CHALMERS

Designing a Shading System

DAVID LARSSON

Master's Thesis

Computer Science and Engineering Program

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Division of Computer Engineering
Göteborg 2005

Designing a Shading System

David Larsson

December 7, 2005

Abstract

This thesis focuses on the interaction between light sources and materials which is often called shading. It describes the design of the shading system implemented for the second version of the Turtle renderer that supports the flexibility and power of the Maya rendering packages shaders. It discusses different approaches to shading, how rendering of passes can be incorporated, multi threading issues and problems with secondary rays in shading systems.

1 Introduction

1.1 Background

Computer generated images and animations are getting more and more common. It is used in many different contexts such as movies, commercial, medical visualization, architectural visualization and CAD. Advanced ways of describing surface and light source properties are important to ensure artists are able to create realistic and stylish looking images. Even when using advanced rendering algorithms such as ray tracing, shading may contribute with a large part of the image creation time. Therefore both performance and flexibility is important in a shading system.

1.2 Organization of the Thesis

The thesis is divided into the following parts

- Background
  Describing the context the shading system is going to be used in. People already familiar with different rendering architectures and shading in general can skip the rest of this section.
- Previous Work
  Previous approaches to shading are described briefly.

- Design Aspects
  Discussion of the previous work, external requirements, properties and problematic aspects of shaders.

- Implementation
  Details on the system design and implementation.

- Conclusions and Future Work
  Here the results, possible improvements and ideas for future work presented.

1.3 The Rendering Equation

The rendering equation was introduced by Kajia[1] and describes how light is transported between surfaces. Most rendering with some kind of physical plausibility are possible to express with this equation, however in most methods approximations and simplifications are used in order to gain performance and stability. The equation looks like (this formulation is taken from[2])

$$L(x \rightarrow \Theta) =$$

$$L_e(x \rightarrow \Theta) + \int_{\Omega_x} f_r(x, \Psi \rightarrow \Omega) L(x \leftarrow \Psi) cos(N_x, \Psi) d\omega_\Psi$$

Where x is the point where the light is computed, Θ is the direction towards the viewer, Ψ the direction towards the light contribution N_x is the normal at the point x, Ω is the hemisphere over the point x centred in the normal.
L$(x \rightarrow \Omega)$ represents the radiance[2] transported from point x in the direction Ω. $L_e(x \rightarrow \Theta)$ is the emitted radiance from point x in direction Θ. $f_r(x, \Psi \rightarrow \Omega)$ is the Bidirectional reflectance distribution function (BRDF see 1.3.1) for the point x with the view direction Ω and light direction Ψ.
The rendering equation says that the outgoing light from a point in a certain direction is the integral of all incoming light in the hemisphere multiplied by the BRDF for the outgoing light and the incoming light direction multiplied with cosine of the angle between the normal of the point and the direction towards the light contribution. This is a very brief description, for more information look at[2] or[1].
Rendering an image can be thought of as evaluating radiance reflected towards a virtual image plane from a set of objects reflecting and emitting lights.

1.3.1 BRDF

The BRDF is a function of light direction, view direction and wavelength that describes how a surface reflects lights. Typical examples of BRDF's:

- Perfect reflection, where the BRDF is 1 for the view direction and the reflection of the view direction, and 0 for all other pairs of incoming and outgoing directions.

- Diffuse, where all pairs of light and view directions have the same value.

To model reality accurately, BRDF's more complex than this are needed. More advanced models are described in section 1.5.2. The wavelength dependency gives the ability to let light of different wavelengths be distributed differently. A diffuse red wall will only reflect light in the red part of the spectrum and absorb the rest of the light. For the sake of simplicity most computer graphics applications treats colors as triples of red, green and blue light (usually referred to as RGB colors) rather than wavelengths. This however cannot capture all effects. This simplification is described in[3]. As an example of the incompleteness of the model, yellow light may be light in the yellow spectrum or a mix of red and green light, the eye cannot tell the difference. If simulating dispersion in a prism, this light would behave differently depending on if it is a mix or not.

1.4 The Rendering Pipeline

A rendering pipeline is a method of transforming a conceptual 3d world into an image. There are many different ways of doing this, but a shading system is an important part of all of them. A well designed shading system should be disconnected from the rest of the renderer as much as possible to make it possible to use it in many different rendering pipelines.

1.4.1 Shading

Shading is a wide expression. The word shading implies light calculation but it's nowadays being used not only for lighting but all customizable parts of a rendering pipeline. Typical "shadable" parts of a rendering pipeline are

- Material Types
- Light Sources
- Camera Models
- Volume Properties
- Geometry Displacement

This project is focused on light calculations and material descriptions, which are the most commonly "shadable" part in a rendering pipeline. Material shaders tends to work on surface fragments, which are infinitely small surfaces with properties such as position in the world, a normal, etc. By using these properties, it is possible to calculate how light is transported.
Shading is an expensive part of rendering, Wald identified it as the biggest individual bottleneck in his high performance renderer[5].

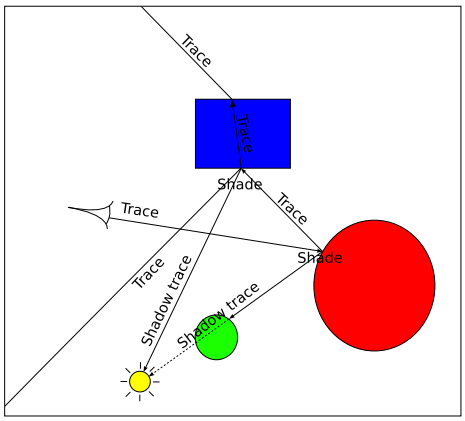


Figure 1: Whitted Style Raytracing Overview

1.4.2 Raytracing

Raytracing[6] is a general way to render images. The original raytracer created by Whitted works by tracing rays from a virtual image plane, through a pinhole camera into the world of 3d objects (see figure 1). The closest intersection point is used as input to a shading calculation. Depending on the surface properties and light configuration, the shader recursively generates "secondary rays" to sample reflections, refractions and making light sources cast shadows.
A more advanced version is distributed raytracing[7] which traces multiple rays to handle features such as motion blur, soft shadows and depth of field. The algorithm has also been extended to handle full global illumination[1]
It's a general and elegant solution to rendering but is traditionally considered computationally expensive. Much of its power lies in the fact that the only way to access the geometry is through the simple ray intersection operation that can be used both for primary and secondary effects.

1.4.3 Scanline Rendering

Scanline rendering is a rough categorisation of rendering algorithms working in the opposite direction of ray tracing. Rather than asking the question of which object is visible on a pixel it asks the question of which pixels an object covers. There are a lot of different flavours and techniques for scanline rendering, for instance the REYES architecture[8] and most graphics hardware. Objects are rasterized on the screen using a perspective transform that mimics a camera. The generated fragments are then shaded.
Compared to ray tracing rasterization is often fast in simple situations but secondary effects like reflections and refractions are difficult to handle. Generally it is done using pre generated images (environment maps[9]). However these approximations are giving incorrect results. More advanced scanline renderers are hybrid renderers supporting ray tracing for this kind of effects. When ren-

dering scenes with global illumination (see 1.5.2), only a small percentage of the rendering time is spent on calculation direct light for points directly visible to the camera and therefore the performance difference between scanline renderers and raytracers are decreasing.

