Workflow Improvements for Real-Time Shader Development

Peter Dahl Ejby Jensen

Kongens Lyngby 2006
Master Thesis IMM-2006

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

This thesis will discuss the design and implementation of a shader graph editor. The editor makes the daunting task of programming shaders accessible for non programmers, as no programming or specific knowledge of graphics hardware is required. Graphics programming complexities such as different types or conversion between mathematical spaces, such as world and object space, is hidden from the user and handled automatically by the system. The work presented here also covers integrating the editor with a game engine, which includes supporting effects such as shadows and different light types in the generated shaders. The editor supports the creation of both vertex and fragment shaders, and discusses optimization issues of the generated shaders.

Resumé

I denne afhandling vil vi diskutere design og implementering af en shader graf editor. Editoren gør den krævende opgave at programmere shadere tilgængelig for ikke programmører, da erfaring med programering eller specifik kendskab til computer grafik hardware ikke er nødvendigt. Kompleksiteterne fra grafikprogrammering såsom forskellige typer samt transformation fra et matematisk rum til et andet, som f.eks. objekt- til verdenskoordinater, er skjult for brugeren og bliver håndteret automatisk af systemet. Projektet gennemgår også integreringen af editoren med en *game engine*, hvilket giver understøttelse af effekter som f.eks. skygger og forksellige lyskilde typer i de generede shadere. Editoren understøtter ydermere både vertex og fragment shadere, og diskutere optimerings overvejelser for de generede shadere.

Preface

This thesis was prepared at image analysis and computer graphics group, which is a part of the Informatics and Mathematical Modeling department at the Technical University of Denmark. The thesis is in partial fulfillment of the requirements for acquiring the M.Sc. degree in engineering. It is nominated to 40 ECTS points, and the project period were 9 months between the first of october to the 30 of june. The project was carried out in cooperation with the danish software company Over The Edge Entertainment, and we frequently discussed the content of the project with them. Our implementation were furthermore integrated with their game development software named Unity. Unity is closed source which means that the source code created in this project is not public, and can not be redistributed. The rest of this report can be redistributed provided that it is not modified without explicit permission from the author.

Lyngby, June 2006

Peter Dahl Ejby Jensen
Student Number: s001733

Acknowledgements

I would like to thank my two counsellors Niels Jørgen Christensen and Bent Dalgaard Larsen for their ever helpful comments during my work. I would also like to thank the whole computer graphics group at DTU, for showing a great interest in this project. A special thank you goes to Over The Edge Entertainment where this project were developed. The whole team at OTEE were always there to help with issues using their game engine, and good tips, ideas and discussions about the project. The high quality artwork and textures in this thesis were provided by Danni Tell and Bill Vinton, to whom I am also very grateful. Finally I would like to thank Martin Valvik and Bo Esbech for their insightful comments during the final steps of the report writing, along with my whole family who has been very understanding during stressful periods.

Contents

Nomenclature

Expression	Definition
Bump map	A texture map that stores the normal of a surface. These normals are often perturbed slightly in comparison with the objects normals. When these normals are used in the lighting calculations, the result is a more rough looking surface.
Connector Slot	The object used to create connections between nodes from and to. Called input and output in figure 5.1. In terms of the generated shader code, a slot should be thought of as an variable of a particular type, which is defined in a particular mathematical space.
Normal map	A normal map is basically the same as a bump map, and we use both of these terms interchangeably in this thesis. The term normal map is often used in the industry, when the goal is to create more precise pr. fragment lighting, and not so much to create a rough looking surface.
Offline Rendering	Any rendering that is not fast enough to be real-time rendering.
Real-Time Rendering	We define real-time rendering to be when the amount of pictures generated each second exceed 25 (fps > 25).
Shader Graph	A collection of connected nodes that has individual functionality. The collection yields a shader when the nodes are combined in the direction of the graph.
Shader	A Shader is a piece of code that handles the coloring of an object, possibly using textures and light calculations to generate the final appearance. Shaders can be executed on the graphics hardware or using the CPU (software rendering).
Subgraph	A subset of nodes defined by all the nodes that a given node connects to, both directly and indirectly through other nodes. The node used as a starting point is included in the subgraph. See figure 5.2 for an illustration.

Swizzeling	Swizzeling is used for selecting a subset of the components in a vector. An example could be selecting only the red component of a color by writing color.r.
Vector Notation	When writing equations vectors will be written in bold letters

CHAPTER 1

Introduction

1.1 Introduction

This thesis will discuss the implementation of a shader graph editor. The purpose of this tool is to make shader programming more accessible, for people with little or no shader programming experience. Experienced programmers could also benefit from this tool, as it could quicken the development time by providing a higher level of abstraction, easier debugging plus other workflow improving features. We will discuss both design and implementation issues throughout the thesis, which will end with a presentation of the final product and a discussion of the results obtained. The shader graph editor will be used to create shaders, which is a special kind of program that handles the shading of an object. Recently more and more applications are beginning to use the so called effect files as a format of the shader program, which is also what we do in this thesis, with the major difference that this file is presented in a graphical manner instead. In this chapter we will discuss effect files, and give a brief description of the format we have chosen to use in this thesis. In chapter 2 we discuss previous work in both real-time and offline rendering. Relevant background theory on materials, graphs and more is discussed in chapter 3. After discussing previous work and background theory, we will present the requirement specification for our editor in Chapter 4. In chapter 5 we discuss how to design a system that fulfills these requirements, and chapter 6 discuss the implementation of this design. Chapter

7 is a case study of how a particular shader can be implemented, using other applications, which we use as a base for comparison with our own product in chapter 9. Between those two chapters we present our own results in chapter 8, and in chapter 10 we conclude on the thesis and project in general.

1.1.1 Target Audience

We will assume that the reader of this thesis, is familiar with the capabilities of the fixed function pipeline in graphics cards, and knows how to render objects with it using OpenGL. This type of reader could be an engineer or similar, which has some previous experience with graphics programming. We also expect the reader to have knowledge about creating computer graphics for games. In this thesis the product developed will be targeted towards game developers, and this means we will be using game development terms throughout this thesis. As the subject of this thesis covers many areas, we are not able to give thorough explanations of all the relevant background theory, so often there will be given references instead, which can be consulted for further information. These references are especially about GPU programming and compiler design, which we present in chapter 3. To sum up, the ideal reader of this thesis should have knowledge about game programming, real-time graphics programming including shader programming, and possibly some knowledge about graph theory and compiler design.

1.2 Introduction to Shaders and Effect Files

In this section we will discuss the evolution of shaders, and account for the underlying technology used to produce them. This will include a description of effect files, which is the basic format for shaders that we use extensively throughout this thesis. Future chapters will show how this format can be presented in an abstracted way as a graphical editor for shaders.

1.2.1 Shading Languages

Only a few years ago, rendering objects using graphics hardware were done by applying a set of fixed transformation and shading formulas in the rendering pipeline. The fixed function pipeline only had support for standard Phong like shading models, where the programmer could set material constants such as specular component and colors. This made realistic rendering of certain materials such as glass or metals difficult, as these materials can have special attributes such as anisotropic highlights, and reflection/refraction of light and environments. Further more the shading were calculated on a pr. vertex basis, which does not look as realistic as current pr. fragment based shading. This all changed with the introduction of the programmable graphics pipeline though. The previously fixed vertex and fragment processors were substituted with programmable ones, but the rasterizer that connects these remained the same. See figure 1.1.

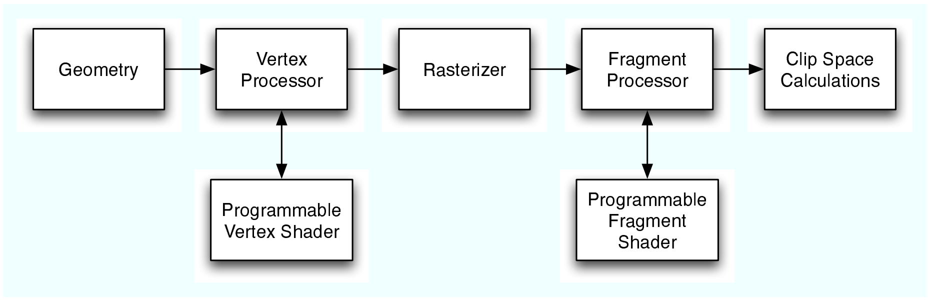


Figure 1.1: *Recent upgrade of the graphics pipeline.*

The first generations of programmable graphics hardware, used low level assembly languages for shader creation. It were difficult to create these programs as debugging were impossible, and it required a great knowledge of the underlying hardware. This has changed with the last few generations of graphics hardware though, where high-level programming languages has emerged. One such language is Cg (C for Graphics), which were created by Nvidia [16]. Other languages includes HLSL [12], GLSL [1] and SH [23]. These languages are very similar to the well known C programming language, with added types for storing vectors, matrices and such. They also have build in functions for working with these types, that maps directly to the underlying hardware for fast algebra computations. Programming shaders with these languages has become very popular the last few years, as it has been relatively easy for programmers to harvest the power of the new hardware. The languages still lack support for advanced data structures. Debugging the high level languages is also a problem. It is possible to perform some debugging through software simulators, but those might not indicate driver bugs, or other very hardware near problems. These languages also does not have any way of altering the current rendering state in OpenGL (or DirectX if that is used). Programs created in these languages can be used to control the rendering within a single pass, but they depend on the application that uses them to set up the state variables such as blending, render target and so on. This leads us to the introduction of a higher abstraction level, namely effect files.

1.2.2 Effect Files

An effect file is a script that handles all the rendering of the objects it manages. Currently there exists two dominant languages for writing effect files; Nvidia's CgFX [13] and Microsoft's Effect Files (FX) [11]. Effect files are often used for implementing shaders, as they have control to setup and handle rendering passes, while they can also incorporate a shading language for customizing the rendering pass. Often the shader is implemented by a single effect file. Most effect file languages are very similar in their structure, and if a user is familiar with one, it should not be a problem to understand effects written in the others. We will now give a brief description to the structure of the effect file used in this thesis. In Appendix B.1 we show an implementation of a shader that does specular lighting, with the specular component modulated by a texture. The shader uses the effect file format called Shaderlab, which illustrated below in figure 1.2. For now it should be thought of as a pseudo format, but it is actually an real effect file format that is a part of the game engine we have chosen to integrate our work with.

The properties scope is used for defining variables that can be modified externally. These could be colors, textures or similar variables used for the rendering. When defined in the Properties scope they will automatically be setup by the effect file language. The Category defines a scope of states common to everything inside it. It is possible to overwrite these state settings in subsequent subshaders and pass scopes though. The subshader scope is used to set up the rendering passes for the effect. Only one subshader will be executed in an effect file. Shader Lab examines the subshaders in a top-down approach, and uses the first one it encounters that will be able to execute on the underlying hardware. If no subshader in a particular effect file is supported, the shader will fall back to another shader, as specified with the Fallback command, that is set as the last thing in the effect file. Within a subshader it is possible to alter the OpenGL rendering state, and have any number of rendering passes. These passes are set up using the Pass scope. Each Pass results in an actual rendering pass being performed by the engine. In a Pass scope it is possible to modify the OpenGL Rendering state, plus specify vertex and/or fragment programs. If such programs are specified they will substitute the fixed function pipeline that OpenGL otherwise uses. If no programs are specified, most effect file formats has a material tag that can be used to set material constants used by OpenGL.

```
Shader "Shader Name" {
  Properties {
  }
  Category {

    // OpenGL States can be written here, they will work in the whole Category scope.

    subshader { // For GPUs that can do Cg programs

      // OpenGL States can be written here, they will work in this SubShader scope.

      Pass {

        // OpenGL States can be written here, they will work in this Pass.

        CGPROGRAM

        // Put your normal Cg code here

        ENDCG
      }

      // ... More passes if necessary.

    }
    subshader { // For all OpenGL complaint GPUs

      // ... Passes that uses other rendering techniques
      // Such as normal OpenGL fixed function

    }
  }
  Fallback " fallback shader name"
```

Figure 1.2: *The structure of an effect file.*

This thesis will describe a system that presents the structure outlined above in a graphical manner. We will show how this results in a far more accessible way of creating shaders, a way that does not require knowledge about programming or the underlying graphics hardware. This results in great workflow improvements for shader development, and will enable more users to have a greater control over the appearance of their scenes.

Chapter 2

Previous Work

The previous work on shader graphs can be divided into two main categories, Industrial and Academic work. In this chapter we will discuss the most relevant work in both categories. The main difference between industrial and academic work, is that the industrial work is only documented towards using the final product, and very little information about the underlying technology is revealed. This is quite logical, as the companies does not wish to reveal any specific technology they have developed to competing companies. The industrial work also tends to be more finalized than the academic work, which mainly focuses on specific problems, rather than creating a full solution. Besides previous shader graph work, we also discuss Renderman, IDE's like RenderMonkey and FX Composer and content creating tools such as Maya and Softimage. These are all relevant industry tools that either contains shader graph editors, or has significant relevance for shader programming.

The project discussed in this thesis is an industrial project, therefore it will be relevant to compare the final result with the industrial work discussed in this chapter. We will also discuss the academic work though, as they reveal more details about their implementation, which we can analyze and compare with our implementation. We will begin this chapter with a brief discussion of the Renderman Shading Language, which were developed by Pixar, as this were one of the first shading languages.

Shader Type	Definition
Light-source Shader	The light-source shader uses variables such as the light color, intensity and falloff information to calculate the emitted light.
Displacement Shader	Displacement shaders can be used to add bumps to surfaces.
Surface Shader	In renderman surface shaders are attached to every primitive, and they use the properties of the surface to calculate the lighting reflected from it.
Volume Shader	Can be used to render volume effects in participating media.
Imager shaders	Performs pixel operations before an image is quantized and output.

Table 2.2: *Shader types in Renderman*

2.1 Renderman Shading Language

The Renderman Interface were released by Pixar in 1988 [31]. It were designed to function as an interface between 3D content creation tools, and advanced rendering software. One of the new features of Renderman, as compared to other similar products of that time, were the built-in shading language. The Renderman shading language specifies five different types of shaders as seen in table 2.2. These different shader types can easily be mixed and matched within a scene, to give the Renderman user a very high degree of control over the final image. Each of the different shaders would be implemented with one or more functions, written in the Renderman Shading Language. The shading language is very similar to the C programming language, and supports loops, if's and other flow control statements in a similar way. The shading language also supports creating polymorphic functions that has the same name, but accepts different arguments. It is notable though, that there is no support for calling a function recursively. Most of the types that C supports such as floats, strings and arrays are also supported in the Renderman Shading Language. Further more the shading language has support for additional types commonly used in 3D graphics such as transformation matrices, vectors and points. These types can have a value like normal C types, but they can further more be specified relative to a specific basis, such as object or camera space. Renderman supplies transformation procedures to transfer from one space to another, which is just a matrix multiplication that does the basis change. Variables are implicitly converted to the "current" space, if defined in another space. The current space is the space the shader executes in, normally world or camera space. If the variable is send to the shader with the Renderman interface, then it will automatically be converted to the shaders space. If the user wishes to perform other transformations, it is possible to specify transformation matrices. Similar to points and vectors, matrices can be specified to lie in a certain space, which means that the matrix will transform from current to that space.

The variable types above can further more be specified as either uniform or varying. Variables specified as uniform are assumed to be constant over the whole surface being shaded, such as a material color might be. Varying variables change over a surface. An example could be an opacity attached to each geometric primitive, which will then be bilinearly interpolated over the surface. In the Renderman shading language a uniform variable used as a varying will automatically be promoted to varying, while it is an error to use a varying variable as uniform.

Shaders are generated by creating the shader function using the relevant keyword, e.g. Surface for surface shaders. Arguments can be passed to the shader, but they are required to have a default value. Later it is possible to change that value through the Renderman Interface. Within the body of the shader there are a number of variables that are available, depending on the type of the shader. For surface shaders variables such as the surface color, normal and eye position are available. A surface shader further more expects the user to set the color and opacity of the incident ray in the shader. For other rendering approaches than ray tracing, this would correspond to the color of that pixel on the surface. When a shader function is written, it can be instantiated using the corresponding interface call such as RiSurface for surfaces.

Renderman Shading Language ships with an extensive library of functions for standard mathematical calculations, noise for procedural texturing, texture functions etc. There are also support for shadow maps and functions to find a shadow intensity by using such a map. This is one of the ways Renderman supports shadows in their shaders. Others could be based on the chosen render-method, such as ray-tracing, radiosity etc. When rendering Renderman subdivides the polygons of the model to a size that is smaller than the pixels on the screen. So in Renderman a vertex and a fragment program is actually the same thing, and therefore there is no dedicated vertex or fragment shaders.

2.2 Content Creation Tools

One of the first industrial uses of shader graphs were in content creation tools such as Maya, 3D Studio MAX and Softimage XSI [6] [5] [33]. Both Maya and Softimage XSI has a shader graph editor, where it is possible to build and edit materials. The free open-source program Blender is currently implementing their own shader graph editor as well. All of these shader graph tools work in real-time, where any change made to the graph is instantly demonstrated in the effected material. 3D Studio MAX uses a graph based representation of their material shaders, but does not have an actual graph based editor. With these tools, it is easy for the user to create a material simply by connecting nodes with different properties with wires. In Mayas Hypershade, the user will typically use a material node such as the phong node, apply a texture and maybe a bump map to create the material effect. The nodes are connected by creating a connection from a specific output slot from one node, to the relevant input slot of the other. This approach enables the artist to have fine control over which colors to use where, but it also requires that she has some knowledge about the meaning of the different connector slots, such as diffuse and specular colors, normal maps and so on. These are symbols well known to artists though, so it is generally not a problem to understand how to use them. Hypershade comes with a lot of different nodes, including both a complex ocean shader node, and more simple nodes like the color interpolation node. While the supplied nodes might be enough for most users, it would be desirable to be able to create your own nodes. This would enable a programmer to create custom material nodes, or other special effects, that the artist then can use in her shader. To our knowledge Maya does not support this though. A similar functionality could be reached with the ability to combine multiple nodes into a single group node, this would enable programmers and artists to create new nodes in an easy way. This functionality is not supported in neither Maya nor 3DS Max.

Setting up connections between nodes works well in Maya. The most common way to do it is by clicking the node you want to connect from, selecting the variable you want to use, and then clicking the node it should be connected to. Upon clicking the connected node, a popup menu will appear, with the possible variables that can be connected to. In this way Hypershade can help the user to make legal connections, by only exposing possible variables that makes sense. As an example no possible connections would appear, if the user tries to connect an UV set to a color interpolation node, because this node is not able to interpolate between UV coordinates. In a similar way swizzeling is supported, so when the user picks the variable to connect from, it is possible to chose either all the components of the possible multicomponent vector, or just a single one. If the user selects only the red component of a texture, she will be able to connect it to any single component in the variables of the node it is being connected to. More control can be obtained by using the connection editor, where the user can explicitly connect a variable in one node to a variable in the other. In the connection editor, Hypershade also makes sure that only variables of similar dimension and type can be connected.

When a material has been created, it can be a applied to an object in the scene, which will then be rendered with the constructed shader. As far as we know, it is not possible to export the shaders created in maya to a common format though. This means that shaders done with Mayas shader editor, would have to be redone with another editor or written in code, if they are to be used outside Maya.

Softimage XSI has a similar shader graph tool called Render Tree. Like Hypershade in Maya, Render Tree has a library of build-in nodes that can be used. In Render Tree it is also not possible to create your own nodes, neither by combining several nodes into one, nor implementing custom nodes by programming. In normal use, Render Tree is very similar to Hypershade. In both products the user drag wires between nodes to set up a particular material effect. Usually the user will use a build-in material and custom their appearance with texture maps and color based functions. There is a slight difference in the way these connections are made though. In Hypershade connections were handled from the same connection slot in the node, while in Render Tree it is possible to drag a connection from a specific output slot to a specific input slot. Viewing the connections more directly like this, gives a greater sense of overview over the whole graph. In Render Tree is is also possible to minimize a node, so all connections emerge from the same connector slot, resulting in the same functionality as in Hypershade. This posibility to switch seamlessly between the two modes on a pr. node basis, allows the user to have both the greater overview over connections, and smaller nodes which also increase overview.

Another important difference is that Render Tree supports connecting different types with each other, something Hypershade does not allow. When connecting a three component vector to a color value, Render Tree will automatically insert a conversion node that converts between the two different types. While it is nice to have this automatic conversion between types, it is not totally problem free. Imagine connecting a floating point to a color, which is similar to the equation: $float\ intensity = Vector3(0.0, 0.5, 1.0)$. What is the expected result of intensity ? Render Tree handles this conversion by setting intensity to 0.5 (green) by default, which can later be changed by configuring the conversion node. This default assignment might not be what the user expects, causing unpredictable results and confusion for the user. The automatic conversion nodes are used to handle swizzeling as well, by allowing the user to explicitly defining which values are moved to where in a conversion node.

When the material has been created, Render Tree supports saving the generated shader as an effect file, for use in real-time applications. While this is a huge step over Hypershade that does not support writing out the shader to a common format, it should be noted that some adjustments might be needed to make this file work in a general render engine. Making shaders work in game engines is one issue that we will address later in this thesis.

2.3 Rendermonkey and FX Composer

In this section we will give a brief description of two freely available shader programming tools, known as Rendermonkey and FX Composer by ATI and Nvidia respectively [22] [14]. Both of these tools are so called IDE's, Integrated Developer Environments, used to aid programmers with their projects. While this is different from a shader graph tool, that aim towards helping non programmers to create shaders, both of these tools exist to enhance workflow in shader creation which makes a brief description relevant.

In Rendermonkey and FX Composer the user has direct access to the code in the effect file. Whenever the user changes something in the code, it has to recompiled to see the result of the change. Besides access to manipulate the code directly, the IDE supplies an interface for easy manipulation of the variables of the effect. The user can inspect the result of changing the variables in the preview window, that is basically a real-time rendering of an object with the created shader applied to it. Both tools has great text editors with syntax highlighting and numerous examples to give inspiration to the user. They also have some debugging functionality, as it is possible to inspect render texture targets, jump to lines with errors and so on. FX Composer from Nvidia supports only HLSL where Rendermonkey also supports GLSL. FX Composer ships with more additional tools for analysis and optimization of the code though. Both tools support the Effect File format from Microsoft (.fx) for outputting shaders.

Compared to the shader graph tools discussed in this thesis, both Rendermonkey and FX Composer requires the user to understand every aspect of shader programming, in order to develop new shaders. It would be possible for an artist, or other non programmer, to experiment with existing shaders though, especially in Rendermonkey that has an artist view, where the complexity of the code is not visible. We believe that there is enough room for both shader graph tools and IDE tools in modern shader programming. For example a non programmer could author a shader in the graphical tool, and it could then be tweaked by a programmer using the IDE.

2.4 Industrial Work in Shader Graphs

In the past couple of years, there has been a few examples of shader graph tools in the industry. These are all commercial products, that it were not possible to obtain a copy of. Therefore we have not been able to evaluate them ourselves for this thesis. The following discussion of these products are based on second hand information, plus videos and other materials found on the products web-pages.

One of the most well known tools in the industry, is the material editor from Unreal Engine 3. As the editor in Unreal engine 3 is a tool for an existing game engine, materials created with their editor combines seamlessly with their engine, supporting shadows and different light types directly. We were not able to find documentation that said how this were handled though. Their tool is focused on creating material effects, much like the ones found in the content creation tools. Pre-made material nodes are available to the user. These nodes has a number of input slots, such as diffuse color, specular power, surface normal and so on. The user can connect the other nodes to these slots, to generate a custom effect like a parallax shader [18]. The material node has a small preview, that shows how this particular effect renders. It is possible to choose between a number of few primitives for this preview, but as far as we know it is not possible to see the result on in-game geometry, without testing it in the game. We were unable to find out if Unreal engines material editor supports grouping several nodes into one group node, or which format they use for the effect file. A reasonable guess would be that they use .fx and the high level shading language (hlsl). Other nodes can also have a preview field. Texture nodes displays the full texture, while other nodes may display information relevant for them. From second hand sources, we have learned that the editor ships with many different nodes, both higher level material nodes, but also low level nodes that contains a single instruction, such as an "add" node.

RT/shader Ginza is one of the most advanced shader graphs in the industry so far [32]. The shader graph tool features render to texture, assembling multiple nodes in templates, high/low abstraction level and advanced lighting models. As with the previous tools, the user creates graphs by setting up connections between individual slots in nodes. Ginza ships with the most common nodes, such as material nodes, texture nodes and so on. Whether it is possible to use the Ginza SDK to create custom nodes, were not clear from the sparse documentation we were able to find. Ginza is a stand alone product, that can be used to generate shaders in an easy way. Using those shaders in a third party game engine, is up to the individual user to support though. According to the Ginza developers, it can require some effort, to make the shaders created with Ginza work in game engines. Further more, supporting shadows and different light types, requires the user to create multiple versions of the shader (at best), or depending on the engine used it may be impossible. The documentation we were able to obtain, does not clarify how types are distinguished nor how transformation between different spaces are handled. It is also questionable if Ginza supports more than one pass, or has a nice interface for setting up blending and other state variables, as this were not demonstrated in the documentation. Ginza were originally sold from the RT/Shader web-page, but all information from the company has disappeared, and their support team does not respond to our requests.

Yet another shader graph application is Shaderworks 3d [7]. Like Ginza Shaderworks is primarily a stand alone tool, but they did feature a SDK for integrating the Shaderworks shaders with real-time engines. Unfortunately we did not have a chance to evaluate Shaderworks, because it had been acquired by Activision by the time we started this project. By reading the documentation left online, and studying the screenshots, we were able to obtain some information though. Unlike the other products discussed, Shaderworks uses a color coding scheme to define the variable type of the connection slots. Each different type is coded in a different color, and we assume that it is therefore not possible to connect two slots of different type. Conversion between different spaces is not mentioned, so we assume that this would have to be set up by the user. The documentation also does not say anything about the ability to group multiple nodes together, which could be useful to make custom shader blocks, which could then be reused in other projects. The output of Shaderworks is a MS .FX file, so Shaderworks should be able to handle rendering states and multi passing, but these features were not discussed on the feature list though. Integrating the exported .FX file can be done with an integration SDK, which uses callback methods to handle constant value, texture and mesh updating. The documentation for the SDK does not explicitly discuss integrating these materials with shadows and different light types, and as shadows are not discussed in any of the Shaderworks documentation, nor visible in any of the screenshots, we assume that this is not possible.

2.5 Academic Work in Shader Graphs

The most original work on representing shaders using graphs, were presented in the Siggraph 1984 paper by Cook [10]. Cooks paper discussed building shaders based on three different types; shade trees, light trees and atmosphere trees. Differentiating between shader types were later adopted by the Renderman API, as discussed earlier in this chapter. In Cook [10] shaders were described in a custom language, which used build-in functions as nodes, and supported the most common mathematical statements to connect these nodes. Custom keywords such as *normal*, *location* and *final color* were used when compiling the shader tree, to structure the tree and link with geometrical input. Using different spaces, such as eye or world space, were supported. The paper does not clarify if any automated approach to convert between spaces were supported. The original work by Cook were very much ground level research. The custom made language were aimed at programmers, and therefore there were no automatic detection of type mismatches or a GUI interface. The work also did not discuss real-time issues and optimizations, as this were not so relevant at the time. The paper did describe how many interesting material effects could be authored using shade trees though, and also how shadows in the form of a light-map could be used.

