simulation, Oriel College, Oxford, 4 - 6 April 2006. pp. 43-47

[30]LaMothe A, 2003. **Floating-Point Unit Math Primer** . Tricks of the 3D Game Programming Gurus, 1st ed. Sams Publishing. ISBN: 0672318350. pp. 468-488

[31]Abramowitz M & Stegun A, 1970. **Circular Sines and Cosines for Radain Arguments**. Handbook of Mathematical Functions. National Bureau of Standards. ISBN: 0486612724/978-0486612720. pp. 142-173

[32]Rokita P, 1996. **Generating Depth-of-Field Effects in Virtual Reality Applications**. IEEE Computer Graphics and Applications, Volume 16, Issue 2. pp. 18-21

[33]Maltby R, 2003. **The Optics of Expressive Space**. Hollywood Cinema, 2nd ed. WileyBlackwell. ISBN: 0631216154/978-0631216155. pp. 319-326

[34]NVIDIA®, 2003. **Depth of Field [online]** . NVIDIA® Corporation. Available at: [URL:http://developer.nvidia.com/view.asp?IO=depth_field]

[35]ATI®, 2003. **Depth of Field [online]**. Advanced Micro Devices, Inc. Available at: [URL:http://www.ati.com/developer/samples/dx9/DepthOfField.html]

[36]Potmesil M & Chakravarty I, 1981. **A Lens and Aperture Camera Model for Synthetic Image Generation**. Proceedings of SIGGRAPH '81. pp. 297-305

[37]Microsoft®, 2003. **Depth of Field Sample [online]**. Microsoft® Corporation. Available at: [URL:http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/programmingguide/tutorialsandsamplesandtoolsandtips/samples/depthoffield.asp]

[38]Serendip, 2003. **Seeing more than your eye does [online]**. Bryn Mawr College. Available at: [URL:http://serendip.brynmawr.edu/bb/blindspot1.html]

[39]Bungie, 2008. **Halo 3 [online]**. Bungie.net. Available at: [URL:http://www.bungie.net/]

[40]Bramwell T, 2007. **Halo 3 lacks depth (of field) [online]**. Eurogamer Network Ltd. Available at: [URL:http://www.eurogamer.net/article.php?article_id=73994]

[41]Demers J, 2004. **Depth of Field: A Survey of Techniques**. GPU Gems. NVIDIA. ISBN: 0321228324. pp. 375-390

[42]Cannon K, 2002. **Pixar Announces Ray Tracing and Global Illumination in RenderMan® Release 11 [online]**. Pixar. Available at: [URL:https://renderman.pixar.com/products/news/prman11.html]

[43]Christensen P, Fong J, Laur D & Batali D, 2006. **Ray Tracing for the Movie 'Cars'**. Proceedings of the IEEE Symposium on Interactive Ray Tracing. pp. 1-6

[44]Snyder J & Lengyel J, 1998. **Visibility Sorting and Compositing without Splitting for Image Layer Decompositions**. Proceedings of SIGGRAPH '98. pp. 219-230

[45]Chia N, Cant R & Al-Dabass D, 2001. **New Anti-Aliasing and Depth of Field Techniques for Games Graphics**. Proceedings of Game-on 2001, 2nd International Conference on Intelligent Games and Simulation. p. 115

[46]Schilling A & Staßer W, 1993. **EXACT: Algorithm and hardware architecture for an improved A-buffer**. Proceedings of SIGGRAPH '93. pp. 85-91

[47]Melles Griot Inc, 2003. **Circle of Confusion [online]**. Melles Griot Inc. Available at: [URL:http://www.mellesgriot.com/glossary/wordlist/glossarydetails.asp?wID=132]

[48]Smith S, 1997. **Linear Image Processing**. The Scientist and Engineer's Guide to Digital Signal Processing, 1st ed. California Technical Publishing. ISBN: 0966017633/978-0966017632. pp. 397-422

[49]NVIDIA®, 2003. **Interactive Order-Independent Transparency [online]**. NVIDIA® Corporation. Available at: [URL:http://www.nvidia.co.uk/docs/IO/1316/ATT/order_independent_transparency.pdf]

[50]Watt A, 2000. **3D Computer Graphics**. Pearson Education Limited. ISBN: 0201398559.

[51]Williams L, 1983. **Pyramidal Parametrics**. ACM SIGGRAPH Computer Graphics, Volume 17, Issue 3. pp. 1-11

[52]LaMothe A, 2003. **Bilinear Texture Filtering**. Tricks of the 3D Game Programming Gurus, 1st ed. Sams Publishing. ISBN: 0672318350. pp. 1250-1256

[53]Hammon E, 2003. **Practical Post-Process Depth of Field**. GPU Gems 3, 1st ed. Addison-Wesley. ISBN: 0321515269/978-0321515261. pp. 583-605

[54]Kakimoto M, Tatsukawa T, Mukai Y & Nishita T, 2007. **Interactive Simulation of the Human Eye Depth of Field and Its Corrections by Spectacle Lenses**. EUROGRAPHICS, 2007. Volume 26, Number 3. pp. 627-636

[55]Krause M & Strengert M, 2007. **Depth-of-Field Rendering by Pyramidal Image Processing**. EUROGRAPHICS, 2007. Volume 26, Number 3. pp. 645-654

[56]Sainz M & Pajarola R, 2004. **Point-Based Rendering Techniques**. Computers and Graphics, Volume 28, Issue 6. pp. 869-879

[57]Wikipedia, 2007. **Texture Mapping [online]**. Wikipedia. Available at: [URL:http://en.wikipedia.org/wiki/Texture_mapping]

[58]LaMothe A, 2003. **Basic Sampling Theory**. Tricks of the 3D Game Programming Gurus, 1st ed. Sams Publishing. ISBN: 0672318350. pp. 976-979

[59]Hecker C, 1995-1996. **Perspective Texture Mapping**. Game Developer Magazine. April/May 1995-April/May 1996.

[60]LaMothe A, 2003. **Perspective-Correct Texturing and 1/z-Buffering**. Tricks of the 3D Game Programming Gurus, 1st ed. Sams Publishing. ISBN: 0672318350. pp.

1207-1250

[61]Beets K & Barron D, 2000. **Super-sampling Anti-aliasing Analyzed [online]**. Beyond3D. Available at: [URL:www.Beyond3D.com]

[62]Haeberli P & Akeley K, 1990. **The accumulation buffer: hardware support for high-quality rendering**. ACM SIGGRAPH Computer Graphics, Volume 24, Issue 4. pp. 309-318

[63]Nyquist H, 1928. **Certain Topics in Telegraph Transmission Theory**. Transactions of AIEE, vol 47. pp. 617-644. Reprinted: Proceedings of IEEE, 2002, Vol 90, No 2. pp. 280-305.

[64]Shannon C, 1949. **Communication in the Presence of Noise**. Proceedings of IRE, Vol 37, Issue 1. pp. 10-21. Reprinted: Proceedings of IEEE, 1998, Vol 86, No 2, pp. 447-457.

[65]LaMothe A, 2003. **Mip Mapping and Trilinear Texture Filtering**. Tricks of the 3D Game Programming Gurus, 1st ed. Sams Publishing. ISBN: 0672318350. pp. 1257-1258

[66]Cant R & Shrubsole P, 2000. **Texture Potential MIP Mapping**. ACM Transactions on Graphics, Volume 19, Issue 4. pp. 164-184

[67]Weinand L, 2004. **ATI's Optimized Texture Filtering Called into Question [online]**. Tom's Hardware. Available at: [URL:http://www.tomshardware.co.uk/ati,review-965.html]

[68]Greene H & Heckbert P, 1986. **Creating Raster Omnimax Images from Multiple Perspective Views Using the Elliptical Weighted Average Filter**. IEEE Computer Graphics and Applications, IEEE Computer Society Press, Vol 6. pp. 21-27

[69]Heckbert P, 1989. **Fundamentals of Texture Mapping and Image Warping**. University of California school of Computer Science.

[70]McCormack J, Perry R, Farkas K & Jouppi N, 1999. **FELINE: Fast Elliptical Lines for Anisotropic Texture Mapping**. SIGGRAPH 1999. pp. 243-250

[71]Shin H, Lee J & Kim L, 2001. **SPAF: Sub-texel Precision Anisotropic Filtering**. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware. pp. 99-108

[72]Heckbert P, 1986. **Survey of Texture Mapping**. IEEE Computer Graphics and Applications, Volume 6, Issue 11. pp. 56-67

[73]Wolberg G, 1990. **Digiatal Image Warping**. ISBN: 0818689447/978-0818689444. WileyBlackwell.

[74]Lansdale R, 1991. **Texture Mapping and Resampling for Computer Graphics**. Masters Thesis, Department of Electrical Engineering, University of Toronto.

[75]Schilling A, Knittel G & Strasser W, 1996. **Texram: A Smart Memory for Texturing**. IEEE Computer Graphics and Applications, IEEE Computer Society Press, Volume 16, Issue 3. pp. 32-41

[76]Norton A, Rockwood A & Skolmoski P, 1982. **Clamping: A method of antialiasing textured surfaces by bandwidth limiting in object space**. ACM SIGGRAPH Computer Graphics, Volume 16, Issue 3. pp. 1-8

[77]Weisstein E, 2007. **Fourier Transform [online]**. MathWorld - A Wolfram Web Resource. Available at: [URL:http://mathworld.wolfram.com/FourierTransform.html]

[78]Schacter B, 1980. **Long Crested Wave Models**. Computer Graphics and Image Processing, Vol 12. pp. 187-201

[79]Mitchell D & Netravali A, 1988. **Reconstruction Filters in Computer Graphics**. ACM SIGGRAPH Computer Graphics, Volume 22, Issue 4. pp. 221-228

[80]Microsoft®, 2003. **Understanding Frames Per Second (FPS) [online]**. Microsoft® Corporation. Available at: [URL:http://support.microsoft.com/kb/269068]

[81]Blinn JF, 1978. **Simulation of Wrinkled Surfaces**. ACM SIGGRAPH Computer Graphics, Volume 12, Issue 3. pp. 286-292

[82]Donnelly W, 2005. **Per-Pixel Displacement Mapping with Distance Functions**. GPU Gems 2, 1st ed. Addison-Wesley. ISBN: 0321335597/987-0321335593. pp. 123-136

[83]Gold M, 1999. **Emboss Bump Mapping [online]**. NVIDIA® Corporation. Available at: [URL:http://developer.nvidia.com/object/emboss_bump_mapping.html]

[84]ATI®, 2007. **Bump Mapping on Consumer 3D Graphics Accelerators [online]**. 2007 Advanced Micro Devices, Inc. Available at: [URL:http://ati.de/developer/sdk/RadeonSDK/Html/Tutorials/RadeonBumpMap.html]

[85]Schlag J, 1994. **Fast Embossing Effects on Raster Image Data**. Graphics Gems IV. Academic Press. ISBN: 0123361559/978-0123361554. pp. 433-437

[86]Cook R, 1984. **Shade Trees**. Proceedings of SIGGRAPH '84. pp. 223-231

[87]Dempski K, 2002. **Per-Pixel Lighting - Bump Mapping**. Real-time rendering tips and techniques in DirectX Edition: 1st ed Journal: Real-time rendering tips and techniques in DirectX, 1st ed. Premier Press. ISBN: 1931841276/978-1931841276. pp. 586-607

[88]Krishnamurthy V & Levoy M, 1996. **Fitting Smooth Surfaces to Dense Polygon Meshes**. Proceedings of SIGGRAPH '96. pp. 313-324

[89]Cohen J, Olano M & Manocha D, 1998. **Appearance-preserving simplification**. Proceedings of SIGGRAPH '98. pp. 115-122

[90]Cignoni P, Montani C, Rocchini C & Scopigno R, 1998. **A General Method for Preserving Attribute Values on Simplifed Meshes**. IEEE Visualization, Proceedings of the conference on Visualization '98. pp. 59-66

[91]Persson E, 2003. **Fragment level Phong illumination**. Shader X2. Wordware. ISBN: 155622902X/978-1556229022.

[92]Cloward B, 2007. **Normal Mapping Tutorial [online]**. bencloward.com. Available at: [URL:http://www.bencloward.com/tutorials_normal_maps2.shtml]

[93]ATI®, 2007. **Dynamic Bump Mapping on Radeon® [online]**. Advanced Micro Devices, Inc. Available at: [URL:http://ati.de/developer/sdk/RadeonSDK/Html/Samples/Direct3D/RadeonDynamicEMBM.html]

[94]Dempski K, 2002. **Reflection and Refraction**. Real-time rendering tips and techniques in DirectX, 1st ed. Premier Press. ISBN: 1931841276/978-1931841276. pp. 479-498

[95]Nuydens T, 2007. **An Overview of Bump Mapping Techniques [online]**. Delphi3D. Available at: [URL:http://www.delphi3d.net/articles/viewarticle.php?article=bumpmapping.htm]

[96]Schilling A, 2001. **Antialiasing of Environment Maps**. Computer Graphics Forum; 20(1). pp. 5-11

[97]Kaneko T, Takahei T, Inami M, Kawakami N, Yanagida Y, Maeda T & Tachi S, 2001. **Detailed Shape Representation with Parallax Mapping**. ICAT 2001. pp. 205-208

[98]Oliveira M, Bishop G & McAllister D, 2000. **Relief texture mapping**. Proceedings of SIGGRAPH 2000. pp. 359-368

[99]Tatarchuk N, 2005. **Practical Dynamic Parallax Occlusion Mapping**. ACM SIGGRAPH 2005 Sketches, Article No. 106.

[100]Welsh T, 2004. **Parallax Mapping with Offset Limiting: A PerPixel Approximation of Uneven Surfaces**. Technical Report: Infiscape Corporation.

[101]McGuire M & McGuire M, 2005. **Steep Parallax Mapping**. Iron Lore Entertainment, I3D Poster; Brown Technical Report.

[102]Tatarchuk N, 2006. **Dynamic Parallax Occlusion Mapping with Approximate**

Soft Shadows. Proceedings of the 2006 symposium on interactive entertainment 3D graphics and games. pp. 63-69

[103]Phong B, 1975. **Illumination for computer generated pictures**. Communications of the ACM, Volume 18 , Issue 6. pp. 311-317

[104]Becker B & Max N, 1993. **Smooth transitions between bump rendering algorithms**. Proceedings of SIGGRAPH '93. pp. 183-190

[105]Sloan P, Hall J, Hart J & Snyder J, 2004. **Clustered principle components for precomputed radiance transfer**. Proceedings of SIGGRAPH '04. pp. 382-391

[106]Fournier A, 1992. **Filtering Normal Maps and Creating Multiple Surfaces**. Department of Computer Science, University of British Columbia. Technical report TR92-41.

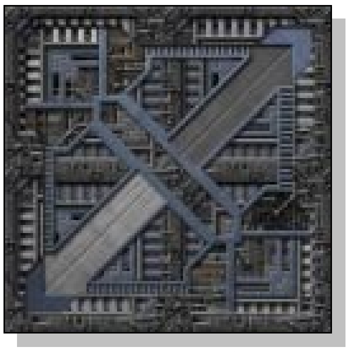
[107]Olano M & North M, 1997. **Normal Distribution Mapping**. UNC Computer Science Technical Report 97-041, Department of Computer Science, University of North Carolina.

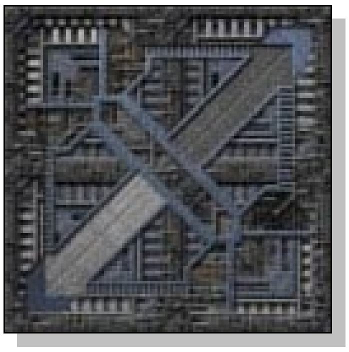
[108]Schilling A, 1997. **Towards real-time photorealistic rendering: challenges and solutions**. Proceedings of the 1997 SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware, Los Angeles, California. p. 7–15

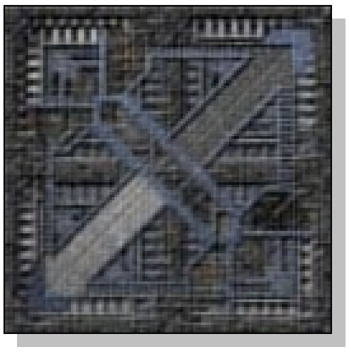
[109]Kohonen T, 1990. **The self-organizing map**. Proceedings of the IEEE, Volume 78, Issue 9. pp. 1464-1480

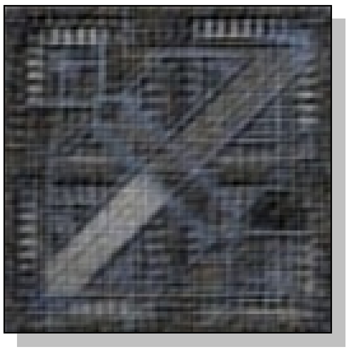
Appendices

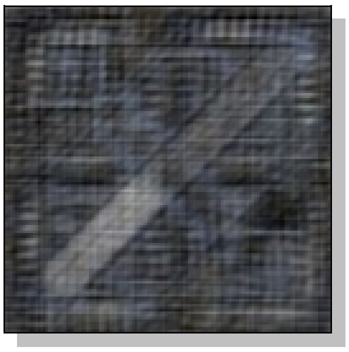
Appendix A: Software Frequency Tests

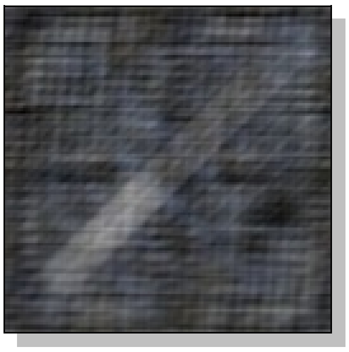

a) Original

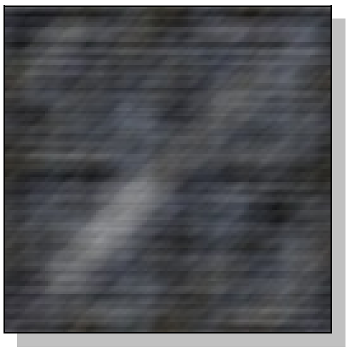

b) 5000 frequencies

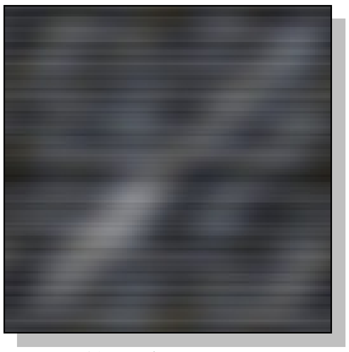

c) 2500 frequencies


d) 1000 frequencies


e) 500 frequencies


f) 250 frequencies


g) 100 frequencies


h) 50 frequencies

i) 25 frequencies

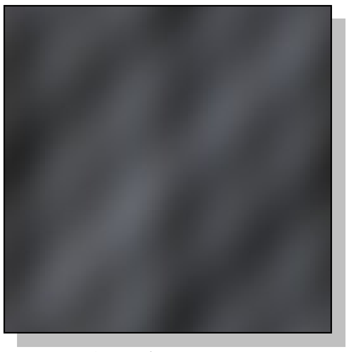
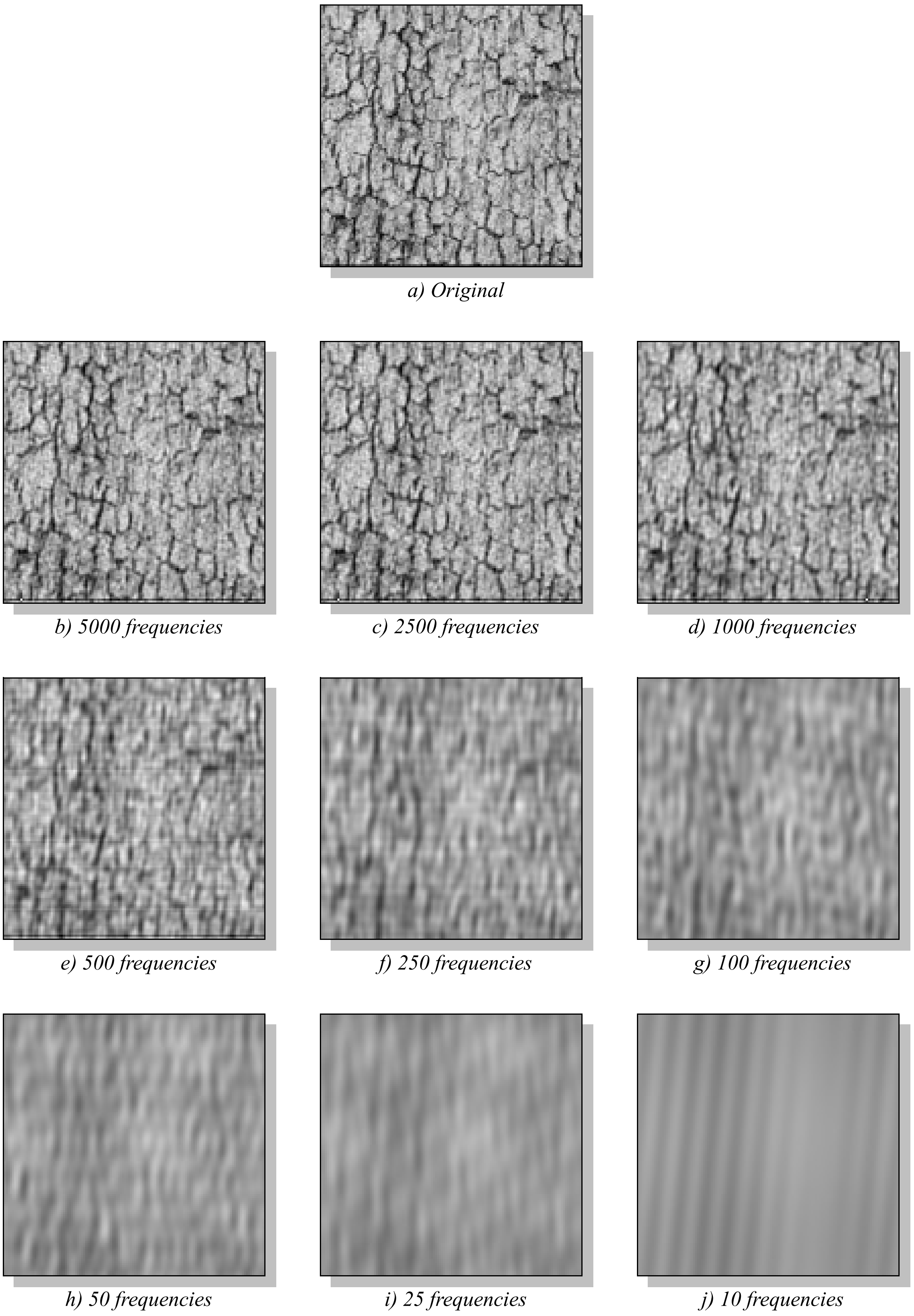

j) 10 frequencies

Figure A1: Software Frequency Tests: Alien Tile



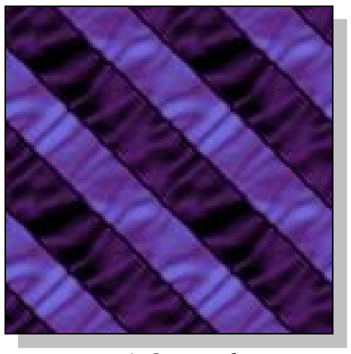
a) Original

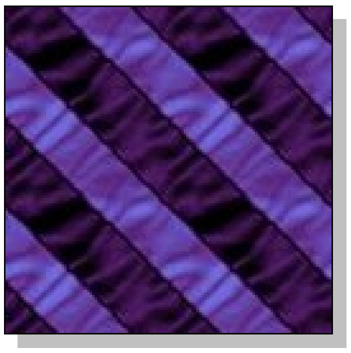
b) 5000 frequencies *c) 2500 frequencies* *d) 1000 frequencies*

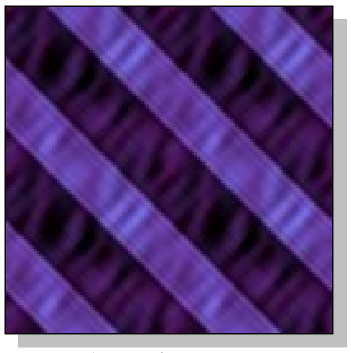
e) 500 frequencies *f) 250 frequencies* *g) 100 frequencies*

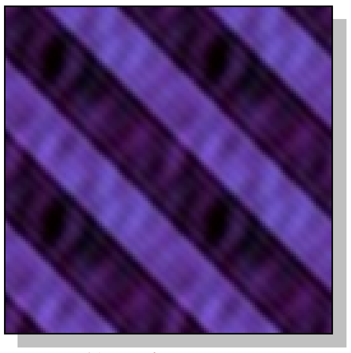
h) 50 frequencies *i) 25 frequencies* *j) 10 frequencies*

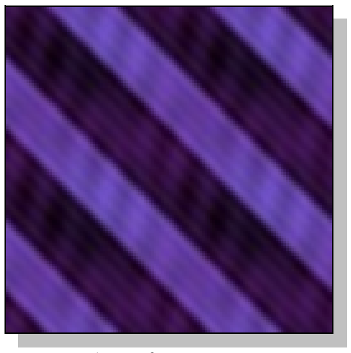
Figure A2: Software Frequency Tests: Bark


a) Original


b) 5000 frequencies


g) 100 frequencies


h) 50 frequencies


i) 25 frequencies

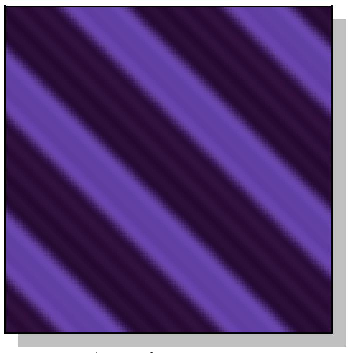
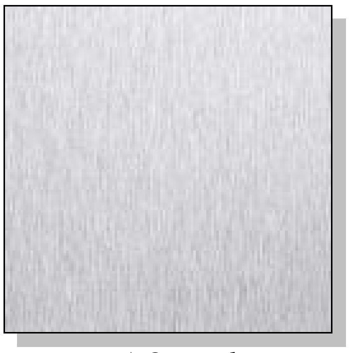
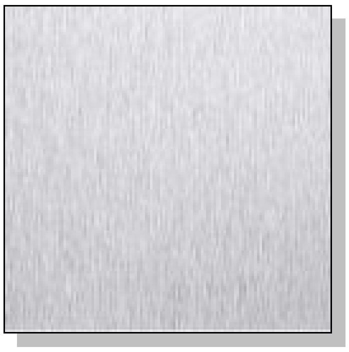
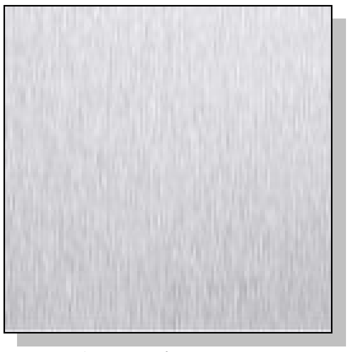
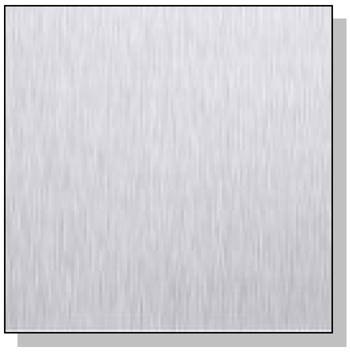

j) 10 frequencies

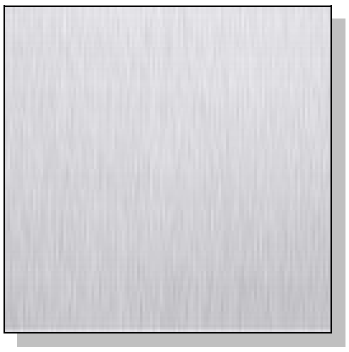
Figure A3: Software Frequency Tests: Silk Stripes

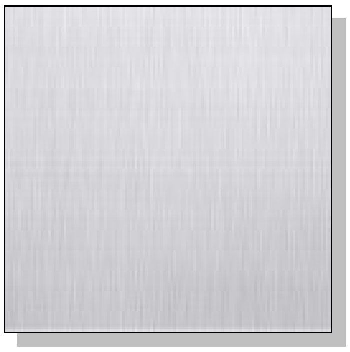

a) Original

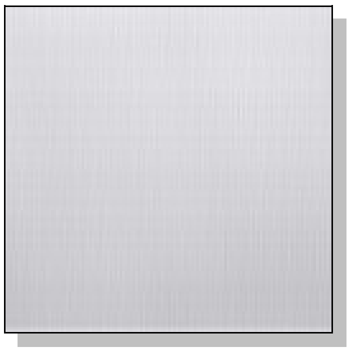

b) 5000 frequencies

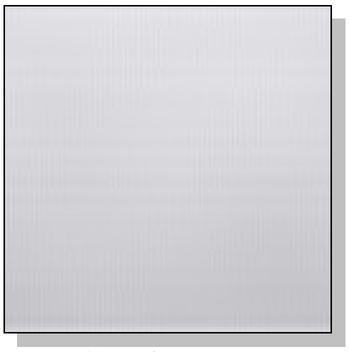

c) 2500 frequencies

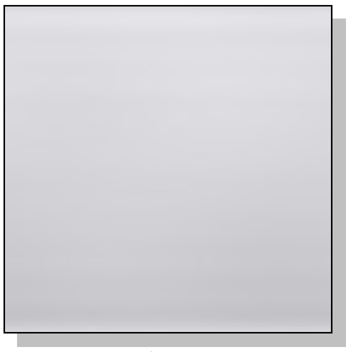

d) 1000 frequencies


e) 500 frequencies


f) 250 frequencies


g) 100 frequencies


h) 50 frequencies


i) 25 frequencies

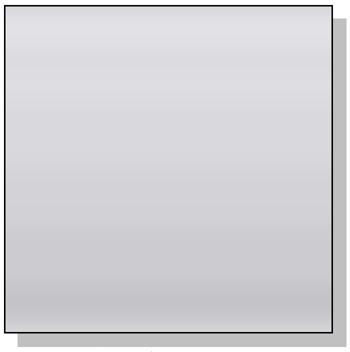
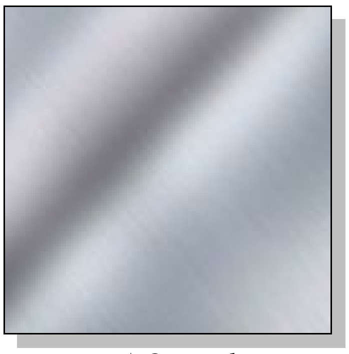
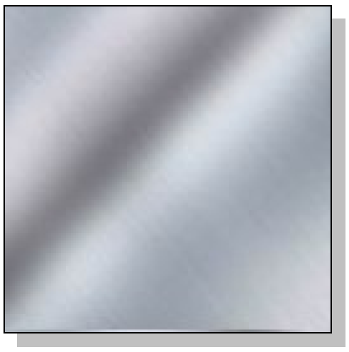
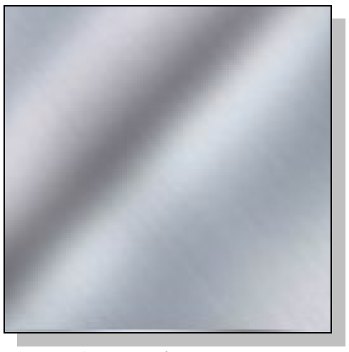
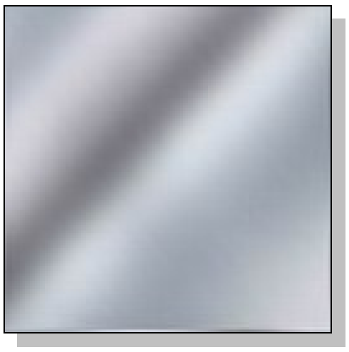

j) 10 frequencies

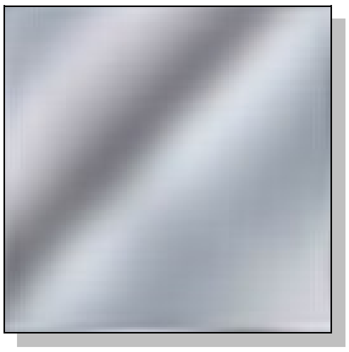
Figure A4: Software Frequency Tests: Brushed Metal

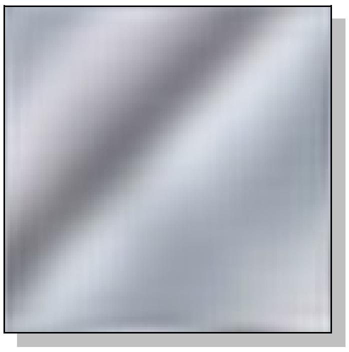

a) Original

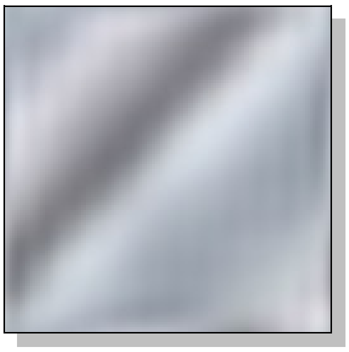

b) 5000 frequencies

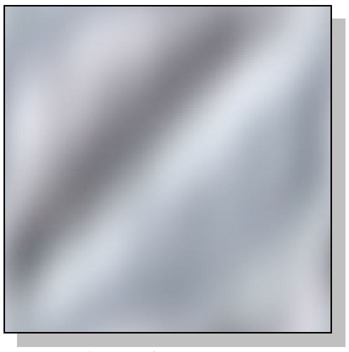

c) 2500 frequencies

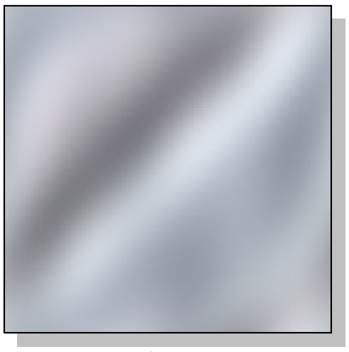

d) 1000 frequencies


e) 500 frequencies


f) 250 frequencies


g) 100 frequencies


h) 50 frequencies


i) 25 frequencies

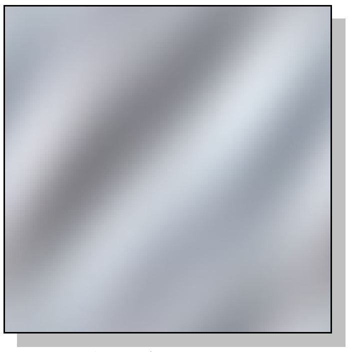
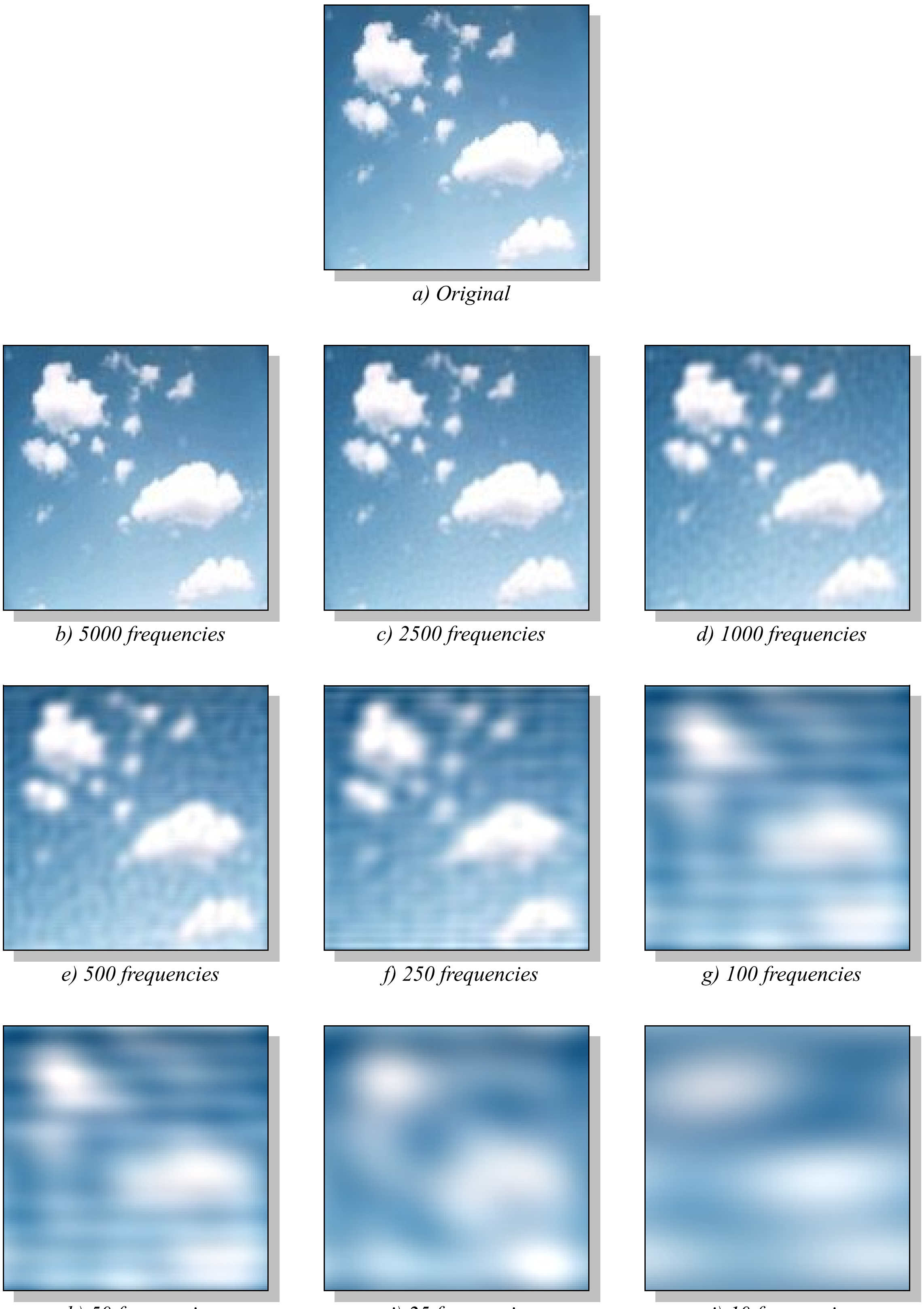

j) 10 frequencies

Figure A5: Software Frequency Tests: Brushed Metal 2



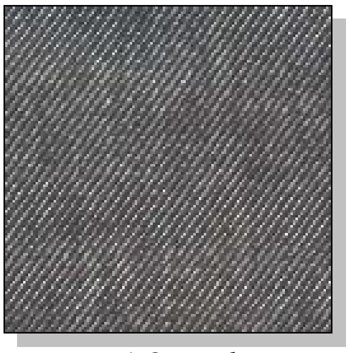
a) Original

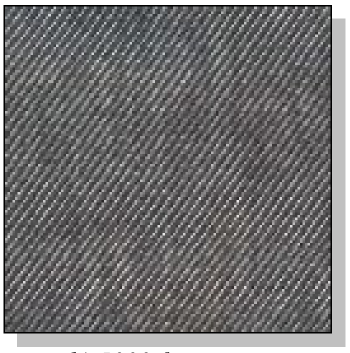
b) 5000 frequencies *c) 2500 frequencies* *d) 1000 frequencies*

