Visual Shader Programming

MARTIN FITGER

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2008
Visual Shader Programming

MARTIN FITGER

Master’s Thesis in Computer Science (30 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2008
Supervisor at CSC was Lars Kjelldahl
Examiner was Lars Kjelldahl

TRITA-CSC-E 2008:043
ISRN-KTH/CSC/E--08/043--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.csc.kth.se
Abstract

One of the greatest breakthroughs in the area of computer graphics over recent years has been the introduction of programmable hardware for rendering 3D graphics in the beginning of the new millennium. This technology has opened the door to a new world of possibilities for real-time rendering, where realistic effects that before only were possible in offline rendering areas such as movie production, are finally possible even in interactive real-time applications and games.

One big problem with this technology is that it is only accessible to experienced programmers. The hardware is controlled by writing special programs, called shaders, in C-like languages that can have a pretty high learning curve even for people with experience in other programming areas. Shading and effect design belongs to the area of artists, and it would be natural to let these people directly influence the design of shaders. Instead they are forced to communicate their visions to programmers which will in turn try to implement them. This process is not only inefficient but will also restrain the creativity of the artists. In order to solve these problems research has been done on more abstract methods of shader development, such as visual programming.

In this report we investigate the possibilities of applying visual programming to shader development. We cover the design, implementation and evaluation of a system for visual shader programming, and investigate possibilities for improving the usability of the system for users without programming experience. We also investigates how shaders created in the system can be integrated with third party applications in a general way (most existing systems for visual shader programming are designed to work with a specific game engine or rendering system), and show how visual shader programming actually can facilitate the integration of shaders with applications. We also discuss the possibilities of using the system as a tool for lowering the learning curve of shader programming.
Visuell shaderprogrammering

Sammanfattning

Ett av de största genombrotten som gjorts inom datorgrafik under de senaste åren har varit introduktionen av programmerbar hårdvara för 3D rendering. Denna teknologi har öppnat helt nya möjligheter för realtidsrendering – komplicerade och realistiska effekter som tidigare bara varit möjliga inom filmindustrin kan nu äntligen användas även i interaktiva realtidsapplikationer och datorspel.


I den här rapporten undersöks möjligheter med att använda visuell programmering för utveckling av shaders. Vi behandlar design, implementering och utvärdering av ett system för visuell shaderprogrammering, och undersöker även möjligheter för att underlätta för användare utan erfarenhet av shaderprogrammering att använda systemet. Vi undersöker också hur shaders som är skapade i systemet på ett så generellt sätt som möjligt kan integreras med andra applikationer (de flesta existerande system är anpassade för att fungera ihop med en viss spelmotor eller ett visst renderingssystem), och visar hur visuell shaderprogrammering faktiskt kan underlätta integreringen av shaders med andra system. Vi diskuterar också möjligheter för att använda systemet för att underlätta undervisning av shaderprogrammering.
# Contents

1 Introduction ....................................................................................1
   1.1 Background................................................................. 1
   1.2 The Problem............................................................... 2
   1.3 Goal ................................................................................. 2
   1.4 Chapter Overview ....................................................... 3

2 Theory ..............................................................................................5
   2.1 The Real-time 3D Rendering Pipeline................................. 5
      2.1.1 Overview ............................................................ 5
      2.1.2 Vertex vs. Pixel Operations.................................... 6
      2.1.3 Coordinate Spaces ................................................. 7
   2.2 3D Rendering Hardware .................................................... 9
      2.2.1 The Evolution of 3D Rendering Hardware ............... 9
      2.2.2 The Hardware Rendering Pipeline .........................10
   2.3 Shader Languages ..........................................................14
      2.3.1 Assembler ..........................................................14
      2.3.2 Cg .................................................................14
      2.3.3 HLSL .............................................................15
      2.3.4 GLSL .............................................................15

3 Related Work ................................................................................16
   3.1 Visual Programming .......................................................16
      3.1.1 Dataflow Programming .........................................16
      3.1.2 Other Applications of Visual Programming .............17
   3.2 Visual Shader Programming ...........................................18
      3.2.1 Shade Trees .......................................................18
      3.2.2 Building Block Shaders .......................................18
      3.2.3 An XML-Based Visual Shading Language .............19
      3.2.4 Abstract Shade Trees ..........................................20
      3.2.5 Interactive Shader Development ..........................20
      3.2.6 Commercial Tools ..............................................21
      3.2.7 Non Graph-based Tools .......................................22