Building on top of Cooks paper, Abram et al. described an implementation featuring both a GUI interface, as well as a low level interface [2]. Their paper primarily discusses the practical issues regarding implementing Cooks shade trees, with a main focus on the user interface. Their primary contribution, a GUI interface for shader creation, featured type checking, click and drag functionality and a preview field. Using their implementation users can visually author shaders in an easy way. They do only discuss the creation of raw shader files though, and not how they can be used in game engines or other software. They also does not discuss how to match variables of different type, which could be done through swizzeling or automatically inserting type converters. Converting between different spaces are not discussed, and ways to combine with shadow or lighting calculations in a generic way were not mentioned.

The 2004 paper by Goetz et al. [20] also discussed the implementation of a shader graph tool. Their tool was geared towards web graphics, and stored the resulting shaders in XML files instead of an effect file. The implementation supports functionality such as swizzeling, setting OpenGL state and grouping multiple nodes in a diagram node. They check for type mismatching but does not try to correct those errors automatically. Their implementation seems to be geared towards programmers, as it displays variable types in the editor, does not assist in converting between spaces and their nodes have very technical names and appearances such as "Calculate_I_N", and it is therefore doubtful if this tool can be used by non-programmers. The paper does not discuss how the outputted XML files can be integrated into a real-time engine, and therefore integration with lighting and shadowing is not discussed.

The latest shader graph system is discussed in McGuire et al.[24]. The approach discussed is very abstract, as the purpose of this tool was to hide all programming relevant information. In their system the user indicates the data-flow between the nodes, using only a single connection arrow. This is different from the previous work, as these relied on the user to connect specific output slots to specific input slots. To generate the shader file, McGuire et al. used a weaver algorithm that were based on custom semantic types. These types abstract out dimensionality and types such as vector/normal/point, precision, basis and length. Using the flow indications set up by the user, the weaver connects the individual nodes by linking the variables in two nodes that has the best match. This weaver automatically handles basis transformation and type conversions, by finding a predefined conversion in a lookup table, based on two slightly different types. In order not to connect very different variables, a threshold for this lookup were implemented. Further more their implementation detected and displayed individual features in the authored shader. The shader trees generated with this tool is very compact compared to several previous tools. It can be a little difficult to understand the tree though, as only a single connection between two nodes are displayed. It is therefore not possible to see or control details about which variables that are connected to where, or even which variables a certain node has. We obtained a copy of their final product for evaluation, and found that it were very difficult to understand what was going on behind the scenes. As variables are automatically linked, it is important that the user has a good understanding of each individual node, so correct flow dependencies can be set up. Furthermore it is very difficult to debug a shader created with this tool, as you do not know what gets linked to what, if anything is converted and so on.

The output of the weaver is a GLSL shader program. The paper does not discuss how this program can be integrated with engine dependent lighting and shadows. Their implementation also does not support grouping multiple nodes, nor having a preview field in each individual node, which could help solve some of the debugging problems. Further more it requires a programmer with deep graphics understanding to create new nodes, as these should use their custom semantic types in order to be linked correctly by the weaver.

Chapter 3

Background theory

In this chapter we will discuss the theory that forms the background for our project. In this thesis we discuss the creation of shaders, that can be used to create realistic material effects, which should render in real-time. It is therefore important to have a basic understanding of what causes the appearance of materials, which we will give here. Besides the material theory, we also use elements of graph theory and compiler design theory in this thesis. These will be discussed here as well, along with the considerations one must make when generating programs for a GPU.

3.1 Material theory

When describing materials there are many different variables to take into account. Which variables that are the most relevant depend on what the application is, for example a construction engineer would like to know how the durance a certain material is, while a physicist might be more interested in the electromagnetic properties. Computer graphics researchers are usually more interested in how the material reflects light, and they wish to develop functions that express this reflectance. Those functions are called BRDF's or Bidirectional Reflectance Distribution Functions, which we will discuss more in the following. We will especially discuss the Blinn-Phong BRDF model, which is used in almost every real-time rendering application. But as more power-full graphics cards has begun to appear, the more advanced BRDF's has also started to appear in state of the art render engines. We will therefore also discuss the key components of those models briefly. Other relevant characteristics of materials, such as the Fresnel effect, anisotropic reflections and so on, will also be discussed later in this section.

In order to find the amount of reflected light from a surface, one must use the reflection formula as given below. Here the formula is presented in a ray manner, instead of the integral equation that is otherwise often used. We do this because we feel that the form presented here, is more relevant for real-time graphics.

$$I_r(\mathbf{x}, \mathbf{r}) = f_r(\mathbf{r}, \mathbf{i}) I_i \cos(\Theta)$$

Where I_r is the intensity of the reflected light. $f_r(\mathbf{r}, \mathbf{i})$ is the BRDF and I_i is the intensity of the incoming light. The vectors are shown in figure 3.1, and $\mathbf{x}$ is the position currently shaded. Θ is the incident angle of the light ray, and the cosine term is known as lambert's cosine law.

3.1.1 BRDF's in real-time graphics

A BRDF is a mathematical function that express the reflection of light at the surface of a material. BRFD functions can be determined in different ways, either by measuring real materials, synthesizing the brdf using photon mapping and spherical harmonics, or by making an analytical model. Measured and synthesized BRDFs are not commonly used in real-time computer graphics, probably due to their large data sets. In real-time graphics an empirical BRDF is often used instead. An empirical (or analytical) BRDF is an analytic function, which describes how light is reflected of a surface. This formula is usually based on observations in nature, and tries to recreate a natural appearance using more or less optically correct calculations. In the case of a diffuse material, the BRDF will be a constant value, as diffuse materials radiate equally in all directions. More info can be found in Watt [37]. The formulation is given by:

$$f_r(\mathbf{r}, \mathbf{i}) = \frac{k_d}{\pi}$$

k_d is the diffuse reflection constant of the surface. The value $\frac{1}{\pi}$ is necessary in order of ensuring energy conservation in the BRDF. Energy conservation is one of two properties that any BRDF must obey. The other property is that it should be bi-directional, which means that the function should give the same result, if the light direction and the viewing direction were swapped. The result of the diffuse BRDF does not depend on the direction of the light or reflected direction, as it is just a constant value. This is not the case for specular materials though, as the specular intensity depend on the angle between the viewer and the reflected light vector. Figure 3.1 illustrates the vectors graphically.

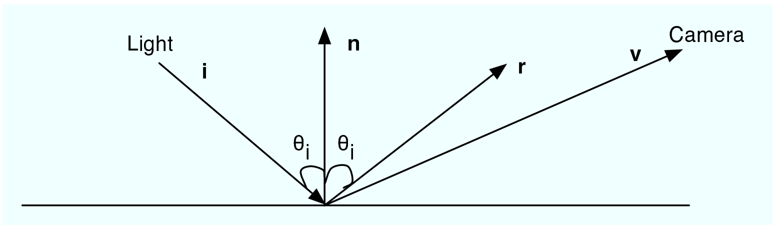


Figure 3.1: *Vectors used in lighting calculations.*

When talking about BRDF's for real-time graphics, one cannot avoid mentioning the work of Phong [30], as his work in many ways pioneered real-time shading in computer graphics. In the original Phong model, the specular contribution is calculated as the cosine between the reflected light vector, and the viewing vector. This requires that the reflected light vector is recalculated for every vertex or fragment being shaded though, which has made this model less used in real-time graphics, as reflectance calculations are somewhat expensive on older graphics cards. Instead most applications, including OpenGL and DirectX, are using the Blinn-Phong BRDF. This model were developed by Blinn [8] a couple of years after Phong presented his work. It relies heavily on the original work of Phong, but instead of using the reflected light vector, the half angle vector is used, which is the vector that lies between the light and viewing vector. The normalized half angle vector is calculated as:

$$\mathbf{h} = \frac{\mathbf{i}+\mathbf{v}}{|\mathbf{i}+\mathbf{v}|}$$

The half angle vector is then dotted with the normal vector, and raised to the exponent given by the shininess value (c_l), to give the specular contribution:

$$s = (\mathbf{h}\cdot\mathbf{n})^{c_l}$$

The full Blinn-Phong BRDF can then be written as:

$$f_r(\mathbf{r},\mathbf{i}) = \frac{1}{\pi}k_d + k_s\frac{s}{\mathbf{i}\cdot\mathbf{n}}$$

Where k_d and k_s is the diffuse and specular constants respectively. Vectors $\mathbf{r}$, $\mathbf{n}$ and $\mathbf{i}$ are illustrated in 3.1. If we put that into the reflectance formula and add an ambient term, then we will see that the result is the same as the one presented in [8].

$$i = k_a + \frac{1}{\pi}k_d\text{max}(0,\mathbf{n}\cdot\mathbf{i}) + k_s s$$

Where $max(0,\mathbf{n}\cdot\mathbf{i})$ illustrates that only the positive amount of the diffuse contribution should be used, and k_a is the amount of constant ambient light.

3.1.2 Advanced BRDF's

Measurements carried out by Torrance and Sparrow [34], has indicated that the position of the specular peak calculated by the Blinn-Phong model, is not entirely accurate for many types of materials. This is because many materials such as metals is not smooth at a microscopic level. This means that more advanced calculations must be employed when finding the specular highlight, as the incoming light will be masked or in other ways interact with these micro facets. Previous work by Blinn [8] and also Cook and Torrance [9], has used the measurements from Torrance and Sparrow [34], to develop a more sophisticated reflectance model which takes these micro facets into account. This model is called the Torrance-Sparrow (or Cook-Torrance) model. The model use three main components to calculate the specular contribution, namely the distribution function of the directions of the micro facets, the amount of light that is masked and shadowed by the facets and the Fresnel reflection law (discussed later). The model gives a more realistic appearance of metals than the Blinn-Phong model, as the highlight were better positioned, and the color of the highlight were not always white as in the original Blinn-Phong model.

Most of the more advanced BRDF's that has been developed, is also based on the theory of micro facets. Two well known models are the diffuse Oren-Nayar model [27] and the specular model by Ward [36]. In the Oren-Nayar model, the diffuse calculation is based on the micro facets, which results in a more flat impression of the reflected light. The model simulates that a high amount of the light is reflected directly back to the viewer, which makes it rather view dependent. It does yield some better results for materials such as clay and dirt though. The Ward model exists in both a isotropic and an anisotropic version.

In the isotropic version the specular highlight is found using a single roughness value, while the anisotropic version requires two roughness values. Using the anisotropic version it is possible to get non circular highlights, which can be used to give a more realistic appearance of materials such as brushed steel.

3.1.3 Advanced Material Properties

When rendering glass or other transparent materials, it is no longer enough to shade the object using only a BRDF. As the object is see-through, it is important that the shading also show what is behind the object. This can be done in two different ways in real-time graphics. The old fashioned way is to set a transparency factor, and then alpha blend these objects with the background. This will look acceptable for thin transparent objects such as windows, but not for thicker objects, as the view through materials such as glass should actually be distorted. The distortion can be found using snell's law, which can be used to calculate the angle the ray will have inside a material. Functions for finding reflected and refracted vectors are implemented in most high level shading languages, and are discussed in [15].

Having found the refracted angle, it is possible to find the refracted vector. This vector can then be used to sample an environment map, which will give the correct distorted appearance. Shading a glass surface with only lighting and the refracted contribution, will not make the glass look correct though. The reflected contribution needs to be included too, as the glass surface also reflects light, which adds a reflective look to the material. The reflected vector can be found using the formula given above, and then an environment map can be sampled to find the reflected contribution. One problem remains though, namely to find the amount of reflected contra refracted light at a given point. This relation can be found using the Fresnel formula, which is discussed in several physics and computer graphics books such as Glassner's book [19]. The relation between reflected and refracted vectors and the Fresnel equation is demonstrated in figure 3.2.

Remember that the Fresnel formula were also used in the calculations of the highlight in the Torrance-Sparrow model. In their model they calculated the amount of reflected light, and scaled the highlight with this value, so the Fresnel formulas are used for other applications than transparency too. In fact both Snell's law and the Fresnel formula is often used in real-time graphics, to give accurate light reflections of the environment in materials. The Fresnel factor is often pre-calculated and put into a texture though, as the calculations are

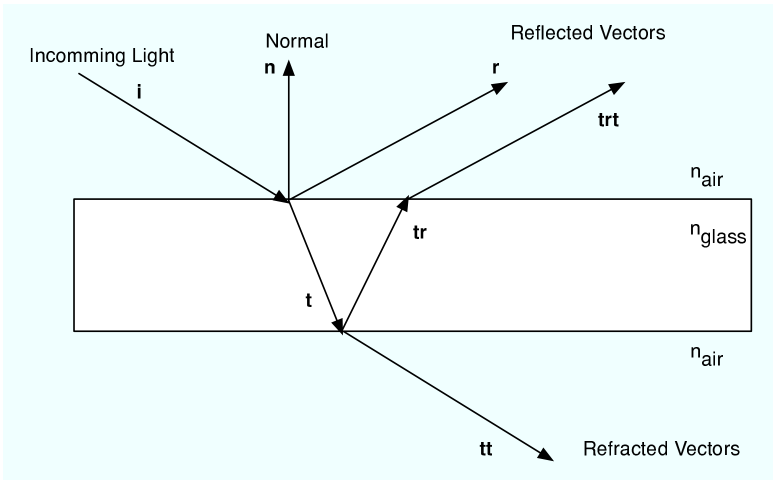


Figure 3.2: *This figure demonstrates light reflecting and refracting in glass.* n_{air} *and* n_{glass} *are the refractive indices of air and glass. The intensity of the reflected/refracted rays can be found by multiplying the intensity of the incomming light with the* **t** *and* **r** *values, where* **r** *is the Fresnel reflection value, and* **t** *is one minus the Fresnel reflection vaule.*

somewhat expensive.

3.2 Compiler Technology

A compiler is a program (or collection of programs), which translates source code into executable programs. Designing and implementing compilers is a very large task, and thus there has been a lot of research on this topic. One of the most recognized books on the topic, which we also have consulted when writing this thesis, is known as the dragon book [3]. It is not the scope of this thesis to give a thorough generalized analysis of compilers though, but as compiler technology is relevant for the project, we will give a brief description of how compilers work here. A standard compiler is often divided into two main parts, the front end and the back end. Those two parts often contain the following phases:

- Compiler Front End:

    - Preprocessing.

    - Lexical analysis.
    - Syntax analysis.
    - Semantic analysis.

In the preprocessing step the code is being prepared for analysis steps. This step can include substituting macro blocks, or possibly handle line reconstruction in order to make the code parseable. The lexical analysis step breaks the code into small tokens, and the sequence of these tokens are then syntax verified in the syntax analysis phase. Finally the code is checked for semantic errors, which means checking that the code obeys the rules of the programming language. When the code has been broken down to tokens, and has been checked for syntax and semantic errors, the compiler back end is ready to perform optimizations and generate the executable code.

- Compiler Back End:
    - Compiler analysis.
    - Optimization.
    - Code Generation.

In the analysis step of the back end, the code is analyzed further in order to identify possible optimizations. This analysis can include dependencies analysis and checking if defined variables are actually used. The analysis step is the basis for the optimization step, and these two steps are tightly bound together. In the optimization step varying code transformations are applied, which results in a more optimal representation of the code. The optimization step is very different from compiler to compiler though, and often depends on user settings for which things to optimize. An aggressive optimizer will apply code transformations that removes all the code that is not relevant for the final result. The optimizer may also perform loop optimizations, register allocation and much more. The resulting code after the optimization step is identical in functionality with the un-optimized code, but it will take up less storage, run faster or in other ways have a more optimal representation. This optimized code is then put through the code generator, which generates an executable program that will run on the target platform.

3.3 Directed Acyclic Graph

A Directed Acyclic Graph (DAG) is a directed graph with no vertex that starts and ends at the same vertex, as defined by the national institute of standards and technology [26]. In plane english that means a structure where nodes (vertices) are connected in a tree like structure, where no loops can occur. The connections are further more directed, usually from the root node and down the graph, as illustrated in figure 3.3.

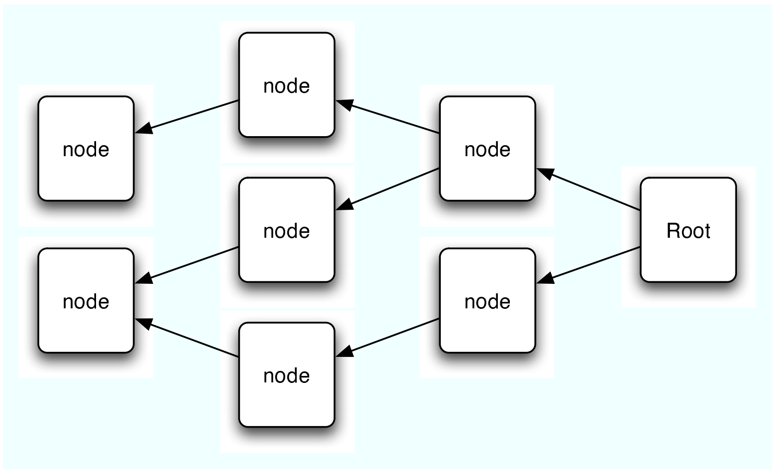


Figure 3.3: *An example of a Directed Acyclic Graph*

DAG's are usually used in applications, where it does not make sense that a node would connect to itself. A shader graph application is precisely such an application, as a cycle in the graph would yield a dependency where evaluating a node would depend on the value of the node. This is obviously not desirable, and would most likely lead to problems in the generated source code. Besides shader graph applications, DAG's are also used in compiler generated parse trees and scene graphs, as well as other applications where cycles are not desirable.

3.4 GPU Programming

During the last years, the graphics cards processors (GPU's) has evolved with magnificent speed. They have surpassed Moore's law, and are actually more than doubling their speed each year. Further more new functionalities are being added with in the same pace, which requires the GPU programmer to spend a lot of time investigating the functionalities of GPU's. This should be combined with the fact that debugging GPU programs is very difficult, the memory and architecture is different from normal system programming and graphics cards drivers does have bugs that can be difficult to track down. All of this means that GPU programming is not very accessible for normal programmers, and a substantial amount of time and experience is required to master the skills of GPU programming.

Knowledge about the GPU's functionality and capability is essential for our project, so in this section we will discuss these issues, as well as the chips basic architecture. We will also discuss the special concerns that should be taken when creating programs for GPU's. The resource used for writing this chapter were the developer pages of ATI and Nvidia, and especially the GPU related papers found there [21] [25].

3.4.1 GPU History and Architecture

Ever since GPU's became programmable, there has been gradual increases in the programmability with each new series of GPU's. In the beginning low level assembly languages were used to create custom vertex and fragment programs, but the GPU's only supported 16 instructions in such a program. This were enough to create bump-map shaders with pr. pixel lighting though, which gave a huge boost to the visual quality for games. Later GPU's supported a more accessible High Level Programming Language (like Cg, HLSL and GLSL), and up to 96 instructions in the fragment program, which became known as Shader Model 2.0 compliant cards. The first games to use these technologies were Far Cry and Doom 3. The latest generation of graphics cards supports Shader Model 3.0, which allows thousands of instructions in both the vertex and fragment programs and dynamic branching and texture sampling in the vertex program. In this thesis the primary focus is on generating shader model 2.0 compliant shader code, but it should be straight forward to expand to shader model 3.0, at least for the vast majority of the new functionalities. Further information about the architecture and capabilities of different graphics cards, can be found on the manufactures home pages.

Common for all current and previous generation GPU's is, that they are highly parallel stream processors. A GPU has a number of vertex and fragment pipelines, depending on the model. On programmable GPU's each pipeline executes the corresponding vertex or fragment program, which implements the desired functionality. This program will be static for an object, which means that all the vertices and fragments of that object, will be subjected to the same vertex and/or fragment program when rendering it. This is typical for graphics applications, where e.g. you would like to have each vertex subjected to the same view frustrum transformations, or calculate the same lighting for each pixel. As the number of pipelines increase (typically with newer GPU's), it will be possible to perform more calculations at the same time, because these pipelines work in parallel, and share the same memory, which typically holds the textures and vertex information. If the multiple objects should be rendered with different programs, it is necessary to bind the new shaders to the graphics hardware before rendering those objects. The binding is usually very fast though, so it is generally not a problem to use many different programs when rendering.

Another important feature of GPUs is that they have hardware support for vector and matrix types. This support is used in common operations such as dot products, which is implemented as a single instruction in the GPU. Compared to the CPU which does not have hardware support for this types, this makes the GPU a far more effective arithmetic processor, which is important for computer graphics, as most graphics calculations are done in 3D using these instructions.

3.4.2 Vertex vs. Fragment Programs

One of the most important issues in GPU Programming is the difference between the vertex and fragment programs. In graphics the vertex programs are usually used to perform the calculations, that can be interpolated linearly over a vertex, in order to save computations in the fragment program. This could be lighting vector, and viewing vector calculations. But more generally the vertex program are able to transform the position of a vertex, thereby altering the final position of the vertex. This can be used to calculate skinning in animations, or add pr. vertex noise, to make an object become deformed. It also indicates one of the main features that the vertex program can do, namely to scatter information into the scene. This is in contrast to the functionality of the fragment program, that are not able to displace a fragment. The only thing a fragment program can do is to calculate the final color (and depth) of the actual fragment. But the fragment program can read information from other fragments, if this information is stored in a texture. This enables the fragment program to gather information from surroundings in the scene, compared to the vertex program. which can only read a minimal amount of constant memory (if vertex texture reads are disregarded).

Often vectors and other values that can be interpolated linearly, are calculated in the fragment program. They should be used in the fragment program though, where the lighting calculations for pr. fragment lighting takes place. This introduces the problem of transferring the variables from the vertex to the fragment program. In the Cg language the transfer is done by binding the variable to one of eight texture coordinates slots, in which case they are interpolated so they are defined for each fragment on the screen, and not just at the vertex positions. It is common to build a structure holding the variables that should be transfered. In the remainder of this thesis we will call this the vertex to fragment structure.

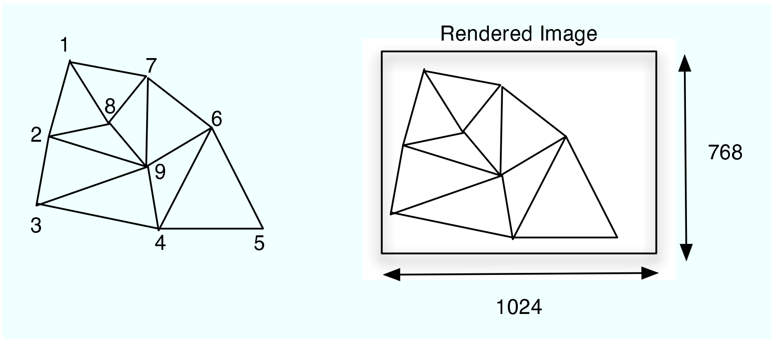


Figure 3.4: *When an object is shaded by a vertex shader, the vertex program will only excecute one time for each vertex, which is nine times in this case. When it is shaded in a fragment shader, the fragment program will execute one time for each pixel the object fills up on the screen, which is more than 400.000 times in the above case, if we assume the object takes up 60% of the image.*

In GPU programming it is generally a good idea to place as many calculations in the vertex program as possible. This is because there are almost always far fewer vertices in a scene, than fragments in the final output image. A reasonable amount of vertices to expect is 15.000 - 20.000, while a common lowest resolution for games are 1024x768 $\approx$ 800.000 fragments. See figure 3.4 for an illustration of this. A common example of this argument is to place the object to tangent space transformations in the vertex shader, when calculating tangent space bump mapping. The transformation is a matrix multiplication that is unrolled to 3 dot products by the compiler, giving 6-9 additional instructions if 2-3 transformations are to be made. It can give substantially extra performance to move these calculations to the vertex program.

Chapter 4

Requirement Specification

The primary goal of the project discussed in this thesis, is to create a shader graph editor, that will enable non programmers to create custom shaders for their applications. The tool should behave somewhat similarly to the ones discussed in Chapter 2, we have tried to pick out the good ideas of their tools, and added new functionalities which should make this tool even bette. The following paragraph gives an overview of the editor created in our project.

The user of our product will be able to generate shaders by connecting the different nodes shipped with the system. If new nodes are required, it will be possible to implement those by using a simple interface. Measures are taken to keep the product and resulting shader type safe, and to aid the user in converting between geometrical spaces. Our nodes can display a preview field, which shows the result of the subgraph from the node. This will aid the user to debug complex graphs, as it will be possible to see where a problem arises. Later in this chapter we will discuss the requirements of our shader graph editor, which should shine some more light over its functionalities.

Early on we decided to integrate our work very closely with Unity. There were several reasons why we made this decision. The most important issue was workflow. Integrating the shader graph editor with Unity, would allow the user to work on shaders in the final scene, without the hassle of having to importing them from an external application. Working on the actual scene geometry when creating the shaders, gives the user possibilities to better obtain the desired look, without shifting back and forth between applications. Another very important issue is the integration with shadows and different light types. As light attenuation is based on the type of the light source (point, spot, directional), this information must be available for the fragment program to calculate correct attenuation , which can make it difficult to use a third party application to create a shader. We will talk about that in greater detail in Chapter 6. The main issue in integrating with Unity, is that we will have to get familiarized with the rather complex code-base of Unity, to be able to perform the integration. It has taken about four years to develop Unity, and the product consists of more than a million lines of code, so it should be expected to use a fair amount of time to understand the code we will need to update. The fact that this project is carried out in collaboration with the company behind Unity, is of significant importance, as it will be possible to obtain first hand support from their developers.

The requirement specification given by this chapter does not contain a thorough listing of all the functional and non-functional requirements, which will be implemented in the final product, as most of these features are obvious in a system of this type. We focus on defining the target user group, discuss the constraints and give a description of the most important functional and non functional requirements. Finally we identify a couple of issues, and discuss precautions that should be taken to avoid that they cause problems in the future.

4.1 Target User Group

As we have chosen to integrate our shader graph tool into Unity, the target user group is the same as that of Unity, which is a very diverse group. Unity is primarily targeted towards indie game developers, creating webbased or casual games. The ease of use combined with the possibility for creating high quality applications, also appeals to people doing architectural visualization, and to educational use in both high-schools, design schools and primary schools. While this is a very diverse user group, they do share some similarities, namely that the user group is most likely a single person or a small team. This means that the user must have a very broad knowledge about the game development, knowing how to script behaviors, create shaders and possibly also do the artwork. This kind of user can be expected to know basic scripting, and to me familiar with graphics expressions such as normal maps, alpha blending and so on. It is unlikely though that this type of user is able to program shaders, as this requires a deeper intimate knowledge about the underlying graphics hardware. This fact has been proven by the Unity community, that has generally been able to create great casual games, which only used the shaders which shipped with Unity. Very few users has created shaders in the one year Unity has been on the marked. The work presented in this thesis, aims towards enabling this user group to create custom shader effects for their games.

4.2 Constraints

The primary constrain for this project is the integration into Unity. The integration means that we should strive towards using features already implemented whenever possible. If a feature that we need is not present in the engine, we should try to implement it in a generic way, so this feature would be of overall benefit to the engine. To uphold backwards compatibility, we are also required to use the effect file language (shaderlab) and shading language (cg) that were already used in Unity.