1.5 Lighting

1.5.1 Traditional Lighting

Lighting a surface fragment in a traditional setup is done by evaluating the light emitted and reflected towards the viewer from all non shadowed light sources for the surface fragment. The OpenGL rendering pipeline uses a similar approach. This kind of lighting suffers from the fact that only light reflected directly from the light source to the viewer is considered. Indirect light bounces, such as light reflected on a mirror to the fragment being lit, is left out.

1.5.2 Global Illumination

More complete solutions to the rendering equation considers indirect light, such as caustics from reflective and refractive materials and light bouncing off diffuse surfaces. Typical algorithms that handle this or parts of it are path tracing[1], radiosity[10] and photon mapping[11].
Path tracing and derivations from it, such as bidirectional path tracing[12] and Metropolis light transport[13], are interesting algorithms since they are unbiased and handle all light contributions. An unbiased renderer is one that given enough rendering time, will actually converge to the correct solution of the rendering equation for the image. The errors shows up as noise in the image. The problem with these algorithms is that they are generally very slow and produces noisy results.
Radiosity[10] is a way to handle diffuse interreflections between surfaces in a noise free way. It works by subdividing the geometry into patches and calculate how much light is transported between the patches. A variation of this algorithm is final gathering[14] which samples irradiance at points in an irradiance cache. This cache is then used to look up nearby irradiance values to interpolate a reasonable irradiance value. This algorithm has the nice property that it doesn't have to subdivide or in any way modify the geometry while rendering, making it is easy to integrate in a raytracing framework.
Photon mapping[15] is an algorithm that handles all light contributions by emitting "photons" from light sources and letting them bounce in the scene. The "photons" are saved in a three dimensional data structure called photon map. Calculating light for a point is done by looking up photons in the vicinity of the point and weighting them to compute the influence. Photon mapping supports all illumination effects, but generally it is used in combination with a traditional lighting setup that calculates the direct light and photon mapping for the indirect light. It may also be used together with a final gather solution that handles the first bounce of indirect light and letting photons contribute

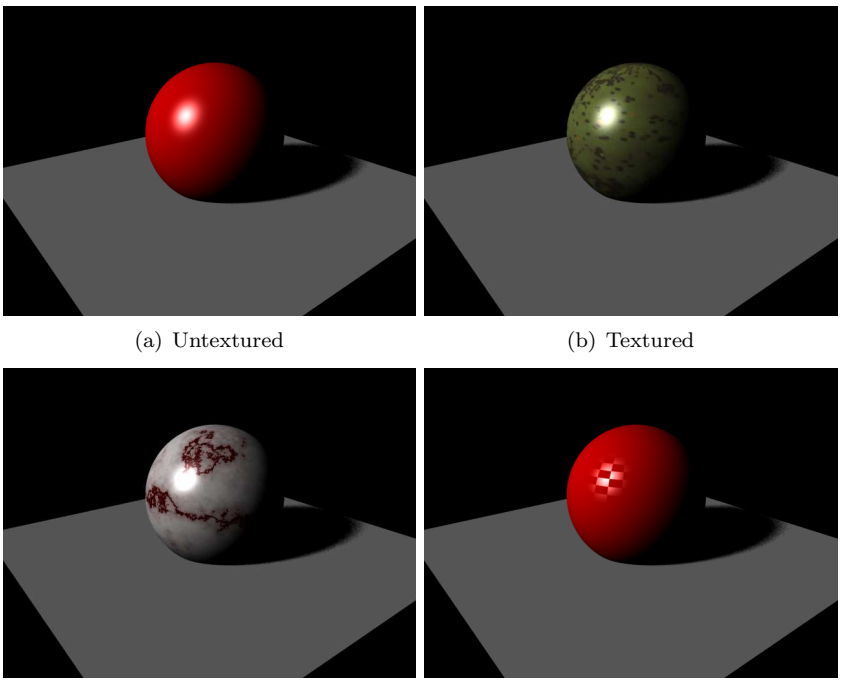
with longer diffuse light paths. It can also handle caustics but it is often done separately from the diffuse inter-reflection photon map since the requirements for caustic photon mapping is quite different.

1.5.3 Material Models

Looking at the rendering equation, the BRDF part of the equation describes the material properties. This is a function of the of viewing direction, incoming light direction and wavelength. There are many representations for this function available. Some are based on intuition such as the Phong model[16]. The Cook and Torrance model[17] is based on micro-facet theory. There are also techniques to measure BRDF's[18] of materials to accurately capture their properties. For a more complete description of different approaches see[19].
It may seem inappropriate not to use a measured material, but the non measured models are still much more popular. Among the reasons are performance, tweakability and the fact that they nicely separate different light contributions. A measured BRDF has no natural parameters to tweak to change the color or the roughness of the material.
The fact that the mathematical models tend to separate specular and diffuse components is not only convenient when tweaking the shader but also opens opportunities for using different approaches for specular and diffuse aspects of global illumination. It also makes it possible to divide light contributions into separate passes (see section 1.5.6) which gives artists the ability to post process calculations to make features such as reflections more or less visible without having to rerender the image. When using more physically accurate rendering methods and material models, it is generally difficult to tell what is a reflection, a specular highlight or diffuse light which makes the render pass separation much harder.

1.5.4 Surface Shaders

A surface material is often not identical for an entire object. This means that the BRDF depends not only on the object but where on the object it is computed. A flexible shading system gives the shader creator freedom in mapping parameters of the BRDF to different functions such as images, ramps and masks in order to make custom materials.
Texture mapping[20] is an example of how to do this. It basically stretches an image (texture) over the object to give it a new look (see figure 2(b)). It is possible to generalize this to map not only the color, but also other parameters of the shading model such as the shininess (see figure 2(d)).
To use this, a mapping from locations on the surface to the two dimensional texture needs to be specified. For polygonal models, this is usually done by storing 2d positions together with the objects vertices, which are interpolated over the polygons and then used when looking up points in textures. This is often referred to as UV mapping (U and V are X and Y respectively in texture space). It is also possible to use three dimensional textures which is convenient



(a) Untextured (b) Textured

(c) 3d texture (d) Textured Specular Color

Figure 2: Different ways of texturing a surface

for materials such as marble and wood (see figure 2(c)).
Texture maps are often in procedural form, meaning that instead of mapping an image over the surface a two dimensional function is mapped over the surface. Typical examples of such functions are noise and fractals, which would be wasteful to store in images since they are simple to compute. By generating them procedurally it is possible to make them more or less infinite in size without repetition.
Texturing is not only about mapping an image as a color for an object but a very general concept where data such as lighting information and pre-computed complex functions can be passed to the shader.

1.5.5 Instances of Shaders

A three dimensional scene often contains many materials based on the same shading model but with different parameters. This means that the same shader may be instanced many different times with different parameters. It is important to be able to share the main aspects of shaders but tweak settings of them independently. This comes naturally in shading languages of different types where functional decomposition lets the shader creator reuse blocks of shading code.

1.5.6 Render Passes

Rendering passes is about outputting renderings in separate components, such as diffuse light, specular light and reflections (see figure 3). In its simplest form it may be that only a certain light contribution should be considered, a more difficult example is to separate reflections so the first reflection bounce ends up in one pass, the second in another and so on.
To be able to handle this in an efficient way the shaders must be able to select components to be rendered. It must be possible to select components based on ray type and ray depths. For instance, if rendering a reflection pass, primary rays should only use the reflection component of the shading computations but reflection rays should be shaded as usual.