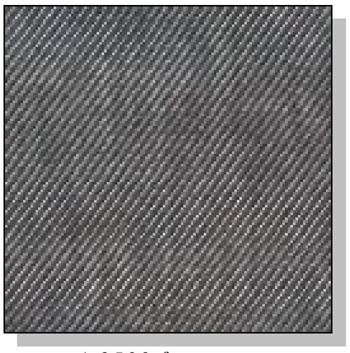
e) 500 frequencies *f) 250 frequencies* *g) 100 frequencies*

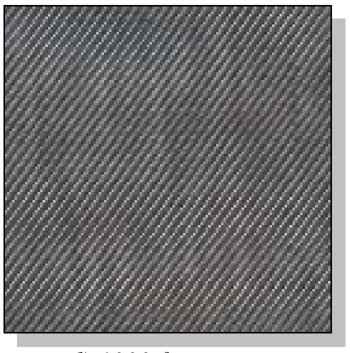
h) 50 frequencies *i) 25 frequencies* *j) 10 frequencies*

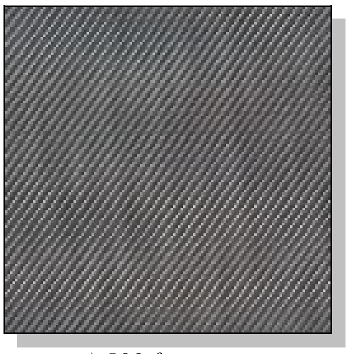
Figure A6: Software Frequency Tests: Clouds

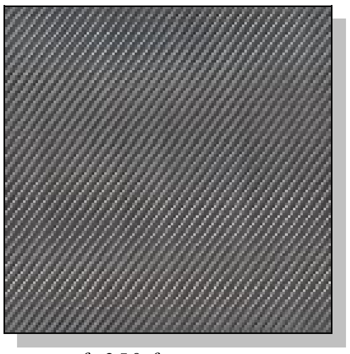

a) Original

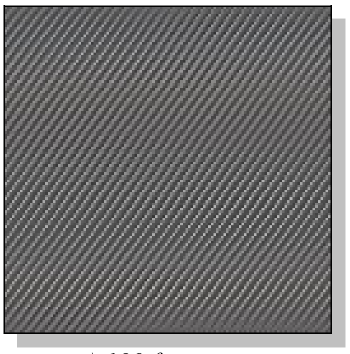

b) 5000 frequencies

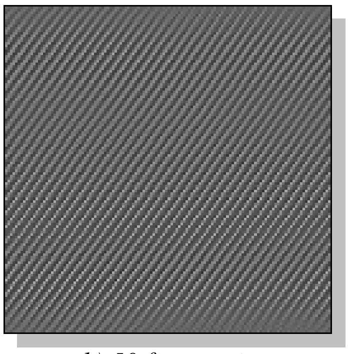

c) 2500 frequencies

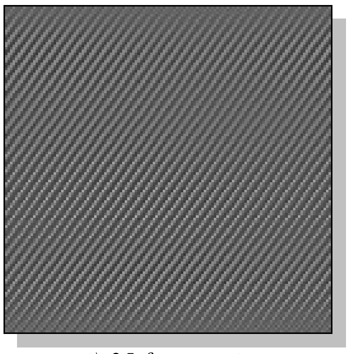

d) 1000 frequencies


e) 500 frequencies


f) 250 frequencies


g) 100 frequencies


h) 50 frequencies


i) 25 frequencies

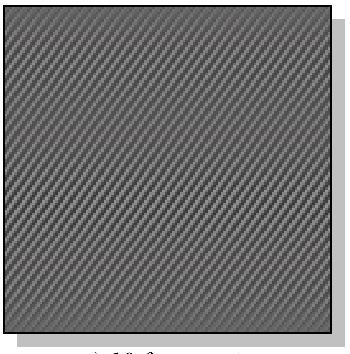
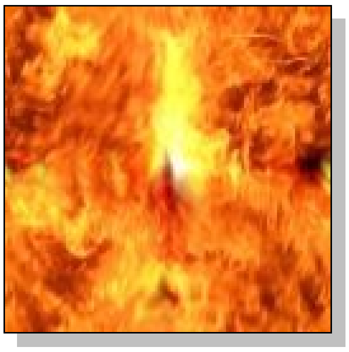
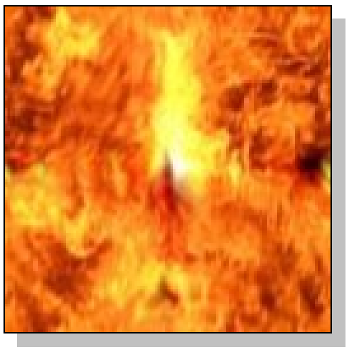
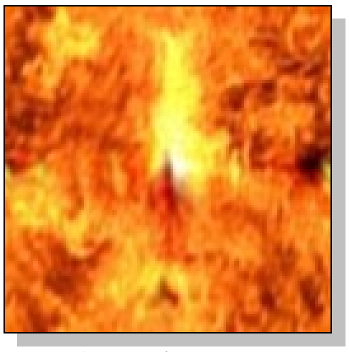
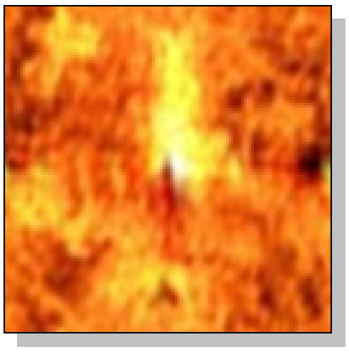

j) 10 frequencies

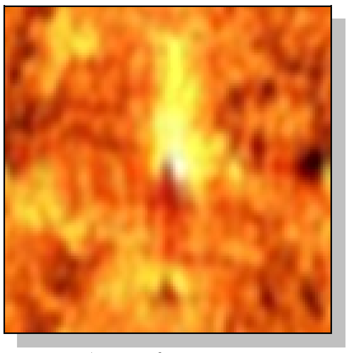
Figure A7: Software Frequency Tests: Black Denim

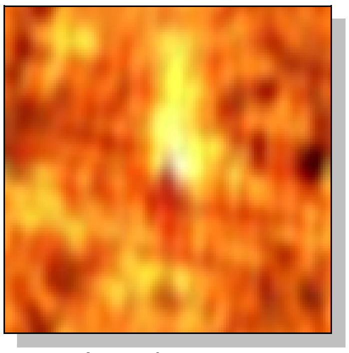

a) Original

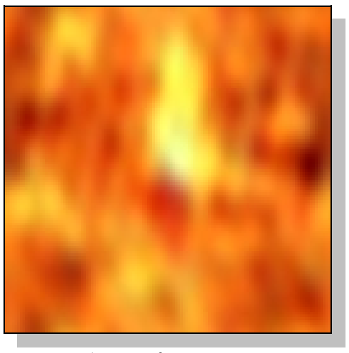

b) 5000 frequencies

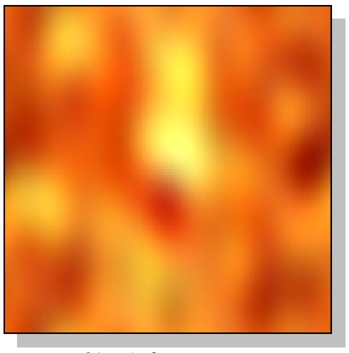

c) 2500 frequencies

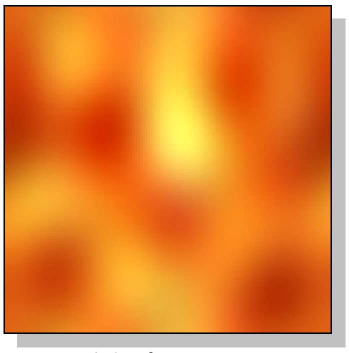

d) 1000 frequencies


e) 500 frequencies


f) 250 frequencies


g) 100 frequencies


h) 50 frequencies


i) 25 frequencies

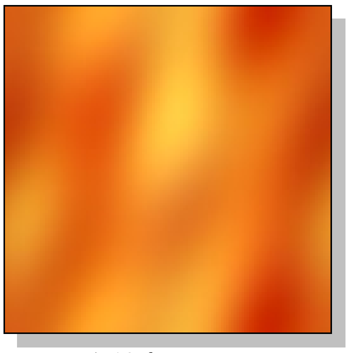
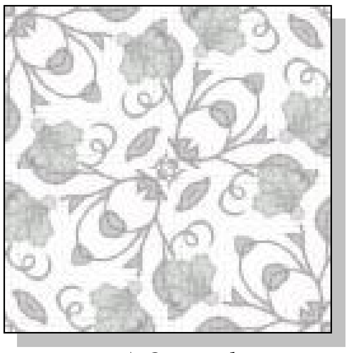
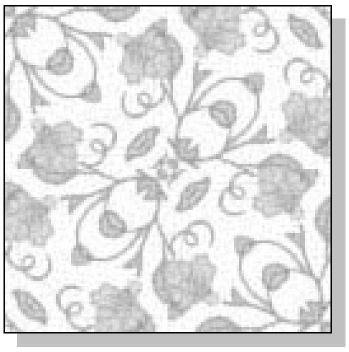
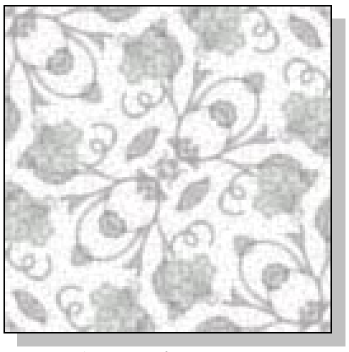
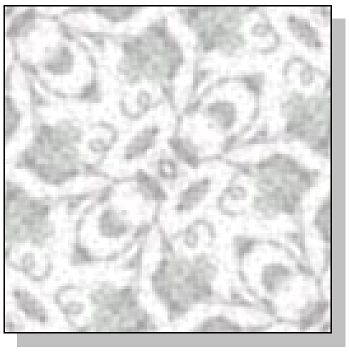

j) 10 frequencies

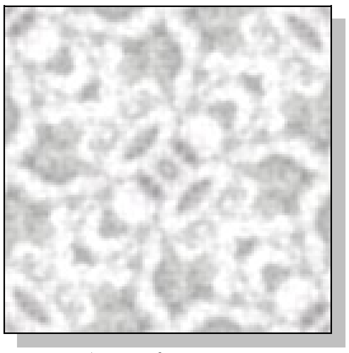
Figure A8: Software Frequency Tests: Fire

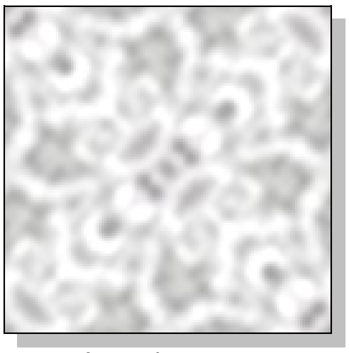

a) Original

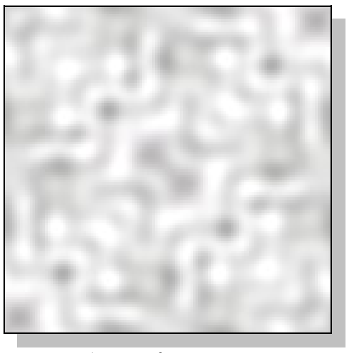

b) 5000 frequencies

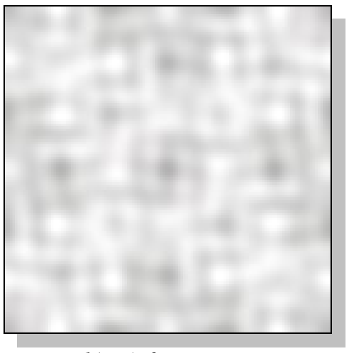

c) 2500 frequencies

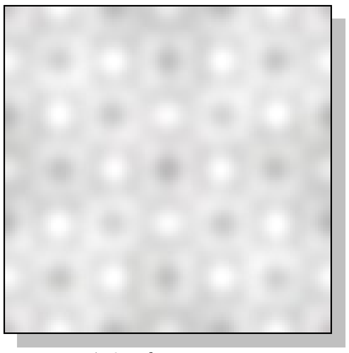

d) 1000 frequencies


e) 500 frequencies


f) 250 frequencies


g) 100 frequencies


h) 50 frequencies


i) 25 frequencies

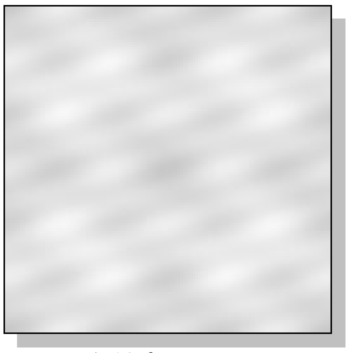
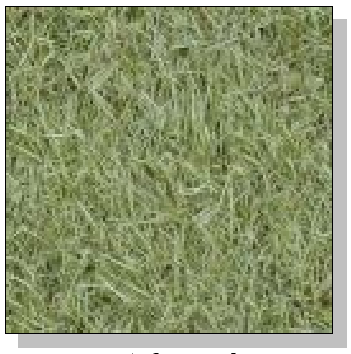
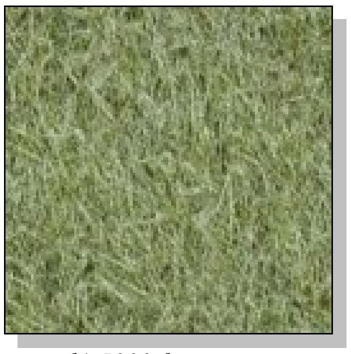
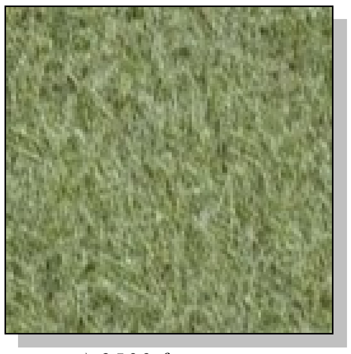
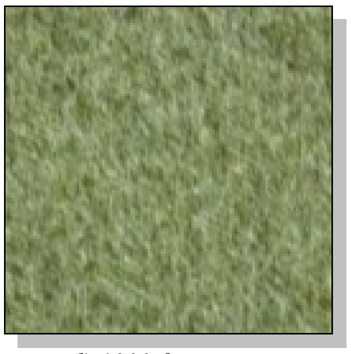

j) 10 frequencies

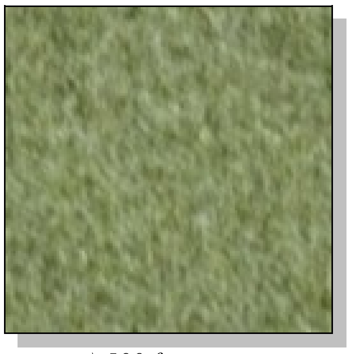
Figure A9: Software Frequency Tests: Pencilled Floral

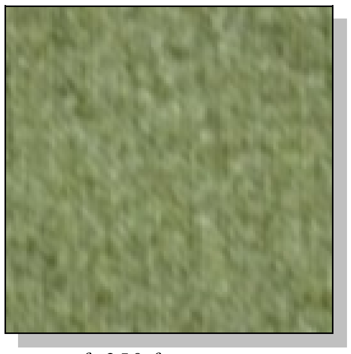

a) Original

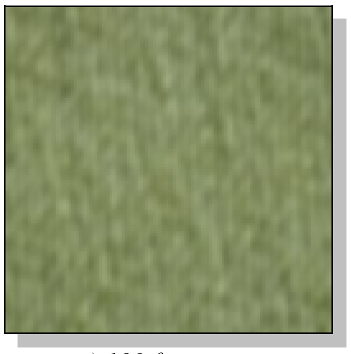

b) 5000 frequencies

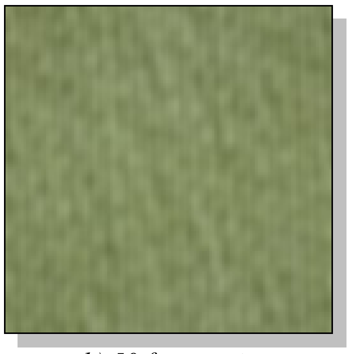

c) 2500 frequencies

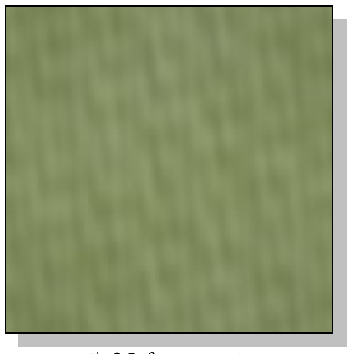

d) 1000 frequencies


e) 500 frequencies


f) 250 frequencies


g) 100 frequencies


h) 50 frequencies


i) 25 frequencies

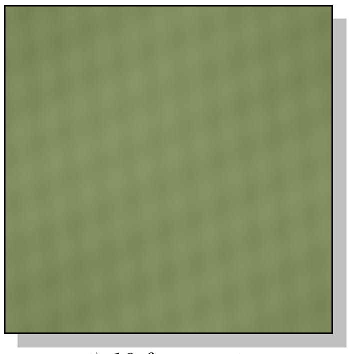
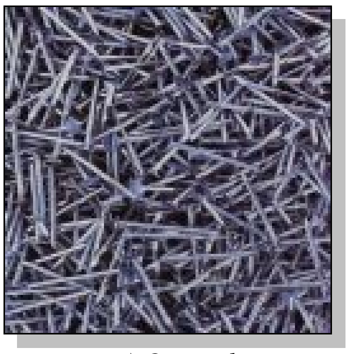
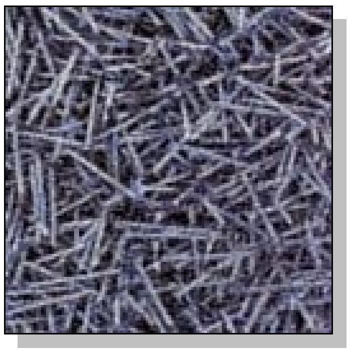
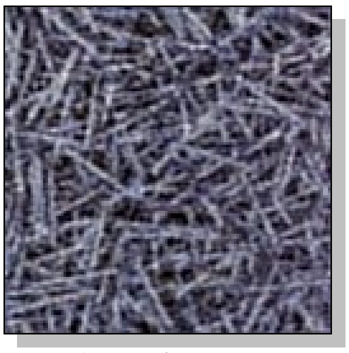
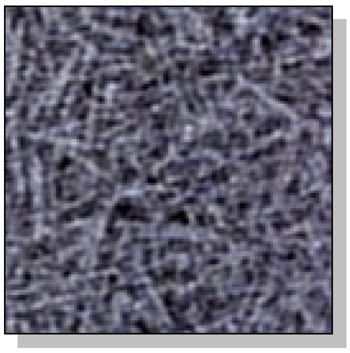

j) 10 frequencies

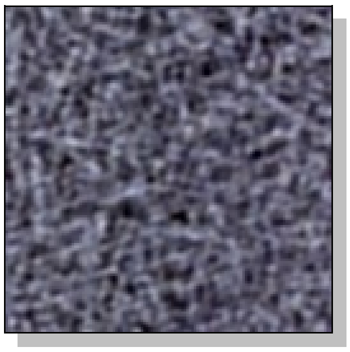
Figure A10: Software Frequency Tests: Grass

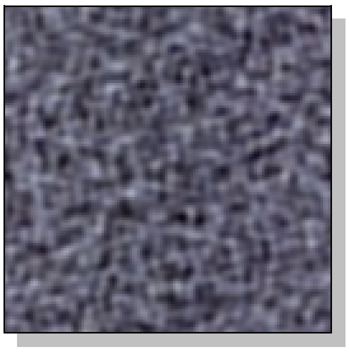

a) Original

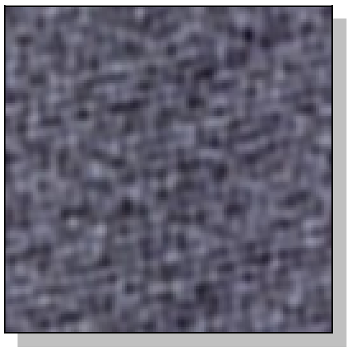

b) 5000 frequencies

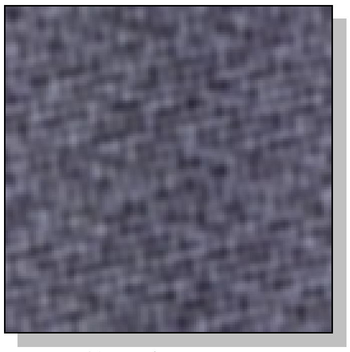

c) 2500 frequencies

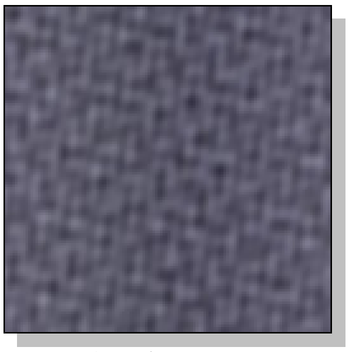

d) 1000 frequencies


e) 500 frequencies


f) 250 frequencies


g) 100 frequencies


h) 50 frequencies


i) 25 frequencies

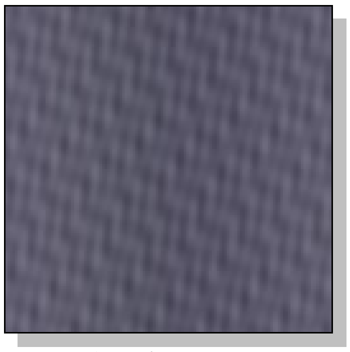
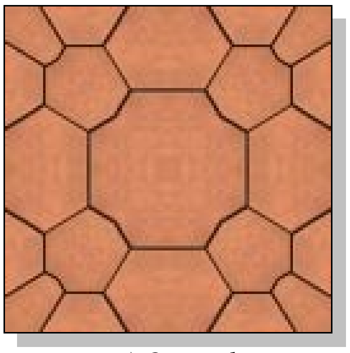
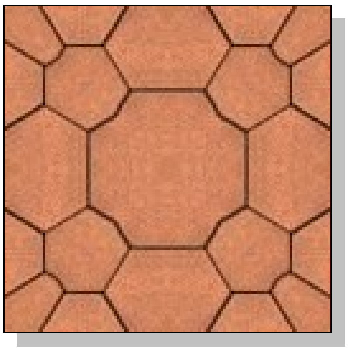
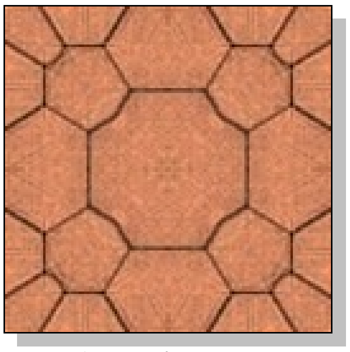
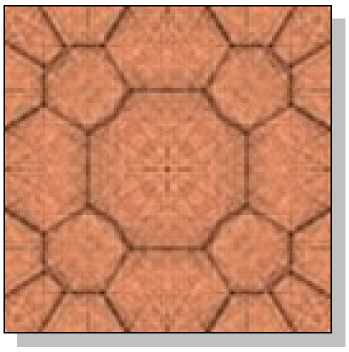

j) 10 frequencies

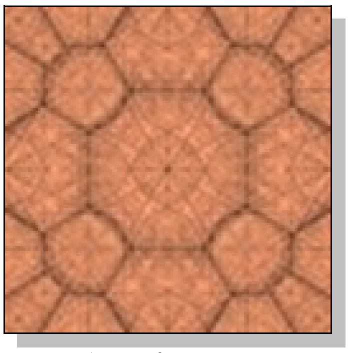
Figure A11: Software Frequency Tests: Nails

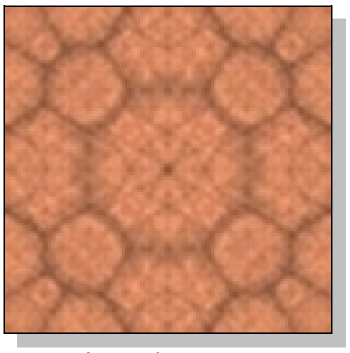

a) Original

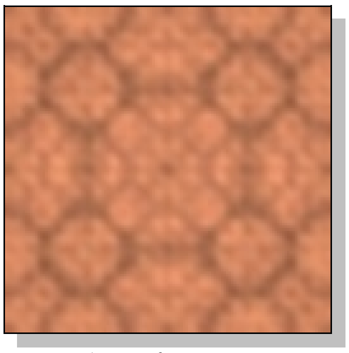

b) 5000 frequencies

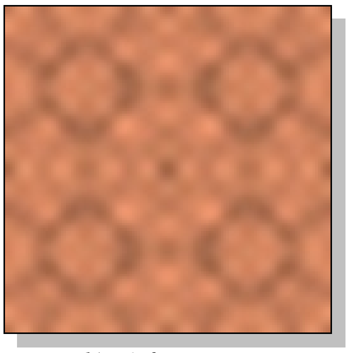

c) 2500 frequencies

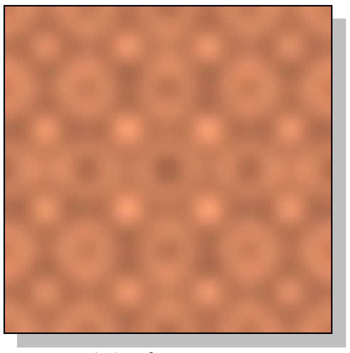

d) 1000 frequencies


e) 500 frequencies


f) 250 frequencies


g) 100 frequencies


h) 50 frequencies


i) 25 frequencies

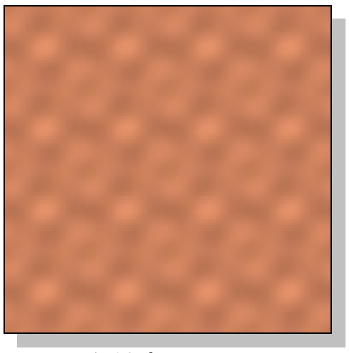
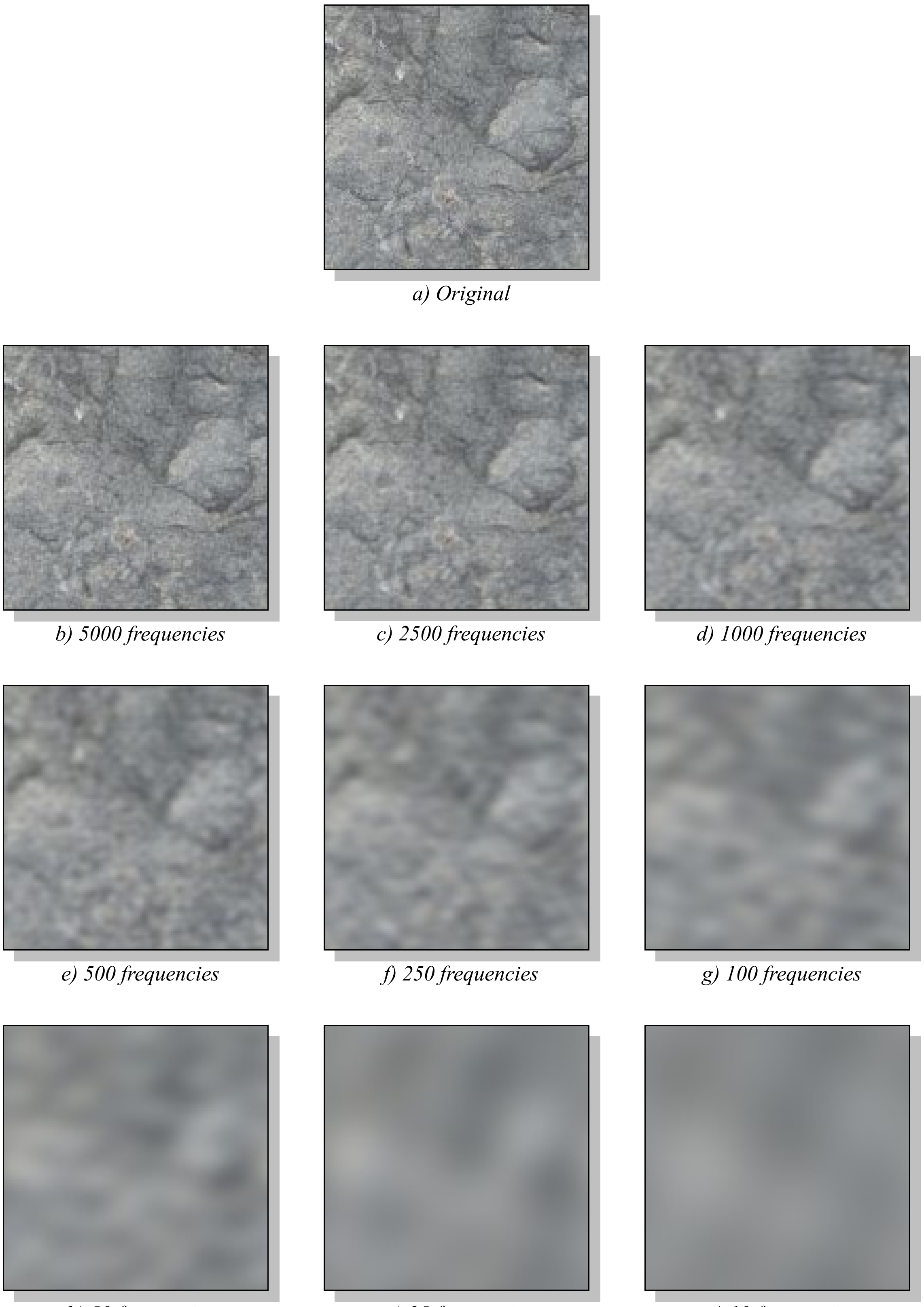

j) 10 frequencies

Figure A12: Software Frequency Tests: Floor Tiles

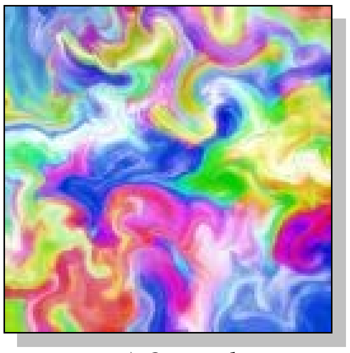

a) Original

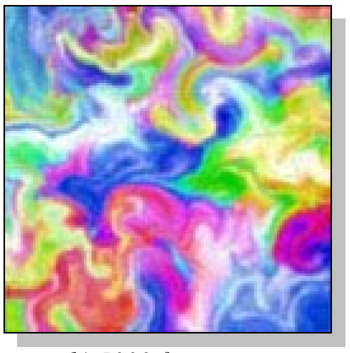
b) 5000 frequencies *c) 2500 frequencies* *d) 1000 frequencies*

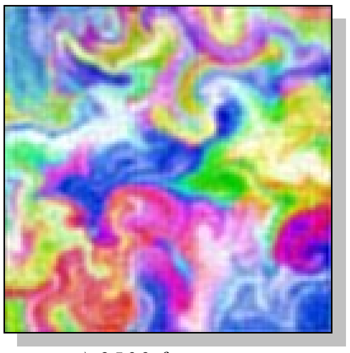
e) 500 frequencies *f) 250 frequencies* *g) 100 frequencies*

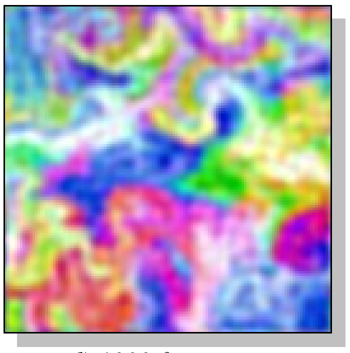
h) 50 frequencies *i) 25 frequencies* *j) 10 frequencies*

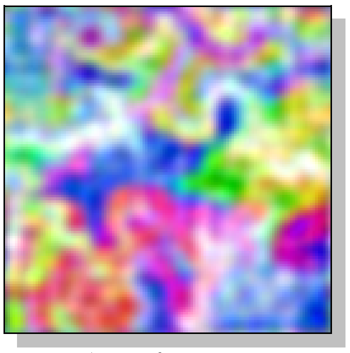
Figure A13: Software Frequency Tests: Rock