The primary goal for our implementation, is to produce a shader creation tool that is highly accessible, that people with little or no experience in shader programming can use. Therefore the accessibility also constrains the design, as certain features that enhances flexibility will be disregarded if they compromise the ease of use.

The implementation will need good text rendering, something Unity does not currently have. Therefore a method to render true type fonts on the screen, in a pr. pixel correct manner, will have to be created. The open-source library Freetype should be used for handling true type fonts, and a generic system for rendering the text should be made [17].

The main programming language of the project will be C#. This choice is made because C# is a powerful but yet accessible language, which has many functionalities that will aid the implementation. Most of the work will be done inside Unity using C#, which will also avoid long compilation and linking times, which is otherwise inevitable if changing the base code of Unity. Some changes to the base code of Unity will be needed for the integration though. These changes will be implemented in C++ and Objective C where relevant.

4.3 Functional Requirements

A preview rendering should be created in relevant nodes, which shows how the subgraph would be rendered. This functionality can function as a debugging method, as it will be easier for the user to identify where an error occurs.

Converting between different mathematical spaces (object space, world space, eye space, tangent space) should be done automatically. Most of the users in our identified user group, will find it difficult to know when to use which space, which could cause problems with for example normal mapping. Therefore the application should abstract these spaces out, and handle the conversion automatically.

It should not be possible to make connections between different slot types, unless connecting from a floating point value. Color codes should be used for easy identification of the different types. The argument for this is that it is that an connection between a two dimensional vector and a 3 dimensional vector does not make sense. An automatic conversion could be made, but it is impossible to know if this conversion is what the user actually wants. Disallowing the connection forces the user to think about what she wants with this connection, and use appropriate swizzeling to set it up. Connection a floating point to a vector of higher than one dimension is allowed though, as it is rather explicit that the user wishes to expand the floating point to the vectors dimension.

Swizzeling should be supported. Swizzeling is the most sensible way to allow a user to modify the dimensionality of vectors, and to extract specific channels from a vector or matrix. These are features that are often used, so it is important to support them.

The created shaders should support shadows, different light types and possibly other engine specific features.

Both the vertex processing and fragment processing part should be a part of the tool, so the user can create shader graphs that does modifications at both of these two stages in the graphics pipeline.

Creation of new nodes for the graph should be possible, through a simple node interface, or by grouping several nodes into one group node.

The user shall be able to setup specific OpenGL rendering states, e.g. to support transparency, turn off culling etc.

The system must be integrated with the Unity Game Engine.

4.4 Non-Functional Requirements

Look and Feel: The GUI should be reasonably nice to look at. Most Mac users expects a certain look for a GUI, for example in respect to the use of transparency, that we would like to have. Further more the user group is mainly creative people, so the GUI should stimulate their creative thoughts whenever possible. The GUI will be designed so that it is similar to the content creation tools, as most of the users can be expected to have some familiarity with those applications.

Usability: Usability is the primary concern in this project. The interface should be simple to use, so names and conventions should be chosen to match those the user is expected to know. The product should be easy to use for a user who knows basic computer graphics terms, and has a basic knowledge of what a shader is and how it works. Requirements from programming such as variable types and different mathematical spaces should be abstract, so users with no knowledge about these concepts still are able to use the tool.

Performance: The created shaders should perform as optimal as possible, and match the performance of hand coded shaders. The design must consider ways to optimize the generated shader code. The GUI interface should be interactive, but high performance is not as important. Performance of both GUI and generated shaders should scale nicely, so it is possible to create both simple and more complex shaders which performs optimally.

Maintainability: It is important that the product is very maintainable, and easy to extent with new additions, especially because it can not be expected that the original developer (the author) will be available to maintain the product in the future. To accommodate this, frequent code revisions will be held with the lead programmer of Unity, to give him an imitite knowledge of the work done. Key components of the code should further more be as simple and extensible as possible.

4.5 Identified Issues

Issue 1: The integration with Unity, as it is a complex piece of software, so it will require some time to understand how to make the best possible integration. We feel that the integration is so important that the extra time and effort will have to be spend nontheless.

Issue 2: GLSL would be a better long term high level shading language to use, as it performs better on ATI hardware, and shader model 3 could be difficult to use with Cg on non-nvidia hardware. Therefore care must be taken to support a future transition to GLSL.

Chapter 5

Design Considerations

In the previous chapter the requirements of our project were specified. We will now discuss how to design a system that will meet those specifications, focusing on ease of use and extendability. We found that this design process was very important, in order to ensure that the final product was maintainable and extendable, which is why a substantial amount of time was used on design related issues. The design related issues were discussed with the lead and graphics programmers of OTEE, in order to come up with the best possible solution. In this chapter we begin with a discussion, of the individual parts in the system. We start by describing the design of the GUI system, which is responsible for the look and feel of the application. Then we discuss the components needed in each individual node, which leads into a discussion of the connector slots that handles connections between nodes. We then discuss how our group functionality should be created, and finally we discuss the design of the compiler that generates the actual shader, and the new processing system we will introduce to bind these shaders in the engine.

This chapter intents to discuss the design considerations we had early in the project, so therefore it is mainly conceptual ideas that we discuss here. In the next chapter we will go into details regarding the actual implementation of the system, based on the concepts presented here.

5.1 GUI design

One of the first things we realized when designing our editor, was that we needed a nice GUI interface, that would feel comfortable for the user to work with. Traditionally the GUI design is very important in Mac products, so we did not want our GUI to be too primitive, which is otherwise the common rule for academic work. The most important aspect of the GUI, was to make it functional. We estimated that the most frequent operation made by an user, would be to connect two slots in the graph. We therefore wanted to support a certain degree of snapping, which means that even if the user does not precisely click the slot, the system will guess that the user wanted to perform an operation on this slot. We further more decided to use distinct colors for the different slots, based on the variable type that represents, such as red for colors and so on. This is illustrated in figure 5.1. When setting up a connection between two nodes, the user should drag a wire from an output slot to an input slot. We will only support handling connections in this way, and it will therefore not be possible to setup a connection by dragging from the input to the output slots. This is actually meant as a help to the users, as it matches the left to right way most people read and do things. It would also give some simplifications to the implementation, if we only support the one direction of making connections. Should the user wish to remove a connection, then this is done by ”picking up” the connection at the input slot, and then reassigning it to another input slot, or dropping it elsewhere, which will remove the connection. Connections between two slots will be shown as a bezier curve of the same color as the input slot.

The slots should also be named, which caused an issue with input and output slots lying directly opposite of each other. We therefore decided to keep all the input slots in the top of the node, then we would have the output slots below them, and if the preview field is active that would be rendered in the bottom. The initial design considerations for the nodes, let to the schematic drawing shown in figure 5.1.

As the maximized node would take up a lot of screen space, we found that it were necessary to have different window modes, which would enable the user to minimize the size of the node. We therefore decided to make three different window types, which are discussed below.

Minimized: When a node is minimized, only the title-bar of the node should be seen. Connections to and from the node should lead directly to the side of the title-bar.

Normal: In the normal view all slots should be visible, but the preview field is hidden.

Maximized: Maximized nodes shows all of the input and output slots, along with a preview of the subgraph, which is rendered in the bottom of the node.

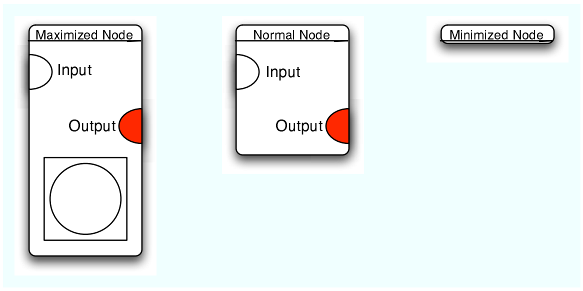


Figure 5.1: *Initial design ideas for the individual window modes of the nodes.*

Besides showing the nodes on the screen, we also need a way to move them around. This should be done by dragging the node in the title bar. We also decided that is should be possible to make multi selections, by marking several nodes with a selection rectangle.

Finally we need to be able to handle the group nodes (group nodes are discussed in detail later). We decided that the user should be able to enter a group node, which would yield a new view containing the grouped nodes. We choose this approach of entering the group nodes, because we want them to be used for creating overview in the graph. If the group nodes were rendered with information about the nodes they contain, they would probably still be large and confusing to have in the graph. Instead the user may double click the group node to enter it and view its contents. In order to navigate back to the original shader graph, we will then render the path in the top left corner of the view. This path will start with the shader graphs name, and then print the name of any entered group nodes recursively. Clicking on a name in the graph, will bring the user to that level in the graph. We will further more color the shader graph name either green, yellow or red, depending on whether the created shader runs fast on all hardware, runs on all hardware but maybe slow, or red if it can not be guaranteed that this shader runs on all vertex and fragment program capable graphics cards. This is meant as a help to the user, which indicates how the shader is expected to work on other machines. Should a shader not work on any system, then that will become apparent as Unity will print a sensible error message that says what went wrong.

5.2 Node Design

As previously discussed, we found it very important to keep the individual nodes as simple as possible. This is important as people might wish to expand the system with custom nodes, and then a simple and understandable framework for the nodes is necessary. It seems like a obvious choice to create a parent class, which holds the common functions a node should possess. We identified the following functionalities:

- Shader Code Containers (discussed in next section).
- Variables for the rendering mode, if the node should have a preview, be minimized or normal, the size of the node plus the nodes name.
- List of the input and output connector slots of the node.
- Type matcher function, which determines if a connection to a slot in this node is valid.
- GUI related functions which returns the sizes of the node, which object were clicked within the node (title-bar/slots), and functions that handles node related GUI events (minimize, maximize, enter group node...).
- Function that forces the content of this node to be fragment program code.
- A preview rendering field, where the effect generated by the nodes subgraph is rendered.

It is obvious that each node needs to store its own state variables for the rendering mode, along with the name of the node. It might not be necessary to store the size information in each individual node though. If we base the size of the node on the amount of connector slots and the window mode, then the dimension of the node can be calculated by a function in the parent class, which is what we have chosen in this project. This takes the flexibility of individual node sizes from the user, but it makes it easier to implement new nodes, because it is then not necessary to think about anything regarding the rendering of the node.

When making a connection from one slot to another, it is important to ensure that the connection is valid, which means that the types must match. The function for matching these two types can lie in both the slot class and the node class, but we chose to put it in the node as we believe we will have access to the nodes more easily. Finally we also wish to have the possibility of showing a preview rendering in each node. This rendering should show the result of the subgraph, as if it were a separate shader graph. See figure 5.2 for an illustration. We can implement this by compiling shaders, where the preview enabled node is the root node in the graph. This compilation will be performed similarly to the compilation of the whole shader graph, which we will discuss later.

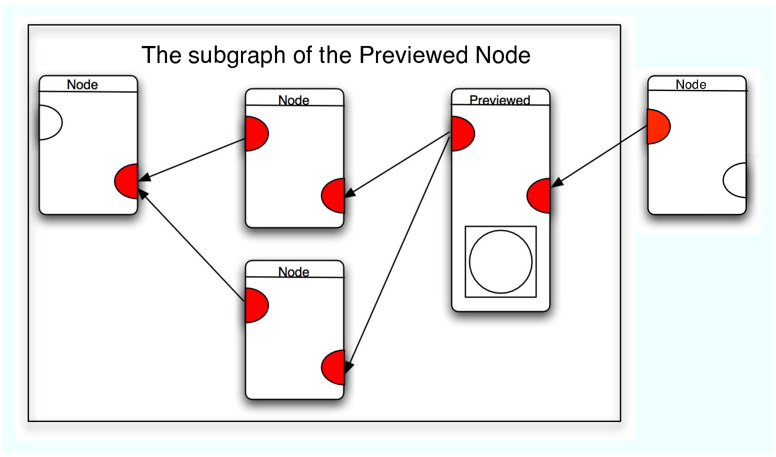


Figure 5.2: *The nodes inside the box are refered to as the subgraph of the previewed node. When the preview shader is compiled, these nodes are treated as an individual shader graph, in order to build the preview shader.*

5.2.1 Shader Code Containers

One of the most important parts of a node, is the five strings which contains the actual shader code. Each node will define its code in one or more of the five strings which are:

Property Code: The property code defines the user controllable variables of the shader, e.g. a diffuse material color editable by the user.

Header Code: The defined property variables and possible textures, are defined in the header in order to be usable in the Cg program.

Vertex Code: This code is always put in the vertex program. Vertex dependent code could be animation code which translates a vertex position, or possibly lighting calculations for pr. vertex lighting.

Fragment Code: This code always put in the fragment program. Fragment dependent code is usually texture lookups, or perhaps pr. fragment lighting calculations.

Generic Code: This code can either be put in the vertex or the fragment program, depending on the type of code the node connects to. Generic code is often simple operations like multiply and alike, which can easily exist in both vertex and fragment programs.

In this project we have chosen to deal with both vertex and fragment code, unlike former shader graph tools where only fragment code were supported. Therefore we need to have both the vertex, fragment and generic code strings. The vertex and fragment code strings are pretty self explaining, if a node defines vertex or fragment code it is put into the corresponding program at the right place. The generic code is more tricky. As there is no generic program type (only vertex/fragment types), we need to figure out if the code should be put in the fragment or vertex program, when the graph is compiled into the shader. We have chosen to solve this problem by investigating the nodes and slots that the generic slot connects to. If the node that contains the generic code is connected to any node that forces fragment code, the generic code is put in the fragment program. If not the code is put into the vertex program. We chose this approach because we find it easy to understand, and it also provides some optimization when as much code as possible is put into the vertex program by default. A more correct approach would have been to flag if the generic code can be interpolated linearly, and then put it in the vertex program if it could. We find this a lot more difficult for the user to understand though, which will make it difficult to create custom nodes, or to determine whether to force fragment code for a specific node. Forcing a node which would otherwise be vertex code to lie in the fragment program, is a feature we wish to implement to give the user more control over the final shader. Imagine as an example, that a user wishes to crate a diffuse lighting effect, by taking the dot product of the light and normal vector. This code would end in the vertex program by default, as the dot node is generic and can produce code for both program types. This will result in a pr. vertex diffuse lighting effect. If the user wish to have a pr. fragment lighting effect, it is important to supply a way of forcing this dot node to generate fragment code.

The functionality to group several nodes in a single node, yields a special node type called the group node. When a group is created, the nodes selected for grouping should be removed from the main shader graph, and moved into the group node. This should be straight forward to implement, as we can just move the nodes from the shader graphs list to a list in the group node. There are some issues with connecting to the group node though, which will be discussed in the following section and in chapter 6. If any nodes have connections to other nodes outside the group, the relevant connector slots should be put into the group node, so these connections serve as the interaction points with the grouped nodes. We will also make support for adding or removing the slots from the grouped nodes to the actual group node, in order to use make these slots available to the main shader graph. We discuss the handling of the connector slots further in the next section.

5.3 Connector Slots

The connector slots will be the users main interface for setting up connections between nodes, therefore the slot must have variables which stores the connection information. A connection should be set up by dragging a curve from an output slot to the input slot of another node. When the connection is made, the system should check if the two slots are defined in the same space, or if an transformation node should be inserted in between. It is also important to check if the connection would result in a loop in the graph. If that is the case, the connection should not be possible. The features of the connection slots are illustrated in figure 5.3.

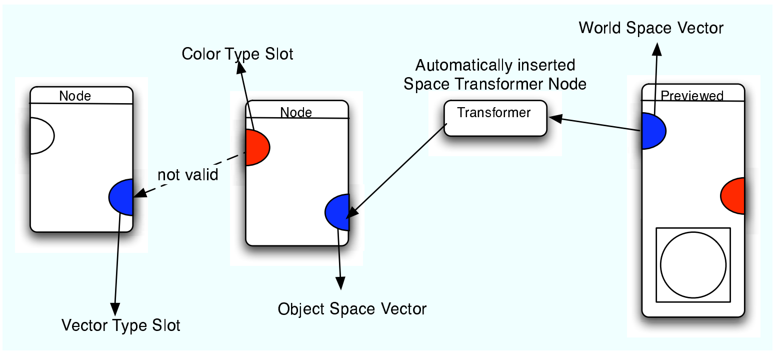


Figure 5.3: *Connector slot features. The leftmost connection is not allowed due to the different types. The right connection occurs between two vectors defined in different spaces, so an transformation node is automatically inserted.*

In respect to the final shader code, a slot should be thought of as a variable in a program. Therefore it is important to define the type of the slot. As we wish to abstract the mathematical space the variable lies in, we also need to store this type information. The two types can be of the following form:

Normal Type: generic, float, vector2, vector3, vector4, matrix3x3, matrix4x4, color, integer.

Mathematical Type: generic, world, object, tangent, eye.

Each of the mathematical types can further more be normalized or un-normalized, which gives 10 different mathematical types in total.

Most of these types are pretty self explaining. The generic type however, is introduced in order to support polymorphic types in slots, which means that a generic slot can be any of the other types. In the following we will discuss these polymorphic types in greater detail, and we will introduce a method to automatically transform between the different mathematical spaces.

Each slot should also have a "published" flag. If the slot belongs to a node inside a group node, publishing will mean that this slot can now only be accessed from the group node. If the slot belongs to the top-level shader graph, publishing it will turn it into a property variable that can be edited in Unity through the material interface.

5.3.1 Polymorph Types

Lets consider a typical node, such as the multiply node. This node should accept two inputs, and give one output which is the result of the multiplication. As we wish to introduce types in the connector slots, in order to make our generated shader type safe, there are two different ways to design the multiply node. One way could be to make many nodes, which support each different type the system operates with. This is the straight forward way of doing it, but it will be damaging for the workflow for the user, as she will have to constantly think in terms of types when she adds new nodes. Another and more elegant way would be to create a generic type that will function as a polymorph type, which means that this slot can be of any type. We have chosen the last method, and have introduced a generic type in the relevant nodes.

The concept of polymorph types does introduce a new set of problems though. Imagine for example that a user should want to connect a two and a three dimensional vector to the same multiply node, what would the output then be ? If we simply put $Vector2 * Vector3$ in our shader code, it will not compile. In other cases it might downcast one of the vectors automatically, but this is also not desirable as the user might become confused and will wonder about what is happening. We have therefore decided that when a user connects a non generic type to a slot that is of the generic type, the rest of the generic types in that node should be set to the same type as the connecting type. If many nodes of generic types are connected together, all of the generic types in this graph will be set to the connecting type. Only the types that are already generic will be changed into the connecting type, so if a node has both generic and other types, only the generic ones are updated. If the last connection from a node (or graph of generic typed nodes) are broken, all the types should be reset to the generic type. We illustrate this type updating in figure 5.4.

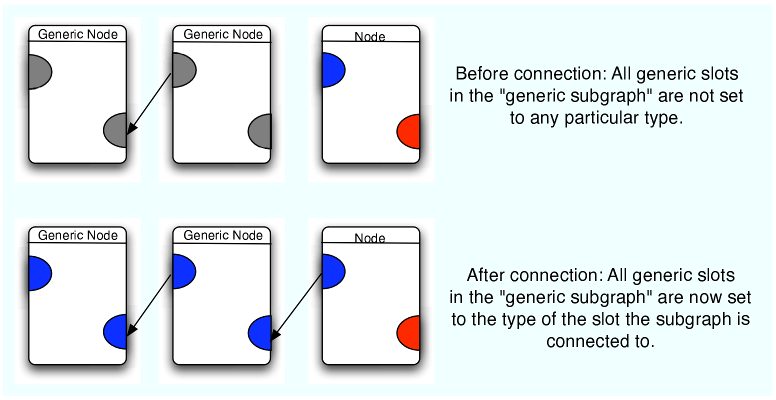

Generic Node
Generic Node
Node
Before connection: All generic slots in the "generic subgraph" are not set to any particular type.
Generic Node
Generic Node
Node
After connection: All generic slots in the "generic subgraph" are now set to the type of the slot the subgraph is connected to.

Figure 5.4: *Demonstration of connector slot polymorphism.*

We have considered the case where a node might have multiple generic types, which are independent from each other. In such a node it might not be desirable to set all the generic types to the type of the incoming connection. A possible solution to this problem could be to create a layer of generic types, so that all types that are of the type "generic1", would be set to the same type upon a connection, while "generic2, generic3" and so on remain generics. This idea will remain a design suggestion though, and will not be implemented as we do not feel we really need it, and it could possibly confuse the users. If the future shows that this becomes an issue, it should be easy to update the program to have this feature.

5.3.2 Mathematical Type Transformations

We have chosen to introduce another type, the mathematical space of the variable, to our connector slots also. In computer graphics vectors, points and other variables can exist in different mathematical spaces, such as object, world, tangent and eye space. Many of the users defined in chapter 4 might find it daunting to think in terms of these spaces, so it is desirable to handle these space conversions automatically. We have chosen to handle these mathematical types by adding a second type defining variable to our connector slots. When a new connection is created, we check the mathematical type of the slot we connect from and the slot we connect to. If those two types does not match, we will insert a converter which converts the slot we connect from to the proper space.

We have considered whether to display these inserted transformer to the user or not. Some users might find it confusing to see if something is automatically inserted, and others might find it annoying if they can not see what is going on behind the scenes. We have therefore chosen to support both displaying and hiding the inserted transformers. If the transformers are displayed and a transformer is deleted by a user, we maintain the connection between the input and output slot of the transformer, but remove the transformer itself. That should give the user the possibility to force a "illegal" mathematical type connection, if that for some reason is desired.

A generic mathematical type should be understood as this slot does not care which space it exists in. The generic types has the same functionality as the polymorph types discussed above, so when connecting a slot of the world space type to a generic slot, the other generic slots in this node will have their type set to world space.

5.4 Compiler Design

One of the most important components in our project is the shader graph compiler. It should be noted that we use the term compiler, though strictly speaking the result of the compilation is a shader file, which is not an executable application, as it needs to be processed further by the game engine. As the output of a compiler should be an executable program, one may argue that the shader graph compiler is not really a compiler, but rather a code transformer that transform form graph representation to shader code. Seen from the view of the shader graph editor though, the created shader is executed by the engine, which makes using the compiler term suitable in that context. The compiler is responsible for translating the user generated shader graph into a shader file, which can be used in a game engine. The compile process is broken into several individual phases:

- Compiler Front End:

- Preprocessing:
  - Make variable names unique (uniquefy).
  - Setup fragment code flags.
  - Setup publishing.

- Compiler Back End:
  - Code Generation:
    - Build vertex to fragment program structure.
    - Build individual shader code string.
    - Build final shader.

The front end of a compiler is usually responsible for preprocessing and lexical, syntax and semantic analysis. In our system however, only the preprocessing phase is relevant, as a graph is always ensured to be valid, and in a process-able graph form. Therefore the shader graph, which is the source for the compiler, is already syntaxical correct, and no lexing or semantic analysis is necessary. The front end of the compiler discussed in this section, does only do preprocessing which are divided into the steps illustrated above.

In the first step, all the variable names are made unique, so that we do not risk having two identical variables in the generated shader code. Remembering that the variable names are actually just the names of the connector slots, it is easy to realize the relevance of this step, as it is obviously viable to have several identical nodes in a graph, which could lead to identical variable names. In that case the names should be made unique.

The next step is to set up the fragment code flags. As discussed above, we have introduced a method for forcing a generic node to generate fragment code. This step scans the nodes for fragment code dependency, and ensures that all following nodes are also set to be fragment code dependent. The dependency of the node can be set it two ways; either by a user who forces fragment code, or by the node itself, if it is a texture node or another type of node, which must generate fragment code.

Finally we need to setup the published connector slots. This step is only concerned with slots that should be published as properties, so this can be done by checking all connector slots in the top-level graph. If a slot is published, it should be added to the properties of the shader. There is one problem though, namely that Unity only supports a few specific types to be published as properties. There are floating points and colors. We have therefore chosen that it should only be possible to publish those two types to properties.

A compilers back end normally consists of an analysis step, which is tightly connected to the optimization step. The compiler presented here transforms from one very high level of abstraction, to another highlevel programming language. The highlevel code is then compiled by the Cg compiler, which means that most of the optimizations are done by the Cg compiler. There are some possible optimizations that we will do though, as for example the vertex to fragment struct, where we optimize the number of interpolation slots used. The Cg language supports transferring eight four component vectors, two colors and one position in this structure. In this project we only use eight vector spots, as some game engines use the color slots for other purposes. The issue is that transferring just a single float in such a vector is just as expensive as transferring the full vector, so we want to pack the variables we transfer into the four components. An example could be that when transferring two UV coordinates (two component vectors), they are packed into a single four component vector. In order to make it work, we will also have to update the UV variables so they can remember which part of the vector they lie in. This can be done by appending the swizzle components ".xy" or ".zw" to its name. The variable packing is illustrated in figure 5.5.

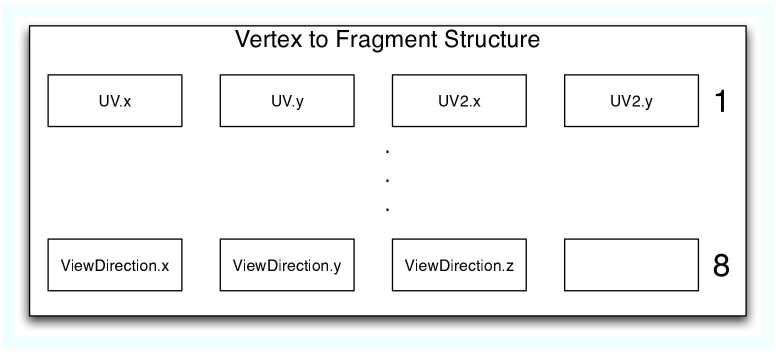


Figure 5.5: *Abstracted illustration of how variables are packed into the structure. The horinzontal boxes indicates a single component in a four component vector. In this example we illustrate how it might look if we packed two UV coordinates and the viewing vector, in the first and last texture coordinate slots.*

In our implementation of the vertex to fragment structure creating function, we will pack the variables as described above in order to save space, and to transfer as few slots as possible. Other optimizations that we will do is the more advanced material nodes, that we will discuss later.

The last step of a typical compiler back end is the code generation. As previously discussed the nodes store code fragments, which the compiler must concatenate into a final shader program during this step. The compiler discussed here starts by building the vertex to fragment program structure though. This structure is responsible for carrying variables from the vertex program to the fragment program. During this process the values in these variables will be interpolated linearly by the graphics hardware. The structure can be build by investigating each connection in the whole graph. If a slot forced to generate fragment code, connects to a slot that generates vertex code, it is necessary to put the variable of the vertex code slot into the structure, to ensure that it is available in the fragment program. Further more the fragment code variable should be updated to accesses the carried variable in the structure.

Generating the shader code strings is relatively simple. Each node defines a function that initializes its code strings, based upon which connections the node has. Most nodes has only one node to initialize its code strings, but for some nodes such as the material nodes depend, the initialization depends on the connections of the node. If for example a normal map is connected to the normal of a material node, this node needs to initialize its tangent space code instead of its normal object space code. We could have omitted this feature, and relied on our automatic space conversion scheme, but that would force a space conversion in the fragment program for the above case, which is not very efficient. When the nodes has initialized their strings, we first concatenate them individually for the five different code strings discussed above. This results in five independent fragments, which we then concatenate to give us the final shader program.