2 Previous Work

2.1 Non Customizable Shading

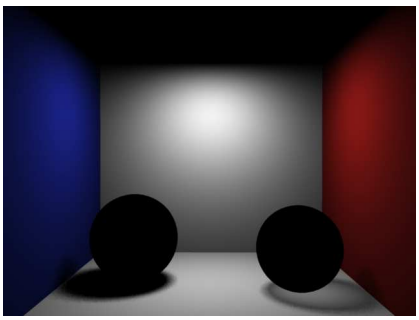
Old renderers often used a fixed shading model where only simple tweaks of colors, roughness parameters, etc were possible for different objects. There are numerous problems with this approach since different materials often requires different reflection models. It is also likely that the implemented shading model has to be a "worst case" model meaning that in situations where a simple shader would be sufficient it will still need to use the advanced model and ignore the disabled parts of it.

2.2 Shade Trees

Shade trees[21] is the first work on customizable shading. It creates expressions of simple operations organized as trees which are interpreted at runtime. Different shaders are used for lights and surfaces. It supports functional decomposition by reusing trees as subtrees in larger trees. All shaders supports texture reads and constants in addition to surface fragment data.

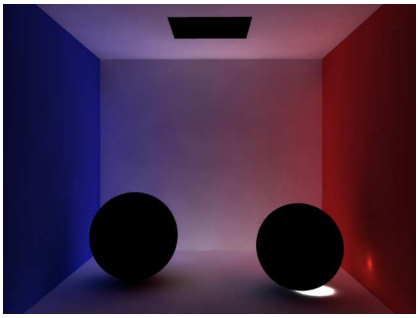
2.3 Renderman Shading language

Renderman Shading language[22] is a C-like language supporting many kinds of shaders. It has a type system appropriate for shaders and supports getting surface properties in different coordinate systems. It also has constructions for gathering and emitting light that looks similar to the rendering equation. This means it nicely separates light sources and materials. Non directed lights (ambient lights) are also supported.
Renderman shading language is owned by Pixar.



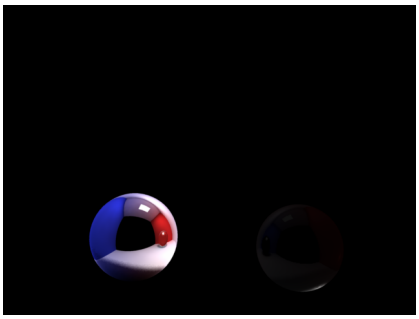
(a) Diffuse

(b) Specular

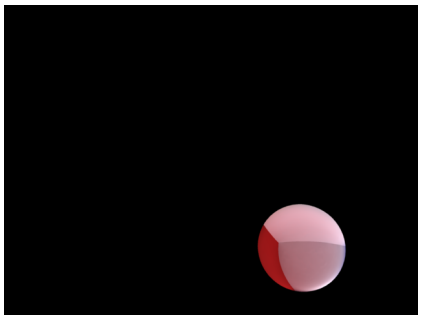


(c) GI

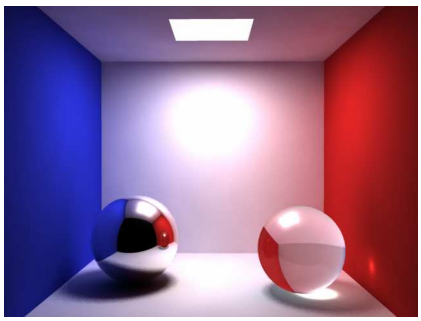
(d) Incandescense



(e) Reflection



(f) Refraction



(g) Complete Image

Figure 3: A Cornell box rendered in passes

2.4 OpenGL Shading Language and HLSL

These languages are quite similar. They are C-like languages. They differ from Renderman Shading Language by having programmability on both vertex level and fragment levels. They are also much lower level languages and features such as light source abstractions and coordinate systems has to be taken care of by the developer manually. These limitations are due to the importance of high performance in interactive applications and limited program size in graphics accelerators. The potential overhead introduced by using advanced shading languages may cause rendering times that's not acceptable for interactive frame rates.

2.5 Programming API's

Exporting a programming interface in a language such as C is another way to implement customizable shading in a renderer. For instance Mental Ray from Mental Images takes this approach for shaders of different kinds. From a compatibility point of view it is a bit of a problem since every shader has to be compiled for each platform it is supposed to be executed on. Another problem is that there are a lots of problems related to dynamically linking binary code, in particular if doing on multiple platforms. On the positive side no custom language has to be learned for people already used to C and the availability of high performance C compilers.

2.6 Building Block Shaders

Building block shaders[23] uses graphs of blocks, quite similar to shade trees, to express shaders. It has generally more complex nodes and also more advanced semantics for execution order. It supports custom nodes which can be freely incorporated in existing networks. By combining existing nodes it is possible to generate complex material models and procedural textures without any custom programming.
A system similar to this is used in the Maya modelling and rendering package.

3 Design Aspects

3.1 Comparison of Previous Approaches

3.1.1 Usability

From an artist point of view the Building Block Shaders approach are probably the most useful since it is possible to create complex shaders simply by combining standard components. The solutions relying on a programming language are always difficult to use for non programmers but allows users to create complex and specialized shaders with less overhead than building block shaders.
It is possible to use a language as a back end for more visual and intuitive tools.

Mayaman is an implementation of this for Maya which converts maya shading networks into renderman shading language.

3.1.2 Performance

From a performance point of view, the question is mainly about using pre compiled blocks like mental ray and maya or a custom language. Using pre compiled blocks made in C or C++ gives access to extremely powerful optimizing compilers for almost every platform. On the other hand, a language being interpreted or compiled by the renderer may do partial evaluations[24] of shaders such as dead code elimination, constant folding and function inlining by analyzing the entire program at compile time.
For instance, a precompiled shading solution interpreting a shader supporting mapping either a color or a texture to an input has to assume the worst and do a virtual function call to figure out the color since the compiler knows nothing about what kind of networks the user will construct. A system compiling shaders at runtime knows the entire shading network structure at the time it generates code and can handle the constant without a function call.
As a conclusion, the custom language is very powerful but it requires more effort and to really make use of the power since it needs a good compiler or interpreter to outperform C or C++ based shading solutions. On the other hand it supports optimizations a pre compiled solution can never do.
When targeting specialized hardware such as GPU's and stream processors using a custom language may be a good solution since C is not always available and using a custom language may "force" the coder to write code suitable for the platform. An example of such a language is Brook for GPU's[25].

3.1.3 Customizing Power

Custom shading languages tends to be focused and optimized for programming shaders and are generally not a very good base to implement customization that goes outside the boundaries of the language. A shading solution based on C/C++ may expose the user to more complex API's and let the user build persistent structures such as irradiance caches and point clouds. Handling this kind of functionalities can be difficult in more specific shading languages and implementations of this often works by rendering multiple passes and store data in textures that are used from the later passes. An example of this is the retrofitted subsurface scattering solution in Renderman used in "Finding Nemo"[26].

3.2 Properties of Shaders

Shader programs have properties making them stand out from normal programs. Since the shaders tends to be consistent over objects, it is often possible to batch

up shading calls for a shader and execute them in a single go. Another nice property is that there are generally no dependencies between shading calculations of different surface fragments. Finally, shaders tends to have simple control flow without too much conditionals.
These properties makes shaders excellent to compute on SIMD machines by batching up shading calls that uses the same control flow, that is calls using the same material, light and render pass configuration, and execute the computations in parallel. Another nice feature is that texture and other data access patterns tends to be coherent between surface fragments that are close to each other on the screen. Not surprisingly shading units of todays graphics hardware are massively parallel stream processors[27] since it is easy to exploit coherence in scanline based rendering pipelines.

