LULEÅ UNIVERSITY OF TECHNOLOGY

2003:306 CIV

MASTER'S THESIS

# Accelerating Real-Time Graphics with High Level Shading Languages

EMIL PERSSON

**MASTER OF SCIENCE PROGRAMME**

Department of Computer Science and Electrical Engineering
Division of Media Technology

2003:306 CIV • ISSN: 1402 - 1617 • ISRN: LTU - EX - - 03/306 - - SE

# Contents

## Abstract

This report covers the work of the author, Emil Persson, done during his time at ATI Technologies, Markham, Toronto, Canada, in the beginning of 2003. Topics in the field of high level shading languages are discussed. The languages themselves in terms of syntax and semantics are discussed as well as utilities and applications of high level shading languages and its relations to low level shading languages. Manual and automatic ports between different shading languages are also discussed. The report covers the theme mostly from a real-time rendering perspective.

Research work for various usages in the fields of real-time rendering is also presented. A number of techniques for gaming, industrial and cinematic use were developed and implemented with the RenderMonkey prototype toolset. Moreover, a few simulations of the behavior of certain materials such as wood and glass were produced, as well as additional techniques for postprocessing images. The techniques are described and explained in detail as well as analyzed and compared in performance, quality and utility aspects.

## Goals

The goal of this thesis and the work behind this report is to explore and analyze various fields of the subset of computational graphics that is high level shading languages. This includes evaluation of the ongoing effort on the OpenGL 2.0 shading language specification, both from usability and technical points of view, but also practical tests of conformance, performance and stability of the alpha driver implementing the specification draft, as well as comparisons between the OpenGL 2.0 high level shading language and other high level shading languages such as DirectX9 HLSL (High Level Shading Language) and the typically software rendered shading language RenderMan. Also included is to investigate the usefulness of research projects such as the Ashli (Advanced Shader Language Interface) project going on at ATI, and finding out how it can best fit into a developer's toolset, as well as making an in depth analysis of the prospects for the future directions of this project.

Another goal of this thesis is to explore the field of high level shading through research on what kinds of visual effects that can be achieved with today's state of the art hardware and existing high level shading languages. This includes developing new techniques for achieving certain goals, finding out what limitations there are and how they can be worked around, what level of hardware support a technique may require, comparing performance attributes between similar techniques or techniques aimed for achieving the similar goals, and analyzing if there are ways to improve them or if there are suitable quality/performance tradeoffs that can be applied. As a secondary goal of this research the RenderMonkey prototype toolset under development by ATI was to be evaluated through the use of it for this research and bugs or other flaws be reported to the RenderMonkey developer team, as well as usability problem or suggestions for improvement.

## Acknowledgements

This master thesis was performed at ATI Technologies Inc. in Markham, Toronto, Canada during the time of February 17 to April 25, 2003.

The author would like to thank the following people for their direct assistance and support in this work:

Michael Smith
Jeremy Sandmel
Glenn Ortner
Evan Hart
Jeff Royle
Natasha Tatarchuk
Callan McInally
Guennadi Riguer
Justin Cooney
Rob Mace

The author would also like to thank the following people for helping me out with issues not directly related but relevant for the performing of my work:

Yasmin Khalfan
Lina Bagchi

The author would also like to thank and apologize to those who may have provided assistance and whose help I may have forgotten or whose name I could not remember.

Furthermore, the author wants to thank all his friends, relatives, classmates, online friends, online communities and others who have encouraged and supported me while doing this and made all this possible.

*Luleå, Sweden - 2003-05-07*

*Emil Persson*

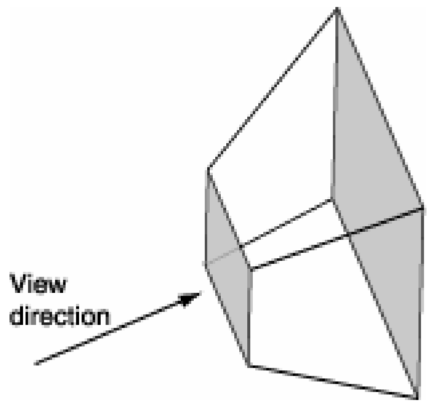
# Preface

## Basic real-time graphics concepts

This report, while written in as simple language as possible, require basic knowledge in 3D graphics for the reader to assimilate the content. The target audience is someone with decent programming skills in any graphics API (Application Programming Interface) or at least someone with decent experience working with 3D graphics applications. To aid the less knowledgeable reader in understanding this report some fundamental concepts of real-time graphics will be presented in layman terms in this section. The experienced reader may decide to skip this section and go directly to the introduction. In addition to this quick presentation of the fundamental principles there is also a glossary and a list of common abbreviations from the field of graphics that the reader may want to refer to occasionally while reading the report.

### Hardware rendering paradigm

Hardware rendering in polygon based. This means that it's based on simple rendering primitives such as triangles. The rendering pipeline begins with application provided vertex data. Vertex data is usually provided through an array of vertices that contains various kinds of data. This includes the vertex's position in space, the surface normal, the corresponding texture coordinate or other data that is needed for the drawing operation. Three vertices make up a triangle. Before the triangle can be rasterized (mapped onto pixels) each vertex of the triangle must be processed. Traditionally this is called transformation, though today the term vertex shading is also used. Evidently, this new name is derived from the term vertex shader. A shader is script for graphics processing. A vertex shader is a kind of shader that processes vertex data, that is, it does the transformation or vertex shading operation. The name transformation comes from the original way vertices were processed, before the shader era, namely by transforming the vertex data by a number of matrixes. This task is still done in most vertex shaders, but vertex shaders tend to also do other work too. The first matrix the vertex goes through is the model-view matrix. This matrix transforms the vertex to a vector space around the "camera". This typically involves translating the vertex to place the camera at position (0, 0, 0), and then rotating the vertex around the origin according to the direction the camera points at. Next the newly transformed vertex goes through a projection matrix. This matrix transforms the vertex into the viewing frustum, that is, a pyramid shaped object with cut off top like in the figure, which represents the volume that contains everything that will be visible on the screen. After the vertex shader has processed a vertex it outputs the transformed vertex, which is a four component vector. It contains the x, y and z coordinates of the final vertex and also w coordinate. In the last processing step, which is done outside of the vertex shader, the x, y, and z coordinates will be normalized, that is, divided with the w coordinate. This division is what provides the perspective. If the final vertex is visible (is inside the frustum), will have all three coordinates in the [-1, 1] range. The x and y coordinates will map the vertex to a position on the screen, while the z coordinate will be used for visibility determination during the rasterization.



When the three vertices have been processed by the vertex shader the triangle is ready for rasterization. Rasterization is the process of laying out the triangle over the set of pixels it covers on the screen. For each pixel belonging to the triangle a fragment will be processed. A fragment is basically a “pixel” being processed. Once it’s processed and written to the framebuffer the fragment becomes a pixel. The difference between a fragment and a pixel is often not that important, which is also the reason why Direct3D tend not to make a clear distinction (OpenGL does however). A fragment is processed by a fragment shader (or pixel shader in Direct3D terms). The fragment shader computes the color of the fragment and may optionally also alter its depth. Normally, when the depth is left unchanged, it is derived from the vertices’ z coordinates from the vertex shader output. This depth is then checked against the values in the depthbuffer, or Z-buffer as it’s also called. The depthbuffer is a screen sized buffer containing depth values of previously written pixels at the same location on the screen. It is used to determine whether this fragment is in front or behind a previously written pixel. This way visibility of the fragment can be determined, and if found to be visible the fragment is written to the framebuffer and the depthbuffer updated, otherwise the fragment is discarded. Optionally the fragment can be chosen to be blended with the current framebuffer pixel rather than overwriting it. This is useful for instance for transparency. The blending operation is a set of fixed functions that can be applied between the two.

These steps are repeated for all primitives that the scene is composed of, which produces the desired image in the framebuffer. This image will then be put onto the screen, and the cycle can be repeated to produce more frames.

## Common graphics terms

In addition to the common real-time graphic rendering paradigm there are also a number of terms frequently used in graphics algorithms that the reader will have to understand. These will be presented shortly here.

Since lighting is an important part in many graphical algorithms it’s natural that this report frequently refers to various common lighting concepts. Many common lighting models consist of a few standard components: diffuse, specular and ambient. Diffuse lighting is roughly how light affects a matte material, such as unpolished wood, fabric, paper etc. Specular lighting on the other hand is how light affects shiny objects, such as metal, many kinds of plastic and glass etc. Finally, ambient is a global illumination approximation, a constant light that illuminates the material uniformly across the whole surface. It does not take the light position or direction into account.

The most important piece of data for lighting computations is the normal vector for the surface. Whenever the word “normal” is used in this report (and in graphics in general), it is understood that it refers to the surface normal, unless stated otherwise. Commonly the normal will be interpolated across the surface to get a smooth appearance, instead of treating triangles as flat entities. Interpolated normals will often have to be normalized, i.e. turned into unit length. This may seem unintuitive at first since all input normals are unit length (or should be), but since the interpolation is linear it will result in normal vectors that are shorter then 1, thus the need for normalization. Depending on the curviness of the surface the normalization may or may not be required for good-looking visuals. Normalization will of course be applied to other kinds of vectors than normals too when needed.

## High level shading languages

The essence of this work is based on high level shading languages. A high level shading language is like any other high level language, except that it's for graphics. For vertex and fragment shaders there exist low level shading languages such as the vs2.0 and ps2.0 (vertex and pixel shader 2.0) in Direct3D and GL_ARB_fragment_program and GL_ARB_vertex_program in OpenGL. These are programmed in an assembly-like language. A high level language is to these models like C/C++ is to assembly language. It provides a language built on variables, structures, functions and other logical constructs, as opposed to direct hardware instructions. The best way to show the difference is by providing an example:

```
float4 lightWood;
float4 darkWood;
float4 ringFreq;

sampler Pulse;

float4 hlsl_rings(float4 Pshade: TEXCOORD0): COLOR {
     float scaledDist = length(Pshade.xy) * ringFreq;
     float blendFactor = tex1D(Pulse, scaledDist);

     return lerp(darkWood, lightWood, blendFactor);

}
```

```
ps.2.0

def c0, 2.0, -1.0, 0.5, 0.5
def c1, 1.0,  1.0, 0.1, 0.0

dcl t0.xyzw

dcl_2d s1

dp2add r0, t0, t0, c1.w
rsq r1.x, r0.x
mul r0, r1.x, r0.x
mul r0, r0, c2.w

texld r0, r0, s1

mov r1, c3
lrp r2, r0.x, c2, r1

mov oC0, r2
```

These two shaders are written in DX9 HLSL and DX9 ps2.0 respectively and do the exact same thing. As should be fairly obvious, the first one is not only shorter, but also much clearer as to what it does than the second one. The advantages of using a high level shading for graphics are pretty much the same as for other programming tasks: higher productivity, more readable code, less bugs and potentially higher performance (depending on the quality of the compiler). There are hardly any disadvantages, except that you lose some low level control, but that's generally not very valuable anyway.

High level shading languages for real-time graphics is a relatively new concept. There have previously been some projects that have mapped general shading onto the standard fixed function OpenGL shading pipeline, but only recently have high level programming languages mapping directly to programmable functionality of hardware been introduced. This report covers this interesting field of graphics these languages opens up.

## *Introduction*

Real-time 3D graphics is a rapidly progressing technology sector that has seen incredible developments over the last few years. From the first real 3D chipset being born with the release of the Voodoo graphics chipset in October 1996, the industry has continuously outdone itself and may be the only technology sector where you can say that every new generation of hardware truly has been more or less revolutionary. Due to this incredible technology development and intensive competition many of the original players on the graphics scene has gone out of business and the market is at the of this writing more or less left with only two major players in the high-end sector, ATI Technologies Inc. and NVIDIA. After this development and some subtle hints in the last generations that the industry is finally about to slow down ATI proved that there is no such thing as slowdown in this industry by releasing the R300 chipset, a chipset that easily can compete for the all-time most revolutionary graphics chip award.

## *History and trends*

There are a couple of relevant trends that are important for the 3D graphics industry that need to be addressed. In order to understand these trends a short summary of the short history of 3D graphics from various perspectives helps to identify these trends.

### Converging of gaming and professional sectors

While 3D graphics hardly is a new concept, 3D graphics for the average consumer doesn't even go a full decade back in time. Silicon Graphics Inc. (SGI) is responsible for a large portion of what 3D graphics of today ended up like. Much of the initial research and experience from SGI is what went into the consumer market once the technology were ready for it, though not through SGI themselves. SGI lived for a long time on the concept of 3D graphics being a topic for a few selected professionals who were ready to pay high premiums for their graphics solutions. In the early 90's, the professional market was pretty much the only market there was in 3D graphics [1]. However, the first revolution started when 3Dfx released their Voodoo graphics chip in 1996 [2]. Suddenly 3D graphics in real-time was available for the normal consumer. SGI and other professional graphic solutions providers reacted by ensuring that they could differentiate themselves enough to still be worth the premium prices they used to charge for their solutions. This was done by offering huge amounts of memory (according to the standard of those days) on their chips, and by accelerating additional features in hardware, features that average consumers were unlikely to request and hardly needed for mainstream consumer graphics applications, games that is. Had the competition only been between 3Dfx and SGI this model could have continued to work for a long time; 3Dfx was targeting the gamer, and SGI targeting professionals. However, there were many other players ready to enter the market and soon 3Dfx lost in the competition and went out of business.

As graphics technology progressed, the difference between the professional solutions and cards available in the cheap consumer market became increasingly narrow. As a result many SGI engineers left for other companies and SGI locked themselves into a very narrow market segment of professionals with extraordinary needs, and today SGI is just more or less a provider of specialized workstations. Other vendors focusing on the professional market such as 3Dlabs has recently shifted focus and begun to turn towards the consumer market to seek new revenue as the professional market narrows and becomes less profitable. Meanwhile, consumer graphics providers such as ATI and NVIDIA have begun to release professional versions of their consumer line cards offering some extra pro features, hoping to capitalize on the higher margins in this market sector.

The trend I'm trying to highlight here is that the traditional professional and consumer markets are converging. Not only are today's consumer cards just as or even more capable than their professional counterparts, but some professional card manufacturers are adjusting their product line to fit the consumer market. One can be fairly confident that these markets are soon to merge, most likely as soon as within a couple of years.

## Converging of hardware and software rendering

The converging doesn't stop at merging market segments. There's another subtle converging trend going on that will affect the future of graphics. So far this text has only addressed hardware accelerated rendering, but there is also a large software rendering industry out there. Traditionally software renderers, modeling applications and animation software has been built for artists and used for movies, commercials, art and cut-scenes in games. The typical user of such applications has not been programmers, but rather artists, and the real-time graphics world has been built around the programmer while the offline software rendering world has been built around the artist. The connection between the software and hardware world has traditionally been through a professional graphics card. These applications have typically performed significantly better on professional cards, which were built for accelerating wire-frame models. As these applications have evolved, as well as consumer graphic chip technology has, these applications no longer require professional cards to perform well. In fact, many of them perform better on typical consumer level hardware. Meanwhile game engine technology has also progressed and the benefits of using modeling applications in the process of game creation have become evident. Also, as budget and time constrains has often hit developers in the game industry many have gone from developing their own applications to using professional modeling software. These factors has emerged a shift in focus on the parts of the producers of these software. While a strong focus on movie production is still very present there is also an increasing amount of focus put on game developers' needs.

Meanwhile, the real-time world has shown an increasing interest in effects common in the offline rendering world. After the latest crop of graphics cards added floating point processing throughout the full fragment pipeline "Cinematic Rendering" has become the buzzword of today. Many people still laugh at the idea of real-time rendering of movie contents due to the sheer amount of data and processing required and the fact that movie contents are today often rendered on huge expensive server clusters, yet often requires hours or even days to render short scenes. However, many are also opening their eyes for the possibilities that real-time rendering of offline digital contents are not as farfetched as it may appear, and more importantly, all the possibilities and implications such a change would have. Many artists today prefer not to use traditional brute force methods of rendering such as ray-tracing, but prefer simpler algorithms that produce similar quality in a fraction of the rendering time. Similarly, many of the effects previously only seen in movies are now possible to render in hardware as the hardware has grown infinitely more flexible over its previous generations. This new flexibility hasn't passed the offline rending world unnoticed, and recently Alias|Wavefront released Maya 5.0 which features a hardware rendering option. While the quality of such rendered content does not match that of the software renderer this move does however confirm the trend of a long-term converging process between the hardware and software rendering worlds. The idea of being able to replace those multimillion dollar server clusters with a set of graphics chips or even a single tiny graphics card to do the work at the same speed feels too good, and people from both the software and hardware worlds are obviously very interested in the prospects of such a transition.

This merging process is relevant to my work in two ways. As an employee at ATI Technologies I would obviously represent the traditional hardware/real-time rendering world.

ATI however recognizes these trends and isn't late to catch on to trends that might affect their future. ATI has initialized a couple of related research projects and one such the development of an API to bridge the gap between these worlds. This API, called Ashli (Advanced Shader Language Interface) provides tools to deliver shaders written in the RenderMan language commonly supported in the offline rendering world in a format useable for the hardware, such as through the GL_ARB_fragment_program and GL_ARB_vertex_program extensions to OpenGL or pixel and vertex shader version 2.0 in Direct3D. Part of my work was to evaluate the usefulness of this effort and how this would best fit into the toolset of game developers and where the focus should be in future development of this API. This will be discussed in detail and the conclusions to be had will be covered later on in this writing.

If a transition from software rendering to hardware rendering of cinematic content is ever going to happen new ways of rendering need to be invented to replace some of the commonly used algorithms in offline rendering. Some algorithms are simply unsuitable for hardware acceleration. Algorithms that are by its pure nature sequential, or rely on random data access or require global scene information in ways that can't be pre-assembled into easily accessible data structures such as textures etc. are hard to transfer to hardware rendering directly. These elements in the rendering process need to be replaced with new methods that line up well with hardware rendering. One thing to remember is the motto of the art of graphics: If it looks good, then it is good. This principle which has been practiced in the real-time world for a long time, where everything is approximations anyway, can just as well be applied to cinematic rendering too. The eye is easily fooled, and if your lighting model looks good, no one's is going to notice that the light doesn't spread exactly as it real life, or that the reflection angle is off by half a degree. As long as there are no visible artifacts you can create convincing scenes without caring a whole lot about physical or mathematical laws that affects the environment. After all, graphics is only half science, the other half is artistry.

A good deal of time was spent researching and developing ways to create a number of ways to implement a long range of elements applicable to (but not limited to) cinematic rendering. Many effects are usable in real-time applications such as games too. Effects range from effects like soft shadows to simulations of water and glass to post-processing effects such as edge-detection. Techniques that are useful outside the typical artistic field of graphics, for instance in various kinds of industrial applications, were also developed. For instance hardware acceleration of the Hough transform used for image classification. With even more flexible hardware in the future it might be possible to offload image processing step in the vision systems of robots to a graphics chip passing a good deal of CPU power back to the system for other tasks.

## Programmability

Another trend that has been loosely touched in the text above is that hardware has continuously revolutionized its world in more or less every generation. A number of important steps have been taken. In the fragment pipeline for instance the step have been taken from gouraud color interpolation, to texturing, to multi-texturing, to texture combiners, to fragment shaders and finally to floating point fragment shaders. It's similar for the geometry-processing pipeline; from vertex processing being completely done in software, to triangle setup engines, to hardware accelerated transform and lighting and finally vertex shaders. The steps you usually talk about in the progress of a certain feature is from first not being supported at all, to a fixed function implementation, to a configurable implementation, and then to a programmable solution. The border between a configurable and a fixed function implementation is fuzzy and hard to define, similarly for the border between a configurable implementation and a programmable one, so often the configurable step is omitted and people talk only about fixed function and programmable.

What is the difference between these terms? Basically, a fixed function implementation implements a set of predefined functions. A typical fixed function feature is the so-called transform and lighting engine that was first introduces on consumer level hardware on the first generation GeForce cards. The transform part can be fed with a matrix, which defines the transform. The incoming vertex will be multiplied with this matrix in hardware before being passed to the clipping and rasterization. This allows vertices to be stored in local onboard memory and avoid having to pass loads of data over the AGP bus every frame and frees up valuable CPU cycles. This is a fixed function implementation though because the hardware can only perform a predefined operation, namely transforming the vertex by the provided matrix. Matrixes are very flexible though, and as commonly known an infinite amount of consecutive transforms can be baked into a single matrix. However, these matrix multiplication transforms are limited to linear transforms in a homogenous four-component space; that is, the operations that can be done are rotation, scaling, translating, skewing, projection etc. There's no way we can do non-linear operations, not even a simple function like multiplying together any of the x, y and z coordinates of the vertex. This limitation can severely constrain what effects can be achieved with such a setup.

Another typical fixed function feature is texture coordinate generation, or often called texgen for short. Normally textures attached to a surface will be laid out linearly over the area, so it often makes sense to automatically generate the texture coordinates linearly from the vertex positions through a matrix. However, we're again limited to linear transforms, something that will limit the usefulness of this feature.

Both features mentioned above, and many others, were replaced by the so-called vertex shaders introduced on the GeForce 3 card. This was the first programmable geometry processing implementation. A vertex shader basically is a small program that is executed by the graphics hardware on each vertex before it's passed to the clipping and rasterization stage. Now we are no longer limited to simply being able to perform linear transformations on our data, we can do just about any math we can think of, as long as we're not overflowing the amount of instructions the hardware can handle.

Regardless if you're using fixed function transform and lighting or a programmable vertex shader you are likely to benefit from having your vertices stored in local graphics memory as mentioned above. At the time the work on this master's thesis began a new OpenGL extension that deals with handling vertex and index buffers had recently been ratified by the OpenGL ARB (Architecture Review Board). This extension, GL_ARB_vertex_buffer_object, replaces earlier vendor specific extensions such as GL_ATI_vertex_array_object and GL_NV_vertex_array_range with a vendor independent interface. At my arrival this feature had just about been implemented in the drivers and I was assigned to write a sample application that would work both as a useful example application for developers but more importantly as a stability, conformance and performance test for the driver with regards to this feature. This is the only part of the work that does not directly involve shaders, though it is relevant for vertex shading as vertex shading without the ability to store vertices onboard is generally less than useful given that the whole reason (or at least a very significant reason) why this task was put on the GPU as opposed to the much more flexible CPU was to avoid the AGP bus bottleneck. The result of this work will be discussed later on in this text.

It’s quite obvious that most features that make sense to make programmable will sooner or later end up in a programmable fashion. With today’s hardware there are programmable vertex shading parts and fragment shading parts. Some features however such as tessellation and primitive creation, blending, texture filtering etc. are still fixed function. I foresee that programmable primitive processors will be added in the not too distant future replacing the fixed function versions of today, such as tessellation through n-patches, rt-patches and even displacement mapping. Blending may in the future be integrated in the fragment shading pipeline, software developers would love to have it that way, though hardware vendors are somewhat hesitant about doing so because of the implications it has for the hardware.

Hardware programmable shading is a fairly new concept in the graphics world and there are still tremendous progress in extending the flexibility and capabilities of these shading units. The majority of the work done in this thesis consists of writing these said shaders and developing various kinds of effects that they can be used for, both for industrial use, digital content creation, for games production or just plain amusement; after all, graphics should be fun. Some of the work was limited by the hardware capabilities of today, and I’ll elaborate on the prospects for the future.

## Buffer handling with GL_ARB_vertex_buffer_object

### *Background*

Drawing geometry in OpenGL has always been very flexible, and still is. Geometry can be provided the API in pretty much any way one may want, interlaced or non-interlaced, immediate mode or vertex arrays, indexed or non-indexed and finally the option of packing function calls into a display-list. The philosophy in OpenGL has always been: give the developer the choice of which method to use, and leave the hard work onto the driver. There is usually a simple method (in this case immediate mode) that may not offer optimal performance, and a less simple method (in this case vertex arrays) that has some performance advantages. In Direct3D on the other hand the philosophy has generally been that to provide the developer with a one size fits all way of doing things. This means that you won't have as much freedom as you will in OpenGL and some artificial restrictions has been enforced by the API, but on the other hand you can be assured that this path will be properly accelerated, which may not always be the case in OpenGL given the multitude of different ways you can do things. OpenGL has also always abstracted away as many hardware details as possible. OpenGL is only concerned with proper operations, not whether things are executed in hardware or software. Direct3D on the other hand will generally only allow hardware-accelerated features (with a few exceptions such as the vertex shader). These differences have a number of implications, for instance that it's generally a less labor-intensive task to write a proper Direct3D driver, but generally much more work is necessary to write a proper Direct3D application, which has caused OpenGL to generally be the API of choice for learners and for research work. OpenGL is also extensible meaning that hardware vendors are free to add features to the API through a simple extension mechanism so long as it doesn't break compatibility with the core API.

The status of OpenGL as being the simpler API has lately been reduced slightly by the large amount of vendor specific OpenGL extensions that has been provided in OpenGL, sometimes forcing developers to write many separate paths for doing the exact same task in order to support all graphics cards on the market. As graphics hardware technology progresses, new extensions need to be brought to the API in order to expose new features of the cards. Given that vendors are not always in sync with features and product cycles sometimes vendors have to add extensions specific to their newly released hardware. If another competing vendor, who later on supports a similar feature set, decides not to implement the same extension because of technical or legal reasons diversity in the API occurs. Such a thing happened to the oh-so-important vertex buffer.

NVIDIA originally introduced the GL_NV_vertex_array_range extension to deal with the task of storing vertex buffers in onboard memory. The extension provides an interface to allocate onboard memory through an OS-dependent API, and accompanying extensions such as GL_NV_fence were provided to enable syncing operations. This extension is quite low-level. The developer will have to deal with synchronizing the application and hardware itself. The API was also quite cumbersome because of the OS-dependency for memory allocation, and because it is prohibitory expensive to switch between different buffers, forcing application developers to store all its data in the same buffer. This latter restriction forces developers to know on beforehand how much memory it will need, which will unnecessarily increase application complexity. It also severely limits the flexibility and may cause applications to use more memory than it need. Many developers expressed their dissatisfaction over this extension and even some NVIDIA engineers themselves expressed their desire for a better interface.