When a node is set to show its preview field, the shader of that nodes subgraph has to be compiled. We do this in the same way as with the normal compilation, except for one important difference. As the node with the preview field is probably not an output node, then we do not know which output slots that should be used as the final color, vertex position or alpha value. Therefore we simply iterate over the output slots, and select the first slot of the color type to be the output color in the shader, and similarly with the vertex and alpha values. This means that nodes with two or more output slots of the same type, should have the the most important slot at the top of the list.

5.5 Game Engine Integration

The previous sections of this chapter discussed how to design a shader graph editor, but there were no real mentioning of how to integrate it with an engine. In order to use shaders in game engines, the engine must have a way of processing and reading the effect files. Changing a game engine to a highly shader driven approach, is a rather complicated task, and we will not go into full details about that here. Readers with an interest in this subject should read the GPU Gems article by O'Rorke [28]. If the game engine chosen for the integrating already had support for reading and rendering with effect files, one would only have to update this support to handle the shader graph shaders too.

The Unity game engine handles shader binding in a two step method. In the first step, any vertex or fragment program is compiled into an assembly program, using the Cg compiler from Nvidia. The original high level code string is then exchanged with the assembly level code, but only in the internal representation, no update of the original shader file occurs. In the next step the processed shader is parsed to an internal rendering structure, using a Yacc based parser and Lex for doing the lexical analysis. The rendering step then uses the internal representation of the shaders, when rendering the objects in the scene.

As the shader graph editor outputs a normal and well defined effect file, we could have chosen to use the shader binding scheme already present in Unity directly, but this presented a problem with supporting different light types, shadows and other shader dependent effects. To illustrate this problem, consider a shader which shades an object using phong lighting calculations. Depending on the light-source type, which can be either a spotlight, a point light or a directional light, the emitted light should be attenuated differently. This attenuation must take place in the fragment program, if the other lighting calculations also are carried out there, in order to correctly attenuate the light. This means that there must exist individual fragment programs for each possible attenuation type, and also for handling shadows and other effects that requires to be calculated in the fragment program. The way Unity used to handle this were based on a very un-extentable autolight system, which compiled the shaders to have support for multiple different light-types. The un-extentability of the system meant that a new way had to be introduced, that would have support for shadows and other effects in the future. As the support of multiple light-source types and effects like shadows were important for this project, we decided to come up with a novel approach to the shader processing system.

The primary problem faced in the shader processing is how to take a single shader file, and process the vertex and fragment programs, to generate multiple programs that are identical, except for the alteration caused by the effects that the shader are compiled to support. In the case of light sources, this means that all the vertex and fragment programs in a shader that handles attenuation, should be compiled in multiple versions that are identical, except for the difference in the attenuation calculations. To keep the system as extendible as possible, we have decided to do the compilation based on keywords that are defined in the shader graph, or in the programming language if the shader is hand-coded. We will illustrate this keyword based approach, by taking a closer look at how the light node should work. The light node should have output slots for the light position (and/or direction), the color of the light and the attenuation value. When multiple lights are used in a scene, we still need only one light node, as Unity renderers each object once pr. light source, so we only need to handle the light source that is being rendered from at that moment. When the attenuation slot is used, the three keywords "point", "spot" and "directional" should be put into the shader. Then when the shader is compiled, the processing system will compile each relevant vertex and fragment program once, with each of these keywords set. This will create three versions of the programs that each support a different attenuation scheme. In order to make this work, the attenuation function will need to be placed in an utility library, and this function should then always be used for attenuation calculations. Further more the function must define three different paths, corresponding to the three keywords discussed above. In the case of other effects such as shadows, it should be possible to use the same keyword approach in a similar way. This would happen by introducing the keyword in the shader, which would cause the shader to be compiled with this keyword defined. The corresponding function that actually implements the effect should be defined in multiple versions too, in order to support the different paths compiled. Figure 5.6 shows a diagram of the integration model used in this thesis.

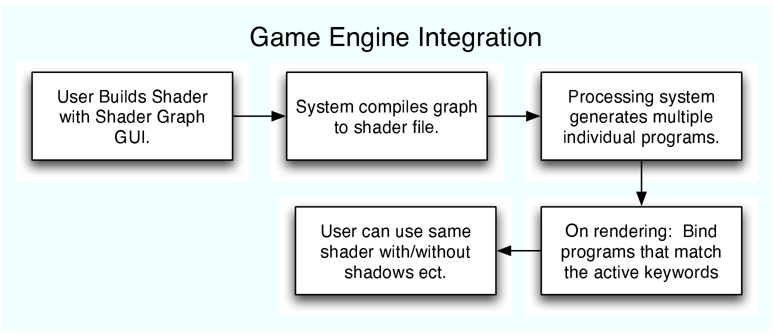

Game Engine Integration
User Builds Shader with Shader Graph GUI.
System compiles graph to shader file.
Processing system generates multiple individual programs.
On rendering: Bind programs that match the active keywords
User can use same shader with/without shadows ect.

Figure 5.6: *Diagram of the path from shader graph to processed shader file.*

When there are more than one effect in play, the system should automatically create all the possible combinations of the effects. An example could be in the case of light attenuation and shadows, where the result should be six different versions of the programs, namely the three attenuation methods in both a shadowed and un-shadowed way. We propose doing this by separating the keywords of individual effects into separate lines. Each combination of keywords should then be created by the compiler during the processing. We further more wish to support keywords that only affect vertex programs or fragment programs, along with keywords that affect both. We will discuss the implementation of this keyword system further in the next chapter.

Of cause one must consider the amount of programs that are being created. If there are several different effects with several keywords each, the number of generated programs will quickly explode. In chapter 9 this problem will be discussed further, and we will present an example that demonstrates the complications encountered in practice.

Chapter 6

Implementation

In this chapter we will discuss the implementation of the shader graph tool, based on the design discussed in the previous chapter. From the beginning of this project, we chose to integrate the shader graph very closely with Unity, for reasons previously discussed. This tight integration had significant implications for the implementation of the tool. First and foremost, we had to make changes to the unity engine itself. As Unity is such a complex piece of software, a fair amount of time was used to get acquainted with the code-base. For the same reason, we early on decided to keep as much of the development as possible, apart from the main code-base of Unity. Therefore the majority of the tool were developed within Unity, using the C# programming language. The additional engine based work that were carried out for this project were presented to the scripting interface too, so we could keep our main development in the C# scripts. Besides the fact that we did not have to bother with the main code-base of Unity, we also had a far less compilation time overhead. Recompiling our entire project takes one or two seconds, where recompilation of the whole Unity engine can take up to 20 minutes.

Figure 6.1 demonstrates the class diagram for the implementation, which shows the dependencies within our project.

In the remainder of this chapter will discuss the implementation of our project,

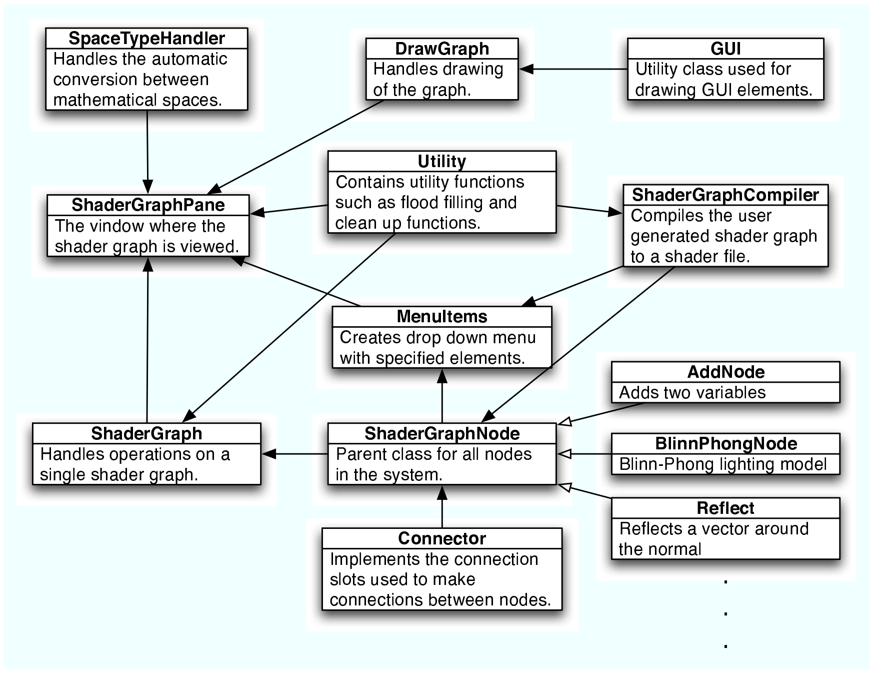

SpaceTypeHandler
Handles the automatic conversion between mathematical spaces.
DrawGraph
Handles drawing of the graph.
GUI
Utility class used for drawing GUI elements.
Utility
Contains utility functions such as flood filling and clean up functions.
ShaderGraphPane
The vindow where the shader graph is viewed.
ShaderGraphCompiler
Compiles the user generated shader graph to a shader file.
MenuItems
Creates drop down menu with specified elements.
AddNode
Adds two variables
ShaderGraph
Handles operations on a single shader graph.
ShaderGraphNode
Parent class for all nodes in the system.
BlinnPhongNode
Blinn-Phong lighting model
Reflect
Reflects a vector around the normal
Connector
Implements the connection slots used to make connections between nodes.

Figure 6.1: *Class diagram of our system. A closed arrow indicates where a class is primarily used, while a open arrow points to the parent class.*

and we will discuss which choices we had to make during this implementation. We start by discussing the implementation of the GUI system. We then discuss the overall implementation of the shader graph tool. This discussion will cover the implementation details about the graph system itself, along with implementation details about the nodes. We will then discuss the implementation of the compiler, and finally round off with a closer look at the integration with the Unity engine.

6.1 GUI Implementation

As we discussed in the last chapter, we chose to do most of the GUI system ourselves, instead of using Cocoa which is the Mac OS X windowing system. This was primarily because we wanted full control over the look, but also because we required features such as OpenGL renderings inside nodes and alike, which could become problematic with regular Cocoa code. Another strong selling point was that this system would become platform independent, a nice feature to have if

Unity should be converted to a windows application in the future.

We implemented the GUI system by drawing textured quads and curves, inside our shader graph view in Unity. This view is actually just an OpenGL viewport, that it is possible to render into. The nodes were simply rendered as quads with a specific GUI texture applied. Predefined texture coordinates were wrapped into easily understandable names, which made it possible to use simple function for drawing the textured quads. The texture we used for the nodes is shown in figure 6.2. The texture was created in part by the author, and by the people behind Unity.

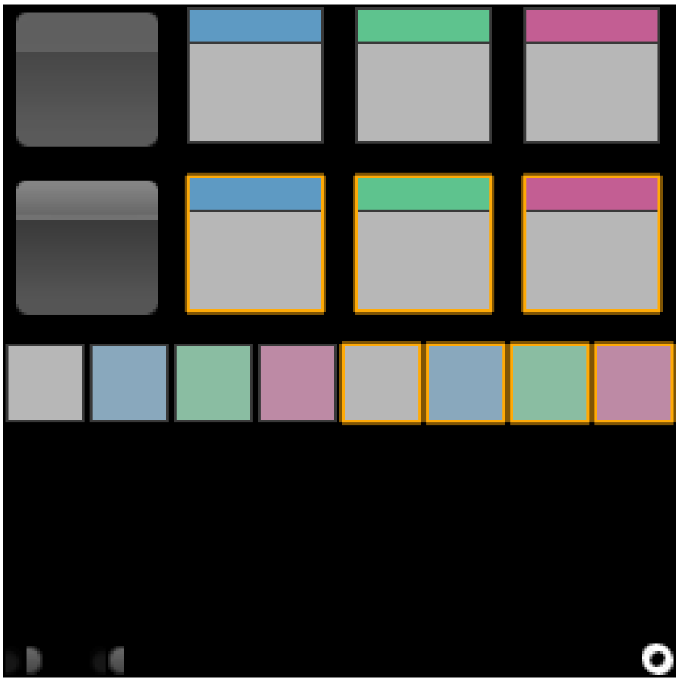


Figure 6.2: *The GUI Texture used for drawing the nodes*

In this project we only use the two gray boxes which are located in the top left corner, over and under each other. The lower of the two are used when a node is selected, while the top left box is used for unselected nodes. The rest of the boxes were not used in the work of this thesis. The images used for the connector slots are located in the bottom left corner. There are different images for when the slot has a connection or not. When the slots are drawn on top of the node, they are modulated with the color that match their type, so it is easier for the user to make type legal connections.

We have implemented support for rendering the nodes in three different sizes, as described in the previous chapter. We added an enumeration to the node class, which determines how to render the node. In the DrawGraph class, we simply use this information when rendering the node.

The Bezier curves which represent connections between the slots, were rendered using the Bezier class provided by the engine. The class initially only supported evaluating points on the curve, so we updated it to support rendering the curve, by drawing small concatenated quads along the curve. We chose to use a Bezier curve instead of a line primarily for ascetic reasons. The color of the curve were set to the same color as the slot the curve connects from.

Finally we also needed a system for rendering text in our GUI system. We use this system for displaying the node and slot names, along with the shader graph path in the top left corner of the view. The text rendering system Unity had was not so suitable, as the text appeared blurry. We therefore made a new importer, which is able to import true type fonts into Unity. The importer uses the Freetype library, and stores information about the glyphs and kerning in a font object. It then renders the individual fonts to a texture, which is used for drawing the text on the screen. Storing the glyphs in a texture were a simple way to get the text rendering into Unity, but it will yield some problems with languages such as Chinese and Japanese, which use many individual signs. This was not a major concern for this project, and if Unity's font rendering is updated in the future, it will be straight forward to support those languages in the shader graph GUI.

Besides rendering GUI elements, we have also implemented functions for handling the users mouse input. When the user clicks on a node, the function iterates over all the nodes, and returns an index which says what the user clicked in the view. Based on this index we find a reference to the object (node, slot etc.) that were clicked, and perform the appropriate function that the user selected. The user can right click to get a menu with possible commands. This menu has been implemented using the Cocoa window interface, as there were not enough time to make a nice platform independent implementation. We also added support for the classic hot-keys for delete and duplicate, which is based on events send by the operating system. If the whole GUI system needs to be made platform independent some day, this right click menu and the operating system events would need to be handled in a more generic way. The code which handles most of the GUI system is in the ShaderGraphPane and DrawGraph classes. The DrawGraph class uses the GUI class for rendering.

6.2 Graph Implementation

The main functionality of the graph system is implemented in the ShaderGraph class. The class has an array which holds the individual nodes, and has functions for adding and deleting nodes from this array. The connection information is stored in the connector slots, which is implemented by another class. The connection slots always exist in a shader graph node, and thus the three classes that makes up the shader graph system is the ShaderGraph, the ShaderGraphNode and the Connector classes. The Utility class is also used though, as it has the flood filling function etc. This section will discuss how these three classes works together, to implement the connectivity and functionality of the actual shader graph system.

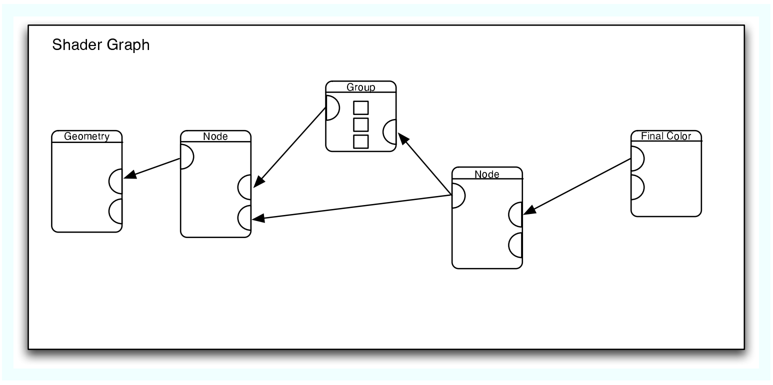


Figure 6.3: *This figure illustrates how the nodes in the shader graph is connected. The arrows indicate the direction of the connection in the DAG. A close-up of the group node can be seen in figure 6.5.*

Figure 6.3 illustrates the connectivity of the shader graph system. The nodes of the system are placed in the shader graph object, except for grouped nodes, which are placed inside the group node. Each node has a number of connection slots that the user use to connect variables in the system, thereby setting up the functionality of the shader. Early on we decided to only store these connections in one direction, going from an input slot to an output slot. This decision was made as it seemed natural that an input variable only can be assigned to one unique variable, and hence only has one unique connection. To better understand that, consider that those variables are on the left hand side of an assignment operation in the generated code: $NodeInputSlot = NodeOutputSlot$.

This means that it only makes sense to have one connection from an input slot, while an output slot can easily be connected to multiple input slots. Of cause this can give problems with supporting swizzeling, as that would require multiple connections to a single input slot, but we chose to solve that problem by having swizzle nodes instead. One of our main concerns were to keep the implementation of the graph system as simple as possible, which also speaks for this simple one way connection system. Of cause it would have been possible to store connections in the other direction too, possibly by having an array of connections for each output slot. This could require us to differentiate between input and output slots though, as input slots should still only have one connection. It would also mean that we had to store and maintain twice the amount of serialized data, which could hurt performance. On the other side, having access to the connection information in one direction only, would make flood-filling and other operations which also move "forward" in the graph more complicated. See figure 6.4 for an illustration of the "forward" term. But as we will not have many of these operations, we decided to go for the simple connection scheme.

In each input slot the connection is stored by having a reference to both the node and the slot that is being connected to. We need information about both, because the slots does not know which node they are a part of.

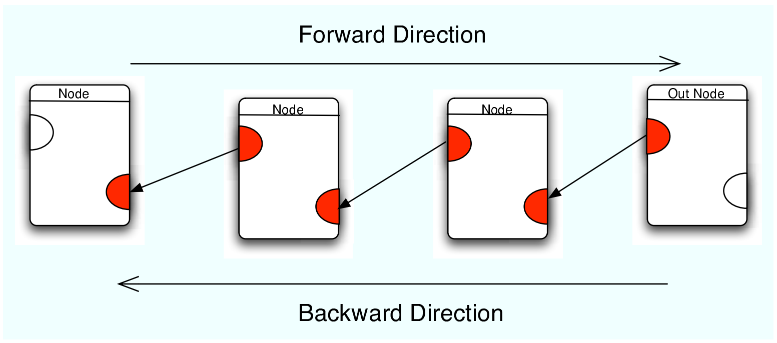


Figure 6.4: *In this thesis we call the left to right direction for the forward direction in the graph.*

To summarize on the above, the shader graph is represented using a Directed Asyclic Graph, and the connections are only stored in one direction. This means that many of the algorithms we had to implement, were recursive algorithms. An example of a recursive algorithm implemented, is the loop finding algorithm 1.

The loop detection algorithm is being called whenever a connection is made

Algorithm 1 Loop Detection Algorithm

```
if current node has been checked then
    return false
end if
mark current node as visited
bool hasLoop = false
for each input slot in current node do
    if slot has connection then
        hasLoop = hasLoop or has loop on connecting node
    end if
end for
unmark current node as visited
return hasLoop
```

between two slots. The algorithm works by detecting if a node in a subgraph, in any way connects to itself, in which case true is returned. Another type of recursive functions that proved a little more challenging, was the function for flood-filling generic type slots. We wanted this function to be as generic as possible, so it could handle large subgraphs with only generic type nodes, and flood fill all the generic slots when a connection were made to one of the nodes. As a connection can just as easily be made to a node in the middle of the subgraph, we would have to be able to flood fill in both directions in the graph. This caused a problem, as we only stored connectivity in one direction. We solved this by iterating over all the input slots in all the nodes, every time we wished to examine if a output slot has a connection. We would then check for connectivity between the output node, and each of the input nodes in the iteration. This might seem as an unoptimized way of doing the flood fill, but remember that high performance is not so important for the GUI interface of the shader graph, as long as it stays interactive. The flood fill algorithm is explained in an abstract form in algorithm 2. In the implementation the algorithm is invoked by calling the *FloodFill* function, with the current node and information about which type to flood fill with as arguments.

The function that implements the flood filling, also handles the logical types, which identifies the mathematical space of the variable (world, object etc.). This makes the actual function a little more complicated, but in general it is implemented like algorithm 2 suggests. Many of the other algorithms implemented for this thesis were also recursive, and has been implemented using the same general idea.

Algorithm 2 Flood-Filling Algorithm

```
for each input slot in current node do
    if slot is generic then
        slot type = flood filling type
    end if
    if slot has connection then
        flood fill the node connected to with the same type
    end if
end for
for each output slot in current node do
    if output slot is generic then
        slot type = flood filling type
    end if
    find the input slots that connects to this slot
    for each found input slot do
        if slot is generic then
            flood fill the node of this input slot with the same type
        end if
    end for
end for
```

6.2.1 Node Implementation

As previously discussed, we wanted to focus on creating a simple interface for the nodes. The implementation reflects this design, by using a parent node class called ShaderGraphNode, which has all the functionalities that are common between most nodes. The most important functionalities of the parent class are:

- Array lists for holding the input and output slots.
- Strings variables for holding the Cg code.
- Type matching function, that returns true if two types are compatible.
- Functions for setting up published connectors and creating preview shaders.
- GUI related functions such as finding the height of a node etc.

These functionalities has been pretty straight forward to implement, and as it is something all nodes needs to have, it seemed the smartest to encapsulate them in a parent class. In order to create a new node for the system, one only has to implement implement the following functionalities:

- Create input and output connection slots, and add them to the nodes array lists.
- Set mathematical space types for the slots, if they should be different from the default, which is no space defined.
- Implement the *SetCodeStrings*() function, which sets up the code string to perform the operation desired for the node.

The *SetCodeStrings*() function is the only demanding task when making a new node. The function should generate the code that will be accessed by the output slots, by using the connection information of the input slots and the Cg code strings relevant for that functionality. An example could be the "Add" node, which should fetch the two variables connected to in the input connectors, put a " + " between their variable names, and store that result in the output connector. In the case where a input slots does not have a connection, the default value of the slot is used, in order to ensure a valid output.

When more advanced nodes needs to be made, the overhead is a little larger though. The basics are the same, but more advanced nodes might require different code paths depending on which slots that has connections, such as our material nodes does. In the material nodes we have two different code paths, depending on whether the normal slot is connected to a tangent space normal, such as a normal map node. In that case we handle the lighting calculations in tangent space in another path, instead of using our automatic space transformation system. This is purely because of performance though, we could have used our automatic space transformer system as well. The material node also needs to transfer viewing and lighting vectors internally. In order to make that work with our automatic scheme for building the transfer structure, we added a flag to the connection slots, that made the struct builder function handle those variables too. The connection slots and compiler implementation will be discussed in the next sections.

The one node that differs the most is the group node, which is actually more like a separate functionality than an actual node. We chose to implement it as a node though, because we wanted to show it like a node, so it were convenient to inherit from the parent class to get the drawing automatically. Unlike the normal nodes, the group node has been implemented to have its own array of nodes. When a selection of nodes are being grouped, we then move the relevant nodes from the shader graphs array to the nodes array. We also create slots in the group node, which are identical to the slots that have connections to or out of the selection. In that way the grouped nodes have the same connectivity through the group node, as they had when they were individual nodes. One problem remains though. As the group node contains multiple nodes, it is difficult to create a single function that creates the code string for all the nodes. We chose to solve this problem by not doing this operation in the group node, but simply relying on the functions in the nodes themselves. This means that the group node just becomes a container for other nodes, and should not be handles as a normal node when performing operations on the graph. We have solved this by adding connections to the output slots in the group node, which point towards the corresponding output slot and node inside the group. The functions that depend on the connectivity of the graph, were then updated to use then connection information when encountering a group node, which makes the functions operate like they would on a flat unfolded graph. The connectivity of the group nodes is illustrated in figure 6.5.

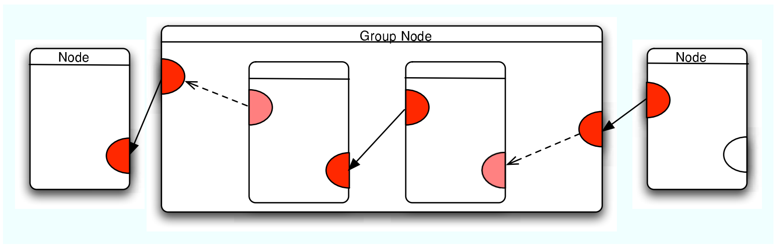


Figure 6.5: *The connectivity of a group node. Notice the additional added connections which goes from the group nodes output slot to the output slot of the internal slot that has this slot. A dashed line means that the connection curve should not be visible. Grayed out slots are published, and can only be accessed via the group node.*

6.2.2 Connection Slot Implementation

Another important component of the shader graph are the connection slots, which are the users only interface for connections in the graph. As previously discussed, we only store connections from input to output nodes, because the input nodes only can have one connection. This is the only difference between input and output slots though, so we decided to use the same class for both types, and just not use the connection information in the output slots, except for group nodes, as discussed above. A connection is implemented by using two variables in the slot, one for the node being connected to, and one for the name of the slot connected to. We decided to do the connections in this way, as we more often would need access to the node connected to, than to the slot directly. In the cases where the specific slot needs to be found, we use a property which loop through the slots of the node, and returns the array index of the named node. The only real issue with this approach, is that it is then not possible to have two slots with the same name within a node. That should probably have been avoided anyway, as that can appear confusing to the users.

Besides the connection information, a slot also has the following important variables:

Published: Slots published in the main shader graph are presented in the material inspector for fast editing (Only floats and colors). Slots published from a node inside a group node, is shown in the group node.

Name: The name of the slot. Slot names must differ within a single node.

Default Value: The default value of the represented variable. Such a value must be set for input slots, but it has no effect for output slots. The default value is implemented as a string.

Type: The type of the variable the slots represents (generic, float, vector3, color etc.).

Logical Type: The space this variable is defined in (object, world, tangent, eye etc.).

Is Generic: Flag that says if this is a generic type, which means that it can be flood filled. Similar flag exist for the logical type.

The connection slots are always contained in the input or output slot lists in a node. They are normally created and handled by the node too, except for the case where additional slots are published to a group node, in which case the system creates the new connector and places it in the group node. If users wishes to create their own nodes, they should familiarize themselves with the variables listed above, as they will be important for make the slots of the node work correctly.

6.3 Compiler Implementation

Early on in the compiler implementation stage, we experimented with building a new compiler tree structure, which had connection information in both directions, and which flattened the group nodes, so the compiler would work on one big shader graph. This proved to be a bad idea though, as that required building a new structure of serialized data, which is a bad idea as it can make the system very unstable. The unstability occurs if the original graph and new generated structure become inconsistent. Besides that it is a waste of space to work on an new representation of the same data. It also seemed to complicate the implementation of the compiler to an unnecessary degree. We therefore came up with another method, where we worked directly on the unmodified shader graph that had been created. We found it to be surprisingly easy to work on this structure, as we could simply use recursive functions to traverse through the graph, and concatenate the shader code strings in the nodes, beginning with the nodes lowest in the graph. This ensures that the code is added in the correct order, so code in early nodes are put in the shader first, so the later nodes can work on the result of those previous nodes. As it is possible for one node to be visited more than once though, we had to maintain a flag to know if the node had already been processed, as nodes should only provide code to the shader one time. With that in place, the problem of actually building the shader string, was reduced to concatenating the code strings of the nodes in a recursive manner. Before that could be done though, the compiler has to go through the following preprocessing steps:

- Make the variable names unique.
- Set nodes generic code to be vertex or fragment code.
- Process published slots.
- Build the vertex to fragment structure.
- Setup the code strings in the nodes.