3.2.1 Secondary Rays

The nice shader properties tends to fail if used with secondary rays in raytracing renderers. Each single ray can take a unique path generating more or less random shading calls which will make the coherence break down in a naive recursive raytracer.
If planning to execute shaders in a raytracer on graphics hardware, this is a big problem since it is important to feed it with large batches to get good throughput[28]. It is possible to handle this by rescheduling the contribution of secondary rays after the shader is done executing. This is done by Pharr et al.[29], even if the primary goal in this example is to achieve memory coherence in the geometry intersection code rather than the shading calls. A problem with this approach is that it is impossible to let the shader depend on the result of rays generated in the shader. For instance a shader generating 10 rays that will give output a different value depending on if more than 5 rays hits a certain surface would be impossible to write. This might sound like a silly limitation, but features such as adaptive sampling of blurry reflections needs to do this.
It is possible to circumvent this by scheduling rays differently, but designing a shading system not supporting low overhead secondary rays would probably be extremely inefficient in most current rendering pipelines.
An example of a rendering pipeline handling this efficiently by doing many shading tasks in parallel is the Kileauea renderer[30]. If one shader thread stalls while waiting for the result of a secondary ray it can start executing a new shader and continue the execution of the old one when the result for the rays are in. If making sure there are a lot of fragments to shade it is likely that there are always shading calculations to do and the shading will never be stalling. It is by no means a perfect solution since the execution state of the partly executed shaders needs to be stored in some way which might not be trivial. The approach also doesn't work for increasing the coherence in shading it merely hides the overhead in shooting rays with latency. Therefore it is not very useful to use together with a GPU that needs a lot of similar shading calls to speed up the calculations.
Future approaches to construct rendering pipelines should take this in consid-

eration and limit the shading interfaces in appropriate ways to force coherence in shading calculations and raytracing since it will open the up the possibility of using stream processors such as GPU's for shading.

3.3 External Requirements

Coding a complete shading system is a complex task and to keep the scope of the project within reasonable limits is therefore a must.

3.3.1 Maya Support

Since the test environment for the project is a renderer for Maya, the shading system should be implemented in a way that it can be used as a back end for the building block shader system used in the Maya renderer. It should also support handling the shaders and features of Maya in a natural way.
This is quite tricky since Maya requires a lot of features that prevents many optimizations and experiments such as dependency on secondary rays and querying light sources of quite specific data to get specular highlights right.

3.3.2 Supported Shader Types

Only materials and light sources are considered.

3.3.3 Global Illumination and Indirect Light

The shading system has to be able to handle light contributions from different kind of indirect light such as Final Gathering and Photon Maps.

3.3.4 Render Passes

The shading system shall be able to render different passes and make sure it is possible to add custom passes without affecting previously coded shaders.

3.3.5 Performance

The system must be able to render shaders with high performance. It must not introduce too much overhead when transferring data from the shader system to the outside world and should be able to access data from the renderer efficiently. It shall also have low overhead allowing it to shade single fragments without too much setup cost which is important in recursive raytracers.

3.3.6 Quality

Make sure the shaders are provided sufficient information to handle filtering properly to avoid unnecessary super sampling in areas with no geometrical or shadow edges.

3.3.7 Thread Safe

Computer graphics is computational expensive and it is crucial to make sure a shading system behaves correctly and scales on well on multi processor systems.

3.3.8 Easy API

Developing shaders is something that should be straight forward. A developer has to be able to construct new nodes in a simple way and focus on the actual shading programming rather than the glue code.

4 Implementation

4.1 Design

4.1.1 Programming Language

The language chosen for the project is C++. The reasons are that it has been used in similar situations before, the performance is good, there are a lot of good developer tools and it is easy to create hooks into the particular renderer we used in the project since it is also developed in C++.
An alternative would be to use an interpreting language such as Lua[31] or Python[32]. It would definitely be an interesting case study to use such languages, but since performance is an important requirement for the shading system and we find it hard to predict the outcome, C++ seems like a more reliable choice.
Implementing an own shading language with an optimizing compiler would probably be the solution with most potential, but writing a good compiler back end or byte code interpreter is a lot of work, in particular if it should support advanced partial evaluation features.

4.1.2 Building Block Shaders

When using C++ as programming language it is difficult to write custom code to extend the shading system without recompiling or providing a binary interface to plug in new modules. It is therefore important to give the user some kind of tool to generate original and complex shaders. Building Block Shaders with a "standard library" of material models, textures and utility nodes is an easy to use and powerful way to do this. By keeping the complexity of the shading nodes balanced it is possible to keep the overhead in virtual function calls to connected nodes low without limiting the flexibility of the system too much. It is also possible to provide a dynamic linkable node interface to make it possible for the user to provide own shaders to combine with the existing set.
The building block shader paradigm is not only nice from an artist point of view but also maps well to object oriented languages. Therefore, it is easy to model in a language such as C++. The building blocks implementation in the project

are inspired by Maya making each building block contain multiple inputs and outputs. Typically, a surface shading block not only contains a color output but also a transparency and glow output. The inputs consists of features such as material colors and specularity settings which are available to read from all outputs of the block.

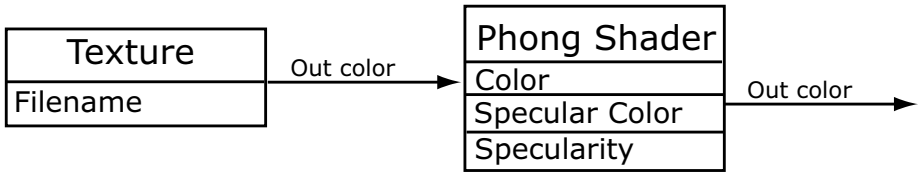
4.2 Shading Data

4.2.1 Explicit and Implicit Data

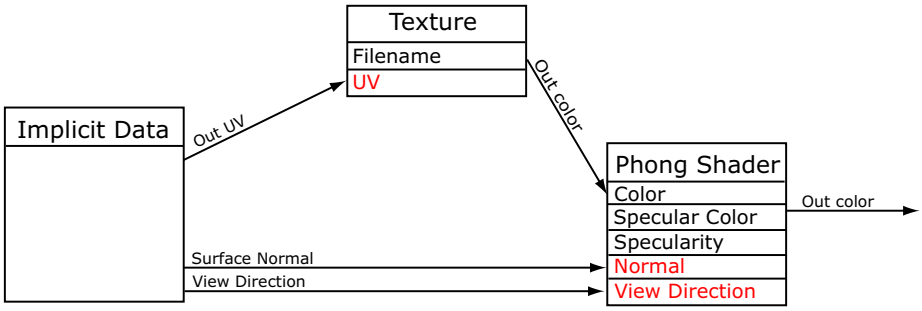
A diffuse material shader would need a diffuse color component, a surface normal, a fragment position and a set of light sources that contributes light to be able to compute the fragment color. If adding a texture map to a color component it would also require mapping coordinates.
The color component is a property of the specific shader that the user want to tweak explicitly, but the surface normal and fragment position are properties of the particular fragment and the scene it is being executed in. This data is modelled as inputs that is bound implicitly to the shader and available to all shaders (see figure 4).
This way of thinking separates the shading system from the renderer being used. By ensuring the implicit inputs correctly maps to the data the renderer outputs it's possible to deploy it in a new renderer. It is also easy to override the default behaviour of an implicit stream. For instance, normal mapping[33] can be implemented by overriding the default surface normal input with a custom node that looks up a normal from a texture.