For these technical reasons, as well as legal reasons, ATI choose not to implement this extension and opted to create its own extension, namely GL_ATI_vertex_array_object. This extension provided an easy interface to upload vertex buffers into local memory and provided a new set of entry points for setting current arrays. The driver automatically handles all synchronization. Overall, this extension provided a much nicer interface than GL_NV_vertex_array_range and has generally similar performance characteristics. However, this extension did not provide an easy interface to directly access vertex buffer data. This is not so much of a deal for the vast majority of the applications, but for a subset of applications that deals with large amounts of dynamic data this is a major drawback. This problem was solved by adding another extension on top of GL_ATI_vertex_array_object, called GL_ATI_map_object_buffer that provided an interface to lock a buffer and retrieve a pointer to it. The driver still automatically handles synchronization for us.

The disadvantage of these extensions is that you need to call separate functions to set current arrays than if you are using system memory vertex arrays. While it's not a particularly hard task to support both VAO (Vertex Array Object) and system memory vertex arrays it can still be somewhat cumbersome to not have a unified interface for accessing both array types and you need to update your code at two different places for every change you make in order to maintain support for both interfaces. Also, this non-orthogonality against the core spec also means that the extension may require to be updated in order to be compatible with other future extensions. For instance was the GL_ARB_vertex_shader extension ratified later on and introduced the concept of vertex attribute arrays. There was no entry point for setting vertex attribute arrays with VAO, which meant that an additional extension, GL_ATI_vertex_attrib_array_object, had to be provided that did nothing beyond adding just that entry point.

In the middle of this mess is the (typically) disgruntled developer who has to deal with a bunch of different interfaces that works differently in order to support NVIDIA, ATI and other vendors' cards. As the calls for a unified interface for onboard buffer grew louder ATI and NVIDIA began co-developing a specification for an extension that not only were supposed to provide an unified interface, but also solve all the problems with earlier vendor specific extensions. On February 12, 2003, less than a week before the work began, the GL_ARB_vertex_buffer_object [3] extension got its ARB approval.

### Interface

The GL_ARB_vertex_buffer_object extension provides a simple interface to create, upload and access vertex arrays similar to that of GL_ATI_vertex_array_object. Creating and uploading a vertex array is as simple as this:

```
glGenBuffersARB(1, &vertexBuffer);
glBindBufferARB(GL_ARRAY_BUFFER_ARB, vertexBuffer);
glBufferDataARB(GL_ARRAY_BUFFER_ARB, size, vertices, GL_STATIC_DRAW_ARB);
```

When setting an array current you'd normally do something like this for setting the source of the vertex stream:

```
glVertexPointer(3, GL_FLOAT, stride, basePointer + offset);
```

When using VAO you'd have to call another function:

```
glArrayObjectATI(GL_VERTEX_ARRAY, 3, GL_FLOAT, stride, vaoBuffer, offset);
```

Preferably you would want to be able to use the same function call to set the array current regardless of if you're using system memory arrays or are storing vertices in onboard memory. The fundamental problem is though that the glVertexPointer function and its siblings as defined already in the OpenGL 1.1 specification take a pointer. Basically it combines two pieces of information as shown above, both the base address of the vertex array and the offset into this array at which the parameter in question is stored. When using buffers onboard you again have two pieces of information, the buffer handle and an offset into this buffer. However, the glVertexPointer et al entry points only have one parameter, and there's no general method to combine a handle value and an offset, unlike a pointer and an offset that can simply be added together. In the VBO (Vertex Buffer Object) extension this problem was solved by letting the application set a current buffer handle. The pointer argument is then used as an offset into this buffer. This way it is easy to support both drivers that support the VBO extension and drivers that don't. A typical call would then look like this:

```
glVertexPointer(3, GL_FLOAT, stride, offset);
```

A typical call that supports both VBO and system memory arrays would look like this:

```
char *basePointer = (supportsVBO? NULL : vertices);
glVertexPointer(3, GL_FLOAT, stride, basePointer + offset);
```

Similarly, when a drawing call uses indices:

```
  char *indexPointer = (supportsVBO? NULL : indices);
  glDrawElements(GL_TRIANGLES, nIndices, GL_UNSIGNED_INT, indexPointer +
offset);
```

VBO treats the buffer handle 0 as a special handle meaning that system memory arrays are being used. With this arrangement you can easily just store either a valid handle plus an offset in your data structure (in case you're using onboard memory), or you can store the handle zero and the base pointer + offset (in case you're using system memory arrays). This way the application will just have to access these two parameters (four if you're using indices) and pass them to the API like this and it will just work either way:

```
glBindBufferARB(GL_ARRAY_BUFFER_ARB, vertexBuffer);
glBindBufferARB(GL_ELEMENT_BUFFER_ARB, indexBuffer);
glVertexPointer(3, GL_FLOAT, stride, vertexPointer);
glDrawElements(GL_TRIANGLES, nIndices, GL_UNSIGNED_INT, indexPointer);
```

## *Test-case*

The driver had at this point in time an early implementation of this new interface. To ensure proper operation of this driver in terms of performance, conformance and stability a sample application was to be written for testing. This sample was also aimed to be a good reference illustrating the usage of this extension for developers.

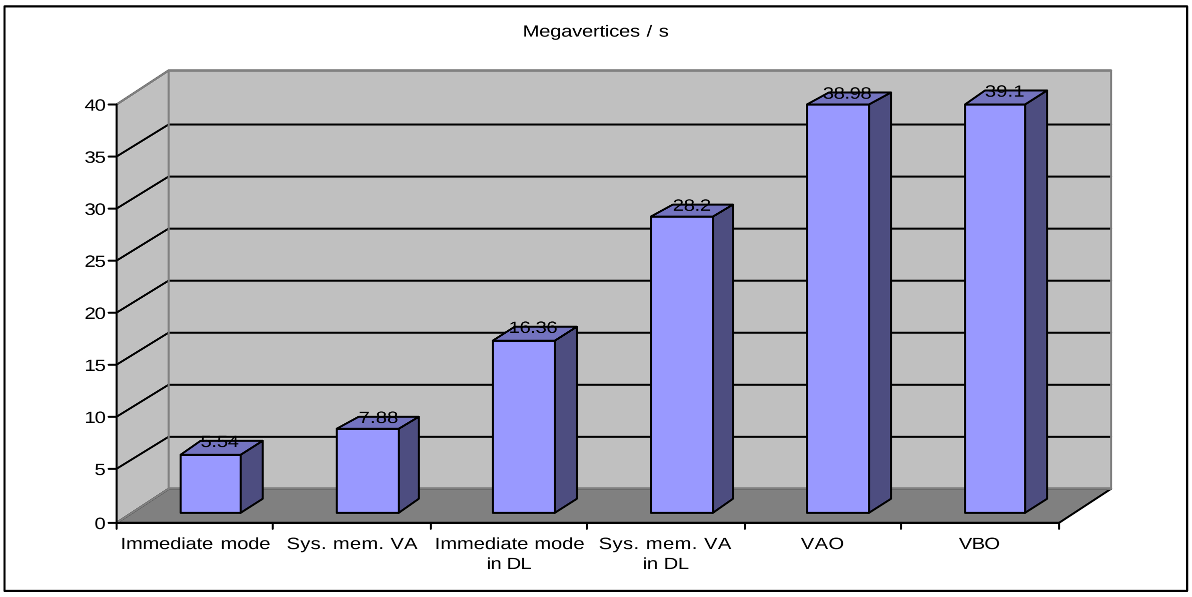
To test the conformance in a useful manner a general data structure that can cope with any valid vertex format had to be defined. For this structure to be useful a way to create such a structure from a model file had to be implemented. For this implementation .3ds model files were supported. The .3ds format however does not support anywhere near all the permutations of vertex structures possible, actually only a very limited subset, so it was necessary to add functionality to dynamically adjust formats, change parameter types, sizes etc. of the vertices. A general way to convert any format to any other format, such as changing float type to integer or unsigned bytes to doubles, had to be implemented. This problem was solved by using a general converting routine that interprets a source pointer given a provided source type enumerator, expands it to a double, optionally scales and biases the results, and writes it to a destination pointer in the specified destination type. Given that this is only a preprocessing step, something that won't affect final performance, this generic (but somewhat slow) implementation suffices. Another routine was added to index up a non-indexed model. Additional routines were provided to remove attributes from the array to be able to test performance when providing a small set of data in the vertex stream. For instance when providing nothing but vertices (no normals or texture coordinates or anything such, just plain vertices), with all forms of lighting turned off and the model is ensured to be out of the view frustum, the transform rate reached should be close to the theoretical maximum of the hardware.

Not all types of attributes are natively supported by the hardware, so it's important to look at performance figures to see whether there are any suspicious drops in performance when using native formats. The Radeon 9700 for instance does not support normals as integers or bytes natively, so the driver needs to first convert such attributes into a format the hardware can read. In such cases it is normal that the performance of using onboard buffers actually is lower than if you're just using plain old system memory vertex arrays. That's because the driver will have to download the vertex array over the AGP bus (unless it has a cached copy in system memory), convert it, and then upload it back over AGP. While performance can be very low under such circumstances there is no way the driver is allowed to fail, generate incorrect visuals or otherwise misbehave.

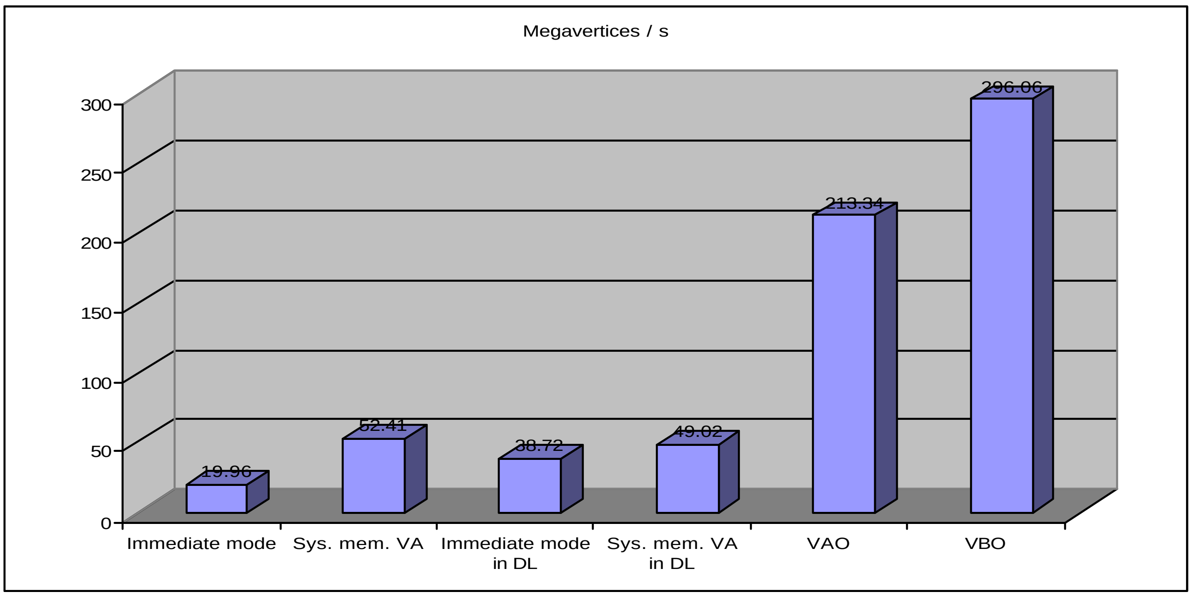
Finally, there is a large number of ways to provide OpenGL with geometric data; and performance and conformance needs to be ensured using all possible permutations of API usage. One can either use immediate mode or vertex arrays. In both these cases one may also opt for packing the drawing commands into a display list. This allows the driver to cache the data and may be able to apply some optimizations if the application behaves well. Then there's the option of using VAO or VBO. These may also be packed into display lists; however, it's unlikely to provide a measurable performance increase, if not actually a slight decrease. It's an unlikely usage scenario, so no work was put down into testing this case. There is also a possibility to use mixed mode, grabbing attributes from active arrays by indexing them one by one, optionally mixing it out with fixed function calls to provide additional attributes or additional vertices. This legacy API however is not a very interesting usage scenario and is quite unlikely to appear in any new real world application being developed today. This was not tested. For all vertex array modes there's also the option to either provide indexed or non-indexed geometry. Further, the vertices can be provided in a huge amount of permutations of types and sizes and indices can be provided in three different kinds of types. On top of all this the geometry can be supplied as many different kinds of primitives, triangles, quads, strips and fans and so on. All this adds up to an enormous amount of permutations, and realistically not all could be tested. Instead a selected subset of tests was run from most run-modes and comparisons were made between them.

## Benchmarks

The first test run presented here is with three fixed function lights and with indexed vertex data. The results look as follows:



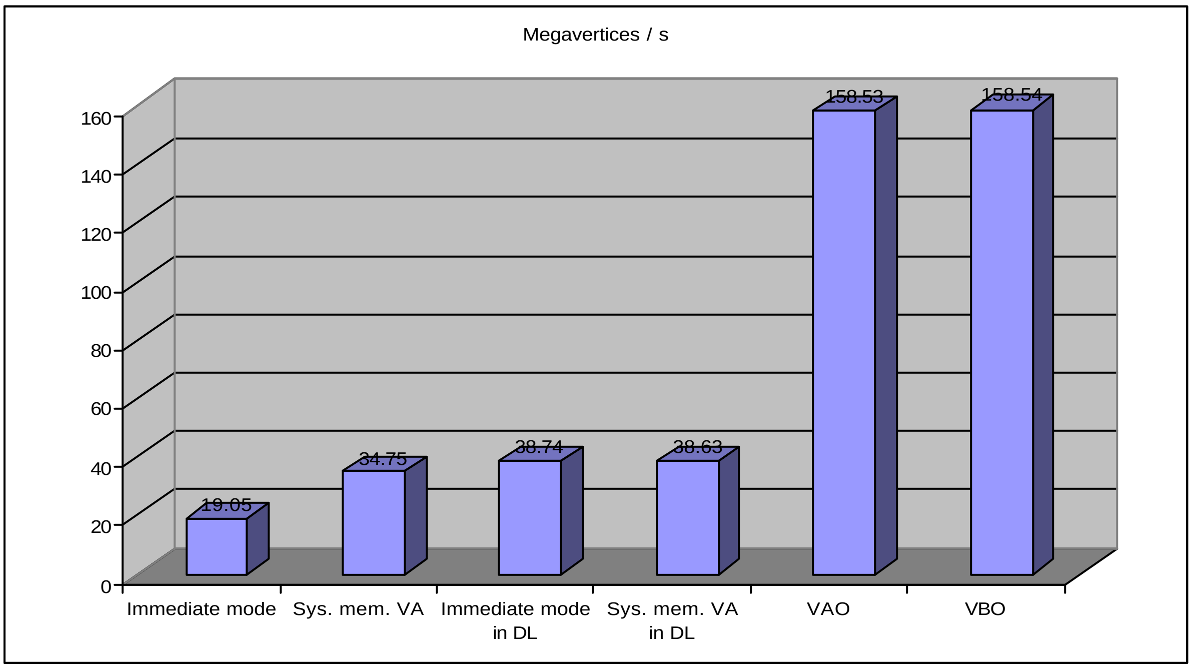
The results are pretty much as expected. Immediate mode is slow running, vertex arrays somewhat faster, display lists adding a good deal of additional performance and finally VAO and VBO fastest with little difference between the two. The VAO implementation however can't store indexes in local video memory, unlike VBO, which fully uses the video memory. One may have thought that this would make a larger difference than the 38.98 MV/s for VAO and 39.10 MV/s for VBO. At this rate chances are that things are shader limited though. By reducing the load this theory can be checked. This was done by completely removing the lighting and thus just outputting plain white. These are the results:



These results are closer to that expected. Here the VAO and VBO are much faster than any of the other methods. VBO is also a good deal (39%) faster than VAO under these conditions.

It may be somewhat unexpected that system memory vertex arrays are slightly faster than those packed into a display list. This doesn't need to mean anything though, especially since the difference is quite small.

Finally another test was run under the same conditions, though with non-indexed vertex data. The results are as follows:



The results are pretty much as anticipated. Now VAO and VBO have virtually identical transform rates, as should be. Immediate mode, within a display list or not, shows minimal change. Indexing the data makes no difference for immediate mode since it's expanded on the CPU side anyway. There's no difference from the driver's point of view. Vertex arrays, which benefits greatly from indexing, shows a clear performance reduction as non-index data is used instead.

## *Evaluation of the driver implementation*

A large number of additional tests were also performed beyond those reported above, though not documented and not reported here. The major reason for this is that it would take a lot of time to document all these cases tested, they were mostly just quickly run, checked and found to work fine, then left for the next test. The actual results of each individual test aren't all that interesting. In the overwhelming majority of the cases the results were as expected. Only in a few cases was there unexpected behavior, which typically didn't consist of low performance figures but rather conformance bugs. These bugs usually occurred in uninteresting corner cases that will hardly ever appear in real applications, for instance when passing data in odd formats such as normals as integers, colors as triplets of bytes etc. These anomalies are bugs nonetheless and were reported to the driver team. Other bugs found were random crashes with VBO when using large amounts of geometry, system freeze with batches of more than 65535 indices of unsigned short when using video memory indices, and no output when using a complex double-sided fixed function lighting model in conjunction with a large number of lights. The last problem was found to be because the Radeon 9700 doesn't have any separate fixed function transform and lighting unit, unlike some other hardware, but rather implements the fixed function transform and lighting through the vertex shader. This works fine in most situations, except a few extreme circumstances where the number of instructions required exceeds what the hardware can handle.

A number of the bugs were confirmed and fixed during the time at ATI, for instance the overflowing vertex shader which was quickly fixed by appropriately forcing it into software mode when the hardware is insufficient. This is of course much slower, but it’s the only correct and conformant behavior in this situation. At the time of the completion of this report it is expected that most if not all of these bugs are fixed, in some cases months ago. Either way, the driver implementation of VBO was despite the bugs mentioned found to be very complete and functional for the at the time young age of the code in question. At the time of this writing the extension has existed close to half a year. By now the extension is very stable, which have been concluded by a number of other applications of the author utilizing the VBO extension that all works with no problems.

# High level shading in OpenGL 2.0

## *Background*

When OpenGL was first introduced in 1992 the goal was to create an open API for high performance 3D graphics. Much of SGI's previous experience from IrixGL went into the API and it was formed to be very forward-looking. At the time of introduction it was leaps and bounds ahead of any existing graphics standard out there. Having been defined long before the ages of hardware acceleration of 3D graphics in the consumer graphics sector it is amazing how well it has served us for a long time even till these days. The fact that the API has never had to be changed in such a way that it broke compatibility with older applications or created problems for new hardware has proven the strength of the original design philosophies of OpenGL. Comparing it to the history of Direct3D - a much younger API - puts it into perspective. Not only have Direct3D been more or less completely revised many times over, it also frequently breaks compatibility and constantly changes the API. Only in the latest revisions has the API begun to settle. True enough, the run-time libraries ensure run-time compatibility with older applications, but upgrading application code to a higher Direct3D version is generally not a quick task. OpenGL tend to be designed with the philosophy, "don't assume anything about the hardware", while Direct3D tends to be designed around the hardware that exists on the market today, something that has caused trouble with newer hardware from time to time. Once upon a time it was for instance common that hardware stored textures and the frame-buffer in a simple linear scan-line format. Given this fact, Direct3D exposed a way for applications to directly access such memory. Hardware of today doesn't work that way though, which caused problems and forced these access methods to be removed in later revisions of the API. Textures are typically swizzled to get better memory access patterns and frame-buffers are often divided into tiles to work better with hierarchical depth buffers or other optimization algorithms.

Being well defined from the start however doesn't save us forever. While the age of the API has shown through at times any problems arising on the way have so far been possible to work around, but now things are at a stage in time where the API no longer leads the hardware, but instead the hardware constantly mandates new updates to the API, which has led to an infinite amount of OpenGL extensions being introduced the latest years. A number of revisions have been made to the core API and at the time of this writing version 1.5 is the latest. However, these are not forward-looking, but rather just updates to let the core API catch up with the mainstream graphics solutions out there. In 2001 a project was initialized at 3Dlabs to aim for another major revision of the API, a forward looking approach like the original API was, a new API that would deserve the 2.0 nomenclature. 3Dlabs presented their work and visions to the ARB and sought feedback and ideas. At first feedback was unanimously positive and it was thought that the process would be fairly quick. However, as time has gone parts of the work has turned into political debates over details such as whether the shading language should target the underlying hardware directly or if it should target a middle layer extensions providing assembly level programming models, whether some features really are all that useful or if it will just bog the driver down. It is expected though that the final OpenGL 2.0 specification will be ratified this year, but it's likely that some features of the original proposal will be omitted.

While there are loads of interesting ideas covering many different topics it is undoubtedly the high-level shading language that is the most interesting part of the OpenGL 2.0 effort due to its central role in the rendering process. The last two hardware generations we have begun to touch on the subject of real-time shading. One once had to deal with setting up zillions of states and was limited to a small subset of operations. Not only was it very limited in what effects could be achieved, but it was also far from a convenient programming model. When fragment and vertex shaders were introduced the programming model first shifted to a configurable model, where a large range of operations were possible, but later changed into a more convenient assembly language format. As is quite well known, writing small applications or small routines in assembly language works fine, but no one is writing full applications in assembly language today. People use high level programming languages such as C/C++. Only in special cases such as in device drivers, OS kernels or optimized media processing routines do people resort to assembly language. Similarly, for short shaders that maybe just combines a few textures or applies a fairly simple lighting model it will work just fine to use an assembly language. However, as shader complexity grows it will become just as unbearable as writing full assembly language applications is today. Even when using assembly it is realistic no one does that simply because the gains are too few, if there are any at all, and holds little weight against the loss of productivity. Shaders will definitely go the same way, and after having tried high level shading languages out for a few months I'm ready to ditch assembly languages for graphics forever.

The task was to evaluate the OpenGL 2.0 shading language, called glslang. At the time of this work ATI had a driver implementing the subset of the 1.01 shader specification that could easily map onto the Radeon 9700 hardware. The conformance and stability of this driver was to be evaluated and feedback provided for further development.

## *Porting from DX9 HLSL to OpenGL 2 glslang*

ATI provides a wide variety of sample applications on their website for developers to look at as a guideline for various techniques and implementations. One such dev-sample, with the very imaginative name HLSL_NoFX [4], was to be ported to the GL2 shading language. This sample implements known techniques, which were previously implemented in RenderMan, but has been ported by other ATI engineers to PS2.0. The sample simulates no less than 9 different kinds of materials.

### Wood

The sample illustrates the process of implementing a wood material in several steps. The final results look like in the picture. The first step is to create a simple ring pattern. Trees grow with one ring each year and these rings create a circular pattern that alters between a bright color and a darker color. Depending on what kind of tree we're talking about, the climate and other factors these colors differs. The natural thing is to feed the shader with three constants, a dark and a light color and frequency at which we want the ring pattern to occur. The vertex shader will then pass the position of each fragment along to the fragment shader through a texture coordinate. Since position is a linear attribute the exact position will be available for each fragment in the fragment shader despite the interpolation.

The cylindrical distance is the interesting part here. Assume that the z-axis is placed along the tree trunk. The distance of interest is the length of the (x, y) vector. That is, z is simply ignored. To get a ring pattern this distance is multiplied with the frequency and mapped into a 1D texture. This 1D texture contains a pulse function that can be chosen pretty much arbitrarily. It only needs to be continuous, tileable and preferably (though not necessarily) map to the full [0, 1] range. This number that is fetched from the pulse function is then used to linearly interpolate between the dark and light colors passed to the shader. At this point there should be a pretty good ring structure around the z-axis. In real world applications though one would typically want to move around objects around and have them in different orientations. So transformation on attributes where direction matters is desired. For rings direction does matter. A transform matrix will thus be passed to the vertex shader that it can then rotate the position with before it passes it to the fragment shader. This will be referred to under the rather arbitrary name "shader space". The matrix thus transforms the position into shader space. In this sample the shader space is simply a copy of the model space, but is lockable so that you can let it stick to the object while moving around.

There are a few mistakes done by the original author of this sample that time was taken to correct in the OpenGL 2 version. Many of these mistakes were repeated in several shaders. The unfortunate backside of going to a high level shading language is namely the same as with high level of abstraction in other areas, namely that while it certainly increases productivity it also adds a dose of sloppiness on its users. The fact that shaders that try to create marble contain variables named liteWood and darkWood would hint that there has been some sloppiness involved. I have the highest respect for the original author of this sample though. It could be assumed that he just didn't walk the extra mile to polish the details, but one can be confident that he very well knows the lack of perfection of his work; especially since some of the constructs from the original RenderMan shaders, which made sense there due to how the language is defined, that the sample was originally ported from remains more or less unchanged in the final shaders. RenderMan shaders do not separate the concepts of vertex and fragment shaders. A RenderMan shader is roughly only a fragment shader that assumes that a certain set of inputs is readily available, such as the position and the normal. This means that when you write a RenderMan shader you will do all work in the shader. When the original author translated these shaders to a vertex and fragment shaders he left some work to be done in the fragment shader that should preferably be done in the vertex shader or sometimes even on the CPU. One such detail was present already in the very first simple rings shader. The cylindrical distance is obviously not a linear operation in homogenous space, so that math needs to be done in the fragment shader if you don't want to approximate. The multiplication with the frequency however can easily be moved out to the vertex shader since scaling a vector before taking the length of it gives the same result as scaling the length of the vector. Thus the vector can be scaled with the frequency in the vertex shader before passing it to the fragment shader.