We first need to make the variable names unique, as multiple variables with the same names would cause errors when compiling with the Cg compiler. We uniquefy the variables names by iterating over all the slots in the graph, and maintain a list of already used names. When a duplication between two variable names is encountered, we append and "I" to the variable name, thereby making it unique. Next we iterate over all the nodes in a recursive manner, and examine if the node should generate vertex or fragment code for its generic code strings. We then examine if any slot is published, in which case we add it to the properties of the shader, which makes it visible in the material editor. The next step is to build the vertex to fragment structure, by identifying the connections that goes from a fragment program variable and to a vertex program variable. When that happens, we pack that variable into the structure, as described in chapter 5. Finally we invoke each nodes function for setting up the code strings, and concatenate the code strings which gives the final shader.

6.4 Game Engine Integration

Implementing the game engine integration were primarily an question about getting the shader graph into Unity, and also to implement the new compilation system discussed in the previous chapter. Integrating the shader graph with Unity were surprisingly easy. An OpenGL viewport named shader graph were implemented by the Unity staff, and in collaboration with the author, the material system were updated so a material could contain a shader graph instead of a normal shader file. As the user work with the shader graph, the relevant data such as nodes, slots and so on are serialized to the disk and saved automatically. This is a benefit of the engine integration, and it means that the graph is always kept saved and updated in the project. Inside the OpenGL viewport the shader graph were rendered, using the previously described GUI system, and scripts were attached which implemented the features of the system. With this in place, the only problem that remained were to implement the new processing system, and do some slight modifications to the binding of shaders.

The design idea for the keyword compilation system, indicated that we should use the Cg Framework to handle the compilation of the vertex and fragment programs. Unfortunately though, the Cg framework crashed frequently on the OS X platform, so we were unable to use it. The Cg compiler worked though, and we was able to compile the programs using that. This meant that we were unable to use several nice framework functions, for compilation and binding of the programs, but we were still able to compile the Cg code though. The compilation were implemented by cutting the Cg code out of the shader, and do a command line compilation of the code into the standard ARB_VERTEX_PROGRAM or ARB_FRAGMENT_PROGRAM assembly code. We used the comment lines from the compiler output to find texture bindings and setup variable bindings too. The whole process were then repeated for each combination of keywords, and the resulting compiled programs were put back into the shader file. When the shader files were processed to have the assembly level code instead of the high-level Cg code, Unity will do the binding of the shaders using standard vertex and fragment program binding functions. We did update the binding too though, as the binding scheme should now use the keywords from the shader too. So when processing a shader file, we stored the keyword information in the shader object. When finding which shader to bind, we checked which keywords were currently active, and iterated over the keyword combinations in the shader to find the one that matched best. This version of the vertex and fragment programs would then be bound. This meant that we had to update the lighting source code too, so it would set the keyword corresponding to the light type, in order to make this work. If future effects require the use of keywords, it is important to make sure the keywords are set when the shader using them are expected to run. That can be done by modifying the engine source code as we did, or by using the *SetKeyword*() function that we implemented during our integration.

There are two main problems with using the Cg command line compiler instead of the framework though. First of all if the commenting style changes in a later version of Cg, the implemented code would have to be updated to reflect those changes. A even more annoying problem is that the command line compiler is very slow to use. It takes about half a second just to start it, and we start the compiler for each keyword combination compiled, and for each vertex and fragment program in those combinations. So when compiling a shader with and ambient pass (no keywords) and the lighting pass (3 keywords) that would lead to starting the compiler 8 times, which cost around 3-4 seconds. That means that we did not have the opportunity to automatically compile the shader graph every time a change occurred, as the system would become very un-interactive to work with. A possible solution to the problem by implementing a compilation console was discussed, but there were not enough time to implement it for this project. Another solution would be to skip Cg and go with GLSL instead, but that would also be a problem as that will break backwards compatibility.

Chapter 7

Implementing Shaders in Other Systems

Since the introduction of the Renderman shading language, the creation of shaders has been a very important aspect of computer graphics. In this thesis we discuss a system for shader creation, which features integration with a game engine for real-time rendering, and where usability is one of the key issues. There are several alternative approaches to shader creation though, e.g. using the original Renderman interface and shading language. This chapter will discuss the creation of a simple shader with both Renderman, RenderMonkey and Maya, for the purpose of comparing these products with the one developed in this thesis. In the next chapter the same shader will be created using our implementation, and chapter 9 discusses the comparison.

The shader we use for comparison is very similar for all the creation methods discusses in this chapter, and it will also be created using the Shader Graph editor implemented in this thesis in the next chapter. The only real difference is that we use build-in functions in Renderman, which calculates the lighting based on the original Phong lighting model. The other shaders are created using the slightly simpler Blinn-Phong model, which is more common in real-time applications. Another difference is that Renderman does not support real-time rendering, but this is not really an issue here, as we just want to discuss how to create shaders in Renderman, and as such we do not care about the rendering speed at this point.

We wanted to keep the shader relatively simple, but still advanced enough to be able to discuss the differences of the products. We therefore decided to implement a bump mapping shader, which creates a rough appearance of an object. We further more gave the shader the twist that it should use two different colors, one from the main texture, and an additional color which should be controlled by the alpha channel of the main texture. This means that where the alpha channel is white, the original texture should shine through, and where the alpha channel is black the color should be modulated onto the texture. This shader was actually a request by an user of Unity, who were not able to program the shader himself. We therefore feel that this shader serves as a good example for comparing the different creation methods, and finally show how this user could have made it in Unity, had the shader graph been available.

7.1 Creating Shaders in Renderman

Even though the Renderman shading language were among the first languages of this kind, it is still arguably the most advanced systems today. The general concepts of Renderman were discussed in chapter 2, and we discussed the different shader types that Renderman operates with. In order to implement the effect discussed above, we had to use four different shaders that we will discuss in the following.

Light Shaders:

> We used two different light source shaders when creating this effect, namely the ambient light shader and the distant light shader. The ambient light shader is the simplest of the two, and it just sets the light color to the intensity defined by the user, or one in the default case. This adds a constant amount of ambient light to the scene. This is then combined with the light created by the distant light shader, which simulates a directional light source. The $solar()$ function call is used in the distance shader, to setup a light-source which casts light in a given direction, and the shader then sets the light color in that given direction. The lighting shaders are automatically used to attenuate the light, when the lighting calculations are performed in the surface shader.

Displacement Shader:

> The displacement shader uses a height map to displace the pixels before the lighting calculations are done. On the contrary to the approach taken by real-time shading where only the normal is altered, the displacement shader actually moves the pixels, and then recalculates the normals based on the new position. The moving of the pixels is important, for Renderman's ability to perform correct hidden surface removal and shadowing on the displaced scene. In this effect it is most important that the normals are updated though, as the perturbed normals will greatly influence the lighting of the object and make it look bumpy.

Surface Shader:

> In the surface shader we find the texture color, and do a linear interpolation between this color and a second color based on the alpha value in the texture. We then use build-in functions to calculate the diffuse, specular and ambient light contributions, which are multiplied with their respective material constants in accordance with Phong's lighting model. We then add these contributions together to get the final lighting contribution. It should be noticed that we do not need to sample a bump map, as the normals calculated in the displacement shader, now matches those that would be found in the bump map.

The source code for the four shaders can be found in appendix B.2, and the result of the rendering can be seen in figure 7.1.We used the freely available renderer called Pixie [4], which is a Renderman compatible renderer. As Renderman is an interface rather than an actual application, no graphical user interface or other code abstracting systems exist, which means that creating shaders with the Renderman shading language is limited to programmers only. The implementation of the effect in Renderman is slightly different from shading languages such as Cg, as it requires multiple shaders to be created. Renderman uses the light-source shaders to calculate the attenuation values of the different light sources in a shader, and a displacement shader for actually displacing the surface, which differs from normal Cg shaders where everything is done in a single vertex and fragment program. Further more Renderman has predefined names for the normal, light and viewing vectors and so on. This means that it requires a quite good understanding of the Renderman shading language to

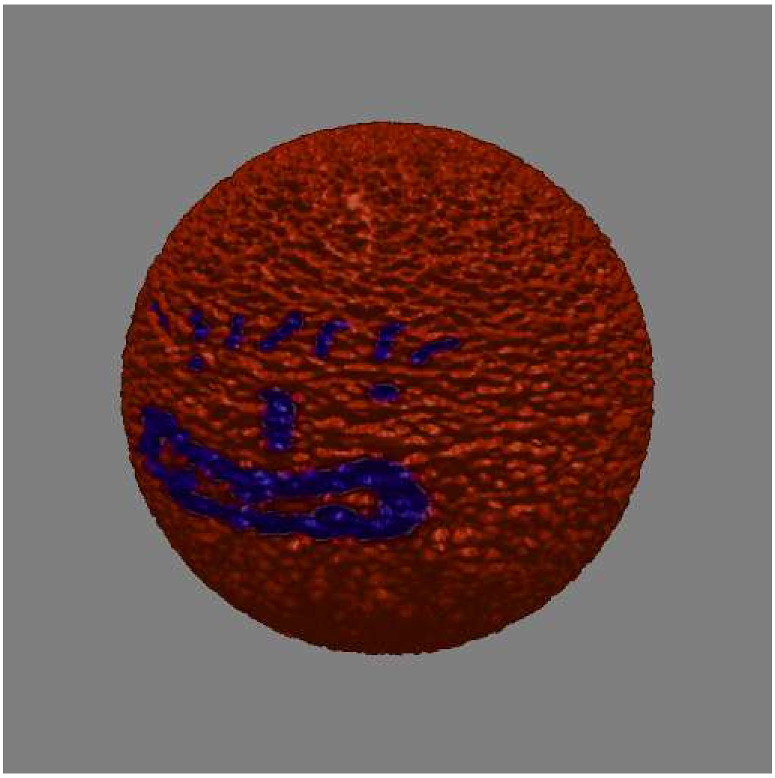


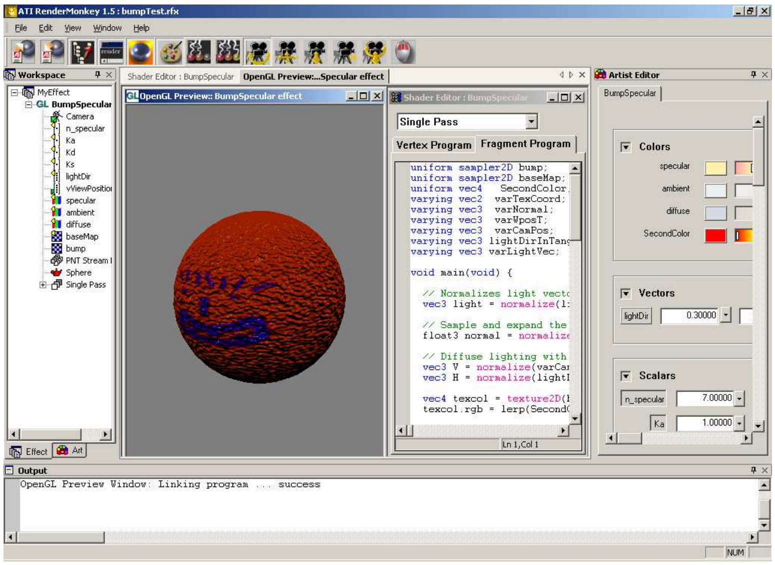
Figure 7.1: *Result from rendering the bumped sphere with Renderman.*

implement special effects, as the programmer would have to be familiar with these names, and how to use the different shader types.

In order to apply the shaders created, one has to make a so called RIB file, which is an abbreviation for Renderman Interface Bytestream. This file is used to setup the scene, which means setting up the camera, objects, surfaces and lighting and so on. The RIB file used to setup the scene, can also be found in appendix B.2.

7.2 Creating Shaders in RenderMonkey

Figure 7.2 demonstrates shader creation in RenderMonkey. The left viewport is an inspector which can be used to setup variables such as textures and colors,



```
uniform sampler2D bump;
uniform sampler2D baseMap;
uniform vec4    SecondColor;
varying vec2  varTexCoord;
varying vec3  varNormal;
varying vec3  varWposT;
varying vec3  varCamPos;
varying vec3 lightDirInTang
varying vec3 varLightVec;

void main(void) {

  // Normalizes light vecto
  vec3 light = normalize(l:

  // Sample and expand the
  float3 normal = normalize

  // Diffuse lighting with
  vec3 V = normalize(varCar
  vec3 H = normalize(lightI

  vec4 texcol = texture2D(l
  texcol.rgb = lerp(Second(
```

lightDir 0.30000
n_specular 7.00000
Ka 1.00000

```
OpenGL Preview Window: Linking program ... success
```

Figure 7.2: *Screenshot from RenderMonkey that shows how the shader were created, using ATI's shader IDE.*

which can then be used in the shader. In the screenshot the inspector shows the unrolled effect named "BumpedSpecular", which is the created effect. It has a single pass, which specifies the second color used to modulate the texture with. Inside the pass we also find the vertex and fragment program used to implement the effect. Using the inspector it is also possible to setup the rendering state for each individual pass, which allows effects such as transparency due to blending etc. The next window to the right is the preview window. It shows the active effect applied to a model, using a directional light which can be setup with a variable in the inspector view. The artist view to the far right is similar to the inspector, but it only allows for changing variables such as colors, vectors and scalars, which are relevant for the shading effect. The artist view is meant as a help for non programmers to tweak the appearance of the effect, by adjusting variables used in the shader, and not having to deal with the more programming oriented view of the inspector. The artist view is only a small help for tweaking existing effects though, it is still only possible to create shaders with RenderMonkey for programmers with shader programming experience.

In the middle to the right we have the code view, where it is possible to edit the shader code of the vertex and fragment program. In this particular example the glsl shading language were used to implement the effect. The vertex and fragment programs implement the Blinn-Phong lighting model discussed in chapter 3, using the normal from the bump map and performing the lighting calculations in tangent space. Tangent space calculations are necessary, because the normal from the bump map is defined in this space. In the vertex program we calculate the viewing and lighting vectors, and rotate them into the tangent space. The fragment program then calculates the lighting contribution, and multiplies this with the texture color, which has been modulated with the second color based on the textures alpha component. This creates the effect seen in the preview. The source code for the shaders can be found in appendix B.3.

7.3 Creating Shaders in Maya

Creating shaders in Maya using Hypershade is very similar to the approach discussed in this thesis. Hypershade is Mayas own shader graph editor, and it is also one of the first graphical shader creation tools demonstrated in the industry. We created the shader by setting up the graph in Hypershade, as seen in figure 7.3.

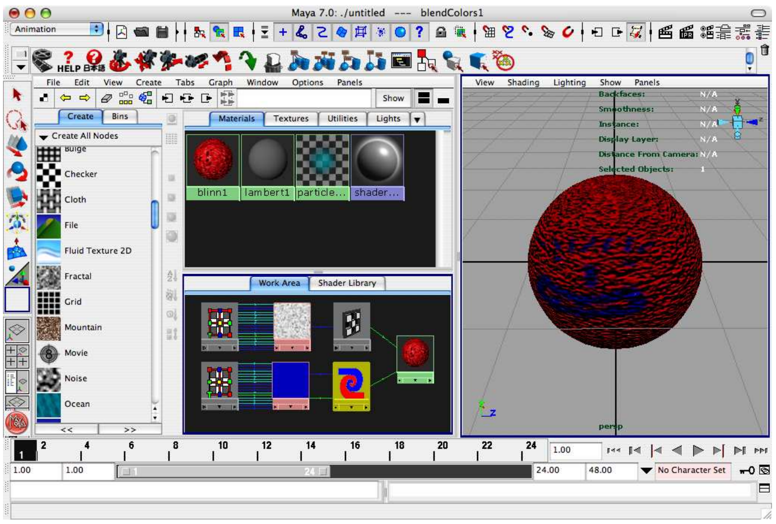


Figure 7.3: *Shader creation using Hypershade in Maya. The left view shows the available nodes, the middle-bottom view shows the created shader graph, and the right scene view shows the shader applied to the spaceship.*

The first node we put in the graph were the Blinn material node, which calculates Blinn-Phong lighting as discussed in chapter 3. We then created two texture nodes, one which held the main texture and one for the height map used for bump mapping. In Maya one uses a height map and a bump map node instead of a pre-calculated bump-map texture. This might be due to tangent space issues for bump maps. The output from the Bump node were connected to the normal of the Blinn node. The color and alpha output from the main texture were then connected to a color blending node, attaching the color to the first input color, and the alpha value to the interpolation slot. The second color were then set to a blue color, and the output of the color blending node were connected to the diffuse color of the Blinn node. The figure illustrates how the material renders when applied to a sphere, which is being illuminated by a directional light.

Chapter 8

Results

The main product of this thesis is the shader graph editing tool, which makes the creation of custom shader effects more accessible. This chapter will show how the tool can be used to create custom material effects, and the features implemented in the system will be presented. Besides the screenshots presented here, the accompanying CD-ROM has a number of videos that demonstrates how shaders are created using the shader graph editor. We also strongly recomend looking at the high resolution images found on the CD, as it should be easier to understand the figures from them. The images has been given the same name as the figures, so they are easy to identify. More information about the CD can be found in appendix C.

8.1 Shader Graph Editor

Figure 8.1 shows how the shader graph can be used to create a basic Blinn-Phong shader. The alpha channel of the texture is connected to the glossy slot of the Blinn-Phong node, which means that the value from the textures alpha channel is used to modulate the specular contribution, giving glossy specular reflections. In this example the Blinn-Phong and the Texture nodes has been set to display their preview fields. The preview field of a node is a feature we have implemented, to help the user understand what happens in the individual nodes.

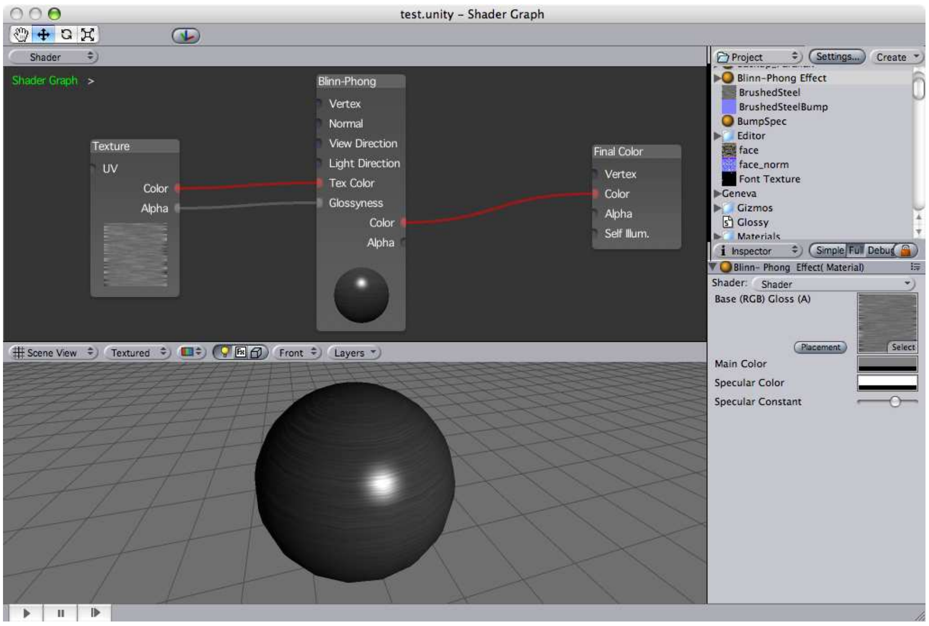


Figure 8.1: *Basic textured Blinn-Phong shader.*

For texture nodes the preview field just displays the texture, which is important if there are many textures, and the user needs to pick the correct one in a given situation. For other node types, the preview field shows a rendering of a sphere illuminated by a directional light, using the subgraph of the node to shade the sphere. Being able to see a preview in each node[1], helps the user identify where in the shader graph something were connected wrong, which could prove to be an important debugging feature. While most of the previous industry work has this feature, most of the previous academic work discussed in chapter 2 did not have this feature. In some of the screenshots shown in this chapter, the preview field has been disabled for the nodes, in order to save space in the graph view. It would have been possible to use the preview of those nodes too though, and indeed the preview were often used during creation of the graphs.

The effect from figure 8.1 is a pretty standard effect, which most game engines has build-in. So let us consider a slightly more sophisticated effect, which will show the shader graph tools ability to quickly modify and create new shaders. Figure 8.2 shows a shader graph which handles an extra game related color, and shades the object using Blinn-Phong shading. This is the same shader as the

[1]Some of the purely geometrical nodes does not have previews

one we demonstrated in the previous chapter.

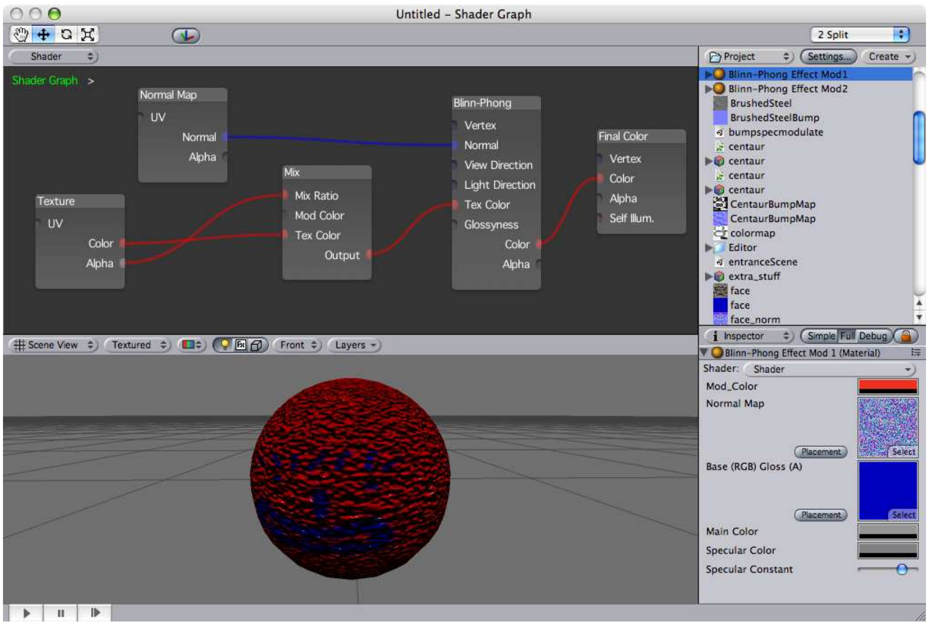


Figure 8.2: *Blinn-Phong shader modified to have an extra color. Same textures and light setup as in the examples in previous chapter.*

In the users case, he wanted to render a space ship using this shader, and he needed the additional color to set the color of the player. In figure 8.3 we show the difference in the result the user were able to obtain without the shader graph, and how simple it would have been to obtain the correct result with the shader graph.

The effect were initially discussed on the Unity forum [35]. The user wished to create a bump-map shader that could color specific parts of his spaceship, based on the alpha channel of the texture. He did not know how to program shaders though, and therefore he sought help from the community forum, as his initial result (left ship in figure 8.3) were not satisfying. It is very likely though, that he would have been able to create the shader as done in figure 8.3, by applying a mix node [2] to mix between the player and main color based on the texture alpha. In order to manipulate the player color slot, it were published using our publishing scheme, which puts published colors or values in the material, for easy editing in the inspector. The graph in figure 8.3 also shown how easy it is to use normal/bump maps using our system. If the normal slot is not assigned

[2]The mix node is equivalent to the lerp command in Cg, and does linear interpolation.

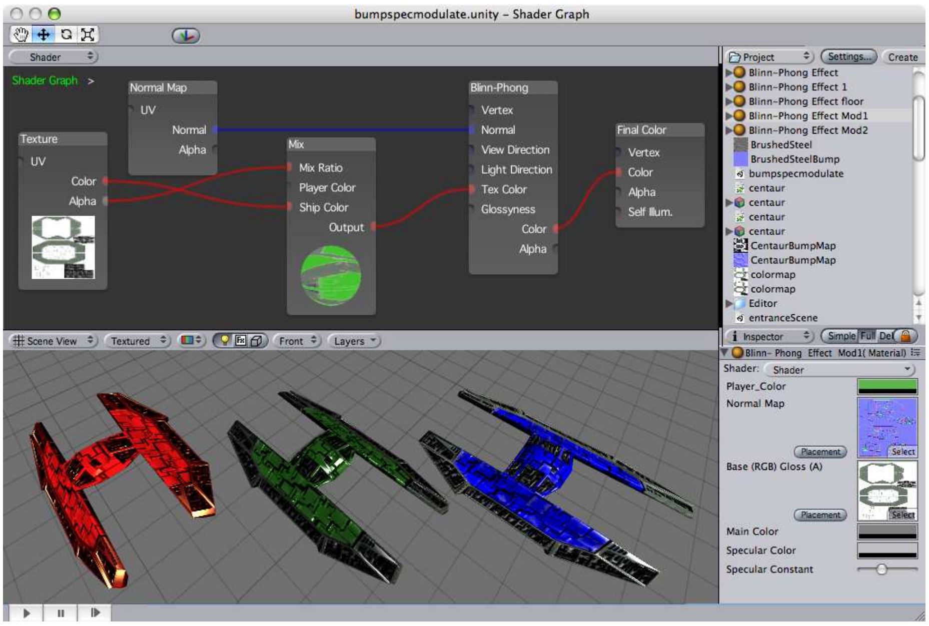


Figure 8.3: *Blinn-Phong shader modified to have an extra game-related color. The leftmost space ship is rendered with the normal build in bump-map shader, and the two right most ships are rendered with the shader presented by the graph, which features the extra player color.*

as in 8.1, then the object spaced normal supplied in the model is used. If the normal map slot is assigned to a normal map, the Blinn-Phong node uses the normal from the map to do normal/bump mapped lighting. The Blinn-Phong node were implemented to perform the required tangent space transformation in the vertex program for optimization purposes, therefore no basis transformation were inserted in this example. In the next example, we will show an effect where the automatic insertion of transformation nodes is used. Figure 8.4 demonstrates a rather advanced shader that creates a silver like material. The shader uses the Blinn-Phong lighting model, but the normals has been fetched from a normal map. The UV coordinates used to find the normal is further more offset according to a height map, which is known as parallax mapping. The shader has also been made to reflect the environment, by sampling an environment map with the reflected viewing vector. The reflected vector has been found by reflecting it on the perturbed normal from the normal map, which ensures that the reflection also appears bumped. This shader operates with three primary spaces, namely object space for the viewing and lighting directions, tangent space for the normal found in the normal map, and the environment map nodes requires the direction vector to be in world space. This gives an chance to see the automatic insertion of transformation nodes in action.