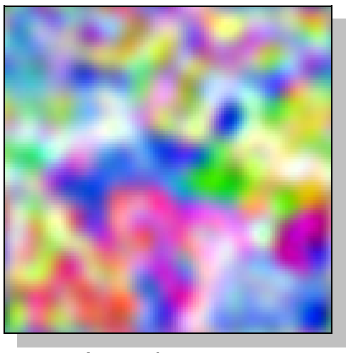

a) Original

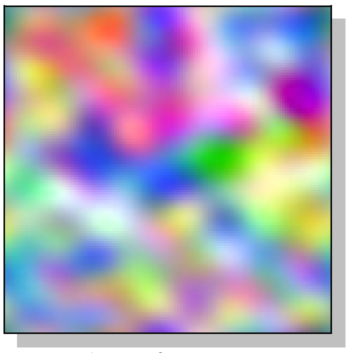

b) 5000 frequencies

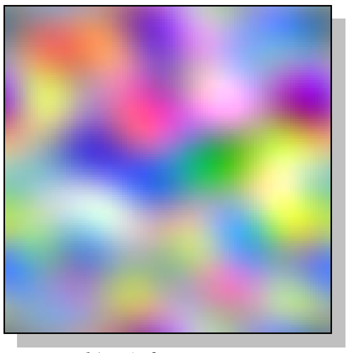

c) 2500 frequencies

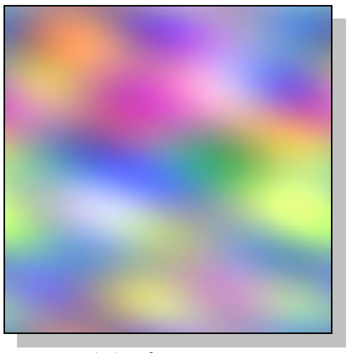

d) 1000 frequencies


e) 500 frequencies


f) 250 frequencies


g) 100 frequencies


h) 50 frequencies


i) 25 frequencies

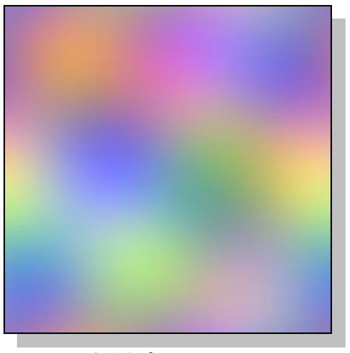
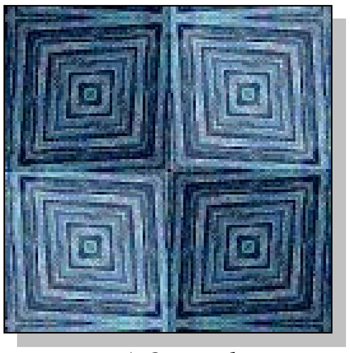
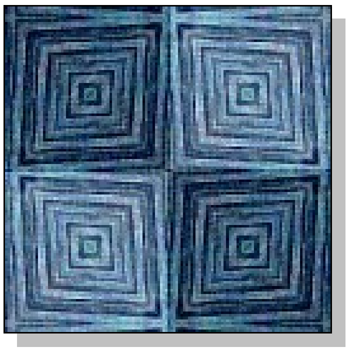
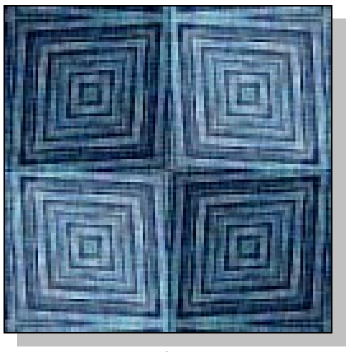
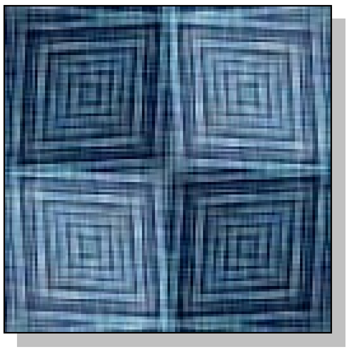

j) 10 frequencies

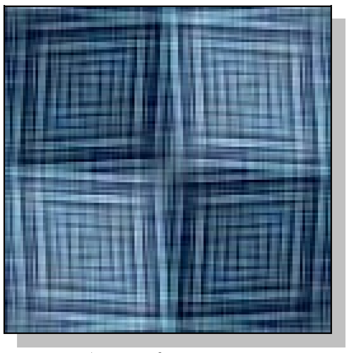
Figure A14: Software Frequency Tests: Colourful Swirl

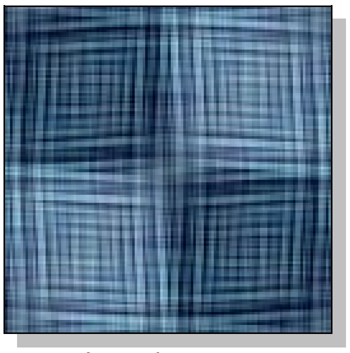

a) Original

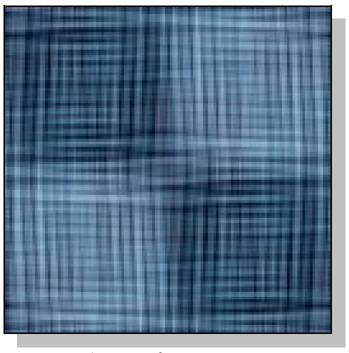

b) 5000 frequencies

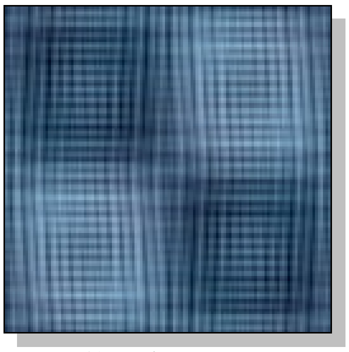

c) 2500 frequencies

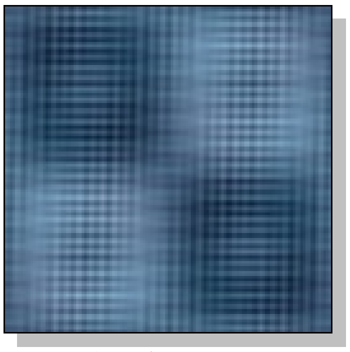

d) 1000 frequencies


e) 500 frequencies


f) 250 frequencies


g) 100 frequencies


h) 50 frequencies


i) 25 frequencies

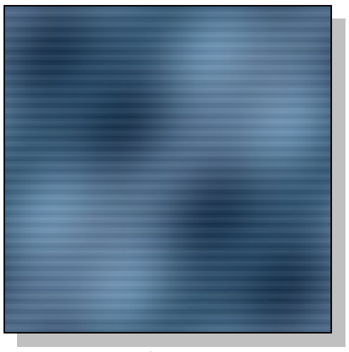
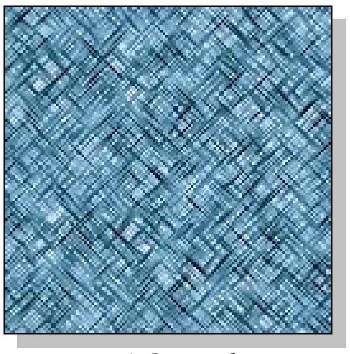
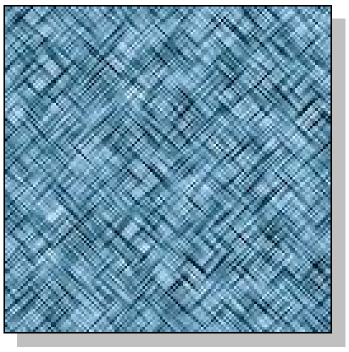
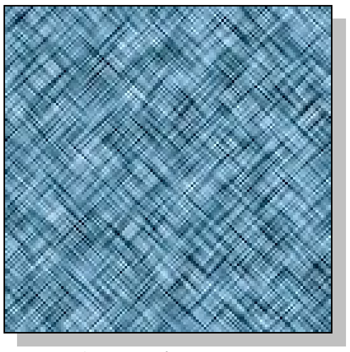
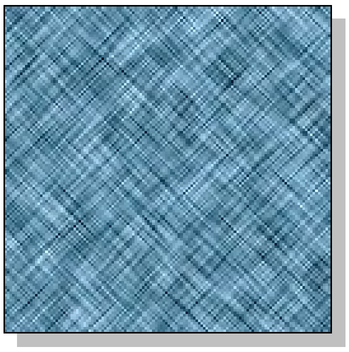

j) 10 frequencies

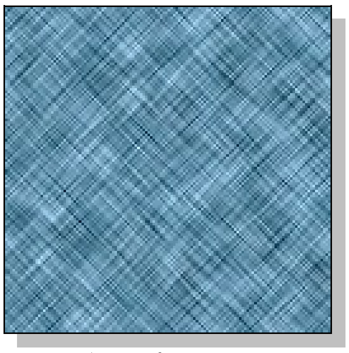
Figure A15: Software Frequency Tests: Tiles

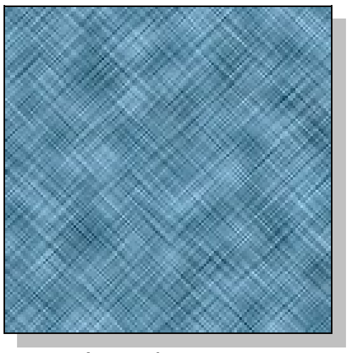

a) Original

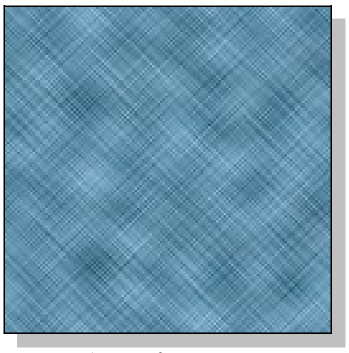

b) 5000 frequencies

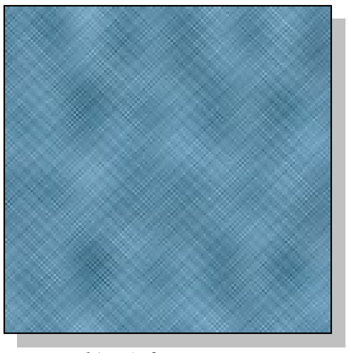

c) 2500 frequencies

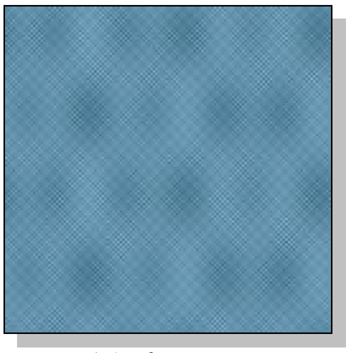

d) 1000 frequencies


e) 500 frequencies


f) 250 frequencies


g) 100 frequencies


h) 50 frequencies


i) 25 frequencies

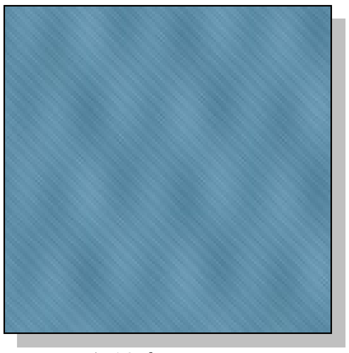
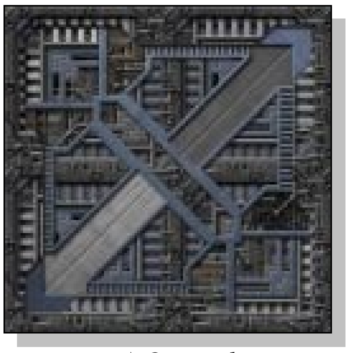
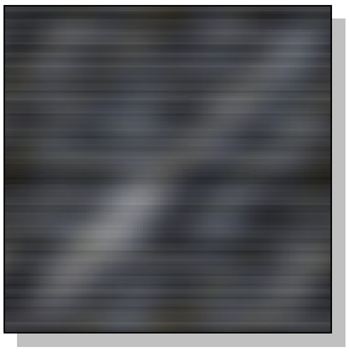

j) 10 frequencies

Figure A16: Software Frequency Tests: Ice

Appendix B: Hardware Frequency Tests

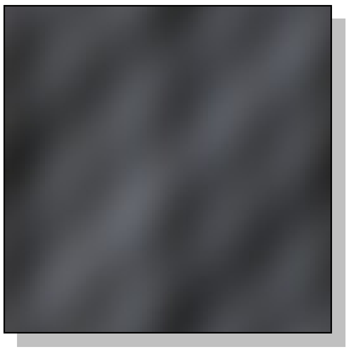


a) Original



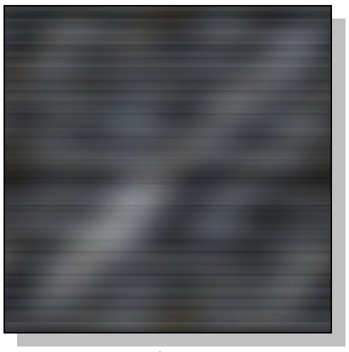
b1) 50 frequencies

b2) 25 frequencies

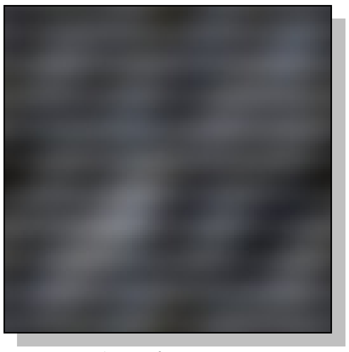


b3) 10 frequencies

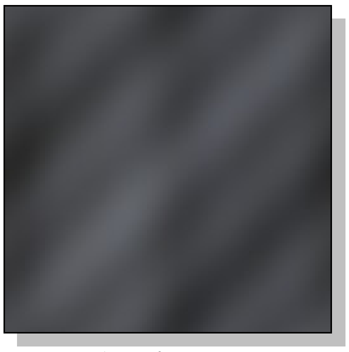
b) Software



c1) 50 frequencies



c2) 25 frequencies



c3) 10 frequencies

c) Hardware

Figure B1: Hardware Frequency Tests: Alien Tile

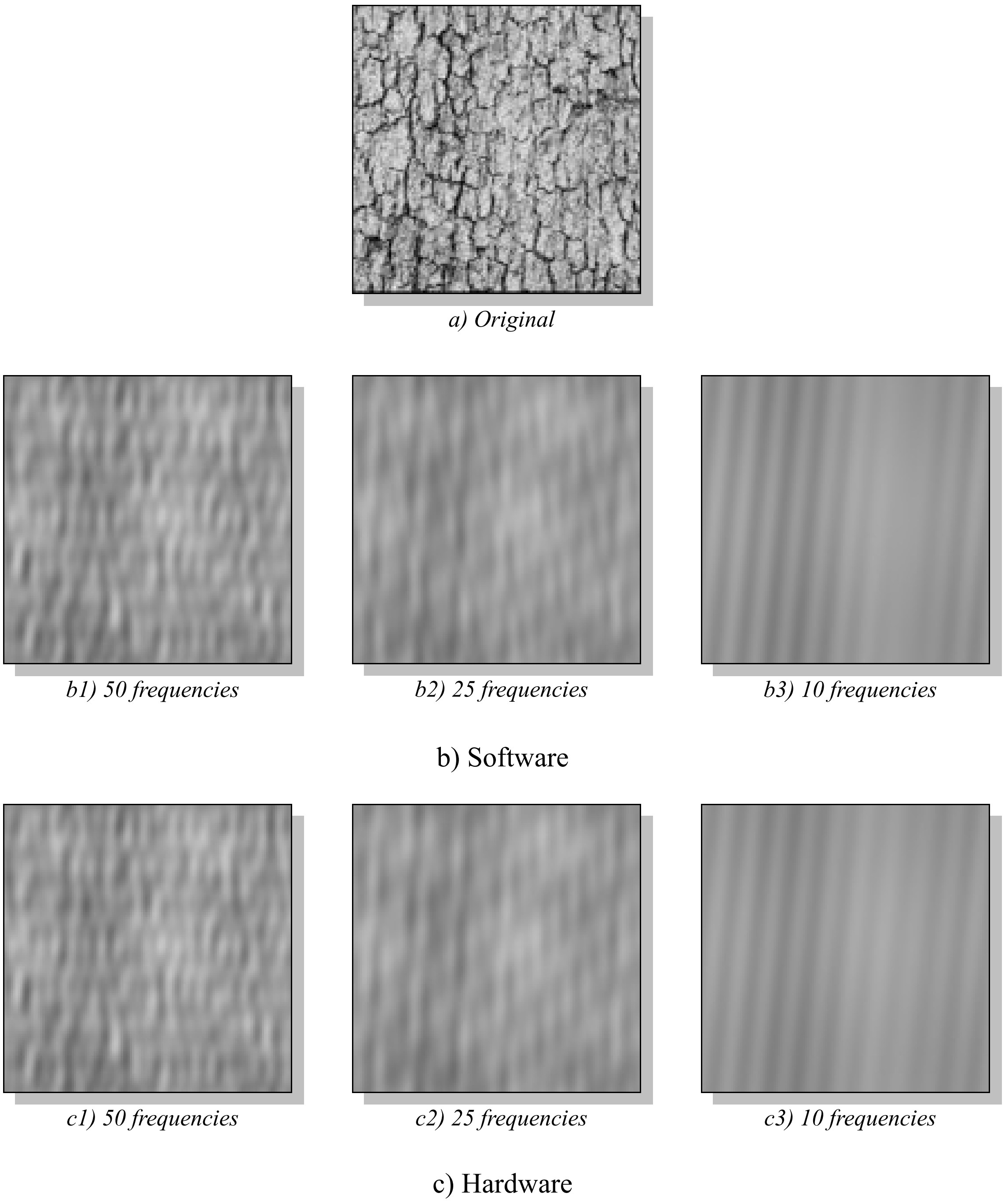
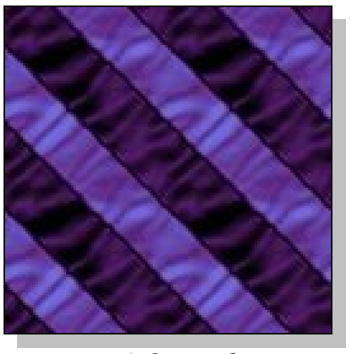
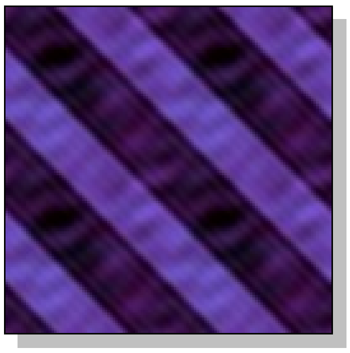
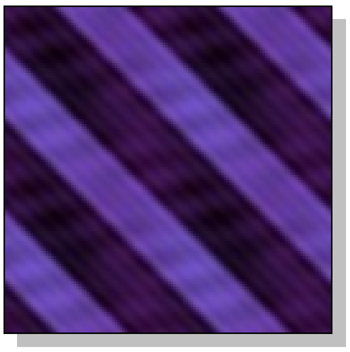
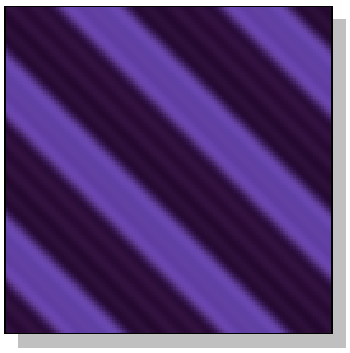


Figure B2: Hardware Frequency Tests: Bark

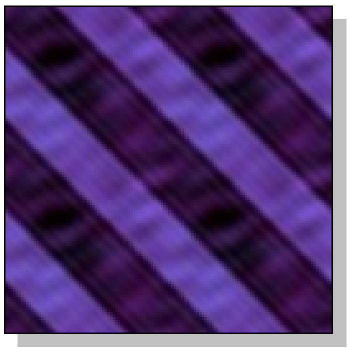

a) Original

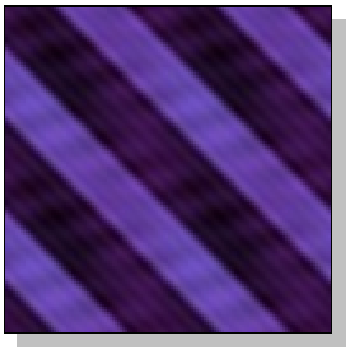

b1) 50 frequencies

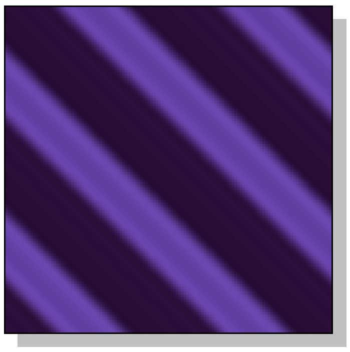

b2) 25 frequencies


b3) 10 frequencies

b) Software

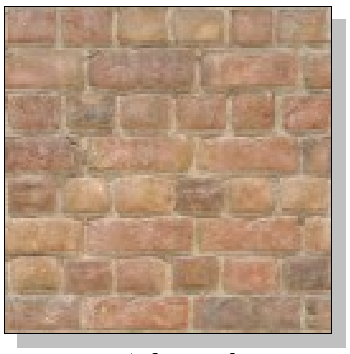

c1) 50 frequencies

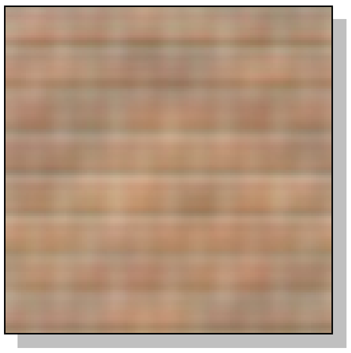

c2) 25 frequencies

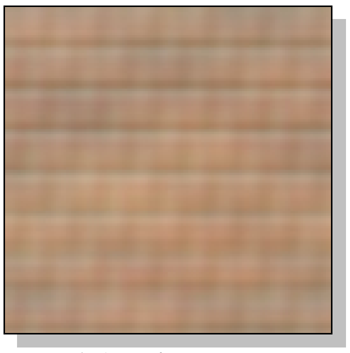

c3) 10 frequencies

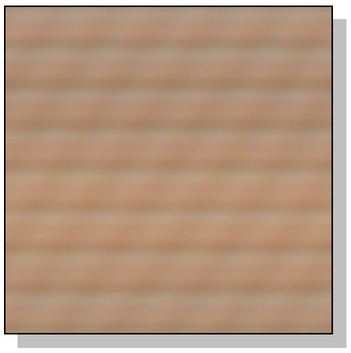
c) Hardware

Figure B3: Hardware Frequency Tests: Silk Stripes

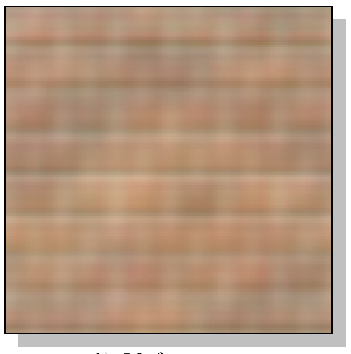

a) Original

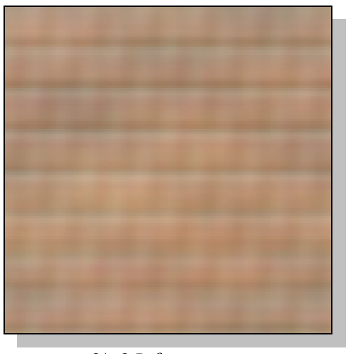

b1) 50 frequencies

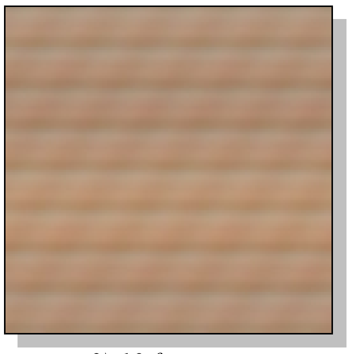

b2) 25 frequencies


b3) 10 frequencies

b) Software

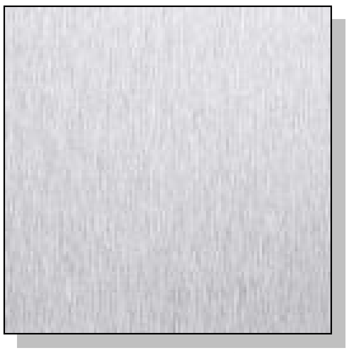

c1) 50 frequencies

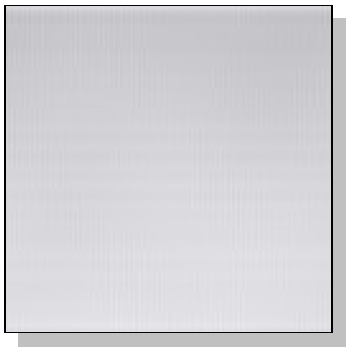

c2) 25 frequencies

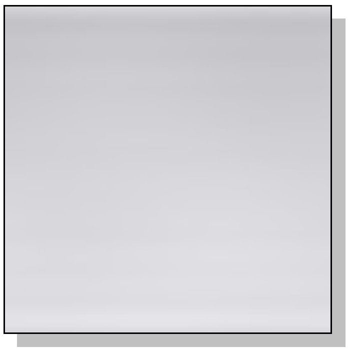

c3) 10 frequencies

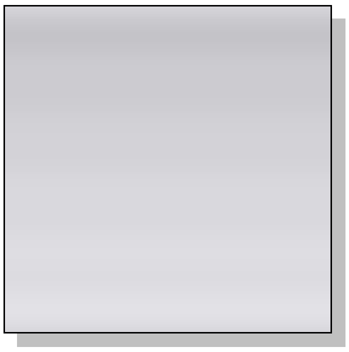
c) Hardware

Figure B4: Hardware Frequency Tests: Brick Wall

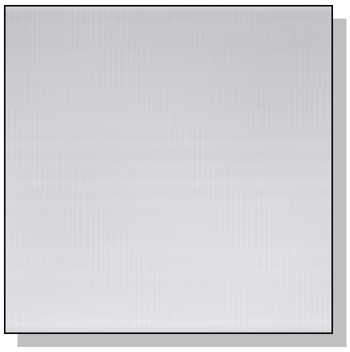

a) Original

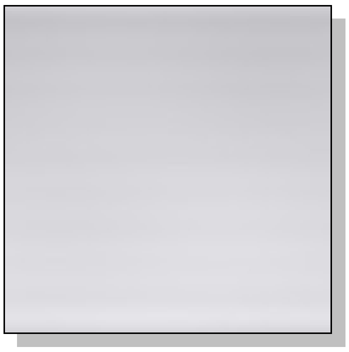

b1) 50 frequencies

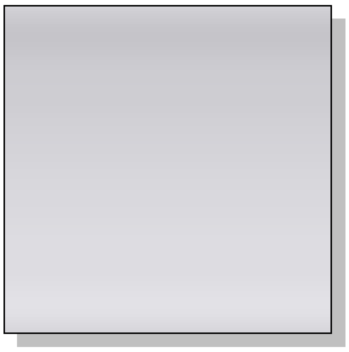

b2) 25 frequencies


b3) 10 frequencies

b) Software

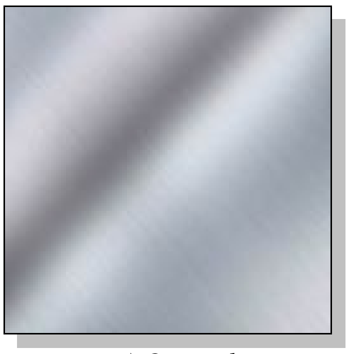

c1) 50 frequencies

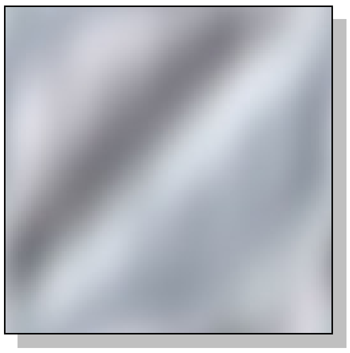

c2) 25 frequencies

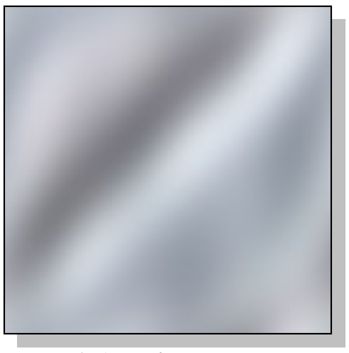

c3) 10 frequencies

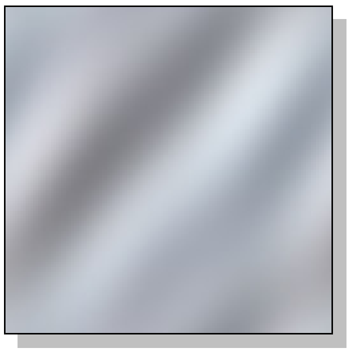
c) Hardware

Figure B5: Hardware Frequency Tests: Brushed Metal

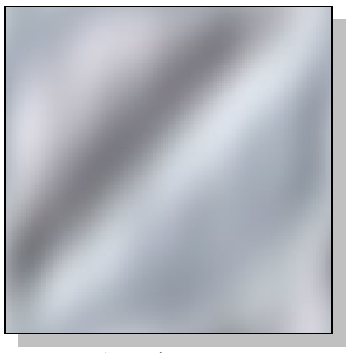

a) Original

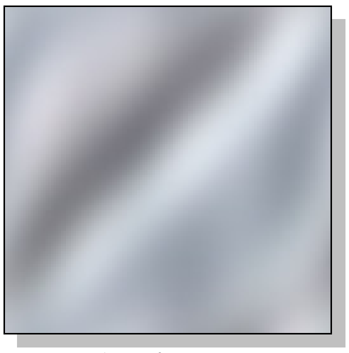

b1) 50 frequencies

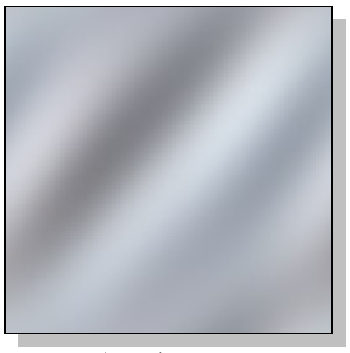

b2) 25 frequencies


b3) 10 frequencies

b) Software

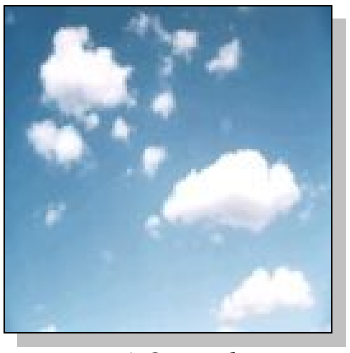

c1) 50 frequencies

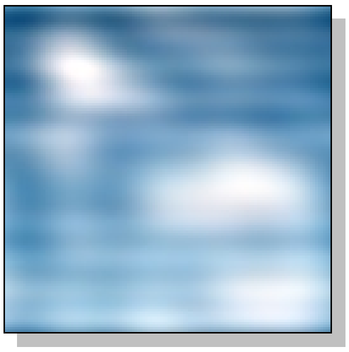

c2) 25 frequencies

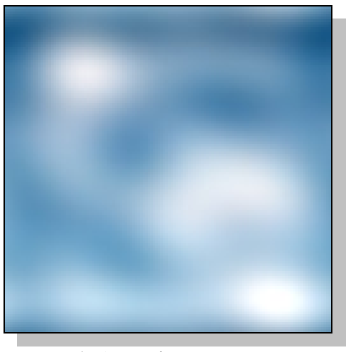

c3) 10 frequencies

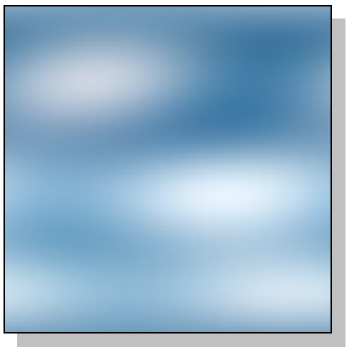
c) Hardware

Figure B6: Hardware Frequency Tests: Brushed Metal 2

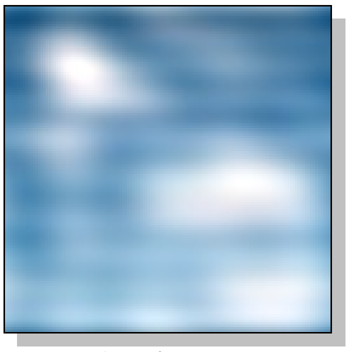

a) Original

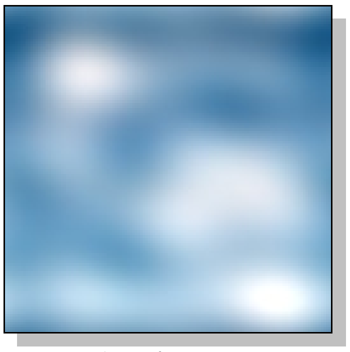

b1) 50 frequencies

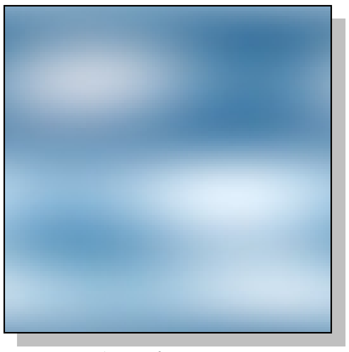

b2) 25 frequencies


b3) 10 frequencies

b) Software

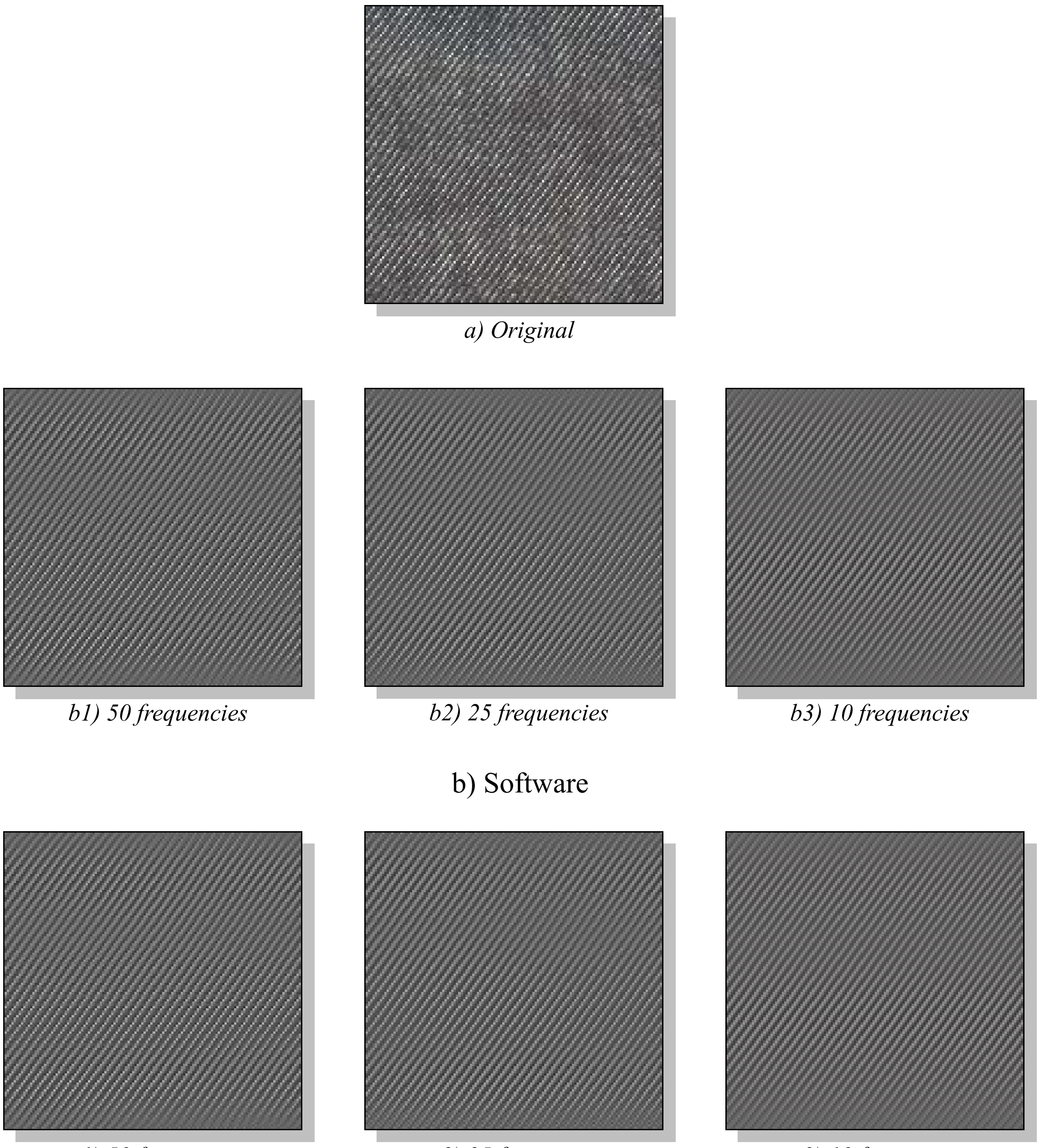

c1) 50 frequencies


c2) 25 frequencies


c3) 10 frequencies

c) Hardware

Figure B7: Hardware Frequency Tests: Clouds



a) Original

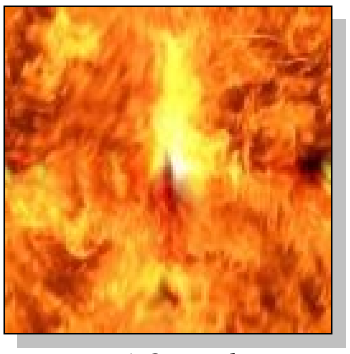
b1) 50 frequencies *b2) 25 frequencies* *b3) 10 frequencies*

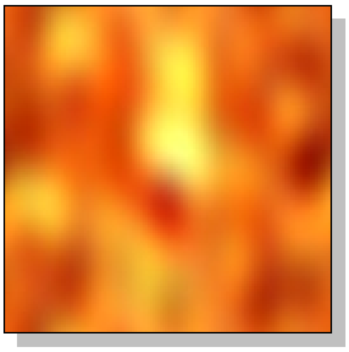
b) Software

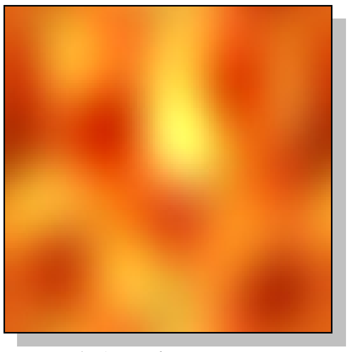
c1) 50 frequencies *c2) 25 frequencies* *c3) 10 frequencies*