4.2.2 Data Types

Inputs and outputs needs to be typed. When reading a normal input it is important that the data provided is actually a vector and not a scalar value. This means that every input and output of a shader is assigned a type and only inputs and outputs of the same or coercible types are possible to connect. In computer graphics, data such as colors and vectors are often used interchangeable. For instance, normal maps are often modelled as RGB textures. Therefore, it is important to be able to type cast colors to vectors, floats to integers etc.
It is important to handle compound types efficiently since much of the data passed through the shading blocks are colors, vectors and points. More complex compound types consisting of non primitive members, such as a spline control point consisting of a position, a time and a tangent, are also useful.
Another important type is the array type. For instance, a ramp of colors may consist of an arbitrary number of colors.
By looking at shaders, it turns out that complex compounds and arrays are generally used as a way of organizing input data rather than as output data. Introducing dynamic arrays as a data type to pass through the connections will introduce dynamic memory allocation and consideration of copy semantics.



(a) Without Implicit Data



(b) With Implicit Data

Figure 4: Implicit Data

Therefore, the shading system only supports arrays as inputs, however each element in the array may be connected to different outputs or constants.
It is also important to make sure it is possible to have a proper granularity in the calculations. For instance, a node that computes colors will in most cases be used to compute entire colors rather than single components, therefore it might be worth the overhead to generate all three RGB components in one computation even for cases where only a single component is actually read. In addition, most of the computations are shared for the different components of the color.
The model used to handle this is to make Vectors, Colors "primitive" types even if they are actually compounds. To avoid confusion primitive types, colors and geometric vectors are called simple types. Output types of nodes has to be simple, i.e. nodes returning arrays or complex compounds are not supported.
The static nature of shading network instances also makes it possible to assume the size of an array is constant for a shader instance. The simple types in the shading system are

- Boolean
- Integer
- Float
- Vec2
- Vec3

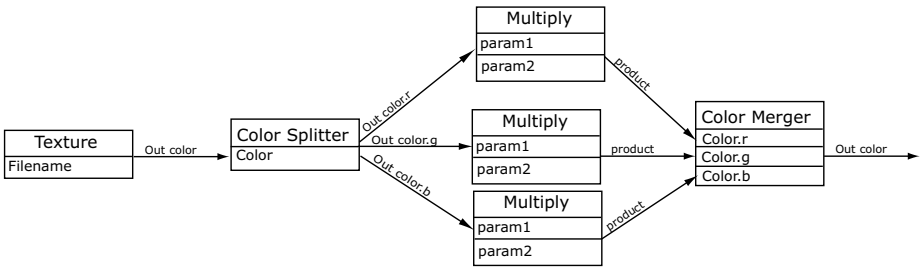


Figure 5: Splitting and merging in a network where R, G and B components needs special treatment

- String
- RGB Color
- 4x4 Matrix

4.2.3 Components of Simple Types

Since the simple types contains vectors and colors, it is important to be able to access and connect components of them independently. Accessing is done using nodes that takes a vector as input and outputs each of the components as floats. Connection is handled using nodes taking scalars as inputs and merges them to a vector. Figure 5 is an example of merging and splitting components. The split and merge nodes in our system are automatically generated in the conversion phase (see 4.5.1), rather than inserted manually in the shader editing tool.

4.2.4 Shader Types

The building block model makes it possible to get higher level type conflicts. For instance, what normal will a Phong shader node connected as shader for a light source get when it reads the normal input? It is of course an option to try to figure out a normal that "makes sense", but the real problem is that the type of a phong shaders is simply wrong for everything but surfaces. On the other hand, a texture might make sense to map on both light source and surfaces, it only requires a mapping coordinate to be evaluated.
It would probably be possible to have an advanced type system to decide whether a shader makes sense to use in a certain context, but the shading system is to be used mainly for execution of shaders and not shader network editing, any higher level type checks are left to the editor. The shading system should handle incompatible mappings gracefully by giving warnings but returning default values such as 0 for floats, black for colors when reading undefined streams.

4.2.5 Implicit Data

The implicit data available to the shading system is the bridge between the renderer and the shading system. For surface shaders information about the surface fragment such as normal, position, tangents and mapping coordinates are mapped to implicit streams. It is also important to supply data about the fragments area from the viewer to handle filtering. Since the renderer the shading system is developed for uses Ray Differentials[34], these are available as implicit streams.
Light sources has the same data available as the surface, with the exception that the mapping coordinates is replaced with a mapping more reasonable (spherical for point lights, perspective projection for spotlights etc). This gives it the ability to get information about the point it should contribute light to without giving up the ability to map the color in different directions.

4.2.6 Coordinate Systems

Different renderers and rendering architectures tends to output data in different coordinate systems and it is important for a shader to know whether the normal it just read is in camera space or world space to be able to compute shading values that makes sense. It is important that light contributions are in the same coordinate system as the point being shaded and avoid coordinate system changes for points and vectors as much as possible from a performance point of view.
A simple solution would be to provide the different implicit streams in all coordinate systems. The problem with this solution is that if some kind of modification, such as normal mapping for a normal, would be applied to one of the streams it wouldn't affect the ones in different coordinate system.
A better solution is to let the stream be the same for all coordinate systems but when referring to it at link time(see section 4.5.2), select what coordinate system it should be in. If the actual data produced by the renderer is in another coordinate system it will be transformed before it is returned. It is important that the shader developer is aware of what coordinate system the renderer usually produces its data since reading data in another coordinate system will trigger matrix vector products. When accessing the light sources a coordinate system must also be specified to avoid problems with the shader.

4.2.7 Upstream Writing

Upstream writing is a problematic but sometimes very useful feature of the shading system. A typical situation is bump mapping[20], that needs to sample a few distorted positions in the texture to find a gradient to perturb the normal. It would be possible to model this as a network with a set of different inputs from the same texture mapping node that each have an unique distortion on the UV-parameter connected to it. This is inconvenient and a better approach is to provide a safe way to distort an implicit stream.
If the implementation of the implicit streams are known to the shader it would

be possible to use some kind of "short cut" into the data that will be read when reading the implicit stream and modify it, but it is not a fully satisfactory solution since it would make the shading system less isolated from the renderer. Another way to handle this is to provide a stack for each implicit input on which it is possible to store distortions. When reading the implicit stream the distorted value is returned instead of the standard value. It is up to the node that that distorted the value to pop the distortion from the stack. Improper use of this stack is easy to detect after a shader execution by checking if the size is not zero. A somewhat unintuitive feature of this is that if bump mapping the normal of the base shader won't affect other shaders connected to the network depending on the normal. If this behaviour is desirable all shaders using the normal has to be explicitly normal mapped. An elegant solution to this is the one used in Building Block Shaders paper[23] where it is possible to set the whether a node should be executed before execution of shaders depending on it or when read.