4 Design Considerations ...................................................................23
   4.1 Overview ..........................................................................23
   4.2 Target Users ......................................................................23
   4.3 Abstraction .......................................................................24
   4.4 Application Integration ...................................................25
   4.5 Educational Aid .............................................................25
   4.6 Key Functionality .........................................................25
   4.7 System Requirements .....................................................26

5 System Design ..............................................................................27
   5.1 Overview ..........................................................................27
   5.2 Shader Graphs ...............................................................28
      5.2.1 Graph Structure ...................................................28
      5.2.2 Node Types .......................................................28
CHAPTER 1

Introduction

1.1 Background

One of the greatest breakthroughs in the area of computer graphics over recent years has been the introduction of programmable hardware for rendering 3D graphics in the beginning of the new millennium. Hardware accelerated rendering of 3D graphics has been available on consumer level since the mid 1990’s, but the early solutions were inflexible and offered only a set of fixed lighting models and effects which left little room for creativity. Because of the lack of generic lighting models and processing power the images produced by these early solutions often had a “synthetic” look.

The programmable hardware makes it possible to create arbitrary lighting and material effects by writing programs called shaders that can be executed directly by the hardware. This technology has opened the door to a new world of possibilities for real-time rendering, where realistic effects that before only were possible in offline rendering areas such as movie production are now possible in interactive real-time applications and games.

The flexibility of the new technology comes with a price though, and that is that you need good programming skills to be able to write the shaders. In the beginning these programs had to be written in an assembler language, but with time several easier and more flexible C-like languages have been introduced. These make it possible to write code that is independent of the underlying hardware, but you still need to be an experienced programmer to be able to write shaders in these languages.

It is a big problem that a programmer is needed for writing shaders because shaders have a big impact on the final appearance of art in for example games, and it would therefore be preferable if the artists themselves could influence their design. Instead the artists have to try to explain their visions to programmers who will then try to implement them, which will both impede the overall workflow and restrain the creativity of the artists. Another problem with the shader programming technology is that there is no standardized way of organizing and reusing shader code. In typical applications variations of a shader is often needed either for different environmental or lighting conditions, variations in complexity, or a combination of both. In bigger projects this leads to problems with code duplication and maintenance of a vast number of variations of shader programs.

In order to solve these problems research has been done on new more abstract methods of shader development. The ease of visual programming and flexibility of dataflow programming has led to the idea of representing shaders as graphs, where nodes represent predefined shader specific operations, and where edges represent flow of data between these nodes/operations.
1.2 The Problem

Visual and abstract ways of designing shaders is an area that is under active research. Variations of tools with differing levels of abstraction have been proposed, but they are still more or less in the experimental stage. Some promising projects have existed but have been bought up by bigger companies to be used as in-house solutions or have been shut down. Some tools based on this technique are available as add-ons for some commercial rendering engines, but these are specialized to work with their respective engines and would be difficult to use with other systems. A general tool that is not dependant on a specific engine would be much more valuable, since it could be used for general shader development and could be reused for several engines.

In an interview with the lead rendering programmer of a big computer game company in Sweden the need for giving artists more influence over shader development was confirmed. Some issues with general shader development tools also became clear, such as problems with integrating such a tool with already existing development pipelines and applications.

The purpose of this project is to explore the possibilities and problems with visual shader programming, and to try to find ways of improving the work that has been done in the area so far. Especially, possibilities of using visual shader programming to make it possible for users without shader programming experience to develop shaders, and ways of making it possible to integrate the generated shaders with third party applications, will be investigated.

The institution CSC at KTH (The Royal Institute of Technology), for which I am making this thesis, is also interested in finding out if visual shader programming can be used as a means of making the process of learning shader development easier, since the learning curve of shader programming is quite high.

1.3 Goal

To be able to thoroughly investigate the possibilities of visual shader programming and to try different ideas that might improve on earlier work, we will create an own system for visual shader development. The system is not meant to be a complete and finished application but rather a framework in which new ideas can be tested, and which can be used to test the technology on actual users. By implementing a real system we will also be able to identify various practical issues that might otherwise go unnoticed if the work would be purely theoretical.

The focus will be on trying to find ways of creating a general stand-alone tool for shader development, because of the need for and lack of such tools. For such a tool to be useful, the possibilities of facilitating the integration of the shaders with third party applications (rendering engines) will also have to be explored. As explained in the previous section, we will also investigate and implement ideas that can make it possible for users without shader programming experience to use the system, and that can make it possible to use the system as an aid in learning shader programming.