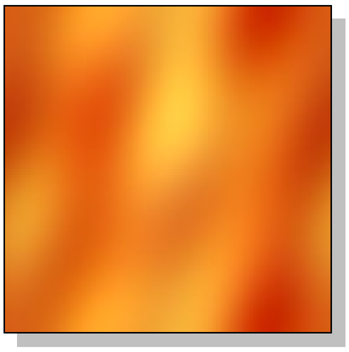
c) Hardware

Figure B8: Hardware Frequency Tests: Black Denim

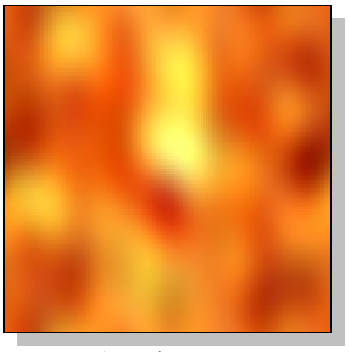

a) Original

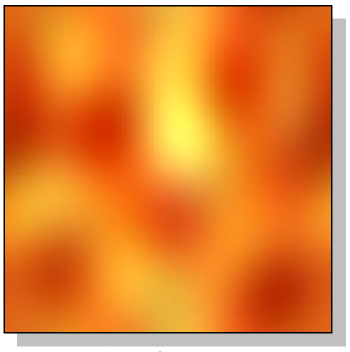

b1) 50 frequencies

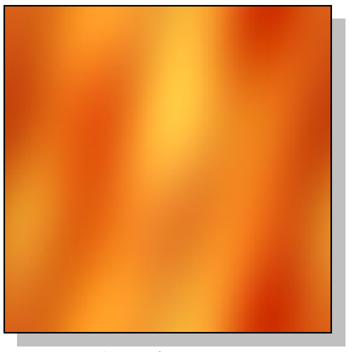

b2) 25 frequencies


b3) 10 frequencies

b) Software

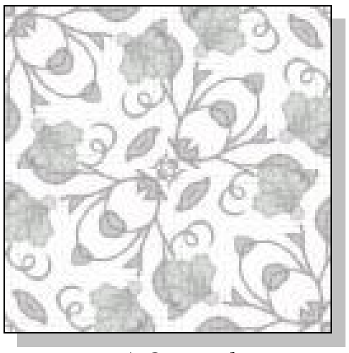

c1) 50 frequencies

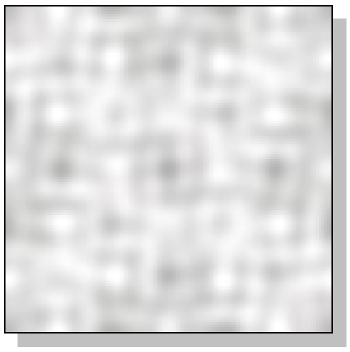

c2) 25 frequencies

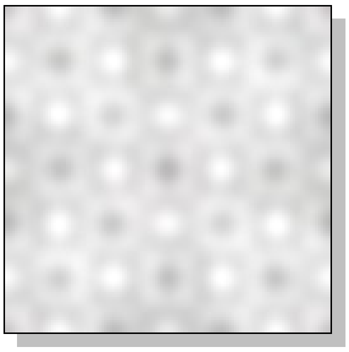

c3) 10 frequencies

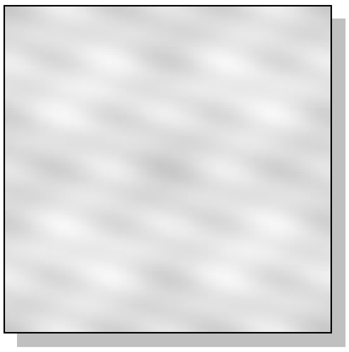
c) Hardware

Figure B9: Hardware Frequency Tests: Fire

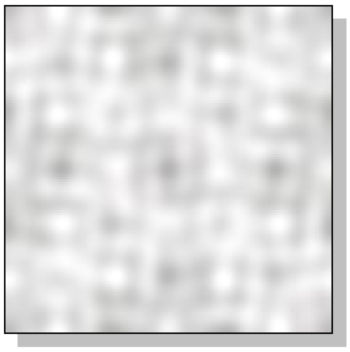

a) Original

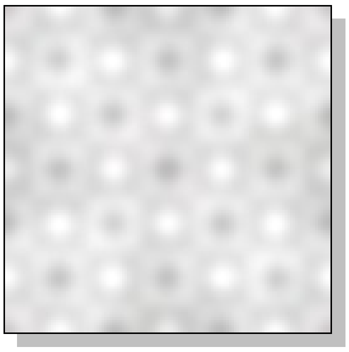

b1) 50 frequencies

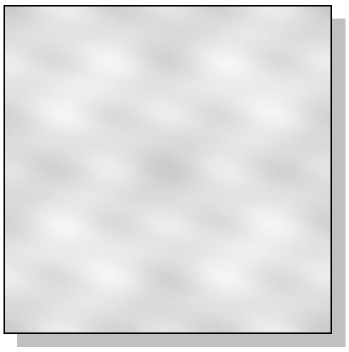

b2) 25 frequencies


b3) 10 frequencies

b) Software

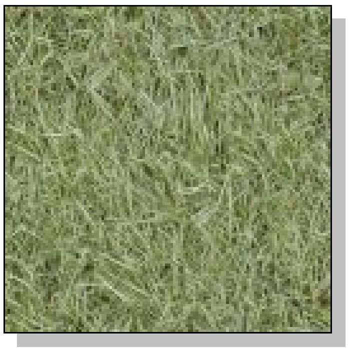

c1) 50 frequencies

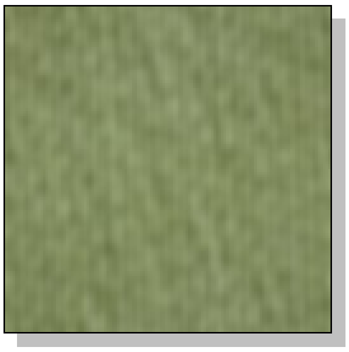

c2) 25 frequencies


c3) 10 frequencies

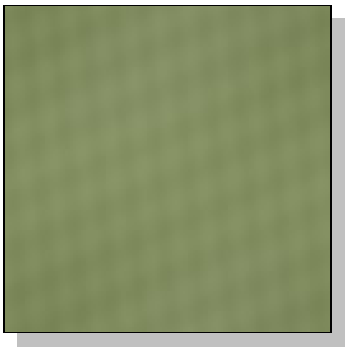
c) Hardware

Figure B10: Hardware Frequency Tests: Pencilled Floral

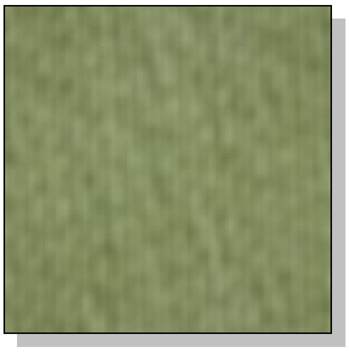

a) Original


b1) 50 frequencies

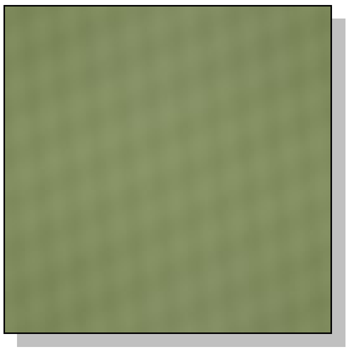
b2) 25 frequencies


b3) 10 frequencies

b) Software

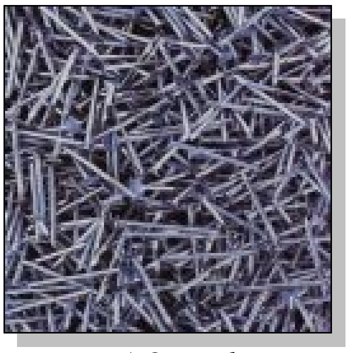

c1) 50 frequencies

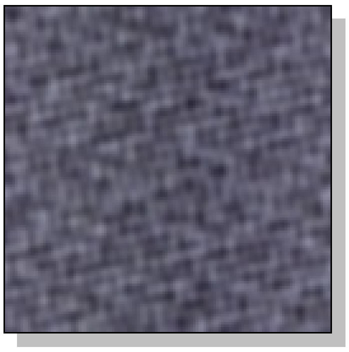
c2) 25 frequencies

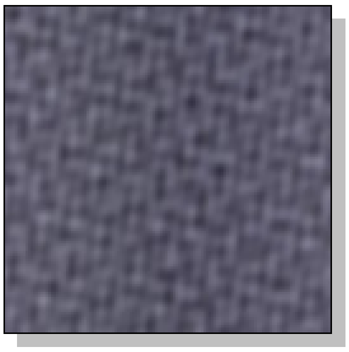

c3) 10 frequencies

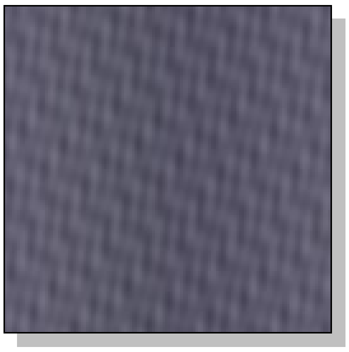
c) Hardware

Figure B11: Hardware Frequency Tests: Grass

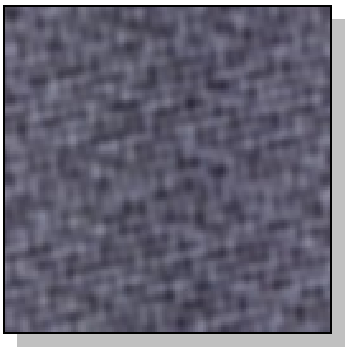

a) Original

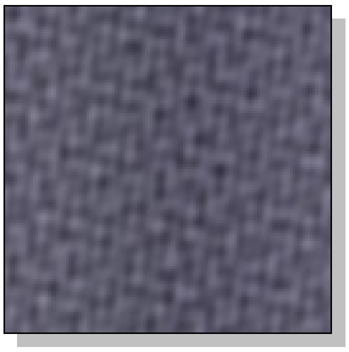

b1) 50 frequencies

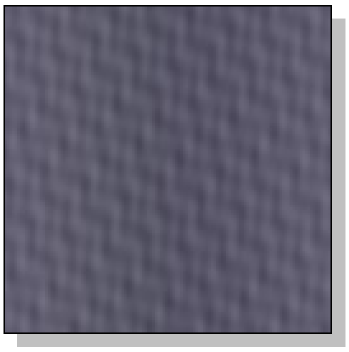

b2) 25 frequencies


b3) 10 frequencies

b) Software

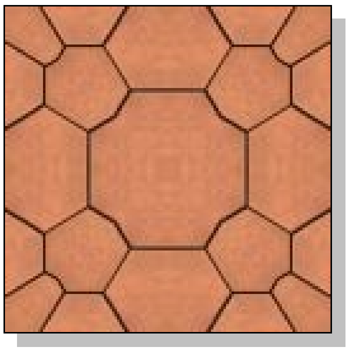

c1) 50 frequencies

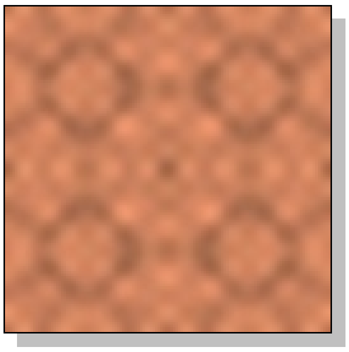

c2) 25 frequencies

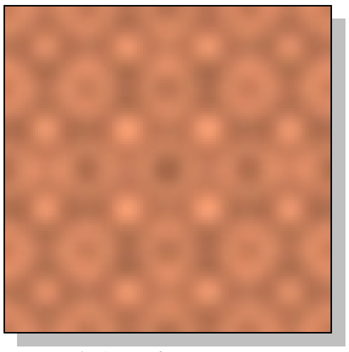

c3) 10 frequencies

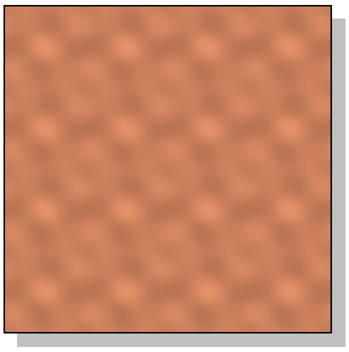
c) Hardware

Figure B12: Hardware Frequency Tests: Nails

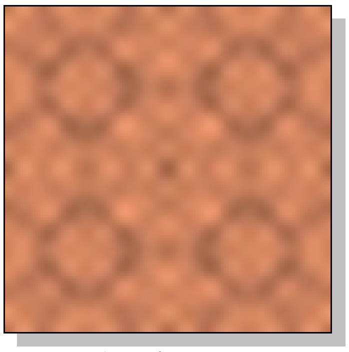

a) Original

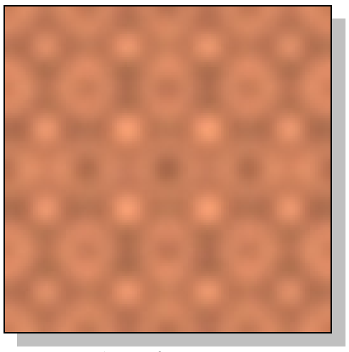

b1) 50 frequencies

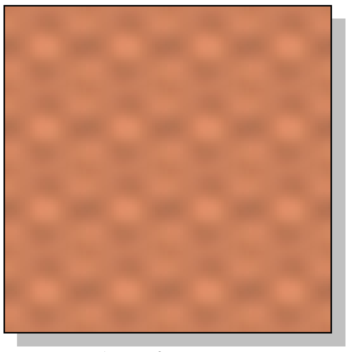

b2) 25 frequencies


b3) 10 frequencies

b) Software

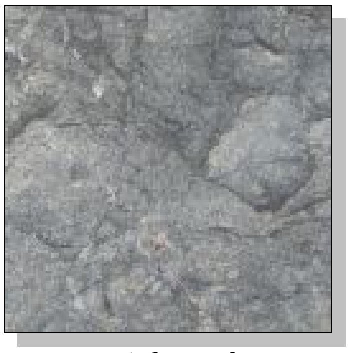

c1) 50 frequencies

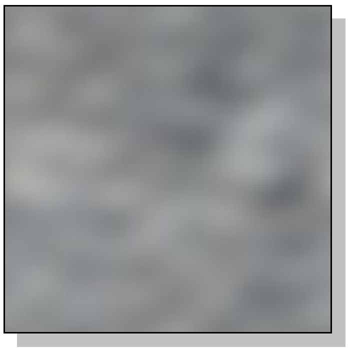

c2) 25 frequencies

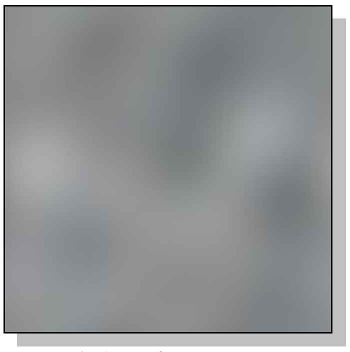

c3) 10 frequencies

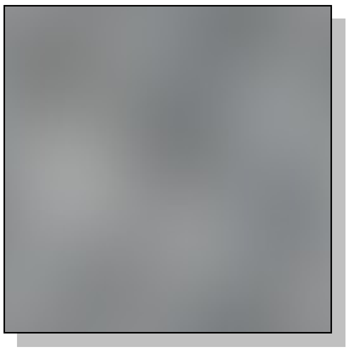
c) Hardware

Figure B13: Hardware Frequency Tests: Tiles

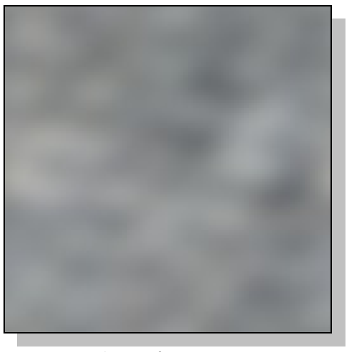

a) Original

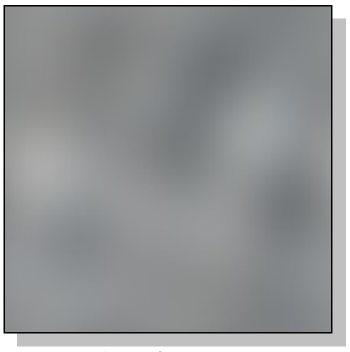

b1) 50 frequencies

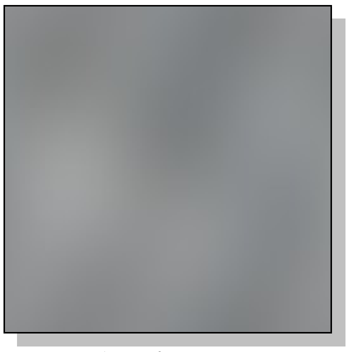

b2) 25 frequencies


b3) 10 frequencies

b) Software

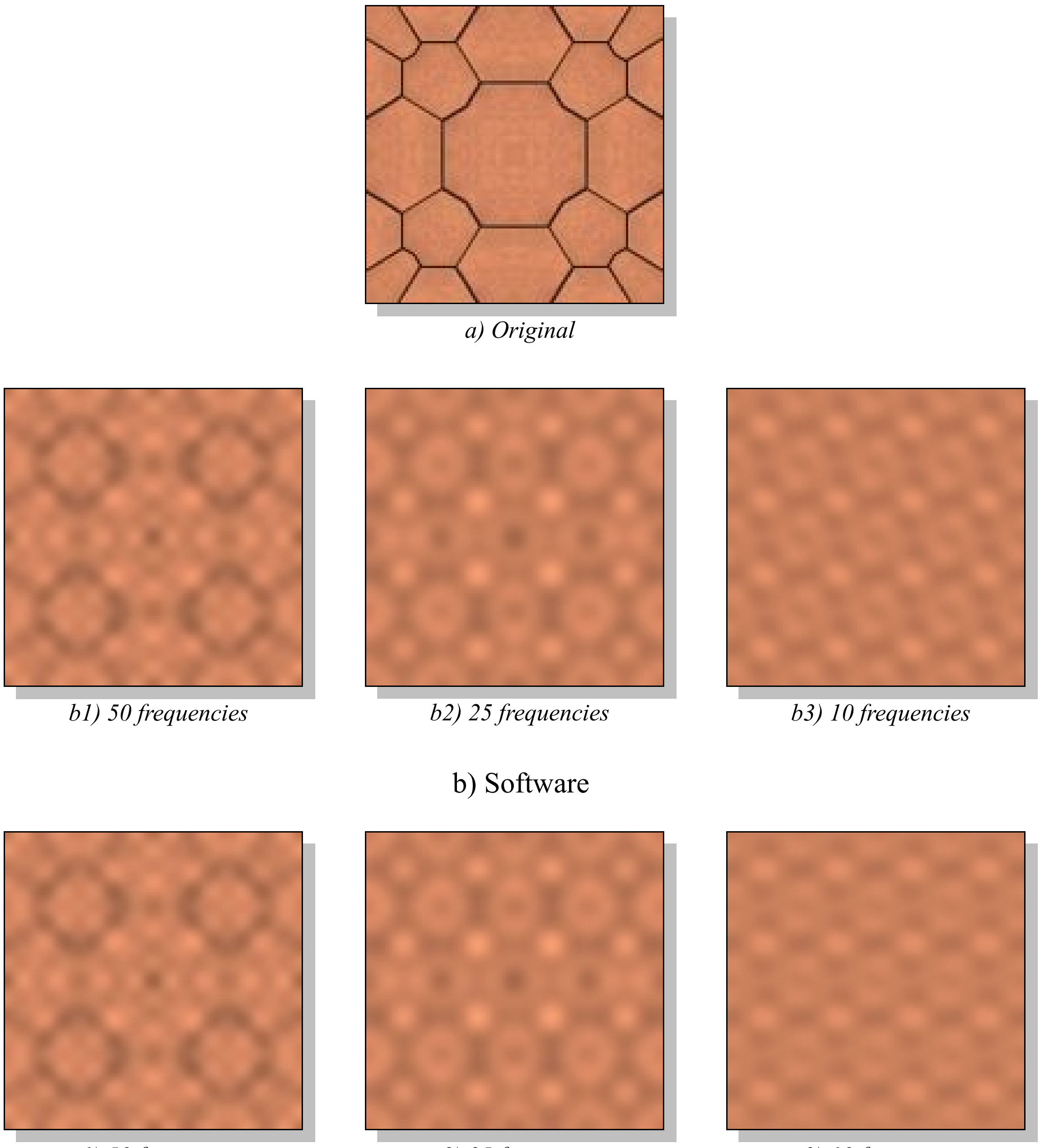

c1) 50 frequencies


c2) 25 frequencies


c3) 10 frequencies

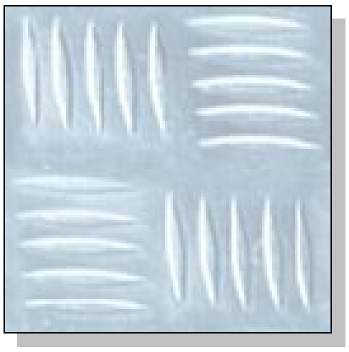
c) Hardware

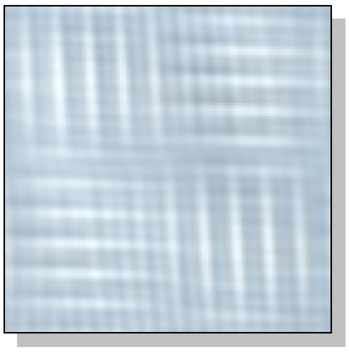
Figure B14: Hardware Frequency Tests: Tiles

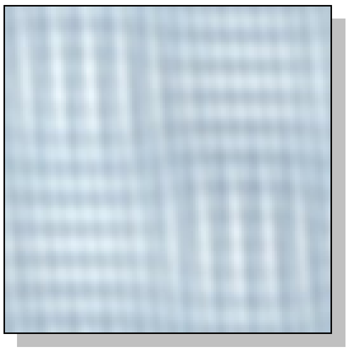


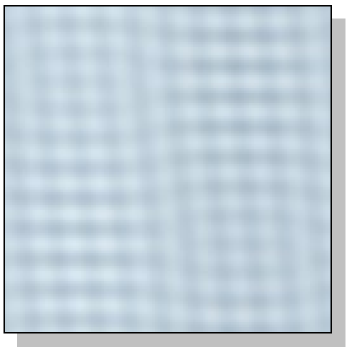
c) Hardware

Figure B15: Hardware Frequency Tests: Rock

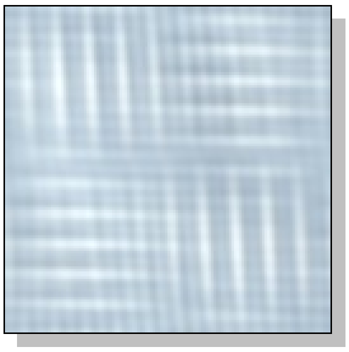

a) Original

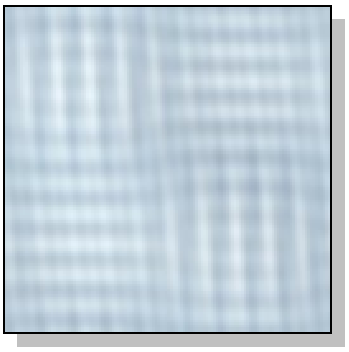

b1) 50 frequencies

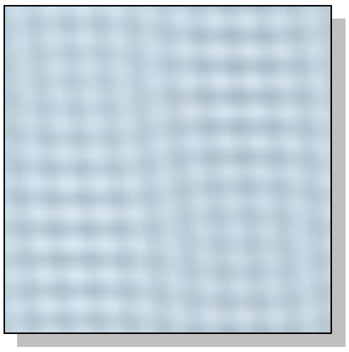

b2) 25 frequencies


b3) 10 frequencies

b) Software

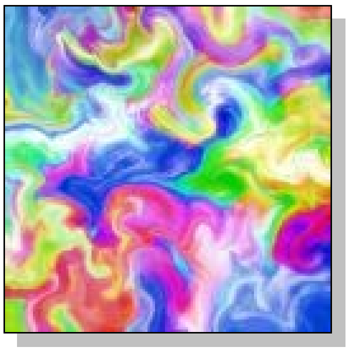

c1) 50 frequencies

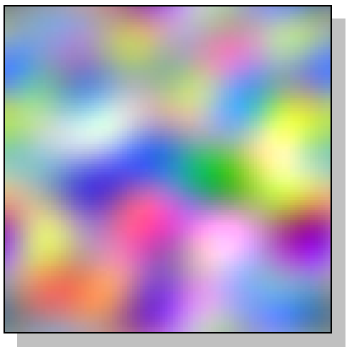

c2) 25 frequencies

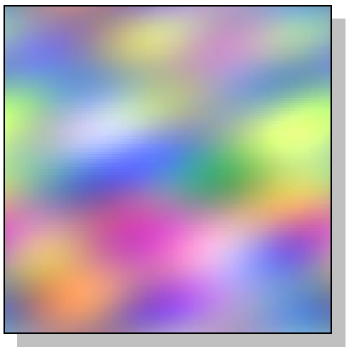

c3) 10 frequencies

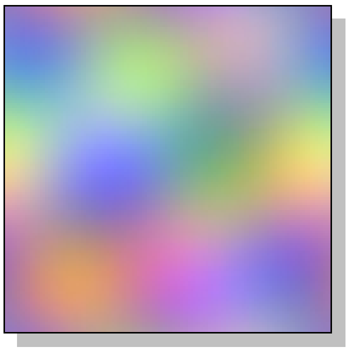
c) Hardware

Figure B16: Hardware Frequency Tests: Metal Floor

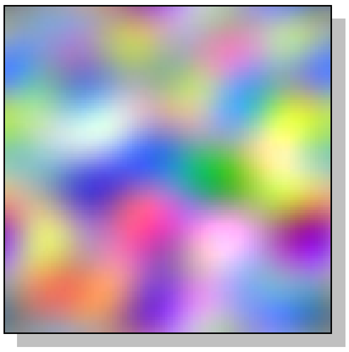

a) Original

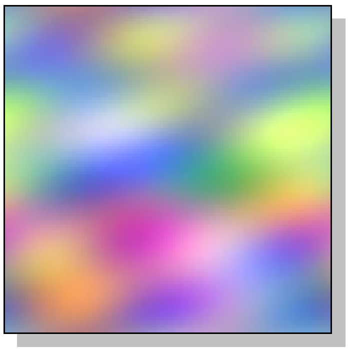

b1) 50 frequencies

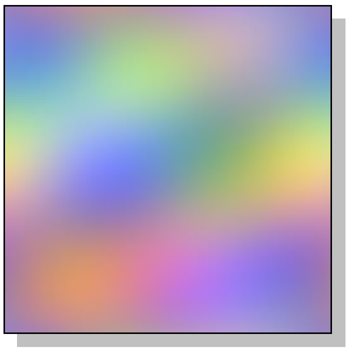

b2) 25 frequencies


b3) 10 frequencies

b) Software

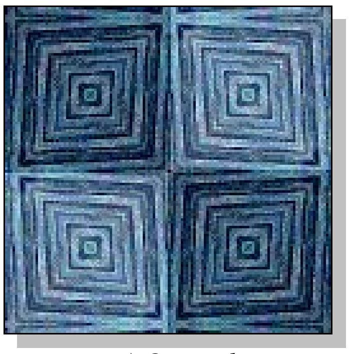

c1) 50 frequencies

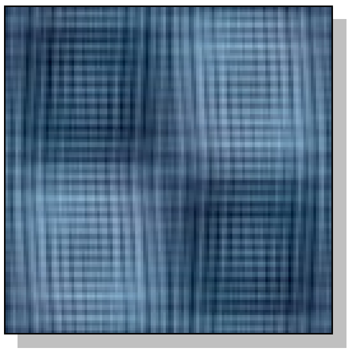

c2) 25 frequencies

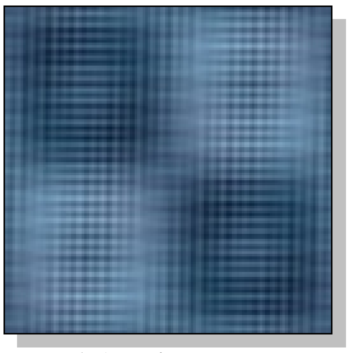

c3) 10 frequencies

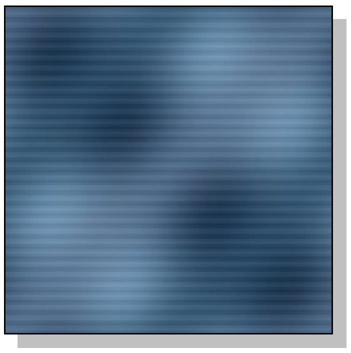
c) Hardware

Figure B17: Hardware Frequency Tests: Colourful Swirl

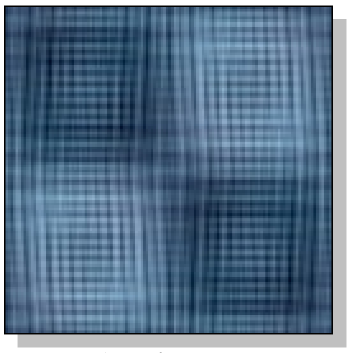

a) Original

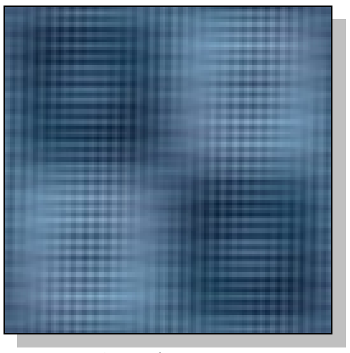

b1) 50 frequencies

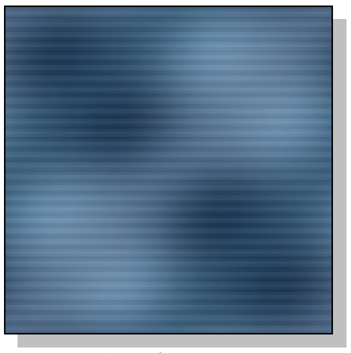

b2) 25 frequencies


b3) 10 frequencies

b) Software

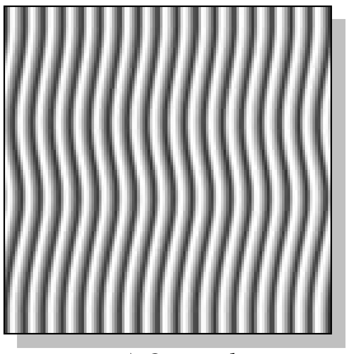

c1) 50 frequencies

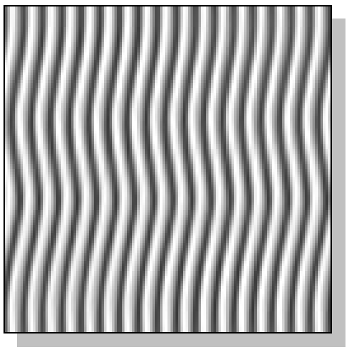

c2) 25 frequencies

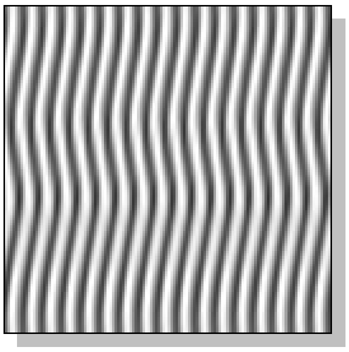

c3) 10 frequencies

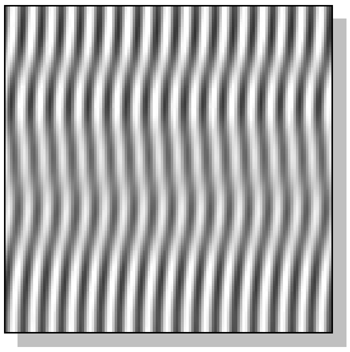
c) Hardware

Figure B18: Hardware Frequency Tests: Tiles

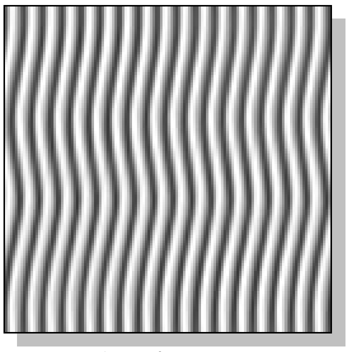

a) Original

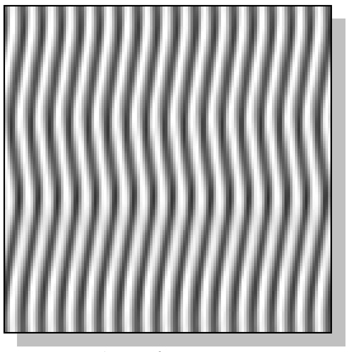

b1) 50 frequencies

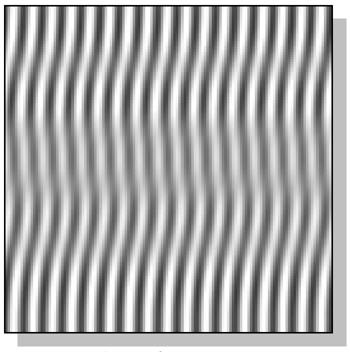

b2) 25 frequencies


b3) 10 frequencies

b) Software

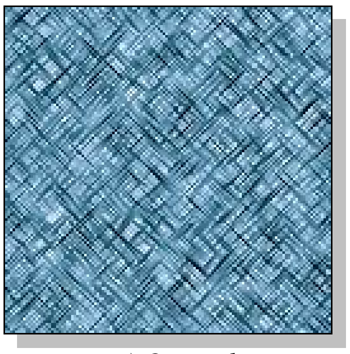

c1) 50 frequencies

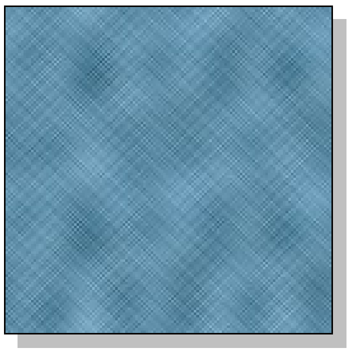

c2) 25 frequencies

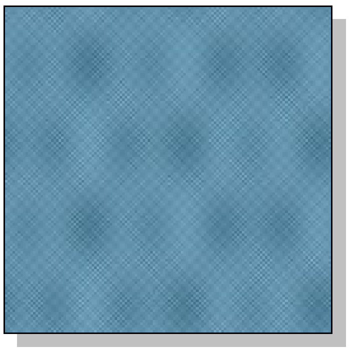

c3) 10 frequencies

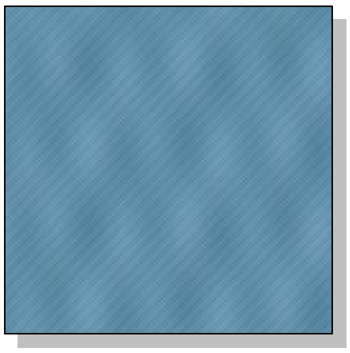
c) Hardware

Figure B19: Hardware Frequency Tests: Stripes

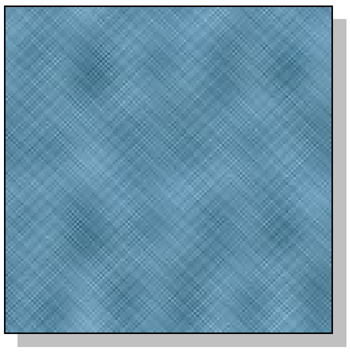

a) Original

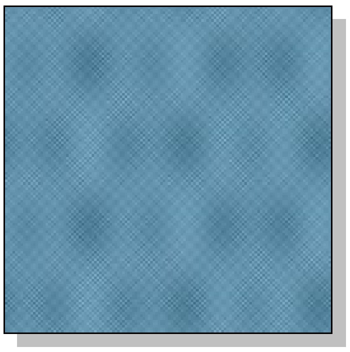

b1) 50 frequencies

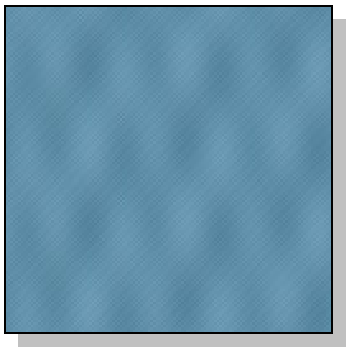

b2) 25 frequencies


b3) 10 frequencies

b) Software


c1) 50 frequencies

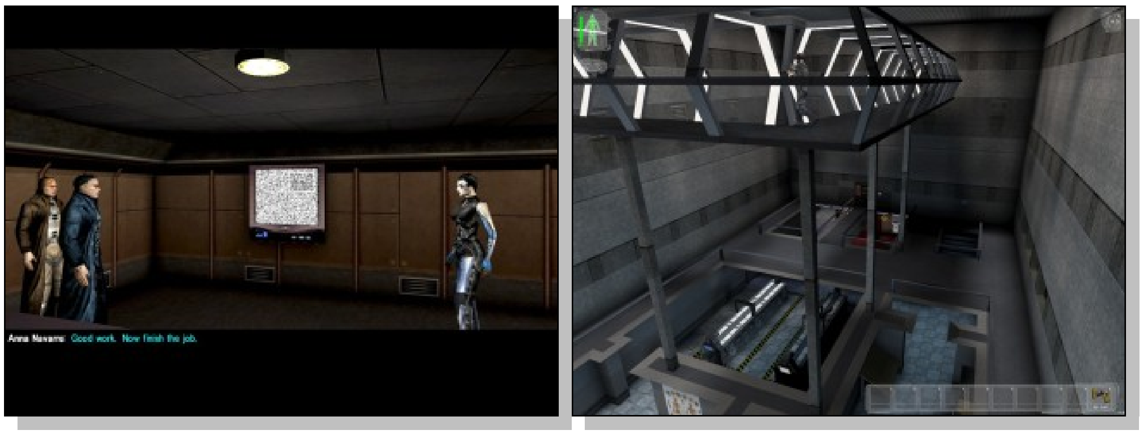

c2) 25 frequencies


c3) 10 frequencies

c) Hardware

Figure B20: Hardware Frequency Tests: Ice

Appendix C: Typical Video Game Scenes



Anna Navarre: Good work. Now finish the job.