4.3 Light Calculations

4.3.1 Light and Material Interaction

It is important to separate light models from material models to avoid treating each light source type differently for each material type. This is not as easy as it may seem, since many shading models are dependent on quite vague parameters of light sources such as a direction towards the light source to calculate specular intensity. This makes perfectly sense for a point light, but for an area light, there is no one direction towards the light source and coming up with one that's represent an integral of the specular light from the area light certainly require the light source to know something about how specular light is computed for the particular material it is contributing light to. An example of this is available in[35], but it is only working with the Phong model. For a complex material such a measured BRDF it would be more or less hopeless to come up with a reasonable integration for highlights.
A "clean" material-light interaction should probably only work in terms of directed or non directed light like Renderman Shading Language does. It is a nice that solution mimics rendering equation and all surface shaders only needs to implement how it responds to this kind of lighting. To handle light sources with extension it is possible to monte carlo sample them as a set of directed light contributions. Since maya support is very important for the shading system and to get identical result with maya and avoid monte carlo noise in area light highlights we had to introduce functionality to query whether a light contribution is an area light. This somehow makes the solution less elegant.
The light contribution solution is in itself a general solution, supporting features such as monte carlo sampling of area lights and importance sampling of HDRI-lights[36], by letting the underlying implementation of the light contribution iterator generate multiple light contributions from single light sources.

4.3.2 Global Illumination and Ambient Occlusion

The light from photon maps, final gather and other indirect light sources are easy to model as an ambient light source which does the actual lookup in the photon map or irradiance cache rather than a traditional light computation.

4.4 Render Passes

Rendering passes is done using two systems, one that makes sure it's possible to shade using only certain light components, such as diffuse or specular. The other selects a set of components to use depending on other properties such as ray type or ray depth.

4.4.1 Shading Components

To render the different light components in separate passes, each one must be decided to belong to a pass and it should only be considered if it's in the active pass. Typically, when rendering passes one renders more than one pass in a go, and a pass may contain more than one component. To support this shader components are internally represented as a set which is implemented as a bitmask where each bit represents a shading component. It may seem like a limitation to handle only 32 unique components in one rendering, but it seems reasonable that bitmasks even if larger than a word will be more efficient than a binary tree for most reasonable number of components.
To be able to create passes on the fly component is specified as a text name. In the linking phase (see 4.5.2), all text names are converted to bits in the pass bitmask. Looking if a certain pass is to be rendered or not can then be evaluated in constant time. When it comes to actually rendering the passes, we have looked into three different approaches:

- Render each pass as a separate image not sharing any computations.
  This approach is nice since it needs only minor changes to the rendering pipeline, however, a lot of work is duplicated. All pixels are rendered in each pass, meaning the primary rays and light loop has to be executed for each pass.

- Shoot the primary rays and shade each fragment with all active sets of pass configurations.
  This solution saves the primary rays, it still needs one light loop per pass and fragment. It needs the renderer to have a separate framebuffer for each pass being rendered.

- Shoot the primary rays, and make one shading call that output all components separated. The components are then composed to the different passes outside the shader.
  This solution is the best when it comes to computation time since it does

the shading only once per fragment. The problem is that it would introduce a complex data type to return data from the shader that doesn't fit into the rest of the shading system architecture.

Render Passes is another feature that would benefit from partial evaluation. It would be possible to specialize the shader for a certain set of components. If for instance only a pass containing the specular components should be rendered, it would be possible to specialize it and avoid even touching conditionals containing other parts.

4.4.2 Ray Matching

The second part of the pass system is a ray matching system selecting what components should be enabled for certain types of rays. To support this, the traced rays needs to carry information about reflection and refraction depths. This is generally no problems since most raytracers do that anyway to be able to terminate rays going beyond maximum ray depths to avoid infinite recursions. Since it is difficult to figure out exactly what kind of overrides users wants to do, this part is programmable too so it is possible to generate passes that suits each user.
Currently our only implementations are hardwired solutions to get the reflections in and refractions in separate passes but it should be possible to create a user interface customizable version of this that allows the user to do matching ray type, ray depth etc.

4.5 Initialization of Shaders

The shader initialization is done in several phases to make sure it is possible to control and customize the behaviour of the system.

4.5.1 Conversion

The shaders used in the system are described as Maya Shading Networks. By implementing our own shader blocks that emulates the ones in maya, we can instantiate our own versions of the shading networks. This is done by traversing the networks, create blocks and set up references between outputs and inputs. These references are only text string references to make sure it is not coupled at all with sources for implicit streams and to make it easy to apply transforms to the networks if some node needs special treatment after conversion. Even if we only support Maya at the moment, it should be possible to use any source of shader networks here.

4.5.2 Linking

The linking phase converts the text references between the input and output nodes into pointer references, to make sure that no string matching needs to take place when executing the shaders.

The linking routine recursively visits all blocks in the network and looks up and type checks all references needed, implicit references are provided by the renderer so the network can access the data of the fragment it shades when it executes.
Custom Render passes are converted from text strings into bitmasks to make sure querying whether a component should be rendered or not is possible to do quickly.

4.5.3 Post Linking

The post linking phase is done after all links are created. It is available to give the shader writer the possibility to do finishing touches. After the post link the shader is ready to execute.
A typical thing done in this phase is constant propagation. It is possible to query if a shading node is constant. It is up to the node to implement how this is determined, but if the shader only depends on constant nodes it may safely assume that it is constant itself. There is currently no automation for this, so it has to be done manually. Data such as inputs to color ramps and transformation matrices are often static and it is therefore a good idea to cache these values in the post link phase to make sure they are not recomputed for each fragment.

4.6 Execution of Shaders

4.6.1 The Shader Context

The shader context is the source of data for the shaders, it contains information about the fragment, what components being rendered, the light sources to consider etc. It is the only parameter passed to a shader when executing it. It is important to notice that this is the only state that's allowed for a shading session, for instance caching data in shading networks or writing to global data is prohibited because it is not thread safe.
Implicit streams are actually links into the shader contexts that accesses fragment information. Also all secondary rays are traced through calling methods on the shader context.
If making the shader context abstract it should be the only way data is passed from the renderer to the shading system. To deploy the shading system in a new renderer this class should be reimplemented. From a performance point of view this may not be optimal since a lot of calls are made to access things such as fragment normal, UV's and tangents. It is not abstract in our implementation to allow these accesses to be inlined.

4.6.2 The Override Manager

The shader override manager sets up render passes by overriding the choice of components to be rendered and what shader to use. It can basically override

everything, but typically it just passes the shader call through or disables components depending on the ray type. Another typical use is to discard the shader for the surface and use for instance a surface normal shader or an ambient occlusion shader to render a normal or ambient occlusion pass.

4.6.3 Execution

The simplest approach to shading network execution is probably the depth first execution order in which every node is executed when it is read from.
If shading a group of surface fragments another approach would be to execute each node for all fragments in a dependency order. This would probably give better cache performance since both the code and the data is more local in the node than the entire network. On the other hand it would be difficult to handle conditionals efficiently in this kind of setup since it is impossible to know whether an input is actually used or not at the time of execution. Another interesting thing about this is that it is probably easier and faster to cache values if reading the same input multiple times in the same shader if using this approach, it would conflict with usage of upstream writing though (see 4.2.7).
In the system built, depth first execution is used, mainly because it is simpler and more predictable in performance for conditional nodes. However it would be interesting to investigate the possible performance gains to cache locality if using breadth first execution.

4.6.4 Threading Consideration

Making shaders thread safe is very much about making sure that all of the execution context are stored in data local to the thread such as the stack or in thread local storage.
Another problem is that the heap is shared between threads. This means that it has to be locked by a mutex, making dynamically allocated memory expensive if used in the internals of the shading system. To avoid this as much data as possible has to be stored on the stack or allocated using thread local memory pools to avoid threads fighting about the memory allocation mutex.
When executing using multiple threads, caching of data is more complex. If for instance two identical texture reads are connected to the same shader node it would be convenient to check if the last access was identical with the current and then return the stored value instead of doing the computation all over. The problem is that there might be race conditions if two threads use the same shader. However it is possible to make thread specific caches which uses the shader state as key to lookup previously calculated values.
Another performance issue we ran into when implementing the shading system was variables shared between threads. Even if using atomic operations when writing to it, there were severe performance penalties using it. It's likely that the reason was that the processors had to reload the cache line the variable belongs to every time the other processor had changed it since when it was last read.