To get feedback on various aspects of the system, it will be tested on a group of students as part of a laboratory assignment on a course in computer graphics. This will be a perfect opportunity to test if users without shader programming experience can use the system (since not all students have experience in shader programming), and to evaluate the
educational aspects of the system. If the testing goes well, it is possible that the system will be used in other courses that teach shader programming in the future.

1.4 Chapter Overview
We will here give a quick overview of the remaining chapters of this report.

2. Theory
This chapter will present background theory that is essential for understanding the work in this thesis. The first section will give a quick overview of the real-time 3D rendering process and will serve to set a context for the work and explain where it fits into the rendering process. The second section is dedicated to 3D rendering hardware, which is the main target of the work, and will serve to show which opportunities and limitations we have to work with.

3. Related Work
In this chapter related work that has been done in the area and that is relevant to the work of this thesis will be presented. We will start by presenting work related to visual programming in general, and will continue with work about applying visual programming to shader development.

4. Design Considerations
In this chapter we will start planning the development of our own system for shader graph development. We will analyze the previous work presented in the chapter 3 and to try to identify problems and ways of improving earlier techniques. We will then use this information and combine it with own ideas to form the design ideas of the system, which is covered in the following chapter.

5. System Design
This chapter will present the design of our visual shader development system. We will start by giving an overview of the system design, and will then describe the heart of the system, the graph structure. We will then present the user interface, and will finally cover special features that have been implemented for improving usability and facilitating integration of shaders with third party applications etc.

6. Implementation
This chapter will cover the implementation of the system. We will describe what tools have been used and how various parts of the system have been implemented. The largest part of this chapter will be dedicated to the compilation process. We will describe each step on the way, from graph to final shader program.

7. The Resulting System
This chapter will give a demonstration of the resulting system. We will start by showing how the system is used by going through some usage scenarios, and will then show some examples of shaders that have been created in the system.

8. Evaluation
This chapter will present a testing of the system that was done as a laboratory assignment for a course in computer graphics. We will cover the background knowledge of the users, the specifics of the assignment and will present the results of an evaluation form that was handed out after the test.
9. Discussion
This chapter will discuss various aspects of the system and its design to see whether our goals have been met, what possibilities and limitations the system has, possible performance issues etc.

10. Conclusions
The final chapter of this report will present conclusions and propose ideas for future research.
2.1 The Real-time 3D Rendering Pipeline

Real-time rendering is a vast and complex area (Akenine-Möller, 2002) and details on the subject are outside the scope of this thesis. This section is just meant to give a very brief overview of the real-time 3D rendering process and to highlight some key aspects that are relevant to the work in this thesis.

2.1.1 Overview

The core of a 3D rendering system is often referred to as the rendering pipeline, and its main purpose is to take a 3D description of a scene and generate a 2D image representation of it. The process is relatively complex and is comprised of several different sub-tasks that handle different computational stages. It is called the rendering pipeline because the architecture resembles a real pipeline in the sense that data from one stage cannot be passed on for processing to a successive stage until the latter has processed its current data. This also implies that the speed of the system is constrained by the slowest stage in the pipeline, often called the bottleneck. Many of the individual stages in the 3D rendering pipeline are often characterized by high parallelism – these can be optimized by adding parallel processing, something that is often exploited in the design of today’s hardware implementations.

The pipeline can be coarsely divided into three separate conceptual stages – application, geometry and rasterization. These stages are in turn comprised of functional sub-stages, where some might be possible to execute in parallel and others forced to be executed in sequence because of dependencies.

![Fig 2.1 The three conceptual stages of the Real-time 3D rendering pipeline.](image)

The Application Stage

The tasks handled by the application stage of the pipeline vary between different kinds of applications, but common tasks are user input, collision detection and response, animation via transforms etc. The product of this stage from the pipeline’s point of view is a
description of the geometry to be rendered along with transformation matrices, material properties, light information etc. It is often just a small part of a scene that is visible through the view for rendering a given frame, which means that just a small part of the scene geometry needs to be passed on from this stage. The amount of non-visible geometry passed on should for efficiency reasons be kept at a minimum, and calculating the set of visible geometry is therefore an important task that is handled in this stage.

The Geometry Stage
The tasks performed in the geometry stage are operations that need to be performed per vertex or per primitive. Common tasks handled here are coordinate transformations (see 2.1.3 Coordinate Spaces for more information) and lighting calculations (can instead be performed per pixel later in the pipeline, see 2.1.2 Vertex vs. Pixel Operations). After all necessary per-vertex operations have been carried out the coordinates are projected, primitives (lines or triangles) are assembled from individual vertices, the primitives are clipped against the view frustum, and the resulting coordinates are finally transformed into screen space.