Figure C1: Deus Ex, 2000

Figure C2: Morrowind, 2002

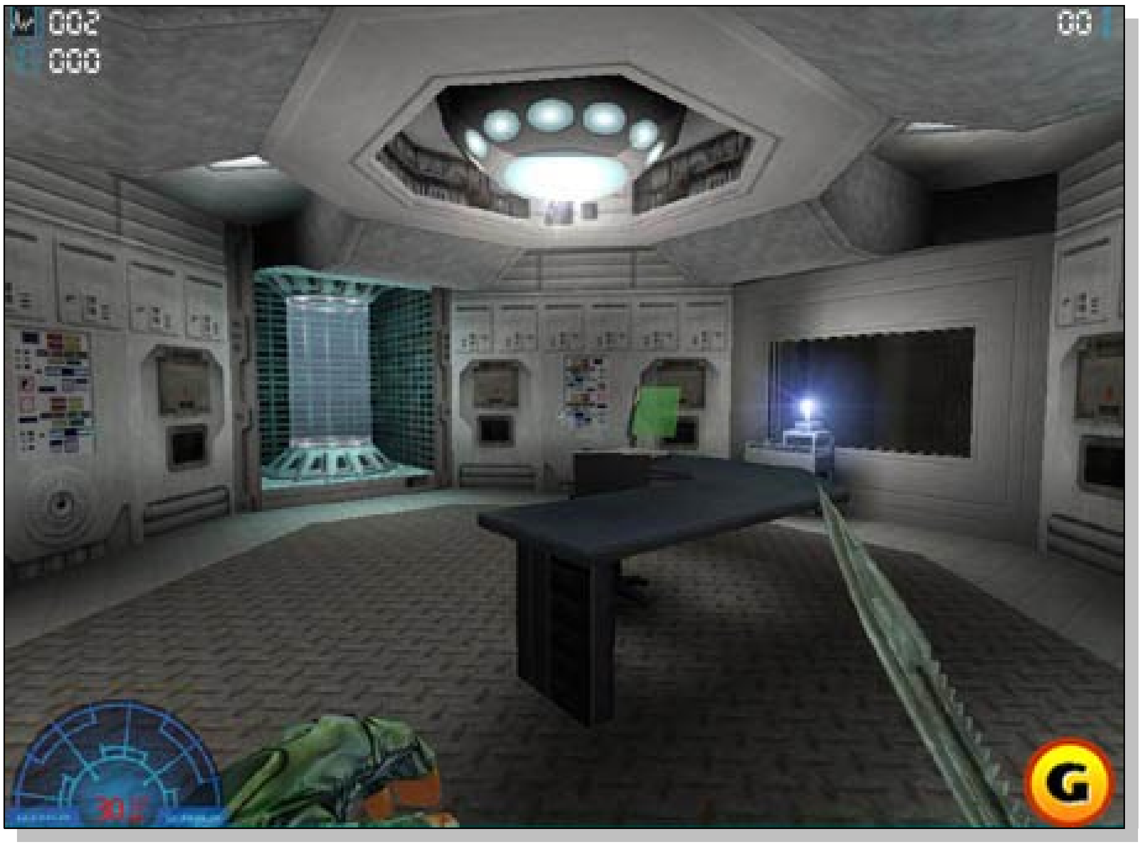


Figure C3: Alien Vs Predator 2, 2001

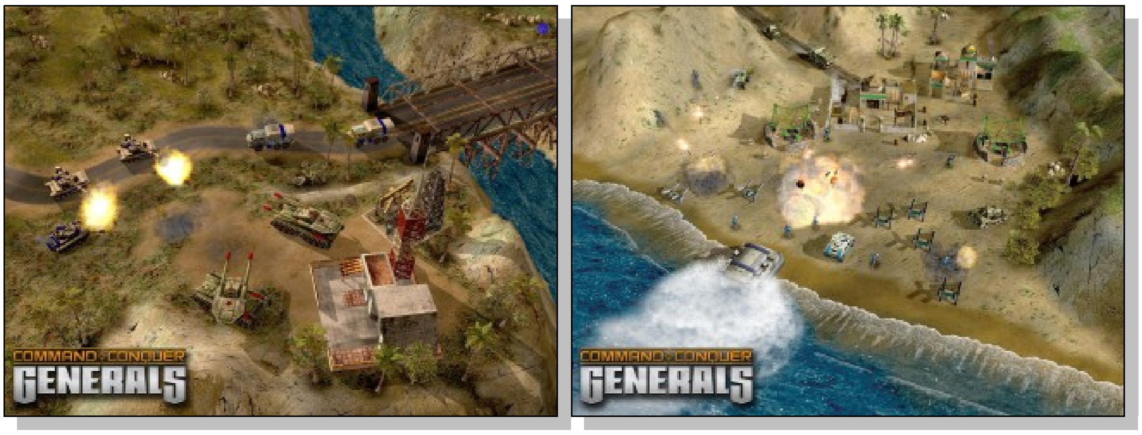


Figure C4: Command and Conquer: Generals, 2003

Figure C5: Half-Life 2, 2004

Figure C6: Civilization IV, 2005

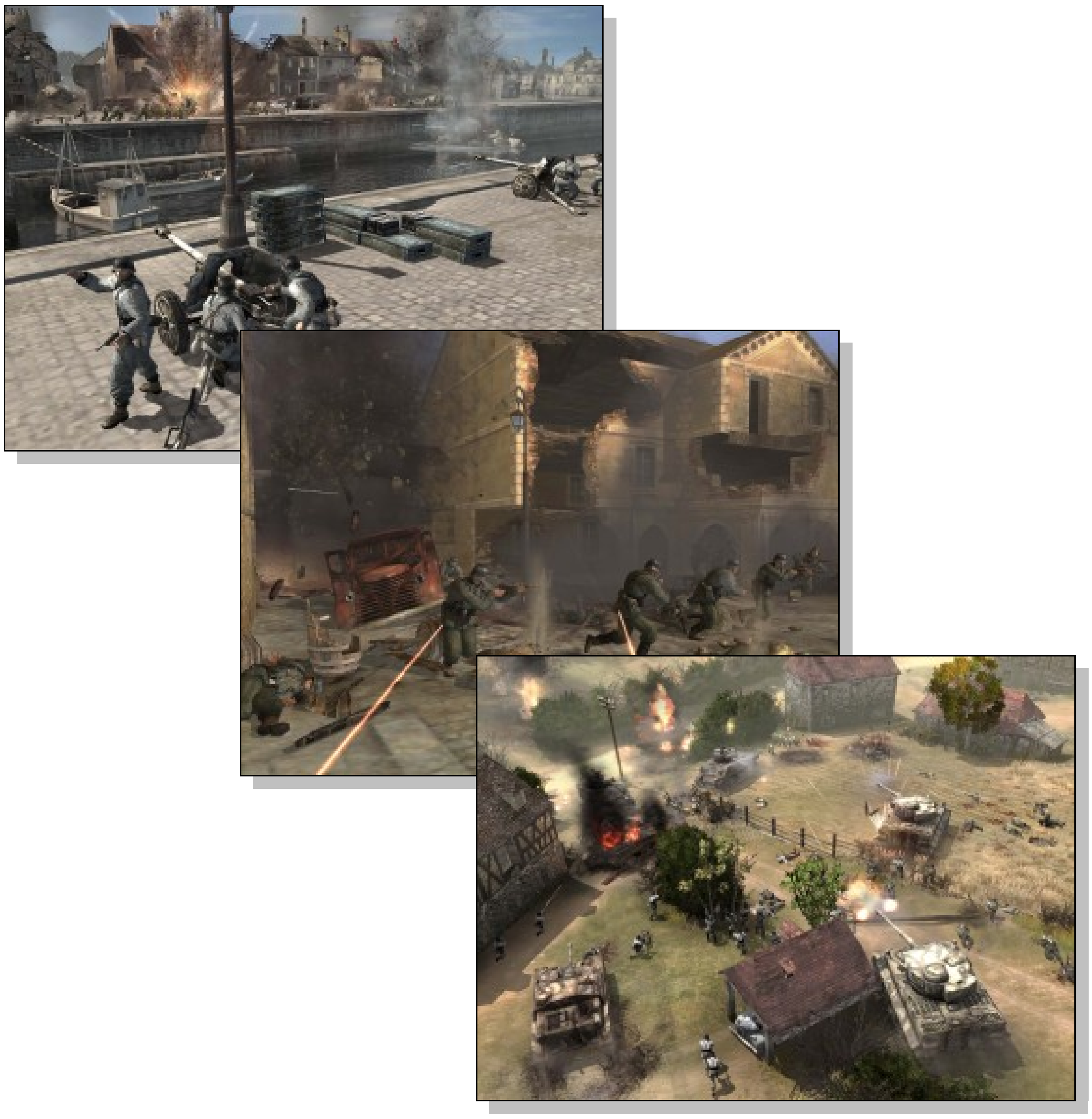


Figure C7: Company of Heroes, 2006

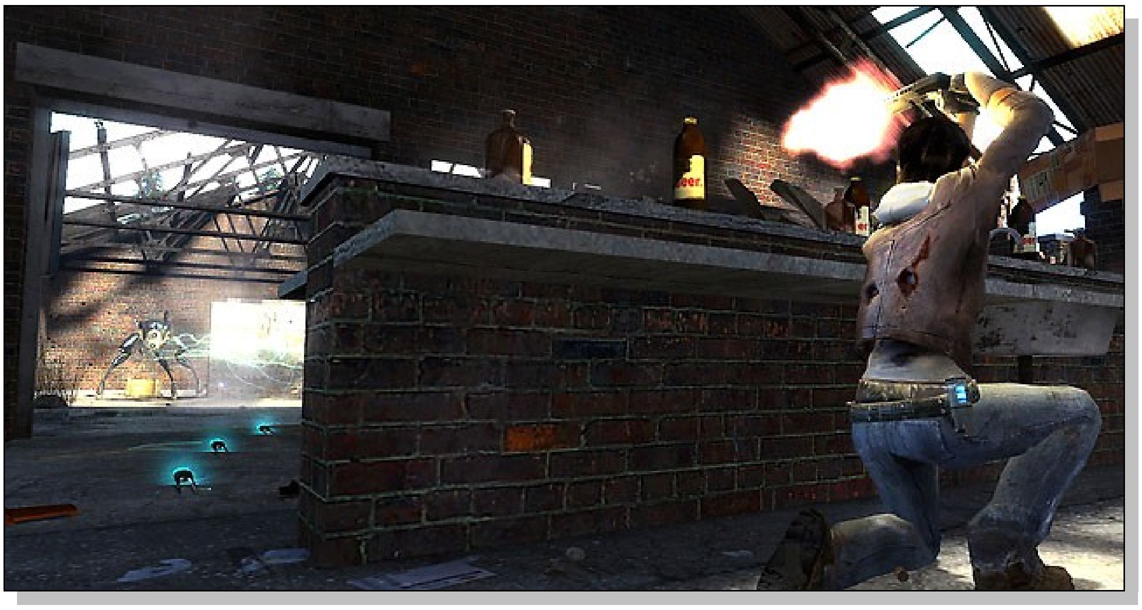
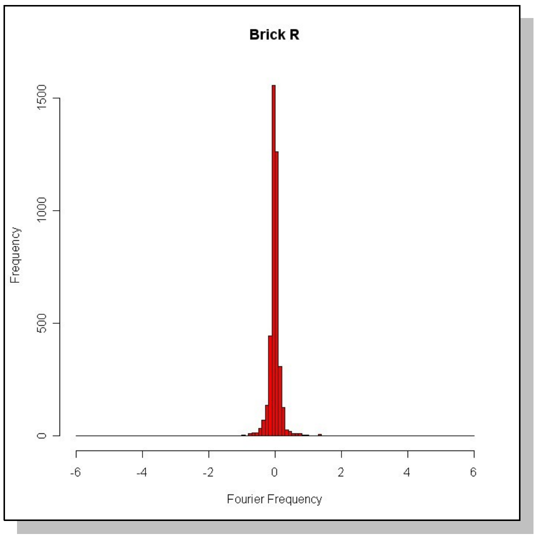
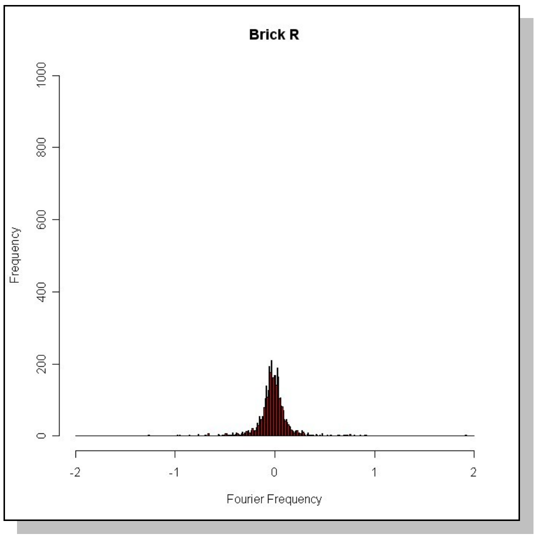