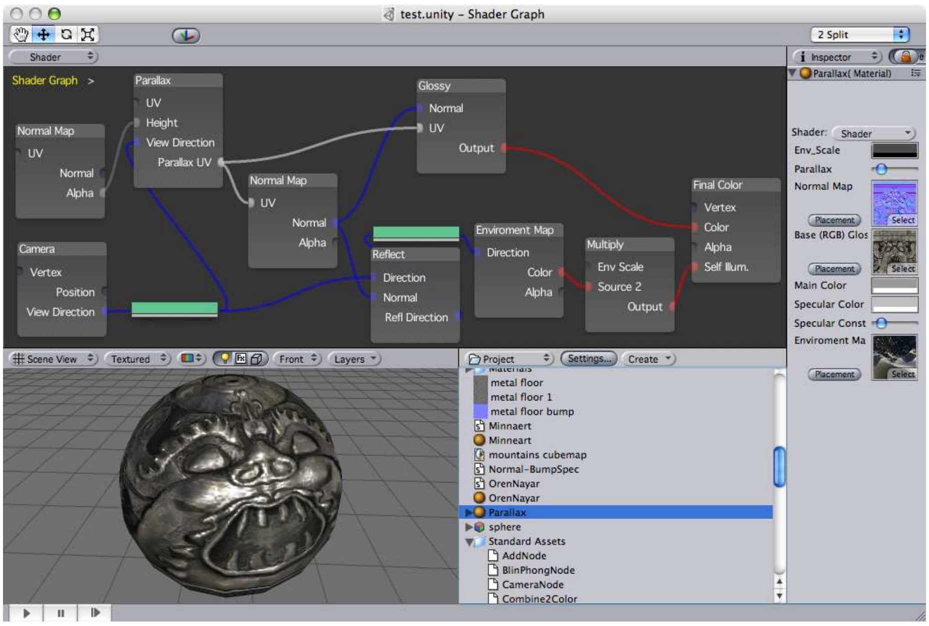


Figure 8.4: *Parallax mapping shader with Blinn-Phong lighting, and reflections in the ambient pass*

The thin green boxes in the parallax shader graph, is the transformers that has automatically been inserted by the system. The leftmost transformer transforms from object to tangent space, while the one to the right transforms the reflection vector from tangent to world space. The left transformer were inserted because this particular reflection node is set to tangent space. This happened because the first connection made to it, the output from the normal map, is defined to be tangent space, which made the whole reflection node tangent space. The environment map needs a world space vector though, which is why the other converter were inserted. Issues with insertion of these transformation nodes are discussed in chapter 9.

Another feature discussed in this thesis is the group nodes, which can be used to encapsulate multiple nodes into one group node. Figure 8.5 shows the content of the group node named Glossy in figure 8.4.

The group node may seem like a trivial thing, nonetheless it were not discussed in the previous academic work presented in chapter 2. The group node represents an important feature, namely the ability to get a far better overview

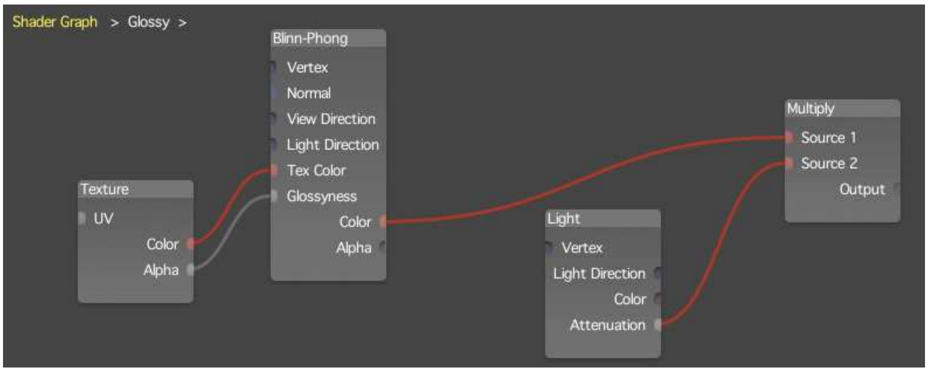


Figure 8.5: *A view inside the glossy node shown in figure 8.4.*

in large shader graphs. The amount of screen space you have available is very important in visually based applications, so if space can be saved on the graph, and the graph further more can have a simplified look, then that is a very important feature to have. The group node works by selecting a number of nodes, and then selecting "Group Selected Nodes". The connections leading into and out from the selected nodes to the rest of the graph, will automatically be setup as connectors in the group node. Should the user wish to expose other slots from the nodes in the group node, so they can be handled from the group node instead, then this can be done by simply right clicking the slot and pressing publish. Another important feature of the group node is that the user can create custom nodes by grouping several nodes with different functionality, like the glossy node shown in figure 8.4 does. Unity has a packaging system which can be used to save these nodes, and share nodes between users, which made it unnecessary to implement this feature in this project in particular.

Until now we have only discussed how the shader graph can be used to create shading effects, which mainly are calculated in the fragment program. The system fully supports modifying the vertices in the vertex program too though, which is something nearly all the other work discussed in chapter 2 did not support. A good example of an useful vertex program, could be a vertex shader which performs the skinning calculations in an animation system. The shader graph editor presented here would be able to perform such an operation, it would require that a relevant node were implemented first though. Unfortunately there were not enough time to create an animation and implement the skinning functions in this project. Instead we illustrates how the vertex shading capabilities can be used to transform the position of the vertices, creating a magnified object.

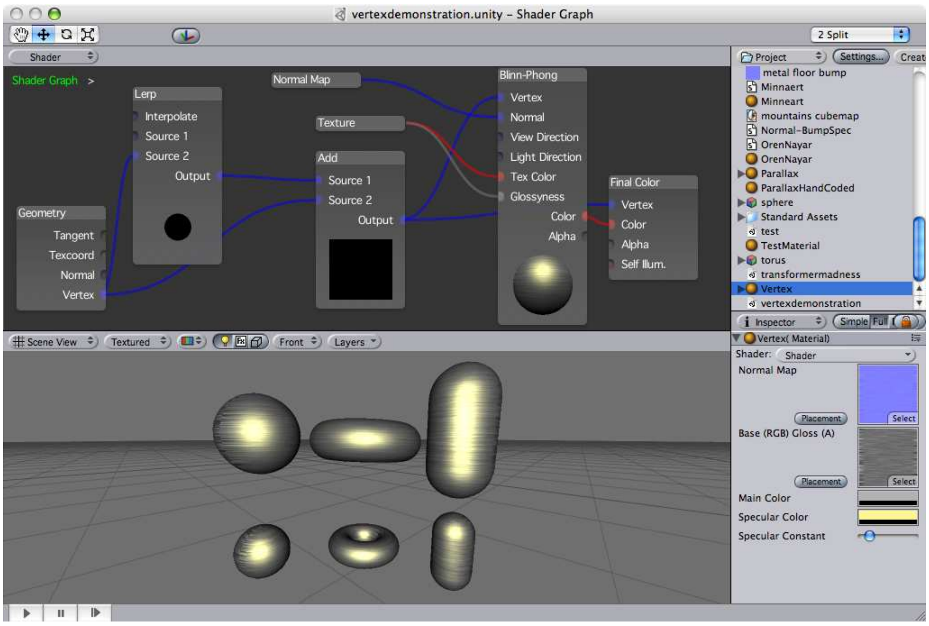


Figure 8.6: *Demonstration of the vertex shading abilities of the shader graph editor. The three objects are (from the left): A sphere, A torus and a cylinder. The vertices are transformed as illustrated in the graph, which causes the modified shape.*

Figure 8.6 presents a shader which magnifies the size of an object by 1.5, and then shades this magnified object with Blinn-Phong shading. The top objects in the scene view is rendered with this shader, while the bottom objects is rendered using a normal Blinn-Phong shader like the one from figure 8.1. The magnification shader interpolates the object space vertex position with the zero vector, with a interpolation value of 0.5 (zero vector and the 0.5 interpolating value are default values). It then adds this result to the vertex position, and uses the new position in the Blinn-Phong shading calculations. The example shows that the shader graph supports custom vertex transformations too, and can use these new vertex positions in the rest of the shader. It also shows how the preview works when altering the vertex position. In the lerp node, the preview sphere is half size of the standard sphere, which is what should be expected. In the add node though, we only see a black box in the preview field. This is because the sphere has becomed too big to fit the preview field, so the sphere covers the whole field. This illustrates the problem that can arise, when performing enlarging transformations on the vertex position, and then trying to fit it in the limited preview field. The problem would be less apparent if the preview rendering automatically adopted a reasonable focusing of the sphere, but then we would still not be able to illustrate the great magnification. In two leftmost previews, the object is rendered black, because these nodes does not output a color value. In the Blinn-Phong nodes preview field, the shading is a one would expect, put the previous transformations are not visible. This is because the Blinn-Phong node does not output a three component vector, so the preview renderer does not know what to use as the vertex position, and thus uses the standard non-transformed position. This could be updated by supporting more customizable preview fields, but that would then propagate the problem of the preview field not showing the whole object, so we felt that it would be better to just show the shading on a single sphere, and disregard the transformation in these cases.

8.2 Integration with Game Engine

One of the most important aspects of this thesis, is the integration of the shader graph editor with a game engine. This integration has allowed us to investigate features such as automatic generation of multiple shaders, depending on which light type that is being used and on other effects such as shadows. The integration has also ensured that an effect file format were used, instead of just a single vertex and fragment program. This has provided the option to render multiple passes with a single shader, such as both ambient and lighting passes. The effect file format also gives the possibility to modify the OpenGL states in the shader, thereby supporting blending, transparency, different culling schemes and so on. This section will present results that we were only able to obtain because of this integration.

Figure 8.7 shows a rendering of the entrance to a building. For this scene we have created three shaders using the shader graph, a glass shader for the glass doors, a floor shader to shade the floor, and the Ward shader shown in the shader graph view. The ward shader uses the Ward node, to calculate anisotropic shading of the brushed steel wall. The figure illustrates how working inside the game engine will allow the user to work on final in-game scenes, which should be very helpful, as things such as light setup and surroundings plays an important role on the shading.

8.2.1 Effect File Capabilities

The glass shader demonstrated in figure 8.8, demonstrates how transparency can be obtained using the shader graph. The shader uses the Blinn-Phong node

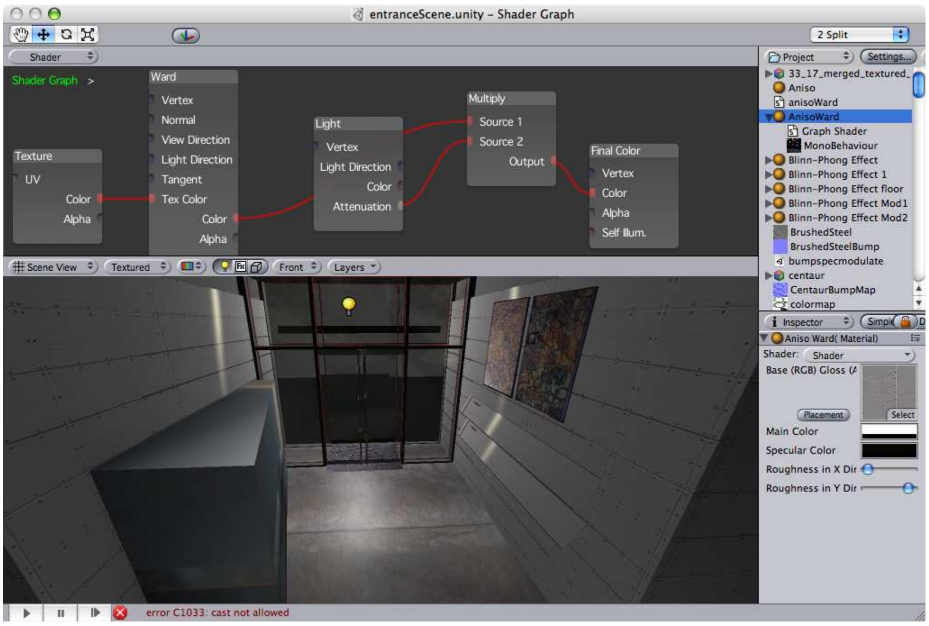
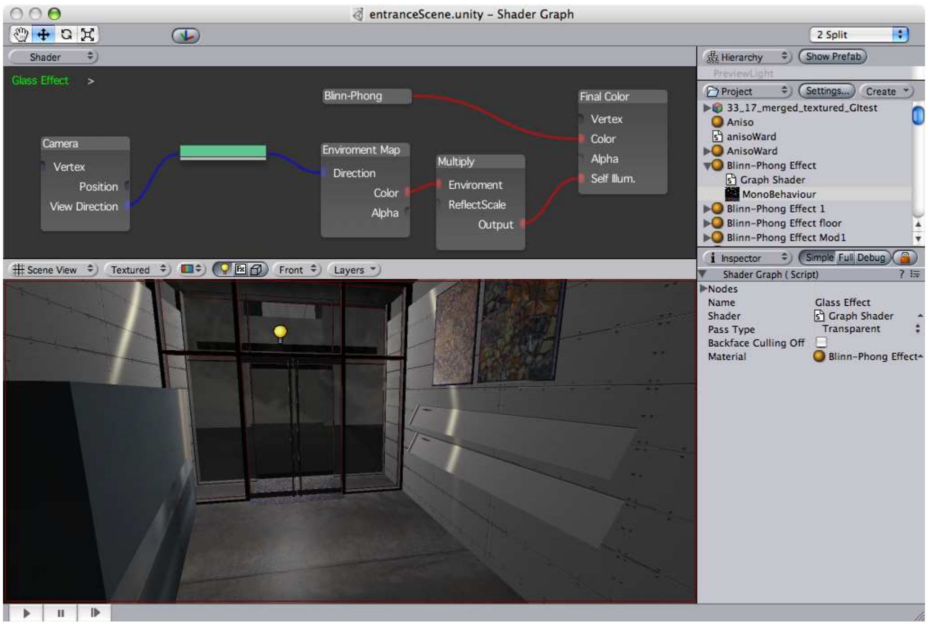


Figure 8.7: *Shader graph editor being used to create the materials of a game scene in a simple way.*

for light calculations (with very dark specular and main color though), and we have added reflections to the ambient pass using an environment map. We then set the pass type for the graph to be transparent, so we can see through the window. This illustrates the ability to setup selected OpenGL rendering states in the shader graph. We can also see that there is a check box for turning culling on/off, which can be used to make the geometry faces visible from both sides.

The environment map used in this scene is the classic mountains cube-map from the ATI SDK, found at ATI's developer pages [22]. This environment does not have much in common with the scene presented here, and a visually far better approach would have been to use a cube map rendering of this scene, had it been available. It would be even more interesting if this cube-map could be updated each frame though, as that would give dynamic reflections in the window, which would greatly enhance the visual experience. An efficient approximation known from the games industry, is to render only one or two faces of the cube-map each frame, which should give good enough quality for the reflections. Currently it is not possible to setup this cube-map rendering using the shader graph. The reason for this is that there is no way to move the camera in the effect file, nor to select a cube-map as the render target. It is not only our system that has this



Name Glass Effect
Shader Graph Shader
Pass Type Transparent
Backface Culling Off
Material Blinn-Phong Effect

Figure 8.8: *The shader graph shown in the top viewport is used to render the transparent window surface in the glass doors. In the inspector view it can be seen how the transparent mode has been chosen, which setup the correct blending state in the effect file.*

deficit, in fact to our knowledge no effect file formats supports this, which means the user has to setup the updating of the cube-map in the engine instead. In the future it could be interesting to experiment with scripting inside the effect file, which would then support operations such as updating a cube-map.

Effect files do support multiple passes though, which is something we have exploited in the shader graph. Shaders created with the graph has two passes as default, an ambient pass and a lighting pass. Nodes connected to the $selfillum$ slot of the final color node, will go into the ambient pass of the shader. An example could be the environment reflections shown in figure 8.8. The ambient pass is necessary in Unity, because without it the object would disappear from the sceneview when it is out of range of any light, due to the culling system.

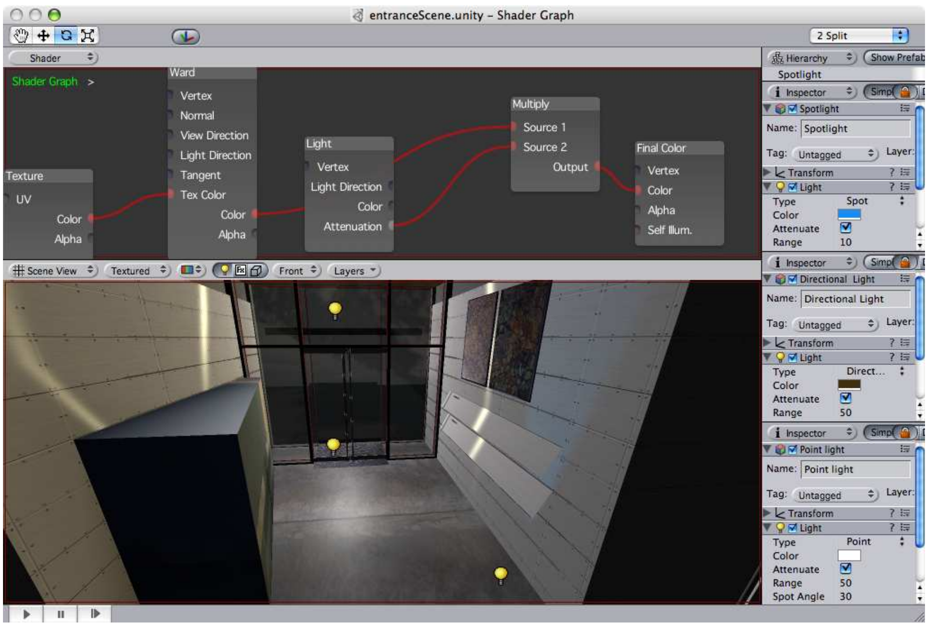
8.2.2 Multiple Light Types in One Shader

In this section we will present the results from the shader processing system. The system is responsible for turning the high level Cg code into assembly level vertex and fragment programs, which are then bound to the graphics card. During this step the system compiles a shader multiple times, based on keywords specified either by the shader programmer, or keywords specified by shader graph nodes. One such node is the Light node, which supports the three classic light types; point lights, spot lights and directional lights. The main difference between these light-types is the way their attenuation values should be calculated. Therefore the Light node specifies the three keywords point, spot and directional, which generates one shaders with multiple different vertex and fragment programs, that supports the three different light types. In figure 8.9 below we demonstrates this, by rendering the entrance scene with one light of each type. The point light by the entrance door is the same as for the other figures. In this scene there further more is an orange colored directional light, to illuminate the left side of the scene. In the right side there is a blue spotlight, which casts light on the right steel wall. It is important to notice that, it is the same shader graph shader that is being used for rendering all three light types, which has automatically been generated to support all three light types based on the keywords.

8.2.3 Integration with Shadows

The keyword system introduced in this thesis, can also be used to support shadows in the created shaders. Unfortunately the shadowing system of the Unity engine is not completed yet, so it were not possible to use it in this thesis. As the integration with shadows is an important argument, we created our own custom shadow implementation, by using shadow rays that are evaluated on the GPU. The shadow rays are intersection tested with the sphere, using ray-sphere intersection as discussed on the siggraph page [29]. The result of the shadow implementation can be seen in figure 8.10.

The shadow information is found in the shadow slot of the light node. The light node has introduced two additional keywords to the shader called "SHADOW" and "NOSHADOW". This causes the processing system to generate additional vertex and fragment programs, which does the shadow calculations. In figure 8.10 we set the SHADOW keyword, which caused the shader to use the shadow information. The keyword can be toggled by using the checkbox named "Cast Shadow" as seen in the inspector. In figure 8.11 the stone floor is rendered using the same shader graph, but with the "NOSHADOW" keyword set, which causes programs without shadow support to be bound and used for rendering.



Spotlight

Name: Spotlight

Tag: Untagged

Type	Spot
Color	🟦
Attenuate	✓
Range	10

Directional Light

Name: Directional Light

Tag: Untagged

Type	Direct...
Color	🟫
Attenuate	✓
Range	50

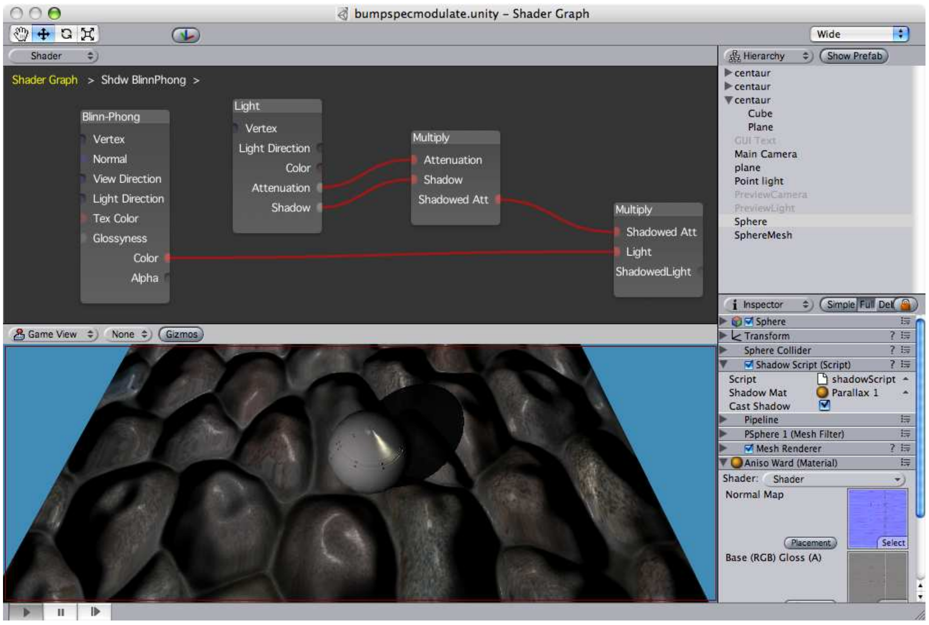
Point light

Name: Point light

Tag: Untagged

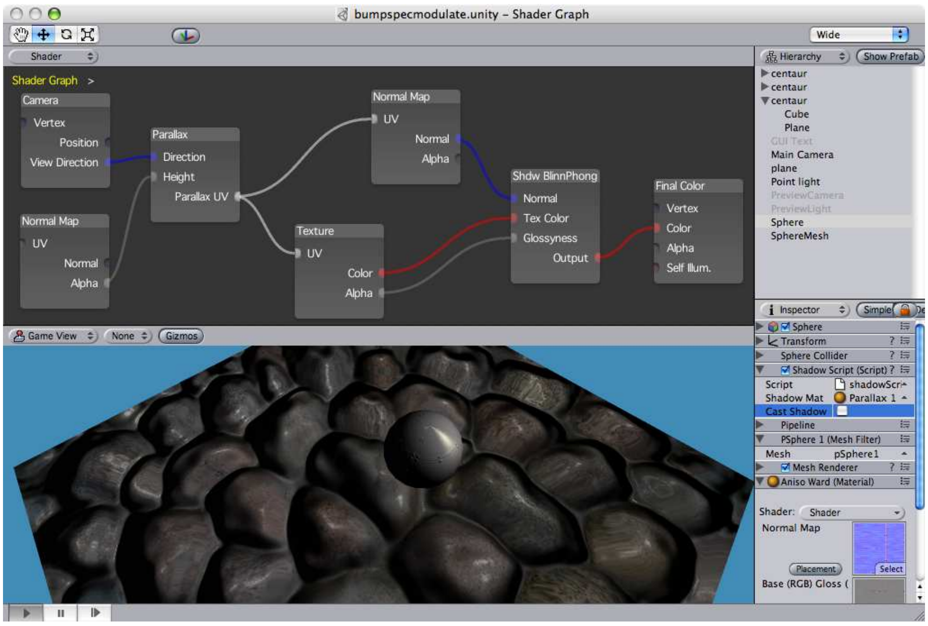
Type	Point
Color	⬜
Attenuate	✓
Range	50
Spot Angle	30

Figure 8.9: *In this scene the graphs created with the shader graph is lit by three different light types, a white point light, a yellow directional light and a blue spotlight. The Ward shader shown in the shader graph view calculates the individual light attenuation from all three light types, using the shader processing system presented in this thesis.*



Script	shadowScript
Shadow Mat	Parallax 1
Cast Shadow	✓
Shader:	Shader
Normal Map	
Base (RGB) Gloss (A)	

Figure 8.10: *Shadow being cast on a parallax mapped stone floor. The sphere is shaded with the anisotropic Ward shader. The sphere is set to cast shadows, which enables the "SHADOW" keyword, so the correct shadow capable vertex and fragment programs are used.*



Script shadowScri
Shadow Mat Parallax 1
Cast Shadow ☐
Mesh pSphere1
Shader: Shader

Figure 8.11: *The sphere is set not to cast shadows, so in this case the "NOSHADOW" keyword is enabled and the versions of the programs without shadows are used.*

Chapter 9

Discussion

This chapter discuss the results obtained with the shader graph editor presented in this thesis. Initially we will discuss how the shader graph can be used to improve the workflow in shader creation. To better argue for this increased workflow, we will compare a shading effect created with the shader graph, to similar effects created with ATI's Rendermonkey and software which uses Pixars Renderman. This comparison will show how fast an shading effect can be created using our system, without requiring any programming. A comparison with hypershade in Maya will also be discussed, which will lead to a discussion about the importance of game engine integration. Further more we will discuss the difference between a hand coded shader, and a shader created with the shader graph tool, with respect to the performance of the shader.

9.1 Comparison With Renderman, RenderMonkey and Maya

Chapters 7 and 8 demonstrated the creation a shader that does bump-mapped lighting, and features an additional game related color, which were modulated onto the object using the alpha value from the texture map. We saw that in order to implement this effect in Renderman, four shaders had to be created, two for the light-source types, one for the surface displacement and one for doing the lighting calculations. While the Renderman approach, with multiple different shader types, does give a large degree of flexibility, it is definitely not the most simple solution. The multiple different shader types and the build in variable names which must be used in the shaders, requires users of Renderman to have a substantial amount of experience with the system, in order to create more advanced effects. Further more Renderman is a programming interface, so only users with programming experience will be able to use it, which leaves out most artists and other types of creative people. So while Renderman remains one of the most advanced systems for creating non real-time shaders, it is also one of the most difficult systems to use, which limits the amount of possible users significantly.

Soon after the introduction of high level programming languages for real-time shader creation, several integrated developer environments came out on the marked, to help shader programmers make their shaders. While these programs does offer some aid, such as easy variable tweaking and rendering state handling, they still require the user to program the shader by hand. In the case of the shader discussed here, that meant writing both a vertex and a fragment program, which implements the bump-map shading effect. When doing this we had to take care of doing the lighting calculations in the same space, which meant rotating the viewing and light vectors into tangent space, as this is the space the bump-map uses for storing the normals. This example illustrates that a programmer not only needs to know the syntax of the shading language, but also needs to understand more advanced topics such as performing lighting calculations in tangent space, in order to implement this effect in Rendermonkey. As the shaders grow more advanced, the programmer will also need to have an even deeper understanding of the underlying graphics hardware and shader programming in general in order to succeed.