Why is it preferable to have this work done in the vertex shader instead of the fragment shader? The first and most important reason is performance. In the vast majority of the applications out there the application has to deal with orders of magnitude more pixels than vertices. Assume even a moderately low resolution of perhaps 1024x768 is being used. This amounts to 786,432 pixels. A typical application also has a fair amount of overdraw, though recent innovations such as Hierarchical-Z reduces the impact of that, but in general one can be assume a 2x or higher pixel load due to overdraw on typical immediate mode rasterizers. That would leave us with roughly 1.6 million pixels to draw per frame. A normal real-time application has a few thousands polygons, but lets for the sake of comparison assume we will have a vertex load of even 100,000 vertices / frame. Assume this is rendered on a Radeon 9700. We then have 8 pixel pipelines that can ideally perform one 3-component vector operation and one scalar operation each every clock-cycle. This means that if this work is done in the fragment shader it will take 1,600,000 / 8 = 200,000 cycles to complete this task. The Radeon 9700 has 4 vertex shading units that can carry out one operation each cycle. This means that it will take 100,000 / 4 = 25,000 cycles to perform complete this task in the vertex shader, a factor or 8 times less work to be done. In typical applications the ratio is even higher, often much higher, especially nowadays when using resolutions such as 1600x1200 is not uncommon.

There is a general rule here. If it's possible to perform the work in the vertex shader as opposed to the fragment shader, then do so. Also, if you can preprocess any work done in either shader on the CPU instead and pass it as a shader constant, then do it. The general idea is that each API call will produce many triangles, and every triangle will produce many pixels. This tends to hold true in the vast majority of the applications out there, but there are of course exceptions. If an application uses extreme amounts of polygons then moving work to the fragment shader can be beneficial. For instance if you have a multimillion triangle mesh to draw it can be beneficial to use Appearance Preserving Simplification [6] or other techniques to reduce vertex shader workload significantly, though at the cost of a higher fragment shader load. It should be said though that such techniques do not really do exactly the same work, but the final image looks very similar. No such work was done while working for ATI, but I have at a later time evaluated similar techniques [7] and been able to produce performance increases of between 2x and 4x by applying such techniques and there is potential for even higher increases.

The next step in the process of creating a wood material is to apply noise. In real wood the rings aren't really 100% circular; there are usually many imperfections. Without these imperfections the image simply screams artificial. It doesn't look particularly real. So kind of noise needs to be used in the fragment shader. Noise is a whole topic of its own however, so if you're not familiar with Perlin Noise I advice you to read about it Appendix B where this particular topic is covered. A short explanation though is that Perlin Noise is a continuous range-limited and band-limited random function. Given a coordinate in a space of arbitrary order it generates numbers in the [-1, 1] range. Common implementations are 1D, 2D and 3D. For this shader tileable 3D noise will be stored in a volumetric texture and it will be sampled according to the fragment's position in shader space. This noise will then be used to offset the fragments position before creating the rings. The original shader samples the textures at three different positions to create the offset vector. This is not optimal, unless you're trying to reduce storage space. It was opted for to simply use a three-channel (RGB) texture instead and storing three different noise functions. This way it's only needed to sample the texture once. As today's hardware store RGB8 textures as RGBA8, but just ignores the alpha channel, the alpha channel was filled with another noise function since it's for free anyway and might be useful later on.

The next step is to slightly wobble the fragment position along the z-axis. The idea is that trees seldom grows straight up in the air, but tend to bend slightly back and forth. This step doesn't add a whole lot of value to the visual impression though and may optionally be excluded. What is done is just to apply noise once again, though this time we won't change the z coordinate. One can then let the artist adjust the amount of wobbling vs. offsetting.

At this time we should have something looks fairly close to wood. As a final touch though, and to get a realistic impression, some basic lighting is applied. It doesn't matter much what lighting model is used, so a standard Half-Lambert diffuse plus Blinn per-pixel specular was used like in the original shader.

## Ivory

Ivory doesn't have a very complex appearance. It is beige or white and if polished it's quite shiny. There's not a whole lot to say about it. Applying a simple lighting model, optionally use a beige-looking base texture, and we're basically done. The original shader raised the specular component to a power of 64 by repeatedly multiplying it with itself. On a CPU this would have been a good optimization. On a GPU on the other hand it's just less readable and actually slower since GPUs don't implement full-precision powers anyway, but rather a table based approximation, and are thus able to perform the task at the same performance as three standard arithmetic instructions. Thus the built-in power function was used for the OpenGL 2 implementation.

## Velvet

Velvet is an interesting material. Its base appearance is typical diffuse. On top of that you have what the original source calls retro-reflective lobe. It sounds overly sophisticated but basically just means a more or less standard specular model. While velvet can appear shiny its shininess is not really very similar to usual specular, so this component is usually fairly low. Instead another kind of shininess that resembles the behavior of velvet more closely will be used to give it its main sheen. Unlike many other materials velvet appears shiny when viewed at a sharp angle, pretty much regardless of incident light direction. This is called horizon scattering.

Since high sheen is desired where the surface normal faces perpendicularly away from the view vector and no sheen when they are parallel to create a horizon scattering effect it would appear that it's the sine of the angle between these vectors is what to look for. The cosine of the angle between two vectors is, as is pretty well-known, the dot product. To get the sine of the angle one can simply use the trigonometric identity, $\sin^2 x + \cos^2 x = 1$. This number can then be raised to a power like with typical specular to set how glossy appearance we want. Combining diffuse, specular and horizon scattering and the appearance will be quite similar to velvet.

The original implementation made one small mistake. Instead of using a specular exponent or shininess, it uses a "roughness" variable. Roughness is basically just the inverse of the shininess. This means that it raises the specular component to the inverse of the roughness. In this shader however the roughness was declared directly in the shader, so it's likely that the compiler will pre-compute the exponent and there won't be any difference. If however in the general case the shader were to be fed with a surface roughness it would be better if the shader was provided with the shininess instead of the roughness to avoid that the same division having to be performed in the shader for every pixel; especially since on some architectures there is no native support for operations on two constants, which means such an operation would even have to be expanded into two operations, one which copies a constant into a register and the other one which performs the actual inversion.

## Marble

Marble has a typical semi-random look, so it should come as no surprise that Perlin noise is being used as the main appearance-defining factor. Marble usually has lots of fine detail due to its composition of fine crystalline limestone generally mixed up with impurities of other kinds of rock. The geological details are pretty irrelevant, but it should be stated that the appearance this creates with a mix of large and small details closely lines up with turbulent Perlin noise. Turbulence is another useful noise tool that's covered in the noise chapter in Appendix B.

To create a marble look one basically just need to adjust the range of the noise and assign it some color. For this sample blue-tinted marble will be used, thus the noise is mapped into a texture containing a blue pulse function to get smooth transitions. Finally it's polished up with some standard lighting. As marble is generally shiny a fair bit of specular lighting is used.

The original implementation had a few mistakes that were fixed for the OpenGL 2 version. First of all it did some work in the fragment shader that's better suited to do in the vertex shader. This time however it has more implications than in the wood shader above. In this shader the texture coordinates are manipulated in the fragment shader before accessing the noise texture. There's nothing inherently wrong with manipulating texture coordinates as a part of an effect, but there are several reasons why you shouldn't do that if it's not necessary. In this case, it isn't necessary in any way. The texture coordinates are simply scaled in the shader, twice for that matter, but this work could be done in the vertex shader. First of all, as explained above, the performance is reduced as additional load is stacked on the already most burdened processing unit. On top of that a normal texture sampling is changed into a dependent texture sample. This creates two problems. First, performance is generally reduced. Why? Because when being fed with a texture coordinate to access a texture directly from the hardware can easily predict and pre-fetch texture samples that will be used later on in impending pixels. When the shader manipulates the texture coordinates it is in the general case impossible for the hardware to look into the future and will have to resort to educated guessing or try to reduce the impact with heavy caching of neighboring texels.

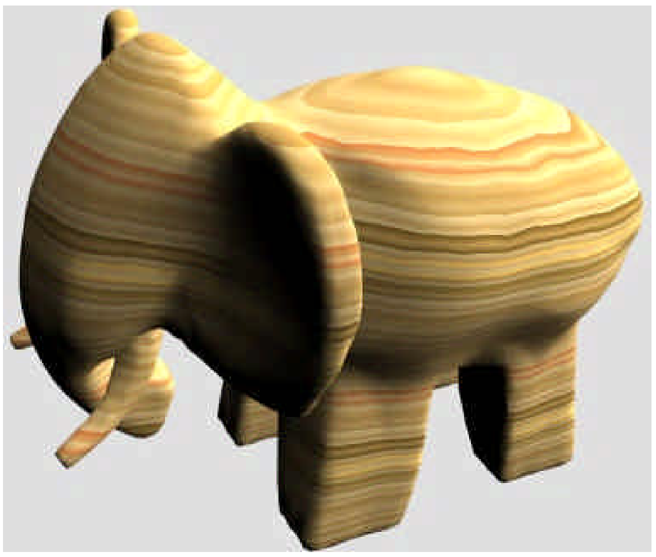
This unpredictability may also have implications on image quality. Texture sampling generally faces the problem of having to deal with the size of a fragment in texture space. When texture coordinate interpolation is performed all parameters are known and the hardware can correctly determine the size of the fragment in texture space and can choose the two closest mipmap levels to sample from and interpolate to give a very good approximation of what the texture coverage of the base texture would look like had it been accessed and perfect coverage been used. Since the shader creates unpredictability it is very hard to correctly determine the correct mipmap to use. The hardware has a couple of options to choose from. The first option is to just always sample from the base texture. This way no detail is lost, but very likely a ton of aliasing is added. Another option is to just use whatever mipmap level was chosen from the texture coordinate interpolation and hope for the best. In many cases this works fine depending on the access pattern and the original texture data. In fact, in most legitimate uses of dependent texture reads you can get away with such an implementation. The third option is to use complex hardware to try to approximate the variation of the texture coordinates after the shader has manipulated it. It is possible to approximate the first order partial derivates in screen-space by sampling the values of a certain register in different pipelines and measuring the difference. This is exposed in for instance the shading model available in the GL_NV_fragment_program extension through the ddx and ddy instructions, but there's no guarantee that the hardware will use this automatically for texture lookups. For this to work and to get both the partial derivate with respect to x and the partial derivate with respect to y you need to arrange your pipelines in a 2x2 manner to begin with. In other words, you need to group pixels in 2x2 block when you process them to get a meaningful semantics for the x and y partial derivates. This in itself is not much of a problem since most hardware is already processing pixels on a tile basis. However, it requires that pipelines work in sync with each other. This is not a problem today, but will be in the imminent future. Already in the next generation it is expected that hardware will be able to execute data dependent branching in the fragment pipeline. Data dependent branching means that different pipelines may branch differently and there's simply no way that synchronous operation can be guaranteed. This will make the semantic meaning of ddx and ddy instructions fuzzy. One could do explicit syncing on such operations, but it's unclear how to interpret the semantics when different pipelines execute such instructions different number of times. There's no doubt however that ddx an ddy are very useful tools for the cases to which it applies, but the point is that there's no guarantee that this will be done for you even on capable hardware, and you'll most likely have to do that math yourself if you wanted to correct mipmap selection on dependent texture reads. It should also be said that ddx and ddy is only an approximation of the derivates. The point of all this is that if you can sample a texture directly from the interpolated texture coordinate, you should do so. There are both quality and performance to be gained from doing so.

## Granite

Granite is a typically dull material. It looks fairly random, so again Perlin noise will be of good use. It's non-shiny so only diffuse and ambient lighting need to be used. It's fairly dark but contains some contrast. To achieve this effect a simple operation of |0.5 – noise| is all that's needed to create the base material color, and then the lighting can be added on that to finalize the picture.

## Stratum

Stratum is fairly simple to create. There are a number of differently colored layers that's for most part clearly separated horizontally. Basically, what's needed is to simply apply a 1D texture according to the y coordinate. To get a less artificial look some noise is used to slightly offset the coordinate before looking up the color in the texture. Since stratum isn't shiny in the least only ambient and diffuse lighting is used.



The original implementation basically repeated the same few mistakes as with marble, adjusting texture coordinates in the fragment shader when it's perfectly possible to just do that math in the vertex shader and sample directly from the interpolated coordinate. Other than that there's not a whole lot to comment on.

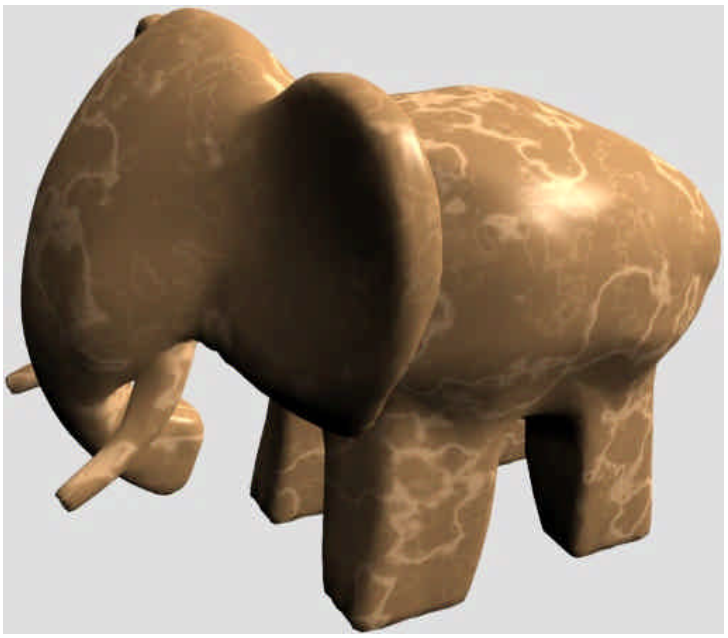
## Saturn

The Saturn shader tries to imitate the look of the Saturn surface. It's for obvious reasons best applied to spherical objects, though in the sample application a model of an elephant was used for all shaders, even though it doesn't make much sense in this particular case. Saturn has a lot in common in its look with stratum, which by the way hardly applies to elephant models either. The Saturn is striped due to its mostly gaseous substance and a fast rotation around its axis, which splits gases into bands according to their density. There are two significant differences between the Saturn look and stratum though; Saturn is mostly symmetrically mirrored around its equator and it's spread roughly according to latitude rather than linearly. Instead of correctly calculating the latitude the original author decided to use another model that behaves roughly similar. One may guess that his reasons were that since he was using noise again to add some realism chances were that he would end up with a negative number under the square root and get very odd results close to the poles. However, one could have just clamped to zero or taken the absolute value instead. Though as the alternative model works just fine too, and may actually be slightly faster, it was used in the OpenGL 2 port too. The alternative method is simply raising the y coordinate to a power of our choice to get a more non-linear spread. Once the coordinate is settled it is addressed into a Saturn colored 1D texture. Then some standard diffuse lighting is applied on it and it's all set.

Again the original implementation does things in the fragment shader that belongs to the vertex shader, however this time one cannot really sample directly from the texture coordinate unless there is a fair amount of tessellation on the model. The spread is not linear. It could however be worth investigating in applying a mirror-once wrap mode and lettings the texture contain the colors pre-spread in a nonlinear fashion. No work was spent on this however as the task at hand was just to port the application. It should also be said though that mirror-once for some obscure reason is not widely supported on graphics cards out there.

## Veined marble

Certain kind of marble has veins due to impurities being mixed up in layers due some obscure geological processes that are not in the scope of this thesis. However, veins in marble are fairly common, which makes it an interesting effect to simulate. How can one create veins? Veins to begin with behaves fairly randomly, twists and turns a little as it goes over the surface, so once again noise is an important component. To create veins one can simply choose to be on a vein whenever the noise value is at a certain level, say 0.5. Why does that work? Think of the 2D case; place the noise on a 2D texture and imagine that it's a height-map. In other words, the value at each pixel says how high it is, 0 is low and 1 is high. Imagine that we would make some kind of terrain from this and look at it from above. Now say we would highlight where the height is 0.5, like is done on a standard map



you use when you go orienteering where you might have one line for every 5 meters height difference, except that we only highlight one height here. What would it look like? Well, it would be lines going a little randomly but fairly consistent over the surface. Basically looking like normal veins in marble.

In real life though, one cannot just select 0.5 like that, sampled texels values are likely to never be exactly 0.5, so a little more flexibility is needed. Something like $0.45 < x < 0.55$ can't be used either since that would case a lot of aliasing. Instead a smooth transition depending on how close to 0.5 we are is desired. For this the so called smoothstep function will be used, which essentially is a normal step function, except that it doesn't abruptly go from 0 to 1 at a certain point, but instead smoothly and seamlessly shifts from 0 to 1 between two points. Details behind this function are provided in Appendix B. Depending on how large veins we want the range of change can be adjusted and depending on how sharp we want the transition to be the result can be raised to a power of our choice. To get better results this process can be repeated with higher frequency noise and scaled down accordingly to get finer details a couple of times until satisfying results are achieved. The final result can then be used to interpolate between a marble color and a vein color to get the final veined marble color.

Again the original shader had a few faults. Again it needlessly manipulates the texture coordinates in the fragment shader. It also raised the specular component to the power of 64 by repeatedly multiplying it with itself instead of properly using the faster pow function. These problems were fixed for the OpenGL 2 version. However, other problems arose during this work. First a couple of odd bugs caused the application to crash when the shader was written in certain ways. It turned out that the built-in smoothstep function in the GL2 shading language would not accept a 1 as its second parameter. Any other number such as 0.9999 or 1.0001 worked fine though. Until that driver bug was fixed a workaround was implemented, namely to simply use 0.9999 instead. Another problem arose soon thereafter. The driver at this point did not support loops in any shape or form, not even static for-loops. Though the hardware does not support branching the driver should be able to expand static for-loops, but that was not yet implemented. This shader used a static for-loop, which the driver wouldn't accept. The workaround was quite simple though, the for-loop just had to be expanded manually. The complexity of this shader along with the nowhere near final status of the GL2 driver caused the shader to exceed the native resources of the hardware however. Though the driver didn't pass back information about what resource it exceeded it was a rather obvious guess that it's the instruction count that went above what the hardware supports. Cleaning up the code a little, removing a few unnecessary details and reducing the complexity of the lighting model to gain some space for a few more instructions and the sample would finally run.

## Depth of field

Another dev-sample to be ported is an implementation of the depth of field cinematic effect [5]. Depth of field is an effect that tries to capture the behavior of lenses, such as in cameras or even our eyes. A lens can only focus at one depth at a time. The farther an object is from the depth of focus, either close to the camera or far behind the focal plane, the blurrier it becomes. The math behind this optical behavior is quite complex however, so one has to settle for an approximation. That shouldn't turn anyone down though; the general rule in graphics still applies: If it looks good, then it is good. This technique will give results that look fairly good. The procedure is explained in detail in presentation [9].

The effect is implemented by rendering the scene to a texture. The depth of the scene is rendered to another texture along with a blur factor. This blur factor is just the difference in depth between the fragment and the focal plane, scaled with the focal range. With these three pieces of information, the scene, the depth and the blur factor, enough material is provided to approximate a depth of field effect.

The scene as rendered to the texture is a sharp image. It needs to be blurred according to the blur factor to get the out-of-focus effect. A simple blur filter can be implemented by taking a number of samples randomly spread in a circle. The kernel size of the blur filter will be adjusted according to the blur factor. More blur means larger filter kernel size. This alone is a fairly good approximation of depth of field already, however, there's one problem left to solve. Using a simple blur filter like this will cause sharp foreground objects to leak into blurry background objects. This is not desirable and doesn't happen in similar images produced by real lenses. The solution is to depth-compare samples to scale back the contribution of samples that are likely to cause leaking.

The shader takes one sample at the fragment position and 12 peripheral samples from its neighborhood. The center sample is given a contribution of 1 and each of the other samples is given a contribution of 1 if they are farther away than the center fragment, otherwise its depth. All samples are averaged according to their contribution summing up to the final output color.

The shader was implemented taking a sample at the center position, and then a static for-loop took care of the calculation for each of the other samples. As was mentioned earlier in this text the GL2 driver did not support loops in any shape or form, so the loop had to be manually expanded. The Radeon 9700 only supports 64 ALU instructions in the fragment pipeline, so with 12 samples it should be fairly obvious that you're quite tight on your instruction budget for each sample. Five instructions per sample would leave you with four instructions for initial setup and assembling the final result. Five instructions are needed for that work. However, it is possible for the compiler to combine the assignment of the constant 1 with the first addition of the contribution of the first loop as it expands it to save one instruction. If the compiler is able to do that, five instructions per peripheral sample will be available. That is also exactly the amount of instruction that we need. This puts some serious quality demand on the compiler. It needs to be able to produce the exact optimum code from our shader. Fortunately, the DX9 HLSL compiler is able to do that with the original DX9 HLSL code. It may be a little too much to hope for that the alpha GL2 driver would be able to perform as well, though one could always cut back the number of samples until it works though, with lower quality of course. As it turns out though, the shader would fail for other reasons.

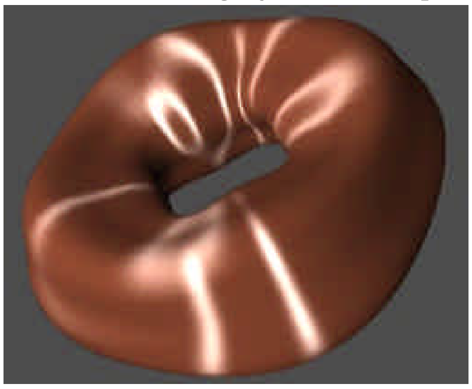
The Radeon 9700 has a limitation in that it can handle at most a 4 level texture indirection chain. Basically, if the shader samples a texture and then uses this value in the texture coordinate calculation for another texture, it has a texture indirection. If that sample value again is used for another texture coordinate, then you have two levels of indirection. The driver had a problem in discerning when there is a direct dependency between different texture lookups and tends to fail to load some shaders even though they are perfectly valid for the hardware. This in not a problem for GL2 shaders only but also affects shaders written in low-level assembly language. Either way, no matter how the statements were arranged the driver rejected the shader. It basically limited us to 4 texture samples, even though there was only have one level of indirection. The hardware should be able to sample 31 samples under such circumstances. Since both the scene texture and the texture with the depth and the blur factor needs to be sampled we are left with just being able to use two sample positions, the center sample and one peripheral sample. Later drivers improved it slightly, so that the first pair of texture lookups that read directly from texture coordinates wasn't counted against the indirection count, which would allow us to use another peripheral sample position, but the full problem was not solved during the time at ATI. Due to this problem, this sample had to be left in an almost finished state, though not quite working. With only three samples used the quality was not anywhere near the original D3D sample, but the output was good enough to conclude that the shader was otherwise working and produced expected results.

## *Porting from RenderMan to OpenGL 2 glslang*

The RenderMan shading language has long been the most commonly used standard for describing surface appearance in the offline rendering world. As been discussed in the preface of this text ATI has put down some serious effort into evaluating the trends of the merging sectors of graphics and figuring out its role in all this in the long term. As such ATI has started a number of internal research projects meant to evaluate these trends, and one such project is Ashli, which will be discussed later on in this text, an interface that can aid in transferring typical offline rendering content to the real-time rendering world. The task was to manually transfer a couple of shaders used in this project that was written in the RenderMan language over to the OpenGL 2 shading language. This helped in forming an opinion on how future development of Ashli should be directed given the difference between how a human would port such application compared to the automated process in Ashli. The conclusions will be discussed later on in this text.

### Brushed metal

A brushed metal shader was chosen to be ported. A screenshot of the sample is provided in the illustration. Brushed metal has an appearance that falls under the category of anisotropic lighting [10]. Anisotropic, which is the opposite of isotropic, basically means it's not uniform. Some materials, such as brushed metal, hair, satin, CDs etc., has directional grain. Its lighting is not only dependent its normal, but also on the rotation around the normal. There are several different models of anisotropic lighting. This shader uses a fairly simple model that applies light according to not only the surface normal, but also according to the tangent. The tangent can be thought of as the direction of the grain.



The lighting model used applied standard diffuse and ambient lighting. The specular however is applied according to the tangent and not the normal. The model also doesn't apply the specular according to the usual Blinn or Phong models. Instead it evaluates sine and cosine of the angle between the tangent and the view vector, the sine and cosine between the tangent and light vector, multiplies the sine components together as well as the cosine components, sums it up and clamps negative numbers to zero. The cosine of the angle is as known from linear algebra the dot-product between the vectors. The sine can then be evaluated from the trigonometric base theorem, $\cos^2 x + \sin^2 x = 1$. The resulting specular component is then raised to a power as usual.

### Flame

There was an interesting flame shader among the shader collection for the Ashli project. It rendered some animated fiery imagery. The usefulness of this effect can be questioned as it is a 2D image space operation. However, there are many scenarios where such shaders still suffice. Distant fires, partly occluded fires or fires in inaccessible parts of the scene geometry are cases where it would still work. 2D fire effects made by procedural textures generated on the CPU has been successfully used in many games and can certainly add a lot to the mood despite being far from accurate. A fire generated in the shader has an advantage though over procedurally CPU-generated textures in that it doesn't add a lot of burden on the AGP bus and that it unlike texture uploads is asynchronous and thus allows better parallelism between the CPU and the GPU.

It should hardly come as a surprise that Perlin noise is an important part of the receipt for fires. The typical fire behavior is somewhat chaotic, and has an upward movement while slightly changing appearance on its way up. The bottom of the fire is typically lighter than the sparkles, so obviously height is an important attribute. A good start in the process of creating a fire is to map the vertical position into a color, the bottom of the fire should map to white-yellowish and the top to dark-red and finally black. The original RenderMan shader does this by mapping into a spline function. There's no direct equivalent to a spline function in the GL2 shading language, but as it is a constant spline it can easily just be pre-computed and stored in a 1D texture. To get the chaotic look we of course bias the height with some noise. In the RenderMan language there's native support for noise functions, and the GL2 language contains similar constructs, however, it's not supported in hardware and thus will have to be implemented in software. The alpha GL2 driver didn't do that however, and frankly it's not desired to run in software mode either, so a texture containing a tileable noise function was used instead. To get the appearance of an upward movement the noise was let to slide upwards. As the image space is 2D it would seem a 2D noise would be enough. However, it would look very unrealistic and just like some sliding image moving up if it was done that way. As mentioned above, the appearance of the fire changes during its movement from the fire's base to its top. This is done by adding another dimension to the noise function and letting the image space slide in that dimension.