Figure C8: Half-Life 2: Episode Two, 2007

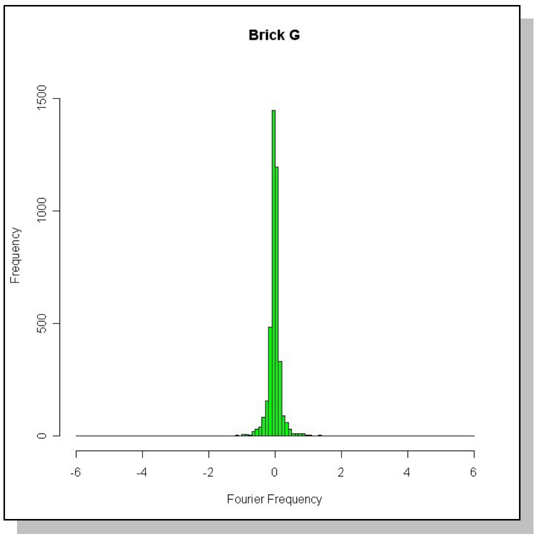
Appendix D: Histograms



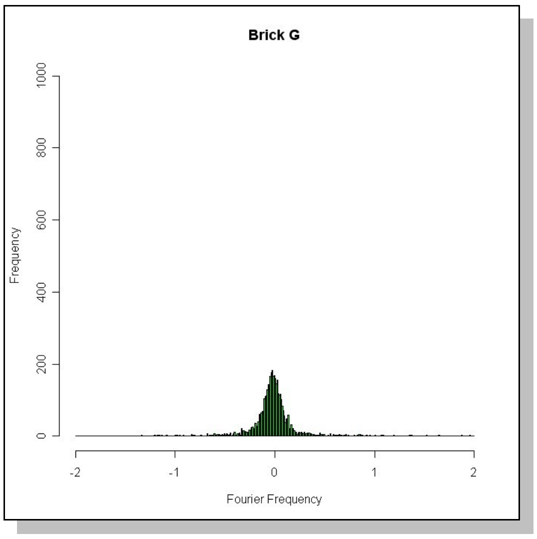
a) Brick red channel full range



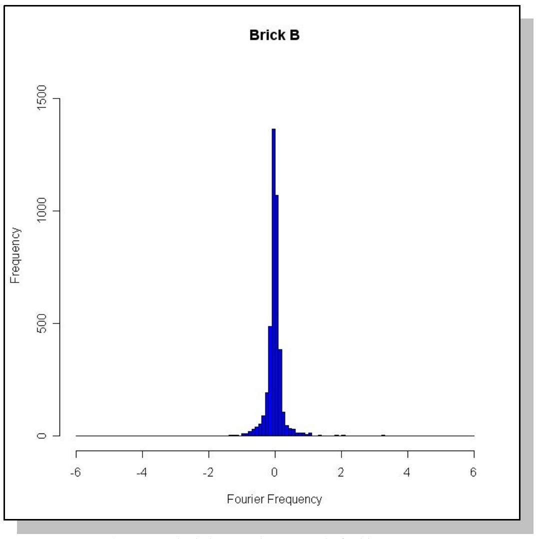
b) Brick red channel range limited



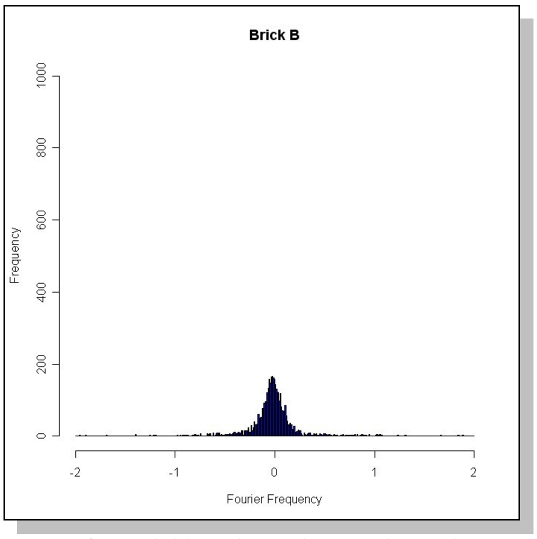
c) Brick green channel full range



d) Brick green channel range limited

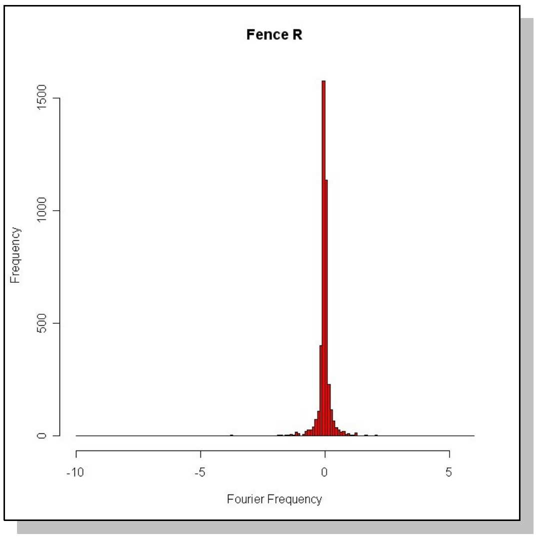


e) Brick blue channel full range

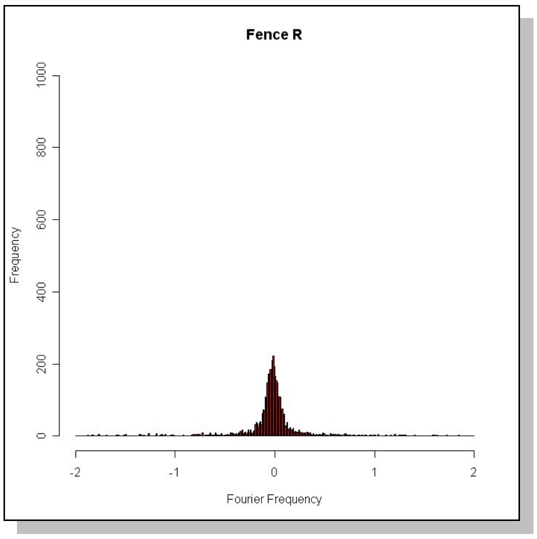


f) Brick blue channel range limited

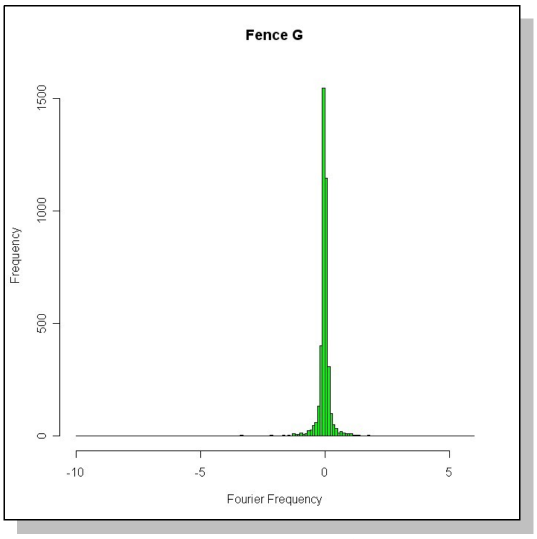
Figure D1: Brick Histograms



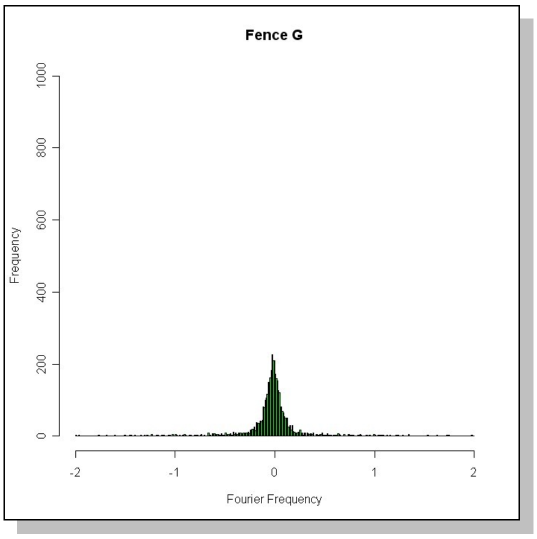
a) Fence red channel full range



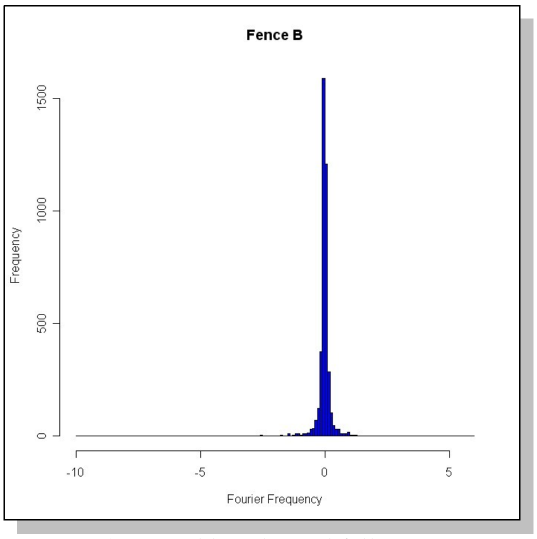
b) Fence red channel range limited



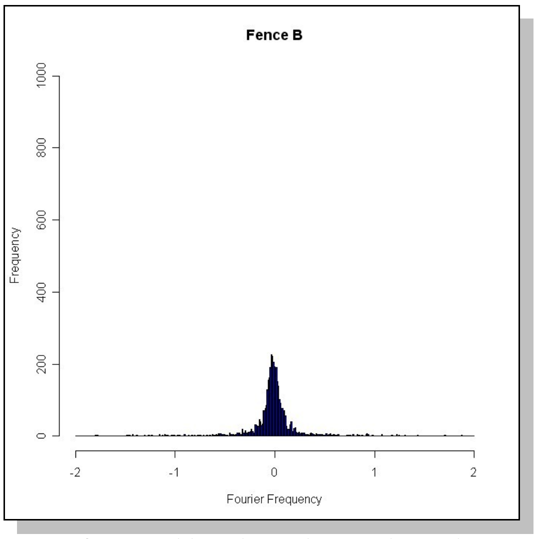
c) Fence green channel full range



d) Fence green channel range limited

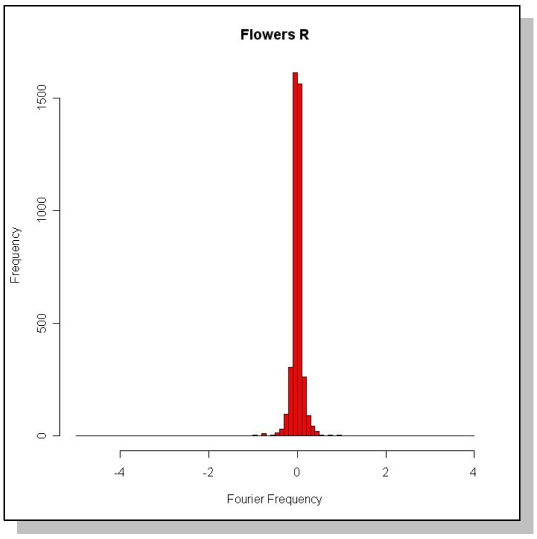


e) Fence blue channel full range

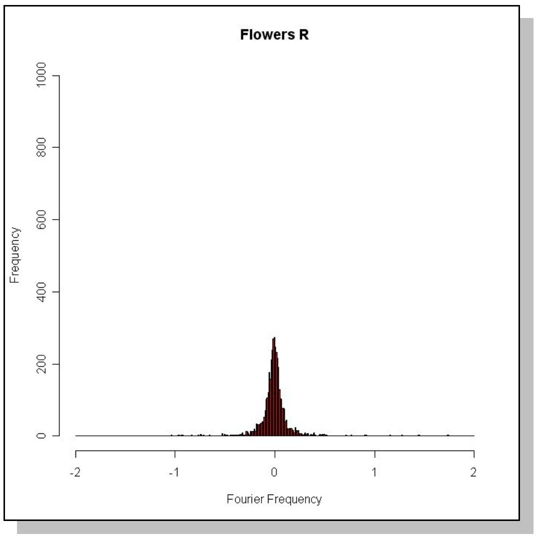


f) Fence blue channel range limited

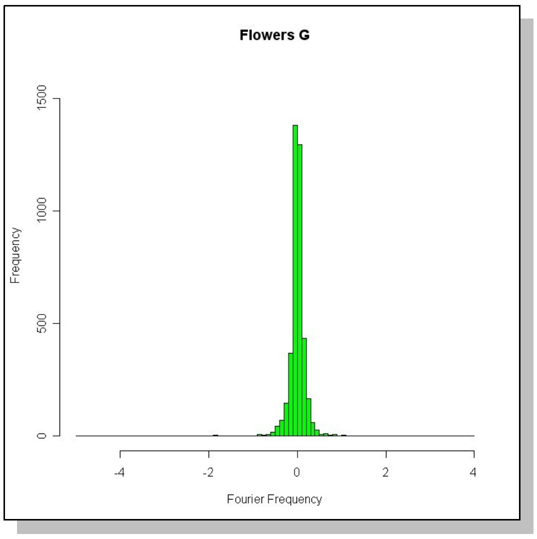
Figure D2: Fence Histograms



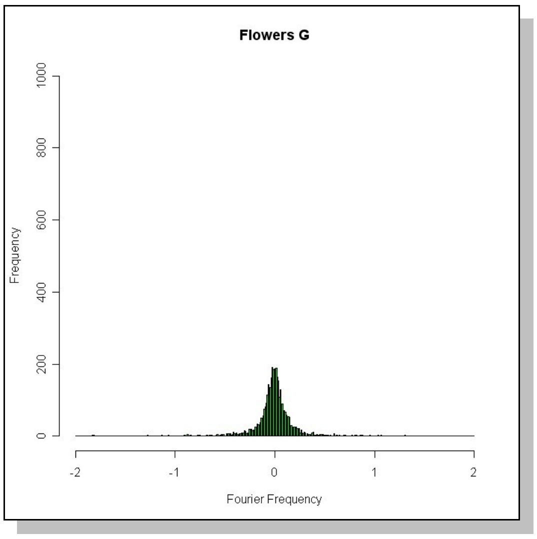
a) Flowers red channel full range



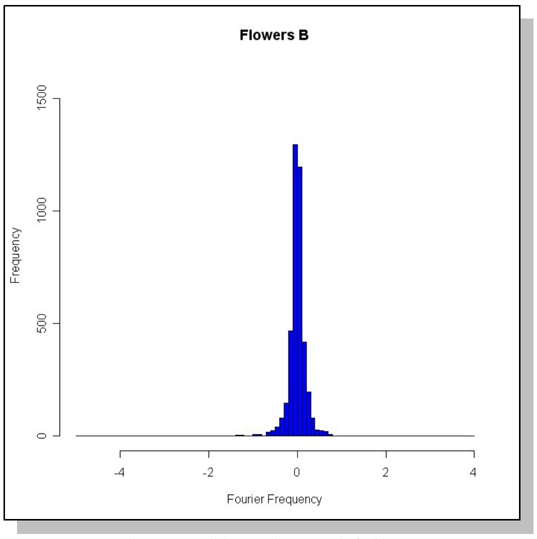
b) Flowers red channel range limited



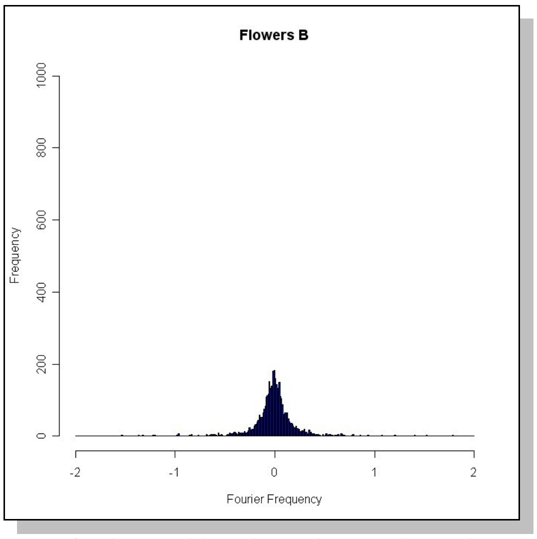
c) Flowers green channel full range



d) Flowers green channel range limited

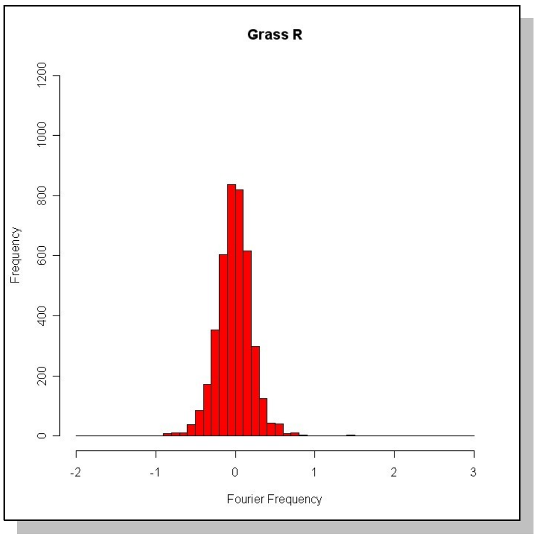


e) Flowers blue channel full range

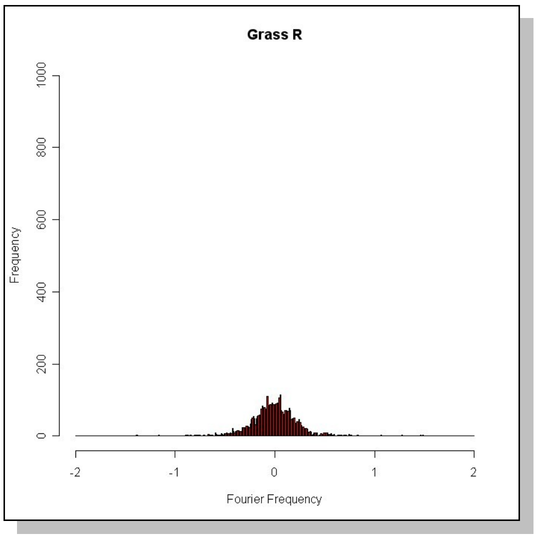


f) Flowers blue channel range limited

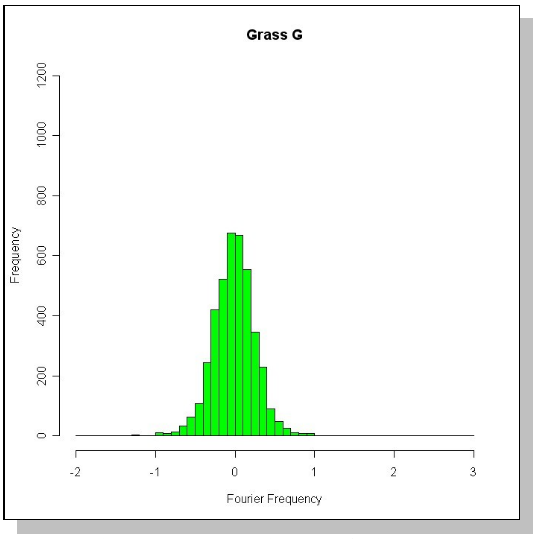
Figure D3: Flowers Histograms



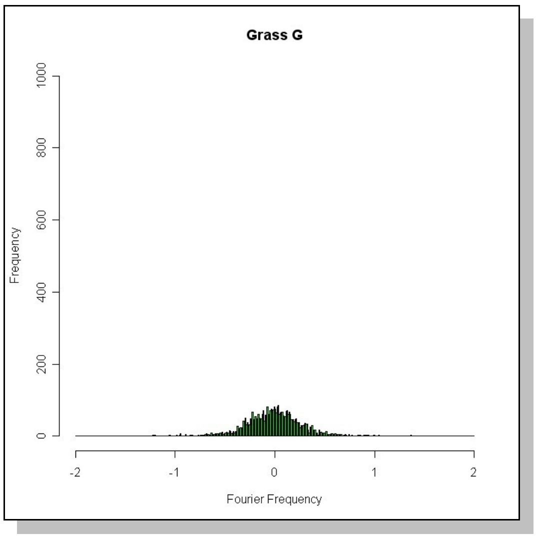
a) Grass red channel full range



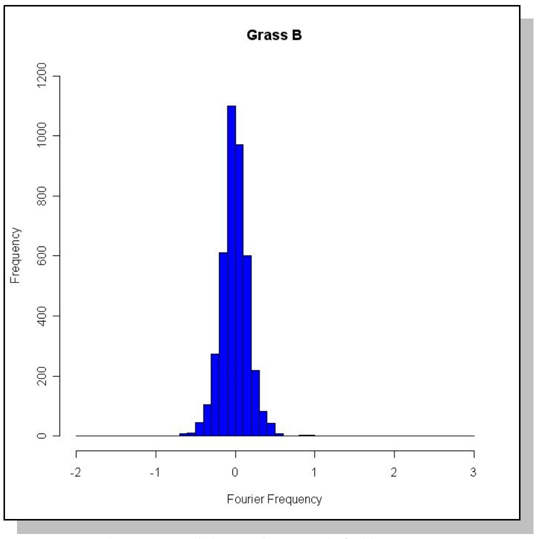
b) Grass red channel range limited



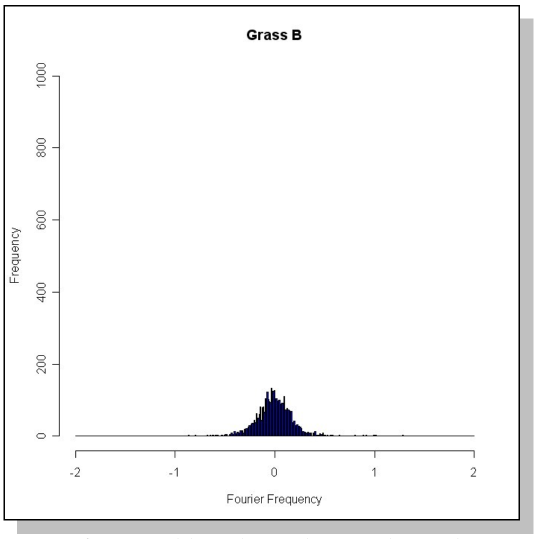
c) Grass green channel full range



d) Grass green channel range limited

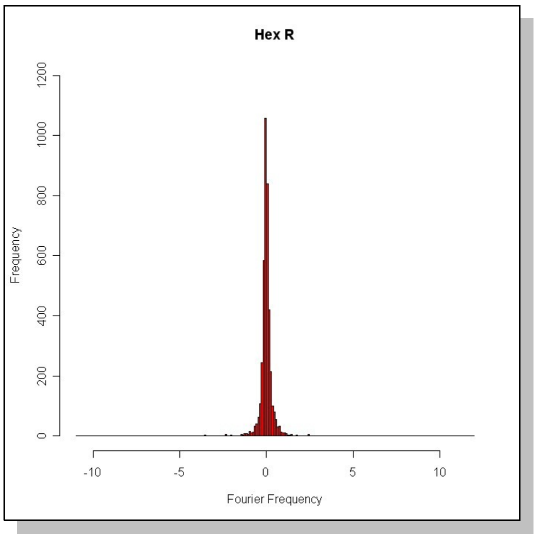


e) Grass blue channel full range

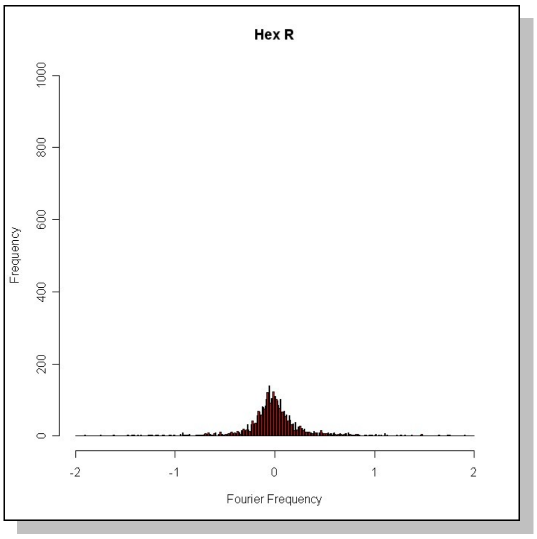


f) Grass blue channel range limited

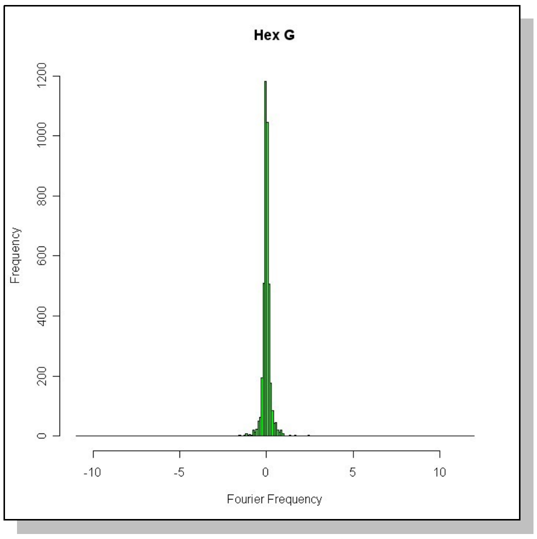
Figure D4: Grass Histograms



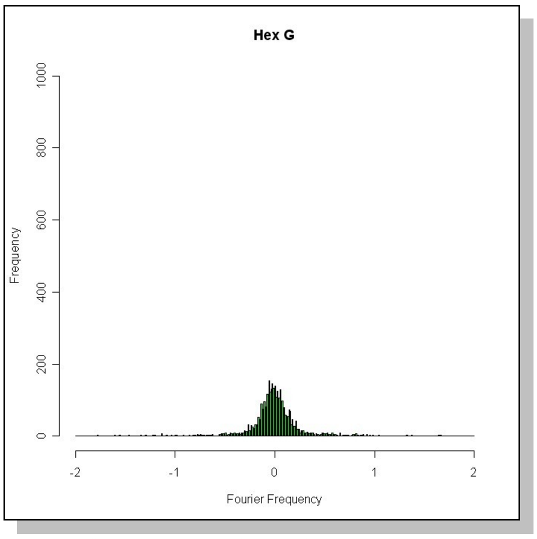
a) Hex red channel full range



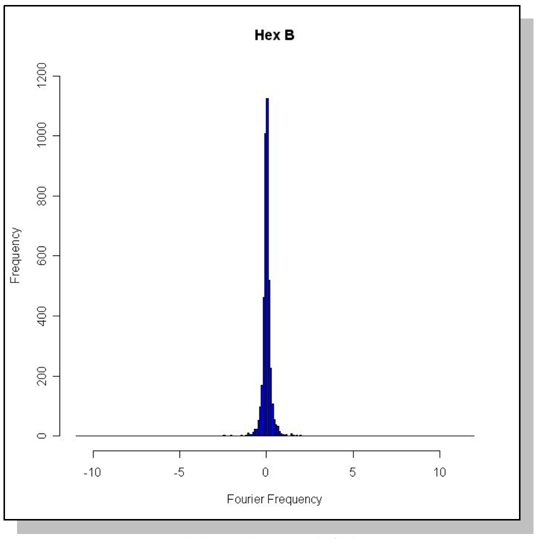
b) Hex red channel range limited



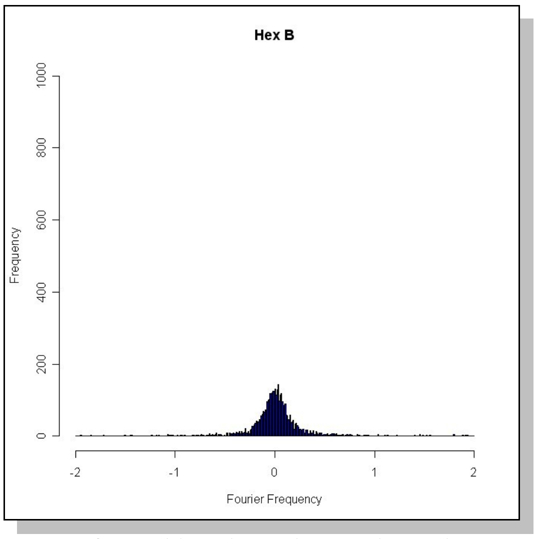
c) Hex green channel full range



d) Hex green channel range limited

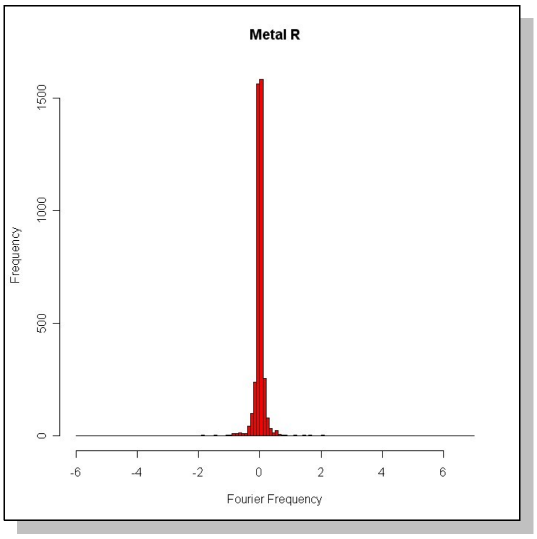


e) Hex blue channel full range

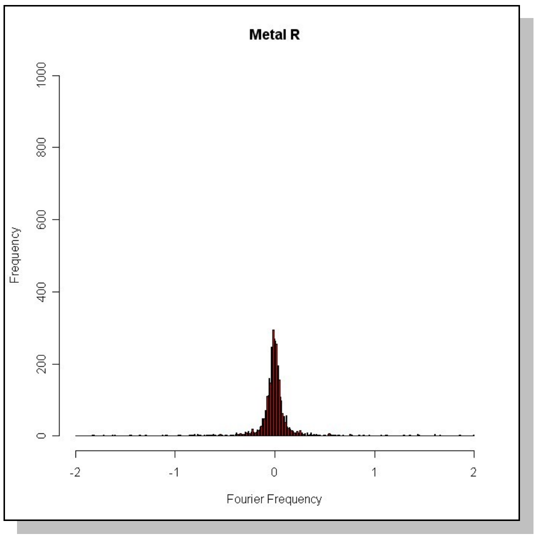


f) Hex blue channel range limited

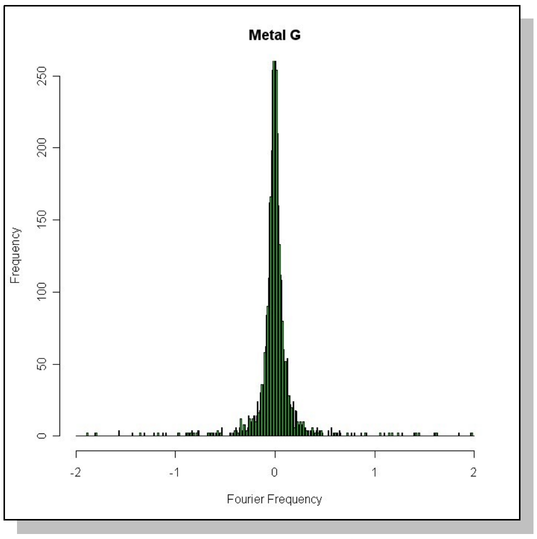
Figure D5: Hex Histograms



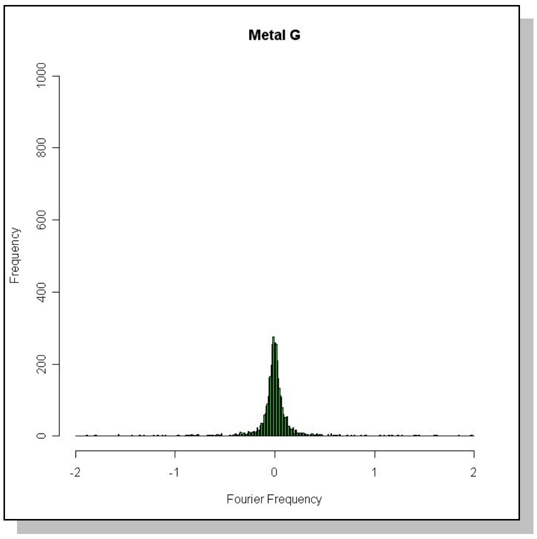
a) Metal red channel full range



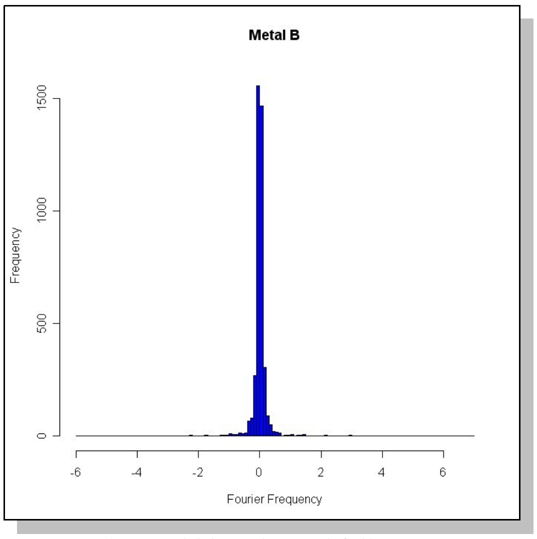
b) Metal red channel range limited



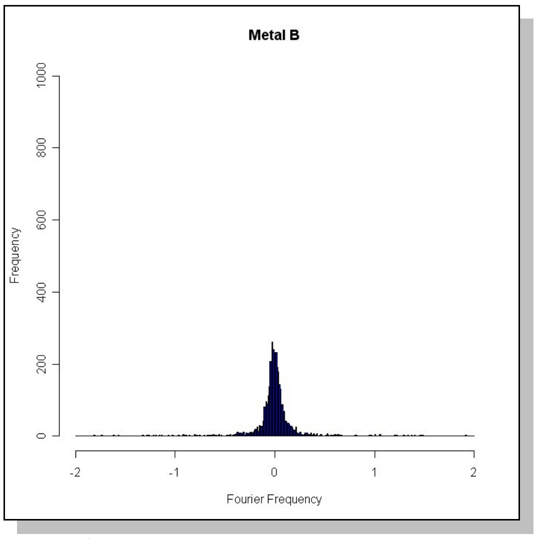
c) Metal green channel full range



d) Metal green channel range limited

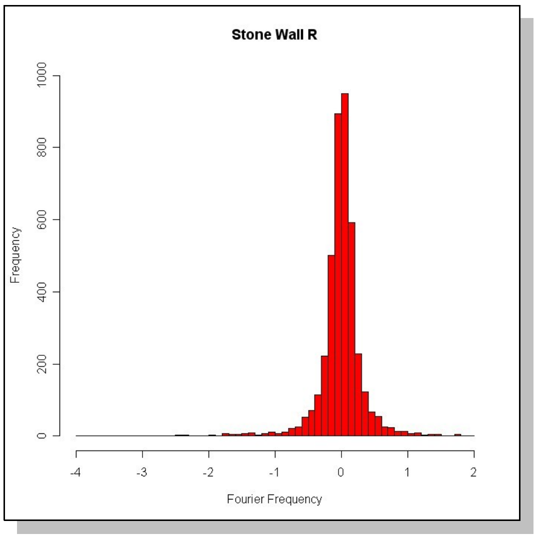


e) Metal blue channel full range

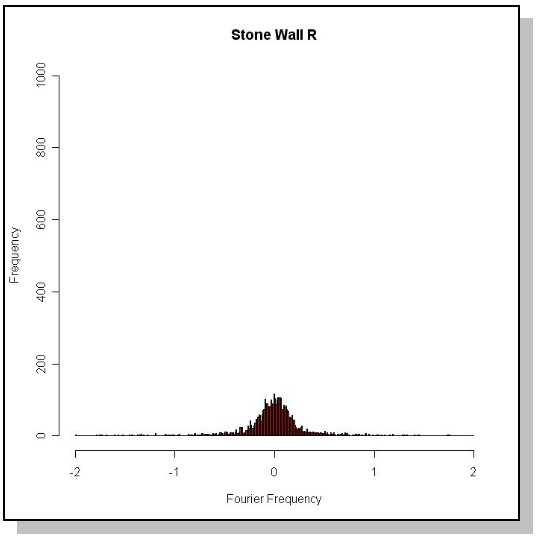


f) Metal blue channel range limited

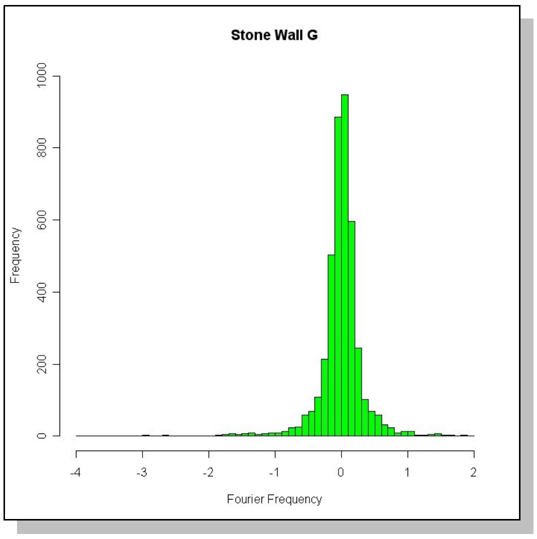
Figure D6: Metal Histograms



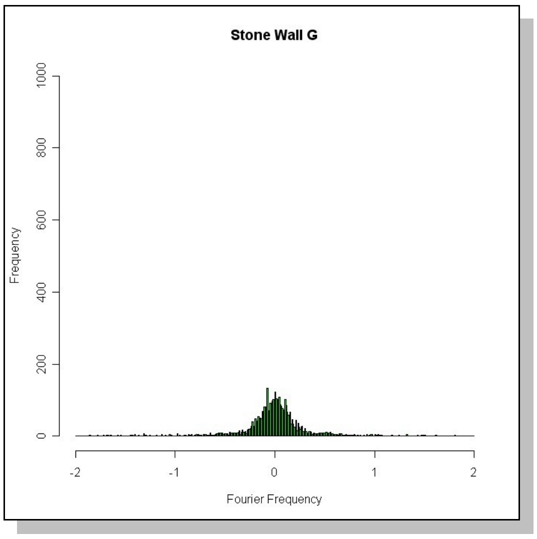
a) Stone Wall red channel full range



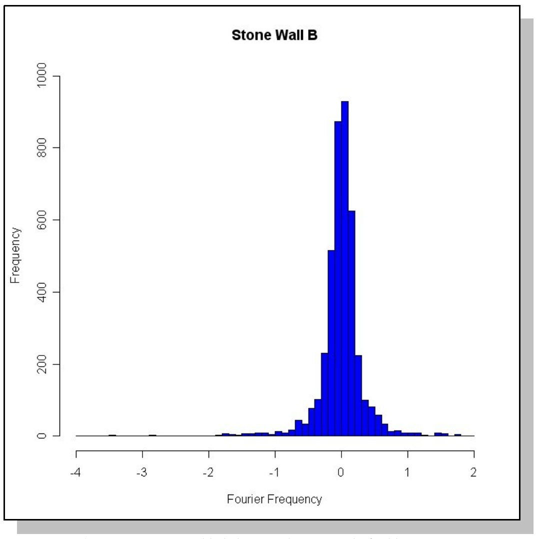
b) Stone Wall red channel range limited



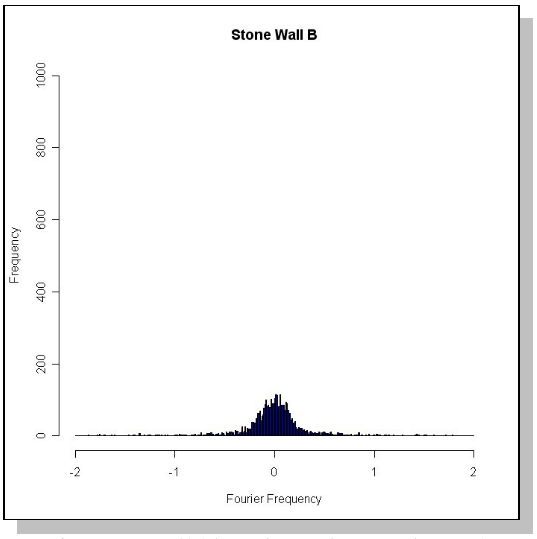
c) Stone Wall green channel full range



d) Stone Wall green channel range limited

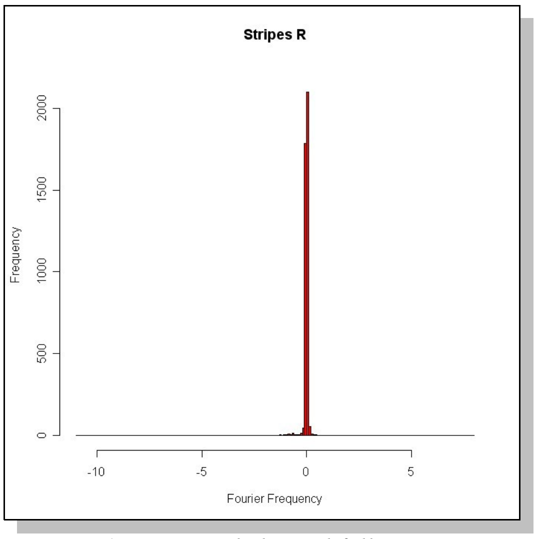


e) Stone Wall blue channel full range

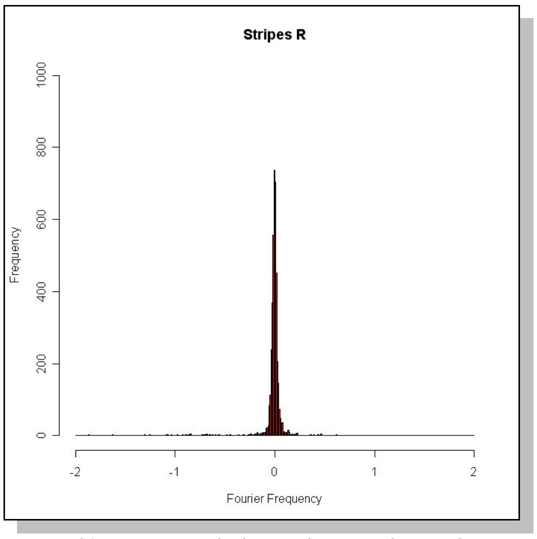


f) Stone Wall blue channel range limited

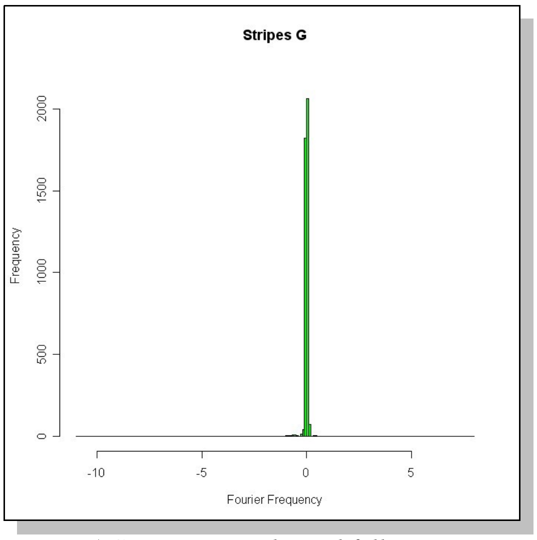
Figure D7: Stone Wall Histograms



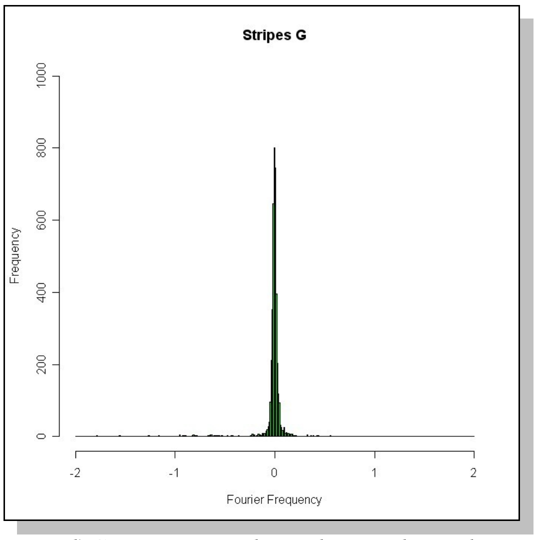
a) Stripes red channel full range



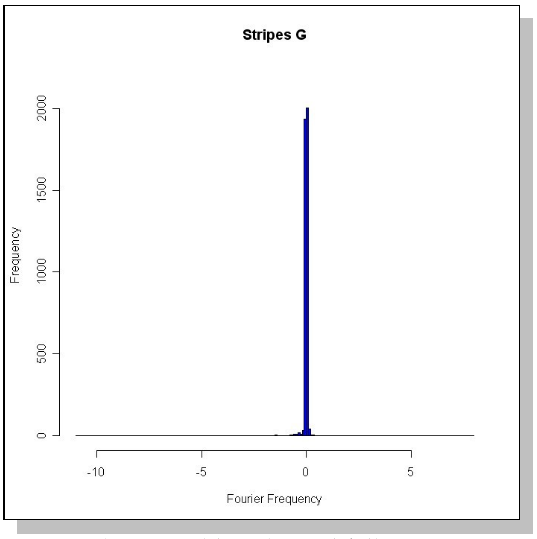
b) Stripes red channel range limited



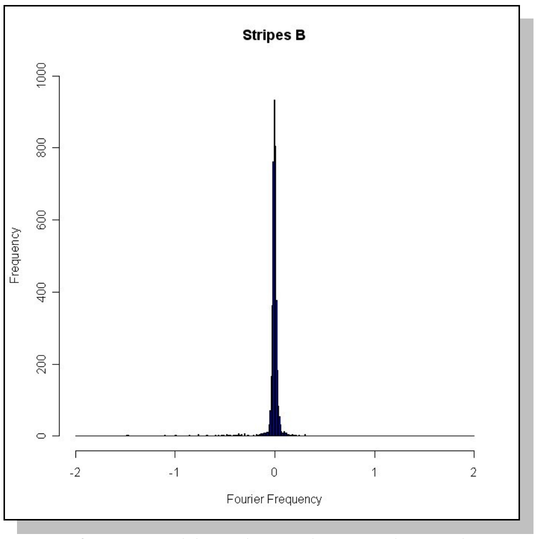
c) Stripes green channel full range



d) Stripes green channel range limited

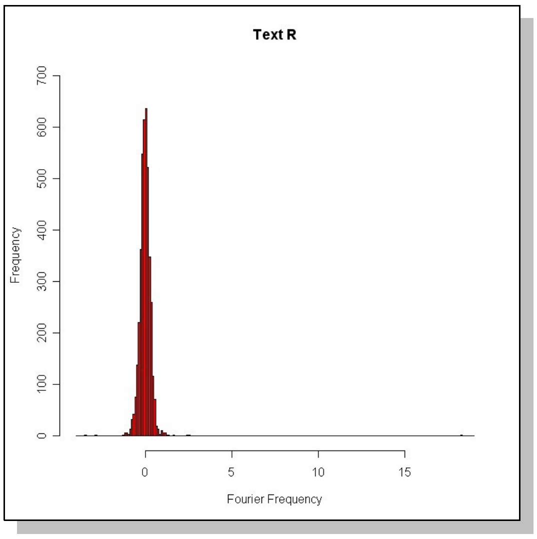


e) Stripes blue channel full range

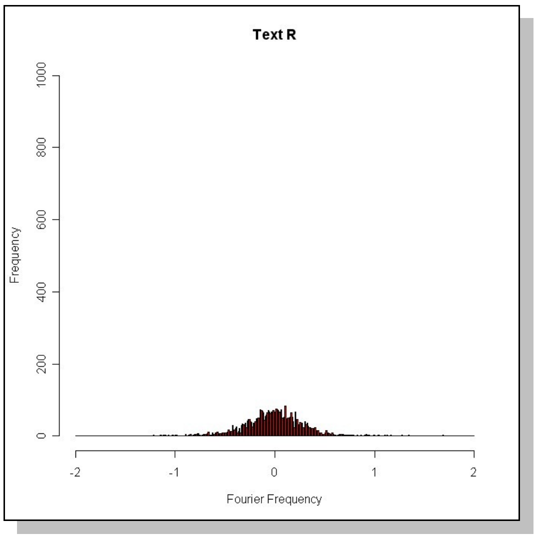


f) Stripes blue channel range limited

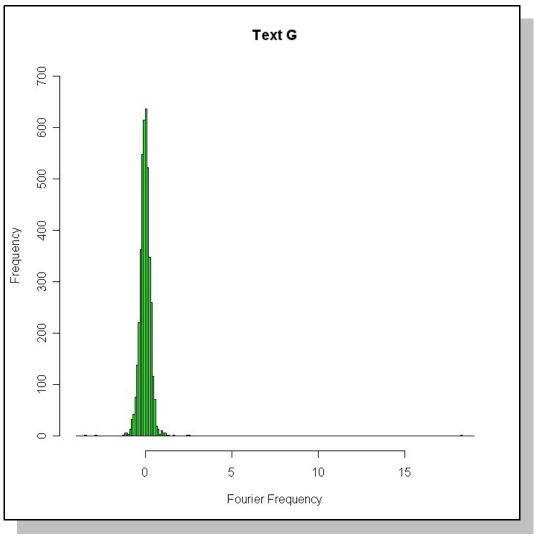
Figure D8: Stripes Histograms



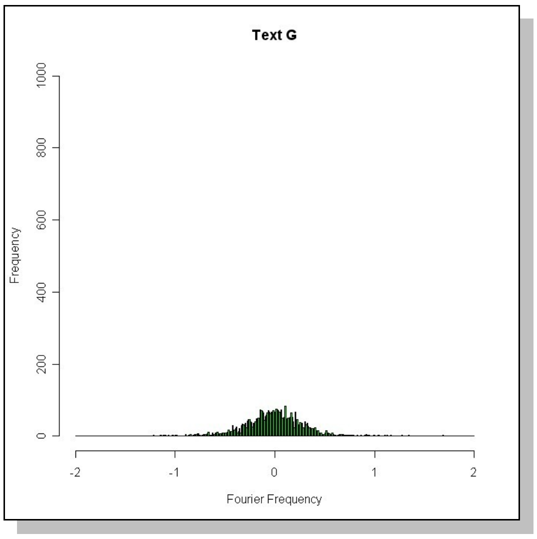
a) Text red channel full range



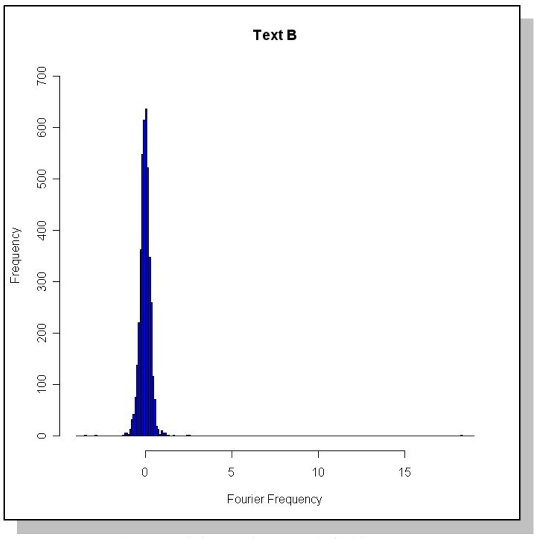
b) Text red channel range limited



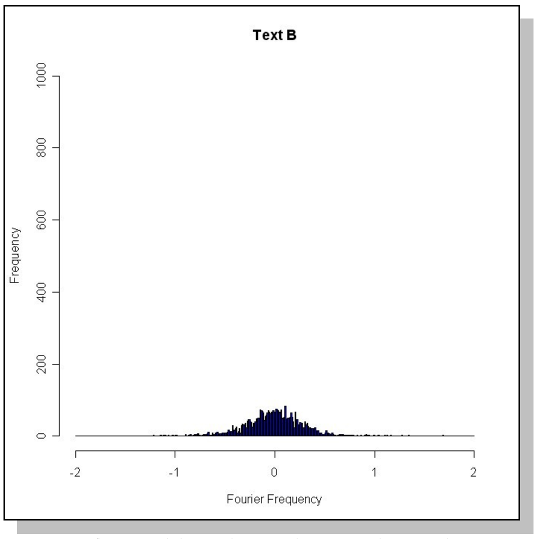
c) Text green channel full range



d) Text green channel range limited



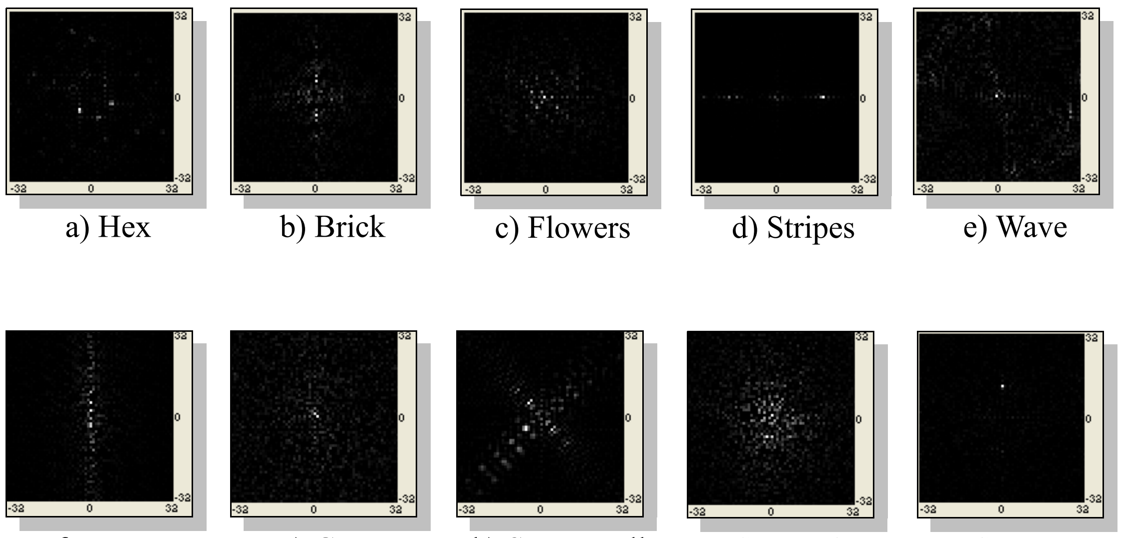
e) Text blue channel full range



f) Text blue channel range limited

Figure D9: Text Histograms

Appendix E: Textures in Fourier Space



a) Hex b) Brick c) Flowers d) Stripes e) Wave

f) Fence g) Grass h) Stone Wall i) Metal j) Text

Figure E1: Textures for Testing: Fourier profiles

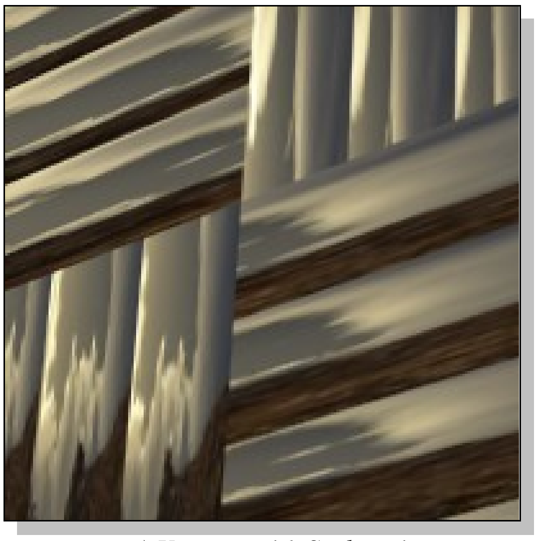
Appendix F: Super Bump Mapping Images

a) Vectors = 1, Scale = 1

b) Vectors = 2, Scale = 1

c) Vectors = 4, Scale = 1

c) Vectors = 8, Scale = 1


e) Vectors = 16, Scale = 1

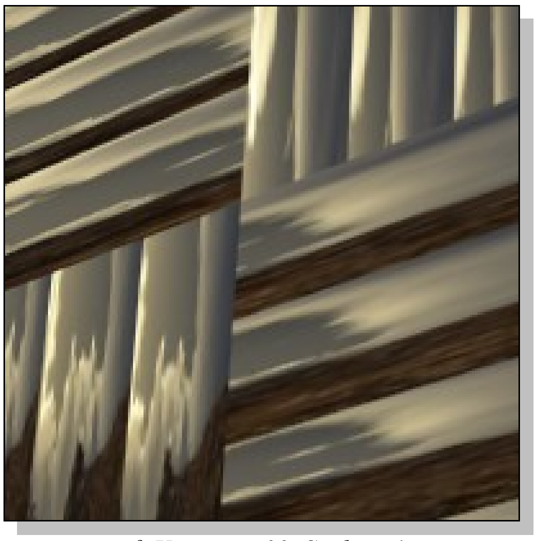
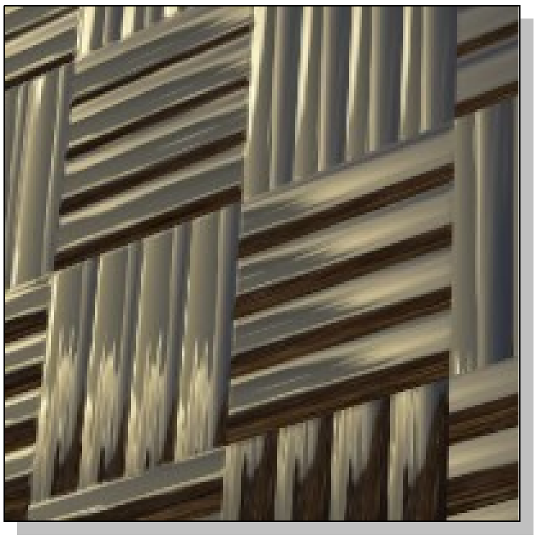
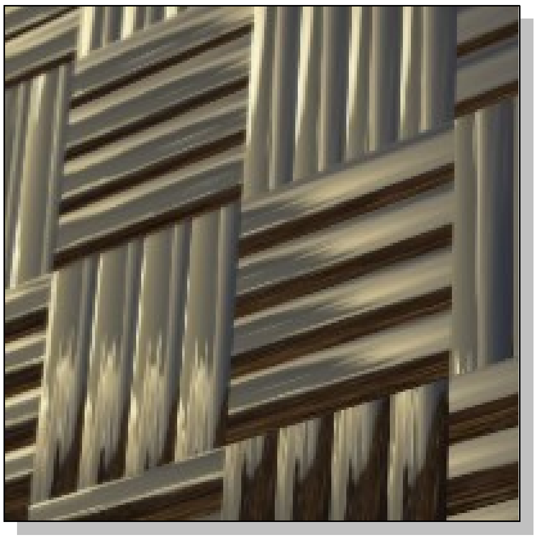
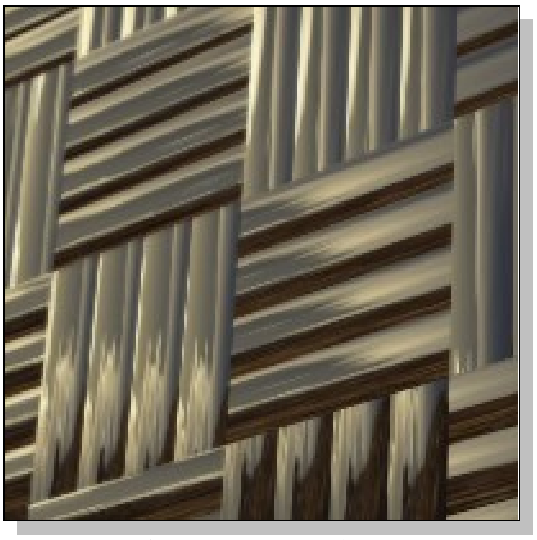
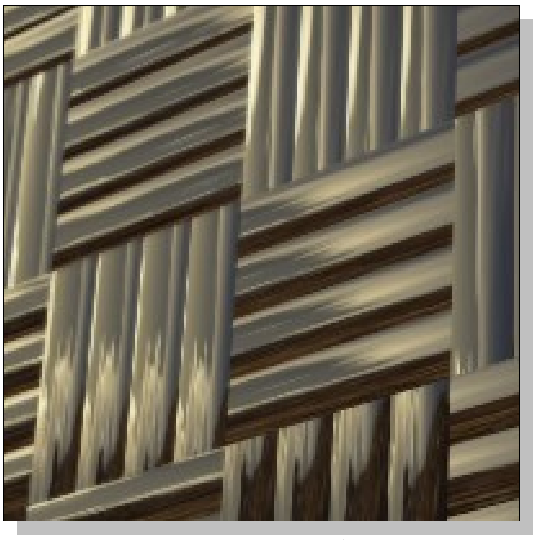