The last of the other products presented in chapter 7, were Mayas material editor Hypershade. Shader creation in Hypershade is very similar to creating shaders using our shader graph editor. Both versions use the Blinn-Phong node as the material node, and uses a bump-map (height map in Hypershade) and a texture map, along with an interpolation node which adds the extra color to the scene. When glancing on the Hypershade shader graph, and our version from figure 8.2, the largest difference seems to be that Maya does not have connector slots, instead they give a popup menu when the user makes the connection, where the appropriate variables can be chosen. Whether that is better than having explicit slots is probably a matter of personal taste. There are other important differences between our system and Hypershade though. One of them is the ability to group several nodes in a single group node. For more complicated shaders, this is an important feature which can be used to create a better overview of the graph. Maya does not have this feature though. Maya does also not have vertex shading support, which makes it impossible to do vertex transformations in Maya.

While the shaders are easy to create in Maya, they are not usable for use with a real-time rendering engine, as it is not possible to export the shader to an effect file. Shaders created in Maya can therefore only be used inside Maya, for rendering images or animations. This is actually the same in our case, where the created shaders are only usable with the Unity engine. This is quite obvious as we are using the custom effect file language called Shaderlab, and because we have chosen to do the tight integration with this specific engine. In the next section we will discuss the pros and cons of having a shader graph editor as a stand alone tool, in a content creation tool or in a game engine.

Another big difference between our system and Hypershade in Maya, is the missing ability to create new nodes in Maya. When playing with Hypershade, we often were missing specific nodes that just were not there, and as it is impossible to create the nodes oneself, this could lead to effects that just cant be made. In our system on the other hand, we have a rather simple interface for creating new nodes, where most of the functionality of a node is already implemented in a parent class. Creating nodes requires a programmer, which uses a little time to understand the node interface, but once a node has been created it may be used many times in many different situations. We therefore felt that it was an important feature to have, in order to insure extendibility of the product.

Figure 8.2 in chapter 8 demonstrates how the discussed shader can be created using our shader graph system. As the figure demonstrates, the creation process should be quite intuitive, as the user just has to connect the individual nodes. Of cause the user must know graphical terms such as normal maps, and know what to use the material nodes for, in order to create shaders using our system. This is in accordance with the target user group presented in chapter 4 though. On the contrary to both Renderman and RenderMonkey, no programming has to be done when creating shaders using our system. The user can easily play with the connections, to create new interesting effects. Whenever a connection between different spaces are made, it is either handled in the node, or a conversion node is inserted to perform the required transformation. The result should therefore always be valid and in accordance with what the user expects from the graph. Further more it is only possible to create legal connections, as only slots of the same types can be connected. We aid the user to make the right connections by coloring different slot types in different colors. This is different from some of the previous academic work [10], where it were possible to setup illegal connections. In figure 9.1 we show the result of rendering with the four different methods side by side. The reason why they have slight variations, is that some were made on a PC and some on a Mac. Those two systems have quite a large difference in the standart gamma setting, and even though we tried to adjust for that it were difficult to find the coresponding intensity and ambient settings. The bumpy look on the Renderman rendering also looks slightly different. This is because Renderman actually displaces the geometry, where the other methods just use the perturbed normal to do their lighting calculations.

9.2 Game Engine Integration

The most important aspect of our system, that really sets it apart from most of the previous work, is our tight integration with a game engine. This integration has let to several things, for example the use of an effect file for storing the shaders, which again leads to support for features such as handling the rendering state, having multiple passes and so on. As far as we know, no academic work has previously discussed storing shader graph shaders in effect files, which is also why previous academic work has not supported changing the rendering state, nor having ambient passes or multiple passes. We support both of these features, which gives a higher degree of flexibility for the shader graph.

In chapters 2 and 7 we have discussed some of the other tools available for creating shaders. These tools can all be divided into three categories, namely stand alone editors, content creation tools and tools for real-time rendering engines. Depending on the category of a specific shader graph editor, there is quite a big difference of what it is possible to use the editor for. The stand alone editors are naturally the most isolated tools. Some of them are able to work on the actual scene geometry, but they have to export the created shader to an effect file, which then must be integrated into a rendering engine. The stand alone systems can therefore only be used for the actual shader generation, and a significant amount of work remains to make it work in the engine. This work includes integration with the lighting system, to support different light types, and integration with the shadow calculations. In order to support lighting and shadows, it will be necessary to create multiple versions of the shader, or possibly find another scheme which handles attenuation and shadow calculations in a generic way.

The content creation tools are more versatile than the stand alone tools, at least with respect to lighting and shadow calculations. Using the shader graph editors in the content creation tools, it is possible to create material effects, which automatically supports different light types and shadows inside the tool. It is unclear how this is supported though, it might be through the rendering scheme used for producing the renderings, such as photon mapping or ray-tracing. The content creation tools has the same problem as the stand alone tools, namely that in order to use their shaders in a game engine, it is necessary to export the effect file, if that is even possible. Exported effect files will face the same problems with shadow and light type integrations, and therefore it will not be straight forward to use them in a real-time engine.

None of the past academic work that we could find, discussed integrating the created shaders with a real-time game engine. In the industry there has also been very few examples of this, one of the only ones are the material editor of Unreal Engine 3, which is not accessible to normal people. This makes the work presented in this thesis rather unique, as we present a full integration with a commercial game engine. Using our product, shaders can be created in a simplified way, which is quite similar to that of both the stand alone tools and the content creation tools. Unlike those other tools though, shaders created using our system will automatically be preprocessed to support different light types, by creating multiple versions of the vertex and fragment programs. When future versions of the game engine will support shadows, the shaders will also automatically have support for those, using the preprocessing system discussed in chapters 5 and 6. The strongest argument for integrating the shader graph with an engine, is that we want the generated shader where we need it. It is common to use several different content creation tools, but rather uncommon to use different engines, when creating games or other real-time rendering applications. So it is unlikely that the generated shader is going to be used outside the engine anyways.

In our system we have further more explored vertex shading using a shader graph editor, which is something none of the other available systems has. Vertex shaders give the user support for implementing animation features, which then runs on the graphics hardware. In the future vertex shaders might also become increasingly relevant for doing physics calculations on the GPU. Vertex shaders also has a relevance in shading calculations though, as operations such as vector calculations and space transformations, can be moved to the vertex program in order to save instructions in the fragment program. As our shader graph has support for vertex programs, we have also explored a method, which automatically keep as many operations as possible in the vertex program. We further more have taken measures to optimize the structure that transfers variables from the vertex to the fragment program, in order to maximize the amount of variables we can put through the interpolator.

There is one possible issue with our game integration, which is the amount of vertex and fragment programs generated, when many keywords are used. In order to investigate the implications of this, we hand-coded a version of the anisotropic Ward shader, and put in the keywords listed in table 9.1.

POINT	SPOT	DIRECTIONAL	
SHADOW	NOSHADOW	HDR	NOHDR
IMGEFFECT	NOIMGEFFECT	MOTIONBLUR	NOMOTIONBLUR
LIGHTMAP	NOLIGHTMAP	DOF	NODOF
NVIDIA	ATI	INTEL	MATROX

Table 9.1: *The Anisotropic Ward shader were compiled with these keywords, yielding 786 individual vertex and fragment programs.*

Using our preprocessing system, 768 versions of the vertex and fragment programs were created, matching the amount of possible combinations of those keywords. The processing time for the shader were quite long (1-2 minutes), as the Cg command-line compiler had to be opened more than 1500 times during the process. When using the shader though, the framerate of the scene rendered remained the same, which indicates that the high amount of keywords, does not slow down the binding of the shader significantly. The only other problem that could arise is the space consumed by the large shaders on the graphics hardware. Lets consider a more realistic scenario, where a shader of 200 instructions is processed to give 20 individual vertex and fragment programs. If we further more say that each instruction takes up 10 bytes of memory on the graphics card, the result is that $200 * 20 * 10 = 40000$ which is approximately 40 kilobytes of memory. If the scene has 100 of these shaders, they would take up about four megabyte on the graphics card, under the assumption that they are not compressed or anything like that. This is roughly the same size as two high resolution textures. Our conclusion is that the preprocessing system should not cause problems under normal or even more extreme use cases, and we feel that it is the best way to give the flexibility of supporting multiple effects in one shader.

Shadows are one of the most important visual elements in computer graphics, as well as in the real world. Shadows is an important visual clue in images, as it can be impossible to determine the spacial position of objects without them, and they can also add mood, information about the time of day and much more. In chapter 2 we discussed how most of the previous industry work, and all of the previous academic work, did not consider support for shadows in game engines. In figure 8.10 and 8.11 we showed how the keyword processing system introduced in this thesis, will give the created shaders support for toggling shadows on or off.

Even though that the shadow scheme used is a custom raytracing like method, the argument is that it does not matter how the shadow calculations are made. So when the shadow system of Unity is completed, it will exchange our custom code, and the generated shaders will then support shadows in a generic way.

9.3 Graph Shaders and Hand-coded Shaders Comparison

In order for the shaders created using the shader graph to be practical, they may not run significantly slower than similar handcoded shaders. If they do run slower, not many would have an interest in them, as optimized rendering and getting as much out of the graphics card as possible is important for games. The advanced shader created with the graph in figure 8.4, implements a reflecting parallax bump-mapped effect, which is used to produce a silver effect on a sphere. We will use this advanced shader to discuss potential performance pitfalls with the shader graph.

The transformers are the most delicate point for the shader graph tool when it comes to performance. If we assume that the nodes are reasonably well implemented, so that they do not create unnecessary compiled code [1], then the inserted transformers should be the only difference between a hand coded shader, and a shader generated using the graph. The transformers are inserted whenever a connection between slots defined in different mathematical spaces are created. Sometimes it is possible to optimize the amount of matrix multiplications though, or at least to move some of them to the vertex program by being a little smart. This means that a hand coded shader, if programmed by a smart shader programmer, may have less instructions and therefore run faster. For comparison we have produced an optimized version of the parallax shader from figure 8.4, which we have compared to the one generated with the shader graph in table 9.2. The generated and hand written source code for the two shaders can be found in Appendix B.4. By studying the code of both of the shaders, one can see that the main difference is that a matrix transpose were omitted in the hand coded shader, and a matrix multiplication were moved to the vertex program instead of the fragment program. The difference is that the hand coded shader setup a tangent to world space matrix in the vertex program, and passes that to the fragment program. In the fragment program, the normal from the normal map is transformed into world space, before it is used with the world space viewing vector to find the reflected vector. In the generated version, the reflection is done in tangent space, and the reflected vector is then transformed

[1] Extra code in the nodes is only a problem if it survives the Cg compilers optimizations.

into world space in the fragment program, which is an operation that requires two matrix multiplications plus a transpose of the object to tangent space matrix. One additional difference is that the hand coded shader uses the world space viewing vector in the parallax calculations, which is not really correct as it should be the tangent space vector. We experimented with both and found the visual difference to be so small that it were neglicable. It would still be more correct to use the tangent space vector though, which would introduce another matrix multiplication in the vertex program of the hand coded shader.

	Vertex				Fragment			
	Graph		Hand-coded		Graph		Hand-coded	
	Lit	Amb.	Lit	Amb.	Lit	Amb.	Lit	Amb.
Point	25	18	23	24	36	35	37	23
Spot	25	18	23	24	38	35	39	23
Directional	21	18	19	24	34	35	35	23

Table 9.2: *This table compares the amount of instructions in the processed vertex and fragment programs. The comparison is done for a hand coded shader, and one generated with the shader graph, for both the ambient and light calculating pass.*

The amount of instructions shown in table 9.2, is the number of assembly instructions generated by the Cg compiler, when the vertex and fragment programs are processed with the preprocessor. The biggest difference is found in the fragment program in the ambient pass. This is where the reflection calculations used with the environment map is calculated, and where the hand coded shader were optimized by avoiding a matrix transpose and a matrix multiplication. This results in 12 instructions less or about 33 percent fewer than the generated shader, which is a quite substantial optimization. As it can be seen, the corresponding vertex program is 6 instructions longer due to the inserted transformation, but that is not a problem as typical game scenes has far fewer vertices than fragments on the screen, so it is usually considered an optimization to move calculations to the vertex program.

Another interesting observation is that the hand coded shader actually is one instruction more expensive in the fragment program in the light pass. It is not apparent why that is so, as no extra calculations goes on in that program, and the result of the two programs are the same. We believe that the extra instruction is due to the Cg compiler who is missing some optimization due to the difference in the high level code. So while the output should be identical between the two shaders, some coding related issue makes the compiler miss a possible optimization. The same seems to be the case for the vertex program of the lighting pass, where the generated shader has two extra instructions compared to the hand coded one.

As it can be seen from this section, one should be careful if the shader graph has a lot of transformation operations. If a person with shader programming experience is available though, it would be possible to use the graph to create shaders in a faster and easier way, and then have this person doing some optimizations by hand afterwards. We believe that while this certainly is an issue, it is not a major one, because most users will not run in to these problems very often. Most users will probably use the nodes that ship with the system, and add extra textures, color ramps and alike to create a custom material effect. It is likely that many of these operations are performed on colors, which does not exist in a particular basis, and therefore no transformers will be inserted. Future optimizations giving a more intelligent automatic transformation system would be interesting though, as this should result in a smaller performance difference between hand written and graph generated shaders.

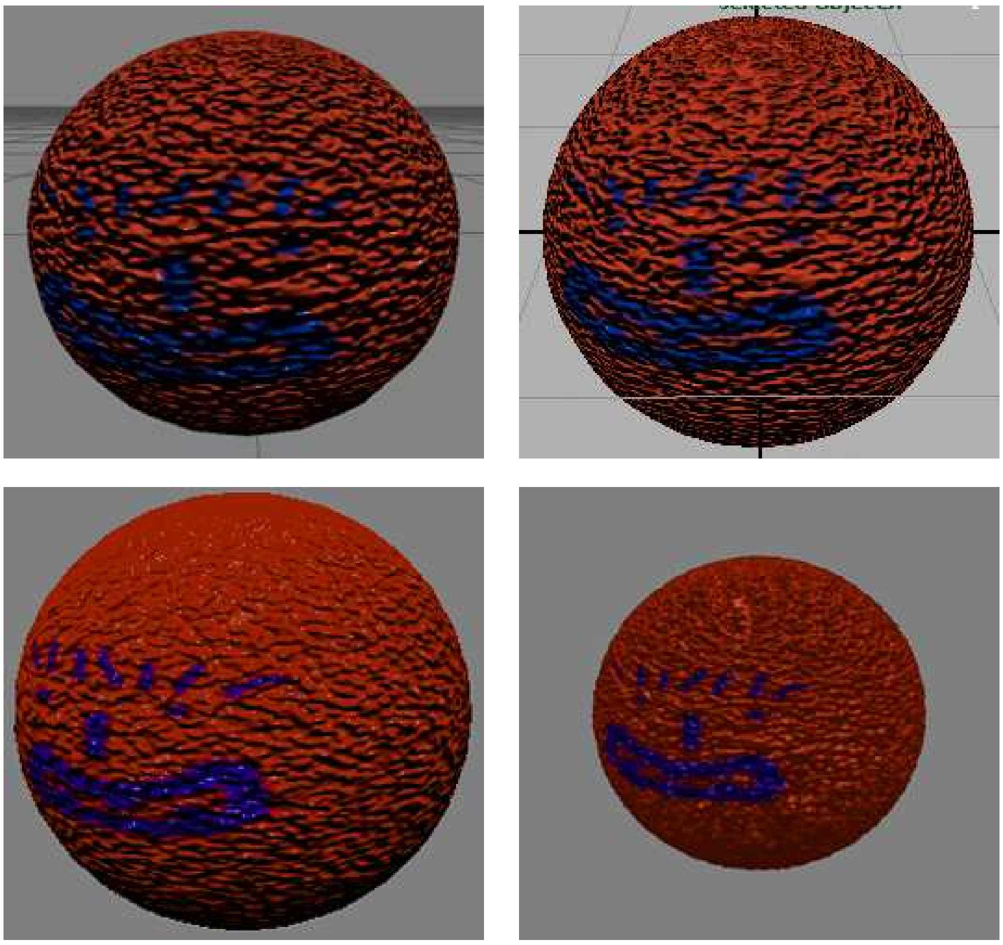


Figure 9.1: *The bumped sphere rendered with the four different systems. Top left is our shader graph. Then using Hypershade in Maya. Bottom left is the Rendermonkey version, and then in the bottom right its the result from Renderman.*

Chapter 10

Conclusion and Future Work

10.1 Future Work

throughout this thesis we have often made arguments that the shader graph editor is very easy to use by non-programmers. Those arguments are based on subjective belief, and the fact that non-programmers are using similar editors made by others. It remains to be tested if users will find our editor just as easy to use though, so future work should definitely include a generalized user test, that we unfortunately did not have time for during this project. Other work that aims towards completing the product, is creating even more nodes such as a Fresnel node. More material nodes that implement other BRDF's could also be interesting. On a more academic note, updating the shader graph to have support for the GLSL shading language would be quiet interesting. This would give us the possibility to support future shader models on all capable graphics cards. An example could be shader model 3.0, where it would be very interesting to experiment with support for dynamic branching and vertex textures.

10.2 Conclusion

This thesis has presented the design and implementation of a shader graph editor. The editor can be used to create shaders in a simplified way, by connecting nodes with different functionality to create an specific effect. The presented system ensures that these connections are both syntactically and mathematically correct, by disallowing connections of differently typed variables, and automatically handling transformations from one mathematical space to another. The automatic space transformation has been demonstrated during the creation of a parallax mapping shader, where automatic transformations to both tangent and world space are performed, without the user has to think about that. It should be noted that under some circumstances, the automatic transformations can cause a graph shader to perform slower than a handcoded shader. Generally the two shaders will perform equally well though.

In order to increase the usability of the editor, we have implemented a grouping feature, so several nodes can be turned into one. We have demonstrated through examples, how this can be used to increase the overview of a shader graph. We have further more introduced an preview option for the nodes, that will demonstrate the intermediate results throughout the shader graph. This preview option will make debugging the shaders generated with the graph easier, as it will be possible to see where something goes wrong. The features such as creating group nodes, are handled from our simple but yet fully functional GUI system.

On the contrary to other accessible shader graph editors, we are able to handle both vertex and fragment shading using the editor described here. This has given the possibility to perform optimizations by putting more calculations in the vertex program, and has enabled the shader graph user to create shaders that actually transform the vertex position. During both vertex and fragment shading, swizzeling can be needed, so this has been supported by making specific swizzle nodes. In the case that more nodes are needed, it should be possible for programmers to create those, by using the simple node interface where basically only a single function has to be created.

The presented shader graph has been integrated with a commercial game engine, which has given us the unique opportunity to support effects such as shadows and different light attenuation in the created shaders. This has been done by creating a novel processing step, where several versions of the vertex and fragment programs are generated, to match the effects that should be supported by the shader. The engine integration has also led to the use of effect files for storing the shaders, which gives the user the option to handle OpenGL rendering state settings, such as blending and culling.

Appendix A

Introduction to Unity

A.1 Introduction to Unity

Unity is a Mac based game development platform, developed with ease of use in mind. The system combines a powerful game engine with an editing interface. Unity has always been developed with ease of use and workflow in mind. Everything in Unity is contained within one window, that can look like the screenshot in figure A.1.

The top-left view of the application is the scene view. When creating a game the user sets up her scene in this view by dragging objects and models into it. Models from most modeling packages are imported into unity automatically, if they are saved in the folder for the active project. It is not possible to do any modeling inside Unity itself. The contents of the project folder can be examined in the project view, which is situated to the bottom right of the scene view. It is possible to use the create button on the top of this view to create new materials, render-textures and so on. Any selection will be shown in the inspector to the right of the project view. Here is is possible to manipulate the selected game object in any way it supports. For the case of the selected material it is possible to assign its texture, change it to use a different shader and so on. The whole scene hirachy is displayed in the middle-top view, and the left-bottom view displays the result as it will look in the actual game. Unity has

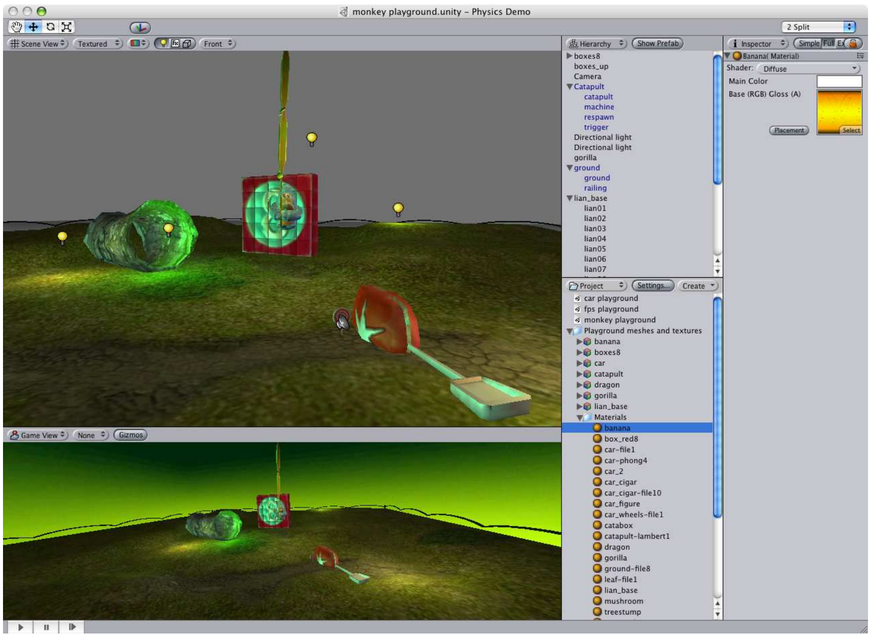


Figure A.1: The Unity application

build-in physics, using the well known Novodex API from Ageia. Physics and other game-play related behaviors can be scripted using the .NET framework (C#, javascript) or the phyton like language Boo. Unity also has packages for importing and playing audio-clips and text rendering. The text rendering were implemented as a part of the shader graph project, and will be discussed briefly later. Unity is primarely aimed towards indie game developers for casual games creation. It is also possible to use it for architectural renderings, 3D commercials and much more. The standard Unity user will be an artist that knows a little programming, or an indie developer that knows a little bit of everything. More information on Unity and projects involving Unity can be found on the applications home-page: www.otee.dk.

As this project concerns the creation of shaders, we will primarily concern ourselves with the material system in Unity, and focus on a way to develop and integrate a graphical approach to shader creation. The motivation for this is that very few users take advantage of the advanced shader capabilities Unity offers. Unity has a build in shading language (discussed in the next section), but most users are intimidated by the level of difficulty, and graphics programming knowledge required, to master such a language. We therefore with to present an more accessible way, to utilize the powerful graphics capabilities of modern computers.

Appendix B

Created Shaders Source Code

B.1 Sample ShaderLab shader

```
Shader " Glossy" {
  Properties {
    _Color ("Main Color", Color) = (1,1,1,1)
    _SpecColor ("Specular Color", Color) = (0.5, 0.5, 0.5, 1)
    _Shininess ("Shininess", Range (0.01, 1)) = 0.078125
    _MainTex ("Base (RGB) Gloss (A)", 2D) = "white" {}
  }
  Category {
    tags { "lod" = "0" }
    Blend AppSrcAdd AppDstAdd

    Fog { Color [_AddFog] }
    // ---------------------------------------------------------
    // Geforce 3, ARB_FP
    // ---------------------------------------------------------
    SubShader {
      TexCount 4    // Get Geforce2s to ignore this shader
      Pass {           // Ambient only
        Name "BASE"
        Tags {"LightMode" = "PixelOrNone"}
        Blend AppSrcAdd AppDstAdd
        Color [_PPLAmbient]
        SetTexture [_MainTex] {constantColor [_Color] Combine texture * primary DOUBLE, texture *
constant}
      }
      Pass {
        Name "BASE"
        Tags {"LightMode" = "Vertex"}
        Lighting On
        Material {
          Diffuse [_Color]
          Emission [_PPLAmbient]
          Specular [_SpecColor]
          Shininess [_Shininess]
        }
        SeparateSpecular On

CGPROGRAM
```

```
#pragma profiles arbfp1
#pragma fragment
#pragma fragmentoption ARB_fog_exp2
#pragma fragmentoption ARB_precision_hint_fastest

#include "UnityCG.cginc"

struct v2f {
  float4 hPosition    : POSITION;
  float4 uv: TEXCOORD0;
  float4 diff       :COLOR0;
  float4 spec     :COLOR1;
};

uniform sampler2D _MainTex;
half4 main (v2f i) : COLOR {
  half4 temp = tex2D (_MainTex, i.uv.xy);
  temp.xyz = (temp.xyz * i.diff.xyz + i.spec.xyz * temp.w ) * 2;
  temp.w = temp.w * i.diff.w;
  return temp;
}
ENDCG
        SetTexture [_MainTex] {Combine texture * primary, constant}
      }
      Pass {
        Name "PPL"
        // Sum all light contribs from 2-tex lights
        Tags {
          "LightMode" = "Pixel"
          "LightTexCount" = "012"
        }
        Blend AppSrcAdd AppDstAdd
        Fog { Color [_AddFog] }
CGPROGRAM
#pragma profiles arbfp1
#pragma fragment frag
#pragma vertex vert
#include "UnityCG.cginc"
#include "AutoLight.cginc"
#pragma fragmentoption ARB_fog_exp2
#pragma fragmentoption ARB_precision_hint_fastest
#pragma MULTIINCLUDE SPOT

struct appdata {
  float4 vertex;
  float4 tangent;
  float3 normal;
  float4 texcoord;
};

struct v2f {
  float4 hPosition    : POSITION;
  float fog     : FOGC;
  float4 uv : TEXCOORD2;
  float3 viewDirection : TEXCOORD3;
  float3 normal : TEXCOORD4;
  float3 lightDirection : TEXCOORD5;
  LIGHTMAPCOORDS
};

Light l;

v2f vert (appdata v) {
  v2f o;
  o.hPosition = mul (glstate.matrix.mvp, v.vertex);
  o.fog = o.hPosition.z;

  // Compute the object-space light vector
  o.lightDirection = _ObjectSpaceLightPos[0].xyz - v.vertex.xyz * _ObjectSpaceLightPos[0].w;
  o.uv = mul(v.texcoord,glstate.matrix.texture[0]);
  o.viewDirection =  _ObjectSpaceCameraPos - v.vertex.xyz;
  o.normal = v.normal;

  TRANSFER
  return o;
}

uniform sampler2D _MainTex ;
uniform float4 _SpecColor;
uniform float _Shininess;
float4 frag (v2f i)  : COLOR {

  // Normalizes light vector with normalization cube map
  float3 light = normalize (i.lightDirection);
  float3 normal = normalize(i.normal);
  float3 V = normalize(i.viewDirection);
  float3 H = normalize(light + V);
  float4 color;
  float4 texcol = tex2D(_MainTex,i.uv.xy);
  float diffuse = dot(normal,light);
  float spec = pow( max(0, dot(H, normal)), _Shininess*128)*texcol.a;
   spec = spec*diffuse;

  color.rgb =  (diffuse * texcol.rgb  * _ModelLightColor[0].rgb + _SpecColor.rgb * spec * _LightColor0.r
  color.a = spec *  _SpecColor.a * LIGHTATT(i);

  return color;
}
ENDCG
        SetTexture [_LightTexture0] { combine previous * texture}
        SetTexture [_LightTextureB0]{ combine previous * texture}
        SetTexture [_MainTex]{ combine previous * texture}
      }
    }
  }
  Fallback " VertexLit", 0
}
```