5 Results and Future Work

5.1 Performance

Comparing shading systems in different renderers is not perfectly straightforward because it is a component in a larger pipeline. The actual interesting thing to measure is the overhead in propagating data between shading nodes and the cost of getting data from the renderer such as fragment normals and points. The other things such as performance of the code executed in the actual shader, the speed of secondary rays and speed of texture lookups are not directly dependent on the shading system being used.
The tests were performed on a Dual Opteron 1.6GHz with 2GB of ram running Windows XP Professional. Maya 7 was used for the test.

5.1.1 Test 1, Multiplications

A huge network of multiplication nodes being used as color on a very simple object. Since a multiplication node is very simple and likely to be implemented in a more or less optimal way in all renderers a network of multiplications connected as color to a simple piece of geometry should give some idea of what kind of overhead there are in transferring data between nodes. To make sure no constant propagation is used the data to be multiplied is the fragment position in world and camera space (to make sure that renderers producing data in different coordinate systems should not lose performance in transforming data to a non native system). This test tests both data transfer between nodes and data transfer from the renderer to the shading system.
The network is built as a tree in n levels where each input are connected to a multiplication output except for the leafs that are sampler info nodes providing world and camera space positions. The number of multiplication blocks are $2^n - 1$ and the number of sampler info blocks are 2^n where n is the level. A network of level 3 is shown in Figure 6. The object used is a box, that's zoomed in to a level where only one face is visible, the test resolution is 640x480, 1 sample per pixel is used for all renderers. The running times are in seconds.
The results for world space and camera space are found in figure 7 and 8 respectively.
The most remarkable in the results are the poor performance of Mental Ray. Our shading system performs slightly worse than Maya. Note that the doubled rendering time in each level is predictable since the number of nodes doubles in each level. The slightly higher performance in Turtle World Space than Camera Space is likely to be related to that Turtle tends to produce sample data in World Space.
The difference between the renderers seems to be quite large, but for real life situations networks larger than 20 nodes, which lies somewhere between level 3 and 4, are not very common. Also more complex nodes hides the overhead in data passing making the actual code executed in the nodes more influential.

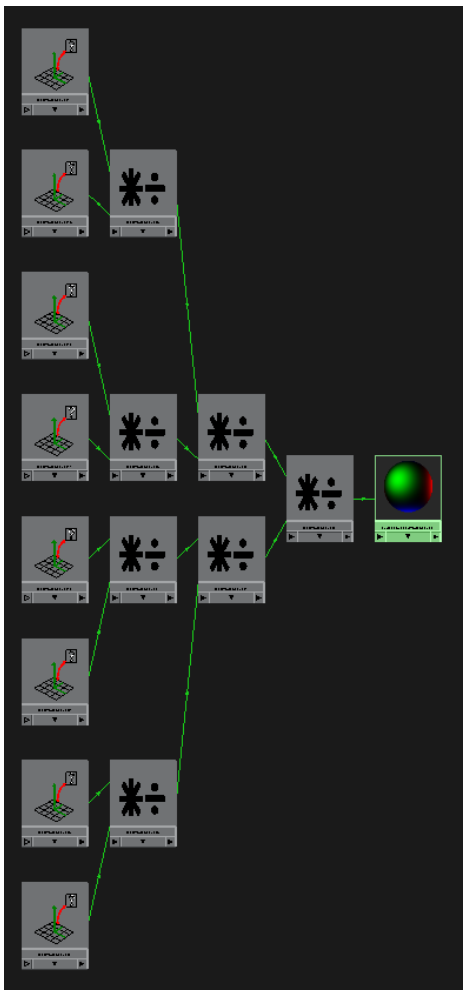


Figure 6: Figure of a multiplication network of level 3. The blocks to the left are the leafs supplying points, the blocks to the right are multiplication blocks

Level	Maya	Mental Ray	Turtle (Dual Proc)	Turtle (Single Proc)
1	1	3	1	2
2	1	4	1	2
3	1	5	1	2
4	2	8	2	2
5	2	15	2	3
6	3	26	3	4
7	5	54	4	8
8	10	105	9	17
9	18	215	19	34
10	36	500	44	73

Figure 7: Running times in seconds for multiplication test in World Space

Level	Maya	Mental Ray	Turtle (Dual Proc)	Turtle (Single Proc)
1	1	3	1	2
2	1	4	1	2
3	1	5	2	2
4	2	8	2	3
5	2	15	2	3
6	3	26	3	5
7	5	52	5	9
8	9	105	11	18
9	18	215	21	43
10	35	525	51	81

Figure 8: Running times in seconds for multiplication test in Camera Space

Level	S. Preproc.	S. Rendering	S. Total	D. Preproc.	D. Rendering	D. Total
7	0.38	6.65	7.03	0.39	3.54	3.93
8	0.89	15.53	16.42	0.86	8.17	9.03
9	2.36	30.83	33.20	2.34	16.38	18.72
10	10.21	61.62	71.84	10.14	33.24	43.38

Figure 9: Running times in seconds for different phases. S denotes single processor D denotes dual

Level	Dual Render / Single Render	Dual Total / Single Total
7	1.90	1.82
8	1.90	1.82
9	1.88	1.77
10	1.85	1.66

Figure 10: Shows how much faster a dual processor rendering are compared to a single processor

5.1.2 Test 2 Multiprocessor Scaling

Looking at the first test it seems like the shading system scales quite poorly when the shading networks gets larger, to take a more detailed look at this, the rendering times for World Space renderings are broken down to preprocessing time and rendering times. Some parts of the time such as raytracing data structure building and shading system loading is not done in parallel. This data has somewhat more precision in the timing than the previous test since we had access to more advanced timing features in Turtle than the other renderers.
The results are found in Figure 9. As we can see the rendering scales good, but the pre processing is more or less constant. It's interesting that the preprocessing time of 9 and 10 level shading networks seems to explode. This is not much to bother about in real life since that size of shading networks are not realistic for anything else than stress testing. This is likely to be the the cause of that Maya outperforms turtle more on level 10 than previous levels.
Figure 10 shows how much faster a dual processor rendering are compared to a single processor and when disregarding preprocessing time, the results looks much better.
This test would be much more interesting if we had access to machines with more processors than 2.

5.2 Possible Extensions

5.2.1 Performance Improvements

Implementing a dependency tracking system where it is possible to for the shader to communicate what implicit streams it actually needs would probably be useful since the implicit streams could be more restrictive in what data it should

generate. For instance it could skip generation of a tangents for the surface fragment unless the shader accesses it. A more advanced caching system where implicit streams are lazily evaluated and results are cached could handle this transparently. However it should be used with care on really complex shaders since it may give severe performance penalties if different data that are accessed multiple times are fighting about the same cache slots.
Implementing a more advanced constant propagation system would be useful. The current querying system requires the shader writer to manually handle this. It is possible to automatically avoid virtual shading calls for constants by storing the value with the reference to the node, and if the node is constant it reads this value instead of the evaluating the node.
Another interesting thing to explore is the difference in performance when passing data by letting the shader write to a variable rather than returning the result as a value. In particular large data such as matrices should gain performance from this unless the compiler can do some kind of return value optimization.