The original implementation had some obscure constructs that I frankly didn't fully understand the purpose of. Instead of letting the image slide upwards in a linear fashion it used a power function. While this worked fairly well as the animation starts it has the effect that the longer the animation goes, the slower it moves and the more compressed the fire looks. In the Ashli framework where the time variable was looped over a fairly short amount of time the effect of this didn't get too large, though a small hiccup was evident every time the loop restarted. In the port of this shader where the time was left to continue looping the effect became very apparent as the time was left to pass. Though this was slightly disturbing it was left the same way in the port. Additional research on developing procedural fires was instead done at a later time, which is presented in later on in this text.

## *Evaluation of the OpenGL high level shading language*

After having worked with both OpenGL 2 high level shading language glslang and the DirectX9 high level shading language (which will be used in the research work later in this text) there are several conclusions to be had on the language and its future potential. The first conclusion is that the language itself leaves very little left to be desired. It basically has all one may wish for as far as current and immediate future technology goes. Undoubtedly there will be extensions needed in the future to expose new hardware features, just as there have been plenty of extensions to the OpenGL 1.x API, but it is my belief that this language spec in itself is a strong basis to stand on for the future.

## Direct shader to hardware mapping

The OpenGL 2 shading language takes a slightly different approach than the DirectX 9 HLSL in many areas. The first and maybe the most significant difference, which does not really affect the programmer much but rather is a significant difference for the IHVs (Independent Hardware Vendor), is that the OpenGL 2 shading language directly targets the underlying hardware. DX9 HLSL on the other hand compiles shaders into assembly language targets, those as specified in the various pixel and vertex shader versions. This means that the driver never gets to see the high level language code and high level semantics are consequently lost. Instead of living with the limitations of the actual hardware with DX9 HLSL we live with the limitations of the specification of the assembly language. While the DirectX9 approach requires less work on the part of the driver, and thus may benefit vendors from a cost perspective, it does however remove plenty of optimization abilities and restricts innovation. It is also true that a shader that is optimal on one vendor's platform may not be optimal on another's. For instance do the GeForce FX series cards support arbitrary swizzles in the fragment pipeline, while for instance the Radeon 9700 series only support a limited set of common swizzles natively. In DX9 this meant that the ps2.0 fragment shader target was defined to only support the limited subset of swizzles that the Radeon 9700 supports. The effect of that is of course that the GeForce FX will have the restrictions of the Radeon 9700 artificially enforced on it. Assume for a while that the ps2.0 target would have specified fully general swizzles ala GeForce FX, as does the OpenGL equivalent, GL_ARB_fragment_program. What would that mean for the Radeon 9700? It would mean that swizzles that aren't natively supported would have to be expanded into several instructions. This would cause performance to be significantly reduced. There are plenty of smart tricks that can be done with general swizzles to improve performance and cut down instruction count, but such efforts will have the opposite effect on the Radeon 9700. So what is optimal on a particular architecture may not be optimal on another. So who's supposed to decide what's optimal? In the DX9 HLSL case it is up to Microsoft to decide what is optimal. However, the hardware vendors themselves obviously know best what works best on their hardware; certainly better than any third party who will always have to settle for various kinds of generic optimizations. That's the philosophy behind the design of the OpenGL 2 shading language. The driver gets the high level shader and it's up to the drivers to make it fit as good as possible onto the underlying hardware. This also gives more room for IHVs to innovate since they aren't bound by a low level specification but rather are free to design the hardware anyway they see fit according to the likely usage scenarios of the hardware. The backside of this is of course that driver complexity increases.

## Hardware abstraction

Another difference is that in D3D the vertex and fragment shader operates independently of each other. A certain vertex shader may for instance be used with many different fragment shaders, or vice versa. The same is true in OpenGL too using the GL_ARB_vertex_program and GL_ARB_fragment_program extensions. In OpenGL2 however, a vertex and fragment shader is compiled and linked together into the same program object. This makes shaders go together in a 1:1 relation. One may call this an artificial restriction, but in practice it actually makes a lot of sense. The typical usage pattern is to write vertex and fragment shaders that go together. Recycling shaders are of limited use, it typically only applies to shaders that are very similar, and for those cases there's nothing keeping you from loading the same shader into many different program objects.

What's the advantage of linking together the shaders then? The first thing is that it allows us to abstracts the hardware to a higher extent than what DX9 HLSL does. In DX9 HLSL you for instance assign each parameter passed from the vertex shader to the fragment shader a semantic, such as primary or secondary color or a texture coordinate interpolator. If you're linking a vertex and fragment shader together this is no longer a need to assign semantics to the interpolators (or varyings as they are called in glslang) since the driver can assign it for us. All you need to do is to come up with a sensible variable name for the parameter in question and the rest is automatic. In DX9 HLSL you'd have to ensure that you're assigning the same semantics to a parameter is both the vertex and fragment shader; otherwise you might be up for a long debugging session and a lot of frustration (trust me). Another thing is that the driver can detect if there are any paths that aren't used. For instance if the vertex shader writes to a certain interpolator, but the fragment shader never reads it, then the compiler can safely optimize that write away. In DX9 HLSL it couldn't do that since it wouldn't know which fragment shader the vertex shader were going to be used with. This kind of information can also be passed back to the developer assisting him in debugging and tuning his shaders. Also, if a fragment shader reads from an attribute that the vertex shader never writes to the driver can detect that and issue a warning.

## Hardware limits

There are unavoidably certain kinds of limitations in hardware. When writing a shader in assembly language the resource usage is typically easy to count. We have a limited instruction count, limited number of interpolators, limited number of texture image units, limited number of temporary registers etc. These limitations are typically to be queried by the application to ensure it's not going to spill over any such resource limits. When moving to a high level shading language however the resource usage becomes compiler dependent and may be hard to count. It is for instance very hard to know how many native instructions a high level shader will expand into. When a shader compiles to a target shader version in DX9 HLSL and the compilation is successful you can be pretty much assured that it will continue to be within the limits since you'll find the same compiler on all client machines, at least until the next revision of the runtime libraries comes around. In the OpenGL 2 shading language where the compiler changes between different hardware vendors and even between different driver revisions for the same vendor it is simply impossible to know whether a shader will stay within the boundaries for some resources. Traditionally OpenGL has always provided a guarantee that things will always work as long as the API is correctly used. There's no guarantee that things will be executed in hardware however. Arguments have been made that this convention should be continued in the OpenGL2 effort. John Carmack, the lead programmer at Id software, the man behind many popular games such as the Quake and Doom series and widely recognized authority in 3D graphics had this to say about this question:

*"I do need to get up on a soapbox for a long discourse about why the upcoming high level languages MUST NOT have fixed, queried resource limits if they are going to reach their full potential. I will go into a lot of detail when I get a chance, but drivers must have the right and*

*responsibility to multipass arbitrarily complex inputs to hardware with smaller limits. Get over it."* [11]

In general I personally agree with Carmack in this question, but not everyone is as committed to enforcing a fully general limitless model. The problem is of course implementation complexity. Letting the driver map a complex shader into a multipass session on hardware with limits is far from a trivial. There is always the possibility however to just ignore that and let shaders that overflow the resource limits to go into software mode. Not everyone likes that his or her shader might end up un-accelerated though and want to be able to know whether a shader will end up running natively or not. On the other hand, there's nothing guaranteeing that a shader that's within the limits will run in hardware either since there may be language construct that are not natively supported either or there may be special cases that the hardware can't handle. As of the shader specification version 1.05 some hard to count resources, such as instruction and register count, cannot be queried and the API effectively ensures infinite such resources. Other easier to count resources on the other hand, such as the number of interpolators and texture images that can be bound, are fixed and the limit is retrievable. It is valid for the driver to reject shaders that exceed such limits. It should be noted that there are some recent innovations, such as the F-buffer that was implemented in the Radeon 9800, which makes it possible for shaders to overcome hardware instruction and temporary limits, thus making it possible to run arbitrarily complex shaders in hardware. Details about the exact implementation of this are at this point sparse however.

## Language

When it comes to the actual language itself there's generally not a whole lot to comment on. Syntactically there aren't a whole lot of advantages of one language over another in general. If it looks similar to C or similar to Pascal or something else isn't really an issue, as long as it's more or less feature complete and not littered down with constructs with questionable usefulness. This can be said about both the OpenGL 2 shading language and the DX9 high level shading language. Working with either is a blessing over having to deal with older assembly languages. But as with the assembly languages, where OpenGL has had an advantage in ease to use, OpenGL 2 holds a slight edge in a few issues that actually matters. I'm talking about semantics. The DX9 HLSL is quite liberal in what constructs are valid while glslang may be somewhat restrictive at times. This has its good and bad sides of course. In DX9 HLSL it is for instance valid to assign between a four-component vector and a float without a typecast. This is valid both ways, either writing a vector to a float or writing a float to a vector. As long as you're writing correct code the DX9 HLSL code may at times look slightly cleaner due to this. However, it might not have been so much of an issue if DX9 HLSL didn't also make it so easy to use an unintended variable type. The types are float, float2, float3 and float4. Mixing up floats and vector types happens all the time, and since there's no need for typecasts in the majority of the cases the compiler tend not to be able to catch these mistakes. It's so easy to just write "float" when you mean "float3" and vice versa. In glslang on the other hand the types are called float, vec2, vec3 and vec4. This way vector and floats are clearly separated and mistakes aren't as common. When mistakes do happen they are generally quickly found when the compiler generates an error on invalid assignment. Experience can tell that this save in debugging time and frustration, which can often be significant, is well worth the slightly less clean look of the code. Typecasts have as much place in a high-level shading language as they have in high-level languages for CPUs.

## Ashli

### *Introduction*

Today as hardware has grown increasingly more complex the transition from offline software rendering to hardware rendering even for typical offline contents is growing more and more interesting. Steps in the convergence have been taken from the offline rendering industry in adjusting its tools to work better with typical real-time contents and offering support for hardware rendering of some content for quick previews and even do final renders, though at quality much less than that of software renderers. Similarly, the hardware industry has always kept an eye at the software rendering industry and recently is has become increasingly popular to compare hardware rendered content to software rendering. While it's obvious that hardware rendering isn't quite there in terms of quality to fully replace software rendering it is important for hardware vendors to get a good analysis of where in the convergence process we currently are. The potential market for hardware rendering of cinematic content is huge. Chances are that at some point in time high-end pro graphics cards will replace the server clusters that today render high-quality movie contents. It is important for hardware vendors to have their eyes open for when this is about to happen and be prepared for the steps in hardware abilities that are necessary to take in order to get there. This is where Ashli comes into the picture.

#### RenderMan

Ashli stands for Advanced Shader Language Interface and is a simple interface for running RenderMan shaders on today's hardware. RenderMan is probably the most widespread high-level shading language there is and there exists a plethora of RenderMan compatible renderers out there and heaps of tools made to work with RenderMan shaders. There is also plenty of know-how and experience working with this language. One goal of the Ashli project is to take advantage of these resources and leverage it directly to the hardware. Another goal is of course to evaluate how close we're to bringing traditional offline rendering into the real-time world. The assigned task was to evaluate the interface, how it could best fit into a game developer's toolkit, how it may work together with third party tools such as ShaderMan, and provide suggestions on where ATI should point their efforts regarding future development of the Ashli toolset.

### *The interface*

The interface is quite simple; you create an instance of a class, tell it what interface you're using (OpenGL, GL2, D3D9 etc.), feed it with the RenderMan shader and get the shaders in return. Besides the vertex and fragment shader returned you also get the formals, that is, the shader constants with their names and assigned default values. Using these shaders and values you can render equivalent images in hardware as the RenderMan shader would have produced would it have been fed to a RenderMan compatible renderer. To prove this point ATI have produced the *Ashli viewer*, an application that does just that and could be said to be a RenderMan compatible renderer working in real-time, except that Ashli doesn't yet implement the full RenderMan spec.

## State of the interface

At the time of evaluation of the interface the vertex shader retrieval wasn't functional. This isn't so much of a deal for software that is in alpha stage; it did however limit the appreciation of the interface. A test application was written, and this problem had to be worked around by writing the appropriate vertex shader manually. This kind of defeats the purpose of the interface. Ignoring this problem (which will be fixed at some time anyway, if it's not fixed already at the time of this writing), the interface is quite clean, though not entirely obvious for somebody who have never worked with RenderMan shaders before. The documentation was quite sketchy, which is understandable given the state of the software in question, though enough to get going. To fully understand the details however contact with the people responsible were necessary.

Another flaw, though understandable, was that not all targets for the hardware shaders were functional. The OpenGL target seemed to work flawlessly, the DX9 target too mostly, though in a few cases using DX9 caused incorrect output in the Ashli viewer. Whether this depends on Ashli or the viewer is hard to know however. The high level targets were almost completely broken. GL2 code was outputted, though invalid, non-functional and would not compile. But then again, OpenGL 2 is not finalized yet at the time of this writing, so that no large effort is put into GL2 at this time is reasonable. DX9 HLSL code could also be outputted, though with similar results as with GL2.

Finally, Ashli does not yet support the full RenderMan specification. Most of this is because of hardware limitations or other constraining factors. In other cases it's just that support has not yet been added but is planned. Most of the common functionality is supported however such that most RenderMan shaders works with Ashli.

## Evaluation of the interface

While writing the test application only one thing to object on was found, namely the way formals are provided to the application. This is done through a long text string. The application then needs to parse this string and extract the information it needs. A cleaner interface would in my opinion be to provide the application with the information in the actual source formats, such as floats and integers. It was argued that the string approach was more platform-independent and thus the interface could be used in an application where the Ashli was used in a compilation server, something I personally found to be a very unlikely usage scenario and argued that the interface should be constructed for the common case, which is that the Ashli interface is compiled directly into the tools and applications where it will be used. For command-line tools it may also be better with text output however. It was also argued that parsing the string hardly is a complex task, and that the hour or two that may be required to write the parser will be just noise on the surface when put into the perspective of the full application development time. As a counterargument the discouragement of having to spend more time than necessary just to get going may turn some developers away from the interface at a very early stage; patience is sparse in today's world. The discussion is mostly over subjective matters however; so two compromise solutions were suggested that would satisfy both parties. The first solution was to use a binary Ashli interface, but for the platform independence for server applications the server application may convert this into a text instead of having the Ashli interface do this. The other solution would be to provide a simple parser implementation with the Ashli interface in a sample application that goes along with it.

## *The game developer’s perspective*

Can a tool for translating RenderMan shaders into hardware shaders be of any real use for a game developer? At first glance it may not seem so. After some thought however a number of possible usage scenarios surfaces. After all, game developers frequently use offline rendering software for modeling, level construction, previewing of game contents and other game development related tasks. There are also a number of game genres in which performance is not a particularly important factor and thus don’t need highly optimized and specialized shaders, but where developers typically rather has to deal a tight development budget. As such a tool for directly leveraging shaders constructed for game contents in modeling software can be used to let the modeling software completely take over the task of constructing game content such that no separate editors are necessary. This should save some precious development time. There are also plenty of tools around to help deal with RenderMan shaders; editors, graphical shader tree constructers, shader libraries etc. With Ashli these can be utilized directly for gaming content. One such tool that special attention was put onto is a small application called ShaderMan.

### ShaderMan

ShaderMan is a small tool for creating RenderMan shaders, previewing and rendering the content. ShaderMan is constructed such that not a single line of code needs to be written and thus may be used even by people who have little or no programming experience. Instead a graphical interface is used to create a shader tree, not too different from for instance the Maya shader tree. Components such as lighting models, different kinds of operators, material properties etc. are placed on a canvas and connected. From this tree the shader it then constructed automatically by the application. For many artists this can be a more intuitive way to work rather than dealing with writing shader code. If Ashli could be integrated with ShaderMan this tool could be very useful for a developer.

ShaderMan works on top of existing compilers and renderers. For ShaderMan to render any content at all a third party renderer must be installed and configured in the ShaderMan preferences. A number of common renderers are preconfigured on installation time such that they can be used without having to deal too much with settings. Already as it is ShaderMan can be used for game development by exporting the RenderMan shader and use a separate Ashli based tool to generate the hardware shaders. This may be a little cumbersome however and the best thing would of course be if a RenderMan compatible compiler and renderer based on Ashli could be constructed and directly used in ShaderMan. This way quick previewing of the content would also be possible, much quicker than that of typical software renderers one might add, who usually take a good deal of time even with simple scenes. Being unfamiliar with RenderMan renderers, how they work and what interface they use it was not obvious if this would be possible. Playing around with the settings it became apparent that adding a command line compiler based on Ashli would be an easy task. Whether this would link well with a renderer however was unclear. Discussing this with the author of ShaderMan (who was very helpful and explained all the related concepts in detail) and learning more how this all worked it became apparent that it was very possible. It was projected that given a fully working Ashli interface this task would take a single developer a few weeks to complete.

## *The future*

For Ashli to be really useful a few things to focus on for future development have to be highlighted. First of all, the performance factor needs to be considered. Today the code produced by Ashli is far from optimal. The compiler needs to be better at optimizing the code. Furthermore, there is the performance/quality tradeoff factor. When for instance interpolating the normals over the surface the normal needs to be normalized at each fragment. This is not for free however, and in many cases it is more or less redundant, for instance when the model is highly tessellated or if the surface is completely planar, such that the normalization provides no visible difference. One may also decide to use full normalization for high-end machines and skip it for lower-end to gain some performance. Being able to choose whether to normalize normals could be very useful for developers. The same applies to tangent vectors too. There are also many other cases where an approximation could do just as well, or would do for low-end machines at least. If a Hermite or cubic interpolation is used, this might be replaced with a linear interpolation as a performance tradeoff for low-end machines. In some cases a texture lookup could do as an approximation of a complex function and may improve performance. It would be very useful it the Ashli interface provided a way to set what kinds of optimizations one desire to use.

Another interesting factor is the shader readability. The assembler shaders produced by Ashli while being fully function and useable are though hard to read and edit. Sometimes one may desire to add additional functionality on hardware shader level, or tune it manually for performance. This is a prohibiting cumbersome task for the assembly language shaders produced; it would likely be faster to just write the shader yourself directly. For these reasons it's important to put more focus on the high level shading languages. In these languages the code produced should not only be readable, but constructs from the original RenderMan shader should also be easily recognizable. Variables, constants, functions etc. get their names transferred and the connection to the source RenderMan shader is obvious. This way editing the final code should be no difficult and no deciphering of the code will be necessary. Until the OpenGL 2 shader language specification is finalized it may be wasteful to spend a whole lot of time of this code target, but DX9 HLSL it ready, ratified and documented today, so there's no reason why DX9 HLSL couldn't be supported already.

Finally, a number of sample applications showing the use of Ashli needs to be produced to aid developer getting started. A few command-line tools such as a compiler and renderer for ShaderMan and other applications need to be created, and lastly the documentation needs to be completed.

# High level shading with RenderMonkey

## *Introduction*

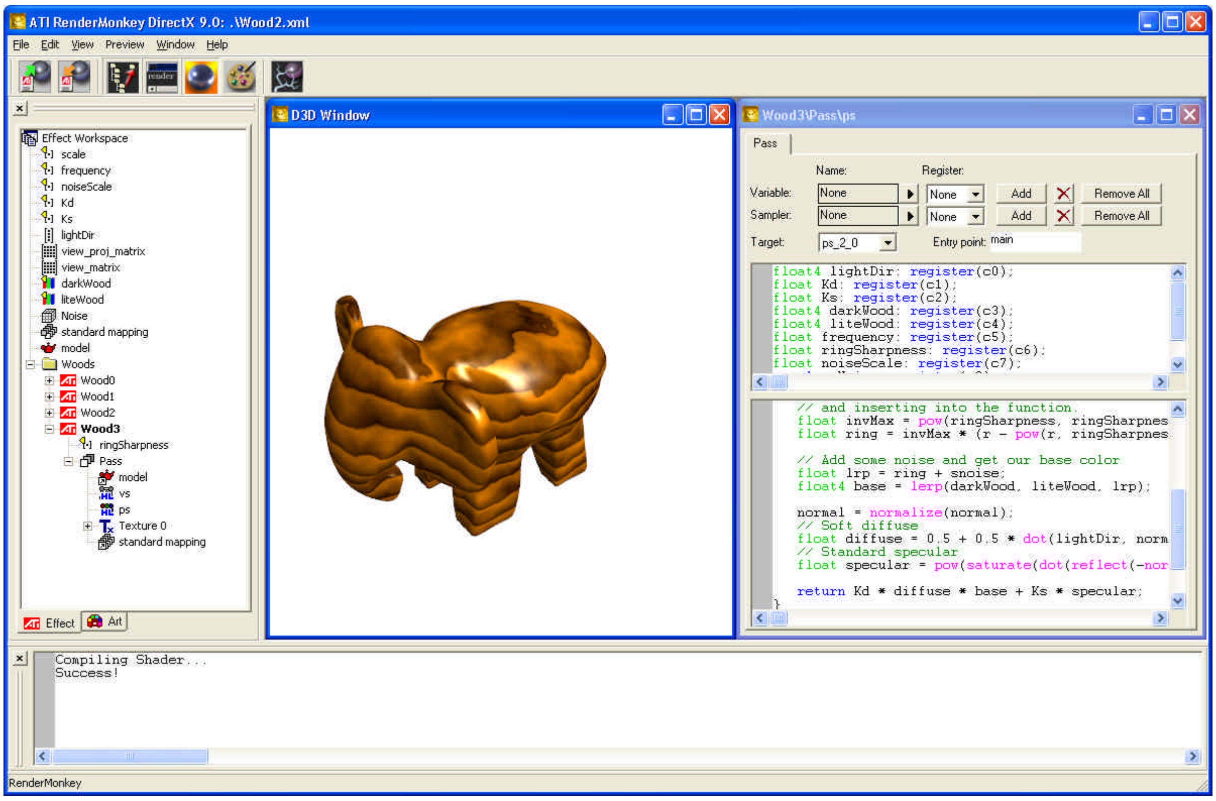
With the introduction of high level languages in graphics the productivity in developing graphics applications got a healthy boost. However, for research work and other kinds of frontline shader development the save may not be all that great compared to total development effort spent. For research work the overhead of having to implement all kinds of peripheral details in order to get the effect up and running may easily take the majority of the time. Research applications illustrating particular techniques are typically small, but things like navigation, performance tracking, API interaction, state handling, OS interaction etc. may take a lot of time away from actual research work. There are some frameworks available, such as glut, that can ease the work when peripheral details are unimportant, but they tend to only take care of a subset of the issues, such as the OS interaction and may not even be that good at it. Other frameworks may take care of more, but may have other limitations. While there are a couple of options for the OpenGL developer the frameworks for D3D shines with their absence. There may exist some frameworks for D3D, but they are likely unknown; it's hard to come up with even a single one, except personal frameworks such as the author's own framework, which never were designed to fill someone else's needs than the designer's. Chances are that researcher will have to write their own framework for their needs. Regardless, even with a full-fledged framework available there is still a good deal of time spent on interacting with the framework, setting up textures, render targets and loading models etc.

Many developers, regardless of if they would regard themselves as researchers or not, will sooner or later find themselves in the researcher's position as they need to prototype different rendering techniques. One often doesn't want to litter down actual game or application code with experimental technique prototypes. One may decide to work on a separate branch of the source tree instead, but the game code may require a significant amount of modification to work with a new technique. If you don't know how well the technique will work you may not want to spend the time modifying a large amount of code, which may in the end be wasted. And if the technique works as hoped, chances are that you may have to spend a considerable amount of time merging the modified source branch with the main source tree.

### RenderMonkey

What we need here is a prototyping tool. After bouncing ideas back and forth to actual developers querying their needs and desires ATI started working on the RenderMonkey toolkit [13]. It is still at the time of this writing in beta and there are plenty of bugs and anomalies left to be ironed out. Nonetheless, it is for most parts a delightful experience to work with the application once you've familiarized yourself with it and know how to circumvent its whimsy habits.

RenderMonkey is built with an UI that's similar to Visual Studio, an environment most developers are familiar with. Techniques are arranged in workspaces and called "effects". Workspaces are stored as plain XML files containing simple references to texture files and other external dependencies. This way workspaces can easily be shared in a platform independent way between developers, though there is currently a number of issues with paths, locales etc. that needs to be resolved. Many effects can be put into the same workspace to compare different techniques for achieving similar output. Each effect has a set of one or more passes. A pass encapsulates a normal drawing pass. Each pass contains a pair of a vertex and pixel shader defining the output. A preview window renders the scene in real-time as you alter objects or commit changes to shaders. It may look like this:



Name: | Register:
Variable: None | None
Sampler: None | None
Target: ps_2_0    Entry point: main

```
float4 lightDir: register(c0);
float Kd: register(c1);
float Ks: register(c2);
float4 darkWood: register(c3);
float4 liteWood: register(c4);
float frequency: register(c5);
float ringSharpness: register(c6);
float noiseScale: register(c7);
```

```
   // and inserting into the function.
   float invMax = pow(ringSharpness, ringSharpnes
   float ring = invMax * (r - pow(r, ringSharpnes

   // Add some noise and get our base color
   float lrp = ring + snoise;
   float4 base = lerp(darkWood, liteWood, lrp);

   normal = normalize(normal);
   // Soft diffuse
   float diffuse = 0.5 + 0.5 * dot(lightDir, norm
   // Standard specular
   float specular = pow(saturate(dot(reflect(-nor

   return Kd * diffuse * base + Ks * specular;
}
```

```
Compiling Shader...
Success!
```

In each workspace you can place a set of objects such as models, textures, render targets, variables, colors etc. Objects are arranged in a tree-view and most objects can be placed at any point in the tree. The tree structure defines the scope of the objects. Each kind of object has its own GUI component that can be used to quickly alter instances of objects and see how that affects the output. Simple scalar variables for instance can be adjusted with a slider and the output is immediately updated thus providing quick feedback to the developer.