f) Vectors = 32, Scale = 1

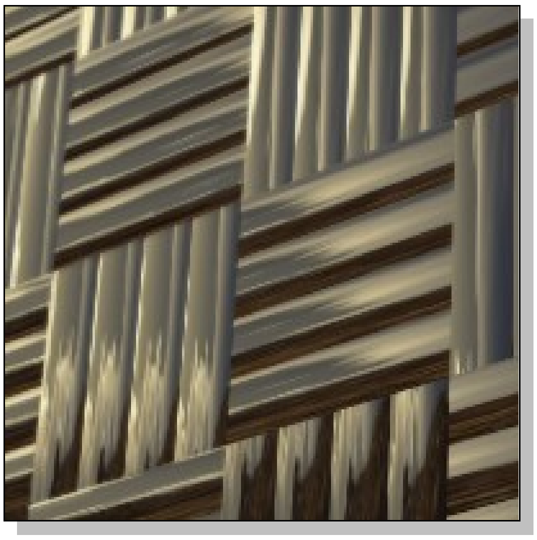
Figure F1: Grid normal map


a) Vectors = 1, Scale = 2


b) Vectors = 2, Scale = 2


c) Vectors = 4, Scale = 2


c) Vectors = 8, Scale = 2


e) Vectors = 16, Scale = 2

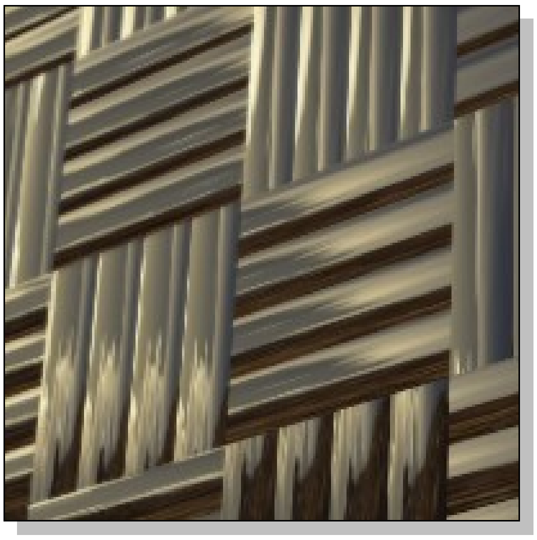
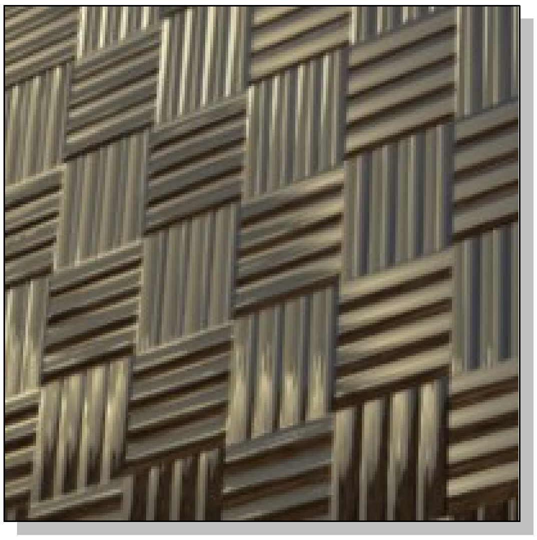
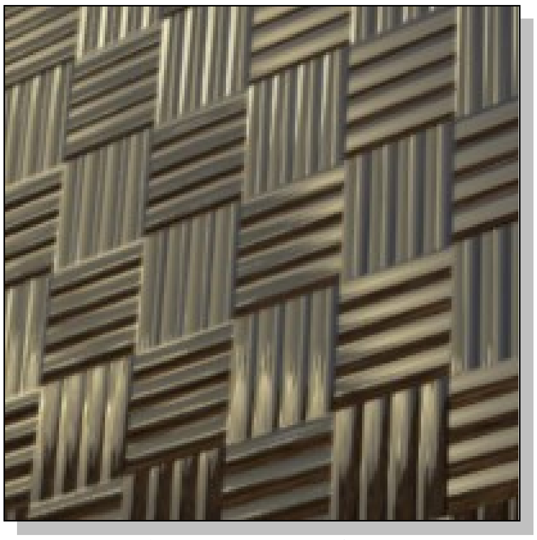
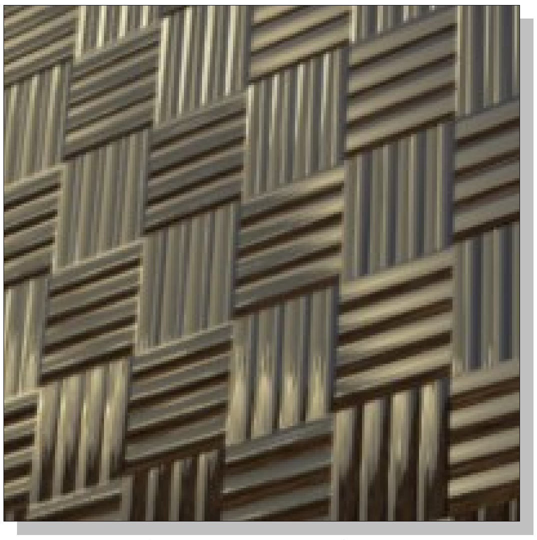

f) Vectors = 32, Scale = 2

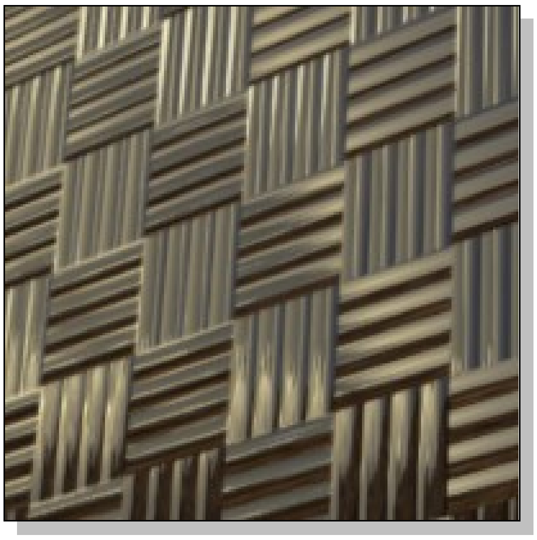
Figure F2: Grid normal map

a) Vectors = 1, Scale = 4


b) Vectors = 2, Scale = 4


c) Vectors = 4, Scale = 4


c) Vectors = 8, Scale = 4


e) Vectors = 16, Scale = 4

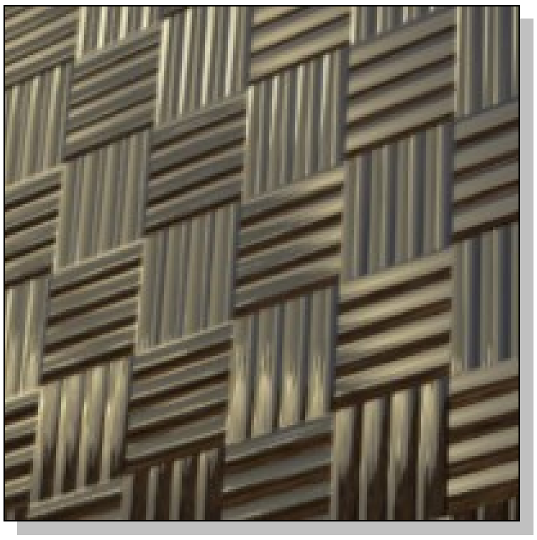
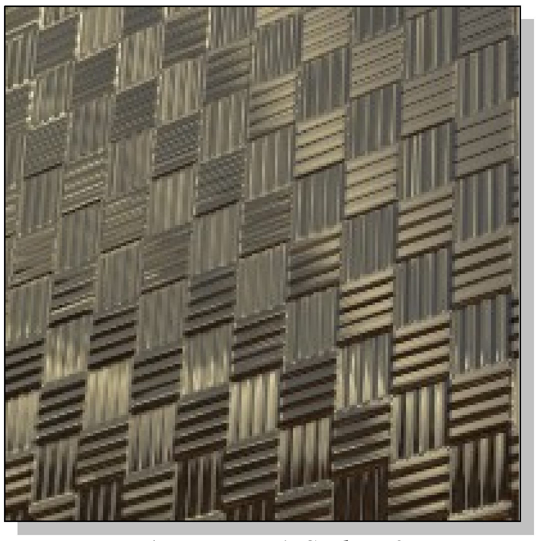
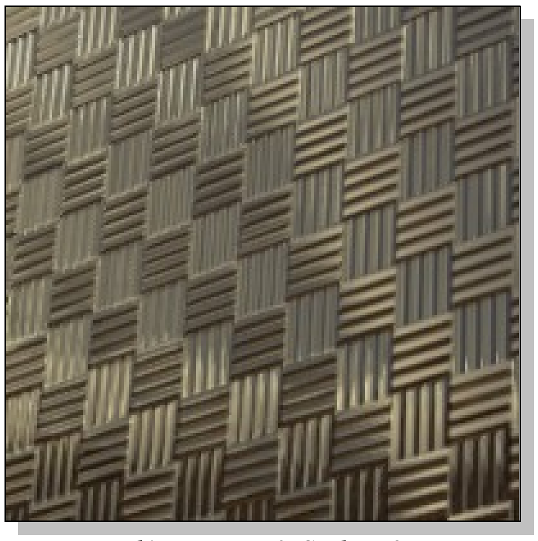
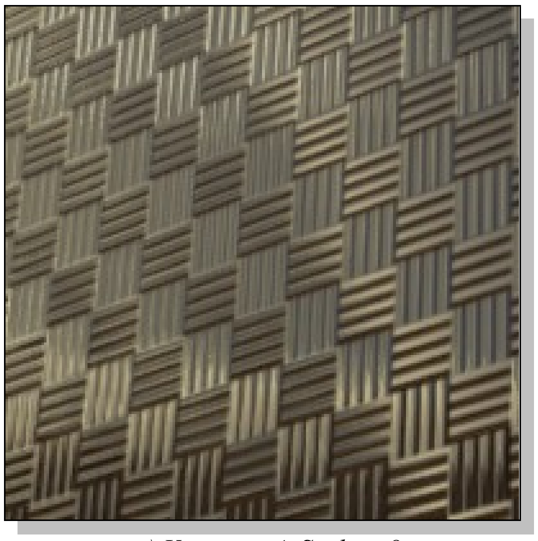
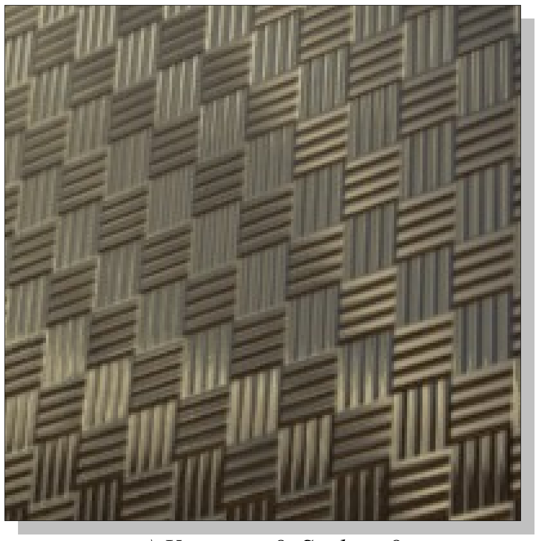

f) Vectors = 32, Scale = 4

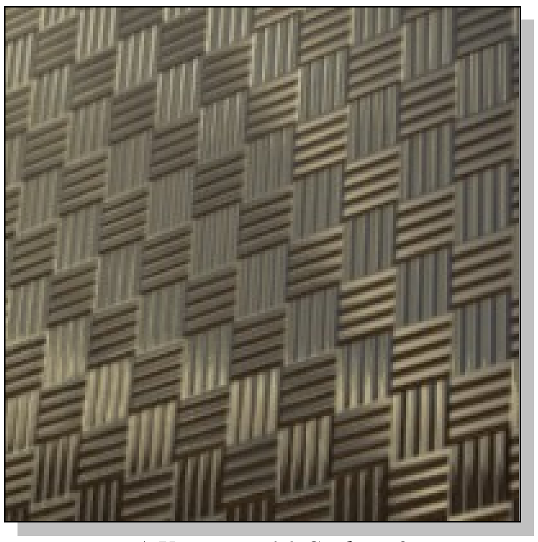
Figure F3: Grid normal map


a) Vectors = 1, Scale = 8


b) Vectors = 2, Scale = 8


c) Vectors = 4, Scale = 8


c) Vectors = 8, Scale = 8


e) Vectors = 16, Scale = 8

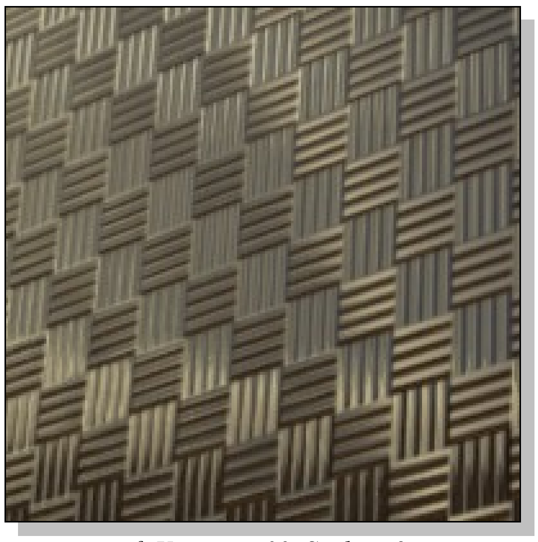
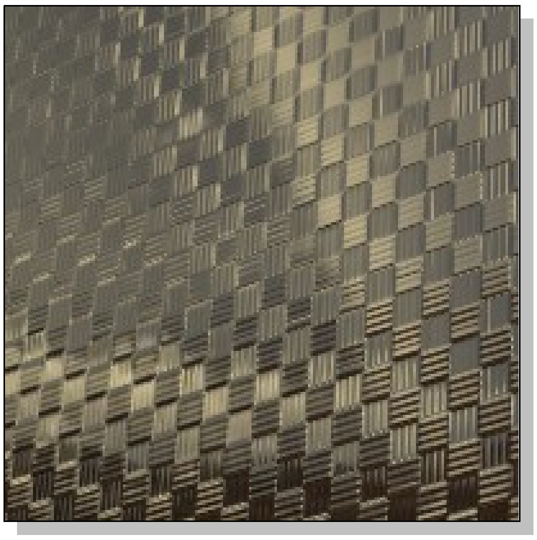
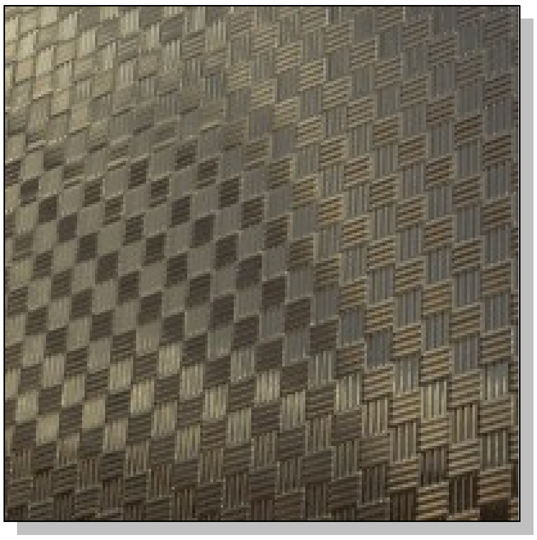
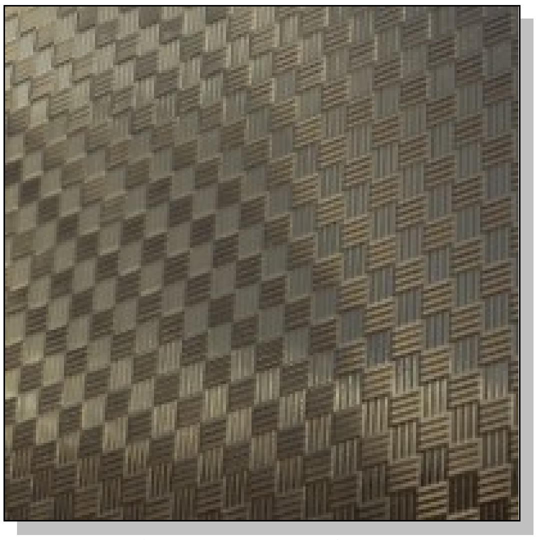
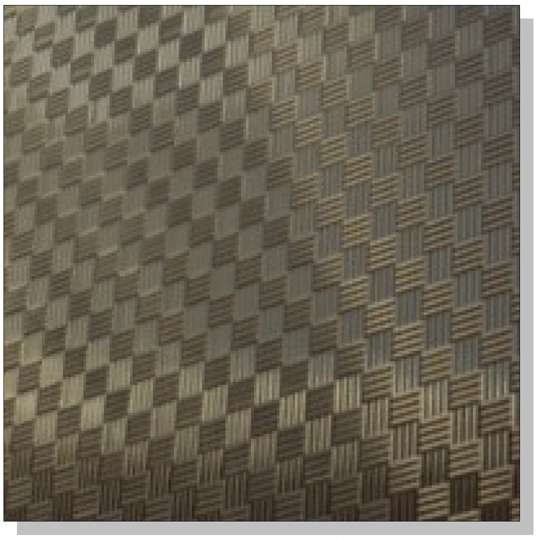

f) Vectors = 32, Scale = 8

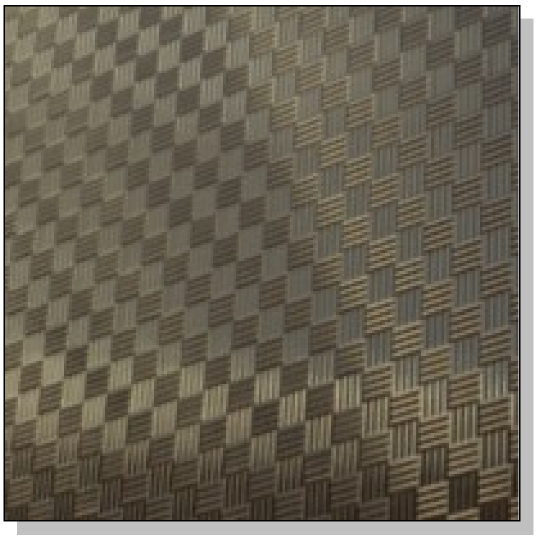
Figure F4: Grid normal map


a) Vectors = 1, Scale = 16


b) Vectors = 2, Scale = 16


c) Vectors = 4, Scale = 16


c) Vectors = 8, Scale = 16


e) Vectors = 16, Scale = 16

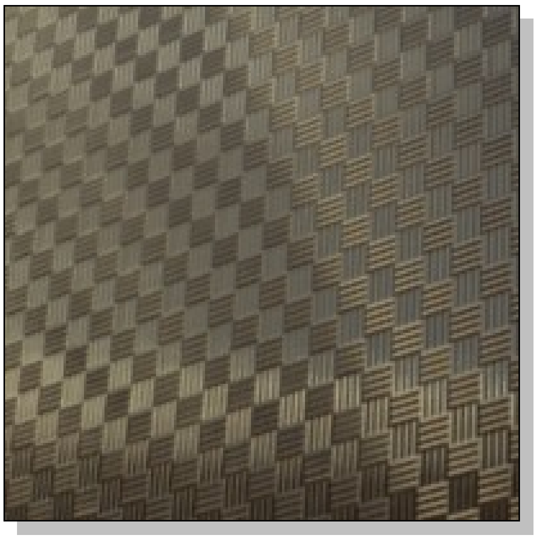
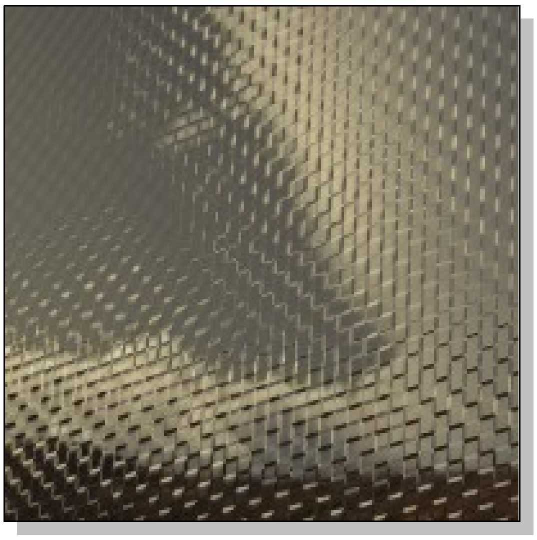
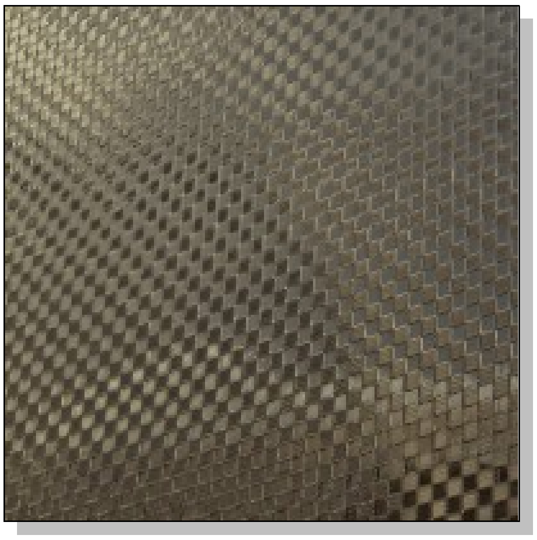

f) Vectors = 32, Scale = 16

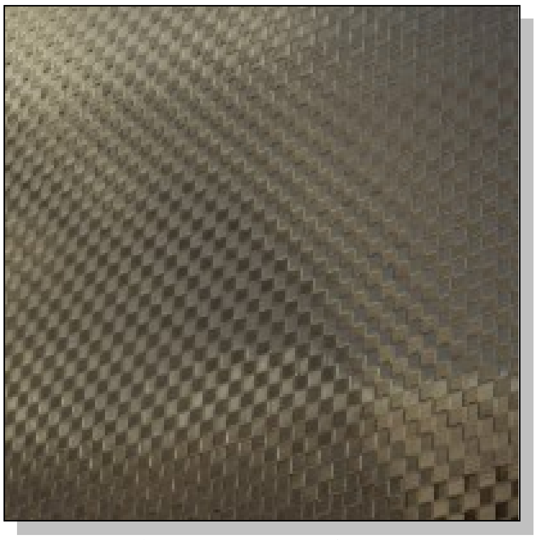
Figure F5: Grid normal map



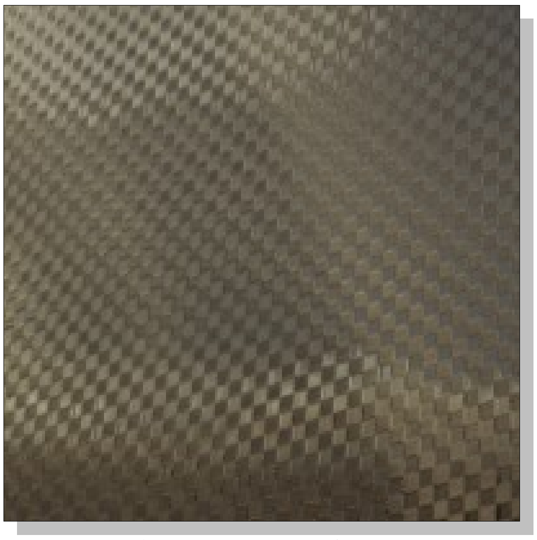
a) Vectors = 1, Scale = 32



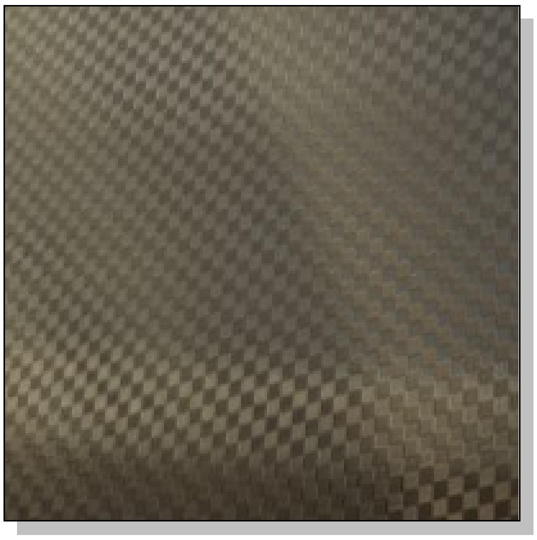
b) Vectors = 2, Scale = 32



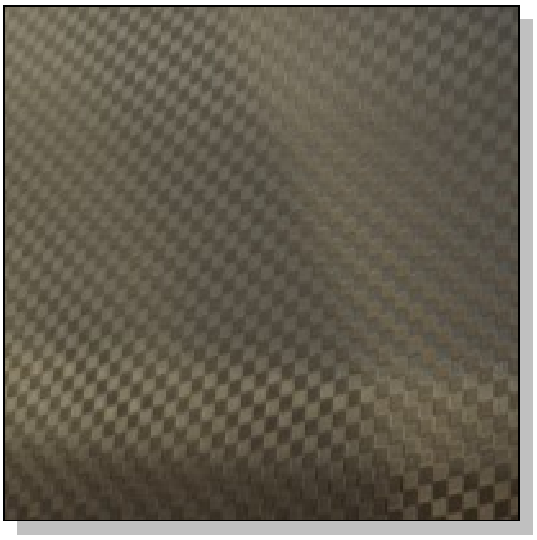
c) Vectors = 4, Scale = 32



c) Vectors = 8, Scale = 32



e) Vectors = 16, Scale = 32



f) Vectors = 32, Scale = 32

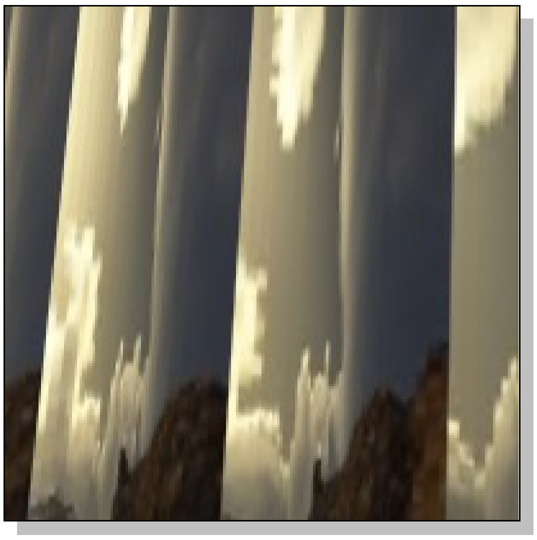
Figure F6: Grid normal map

a) Vectors = 1, Scale = 1

b) Vectors = 2, Scale = 1

c) Vectors = 4, Scale = 1

c) Vectors = 8, Scale = 1


e) Vectors = 16, Scale = 1

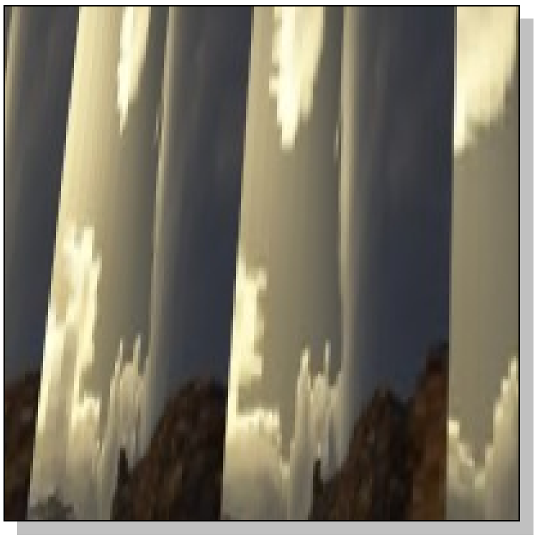
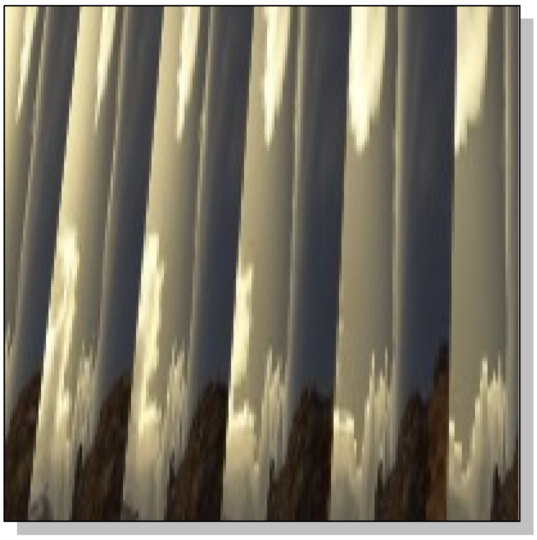
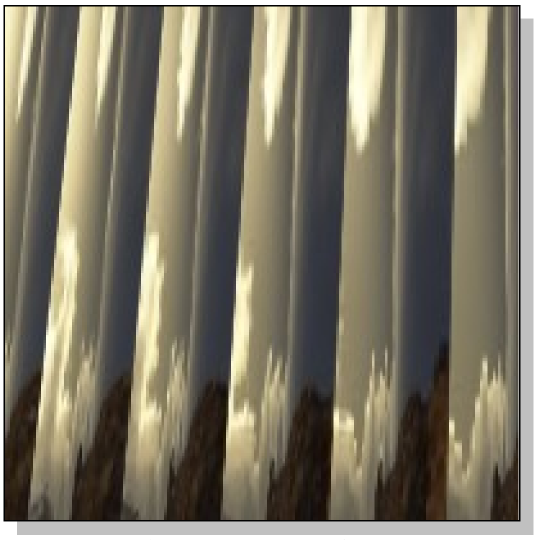
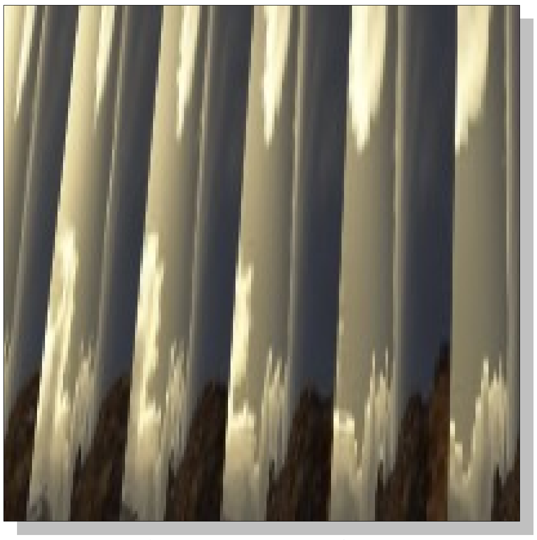