5.2.2 API

Constructing a solid cross platform API for C++ is not trivial. First of all the actual functionality to expose is a difficult question. It would probably be a good idea to start looking at other solutions such as Mental Ray. The second problem is to make sure data is passed transparently from dynamically linked code to the renderer. Since calling conventions and naming conventions for C++ object code is different for different compilers and even compiler versions and it is difficult to get classes work properly. It is likely that a solution using a C-interface or a CORBA-solution is the most practically useful. Another problem to consider is the fact that dynamic linking is done differently on different platforms. To sum it up, it would be easy do get fed up by the platform specific parts of this problem.
An interesting way of handling the platform specific issues would be to use a custom language such as Lua for the API. It would of course introduce overhead but the scripts would be platform independent.

5.2.3 Graphics Hardware Rendering

Graphics hardware would be a very interesting to use for shaders in offline rendering. It is currently being used in Gelato[37]. Gelato is a scanline renderer making it easier to exploit coherence than in a raytracer. It is developed by NVIDIA which means that the people creating it may take advantage of knowing the internals of their graphics cards. If undertaking a similar project there are a lot of things to consider such as resource virtualisation, mapping of functionality to graphics hardware, batching data etc. A lot of work has been done on this such as SH[38][39] and brook for GPU's[25] which both would be good starting points for this kind of work.
The interaction between the shaders and external data such as photon maps, irradiance caches and point clouds may be a tough challenge since these struc-

tures may be difficult to look up data in from the GPU. It is likely that lookups in this kind of structures has to be done before executing the GPU shader. A starting point could be an approach similar to the SPOT shaders in the Kilauea renderer where the shaders are broken down into parts depending and not depending on external data such as secondary rays. The parts not depending on external data may be executed on the GPU and the rest on the CPU.

5.2.4 Coherence Analysis and Reorganization of Rendering

The key to fully make use of stream processing hardware such as GPU's in ray tracing contexts is coherence in shading calls. It is likely that there is high coherence in the primary rays, but for reflections and refraction, scenes with many different shaders may actually get choked by the overhead in transferring small batches of surface fragments to the cards. In particular sampling of hemispheres which is important in final gathering generates a lot of incoherent rays spawning random shading calls.
It is an interesting challenge to create a rendering pipeline that hides the overhead in transferring data to graphics cards and handles secondary rays efficiently. A possible key element in this kind of pipeline may be to have shaders in a high level representation that are possible to compile for many different processors. This would make it possible to use the high overhead, high performance back end when having large batches and the low overhead but lower performing back end for small batches.

5.3 Results

We've implemented a shading system capable of executing material and light shaders in a renderer that supports most of the features of Maya and a lot of other features such as global illumination. The performance in just propagating data is comparable to Maya. It has been successfully used for creating images and baking lighting and surface features on models in commercial projects. The approach has afterwards been extended to support "shading" of photon bounce features of surfaces.

References

[1] James T. Kajiya. The rendering equation. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150, New York, NY, USA, 1986. ACM Press.

[2] Philip Dutré, Philippe Bekaert, and Kavita Bala. *Advanced Global Illumination*, chapter 2, The Physics of Light Transport, pages 15–46. A K Peters, 2003.

[3] Tomas Möller and Eric Haines. *Real-Time Rendering 2nd edition*, chapter 6, Advanced Lighting and Shading". A K Peters, 2002.

[4] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 511–518, New York, NY, USA, 2001. ACM Press.

[5] Ingo Wald. Realtime Ray Tracing and Interactive Global Illumination. *PhD thesis, Saarland University*, 2004.

[6] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.

[7] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 137–145, New York, NY, USA, 1984. ACM Press.

[8] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 95–102, New York, NY, USA, 1987. ACM Press.

[9] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Commun. ACM*, 19(10):542–547, 1976.

[10] Cohen, Greenberg, Immel, and Brock. An efficient radiosity approach for realistic image synthesis. In *IEEE Computer Graphics and Applications 6*, pages 23–35, 1986.

[11] Henrik Wann Jensen. Global Illumination Using Photon Maps. In *Rendering Techniques '96 (Proceedings of the Seventh Eurographics Workshop on Rendering)*, pages 21–30, New York, NY, 1996. Springer-Verlag/Wien.

[12] Eric P. Lafortune and Yves D. Willems. Bi-directional Path Tracing. In H. P. Santo, editor, *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*, pages 145–153, Alvor, Portugal, 1993.

[13] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 65–76, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[14] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A ray tracing solution for diffuse interreflection. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 85–92, New York, NY, USA, 1988. ACM Press.

[15] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*, chapter 4, The Photon-Mapping Concept. A K Peters, 2001.

[16] Bui-T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[17] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. In *SIGGRAPH '81: Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, pages 307–316, New York, NY, USA, 1981. ACM Press.

[18] Sing Choong Foo. A gonioreflectometer for measuring the bidirectional reflectance of material for use in illumination computation.

[19] Michael Ashikmin, Simon Premoe, and Peter Shirley. A microfacet-based brdf generator. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 65–74, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[20] Tomas Möller and Eric Haines. *Real-Time Rendering 2nd edition*, chapter 5, Texturing". A K Peters, 2002.

[21] Robert L. Cook. Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231, New York, NY, USA, 1984. ACM Press.

[22] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 289–298, New York, NY, USA, 1990. ACM Press.

[23] Gregory D. Abram and Turner Whitted. Building block shaders. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 283–288, New York, NY, USA, 1990. ACM Press.

[24] Brian Guenter, Todd B. Knoblock, and Erik Ruf. Specializing shaders. *Computer Graphics*, 29(Annual Conference Series):343–350, 1995.

[25] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Mike, and H. Pat. Brook for gpus: Stream computing on graphics hardware, 2004.

[26] Eric Tomson. Human skin for finding nemo. In *RenderMan, Theory and Practice, Siggraph 2003 Course Notes*, pages 89–100. 2003.

[27] Alan Chalmers, Timothy David, and Eric Reinhard. *Practical Parallel Rendering*, chapter 4, Overview of Parallel Graphics Card and 8. A K Peters, 2002.

[28] Pedro Trancoso and Maria Charalambous. Exploring graphics processor performance for general purpose applications. *Proceedings of the Euromicro Symposium on Digital System Design, Architectures, Methods and Tools (DSD 2005)*, 2005.

[29] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 101–108, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[30] Alan Chalmers, Timothy David, and Eric Reinhard. *Practical Parallel Rendering*, chapter 8, The "Kilauea" Massively Parallel Ray Tracer. A K Peters, 2002.

[31] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua — an extensible extension language. *Software Practice and Experience*, 26(6):635–652, 1996.

[32] The python programming language. Website. http://www.python.org.

[33] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving simplification. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 115–122, New York, NY, USA, 1998. ACM Press.

[34] Homan Igehy. Tracing ray differentials. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 179–186, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[35] Tanaka, Toshimitsu, and Tokiichiro Takahashi. Shading with area light sources. In *Proc. of Eurographics '91*, 1991.

[36] Sameer Agarwal, Ravi Ramamoorthi, Serge Belongie, and Henrik Wann Jensen. Structured importance sampling of environment maps. *ACM Trans. Graph.*, 22(3):605–612, 2003.

[37] Gelato. Website. http://film.nvidia.com/page/gelato.html.

[38] M. McCool, Z. Qin, and T. Popa. Shader metaprogramming, 2002.

[39] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.