## The work

I had the pleasure to work freely and exclusively with RenderMonkey for more than a month researching and developing techniques and shaders of my choice. Over this time a collection of more than 60 workspaces and hundreds of shaders were constructed showing off methods and algorithms usable in many areas. Only a subset of the most interesting results will be presented however given the amount of shaders produced and the fact that it takes significantly longer to actually describe an effect than it takes to take it from idea to implementation in RenderMonkey. Like in any other kind of research work, not all work brought forth a whole lot of valuable results, and everything did not pioneer into new fields or even was particularly useful at all. The focus will thus be on the shaders and techniques that actually were. Those interested in the work that's not being presented may want to download the full package of the actual workspaces once it becomes available for public access at www.ati.com.

## *Materials*

It is very frequently desired to imitate the appearance of materials in the real world in our applications. The traditional solution is to just use a texture of the material in question. This is still, even with the arrival of hardware shading, often the method of choice; for good reasons I might add. Texturing has a number of desirable properties such as being easy, fast and accurate. The hardware provides fast filtering that pretty much eliminates aliasing. However, texturing has a few limitations too. The resolution is limited, and would there be no hardware resolution limit it would still take quite a lot of memory to use high resolution textures. High resolution textures may also take longer to create for the artist. The resolution limits leads to another problem. Textures often need to be wrapped and repeated over larger surfaces. The repeating pattern tends to reduce the realism of the scene.

With shaders entering the stage we have a new powerful tool to simulate the appearance of various kinds of materials. Mathematical functions can now be used to define the material color. The advantage is of course that mathematical functions have infinite resolution and may be infinitely non-repeating. It is also easier to adjust the appearance with just a few parameters sent to the shader. Aliasing however tend to quickly take the euphoria over this away. Unfortunately, aliasing is a complicated problem and tools to aid you in this task are sparse in current hardware. However, when using low-frequency functions aliasing is low and no shader antialiasing will be needed. Implementations of this will be discussed in the Wood and Fabric subsections below. It should also be mentioned that there's nothing that prevents us from using both textures and mathematical functions to define the surface appearance. Indeed, this is often the best idea. One can take advantage of both the performance and filtering of textures and build our shader upon that. The filtering will help reduce the aliasing of the shader. A good example is when using a noise function, (which can't be said enough how useful it is,) in which case you can pack a tileable noise function into a texture and repeat it over the surface. The alternative, to calculate the noise function directly in the shader would not only be much more calculation heavy and consequently much slower, but it would also likely cause aliasing in many cases. The texture filter however would automatically reduce the noise function's frequencies to not exceed the fragment-rate in screen-space. One may ask why we do anything in the shader at all; can't we just pack everything into the texture anyway? Well, often we can; however, there are cases when it's desirable to do some of the math in the shader. It is often more flexible to do it that way. A few parameters can adjust the appearance and one can basically have one shader do loads of different kinds of wood for instance. Also, textures can (as mentioned) take a lot of memory, especially if volumetric textures are desired. A noise function texture can be shared between shaders, but a certain full-blown wood texture for instance isn't particularly useful for another wood material.

The most important aspect of shaders is that it is now much easier to make extensive calculations to better match the behavior of certain materials. This will be illustrated in the Glass subsection where the refraction vector of light passing through the glass will be calculated according to the real physical law.

## Wood

Wood is one of the most common materials used in the real world and therefore is one of the more interesting materials to simulate. Many of the properties of wood were discussed earlier in this text in the *Porting from DX9 HLSL to OpenGL 2 glslang* section, so no lengthy discussion will be added here. A number of variations of wood will be briefly discussed and implemented. The basic idea is similar to that discussed earlier. All implementations sample a volumetric noise texture according to spatial location of the fragment in world-space and all implementations use the same simple lighting model, basic half-Lambert diffuse plus some Phong specular.

The first implementation creates a kind of layered wood. Rings are created along the z axis (pointing upwards). The z coordinate will be scaled by a ring frequency and noise added. Using the method in the wood shader discussed in the *Porting from DX9 HLSL to OpenGL 2 glslang* section it would be mapped into a 1D texture. This time an alternative approach will be used however. First the number needs to be reduced to a period, just taking the fractional part of the number will do. We now have a number between 0 and 1. In order to get a smooth image we need to remap this into a function that maps from zero to one and then back to zero again within this interval. For this use a function is used that from now on will be called the bounce function and is defined as f(x) = 4x(1-x). More details about this function under *Additional mathematical reasoning* in Appendix B. The resulting number after this remap will be used to linearly interpolate between a dark and light wood color. At this point there should the shader should generate a fairly woody look. As an additional tweak the result from the remap function will be raised to a provided power in order to adjust the look of the wood. A lower exponent and the dark rings get thinner while a high exponent creates thick dark rings and thin light ones. It should be noted that the bounce function squared is the smoothbump function also discussed in Appendix B. An exponent of 2 will thus yield equally thick dark and light rings and the overall smoothest appearance.

The second implementation isn't all that different from the first one. In fact, it's just one detail that differs. Instead of using the z coordinate it's using the cylindrical distance, the length of the (x, y) vector. Otherwise it's identical. The result is that the model looks more like it's been carved directly out of the tree.

The third implementation tries to create a woody look that's less regular, le ss circular or planar. The base is a sine function of the z coordinate. In addition the circular distance to the z axis is added along with some noise. Finally the sine is taken on all that and is scaled and biased before it's used for the interpolation between the dark and light wood color.

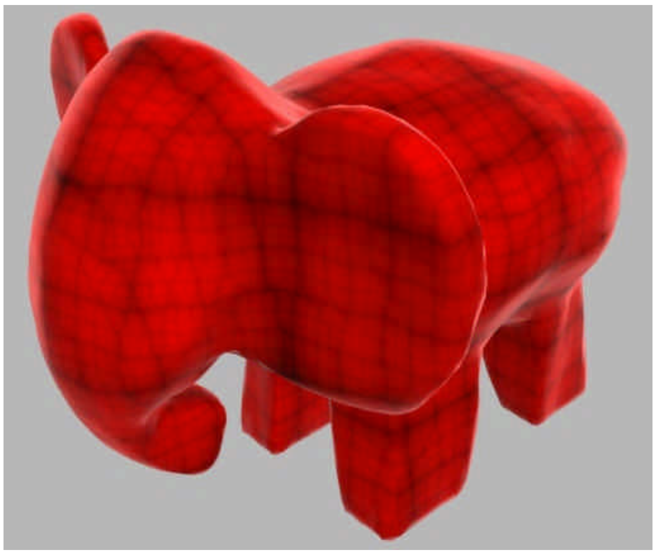
The fourth and last implementation makes an attempt on another layered kind of wood. It's supposed to look roughly like plywood. The same basic idea will be used as in the first implementation, but with another remap function. Again it's desired to go from zero to one and back to zero again within the [0, 1] range. However, another function will be used that goes up almost linearly, smoothly fades off at its peak and then sharply declines back to zero again. Practical tests show that for instance $f(x) = x - x^a$ behaves this was. This function doesn't reach 1 at its maximum however, so one need to find its maximum and divide the function with that number. To solve that task one need to solve the equation $f'(x) = 0$ given the value of a. Inserting the result into f(x) gives us the maximum. Since it's faster to multiply than divide it's beneficial to look for the inverse instead. Instead of going through all not that interesting steps needed to solve this task it's better to just state the conclusion:

$$\frac{1}{f_{\max}} = \frac{a^{\frac{a}{a-1}}}{a-1}$$

The mathematically interested reader may choose to derive this result himself. Computing the inverse maximum should preferably be done on the CPU and passed to the shader, however, RenderMonkey doesn't have any support for doing anything on the CPU (it's just a prototype tool anyway) so it needs to be implemented in the shader.

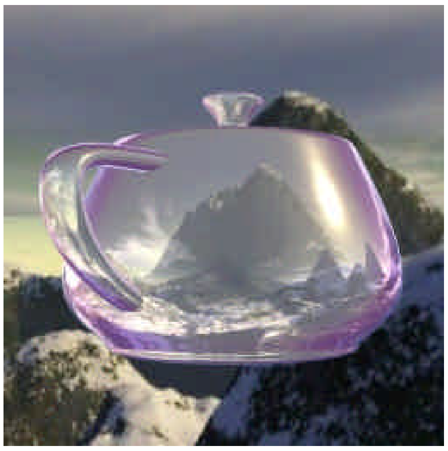
## Fabric

There is a multitude of kinds of fabric, all with different kinds of patterns and lighting behavior. A kind of Scottish looking pattern was implemented and the lighting model chosen was similar to that of the velvet shader discussed earlier in this text. The base function of the pattern is a grid function. The grid function is essentially just the bounce function applied in three dimensions. The spatial location coordinates are scaled by the frequency; all components are reduced to their fraction part, and finally passed through the bounce function and summed together. The grid function alone is a good start, but doesn't give all that interesting results directly and causes some aliasing if using a high frequency. The solution is to repeatedly apply the grid function. You begin with low frequency and high amplitude, and in a number of iterations you double the frequency and scale down the amplitude and sum it together. The way of working is very similar to that of creating a turbulence function out of noise functions. This both creates a more interesting pattern and reduces the aliasing significantly. Once the lighting is applied the effect is finalized.



## Glass

Glass has a number of interesting properties. First and foremost, it's transparent. Light doesn't just pass straight through glass as one may think. When we think of glass we tend to think of windows, but that's just one shape of glass there is. As we all learned in basic optics at school, light refracts when it passes between optically different materials. With simple shapes such as thin glass the refraction is hardly visible since it is refracted back to the same angle again on the other side of the glass and the slight offset is only a fraction of the thickness of the glass. When going from optically thinner materials such as



air to optically thicker materials such as glass or vice versa the light is refracted according to Snell’s refraction law:

$$n_i \sin q_i = n_r \sin q_r$$

In this sample Snell’s law will be used to roughly approximate the appearance of refraction through glass. Only one level of refraction will be used, while in real life there should also be refraction where the light exits the glass object too. It would be significantly harder to use more than one level of refraction though. In order to simplify the equation a little the ratio between the refraction indexes of the materials will be used instead of performing the division in the shader. The angles can’t easily be extracted from the provided data; however, the cosine of the angle between the incident vector and the normal can be found through the dot product between these vectors. The sine can then be evaluated as

$$\sin q_i = \sqrt{1 - \cos^2 q_i}$$

By multiplying with the index ratio sin ?$_r$ can be found. Once the sine is found the cosine of the refraction angle is also desired. Similarly,

$$\cos q_r = \sqrt{1 - \sin^2 q_r}$$

Out of the sine and cosine of the angle between the refraction vector and the normal the refraction vector itself can be constructed given two base vectors. These two base vectors are the negative normal and a tangent vector along the path of the incoming vector relative to the surface. We define

$$x = -N$$
$$y = normalize((V \times N) \times N)$$

where N is the normal and V is the view vector. The refraction vector is then

$$R = x \cos q_r + y \sin q_r$$

Given the refraction vector the refraction color can be looked up in an environment map that can either be precompiled or rendered to in real time. In this prototype implementation a precompiled cubemap was used.

As a side effect of refraction the colors may spread. Colors don’t refract the same, as can be proven by using a simple glass prism and white light. This effect is not visible on normal glass windows, but may be seen on glass spheres or other kinds of decoration objects made of glass. The effect is that at sharp angles you may get a rainbow looking tint at the edges. A cheap approximation of this effect can be done by packing the colors of the rainbow into a 1D texture. This texture can then for instance be looked up according to the cosine of the incident angle, which is already given from the refraction computation above. As an additional tweak the cosine can first be raised to a power before the texture sampling to push the colors closer to the edges.

Finally, glass reflects light too. The reflection vector is simple to compute, especially since DX9 HLSL has a built-in reflect function (which unlike the refract function actually works in my experience). The reflection vector is simply

$$R = 2(V \bullet N)N - V$$

The reflection is then looked up in the environment map with this vector in a similar fashion as the refraction.

To construct the final image the pieces only need to be assembled to get an image like the screenshot above. One can conclude that there is more light reflected at sharp angles and less light refracted. There is also more color separation of refracted light at sharper angles. So reflection will be scaled with the sine of the incident angle, the refraction with one minus the sine of the refraction angle and the rainbow refraction with the sine of the refraction angle. Note that this is not the final truth about how things should be combined, but rather an assembly that proved to work in practice.

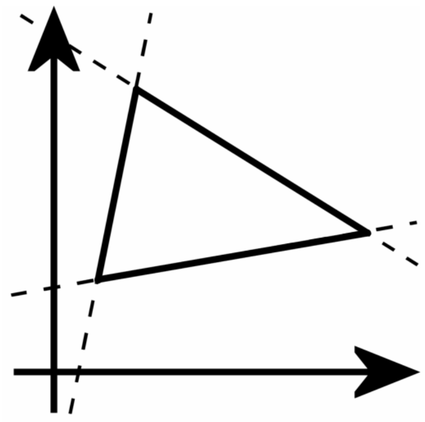
## *Techniques for industrial use*

There are many applications in industrial functions where graphics or image processing plays a central role. For instance in robotics and automation there are commonly camera images that need to be processed and information extracted. Often one or many high speed CPU's is the major player or expensive specialized hardware is used to solve the problems and fulfill the needs in industrial applications. Many of these applications could just as well use cheap off the shelf graphics processors reducing developing time and cost. A few examples of concepts useful in industrial processes implemented on the GPU side were constructed.

### Image classification with the Hough transform

It is often needed in automated industrial processes to classify objects captured by a camera. It may for instance be objects passing by on a conveyor belt that needs to be sorted or possibly checked for construction faults. One may either choose to hire a guy doing this work manually, but often it is desirable to do this automatically for cost and performance reasons. Also, few people want to do this kind of boring and monotonous work, it's definitely a job better done by machines. Unfortunately, the task of automatically identifying objects on camera images is hardly trivial. However, there are a number of tools in the industrial image processing toolbox that can help us extract classification information from simple pixel images. The Hough transform is such a tool.

The Hough transform is basically an operation that transforms an image from Cartesian space (x, y) to a polar form (r, ?). It typically operates on a grayscale or monochrome edge-detected image. It may come from a camera image and may have been passed through a dilate-erode-erode-dilate sequence, median-filtered or otherwise cleaned up before being edge-detected with Sobel, Laplace or some other edge detection filter. The Hough transformed image contains an image where the value at each point represents the sum of the pixels along the line with the angle ? at the distance r from the origin. The resulting image in itself isn't very interesting and isn't much more useful than the original image. What's interesting though is the information that can be extracted from it. In the triangle example shown in the illustration there would be three strong peaks showing up in the image for set of lines that closely lines up with the actual line in the image. These are very close to each other in a polar space and will form blobs together. By counting the number of spots in the Hough image you know how many edges there are in the original image. This way you can discern for instance an undamaged box (which should have 4 edges) from a damaged one (which might have 5 or more edges) as it passes by on the conveyor belt. Also, by locating the blobs you also know the equations of the lines and can find damaged boxes by checking if the angles differs more than a certain threshold from 90 degrees.



The Hough transform, as useful as it may be, is however a quite expensive operation and has roughly $O(w_H \cdot h_H \cdot (w_I + h_I))$ execution time, where $w_H$ and $h_H$ is the width and height of the Hough transformed image and $w_I$ and $h_I$ is the width and height of the original image. The operation is quite simple: for each (r, ?) in the Hough image, figure out the line and sum together all pixels along that line in the original image. Though the operation is expensive, you can often get away with using lower resolution images.

It should be noted that the task lends itself very well to hardware acceleration. It consists of what graphic chipsets are good at, and what CPUs are notoriously bad at, namely repeatedly sampling loads of pixels from an image. Not only is the sampling itself faster and can provide filtering basically for free (which is very expensive on CPUs), but the algorithm doesn't need to waste cycles figuring out the exact memory location or ensure that sampling occurs within the image boundaries since the chip can do that automatically and without any performance penalty. Graphic chips also tend to have higher bandwidth, and though no analysis was made to prove this statement it is intuitively understandable that the algorithm has better caching and prefetching behavior running on GPUs.

The GPU implementation is exemplarily simple. A screen-sized quad is placed over the whole screen. The vertex shader passes r and ? through a texture coordinate and lets r range from 0 to 1 horizontally and ? range from -p to p vertically on the screen. The image to be Hough transformed is of course stored as a texture. In the fragment shader the line equation given r and ? will be determined. Both the sine and cosine of the angle needs to be evaluated, which conveniently can be done with the sincos built-in function in DX9 HLSL. There are also sin and cos functions, but there is a performance gain to be had by using the sincos function instead when both the sine and cosine are needed. Some hardware, for instance the Radeon 9700, does not support trigonometric functions natively. Instead it's supported through a Taylor-series expansion, which takes many instructions and requires that the argument is reduced to the range [-p, p] in beforehand. Unlike the separate sin and cos functions the sincos function however simply assumes that the argument is already within the range and the reduction is skipped. Since the values will always be in this range anyway this fits the situation perfectly. It may be argued that constructs like the sincos doesn't really belong in a high-level shading language, but since it's there, benefits some of today's hardware and works for us we'll use it. Once the sin and cosine values are ready a mid-point and a sampling distance vector can be determined. The texture will then be sample at the mid-point plus discreet multiples of the distance vector and the sampled values accumulated. Depending on how accurate results are needed the sampling distance and number of samples can be chosen. In the RenderMonkey implementation a total of 61 samples were taken. In order to fit within the resource limits of the Radeon 9700 it was split up into three passes. It should be noted though that even with the two last passes disabled the results already look pretty good. The important thing is of course that the Hough transform is performed at fairly high resolution at interactive frame-rates. RenderMonkey doesn't include any performance analyzing tools at this point however, not even a simple frame-rate counter, so it is hard to give a better analysis than that, especially since the output is static and thus hardly possible to judge by your eyes at what rates it is updated either. However, by maximizing the rending window one can see that even high resolution image like that (in my case around 1400x800) it was updated pretty much immediately, thus updated at least a number of times a second even at that resolution, and consequently much faster at smaller resolutions.

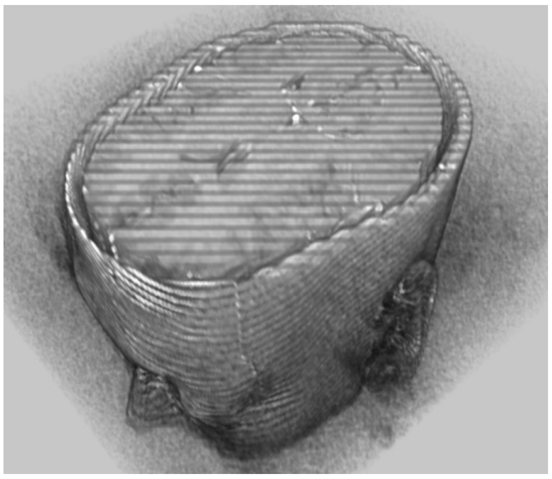
It's not entirely clear whether this work is significant; much further analysis is needed to draw any definite conclusions. I have little insight into the needs of solutions like this in industrial processes, but suppose it would be possible to reduce costs by accelerating heavy tasks like this with cheap graphics chips. The development cost could probably be reduced by a fair amount too if standard stock chips can be used more or less directly with just the right software. It should be said too that the Hough transform in itself isn't all that useful. Capturing the image from a camera needs to be done and the image needs to be filtered and edge-detected. Edge-detection is fortunately easy to do in hardware too on graphics chips and this task can simply be done as a preprocessing step to the Hough transformation. Standard filters like Sobel and Laplace were successfully implemented and will be discussed in the postprocessing sector. Different ways of cleaning up an image before edge-detection such as blur filters, median filters etc. can also be implemented in hardware too. Median filters can be tricky on current hardware, but should be doable. The next generation hardware shouldn't have any problems whatsoever.

The information from the Hough image needs to be extracted too for it to be of any use. Thinning filters to reduce blobs to single points that can be located and counted should be possible to implement in a shader too. Given a point-reduced image there's a trick that can be applied to feed back the number of points in the image. One can simply clear the depth buffer to 0.5, then in the shader read the image, threshold it and output the result to the depth. Then an occlusion query can be used to find how many pixels passed the depth test. This way only the number of edges in the original image is retrieved; not their location and direction. If such information is needed the last step will have to be performed on a CPU, but this task isn't particularly computation heavy either and should not bottleneck the process the least.

## Volume visualization

It is sometimes desirable to view volumetric information captured by scanning real objects. For instance in medical applications one may want to study a brain scanning looking for tumors or damages. The information from a typical object scanning is generally not directly useable for viewing however. Scanning a brain you may get something like the viscosity at each point in the brain as a three-dimensional array. Two-dimensional information is not that hard to view, that's just an image to draw; load it as a texture, draw and you're set. For volumetric data it's not that simple though. There is support for three-dimensional texturing since a couple of hardware generations, but we still need geometry to apply it to, and it's still applied to two-dimensional surfaces as it has always been, and will likely stay that way for times to come.

When scanning ordinary solid objects like for instance a teapot one can simply do a range-scan and come up with a set of points. There are algorithms to create polygonal models from such sets of points, which can solve the problem for such objects. It may not necessarily be fast or easy, but it's certainly both doable and useful. For brain-scanning or other kinds of soft or organic materials it is not so easy or particularly useful to extract a polygonal model however. For things like bones a polygon model could be useful and intuitively easy to define, but what about fluids, soft biomass like brain substance, blood etc? There is no clear distinction between what to consider being part of the object and what should be thought of as empty space. A method to draw the volume directly from the data is what's desired. Two methods to view volumetric data were implemented. The data used was from a brain-scanning containing the solidity at each point in the volume.



The first method draws a cube of polygon slices. 100 slices were evenly spread through the cube parallel to the XY plane. Similarly for the XZ and YX plane. The volume texture is then accessed according to spatial location. For blending the max function is used, which will give us the maximum value in that direction through the volume at each pixel. This way the significant parts in the volume are extracted, though at the expense of lesser details and some loss of the volumetric feeling. This method should do very well as a way to find the parts we're likely most interested in (tumors etc.).

The other method uses screen-aligned quads cutting through the volume and again sampling the texture according to spatial location. By using a set of quads that always faces the screen it is very easy to guarantee back-to-front draw order, which is required to do transparent blending. Transparent blending should give a better volumetric feeling over the volume. The volume contains solidity factors that can be directly used as transparency. We don't have any colors of the material however, which is necessary for the blending to be meaningful. So one need to make educated guesses and assign suitable pseudo-colors. High solidity would map to bones, which are white, while low solidity represent body-fluids, which can be assumed to be dark. So directly transferring solidity to a grayscale color should results in a reasonable good picture and practical tests verified this. Additional control over color and transparency is needed however and simple scale and bias operations was implemented. This way the picture can be adjusted according to what details in the data we're interested in, for instance as a fuzzy cloud with everything visible or closer to a solid object though with fewer internal details visible. By scaling up the brightness and scaling down the solidity the resulting image is closer to that of the previous technique with significant details maybe not quite as clearly visible but with better volumetric feeling over the picture. By using a moderate brightness and a high solidity a more solid look of the object is achieved.

In conclusion; the latter technique is much more flexible and generally preferable over the former. The second technique suffers somewhat from artifacts where the slices enters and exits the object in the volume. Dark rolling lines appears on the surface as the volume is rotated, mostly when viewing the volume in a more solid fashion. This phenomenon isn't as disturbing as one may think however as you get used to it. It would of course be much better if it wasn't there at all. A possible solution may be to add a preprocessing step that takes the solidness, applies a volumetric dilate filter and uses that as the color. This was not tested though. Both techniques are slow, but scalable. The number of slices used determines the quality, but rendering time is of course increased linearly with it.

It should also be mentioned that there are plenty of techniques out there covering the topic of volume visualization. In all likelihood there are more sophisticated methods out there, though how well they map onto hardware, what performance figures they reach etc. is another question. The methods presented here though map very well to hardware acceleration and is reasonable fast for real-time interaction.

## *Techniques for the gaming industry*

The gaming industry has a number of special needs and interests. A game generally is a mainstream product used by large quantities of people and is supposed to run on whatever system they have. Games also often take a long time to develop, often around 12 to 18 months. This can often be somewhat problematic and tend to cause a lag between technology introductions on the hardware side until they appear in ordinary games. It is in the interest of hardware vendors to get game developers to adopt new technology as soon as possible since that helps sales of newer graphic accelerators. While hardware vendors hardly can do anything about a game's development roadmap they can help shorten the development time somewhat by educating developers about techniques and providing illustrating samples.

An important aspect to think of when dealing with techniques aimed for the gaming industry is the performance. A game may be very beautiful and artistic, but it won't be much fun if it's sluggish or jerky. While graphics is quite important for how the game is perceived, especially in single player adventure games, games are in the end entertainment products rather than artistic masterpieces and are supposed to be enjoyable to play. So perfection is usually traded for some performance. When constructing techniques for gaming use you tend to think from a natural human point of view rather than complicated physical models. The old saying that if it looks good then it is good is even truer when it comes to games. A bunch of techniques for various applications within the gaming sector were constructed with these things in mind.

### Volumetric lighting

Volumetric lighting is a really cool effect that can enhance the mood quite a lot in many gaming scenes. Volumetric lighting tries to imitate the effect of light being reflected on particles in the air, such as when there is mist or fog, which creates a kind of corona around the light. Adding volumetric lighting to a gaming scene enhances the atmosphere and can make the scene look closer to what the artist built up in his mind; a chilly night looks chillier with a gray-white corona around the street lights and hot burning lava looks hotter if there's some red-orange glow around it. Volumetric lighting has been successfully used in a number of games such as Unreal.