B.2 Renderman Source Code

```
surface bumpspecmodulate ( float Ka = 1, Kd = .2, Ks = .2, roughness = 0.2;
                           color specularcolor = 1;
                           string texturename = "";
             string texturename2 = "";
             color playercol = (1.0,0.0,0.0);){
    Ci = Cs;
    normal Nf;
    color texcol;
    float alpha;
    if (texturename != "") {
  texcol = texture (texturename);
        alpha = texture (texturename2);
    }

    Nf = faceforward (normalize(N),I);
    Ci *= playercol + ((alpha) * (texcol - playercol));

    Ci = Ci * (Ka*ambient() + Kd*diffuse(Nf)) + specularcolor * Ks*specular(Nf,-normalize(I),roughness);
    Oi = Os;  Ci *= Oi;
}


displacement bumpy(
      float Km = 0.03;
      string normalmap = "";)
{
  float amp = Km * float texture(normalmap);
  P += amp * normalize(N);
  N = calculatenormal(P);
}


# Texture Mapping example using quadrics
# GSO 7-15-98
Display "quads_text1.tif" "file" "rgb"
Format 512 512 -1
Projection "perspective" "fov" [30]
Translate 0 0 3
Rotate 80 1 0 0
Rotate 80 0 0 1
WorldBegin
 LightSource "ambientlight" 1 "intensity" [0.3]
 LightSource "distantlight" 2 "intensity" [2]"from" [0.0 0.9 0.0]"to" [20 -5 5.5]
 Color [1 1 1]
 Surface "bumpspecmodulate" "texturename" "colormap.tif"
 Displacement "bumpy" "normalmap" "bumpheight.tif"
 Sphere[0.5 -0.5 0.5 360]

WorldEnd


light
ambientlight(
      float intensity = 1;
      color lightcolor = 1;) {
  Cl = intensity * lightcolor ;
}
```

```
distantlight(
      float intensity = 1;
      color lightcolor = 1;
      point from = point ÓshaderÓ (0,0,0);
      point to = point ÓshaderÓ (0,0,1); {
  solar(to-from, 0.0)
  Cl = intensity * lightcolor;
}
```

B.3 RenderMonkey Source Code

```
attribute vec3 rm_Tangent;
attribute vec3 rm_Binormal;

uniform vec4 lightDir;
uniform vec4 vViewPosition;


varying vec2  varTexCoord;
varying vec3  varNormal;
varying vec3  varWposT;
varying vec3  varCamPos;
varying vec3 lightDirInTangent;
varying vec3 varLightVec;

void main(void)
{
   // Output transformed vertex position:
   gl_Position = ftransform();

   // Compute the light vector (view space):
   varLightVec   = lightDir.xyz;

   mat3 tangentMat = mat3(rm_Tangent,
                          rm_Binormal,
                          gl_Normal);

   lightDirInTangent = normalize(lightDir.xyz) * tangentMat;

   // Compute view vector (view space):
   varCamPos = vViewPosition.xyz * tangentMat;

   varWposT = gl_Vertex.xyz * tangentMat;
   varTexCoord = gl_MultiTexCoord0.xy;
}


uniform sampler2D bump;
uniform sampler2D baseMap;
uniform vec4   SecondColor;
varying vec2  varTexCoord;
varying vec3  varNormal;
varying vec3  varWposT;
varying vec3  varCamPos;
varying vec3 lightDirInTangent;
varying vec3 varLightVec;

void main(void) {

  // Normalizes light vector with normalization cube map
  vec3 light = normalize(lightDirInTangent);

  // Sample and expand the normal map texture
  float3 normal = normalize(texture2D(bump,varTexCoord ).xyz * 2  - 1);

  // Diffuse lighting with spec and bump
  vec3 V = normalize(varCamPos - varWposT);
  vec3 H = normalize(lightDirInTangent + V);

  vec4 texcol = texture2D(baseMap,varTexCoord);
  texcol.rgb = lerp(SecondColor.rgb, texcol.rgb, texcol.a);

  float diffuse = dot(normal,light) * .6;

  float spec = pow( max(0, dot(H, normal)),100) * .5;

  gl_FragColor.rgb = (diffuse * texcol.rgb + spec) * 2.0 + (0.2 * texcol.rgb);


  gl_FragColor.a = 1.0;
}
```

B.4 Handcoded and Generated Parallax Mapping Shaders

```
Shader " GeneratedShader"
{ Properties {
Env_Scale ("Env_Scale", Color) = ( 0.5, 0.5, 0.5, 0.5)
_Parallax ("Parallax", Range (0.0, 0.06)) = 0.02
_Normal_Map ("Normal Map", 2D) = "white" {}
_Texture ("Base (RGB) Gloss (A)", 2D) = "white" {}
_Color ("Main Color", Color) = (.5, .5, .5, .5)
_SpecColor ("Specular Color", Color) = (0.5, 0.5, 0.5, 1)
_Specular ("Specular Constant", Range (0.0001, 0.99)) = 0.4
_Enviroment_Map ("Enviroment Map", Cube) = "white" {}
Env_Scale ("Env_Scale", Color) = ( 0.5, 0.5, 0.5, 0.5)  }
  Category {
    tags { "lod" = "0" }
      Fog { Color [_AddFog] }
      Blend AppSrcAdd AppDstAdd
    SubShader {
      // Ambient pass
      Pass {
        Tags {"LightMode" = "PixelOrNone"}
        Color [_PPLAmbient]
CGPROGRAM
#pragma profiles arbfp1
#pragma fragment frag
#pragma vertex vert
#include "UnityCG.cginc"
#include "AutoLight.cginc"
#pragma fragmentoption ARB_fog_exp2
#pragma fragmentoption ARB_precision_hint_fastest

struct appdata {
  float4 vertex;
  float4 tangent;
  float3 normal;
  float4 texcoord;
};

struct v2f {
  float4 hPosition : POSITION;

float4 transferSlot0 : TEXCOORD1;};

float3 Env_Scale;
uniform float _Parallax;
uniform sampler2D _Normal_Map;
uniform float4 _Normal_Map_ST;
uniform sampler2D _Texture;
uniform float4 _Texture_ST;
uniform float4 _Color;
uniform samplerCUBE _Enviroment_Map;
v2f vert (appdata v) {
  v2f vertexToFragment;
  TANGENT_SPACE_ROTATION;
  float3x3 invRotation = transpose(rotation);

vertexToFragment.transferSlot0.zw = v.texcoord.xy*_Normal_Map_ST.xy+_Normal_Map_ST.zw;
vertexToFragment.transferSlot0.zw = v.texcoord.xy;
float3 Light_Direction = _ObjectSpaceLightPos[0].xyz - v.vertex.xyz * _ObjectSpaceLightPos[0].w;
float3 View_Direction = ObjSpaceViewDir(float4(v.vertex.xyz,1));float3 Position = _ObjectSpaceCameraPos;
float3 Inverted_Vec = -View_Direction;
 vertexToFragment.transferSlot0.yzw = mul(rotation, Inverted_Vec);
vertexToFragment.hPosition = mul (glstate.matrix.mvp, float4(float3(v.vertex.xyz),1));
return vertexToFragment;
}

float4 frag (v2f vertexToFragment)  : COLOR  {

float4 normalMapNormalI = tex2D(_Normal_Map, vertexToFragment.transferSlot0.zw);
float3 NormalI = normalMapNormalI.rgb * 2.0 - 1.0;
float AlphaIII = normalMapNormalI.a;
float2 Parallax_UV = vertexToFragment.transferSlot0.zw + (float3(1,1,0).xy * (AlphaIII * (_Parallax*2) -
vertexToFragment.transferSlot0.xy = Parallax_UV*_Normal_Map_ST.xy+_Normal_Map_ST.zw;
float4 normalMapNormal = tex2D(_Normal_Map, vertexToFragment.transferSlot0.xy);
float3 Normal = normalMapNormal.rgb * 2.0 - 1.0;
float AlphaII = normalMapNormal.a;
vertexToFragment.transferSlot0.zw = Parallax_UV*_Texture_ST.xy+_Texture_ST.zw;
float4 texColorTexture = tex2D(_Texture, vertexToFragment.transferSlot0.zw);
float3 ColorIIII = texColorTexture.rgb;
float AlphaIIIII = texColorTexture.a;
float3 ColorII = _Color.rgb * _PPLAmbient.rgb * ColorIIII;
float Attenuation = 1.0;
```

```
float3 ColorIII = _ModelLightColor[0].rgb;
float3 OutputI = ColorII*Attenuation;
float3 Refl_Direction = reflect(vertexToFragment.transferSlot0.yzw, Normal);
float3 Out = mul((float3x3)_Object2World, mul(invRotation, Refl_Direction));
float4 texColorEnviroment_Map = texCUBE(_Enviroment_Map, Out);
float3 ColorI = texColorEnviroment_Map.rgb;
float AlphaI = texColorEnviroment_Map.a;
float3 Output = Env_Scale*ColorI;
float3 colorResult = OutputI;
return float4(Output + colorResult, float(float(1)));}
ENDCG
}
      Pass {
        Tags {
          "LightMode" = "Pixel"
          "LightTexCount" = "012"
        }
CGPROGRAM
#pragma profiles arbfp1
#pragma fragment frag
#pragma vertex vert
#include "UnityCG.cginc"
#include "AutoLight.cginc"
#pragma fragmentoption ARB_fog_exp2
#pragma fragmentoption ARB_precision_hint_fastest

struct appdata {
  float4 vertex;
  float4 tangent;
  float3 normal;
  float4 texcoord;
};

struct v2f {
  float4 hPosition : POSITION;

float4 transferSlot0 : TEXCOORD1;
float4 transferSlot1 : TEXCOORD2;
float4 transferSlot2 : TEXCOORD3;
float4 transferSlot3 : TEXCOORD4;
#pragma multi_compile POINT SPOT DIRECTIONAL
#pragma multi_compile SHADOW NOSHADOW
LIGHTING_COORDS};

float3 Env_Scale;
uniform float _Parallax;
uniform sampler2D _Normal_Map;
uniform float4 _Normal_Map_ST;
uniform sampler2D _Texture;
uniform float4 _Texture_ST;uniform float _Specular;
uniform float4 _Color;
uniform float4 _SpecColor;
Light l;
uniform float4 _SphereInf;
v2f vert (appdata v) {
  v2f vertexToFragment;
  TANGENT_SPACE_ROTATION;
  float3x3 invRotation = transpose(rotation);

vertexToFragment.transferSlot1.zw = v.texcoord.xy*_Normal_Map_ST.xy+_Normal_Map_ST.zw;
vertexToFragment.transferSlot1.zw = v.texcoord.xy;
vertexToFragment.transferSlot2.xyz = mul(rotation, ObjSpaceLightDir( v.vertex ) );
vertexToFragment.transferSlot3.xyz = mul(rotation,  ObjSpaceViewDir( v.vertex ) );
TRANSFER_VERTEX_TO_FRAGMENT(vertexToFragment)
float3 Light_Direction = _ObjectSpaceLightPos[0].xyz - v.vertex.xyz * _ObjectSpaceLightPos[0].w;
float3 ray = _ObjectSpaceLightPos[0].xyz - v.vertex.xyz;
float3 center = mul(_World2Object, float4(_SphereInf.x, _SphereInf.y, _SphereInf.z, 1)).xyz;
float radius = _SphereInf.w;
float a = dot(ray, ray);
float b = dot(2*ray, v.vertex.xyz - center);
float c = dot(v.vertex.xyz - center, v.vertex.xyz - center) - (radius*radius);vertexToFragment.transferSlot0.
vertexToFragment.hPosition = mul (glstate.matrix.mvp, float4(float3(v.vertex.xyz),1));
return vertexToFragment;
}

float4 frag (v2f vertexToFragment)  : COLOR  {

float4 normalMapNormalI = tex2D(_Normal_Map, vertexToFragment.transferSlot1.zw);
float3 NormalI = normalMapNormalI.rgb * 2.0 - 1.0;
float AlphaIII = normalMapNormalI.a;
float2 Parallax_UV = vertexToFragment.transferSlot1.zw + (float3(1,1,0).xy * (AlphaIII * (_Parallax*2) - _Par
vertexToFragment.transferSlot1.xy = Parallax_UV*_Normal_Map_ST.xy+_Normal_Map_ST.zw;
float4 normalMapNormal = tex2D(_Normal_Map, vertexToFragment.transferSlot1.xy);
float3 Normal = normalMapNormal.rgb * 2.0 - 1.0;
float AlphaII = normalMapNormal.a;
vertexToFragment.transferSlot1.zw = Parallax_UV*_Texture_ST.xy+_Texture_ST.zw;
float4 texColorTexture = tex2D(_Texture, vertexToFragment.transferSlot1.zw);
```

```
float3 ColorIIII = texColorTexture.rgb;
float AlphaIIIII = texColorTexture.a;
 vertexToFragment.transferSlot0.xyz = Normal;
vertexToFragment.transferSlot2.xyz = normalize(vertexToFragment.transferSlot2.xyz);
vertexToFragment.transferSlot3.xyz = normalize(vertexToFragment.transferSlot3.xyz);
float3 halfAngleVector = normalize(vertexToFragment.transferSlot3.xyz + vertexToFragment.transferSlot2.x
float diffuse = dot(vertexToFragment.transferSlot0.xyz, vertexToFragment.transferSlot2.xyz);
float nDotH = saturate(dot(vertexToFragment.transferSlot0.xyz, halfAngleVector));
float specular = pow(nDotH, _Specular * 128) * AlphaIIIII;
specular *= diffuse;
float3 ColorII = (_Color.rgb * _ModelLightColor[0].rgb * diffuse * ColorIIII + _SpecColor.rgb * specular
float AlphaIIII = specular * _SpecColor.a;
float Attenuation = LIGHT_ATTENUATION(vertexToFragment);;
SHADOWCALC(vertexToFragment.transferSlot0.w);
float3 ColorIII = _ModelLightColor[0].rgb;
float3 OutputI = ColorII*Attenuation;
float3 colorResult = OutputI;
return float4(colorResult, float(float(1)));}
ENDCG
}
}
}
}


Shader "ParallaxHandCoded" {
  Properties {
    Env_Scale ("Env_Scale", Color) = ( 0.5, 0.5, 0.5, 0.5)
    _Parallax ("Parallax", Range (0.0, 0.06)) = 0.02
    _Normal_Map ("Normal Map", 2D) = "white" {}
    _Texture ("Base (RGB) Gloss (A)", 2D) = "white" {}
    _Color ("Main Color", Color) = (.5, .5, .5, .5)
    _SpecColor ("Specular Color", Color) = (0.5, 0.5, 0.5, 1)
    _Specular ("Specular Constant", Range (0.0001, 0.99)) = 0.4
    _Enviroment_Map ("Enviroment Map", Cube) = "white" {}
    Env_Scale ("Env_Scale", Color) = ( 0.5, 0.5, 0.5, 0.5)
  }
    Category {
      tags { "lod" = "0" }
      Fog { Color [_AddFog] }
      Blend AppSrcAdd AppDstAdd
      SubShader {
        // Ambient pass
        Pass {
          Tags {"LightMode" = "PixelOrNone"}
          Color [_PPLAmbient]

CGPROGRAM
#pragma profiles arbfp1
#pragma fragment frag
#pragma vertex vert
#include "UnityCG.cginc"
#include "AutoLight.cginc"
#pragma fragmentoption ARB_fog_exp2
#pragma fragmentoption ARB_precision_hint_fastest
#pragma multi_compile POINT SPOT DIRECTIONAL

struct appdata {
  float4 vertex;
  float4 tangent;
  float3 normal;
  float4 texcoord;
};

struct v2f {
  float4 hPosition : POSITION;
  float4 viewDirection : TEXCOORD1;
  float4 viewDirectionT : TEXCOORD6;
  float4 texcoord : TEXCOORD2;
  LIGHTING_COORDS
  float4 tangent : TEXCOORD3;
  float4 binormal : TEXCOORD4;
  float4 normal : TEXCOORD5;
};

float3 Env_Scale;
uniform float _Parallax;
uniform sampler2D _Normal_Map;
uniform float4 _Normal_Map_ST;
uniform sampler2D _Texture;
uniform float4 _Texture_ST;
uniform float4 _Color;
Light l;
uniform samplerCUBE _Enviroment_Map;

v2f vert (appdata v) {
  v2f vertexToFragment;
  TANGENT_SPACE_ROTATION;
    vertexToFragment.tangent.xyz = mul(rotation, _Object2World[0].xyz);
  vertexToFragment.binormal.xyz = mul(rotation, _Object2World[1].xyz);
  vertexToFragment.normal.xyz = mul(rotation, _Object2World[2].xyz);

  vertexToFragment.texcoord.zw = v.texcoord.xy*_Normal_Map_ST.xy+_Normal_Map_ST.zw;
  float3 View_Direction = ObjSpaceViewDir(v.vertex);
  float3 Position = _ObjectSpaceCameraPos;
  vertexToFragment.viewDirection.xyz = mul( (float3x3)_Object2World, -View_Direction);
  vertexToFragment.viewDirectionT.xyz = mul( rotation, View_Direction);
  vertexToFragment.texcoord.zw = v.texcoord.xy;
  vertexToFragment.hPosition = mul (glstate.matrix.mvp, float4(float3(v.vertex.xyz),1));
  return vertexToFragment;
}

float4 frag (v2f vertexToFragment) : COLOR {
    float3x3 rotation = float3x3( vertexToFragment.tangent.xyz, vertexToFragment.binormal.xyz, vertexToFragme

  float4 normalMapNormalI = tex2D(_Normal_Map, vertexToFragment.texcoord.zw);
  float height = normalMapNormalI.a;

  float2 Parallax_UV = vertexToFragment.texcoord.zw + (vertexToFragment.viewDirectionT.xy * (height * (_Paral
  vertexToFragment.texcoord.xy = Parallax_UV*_Normal_Map_ST.xy+_Normal_Map_ST.zw;

  float3 Normal = tex2D(_Normal_Map, vertexToFragment.texcoord.xy).rgb * 2.0 - 1.0;

  vertexToFragment.texcoord.zw = Parallax_UV*_Texture_ST.xy+_Texture_ST.zw;

  float3 Refl_Direction = reflect(vertexToFragment.viewDirection.xyz, mul(rotation, Normal));
  float3 reflection = Env_Scale*texCUBE(_Enviroment_Map, Refl_Direction).rgb;
  float3 colorResult = _Color.rgb * _PPLAmbient.rgb * tex2D(_Texture, vertexToFragment.texcoord.zw).rgb;
  return float4(reflection + colorResult, float(float(1)));
}
ENDCG
        }
        Pass {
          Tags {
            "LightMode" = "Pixel"
            "LightTexCount" = "012"
          }

CGPROGRAM
#pragma profiles arbfp1
#pragma fragment frag
#pragma vertex vert
#include "UnityCG.cginc"
#include "AutoLight.cginc"
#pragma fragmentoption ARB_fog_exp2
#pragma fragmentoption ARB_precision_hint_fastest
#pragma multi_compile POINT SPOT DIRECTIONAL

struct appdata {
  float4 vertex;
  float4 tangent;
  float3 normal;
  float4 texcoord;
};

struct v2f {
  float4 hPosition : POSITION;
  float4 texcoord : TEXCOORD2;
  float4 viewDirection : TEXCOORD3;
  float4 lightDirection : TEXCOORD4;
  LIGHTING_COORDS
};

float3 Env_Scale;
uniform float _Parallax;
uniform sampler2D _Normal_Map;
uniform float4 _Normal_Map_ST;
uniform sampler2D _Texture;
uniform float4 _Texture_ST;uniform float _Specular;
uniform float4 _Color;
uniform float4 _SpecColor;
Light l;

v2f vert (appdata v) {
  v2f vertexToFragment;
  TANGENT_SPACE_ROTATION;

  vertexToFragment.viewDirection.xyz = mul(rotation, ObjSpaceViewDir( v.vertex ));
  vertexToFragment.texcoord.zw = v.texcoord.xy;
  vertexToFragment.lightDirection.xyz = mul(rotation, ObjSpaceLightDir( v.vertex ) );
  TRANSFER_VERTEX_TO_FRAGMENT(vertexToFragment)
  vertexToFragment.hPosition = mul (glstate.matrix.mvp, float4(float3(v.vertex.xyz),1));
  return vertexToFragment;
}
```

```
float4 frag (v2f vertexToFragment)  : COLOR  {

  float height = tex2D(_Normal_Map, vertexToFragment.texcoord.zw).a;
  float2 Parallax_UV = vertexToFragment.texcoord.zw + (vertexToFragment.viewDirection.xy * (height * (_P
  vertexToFragment.texcoord.xy = Parallax_UV*_Normal_Map_ST.xy+_Normal_Map_ST.zw;
  float3 Normal = tex2D(_Normal_Map, vertexToFragment.texcoord.xy).rgb * 2.0 - 1.0;
  vertexToFragment.texcoord.zw = Parallax_UV*_Texture_ST.xy+_Texture_ST.zw;
  float4 texColor = tex2D(_Texture, vertexToFragment.texcoord.zw);

  vertexToFragment.viewDirection.xyz = normalize(vertexToFragment.viewDirection.xyz);
  vertexToFragment.lightDirection.xyz = normalize(vertexToFragment.lightDirection.xyz);
  float3 halfAngleVector = normalize(vertexToFragment.lightDirection.xyz + vertexToFragment.viewDirectio
  float diffuse = dot(Normal, vertexToFragment.lightDirection.xyz);
  float nDotH = saturate(dot(Normal, halfAngleVector));

  float specular = pow(nDotH, _Specular * 128) * texColor.a;
  specular *= diffuse;

  float3 color = (_Color.rgb * _ModelLightColor[0].rgb * diffuse * texColor.rgb + _SpecColor.rgb * specu

  float Attenuation = LIGHT_ATTENUATION(vertexToFragment);
  return float4(color*Attenuation, 1);
}
ENDCG
      }
    }
  }
}
```

Appendix C

CD-ROM Contents

C.1 Files on the CD-ROM

The CD-ROM has three folders, one which has alle the source code written in this project, one with all the figures in high res. and one with videoes on how to use the Shader Graph. The videoes are in the Quick Time format, and can only be garantied to work with the latest version of Quick Time. Futher more the CD-ROm contains the PDF file of the thesis. We strongly recomend looking at the high res. image files instead zooming in on the PDF, as they are of higher quality.

Bibliography

[1] 3DLabs. Opengl shading language specification. Online resource at: $http://www.opengl.org/documentation/glsl/$, February 2006.

[2] Gregory D. Abram and Turner Whitted. Building block shaders. In *SIGGRAPH 92*. ACM, 1990.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. In *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] Okan Arikan. Pixie. Online resource at: $http://www.cs.utexas.edu/\ okan/Pixie/pixie.htm$, November 2005.

[5] Autodesk. Autodesk 3d studio. Online resource at: $http://usa.autodesk.com/adsk/servlet/index?id = 5659302\&siteID = 123112$, March 2005.

[6] Autodesk. Autodesk maya. Online resource at: $http://usa.autodesk.com/adsk/servlet/index?id = 6871843\&siteID = 123112$, March 2005.

[7] Scott Bean. Shaderworks. Online resource at: $https://www.shaderworks.com$, March 2005.

[8] James F. Blinn. Models of light reflection for computer synthesized pictures. In *SIGGRAPH 77*. ACM, 1977.

[9] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. In *SIGGRAPH 82*. ACM, 1982.

[10] Robin L. Cook. Shade trees. In *SIGGRAPH 84*. ACM, 1984.

[11] Microsoft Corporation. Microsoft directx - effect reference. Online resource at: $http : //msdn.microsoft.com/library/default.asp?url = /library/en - us/directx9_c/dx9_graphics_reference_effects.asp$, February 2006.

[12] Microsoft Corporation. Microsoft directx - hlsl. Online resource at: $http : //msdn.microsoft.com/directx/$, February 2006.

[13] Nvidia Corporation. The cg toolkit. Online resource at: $http : //developer.nvidia.com/object/cg_toolkit.html$, September 2005.

[14] Nvidia Corporation. Fx composer. Online resource at: $http : //developer.nvidia.com/object/fx_composer_home.html$, September 2005.

[15] Randima Fernando and Mark J. Kilgard. In *The Cg Tutorial*. Addison-Wesley, 2003.

[16] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial*. Addison-Wesley, 2003.

[17] Freetype. Freetype font library. Online resource at: $http : //www.freetype.org/$, November 2005.

[18] Epic Games. Unreal engine 3 shader graph. Online resource at: $http : //www.unrealtechnology.com/screens/MaterialEditor.jpg$, November 2005.

[19] Andrew S. Glassner. In *Principles of Digital Image Synthesis*. Morgan Kaufmann, 1995.

[20] Frank Goetz, Ralf Borau, and Gitta Domik. An xml-based visual shading language for vertex and fragment shaders. In *Proceedings of the ninth international conference on 3D Web technology*. ACM, 2004.

[21] Mark Harris. Gpgpu: Beyond graphics. In *Procedings of Game Developers Conference 2004*. Nvidia, 2004.

[22] ATI Inc. Rendermonkey shaderprogramming ide. Online resource at: $http : //www.ati.com/developer/rendermonkey/index.html$, February 2006.

[23] RapidMind Inc. Sh metaprogramming language. Online resource at: $http : //libsh.org/$, February 2006.

[24] Morgan McGuire, George Stahis, Hanspeter Pfister, and Shriram Krishnamurhi. Abstract shade trees. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*. ACM, 2006.

[25] Nvidia. Gpu programming guide. In *Nvidia online resource*. Nvidia, 2006.

[26] National Institute of Standarts and Technology. Definition of a directed acyclic graph. Online resource at: *http : //www.nist.gov/dads/HTML/directAcycGraph.html*, April 2004.

[27] Michael Oren and Shree K. Nayar. Generalization of lambertÕs reflectance model. In *SIGGRAPH 94*. ACM, 1994.

[28] John O'Rorke. Integrating shaders into applications. In *GPU Gems*. Addison-Wesley Professional, 2004.

[29] G. Scott Owen. Ray-sphere intersection. Online resource at: *http : //www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter1.htm*, June 1999.

[30] Bui Tuong Phong. Illumination for computer generated pictures. In *SIGGRAPH 75*. ACM, 1975.

[31] Pixar. Pixar renderman interface. Online resource at: *https : //renderman.pixar.com/products/rispec/index.htm*, November 2006.

[32] RTzen. Rt/shader ginza. Online resource at: *https : //www.rtzen.com*, March 2005.

[33] Softimage. Softimage xsi. Online resource at: *http : //www.softimage.com/*, March 2005.

[34] Kenneth E. Torrance and E. M. Sparrow. Theory for off-specular reflection from roughened surfaces. In *JOSA, Vol. 57 Nr. 9*. Optical Society of America, 1967.

[35] Bill Vinton. Shader related question. Online resource at: *http : //forum.unity3d.com/viewtopic.php?t* = 932, November 2005.

[36] Gregory J. Ward. Measuring and modeling anisotropic reflection. In *SIGGRAPH 92*. ACM, 1992.

[37] Alan Watt and Mark Watt. In *Advanced Animation and Rendering Techniques*. Addison-Wesley, 1992.