f) Vectors = 32, Scale = 1

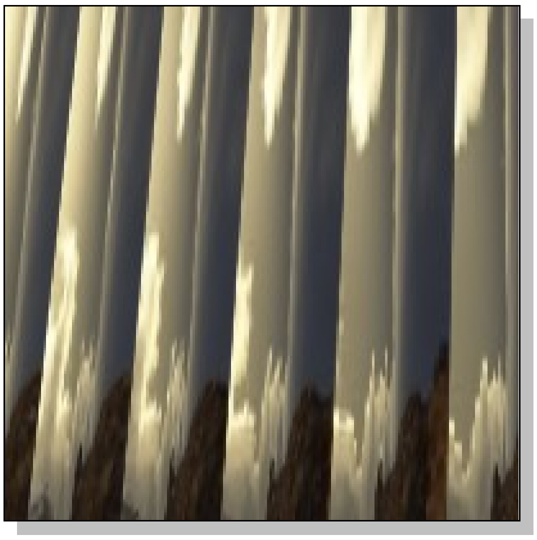
Figure F7: Reeded normal map


a) Vectors = 1, Scale = 2

b) Vectors = 2, Scale = 2


c) Vectors = 4, Scale = 2


c) Vectors = 8, Scale = 2


e) Vectors = 16, Scale = 2

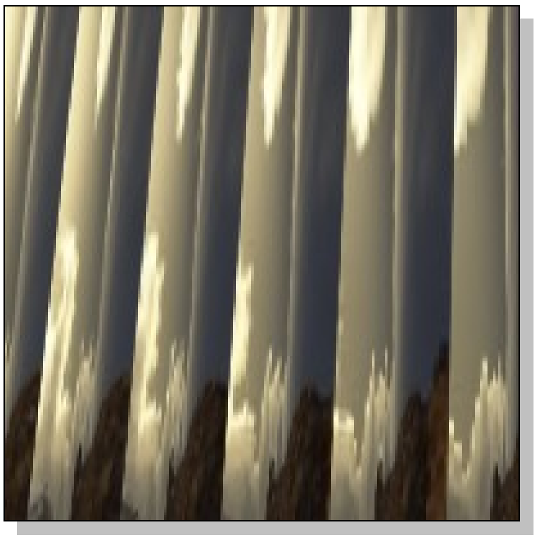

f) Vectors = 32, Scale = 2

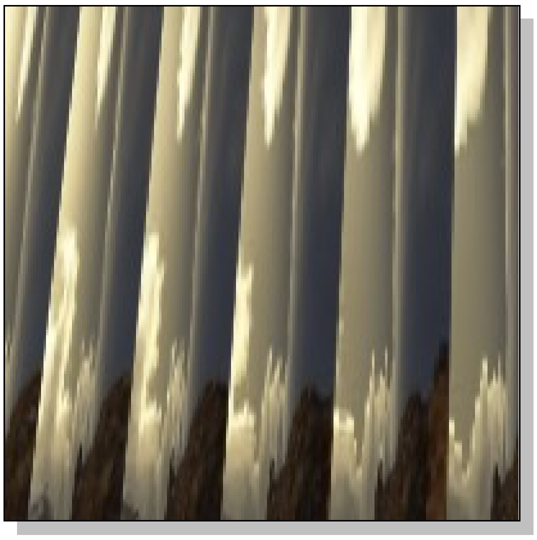
Figure F8: Reeded normal map

a) Vectors = 1, Scale = 4

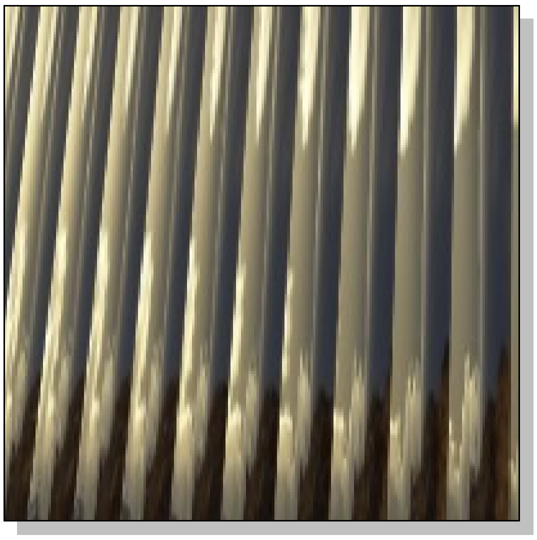
b) Vectors = 2, Scale = 4

c) Vectors = 4, Scale = 4

c) Vectors = 8, Scale = 4



e) Vectors = 16, Scale = 4



f) Vectors = 32, Scale = 4

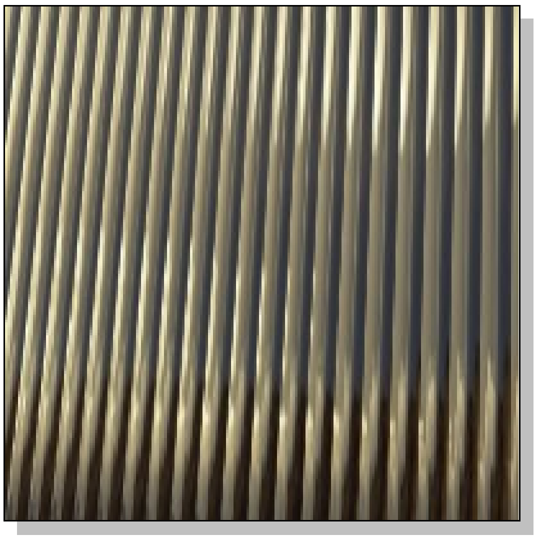
Figure F9: Reeded normal map

a) Vectors = 1, Scale = 8

b) Vectors = 2, Scale = 8

c) Vectors = 4, Scale = 8

c) Vectors = 8, Scale = 8


e) Vectors = 16, Scale = 8

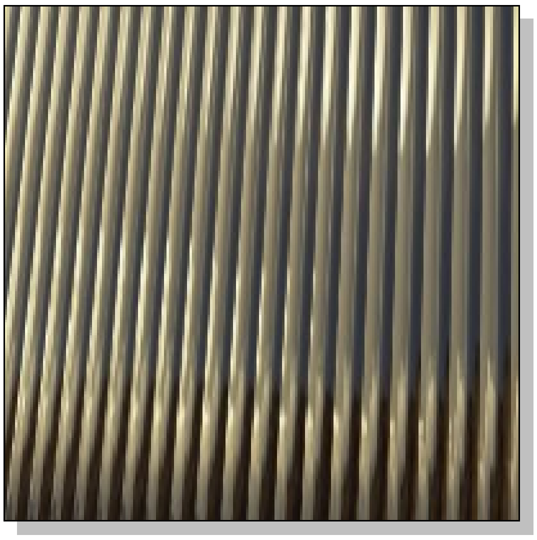

f) Vectors = 32, Scale = 8

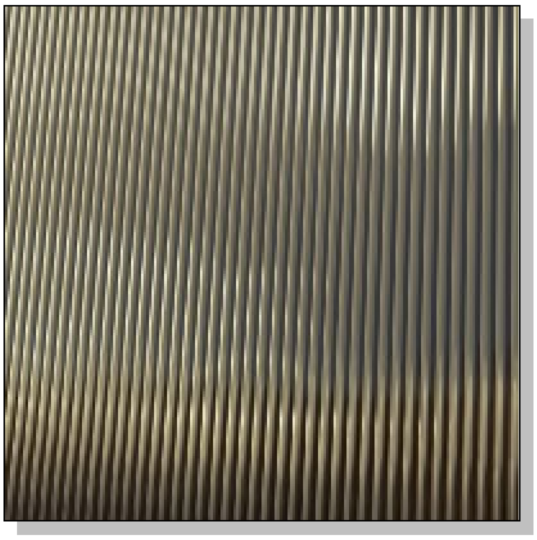
Figure F10: Reeded normal map

a) Vectors = 1, Scale = 16

b) Vectors = 2, Scale = 16

c) Vectors = 4, Scale = 16

c) Vectors = 8, Scale = 16


e) Vectors = 16, Scale = 16

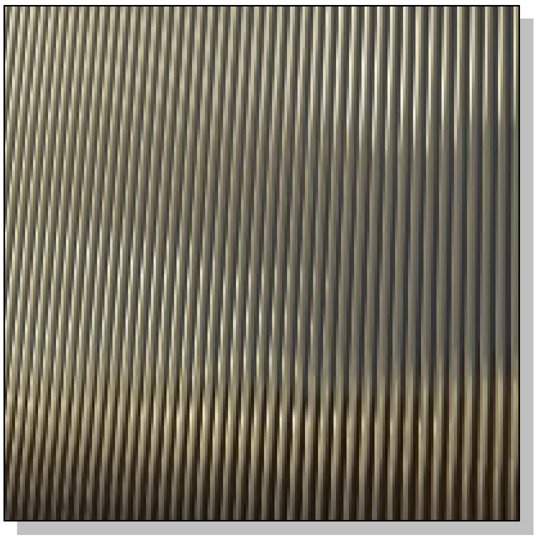

f) Vectors = 32, Scale = 16

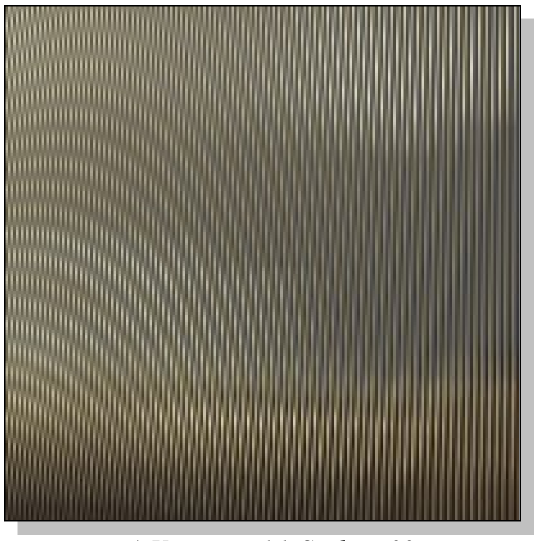
Figure F11: Reeded normal map

a) Vectors = 1, Scale = 32

b) Vectors = 2, Scale = 32

c) Vectors = 4, Scale = 32

c) Vectors = 8, Scale = 32


e) Vectors = 16, Scale = 32

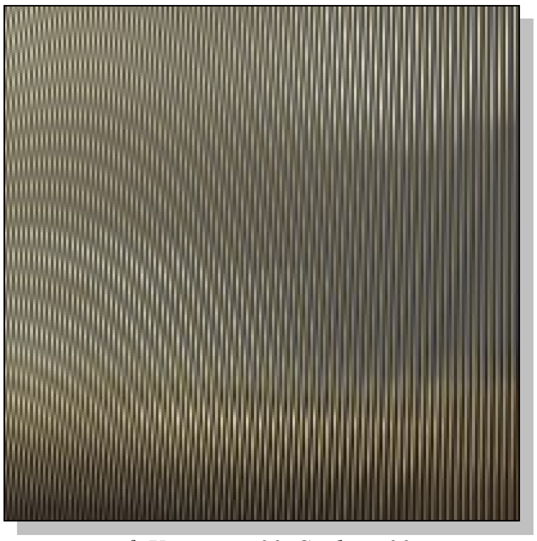

f) Vectors = 32, Scale = 32

Figure F12: Reeded normal map

Appendix G: Super Bump Mapping Performance Graphs

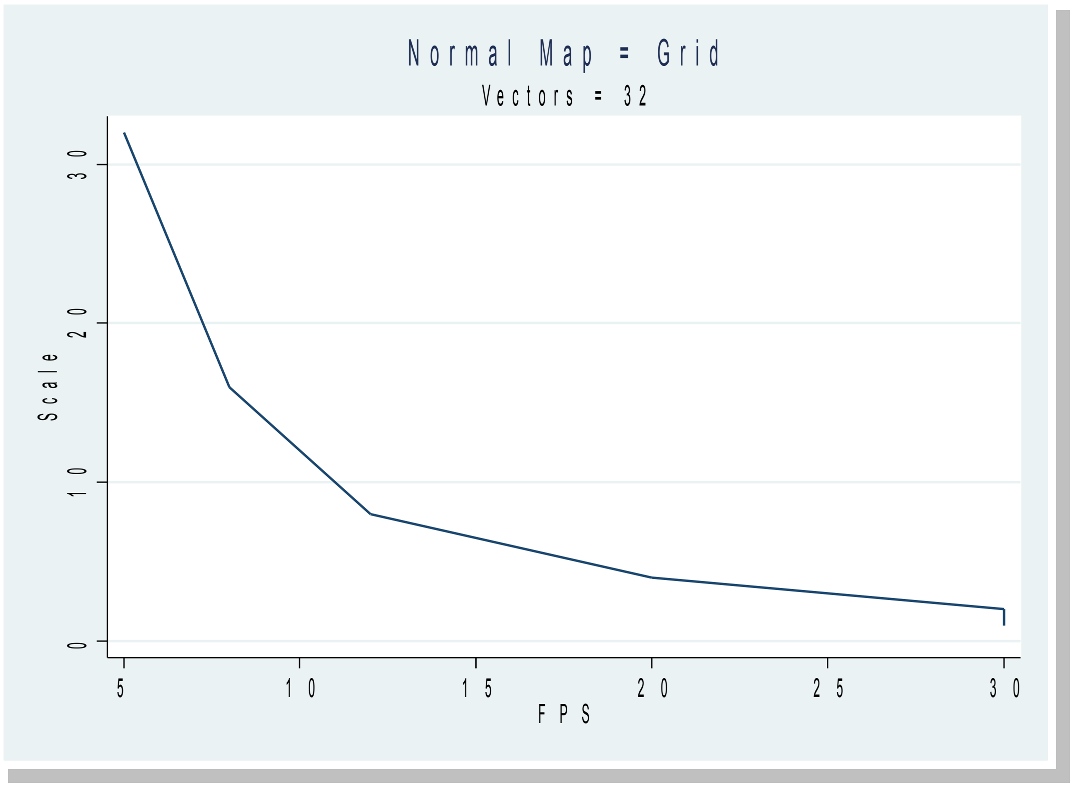


Figure G1: Grid – varying scale

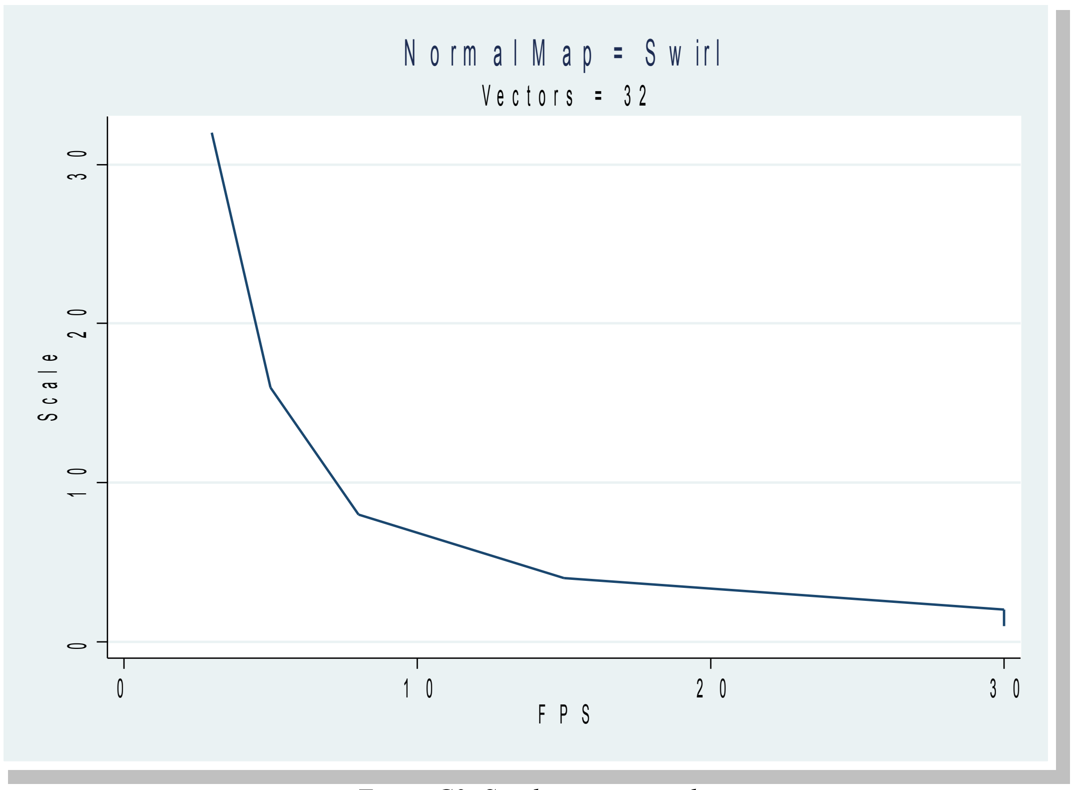


Figure G2: Swirl – varying scale

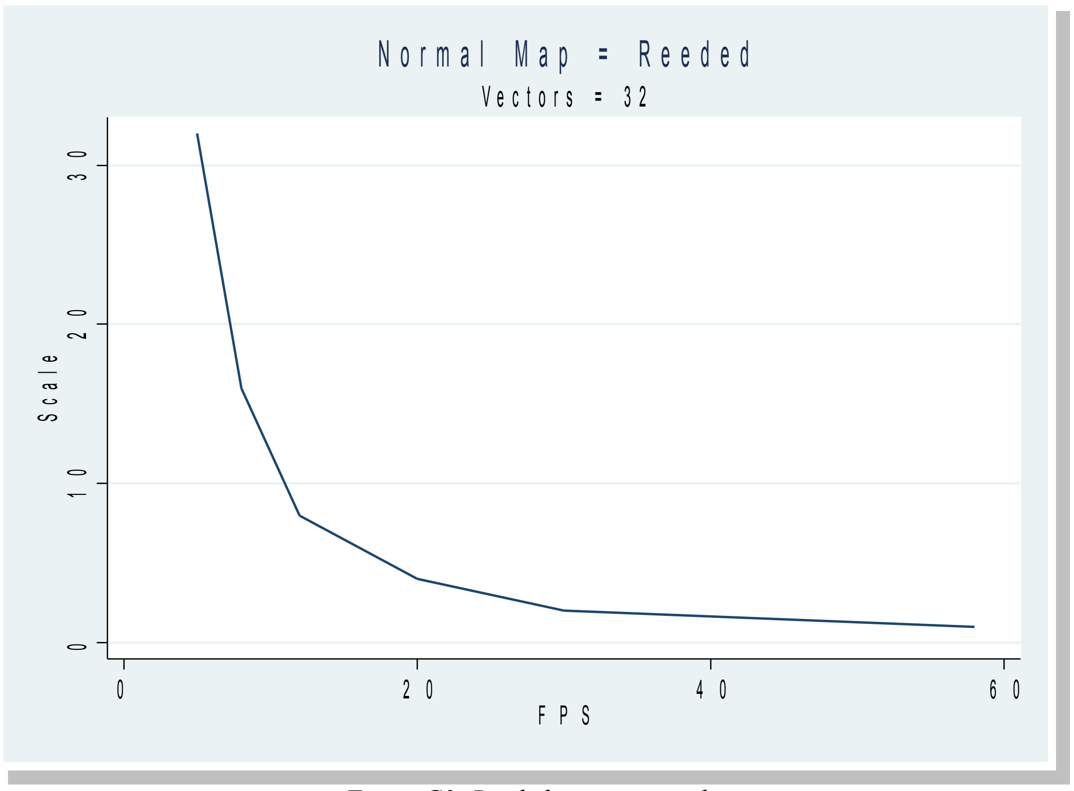


Figure G3: Reeded – varying scale

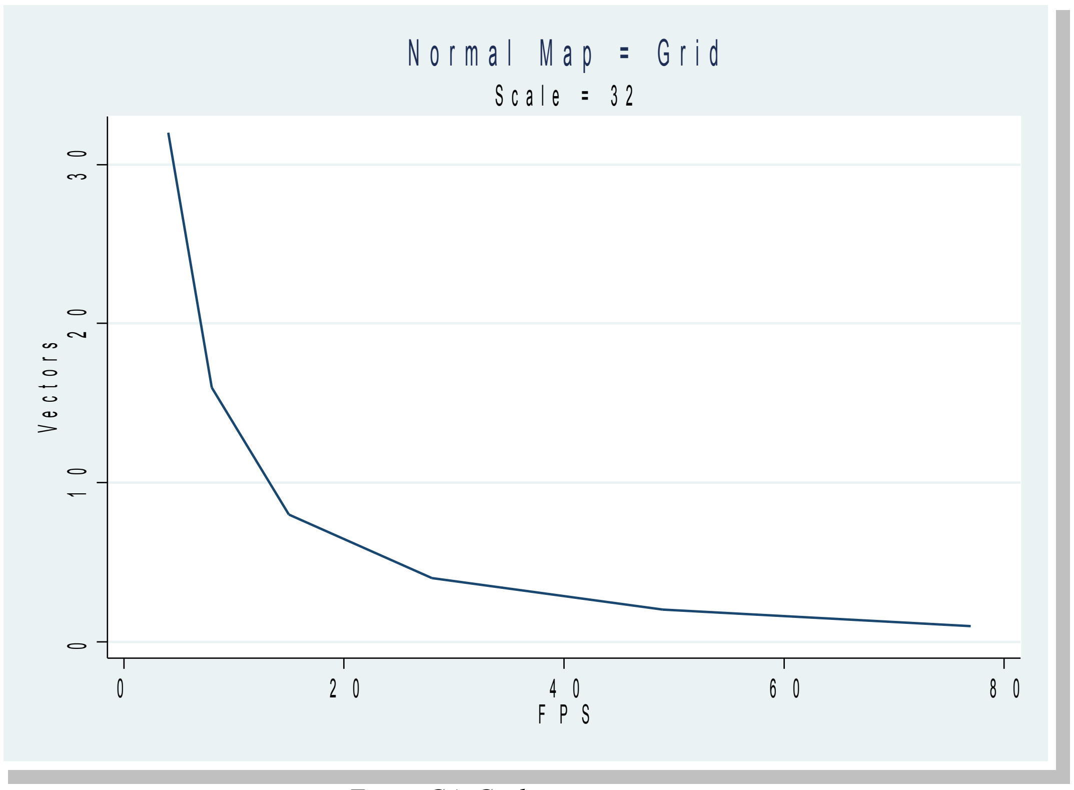


Figure G4: Grid – varying vectors

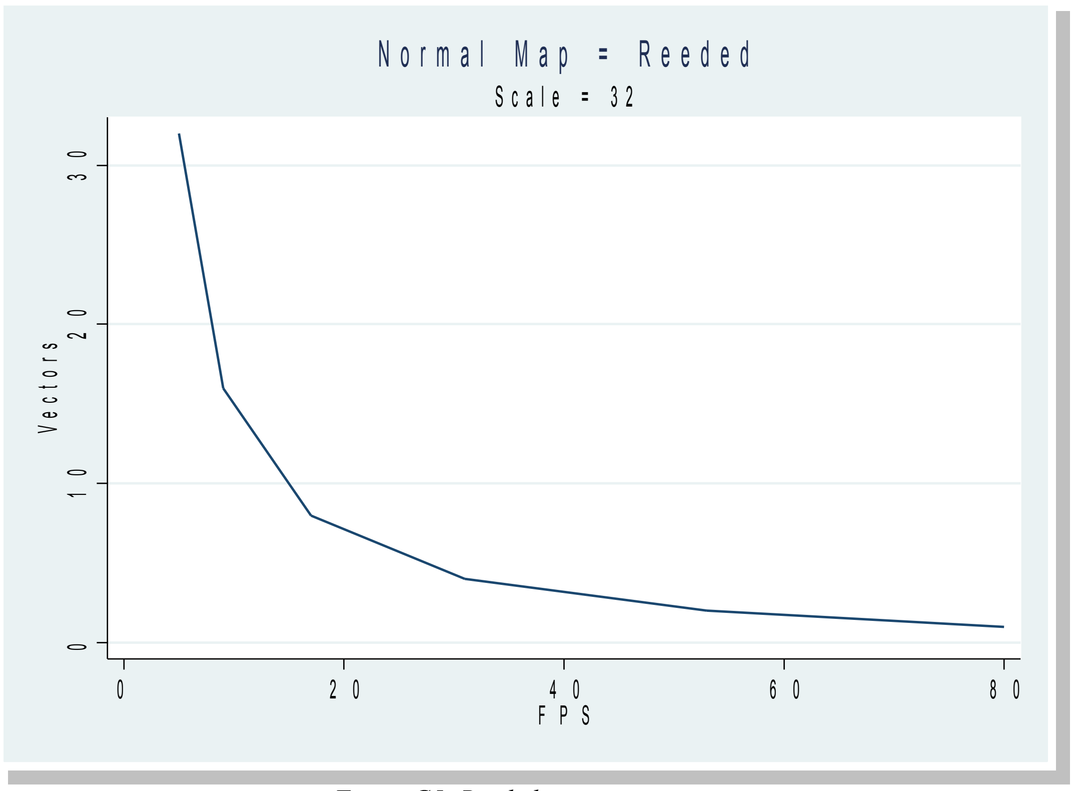


Figure G5: Reeded – varying vectors

Appendix H: Shader Code extracts

Trilinear Fourier Pixel Shader

```
//-----------------------------------------------------------------------------------------
// File:              trilinearfourier.ps
// Version:           V3.0
// Author:            Daniel Rhodes
// Description:       Texture Mapping Pixel Shader in GLSL
// Notes:             GLSL documentation can be found at:
//                    http://developer.3dlabs.com/openGL2/index.htm
//-----------------------------------------------------------------------------------------
// GLSL version
#version 110
#pragma optimize(on)  // Turn off some (but not all!!!) optimisations,
                      // makes more readable ASM
#pragma debug(off)    // Adds extra debug info to output log (ilog_*.txt etc.),
                      // need NVemulate.exe to activate on NVIDIA cards
uniform sampler2D BMPTexture;
uniform sampler2D FreqTexture;
uniform float fFilterWidth;
uniform int iNumFreqs;
uniform vec2 TexMapSize;
uniform vec2 FreqTexMapSize;
uniform int iAngleOffset;
uniform int iLevelOffset;
uniform int iNumLevels;
const float PI = radians( 180.0 );
//-------------------------------------------------------------------------
// Name:    LOD
// In:      dtx, dty
// Out:     subLevel   = calculated sub level (i.e. the LOD from the minor axis of the
pixel footprint),
//          angle      = calcuated current angle sector,
//          fFreqScale = Used to scale the number of frequencies used dependant on
```

```
angle
// Returns:  Level of Detail
// Desc:     Calculates the level of detail
//--------------------------------------------------------------------------
float LOD( vec2 dtx, vec2 dty, out float subLevel, out float angle, out float fFreqScale )
{
            vec2 vTx = vec2( pow( dtx.x, 2.0 ), pow( dty.x, 2.0 ) );
            vec2 vTy = vec2( pow( dtx.y, 2.0 ), pow( dty.y, 2.0 ) );
            // Find top X
            float fTopX = 2.0 * ( ( dtx.x * dty.x ) + ( dtx.y * dty.y ) );
            // Find bottom x
            float fBottomX = vTx.x - vTx.y + vTy.x - vTy.y;
            // Find first extremum angle
            float fTheta= 0.5 * atan( fTopX, fBottomX );   // Angle in screen space,
            // later need angle in tex space for working out components
            float fCosTheta          = cos( fTheta );
            float fSinTheta          = sin( fTheta );
            vec2 vdtp = vec2( 0.0, 0.0 );
            vec2 vdtm = vec2( 0.0, 0.0 );
            // Vector to first extremum (major axis)
            vdtp.x = fCosTheta * dtx.x + fSinTheta * dty.x;
            vdtp.y = fCosTheta * dtx.y + fSinTheta * dty.y;
            // Vector to second extremum (minor axis)
            vdtm.x = -fSinTheta * dtx.x + fCosTheta * dty.x;
            vdtm.y = -fSinTheta * dtx.y + fCosTheta * dty.y;
            // determine length of major axis to set up probes (as defined by
McCormack probes = isotopic filtering operations like trilinear or Gaussian)
            float fPixMajorAxis    = pow( vdtp.x, 2.0 ) + pow( vdtp.y, 2.0 );
            // determine length of minor axis for mip map level selection
            float fPixMinorAxis    = pow( vdtm.x, 2.0 ) + pow( vdtm.y, 2.0 );
            float axisTemp = 0.0;
            if( fPixMinorAxis > fPixMajorAxis ) {
                        axisTemp               = fPixMajorAxis;
                        fPixMajorAxis          = fPixMinorAxis;
                        fPixMinorAxis          = axisTemp;
            }
            // phi(footprint angle in texture space) = arctan(vdtp.y, vdtp.x);
            angle = atan( vdtp.y, vdtp.x );
            angle += radians(22.5);
            if ( angle > PI )
                        angle -= PI;
            float fTexelDiff = ( ( 1.0 / TexMapSize.x ) * ( 1.0 / TexMapSize.x ) ) +
( ( 1.0 / TexMapSize.y ) * ( 1.0 / TexMapSize.y ) );
            // Work out Level of Detail & compensate for not taking square root
            float fLOD             = log2( fPixMajorAxis / fTexelDiff ) / 2.0;
            subLevel               = log2( fPixMinorAxis / fTexelDiff ) / 2.0;
            fFreqScale = ( fLOD - subLevel ) / float(iNumLevels);//0.125 * ( fLOD -
subLevel );
```

```
		fFreqScale *= 5.0;
		fFreqScale = clamp(fFreqScale, 0.0, 1.0);
		return( fLOD );
}

//---------------------------------------------------------------------------
// Name:	BilinearInterpolation
// In:		iLOD = Level of Detail in integer format
// Out:		-
// Returns:	Pixel values after interpolation
// Desc:	Bilinearly interpolates our MIP map level
//---------------------------------------------------------------------------
vec4 BilinearInterpolation( int iLOD )
{
		// Find point sampled texels coodinates in image space
		float fConvfactorToLODSpace = exp2( float(iLOD) );
		vec2 MipCoordImageSpace = ( gl_TexCoord[0].xy * TexMapSize ) /
fConvfactorToLODSpace;
		// Work out Texture coodinates (in image space) for all texels to sample
		float fFloorX		= floor( MipCoordImageSpace.x );
		float fFloorY		= floor( MipCoordImageSpace.y );
		float fCeilX		= ceil( MipCoordImageSpace.x );
		float fCeilY		= ceil( MipCoordImageSpace.y );
		// Work out reciprical to save on divides
		vec2 MipCoordImageSpaceRecip = fConvfactorToLODSpace /
TexMapSize;
		// Multiply by reciprocal, same as fFloorX / TexMapSize.x
		float fFloorXOverTexX = fFloorX * MipCoordImageSpaceRecip.x;
		float fCeilXOverTexX = fCeilX * MipCoordImageSpaceRecip.x;
		float fFloorYOverTexY = fFloorY * MipCoordImageSpaceRecip.y;
		float fCeilYOverTexY = fCeilY * MipCoordImageSpaceRecip.y;
		vec2 TexCoord00 = vec2( fFloorXOverTexX, fFloorYOverTexY );
		vec2 TexCoord10 = vec2( fCeilXOverTexX, fFloorYOverTexY );
		vec2 TexCoord01 = vec2( fFloorXOverTexX, fCeilYOverTexY );
		vec2 TexCoord11 = vec2( fCeilXOverTexX, fCeilYOverTexY );
		// Use texture coodinates to sample actual texels
		vec4 vTexel00 = texture2DLod( BMPTexture, TexCoord00, float(iLOD) );
		vec4 vTexel10 = texture2DLod( BMPTexture, TexCoord10, float(iLOD) );
		vec4 vTexel01 = texture2DLod( BMPTexture, TexCoord01, float(iLOD) );
		vec4 vTexel11 = texture2DLod( BMPTexture, TexCoord11, float(iLOD) );
		// Find weight value for linear interpolation
		vec2 Weight		= fract( MipCoordImageSpace );
		vec4 vLerpTop		= mix( vTexel00, vTexel10, Weight.x );
		vec4 vLerpBottom	= mix( vTexel01, vTexel11, Weight.x );
		vec4 vFinalPixColour	= mix( vLerpTop, vLerpBottom, Weight.y );
		return( vFinalPixColour );
}
```

```
//---------------------------------------------------------------------------
// Name:     AnisotropicFilteringWithFourier
// In:       -
// Out:      -
// Returns:  Pixel values after filtering
// Desc:     Applies Fourier based texture filtering to our image
//---------------------------------------------------------------------------
vec4 AnisotropicFilteringWithFourier( )
{
            // Get partial derivates
            vec2 dtx = dFdx( gl_TexCoord[0].xy );
            vec2 dty = dFdy( gl_TexCoord[0].xy );
            float subLevel          = 0.0;
            float fAngle            = 0.0;
            float fFreqScale        = 0.0;
            float fLOD              = LOD( dtx, dty, subLevel, fAngle, fFreqScale );
            int iAngle              = 0;
            if ( fAngle < 0.0 )
                        iAngle = int(floor( ( fAngle + PI ) / radians( 45.0 ) ));
            else
                        iAngle = int(floor( ( fAngle ) / radians( 45.0 ) ));

            if ( iAngle < 0 )
                        iAngle = 0;
            if ( fLOD < 0.0 )
                        fLOD = 0.0;
            if ( subLevel < 0.0 )
                        subLevel = 0.0;

            // Fourier
            vec2 Lcood = vec2( 0.0, 0.0 );
            // Scale for texture size
            vec2 dtxScaled = TexMapSize.x * dtx;
            vec2 dtyScaled = TexMapSize.y * dty;
            mat2 mM = mat2( dtxScaled.x, dtxScaled.y , dtyScaled.x, dtyScaled.y );
            vec2 vPpkqk             = vec2( 0.0, 0.0 );
            vec2 fpdotMpkqk         = vec2( 0.0, 0.0 );
            float fSpkqk            = 0.0;
            float Scale  = 0.003921568627450980392156862745098;  // = 1 / 255
            float index  = 0.0;
            vec4 temp1 = vec4( 0.0, 0.0, 0.0, 0.0 );
            vec4 temp2 = vec4( 0.0, 0.0, 0.0, 0.0 );
            float adjxfreq = 0.0, adjyfreq = 0.0;
            float angle  = 0.0, sinA = 0.0, cosA = 0.0;
            float ftotR  = 0.0, ftotG= 0.0, ftotB = 0.0;
            float fR     = 0.0, fG = 0.0, fB = 0.0;
            float xfreq, yfreq;
            Lcood.y = ( ( float(iAngle) * float(iAngleOffset) ) + ( float(int(fLOD)) *
float(iLevelOffset) ) + float(int(subLevel)) ) / FreqTexMapSize.y;
            for ( int i = int(float(iNumFreqs)) - 1; i >= 0; i-- ) {
                        // Calculate index to get approprate pixel from texture
                        index = ( float( i ) * 2.0 );
                        Lcood.x = index / FreqTexMapSize.x;
                        // Using calculated texture coordinates perform texture fetches
                        // Assuming texture size = 512
                        temp1       = texture2D( FreqTexture, Lcood );
                        Lcood.x = ( index + 1.0 ) / FreqTexMapSize.x;
                        temp2       = texture2D( FreqTexture, Lcood );
                        xfreq = temp1.r;
                        yfreq = temp1.g;
                        // Adjust for texmap size
                        if ( xfreq > ( FreqTexMapSize.x / 2.0 ) )
                                adjxfreq =  xfreq - FreqTexMapSize.x;
                        else
                                adjxfreq =  xfreq;
                        if ( yfreq > ( FreqTexMapSize.y / 2.0 ) )
                                adjyfreq =  -yfreq + FreqTexMapSize.y;
                        else
                                adjyfreq =  -yfreq;
                        // Work out anti-aliasing factor
                        vPpkqk = vec2( ( adjxfreq / FreqTexMapSize.x * 2.0 * PI ),
( adjyfreq / FreqTexMapSize.y * 2.0 * PI ) );
                        fpdotMpkqk = vPpkqk * mM;
                        fSpkqk = exp( -( ( fFilterWidth / 2.0 ) * dot( fpdotMpkqk,
fpdotMpkqk ) ) );
                        // phase angle
                        angle = ( adjxfreq * gl_TexCoord[0].x + adjyfreq *
gl_TexCoord[0].y ) * 2.0 * PI;
                        cosA = cos( angle );
                        sinA = sin( angle );
                        fR = ( temp1.b * cosA - temp1.a * sinA );
                        fG = ( temp2.r * cosA - temp2.g * sinA );
                        fB = ( temp2.b * cosA - temp2.a * sinA );
                        // Include anti-aliasing factor
                        fR *= fSpkqk;
                        fG *= fSpkqk;
                        fB *= fSpkqk;
                        ftotR += fR;
                        ftotG += fG;
                        ftotB += fB;
            }
            vec4 AnisoPixColourHighRes = vec4( ftotR * Scale, ftotG * Scale, ftotB *
Scale, 0.0 );
            vec4 BilinearPixColourHighRes    = BilinearInterpolation( int(fLOD) );
            vec4 BilinearPixColourLowRes     = BilinearInterpolation( int(++fLOD) );
            // This is an intentional precriment to set fLOD for use in the next Fourier
```

```
loop
	vec4 vMipCol = mix( BilinearPixColourHighRes,
BilinearPixColourLowRes, fract(fLOD) );
	fR = 0.0;
	fG = 0.0;
	fB = 0.0;
	ftotR = 0.0;
	ftotG = 0.0;
	ftotB = 0.0;
	Lcood.y = ( ( float(iAngle) * float(iAngleOffset) ) + ( float(int(fLOD)) *
float(iLevelOffset) ) + float(int(subLevel)) ) / FreqTexMapSize.y;
	for ( int i = int(float(iNumFreqs) * fFreqScale) - 1; i >= 0; i-- ) {
		// Calculate index to get approprate pixel from texture
		index = ( float( i ) * 2.0 );
		Lcood.x = index / FreqTexMapSize.x;
		// Uing calculated etxture coordinates perform texture fetches
		// Assuming texture size = 512
		temp1	= texture2D( FreqTexture, Lcood );
		Lcood.x = ( index + 1.0 ) / FreqTexMapSize.x;
		temp2	= texture2D( FreqTexture, Lcood );
		xfreq = temp1.r;
		yfreq = temp1.g;
		if ( xfreq > ( FreqTexMapSize.x / 2.0 ) )
			adjxfreq =  xfreq - FreqTexMapSize.x;
		else
			adjxfreq =  xfreq;
		if ( yfreq > ( FreqTexMapSize.y / 2.0 ) )
			adjyfreq =  -yfreq + FreqTexMapSize.y;
		else
			adjyfreq =  -yfreq;
		// Work out anti-aliasing factor
		vPpkqk = vec2( ( adjxfreq / FreqTexMapSize.x * 2.0 * PI ),
( adjyfreq / FreqTexMapSize.y * 2.0 * PI ) );
		fpdotMpkqk = vPpkqk * mM;
		fSpkqk = exp( -( ( fFilterWidth / 2.0 ) * dot( fpdotMpkqk,
fpdotMpkqk ) ) );
		// phase angle
		angle = ( adjxfreq * gl_TexCoord[0].x + adjyfreq *
gl_TexCoord[0].y ) * 2.0 * PI;
		cosA = cos( angle );
		sinA = sin( angle );
		fR = ( temp1.b * cosA - temp1.a * sinA );
		fG = ( temp2.r * cosA - temp2.g * sinA );
		fB = ( temp2.b * cosA - temp2.a * sinA );
		// Include anti-aliasing factor
		fR *= fSpkqk;
		fG *= fSpkqk;
		fB *= fSpkqk;
```

```
                    ftotR += fR;
                    ftotG += fG;
                    ftotB += fB;
          vec4 AnisoPixColourLowRes        = vec4( ftotR * Scale, ftotG * Scale,
ftotB * Scale, 0.0 );
          vec4 AnisoPixColour    = mix( AnisoPixColourHighRes,
AnisoPixColourLowRes, fract(fLOD) );
          return( vMipCol + AnisoPixColour );
}

//-----------------------------------------------------------------------------
// Name:     main
// In:       -
// Out:      -
// Desc:     Main function
//-----------------------------------------------------------------------------
void main( ) {
          gl_FragColor = AnisotropicFilteringWithFourier( );
}
```