There are a number of existing implementations of volumetric lighting. Unreal for instance computes the volumetric lighting on a per vertex basis on the CPU. This works for a low polygon count game like Unreal. For next generation games this won't do though as polygon count goes up. Doing it on a per vertex basis also introduces some artifacts as can be observed occasionally in Unreal when the light volume at times changes shape along background polygon edges.

An implementation of volumetric lighting solving these problems was implemented. The basic idea is to take the line between the viewpoint and the current fragment and track the distance between this line and the light. The fragment shader was fed with the fragment position, the camera position and the light position. This distance can then be used to decide how much light volume effect to get for the fragment in question. As can be found in any decent linear algebra book or easily derived oneself the closest point p on the line ($p_0$, $p_1$) to the point $p_2$ is:

$$t = \frac{(p_2 - p_0) \bullet (p_1 - p_0)}{(p_1 - p_0) \bullet (p_1 - p_0)}$$

$$p = p_0 + t \cdot (p_1 - p_0)$$

Once the closest point has been found the distance to the light can obviously be found with Pythagoras theorem. It is important to mention at this time that a value on t of zero would return $p_0$ and a value of one would return $p_1$. Only points between zero and one are of interest, so the value of t needs to be clamped to the [0, 1] range. This may not be obvious at first, but running tests without clamping illustrates the problem. If for instance the light is behind an occluder hiding the light from the camera's point of view there should obviously not be any volumetric effect showing up on this occluder. With t unclamped however you would see the light kind of showing through occluders. This is undesired behavior. In this case the value of t would be larger than one and the point p close to the light. When clamping the value of t to one and thus p to $p_1$ the distance to the light will be that of the surface to the light, which is the property one would rather want in this case. In the other case, where t is negative, in other words the light is behind our back the same problem occurs. The light behind our back shows up as a spot on the objects in front of us. Again, this is undesired behavior. Clamping t to zero and thus p to $p_0$ result is the distance from the camera to the light, which is the desired property in this case. There is no longer any backlighting and if the viewpoint is close to the light the volumetric effect will covers the whole screen and will the feeling that the viewer is inside the light volume. Practical tests show that clamping is enough to solve the problem in the general case. Being close or far from the light, in front or behind it, being occluded or not; all cases works. Some algorithms that for instance use geometric light volumes may on the other hand have problems with some of these cases. As for the clamp itself, it is the cheapest operation one can imagine on current hardware. It is basically for free in the majority of the cases since it maps directly to the _sat instruction modifier.

When the point p has been settled the distance d can be evaluated and a simple attenuation function be used to find the volumetric factor of the fragment in question:

$$a = \frac{1}{c + q \cdot d^2}$$

Here c is a constant and q is a quadratic attenuation factor. Since the squared distance is used we can drop the square root out of the distance calculation. The thickness of the mist is controlled with the c variable. A c of 1 would result in a lighting that completely takes over the scene for the line passing through the light and would result in a thick fog. A c of maybe 2 or so would result in a gentle mist. The q parameter on the other hand controls the spread of the light. A low q results in a larger light volume and a high q in just a small corona close to the light.

In the final stage the volumetric lighting factor needs to be combined with the color of the fragment as grabbed from a texture or otherwise computed. Assuming the fragment color $f_{col}$ has been uncolored lit the final combination stage would look something like this, where $m_{col}$ is the color of the mist and $l_{col}$ is the color of the light:

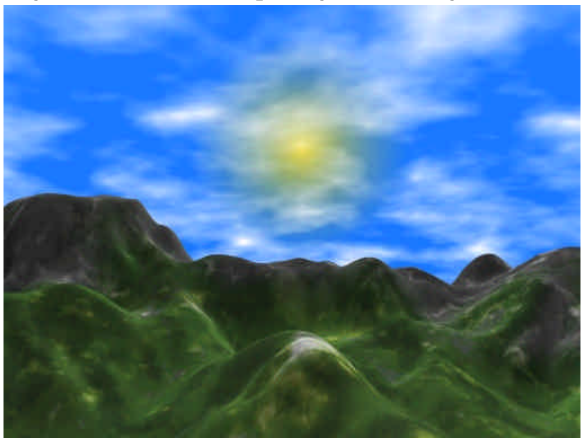
$$color = l_{col}(f_{col} \cdot (1-a) + m_{col} \cdot a)$$

This equation maybe isn't that obvious at first, but makes sense. The higher the volumetric lighting factor the higher contribution we get from the mist. The same mist that causes the air to light up should also interfere slightly with the light from the background, hence the 1-a factor. One may instead use some kind of fullscene fogging in which case that factor should be omitted. If the background has been uncolored lit, as assumed above, it's only necessary to multiply it with the light color to get colored light. The color of the light also affects the mist, hence it's place outside the parenthesis.

In conclusion it needs to be said that this technique has a number of positive attributes. The most important, that the volumetric feel of the light volume is very convincing, should be the most pleasing. It is not that costly to implement, while not being quite for free either, but certainly realistic to use on today's hardware and definitely tomorrow's. It works directly with more than one light too and should be easy to integrate with most lighting implementations. Beyond the sample RenderMonkey implementation the author decided to try this technique in a real application at a later time after the full work at ATI had been finished which show the effect in an environment closer to a real life gaming situation [14] which should firmly put the stamp of approval on this technique.

## Sky rendering

Rendering a believable sky is a key component in most outdoor scenes. The most common way to draw the sky is to just use a pre-rendered skybox. While the quality is usually excellent it has the drawback of being completely static. To solve this problem many games use additional layers to simulate moving clouds in the sky. However, these layers tend to be static themselves and simple animation of the texture coordinates is used to move the cloud texture over the sky. In real life clouds are seldom static though. As they move over the sky the shape and size is often changed, especially if it's very windy. Fortunately, simulating this behavior is surprisingly simple.



The solution is again to use noise. If there was any doubt about the usefulness of noise in graphics this example again ascertains its utility. Clouds are somewhat random looking, so that noise is used shouldn't come as a surprise. To begin with a volumetric texture is filled with tileable three-dimensional turbulence. This texture will then be mapped onto the sky according to horizontal position, or the (x, y) coordinates. By panning these coordinates with time the technique mentioned above is basically repeated, except it is just noise being panned over the sky instead of a cloud texture. Noise and clouds looks fairly similar though, so we're basically at the same point. But there's a reason behind the use of a volumetric texture. The texture will also be panned slowly in the z direction. Since the noise is continuous in all directions the achieved effect is that the clouds slowly are morphing and changing shape over time as they move. Once the noise is sampled it can be used to interpolate between the background sky (which may just be a constant color (blue or so) or a background skybox) and the cloud color. The final touch is put onto the image by putting scale and bias on the sampled noise and clamping it to the [0, 1] range. By altering these parameters the appearance of the clouds can be changed. Low scale and small bias for instance means a thin almost constant thickness layer of clouds. High scale on the other hand means thick clouds, and with a bias close to zero there are just a few clouds on the heavens while a larger negative bias puts clouds pretty much all over the sky.

## Fire

Fires are another element that can really lift the mood and enhance the ambience of the scene. Many games use two-dimensional procedural textures to create fires, with mixed results. Some games such as Quake3 use a short animated sequence of fire images. While this is cheap the results have a tendency to be jerky and repeating, which immediately takes the realism away. The jerkiness can be helped by inserting more images, but that may be costly in terms of video memory. To make the fire looking less repeating one may have to make the sequence significantly longer, which again can be costly in terms of video memory. Other games such as the earlier Unreal series games uses CPU generated procedural fire textures. While this can solve the jerkiness and repeating behavior it has its own set of drawbacks. The number one negative aspect is that texture uploads seldom are cheap operations. Texture uploads are synchronous operations meaning that the driver must finish its access to the memory being uploaded before the call returns. If the texture is in use this may mean a stall or that the driver needs to make a copy of the texture of its own. It may also take a good deal of CPU power to generate textures on the system processor. Newer Unreal series games mostly use particle systems to generate fires and similar effects. Particles systems are generally preferable since they can easily be used for many purposes other than fires and unlike procedural textures are volumetric. This doesn't necessary mean that procedural textures are quite dead yet, newer Unreal games still use procedural textures to some extent along with the particles systems, but a good reform of the old trusty procedural textures may be motivated. Particles systems have their set of drawbacks too. The largest disadvantage is that they can be quite fillrate heavy. They also put additional burden on the CPU, unless an approach similar to that discussed later in this text under the *Particle systems* subsection is taken.

The necessary reform of procedural textures is that the work needs to be put on the GPU side to avoid CPU and AGP bus burden and allow better parallelism between the CPU and

GPU. This means shader generated procedural effects. Shader generation naturally has its set of drawbacks however, which shouldn't be ignored. The number one drawback is that it can be quite demanding on the fragment pipeline. Fortunately, many effects can be reduced to only a short shader. A quite complex shader with a lot of flexibility and many adjustable parameters will be implemented here; though a fire effect doesn't necessarily need more than eight instructions as was shown in a demo based on the idea behind this effect [15], and by removing the sine wave tracking behavior one may do it in just four instructions or even just three when dropping the curvature.

As discussed earlier under the *Porting from DX9 HLSL to OpenGL 2 glslang* section fires have a fairly random behavior, though a controlled such, so it's hardly surprising that the base of this effect is noise. Like earlier, a tileable turbulence function will be packed into a volumetric texture. It will then be mapped over the screen slowly sliding upwards, but also sliding in z direction to get a slowly morphing fire. To get a stronger fire at the bottom and to let it fade away towards the top of the screen the position in y-direction will be subtracted. To get a fire look this parameter, which may represent the heat, will be mapped into a 1D texture containing fiery colors going from white-yellow to orange, red, dark blue and finally black. If one would want transparency for the fire one would only need to pack an alpha channel along with colors as to map black to completely transparent and the rest to various levels of translucency. Directly mapping the heat into this flame texture gives us a fairly good fire effect already. To adjust the shape of the fire to something more like a real fire rather than a constant height fire as we got right now a number will be added to the heat according to the position in x-direction. Our x coordinate is ranging from -1 to 1, so a $1 - x^2$ function will do. To be able to scale how sharply the fire should fade towards the sides two parameters was added to form this final expression:

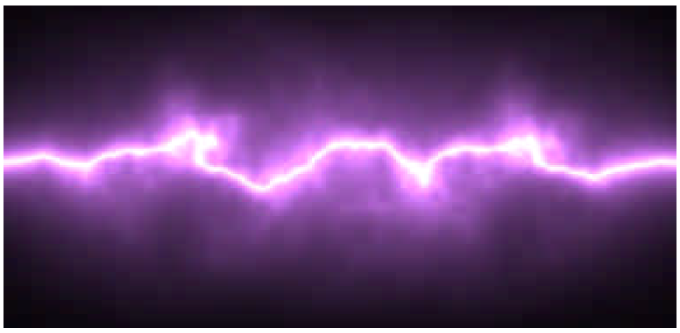
$$f_{sharpness}(1 - f_{scale}x^2)$$

Finally, fires are seldom just moving upwards. There's typically a waving movement, and sparkles are shooting off left and right. There's plenty of ways of getting this waving behavior into the fire, neither is really preferable except performance-wise maybe (though no comparison was made), otherwise it's mostly a matter of taste. One method is to let the fire track a sine wave in y-direction. One may for instance calculate an offset in x-direction for the position where to sample the noise using some function of y. Using $\sin(\text{time} + c(1 - y)^2)$ worked fine which have the property of quicker waving movements closer to the base of the fire and larger movements at the top. A static waving movement may not be all that much more convincing than no waving at all, that's why the time is also in the expression to let the waving behavior morph slightly too. An alternative approach is to just sample more noise and use that to offset in y-direction.

## Electric flash

In some games electric flashes are used to create cool effects or strengthen the feeling of danger as your character passes through some high tech area where perhaps aliens may hide. While sparkling electrums hardly are useful in real life they have this special high tech feeling one may want to enhance the scene with, just like a laser beam, but cooler and bring more ambience to the scene. Typically the flash is placed between two nodes, where one may assume an extremely high voltage difference exists, and a shaking sparkling ray goes between them.

Electric flashes are usually implemented in a similar fashion as fires with procedural textures, and naturally it faces the same problems as discussed above. Instead of repeating what was said under the *Fire* section it's better to go straight to the solution and implementation. The solution is the same as with fire, a shader generated procedural image. Again the shader presented is more complex and flexible than what really is required, and again it was showed with a separate demo application provided at a later point that the effect can be fast and with some optimizations requires no more than nine assembly language instructions [16].



At this point I suppose hardly anyone raises his eyebrows as it is declared that Mr. Perlin is honored with yet another application where his noise is the central part of the effect. Electric flashes are undeniably quite random behaving, which leans itself towards the use of noise. Like with fire noise will be mapped over the screen space and will be let to slide slowly in z-direction to get the noise to morph. To get the sense of a power field around the beam the mapping in y-direction will be mirrored around the beam, in other words using the absolute value of y instead of the signed y. It will then slide with time, which creates a flowing effect away from the beam. To create the beam itself the beam's position in y-direction will be decided and the intensity set at a certain fragment according to its distance to the beam as follows:

$$glow = (1 - \left|y - b_{pos}\right|^{g_{falloff}})^2$$

This formula is hardly intuitive and not supposed to be treated as a final truth either. It's the result of a number of practical tests and was found to create the desired appearance, but there are likely other solutions that could have worked just as well. By setting the glow falloff factor one can adjust the size of the beam. The beam's position can basically be signed noise, that is, noise scaled and biased as to change it from [0, 1] range to [-1, 1]. However, to make it more useful for the typical scenario where the beam is placed between two nodes that are connected with it the beam is tuned to be fixed at its edges. This to ensure that it always looks like the beam leaves and enters the nodes and not just around it regardless how the nodes are designed and what size of the beam is desired. To get this behavior our signed noise will be multiplied with $1 - x^2$ as x varies from -1 to 1. This will constrain the flash into a bow-formed area.

Finally, some ambient glow is wanted to get a better feeling of power in the air. The same bow-formed shape will be used and simply by multiplying it with one minus the absolute value of y. This creates a nice ambient light around the beam and enhances the effect a little. Adding together the beam's glow and the ambient glow and multiplying it with a color of our choice, typically blue-violet or so, and we're set.

## Particle systems

A particle system is basically just a system of particles that are animated to simulate various kinds of effects. The particle is generally just a screen-aligned textured quad with a picture of some kind of particle on it. Typically the particles are sprinkled out from a certain position in particular direction in a semi-random fashion. The particle then has a short time to live until it fades away. Usually it changes color during its lifespan, for instance going from yellow to red to black if used to simulate a fire. By adjusting certain parameters such as size, texture, number of particles, spread, speed, and color scheme etc. the particle systems can be usable for many kinds of effects. Common usages are for instance fire, smoke, haze and flowing water. As discussed above, the advantage of using particle systems instead of for instance procedural textures is that the effect then gets volume.

Particles are in general simulated on the CPU and uploaded to the graphic card every frame, thus putting strain on both the CPU and the AGP bus. Depending on the amount of particles this may or may not be a problem. To ease the strain on the AGP bus in applications like this innovations such as point sprites have been introduced not too long ago, which significantly reduces the amount of geometry that needs to be sent, thus offering the AGP bus some relief. It would still at times be desirable to be able to do everything on the GPU in order to free some CPU power and AGP bus bandwidth. In many cases the additional flexibility of CPU work is very valuable, for instance if feature like collision detection or iterative processes are desired, though in many cases such features aren't needed. Fire is such a case. A sample of a fire particle system completely implemented on the GPU was produced to illustrate the idea.

The central part of the implementation is the vertex shader. This time we're dealing with dynamic geometry, thus the focus on the geometry part. To avoid AGP traffic static geometry needs to be in video memory. For this a static array will be used which consists of a hundred quads parallel to the XY plane with x and y ranging from -1 to 1. To differentiate particles each quad is assigned a z value between 0 and 1 at even distances. This z value will decide where in the animation loop the current particle is. Adding the time to this value and taking the fractional part we have a system where particles go from the beginning to the end of a cycle and then restarts.

One might have thought that we again would use noise to simulate the semi-random behavior of fire, and really, we would if we could, but unfortunately we can't. In the fragment shader we the noise function has typically just been packed into a texture. However, in current hardware there's no way to access textures from the vertex shader. It is likely that consecutive generations will sport this feature though, which would be useful both for situations like this and for other purposes such as displacement mapping. But for today one will have to do with a kind of pseudo-noise. Philosophizing a little over the properties of noise, ranging from -1 to 1 and being a band-limited continuous function, one may realize that these properties holds true for the normal sine function too, which is available in the vertex shader, though the sin(x) function obviously is much more band-limited than the noise function; it has exactly one frequency rather than a frequency band. With a careful use of the sine one may get pretty good results though. The vector components of the direction of each particle were set to be:

$$x_{dir} = \cos 62z$$
$$y_{dir} = \sin 163z$$
$$z_{dir} = 1$$

These formulas are based on practical tests and found to work pretty well rather than being based on some kind of mathematical reasoning. These expressions spread the particles nicely, though may look a little too linear; it is also preferred to be able to adjust the shape of the system. First of all, it's desirable to be able adjust the spread of the particles. To do that one simply multiply the x and y components of the direction vector with a constant that defines the amount of spread. To get a more non-linear look of the spread a factor s is added that is the animation position t raised to a power, typically between 0.5 and 2, that can change the shape from a funnel, to a cone and finally to something more like a wineglass. It's also desirable to be able to adjust the height of the system. Since the position t in the animation loop is ranged between 0 and 1 it's simply t multiplied with the height that sets the vertical position. The final expressions look as follows:

$$s = t^{shape}$$
$$x = p_{spread} \cdot s \cdot \cos 62z$$
$$y = p_{spread} \cdot s \cdot \sin 163z$$
$$z = p_{height} \cdot t$$

Once the position has been settled the particle quad needs just be billboarded at that position. The world space vectors of the x and y direction in view space are needed for that operation. These vectors can easily be extracted from the view matrix, the first row is the x vector and the second is the y vector. As mentioned earlier the input quads has x and y coordinates of (-1, -1) to (1, 1). So to get a perfect screen aligned quad around the computed position one only need to multiply the vectors with the input x and y coordinates. As of now our particle system is located at the origin, so in the final step we only need to add the position where we want the particles to sprinkle from.

A few words on the fragment shader too. Often a simple texture image is placed over the particle. This works fine and is often preferable, though we can do it with math too. In the sample prototype implementation the following expression was used:

$$f = 1 - (c \bullet c)^{p_{shape}}$$

The texture coordinate vector c ranges from (-1, -1) to (1, 1), so there will be full intensity in the middle of the quad and at a distance of one (the circle around the center touching the edge of the quad) it's down to zero. The $p_{shape}$ variable defines how the curve in between looks. A value of 0.5 on $p_{shape}$ obviously results in linear fading with distance to the center. A value of around 0.35 to 0.40 was subjectively found to look best for this application.

## Water

Water is a common component in many scenes. There are many types of water and water effects, which may need different kinds of implementations. A lake looks different from a river or creek. Water in a pod behaves differently than water in an ocean. A pod may look fairly static and almost completely translucent while an ocean tends to have waves and the bottom of the sea hardly visible. There are also things like waterfalls etc. that may need completely different approaches using for instance particles systems. The focus will be on the most common and maybe the most interesting kind of water, namely slowly waving water as in a silent ocean, as shown in the screenshot below. The effect may also be used in indoor pools or other larger collections of water with some changes.

Waves flowing over the water surface are quite random appearing, so the base for this effect is naturally noise. No real waves are going to be created as such by using geometry though. If it was possible we probably would since it creates a more realistic appearance, especially with larger waves. However, geometry means being processed in the vertex shader. Since our noise is accessed through a texture, which isn't available in the vertex shader in current hardware, we can't use any noise in the vertex shader. In the *particle system* section above a way around that problem was found in that particular application, however, it would be hard to use it for something like water. Instead the effect will be created purely in the fragment shader. The water surface will be nothing but a large quad. Over this quad the spatial position of the fragments in world space will be interpolated. To create the effect of waves imagined normal vectors will be created using noise. For this two independent noise functions are needed, one for x and one for y. Since unit vectors are desired and it's supposed to point upwards the z coordinate (pointing upwards in this sample) of the normal vector can be evaluated from x and y. To get the impression of two independent noise functions the same noise function will be addressed with the position and with the position offset by 0.5 in all directions to get two noise values. To get signed noise a value of 0.5 is subtracted from the noises so that the noise ranges from -0.5 to 0.5. Typically it isn't desired to have normal vector components of that magnitude, one rather want small motions and moderate bumps, so they have to be scale down a little before the z coordinate is computed. To let the waves move the noise will be sliding with time in the direction we want the waves to move. In this case the noise slid in positive x direction. To get dynamic waves, rather than just waves sliding around, a volumetric texture with a 3D tileable noise function will be used and let to slowly slide in z direction too.

Once the normal vector is ready a reflection of the environment on the water is to be created. To find the reflection vector the built-in reflect function is used. The result can be directly mapped into an environment cubemap. In the sample implementation a simple preassembled cubemap was used. Thus far there is a pretty good watery feeling in the scene, though with only reflection it has something more of a metallic appearance maybe. A sea of mercury doesn't have quite the same mood as water. In the deep sea the bottom isn't quite visible, so that's obviously not the missing component. Instead the reflection just has to be mixed up with some deep sea color, such as dark blue or maybe blue-green for a more exotic feeling, to get the full water appearance. As should be pretty well-known, light reflects to a much higher extent the flatter the incident angle towards the surface is. When looking straight down into the water there's much less reflection than when you look towards the horizon. So the dot-product between the view vector and the normal will be used as a factor when mixing the sea color and the reflection. Using a direct linear interpolation between the two unfortunately doesn't give quite the desirable results, so a couple of tweaks will be added to finally come up with this expression:

$$lerp(w_{color}, r_{color}, saturate(b + (1 - V \bullet N)^f))$$

This isn’t supposed to be taken as a final truth but rather as a route that works in practice. The parameters b and f can be adjusted to allow the transition from the water color to the reflection color to go in a way that’s looks fairly natural.

In conclusion it was found that this technique provides fairly good-looking water, as evident by the screenshot above, without needing any particularly complex shader. A better implementation, though likely more demanding, may be used in the future as hardware provide access to textures in the vertex shader and thus enables dynamic displacement mapping. Until then, and maybe even by then too in many occasions, this technique will suffice.

## *Cinematic rendering*

Cinematic render has become something of a buzzword these times in the real-time graphics sector. While the usage of the word “cinematic” tend to be highly exaggerated in pretty much any context where it’s been used within the real-time industry lately there is clearly a trend that things will move closer to the offline rendering world over the coming years. While nobody is going to render any movie contents here a few interesting effects common in the offline rendering world will be borrowed over to the real-time world.

### Glow

Glow is really just an artifact that can occur under strong light when snapping photos and is hardly a real physical phenomenon. The reason that it exist is that strong light into the camera lens will cause strong enough light reflections within the camera to affect nearby areas on the film. An ideal camera would be completely black behind the lens, however, there is no known material that is perfectly black, there’s always some light reflected. If the light is strong, even a tiny reflection factor will accumulate to become significant. In real life you may at times get a similar feeling if there’s a really strong blinding light right in your eyes as there can be some light reflections within your eyes too.

Despite being just an artifact it can be quite useful as an image enhancing tool. Glow gives the impression of a really strong light, or that an object is really shiny. Photos and monitors are range limited; you can’t represent stronger light than white, which is hardly blinding on most monitors. Thus you can’t create any actual strong light. This can be somewhat of a problem at times. If we for instance want an explosion to light up the scene with a strong light the effect can be lost due to this range limit, so various tricks have to be applied to get this appearance. One trick that can be used is to prepare for the effect by using mostly dark scenes the closest time before it to let our eyes adjust to darker surroundings. Then the light appears much stronger when it hits. This technique however means that the flow of the scenes must be known on beforehand. This works for movies or for cut scenes in games, but for interactive games it is harder to use. Glow on the other hand is a tool that can be used at any time and also gives an impression of a strong light. While not having any physical effect of a stronger light it has a psychological such effect. Glow can also have something of an artistic value in many scenes; it can make images looks warmer and more pleasant. For instance in many ads, especially for tourist attractions or when brokers show up the dream house, glow is added in a postprocessing step to add this feeling of warmth and pleasantness to the imagery. A sample where glow was used to enhance the impression of shininess and strong background light on a golden coin was implemented.

First of all the coin is drawn. A disc model was used and a texture of a two pound coin was used. To make the coin look more realistic some bump-mapping was applied. Coins are namely seldom flat, but rather have grooves and bumps. A height-map of the coin was created through some manipulations of the source image in a photo editing application. This is not an exact science but rather just artistry. We aren't looking for an exact representation of the topology of the coin, just something that will look real enough. So where it was imagined from the source coin image that the surface was raised one ensured that the color was bright and where the coin surface was thought to be low a dark color was assigned. From this height-map a normal-map was created by applying a Sobel filter. This is a fairly well-known technique, the Sobel filter in x direction gives the x component of the normal and Sobel filter in y direction gives the normal's y component. The z component is assigned a value of 1, the vector is normalized and the normal is done.

The normals stored in the normal-map can now be used for the lighting as the coin is drawn, thus getting a per-pixel lighting model. The normals stored though are in tangent space and not in world space. Thus the light vector needs to be transformed into tangent space to get correct lighting. The tangent vectors can be precomputed for the model based on the texture coordinates, something that RenderMonkey does automatically for us. For this sample eye-space lighting is desired to get the impression that the light source is fixed to the camera and to get a feeling of strong backlighting when the coin is facing the viewer. The tangent vectors and the normal however are in world-space, so they need to be transformed into eye-space. Multiplying them with the view matrix does this. Assuming the camera is the light source the light vector is the same as the view vector. So by transforming the vertex into eye-space and negating it the light vector is done. This light vector needs then to be transformed into tangent-space. Once that is done our lighting model can be applied as usual using the provided tangent space vectors and normal from the normal-map. A simple lighting model was used to create a shiny looking coin.

To create the glow effect the idea is to render the parts of the scene that generates glow into a texture, then apply some blur filter, and then add that on top of everything. Either the glowing parts of the scene can be rendered to the textures as it is, or another representation that better depicts the glow can be rendered. The former method may sometimes give better results, though it tends to require excessive blurring. It was tried in a separate sample at a later time [17] and found to work fairly well, though more blurring was needed. In this sample however another representation was used for glow instead of just drawing the coin and blurring it a number of times. This other representation is just the coin is extracted to fill a larger glow volume from the beginning so that fewer blurring passes is needed to get the glow to spread enough. To expand the model the vertices were simply extended along the vertex normal. The vertex shader simply adds the vertex normal times the desired amount of expansion along the normal to the vertex. For the lighting model the per-pixel lighting used earlier was simplified a little to gain performance, there's no need for that much details in something that's going to be blurred anyway. The normal-map was done away with and interpolated vertex normals used instead. The lighting was also reduced to only the specular component.

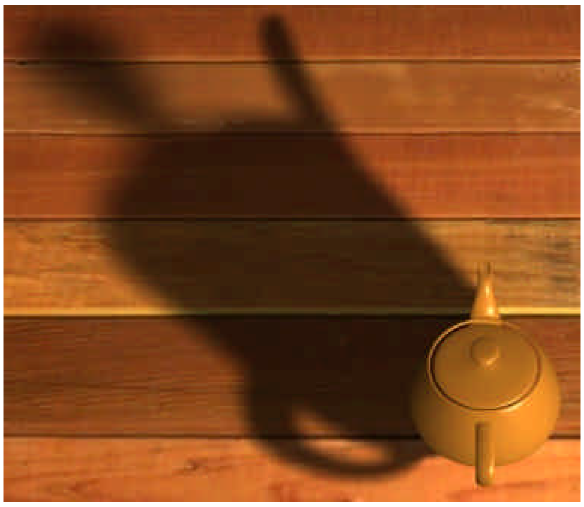
In the final pass the glow render from the previous pass was passed through a blur filter. With the actions taken in the previous pass no more than one blurring pass is needed to get the desired results. 13 samples were spread within a circular neighborhood area and averaged. The blending was set to add the result onto the image left in the frame-buffer from the first pass and the effect is complete. The result is a fairly good and quite cheap implementation of glow.

## Soft shadows

Shadows are an important aspect of most scenes. A scene without shadows hardly looks realistic. Not only because shadows represent a genuine physical phenomenon, but also because it goes hand in hand with the perspective in placing a sense of depth into the scene. Unfortunately, shadows are one of the hardest pieces to get in place in the lighting equation. While there are lighting models that works fairly well on independent geometry shadows have the problem of being defined by global scene information. You can illuminate a triangle with the only information of the position of the light and its parameters. To put a shadow on the triangle however you need to know where other triangles are located that may shadow it. This is the fundamental problem with shadows. It should be mentioned that in real life lighting is also dependent on global information, there are indirect lighting from other objects since all objects reflect light (otherwise we wouldn't be able to see them), and there are lighting models that take this into account and not just fake it in a simplistic way with standard ambient lighting. However, these models are seldom used in real-time graphics since they generally are hard to map to hardware or are very slow.

There are a number of techniques available to create shadows for real-time applications, but they all have their sets of advantages and drawbacks. The simplest technique is maybe projected shadows, but also the most limited of them all. The concept of projected shadows simply means that the geometry of the model casting a shadow is projected down into a plane that may represent the ground. The advantage of this technique is that it's fast and simple and works seamlessly with multisampling and thus we get smooth shadow edges if antialiasing is used. The obvious disadvantage is that its utility is very limited. It's only useable in a few applications that have a flat base on which all action takes place, such as maybe a chess game with advanced graphics. There's also no self-shadowing with projected shadows. A more promising technique is stencil shadows. It basically works by extracting shadow volumes from possible occluders and the light. The shadow volume is then drawn and pixels are tagged as within or outside the light using stencil-operations. The advantage of this technique is that it works regardless of how the scene is composed and allows self-shadowing. It also works seamlessly with multisampling, so shadow edges are smooth if it's enabled. There are a number of disadvantages though. It can be quite complicated to implement. The silhouette of objects must be extracted depending on the light and the position and orientation of the model relative to the light. It is also very fillrate heavy. To overcome that problem a number of measures have been taken. Some architectures for instance provide a number of additional depth/stencil units in addition to those available with the full-fledged pipelines. This is fairly cheap hardwarewise but can significantly improve performance when drawing shadow volumes as they only utilize depth and stencil. In addition a number of techniques have been presented and additional hardware support for various optimizations such as depth clamp and depth range hints. While this may reduce the fillrate dependency to some extent it also adds to the implementation complexity. Finally, there is shadow mapping, which is probably the overall best shadowing technique. It works by rendering the depth into a texture, the shadow-map. The depth test ensures that the closest depth is stored at each position. Pixels are then found to be lit or not depending on whether the depth matches that of the depth stored in the shadow map. Advantages of this technique are that it's very generic and easy to implement. It is not too costly fillratewise either and it is also scalable; slower machine may use a lower resolution shadow-map. It also allows self-shadowing. The disadvantage is though that it tends to cause some aliasing, and there's nothing multisample antialiasing can do about it. If the shadow map is too small the shadow tends to look blocky. There are a number of techniques around that tries to fix these problems, but they tend to have other drawbacks. Some architectures has support for percentage closer filtering on shadow maps which allows for smooth shadow edges, however, this only works for 2D shadow maps, thus is limited to spotlights only. For omni-directional lights one is forced to apply one's own filtering in the shader.

All these shadowing algorithms have one serious drawback in common: they produce sharp shadows. This may not completely off the mark in some occasions, but very often is a very inaccurate representation of how things would look in real life. So what does it mean that shadows are soft, and why does this phenomenon exist? Soft shadows are shadows that have fuzzy edges. There's no direct place where it goes from being in shadow to being in light, instead it slowly goes from shadowed to being lit over an extended distance, like in the screenshot to the right. The middle section where it isn't either completely in light or completely in shadow is called penumbra. The whole reason for the existence of a penumbra is because the light source isn't a zero-dimensional point in space as assumed by most lighting models. Instead light sources are spatially extended; a light-bulb is for instance better approximated with a sphere with a radius of around 5 cm or so than as a singular point in space. A point in the penumbra is thus a point where the light source is only partly occluded. If half the light-bulb is visible at a point in space that point is only lit with half the amount of light a point would receive would the whole light-bulb be visible, thus the point would appear only half as bright assuming there's no indirect lighting or any other lights in the scene. When talking about sunlight or moonlight though a sharp shadow can be reasonable. The sun may be physically extremely large, though at an incredibly long distance. It's easy to fool yourself to believe that the sun takes a significant space on the sky, though reality is that it lays claim on less than an angle of one degree of the firmament. So the penumbra effect of sun-light can safely be disregarded. In most other situations however it can't without interfering with the realism of the scene.



A fourth shadowing technique that haven't been mentioned yet, that can actually use soft shadows without any performance penalty whatsoever, is lighting mapping. Not only that, it is also very fast. Not surprisingly it is the most popular lighting and shadowing technique around. Unfortunately, it is completely static. It works fine for the static scene geometry, however, there won't be any shadows from the games characters, nor will the characters receive any shadows from the scene geometry. So while light mapping may be a good solution for static components of a scene it's insufficient for the dynamic components such as game characters, movable objects, dynamic light sources etc. It is fully possible though to combine light mapping with any of the other shadowing techniques for dynamic objects. Two possible solutions were constructed that solves the problem of soft shadows for dynamic objects.

The first solution may be the most obvious and is a fairly well-known technique. Since the light has physical size a lighting model that takes light size into account is needed. Our lighting model and shadowing techniques only recognize zero size point lights however. So the idea is simply to use many point lights to fill the volume that makes up the light source. It's not needed to evaluate the full lighting equation for each of these lights and average, the difference would hardly be visible, but the shadowing part will be implemented for each of these lights and the average used as the shadow factor. This idea should work regardless what shadowing algorithm in being used. In the prototype implementation the choice fell on using projected shadows, a choice one seldom would make otherwise, but since the shadowing algorithm doesn't matter and this is just a prototype the option of saving some time and effort while still illustrating the point made this the best alternative. Also, since there's no way to repeat a pass with other parameters or reuse a shader in RenderMonkey, and because the copy 'n' paste operation was not functional at the time of implementation it was desirable to keep it as simple as possible since the expectation was that many passes would be needed.

To use projected shadows one need to project the geometry of the model down into the plane the shadows are cast onto. This can be done with a simple matrix operation. Normally you would have evaluated the shadow projection matrix on the CPU and multiplied it into the view matrix in beforehand since it is constant for the whole scene, but this can't be done in RenderMonkey as mentioned before, so this task must be done in the vertex shader. The shadow projection matrix is a well-known matrix looking like this, where l is the light position and p is the plane the shadow is projecting onto and d is the distance from the light to the plane:

$$\begin{bmatrix} d - l_x p_x & -l_x p_y & -l_x p_z & -l_x p_w \\ -l_y p_x & d - l_y p_y & -l_y p_z & -l_y p_w \\ -l_z p_x & -l_z p_y & d - l_z p_z & -l_z p_w \\ -l_w p_x & -l_w p_y & -l_w p_y & d - l_w p_w \end{bmatrix}$$

To complete the projection the input vertex needs to be multiplied with this matrix and then the result multiplied with the view-projection matrix. Normally the shadow matrix should have been right-multiplied into the view-projection matrix passed to the shader in which case only one matrix multiplication would be needed in the vertex shader, but as said, this can't be done in RenderMonkey. Before composing the matrix the light position will be offset slightly for each pass by a pass-dependent constant vector. A total of fifteen passes were added with one sample of the light position in the middle of the light sphere and the rest evenly spread at the sphere's perimeter. In each pass the resulting shadow was rendered to a texture. Since the shadow factor is a scalar you can squeeze up to four factors into each RGBA texture. This isn't necessarily an optimization over using luminance or other single channel textures, however, there's no single channel 8 bit render target format supported on the Radeon 9700, so the best option was to use a full RGBA8 texture and pack the shadow factors into the different channels of the render texture by setting the proper color mask before each rendering pass. In the final pass eye-linear texture coordinates were generated for the plane quad on which the shadows were to be drawn. The render textures with the shadow factors were then sampled and the factors averaged. The quickest way to sum together the components of the four-component vector as sampled from the render textures is to simply take the dot-product of it and the constant one, which can be executed in one hardware instruction. Finally a simple lighting model was applied and the result multiplied with the averaged shadow factor.

This method has a number of shortcomings. First of all the result isn't quite as convincing as one might have hoped. Even with fifteen samples it is often clearly visible that the shadow is composed by many shadows. Instead of the impression of a smooth transition over the penumbra it tends to look stair-stepped in many cases. This problem could be solved by increasing the number of samples taken. However, the performance is of course inversely proportional to the number of samples. With 15 samples there is already a 15x performance hit over using sharp shadows, and one would probably need to increase it to maybe 30-60 samples in order to get universally good results.

Another technique was tried in order to overcome these problems. The idea is to instead of drawing many shadows and average to just draw one shadow and then perform a number of operations on this shadow in order to receive the same effect. The thought is that if one could just blur the shadow where it is supposed to be soft that ought to do the trick. In order to implement such a system one need to first come up with a way to define the softness of the shadow for a certain point. Thinking of a shadow edge one find that the shadow edge should be soft when the distance to the light is small, or when the distance from the object to the place where the shadow is cast is large. Conversely, if the distance to the shadow is small or if the light is distant the shadow should be sharp. Thus a sharpness factor could be defined as follows, where $l_{pos}$ is the position of the light, $o_{pos}$ is the position of the object, and $s_{pos}$ is the position of the shadow it casts:

$$f = \frac{\left\| l_{pos} - o_{pos} \right\|}{\left\| l_{pos} - s_{pos} \right\|}$$

To begin with the depth is rendered into a texture. For the same rationales as in the previous technique, and to more easily compare the techniques, projective shadows will again be used as the base shadowing system. For projective shadows the factor f can be evaluate in its entirety when rendering it into the texture since it's easy to find the point where the shadow is cast by completing the projection as usual. Would another more flexible shadowing system be used the denominator can be evaluated at a later point when the fragment is rendered.

The shadow is rendered to a texture as usual. The shadow is produced the exact same way as earlier, with one exception. The shadow isn't rendered to a screen-space texture as in the previous technique; it is rendered into a texture as mapped over the shadow plane quad the shadows are projected onto. This is done to better utilize the texture space.

The next step is to apply a flexible blur filter on the shadow texture according to the blurriness factor. 13 samples are taken semi-randomly spread within a unit circle. The kernel size is adjusted according to the blurriness factor. The softer the shadow is supposed to be the larger kernel size. To get better smoothing another blur pass is used to blur the blur. As a final step a basic lighting model is applied and the selectively blurred shadow factor is multiplied into it. At this point there should be nice soft shadows one may think. However, one problem remains. As it is now there is only have half the penumbra there should be. The outer part is directly missing and causes a sharp edge just after the softening has reached half shadow. Why is that? The reason is that the stored depth image has a sharp edge. At the edge there is a sharp contrast between fragments where strong blurring should occur to get the soft shadow and fragments just next to it that fell outside the object drawn into the depth texture. One simply must to look into the neighborhood and use the highest possible blurring factor. Finding the maximum in a neighborhood is nothing but a dilate filter. So a dilate filter implemented in a very similar way as the blur filter will be squeezed in. 13 samples are taken within a certain radius and the maximum value is returned. To get better results the dilate filter is first applied using a fairly large filter to get a good spread. Then possible cracks are filled with another dilate filter with smaller radius. It's required that it is dilated enough to let the shadow soften out and disappear before the edge to avoid any artifacts. Once the dilate filter sizes are adjusted the effect is pretty much done. What's left is to add a few parameters to tune to get the proper look. The ability to scale the filter size is a needed, and to control the shape of the blurriness the blur factor is raised to a power.

In conclusion it can be said that this technique provides significantly better results. There's no stair-stepping effect, just smooth transitions. On the other hand this technique is hardly rooted in a reality based model; unlike the previous technique there's no notion of a physical light size. Instead you need to adjust a number of parameters until it looks natural. However, the old saying in graphics still holds true: if it looks good, then it is good. It certainly looks quite good once tuned in, though the disadvantage is that it is a little trickier to tune in properly. When it comes to performance it should be a good deal faster than the previous method, especially if the previous technique uses as many as 15 samples or even more. No performance comparison was made however. Another good thing about this technique, the render of the depth into a texture may come effectively for free if used in combination with shadow mapping. Shadow mapping has to render the depth into the shadow map anyway, so this property could be ready to go forward with in such a case.

A few final words on using this technique; in this prototype implementation the shadow was drawn into a texture mapped directly over the full quad space. This was convenient for this sample, though in a real scenario this usage is hardly feasible. If a technique such as shadow mapping was to be used instead one would probably have to render the shadow into a texture from the normal camera's viewpoint. Screen-space blur and dilate may not yield optimal results on shadows that for instance are put on a sharp angle towards the camera; however, it isn't clear whether this has any real impact in practice. It is possible that one may have to take such things into account, which may make the implementation quite a bit more complex. Regardless, while this may be a problem the author certainly plans to go forward with this technique and make a deeper investigation of its utility in real-world applications.

## *Postprocessing*

Postprocessing is an exciting field, which have already been touched a little in this writing, with many interesting topics. A couple of samples digging a little further into a few subtopics of the subject were constructed.

### HSI-RGB transformation

There are loads of color spaces that are used in different applications. The RGB color space is probably the most common and that's for good reasons. The RGB color space is linear and very easy to perform operations on. Thus this is the color space of choice for TV's and monitors and for most kinds of graphics operations. That being said doesn't need to mean it's optimal, or that other color spaces are useless. What's optimal is dependent on the application and what task it is trying to solve. For instance in JPEG compression the YCrCb color space is used. This color space is made to fit the way the eye perceives light and has good properties for color compression. It separates the important luminance factor from the less important chrominance part, which can be sampled and stored at a much lower rate, thus cutting the storage space to half or less without any perceivable difference in quality on pretty much any image. In printing on the other hand it is common to use the CMYK color space, this because the base colors, black, cyan, yellow and magenta are cheaper and easier to produce than for instance red, green and blue.

A more interesting color space for graphics applications is the HSI color space. The HSI color space is composed of hue, saturation and intensity. The hue component picks a color from a spectrum going from red to yellow, to green, to cyan, to blue, to violet and back to red again. The hue is typically thought of as an angle. The saturation is how pure the color is. Red is a pure color (high saturation), while pink is not (low saturation). Saturation is thought of as a radius. The intensity is simply the lightness. Together these form a space looking something like a hexagon shape cone, the hue and saturation is put on the hexagon base where the intensity is addressed along the height axis. This color space is interesting because it can be used to enhance images or to extract information from it. If for instance an image looks bleach one can easily fix that in HSI space by simply scaling up the saturation. Color tints can be helped somewhat by adjusting the hue. When extracting information from an image a HSI color space is usually much more suitable to the task. Colors being close to each other in HSI space typically are more likely to belong to the same object than if they are close in RGB space. Extracting a brown hat from a gray background can be very tricky in RGB space, especially if there are shadows in the image, while in HSI space they are clearly separated in saturation.

To utilize the HSI color space a way to transform colors from RGB to HIS is needed, and then a way to transform back to RGB again. The formula’s are well-known and look as follows:

$$h = \cos^{-1}\frac{2r-g-b}{2\sqrt{(r-g)^2+(r-b)(g-b)}}$$

$$i = \frac{r+g+b}{3}$$

$$s = 1-\frac{\min(r,g,b)}{i}$$

As the formulas are written here s and i ranges from zero to one; h on the other hand being thought of as an angle ranges from –p to p. Typically one want h to be normalized too, in which case the negative angles simply will be mapped to their positive counterparts to get positive angles from 0 to 2p, and the result then divided with 2p to range from 0 to 1.

Converting back to RGB, assuming normalized h, is slightly trickier. The following expressions are needed:

$$w = \frac{\tan(2p(h-\frac{2\lfloor 3h\rfloor+1}{6}))}{\sqrt{3}}$$

$$x = (1-s)i$$

$$y = \frac{(3+3w)i-(1+3w)x}{2}$$

$$z = 3i-x-y$$

$$(r,g,b)=\begin{cases}(y,z,x), & h<1/3\\(z,x,y), & h>2/3\\(x,y,z), & otherwise\end{cases}$$

The goal put up for this sample was to load some kind of image, map it over the screen space and perform a conversion to HSI space. Then some operations such as scale and bias should be performed on the HSI components. Finally it should be converted back to RGB space and the results displayed. Two ways of doing this were implemented.

The first and most obvious way to do this is to simply implement the formulas as written above in a shader. As should be pretty obvious, the expressions are fairly complex, and with an instruction budget of 64 ALU instructions on the Radeon 9700 there might be a problem implementing the transition from RGB to HSI, the operations on the HSI data, and then the conversion back to RGB in one pass. This was tried anyway, though as expected the instruction budget was too small and such a shader would not compile to fit even with comprehensive optimizations and efforts trying to reduce complexity. Instead the operation needed to be split into two passes. The first pass samples the input image, converts it into HSI using the expressions above. This is simple enough to compile within the hardware limits and produced typical HSI output. The HSI output was then rendered it into a texture. In the second pass this texture was sampled and a scale and bias operation was performed on each component with settable parameters. The s and i components were then clamped to the [0, 1] range; h on the other hand being rotational it made more sense to just take the fractional part rather than clamping. The conversion back to RGB was then implemented according to the above expressions. As to create a more clean looking code the 3h component as used in the floor function for w was reused for the last component order operation such that comparisons could be made with whole numbers, 1 and 2, rather than 1/3 and 2/3. Outputting the resulting color and we're set.

The second implementation uses another method that significantly reduces the shader complexity. Considering that a transformation from a three-dimensional space to another three-dimensional space is what's being performed, which furthermore also is spatially limited to coordinates between zero and one, one could simply use a volumetric texture as a lookup table for our conversions. The RGB components can simply be used as a texture coordinate and at each position within the lookup texture a value of the coordinate transformed to HSI is stored. This way a simple texture lookup performs the full transformation. The scale and bias operations can then be performed and the same method used to transform back to RGB using another volumetric texture. Outputting the results and we're set; and no more than six rows of code was needed to complete the full task of which three were to perform the scale and bias operations.

The transformation texture is static and can be composed offline using the expressions above. The interesting thing is that unlike what was first thought this texture need not be of very high resolution to provide good results. Beginning with a 128x128x128 texture the results were undistinguishable from using the previous method using the mathematical functions directly. Reducing the size first to 64x64x64 and then to 32x32x32 this still held true. Chances are that even lower resolutions would do too, though with 32x32x32 the size it takes is just 128kb anyway, which is already even smaller than most normal 2D textures, so reducing it further doesn't have a whole lot to offer.

As a final conclusion it should be pretty obvious that this second technique is for most part superior to the first. The shader is significantly simpler and thus less demanding on the shader part. While the first technique is more precise the advantage is more of theoretical nature rather than practical.

## Edge detection

In many applications it is useful to track the edges in the image, either for a practical use such as in the Hough transform example mentioned above or for use in different kinds of effects. There are plenty of techniques to extract edges from an image, which can be learned in a standard course in industrial image processing. These are usually implemented on the CPU side, partly because many applications that use such operations aren't tuned for GPU operation at all, but also because until the last generation it has been hard to implement it on the GPU side due to limited fragment shaders. In the latest crop of graphics processors though they are often fairly straightforward to implement in a shader and a number of edge-detection algorithms were implemented and a practical usage of edge-detection to create the effect of a whiteboard drawing was produced.

The first method out is dilate and erode. The basic idea is to first apply a dilate filter on the source image, then apply an erode filter on the same source image. The edge-detected image is then the difference between the two. The implementation is as simple as the concept. In the first pass the scene to be edge-detected is rendered to a texture. In the second pass the dilate filter of this image is rendered to the framebuffer. The dilate filter is implemented by for each fragment sampling the eight surround texels in the source image and outputting the maximum value of these. The erode filter works by the same principle; the only difference is that it returns the minimum value instead. Since the same samples are needed for both filters, only different operations on them are used, both filters can be put into the same shader and the difference taken right there. The result in the framebuffer is thus dilate minus erode and our edge-detected image is done.

The next method is to use the Laplace filter. The Laplace filter approximates the second order derivate of the image. This gives double response on image edges, that is, it reacts on the places where the slope changes, which is where the edge begins and ends. It reacts with a positive response on the beginning or a rising edge and a negative response at the end, and negative response on the beginning of a falling edge and a positive response at its end. Often the absolute value of the Laplace filter is taken to preserve all slope changes. An alternative approach is to normalize it by adding 0.5 instead and to let negative responses go towards black and positive response towards white.

With the Laplace filter an edge in the source image may appear as two different edges if it is wide or blurry. If on the other hand the edges are sharp the resulting edge-detected image will have thick edge lines if the absolute value is used or as double-drawn lines if it is normalized. The advantage of the Laplace filter is that it is cheap. It requires only five samples. The filter kernel looks like this:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Implementing the filter is a piece of cake. Take the five samples, multiply the middle sample with minus four and sum together. Optionally one may decide to scale it to enhance or reduce the contrast in the image. Finally the choice is between taking the absolute value or to normalize it, and the edge-detected image is done.

Another edge-detection technique is the Sobel filter. The Sobel filter evaluates the first order derivate, the slope in the image in other words. It has a single response on edges and reacts to the whole area of change. Unlike the Laplace filter the Sobel filter not only provides information on the rate of change, it also provides information on in which direction it is changing. On the other hand it is more expensive than the Laplace filter.

The Sobel filter is really two filters; one in x direction and one in y direction. Each filter requires six samples; however, there is some overlapping so that only eight samples are needed to evaluate them both. Combining these filters the total rate of change can be evaluated and if desired direction vectors or direction angles may be computed. The filter kernels look as follows:

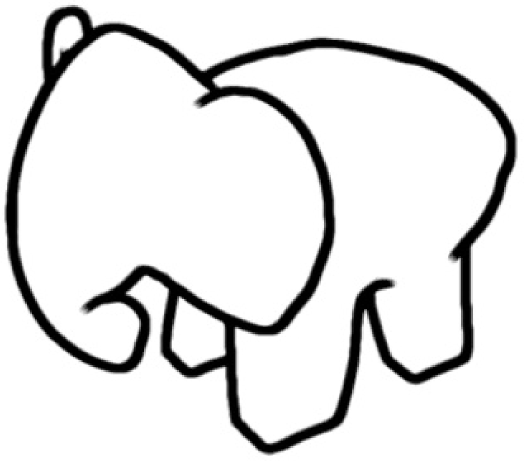
$$\begin{array}{cc} SobelX & SobelY \\ \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} & \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \end{array}$$

Implementing these filters isn't much harder than the Laplace filter. In the first step the eight samples are taken from the texture. These are then composed together according to these filter kernels into x and y components of the direction vector. The rate of change, or the edge factor as one may also call it, can then be computed as follows:

$$rate = \sqrt{x^2 + y^2}$$

Outputting the rate of change for each pixel and the edge-detected image is done.

For the practical use of edge-detection in a graphics application as mentioned earlier to create the effect of drawing on a whiteboard the choice fell on using Sobel filters. A whiteboard drawing contains the edges of the object being drawn, so one could say we're half done already, but just half though. What's left is to ensure all the edges are included, not just the contour. If only the contours are desired one may render the object to be edge-detected into a texture with a constant color. If internal edges are desired that can't be done however. A contrast between parts of the object that are in front of other parts of the object must be created. The natural way to do that is to render the depth into the texture. This way an edge will be created wherever there is a large difference in depth over a small amount of pixels. This works well on fairly smooth and uncomplicated



objects, like the elephant model that is being used for this sample. Color-coding different parts of the object is another idea that can be used for hard and complicated models made up of many easily separable parts such as an engine model. It's hard to use for smooth curvy objects though.

Whenever it is decided for a certain fragment that it is on an edge equally sharp lines are desired regardless of whether the depth difference is moderate or very large. Thus the rate of change can't be used directly. What's needed is to threshold the value. When the rate of change is larger than a certain threshold the fragment is deemed to be on an edge and full intensity is used. Since the threshold value is a constant a simple trick to improve performance can be applied. Instead of thresholding the rate of change its square will be thresholded against the square of the threshold value. This way the square root can be removed and two hardware instructions saved.

Unfortunately though, to threshold the value causes another problem: aliasing. A threshold function has infinite frequency at the point where the step takes place, which is the source of the aliasing. Since this is shader generated aliasing multisampling antialiasing can't do anything about it. Thus we need to somehow antialias it ourselves. Considering that only a sketchy and undetailed image is being used one may just as well simply apply a slight blur filter to soften the edges without otherwise hurting the image quality in any way. Using a blur filter the edges can also be fattened and a better representation be had of how the lines would look as drawn on a whiteboard. A blur filter similar to those used earlier was applied. As in earlier cases, 13 samples within a circular neighborhood were taken. Instead of just averaging them and outputting it some minor changes were made in the end to complete the effect. Up until now it has been a blackboard rather than a whiteboard, so the color needs to be inverted to begin with. Instead of using the average of the samples the samples was added together and multiplied with a settable constant. This way the hardness of the image can be adjusted. Using a high hardness the image looks fairly sharp and by using a low sharpness one can get the impression of the image having been drawn with an almost dried out felt pen. Completing this final step and we get a picture like the screenshot above.

## Conclusions

This report has covered the topic of high level shading and met the goals set up for this thesis. The OpenGL glslang shading language has been evaluated and found to be a solid base for future shader language revisions to stand on. It is found that the philosophy behind the OpenGL shading language is generally better and more properly set up for the future than DirectX9 HLSL, though it's been concluded that both languages are significantly superior to any low level shading language in existence. It has also been concluded that high level shading is soon to phase out any low level shading language, just as low level languages for CPUs only employs a minimal part of the workforce of programmers today, and just a few months after the completion of this work the trend has already begun to catch on among a few game developers. The ATI alpha implementation of the OpenGL shading language was found to be in fairly good condition considering its young age and the complex task it is facing, though large parts of the shading language remain unimplemented. There are lots of work left on performance and conformance, but the driver was mostly functional when utilizing the subset of the functionality that was implemented.

The Ashli project was found to be an interesting view into the where the state of the art in hardware shading is today and how it relates to the state of software shading. It was concluded that there is still a large gap to the software rendering world, though strong progress is being made and the hardware rendering world quickly approaches. Even today a significant subset of the existing shaders can already be shaded in hardware, and a large part of the remaining will be possible to run once dynamic branching is added to the fragment pipeline and various resource limitations (such as number of temporaries and instruction) are overcome. It was found that the Ashli interface had some work on it left to be done to be fully functional, and some slight modifications to the interface would be helpful. It was further concluded that adding functional support for hardware shading languages such as DX9 HLSL and glslang would vastly improve the utility of Ashli. Integrating Ashli into third party tools was found necessary for it to reach its full potential, and the investigation showed that integrating it into ShaderMan was both possible and useful. Finally, performance, usability and features would need improvement.

The RenderMonkey toolset was found to be very useful for prototyping and research work; though in beta version 0.9 have a number of flaws and limitations. The productivity using RenderMonkey was very high and a large number of workspaces could be produced and multiple techniques tried out in a quite short amount of time. Of the techniques produced many resulted in that the intended visual effects could be achieved and many interesting fields were opened for investigation. Techniques useful for many areas of real-time rendering were created, for instance a technique for convincing volumetric lighting useful for gaming situations, as well as water simulation, fires, flashes and other phenomena that can enhance the gaming experience. Cinematic effects such as glow and soft shadows were implemented and found to be realistic for real-time rendering. Simulations of a number of materials such as wood, fabric and glass were done and the field of postprocessing in image space touched with techniques for color space conversions and a number of edge-detections algorithms. Finally, a few utilities of hardware shaders in industrial applications were analyzed and room was left for future expansion of the ideas. Limitations and performance attributes of all techniques have been discussed and analyzed according to the needs of the respective field they were intended for. In conclusion, while many of the techniques are quite useful in the state they appear in this text, most of them also have room for future research and development, as have also been discussed in the final words for each of the techniques as they appear in this text.

# Appendix A – Glossary and abbreviations

## *Glossary*

**Aliasing –** Artifacts such as pixel popping, shimmering etc. Aliasing is caused by undersampling. Undersampling unavoidably occurs at geometry edges since a geometric edge is a discontinuity and thus by definition has infinite frequency. Aliasing is solved or reduced by applying antialiasing.

**Antialiasing** – Measures taken in order to reduce aliasing. Common techniques are multisampling and supersampling.

**Billboarding** – Aligning something to always face the viewer; same as "screen-aligned".

**Dependent texture read** – A texture read that is dependent on results from earlier texture reads.

**Direct3D** – A graphics API originally released in 1995 under the common idea around that time that OpenGL was too complex an API for consumer level hardware (which was soon proved to be false though). Direct3D is part of the DirectX toolset and is controlled and developed by Microsoft. It has been frequently updated and as of this writing the current version is 9.0. The first API revisions were not that successful and generally viewed as unnecessarily cumbersome. It's a common view that version 5.0 were the first "useable" version. Today OpenGL and Direct3D are generally comparable in terms of features support and easy of use.

**Display List** – An assembly of many drawing commands packed into a single object that can be called to issue all stored commands.

**Fragment –** A "pixel" on its way through the fragment pipeline. A fragment, unlike a pixel, consists of more than just color data. It also has depth, texture coordinates, stencil and other data. Once it is written to the frame-buffer it becomes a pixel.

**Fragment shader** – A script that processes a fragment to define its final color and optionally its depth. A fragment shader takes its input from textures, shader constants (also called uniforms or attributes), and texture coordinates provided by the vertex shader.

**Framebuffer** – The pixel buffer that is being rendered to.

**GL_ARB_fragment_program** – The OpenGL extension that adds fragment shading capabilities to OpenGL.

**GL_ARB_vertex_program** – The OpenGL extension that adds vertex shading capabilities to OpenGL.

**Mipmapping** – To use a set of reduces size textures, say 256x256, 128x128, 64x64 … 1x1, to ensure that the pixel / texel ratio is as close to one as possible. This is done to get rid of the aliasing that would otherwise occur. The most suitable mipmap level will be used for each fragment. With trilinear filtering the two closest mipmap levels are chosen and interpolated between.

**Multisampling** – An antialiasing technique that only takes additional samples at geometry edges and consequently only antialiases edges. It is the preferred antialiasing method since it unlike supersampling doesn't waste effort on areas that shouldn't need antialiasing, thus saves a good deal of fillrate. There are also situations where antialiasing is undesired, such as when using tiny pixel aligned fonts, in which case supersampling will hurt more than help meanwhile multisampling will leave it untouched.

**OpenGL** – A graphics API originally released in 1992 by SGI. It has gone through a set of revisions, version 1.1 in 1995, version 1.2 in 1998, version 1.3 in 2001 and version 1.4 in 2002. There is currently a version 2.0 proposal under revision and investigation by the ARB.

**Pixel** – A single element in a frame-buffer.

**Pixel shader** – See Fragment shader. This term that Microsoft calls a fragment shader is incorrect as the fragment shader is fed with a fragment and outputs a fragment. The fragment shader never deals with pixels. The fragment will pass through a number of stages later in the pipeline, such as blending, alpha testing, depth testing, stencil testing and may optionally be discarded alternatively become a pixel as it's written to the frame-buffer.

**RenderMan** – A shader language commonly used in offline software rendering.

**Shader** – A script that performs shading. In real-time graphics it is a collective word for vertex and fragment shader.

**Shader constant** – An application provided parameter that is passed to the shader. See also Uniform.

**Shading** – Processing a vertex or pixel as to define its final position, color, depth etc. through a shader.

**Swizzle** – To rearrange, distribute and mask components at read or write in a shader. This is done by appending something like .xyyw or .rgba to the argument. For instance to reverse the components in the variable var, one could write MOV var, var.wzyx; in the GL_ARB_fragment_program language.

**Texel** – A texture element. Generally referring to a filtered element as it's being accessed in the fragment pipeline.

**Uniform** – Another word for a shader constant. In GL_ARB_vertex_program and GL_ARB_fragment_program it's called constant, while OpenGL2 calls it uniform.

**Vertex shader** – A script that processes geometry before it's clipped to the viewport and fed to the rasterizer. It takes input from streams of data such as vertices, normals, texture coordinates and vertex attributes. It also takes input from application provided constants (also called uniforms).

## *Abbreviations*

**API** – Application Programming Interface  
**ARB** – Architecture Review Board  
**Ashli** – Advanced Shader Language Interface  
**CPU** – Central Processing Unit; the main processor in a computer.  
**DX** – Short form of DirectX.  
**DX9** – Short form of DirectX 9.  
**GL** – Short form of OpenGL.  
**GL2** – Short form of OpenGL 2.  
**GPU** – Graphic Processing Unit. Refers to a graphics card or graphics chip.  
**HLSL** – High Level Shading Language  
**IHV** – Independent Hardware Vendor  
**PS** – Pixel shader  
**VAO** – Vertex Array Object. Refers to the GL_ATI_vertex_array_object extension.  
**VAR** – Vertex Array Range. Refers to the GL_NV_vertex_array_range extension.  
**VBO** – Vertex Buffer Object. Refers to the GL_ARB_vertex_buffer_object extension.  
**VPU** – Visual Processing Unit. Same as GPU, just hipper.  
**VS** – Vertex shader

# Appendix B – Additional mathematical reasoning

## *Perlin Noise*

### Introduction

Ken Perlin did some great job in this area [8] and the concept Perlin noise was named after him. Conceptually Perlin noise is a range-limited and band-limited continuous random function. The randomness exists over spatial locations; the same input always yields the same result. A location is fed into the noise function and a value between -1 and 1 is returned. Perlin noise, often just referred to as noise, may exist in one, two, three or more dimensions and is continuous in all dimensions. The concept is much more useful than it may first appear and after working with it for a while its utilities becomes much clearer. In pretty much any application where there's some kind of semi-random behavior noise can be used to implement it in a straightforward way. It may be everything from a water surface to fire, subtle face movements, trees moving in the wind, terrain topology, patterns in wood, marble etc; the list goes on.

### Concepts and implementation

There are a number of ways of implementations of noise. Typically an array is initially filled with random numbers between -1 and 1 using a normal pseudorandom function such as the rand() function in C/C++. Then the noise is generated by interpolating between the stored values using the s-curve as discussed later on. Typically this is done in a way that generates non-repeating noise, at least over a significant distance. The simplest way is to just create a large grid structure and directly map into it and interpolate. Ken Perlin's original implementation from 1983 however used a single one-dimensional array which he with a few tricks could map two- and three-dimensional into in a consistent way.

There is also the concept of turbulence. Turbulence is a sum of noises of different frequencies where the contribution is scaled down with the frequency. The base low-frequency noise gets high amplitude while the highest frequency noise fills in some fine detail. Turbulence is usually the preferable kind of noise to use since it has both the high amplitude base, but also the fine details.

Often you want tileable noise and turbulence, that is, noise that repeats itself over a certain period. This can be useful for instance if noise is packed into a texture and the texture coordinate is wrapped to get rid of the edge that would otherwise appear as the texture is repeated. Tileable noise can be created by mirroring the noise function over the period in all dimensions and interpolate between the combinations of mirrored and non-mirrored there is. For a one-dimensional noise function there are two combinations, four for two dimensions and eight for three dimensions etc. A tileable noise function over two dimensions could for instance be implemented as follows:

```
float tilableNoise2(float x, float y, float w, float h){
    return (noise2(x,     y)     * (w - x) * (h - y) +
            noise2(x - w, y)     *      x  * (h - y) +
            noise2(x,     y - h) * (w - x) *      y  +
            noise2(x - w, y - h) *      x  *      y) / (w * h);
}
```

## Noise in vertex and fragment shaders

Being able to evaluate noise in a shader is of highest importance and extremely useful. Unfortunately, neither the vertex shader nor the fragment shader has any support for any noise functions in current hardware. Fortunately, the problem can for most parts be solved for fragment shaders. The solution is to pack the noise or turbulence function into a suitable texture. Two-dimensional noise goes into a regular 2D texture while three-dimensional goes into a volumetric texture. Practical tests show that the resolution needn't be very high either for most applications. An eight bit 128x128x128 luminance texture is usually more than enough and takes only 2MB memory. Since the texture can be shared between all shaders that are using noise there's no need to use more than one texture either. A price of two megabytes memory for the usefulness of noise should be considered cheap.

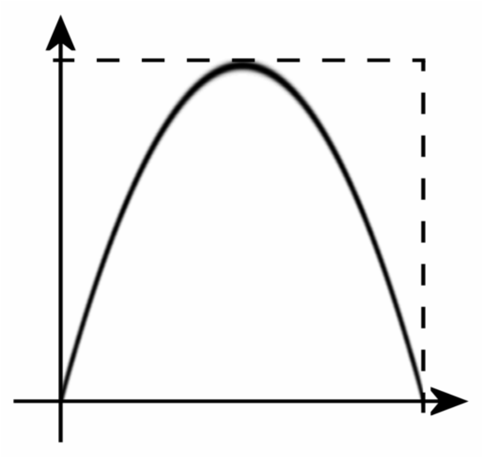
Using textures for the noise has a number of advantages too. The mipmapping will help get rid of aliasing that might otherwise have been caused by undersampling the noise function if such a function would have been available and used directly. Accessing a texture is also very quick compared to evaluating a complex noise function and especially the turbulence function that's even worse and typically many times slower. The disadvantage though is that you're forced to either use a limited range or to use tileable noise. In many applications this doesn't matter, however, there are a number of cases where the repeating behavior can be quite obvious. By carefully taking this into account when constructing shaders the problem can usually be solved. One may for instance in some cases take another non-repeating parameter, such as the fragment position, into the equation and break the repeating behavior. One may also use more than one noise function and map them at different frequencies that doesn't line up with each other at any even multiples and then use the difference or average instead.

For the vertex shader there is a more problematic situation. There's no access to textures in the vertex shader in the current generation hardware. It is very likely that this will appear in the next generation however, which will solve the problem for the vertex shader in the same way as it can be solved for the fragment shader today. Until then there are a number of options. There have been samples showing a way to implement limited noise functions by packing a lot of random parameters into vertex shader constants and perform interpolation in the shader. This is very computation heavy though and consumes lots of vertex constants and eats loads of instructions. In some cases, as is discussed at a few points in this writing, you can get away with using functions with similar behavior such as the sine and cosine with some careful tuning.

## Remapping functions

### Bounce

It is often desired to remap values between 0 and 1 to a smooth curve going from 0 up to 1 and back to zero again according to the illustration. As anyone with a decent amount of experience should figure out fairly quickly what we want seems to be a second order polynomial. So, we assume that:



$$f(x) = ax^2 + bx + c$$

We know the following:

$$f(0) = 0$$
$$f(1) = 0$$
$$f(\tfrac{1}{2}) = 1$$

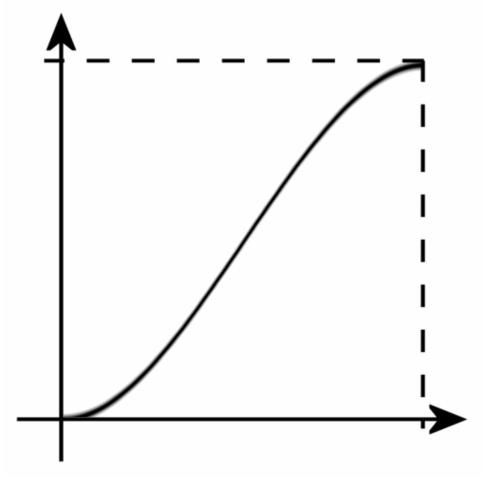
These statements lead to the following conclusions by inserting the values in the function:

$$\left.\begin{aligned} &c = 0 \\ &a + b + c = 0 \\ &0.25a + 0.5b + c = 1 \end{aligned}\right\} \Rightarrow \left.\begin{aligned} &a + b = 0 \\ &0.25a + 0.5b = 1 \end{aligned}\right\} \Rightarrow \left.\begin{aligned} a &= -4 \\ b &= 4 \end{aligned}\right\} \Rightarrow f(x) = -4x^2 + 4x = 4x(1 - x)$$

Ergo, the bounce function is f(x) = 4x(1-x).

### S-curve

Another common remapping function one may desire is the s-curve. One want to remap the 0…1 into 0...1, though not linearly, but rather begin and end with a slope of zero and a smooth curve in between as in the illustration. This looks like it’s between the negative and positive bump in a third degree polynomial. So, we assume that:



$$f(x) = ax^3 + bx^2 + cx + d$$

Thus:

$$f'(x) = 3ax^2 + 2bx + c$$

We know the following:

$$f(0) = 0$$
$$f(1) = 1$$
$$f'(0) = 0$$
$$f'(1) = 0$$

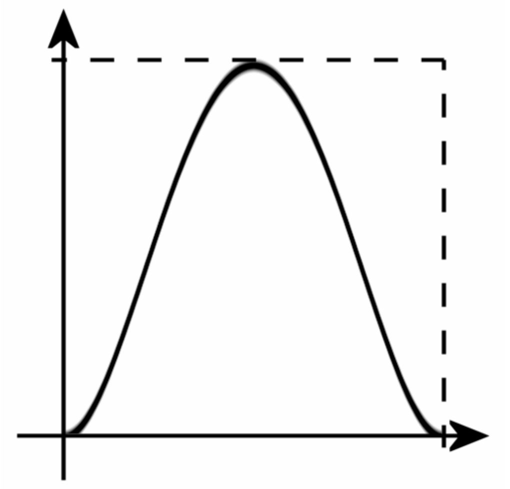
By inserting these into f(x) we conclude:

$$\left.\begin{array}{l} d=0 \\ a+b+c+d=1 \\ c=0 \\ 3a+2b+c=0 \end{array}\right\} \Rightarrow \left.\begin{array}{l} a+b=1 \\ 3a+2b=0 \end{array}\right\} \Rightarrow \left.\begin{array}{l} a=-2 \\ b=3 \end{array}\right\} \Rightarrow f(x)=-2x^3+3x^2=x^2(3-2x)$$

Therefore, the s-curve function is f(x) = $x^2$(3-2x).

## Smoothbump

At times one may want the idea of the bounce function, but also want the slope to be zero at the beginning and end of the 0…1 interval as to create a smooth bump going from zero to one and back to zero again. Repeating this function won't cause any sudden discontinuities in velocity unlike the bounce function. What we want is something like in the figure. This appears to be something like between the bumps in a fourth order polynomial function. So we assume:



$$f(x)=ax^4+bx^3+cx^2+dx+e$$

Therefore:

$$f'(x)=4ax^3+3bx^2+2cx+d$$

The desired properties are:

$$\begin{array}{l} f(0)=0 \\ f(\tfrac{1}{2})=1 \\ f(1)=0 \\ f'(0)=0 \\ f'(1)=0 \end{array}$$

Inserting this into f(x) leads to these conclusions:

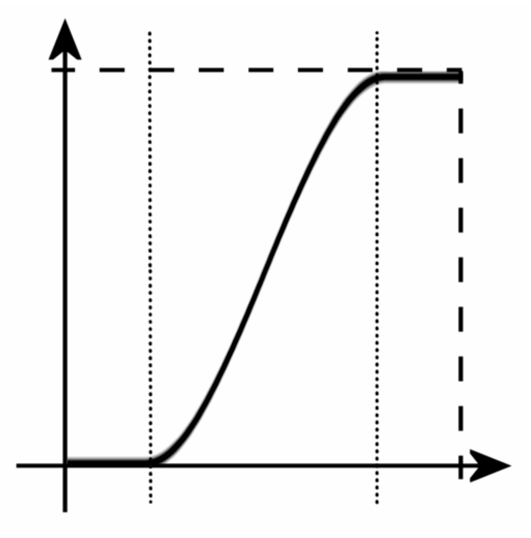
$$\left.\begin{array}{l} e=0 \\ 0.0625a+0.125b+0.25c+0.5d+e=1 \\ a+b+c+d+e=0 \\ d=0 \\ 4a+3b+2c+d=0 \end{array}\right\} \Rightarrow \left.\begin{array}{l} 0.25a+0.5b+c=4 \\ a+b+c=0 \\ 4a+3b+2c=0 \end{array}\right\} \Rightarrow \left.\begin{array}{l} a=16 \\ b=-32 \\ c=16 \end{array}\right\} \Rightarrow$$

$$\Rightarrow f(x)=16x^4-32x^3+16x^2=16x^2(x^2-2x+1)=16x^2(x-1)^2=(4x(1-x))^2$$

Ergo, f(x) = $(4x(1-x))^2$, which interestingly enough is the bounce function squared.

## Smoothstep

The normal step function is a well-known mathematical function with many utilities. In many applications however another smoother step function is desired that doesn't abruptly spike from zero to one, but rather goes from zero to one over a certain range in a smooth curve, as illustrated in the diagram.



The range $[x_0, x_1]$ as marked with thin dotted lines in the illustration will need some kind of function that begins and ends with a first order derivate of zero. Over the interval it needs to go from zero to one. This holds true for the s-curve, except that the s-curve's range always is [0, 1]. So we need to normalize the x-value before passing it to the s-curve function. We also want the function to stop at zero or one infinitely outside this range. The quickest way to do this is to just clamp the normalized value of x to the [0, 1] range. Mathematically expressed the smoothstep function is:

$$f(x) = \begin{cases} 0, & x < x_0 \\ 1, & x > x_1 \\ a^2(3-2a), & otherwise \end{cases}$$

*where*

$$a = \frac{x - x_0}{x_1 - x_0}$$

Codewise in a graphics high level language such as DX9 HLSL this can be expressed in quite simple terms as

```
float smoothstep(float x, float x0, float x1){
   float a = saturate((x – x0) / (x1 – x0));
   return a * a * (3 - 2 * a);
}
```

In the general case this need not expand into more than 7 hardware instructions. In the more common case where x0 and x1 are constants no more than 4 hardware instructions are needed.

## References

1. William Shoaff (2000-08-30). A short history of computer graphics.
Retrieved 2003-08-09 from the World Wide Web. URL:
http://www.cs.fit.edu/~wds/classes/graphics/History/history/history.html

2. Thomas Monk (2003-04-18). 7 years of graphics.
Retrieved 2003-08-09 from the World Wide Web. URL:
http://accelenation.com/?ac.id.123.1

3. ARB_vertex_buffer_object (2003-01-21).
Retrieved 2003-08-09 from the World Wide Web. URL:
http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_buffer_object.txt

4. ATI Technologies (2003). HLSL_NoFX.
Retrieved 2003-08-09 from the World Wide Web. URL:
http://www.ati.com/developer/samples/dx9/HLSL_NoFX.html

5. ATI Technologies (2003). Depth of field.
Retrieved 2003-08-09 from the World Wide Web. URL:
http://www.ati.com/developer/samples/dx9/DepthOfFieldHLSL.html

6. Jonathan D. Cohen (1998-08-13). Appearance-preserving simplification.
Retrieved 2003-08-09 from the World Wide Web. URL:
http://www.cs.unc.edu/~geom/APS/

7. Emil Persson (2003-05-23). Detail preserving simplification.
Retrieved 2003-08-09 from the World Wide Web. URL:
http://esprit.campus.luth.se/~humus/?page=3D&id=34

8. Ken Perlin (No date provided). Some web pages for Ken Perlin.
Retrieved 2003-08-09 from the World Wide Web. URL:
http://www.noisemachine.com/

9. Jason L. Mitchell (2003). Real-time 3D scene post-processing.
Retrieved 2003-08-09 from the World Wide Web. URL:
http://www.ati.com/developer/gdc/GDC2003_ScenePostprocessing.pdf

10. David Blythe (1999-08-06). Anisotropic lighting.
Retrieved 2003-08-09 from the World Wide Web. URL:
http://www.opengl.org/developers/code/sig99/advanced99/notes/node154.html

11. John Carmack (2002-06-27).
Retrieved 2003-08-09 from the World Wide Web. URL:
http://www.bluesnews.com/cgi-bin/finger.pl?id=1&time=20020627230700

12. William R. Mark and Kekoa Proudfoot (2001). The F-Buffer: A Rasterization-Order FIFO Buffer for Multi-Pass Rendering.
Retrieved 2003-08-09 from the World Wide Web. URL:
http://graphics.stanford.edu/projects/shading/pubs/hwws2001-fbuffer/

13. RenderMonkey toolsuite (2003).
Retrieved 2003-08-09 from the World Wide Web. URL:
http://www.ati.com/developer/sdk/radeonSDK/html/Tools/RenderMonkey.html

14. Emil Persson (2003-05-23). Volumetric lighting II.
Retrieved 2003-08-09 from the World Wide Web. URL:
http://esprit.campus.luth.se/~humus/?page=3D&id=36

15. Emil Persson (2003-05-23). Fire.
Retrieved 2003-08-09 from the World Wide Web. URL:
http://esprit.campus.luth.se/~humus/?page=3D&id=33

16. Emil Persson (2003-05-23). Electro.
Retrieved 2003-08-09 from the World Wide Web. URL:
http://esprit.campus.luth.se/~humus/?page=3D&id=35

17. Emil Persson (2003-05-23). Glow.
Retrieved 2003-08-09 from the World Wide Web. URL:
http://esprit.campus.luth.se/~humus/?page=3D&id=32