The Rasterizer Stage
The goal of the rasterizer stage is to assign the correct colors to each individual pixel during rendering. As opposed to the per-vertex operations performed in the geometry stage, the operations handled by this stage are carried out per pixel. The primitives (lines, polygons etc) defined by the two-dimensional screen space coordinates outputted by the previous stage are first mapped to actual pixels through a process called rasterization or scan conversion. Then, additional data such as colors, texture coordinates, light intensity etc. that come with each coordinate are interpolated over the primitive so that each pixel get its own set of values. These values are finally used to determine the color of each pixel through per-pixel operations such as texture lookups, blending or even lighting calculations (lighting calculations can be carried out either in the geometry stage, pixel stage or both depending on the required level of detail, see 2.1.2 Vertex vs. Pixel Operations). Some effects such as semi-transparent materials also require reading color values from the screen (frame buffer) and blending these with the calculated pixel color before finally writing to the pixels.

The work in this thesis mainly addresses the geometry and rasterizer stage. A more detailed pipeline with functional stages for these two stages is shown in figure 2.2.

2.1.2 Vertex vs. Pixel Operations
An important aspect of the real-time rendering pipeline, which has a big impact on both performance and the quality of the rendered image, is to decide which operations to perform during vertex and pixel processing respectively. Some operations naturally fall into
one of the two categories. Vertex transformations can for example only be performed during vertex processing, while texture lookups and color blending operations are performed per pixel. Other calculations involving lighting etc. can be performed in either one. It is generally better performance wise to perform as many of the calculations as possible per vertex since pixel processing is normally performed more frequently, but moving calculations from vertex processing to pixel processing can result in higher image quality and realism.

Gouraud shading and Phong shading are examples of two common shading methods that are based on vertex processing and pixel processing respectively (Phong, 1975). In Gouraud shading the lighting calculations are performed per polygon vertex, and then the resulting color at each vertex is linearly interpolated over the pixels of the polygon. In Phong shading the vertex normals of each polygon vertex is interpolated over the pixels of the polygon, and is then used in lighting calculations performed at each pixel. Naturally, Phong shading will need more processing power than the Gouraud Shading, but it will also produce much better visual quality than the latter. Sub-polygon details like highlights located within a polygon will for example “disappear” in the Gouraud model because of the linear interpolation of colors obtained at the vertices of the polygon (see figure 2.3). Normal mapping is an example of another shading method that uses per-pixel processing to add sub-polygon details. In this method, instead of using the interpolated vertex normals in the lighting calculations, the surface normal for each pixel fragment is obtained by sampling a special texture map that stores surface normals encoded as colors (see figure 2.3).

Fig 2.3 From left to right: Gouraud Shading (per-vertex lighting), Phong Shading (per-pixel lighting) and Normal mapping (per-pixel lighting).

2.1.3 Coordinate Spaces
As the geometry passes through the pipeline it is transformed between different coordinate spaces. Operations that involve several objects need to be carried out in a coordinate space that is common for all the objects, while other calculations need to be performed early in the pipeline where a common coordinate space is not yet defined. Yet other calculations might benefit performance wise from being performed in a special coordinate space in which calculations are being simplified compared to other spaces. Here follows a short overview of commonly used coordinate spaces:

Object Space
The geometry of the different objects in a scene is often originally specified in separate local coordinate systems individual to each object. The origin of each coordinate system is
placed at a reference point convenient to the object at hand, for example centered between the feet of a character or in the rotation center of a mechanical arm etc.

**World Space**

When objects are put together to form a scene a common coordinate space has to be used, which is often referred to as *World Space* (see figure 2.4). The geometry of each object is positioned in World Space through a series of translation, rotation and scaling transformations collectively called a *World transformation*. This technique also makes it possible to use several instances of an object in a scene by just supplying a unique world transformation for each instance.

**Eye Space**

In order to render a scene we have to specify a point and direction from which to view the scene - an imagined eye or camera that is placed at a specific position and looking in a specified direction. To simplify the projection of a scene, all geometry is first transformed into *Eye Space* which is a coordinate space relative to the position and orientation of the camera or eye from which to view the scene (see figure 2.4). The origin of the *Eye Space* coordinate system is placed at the position of the eye; with the z-axis pointing in or towards the view direction (depending on if a left-handed or right-handed coordinate system is used).

**Clip Space and Screen Space**

To achieve the phenomenon of objects being close to the eye appearing bigger than objects being far from the eye, a perspective transformation (involving division of the z-coordinate) from Eye Space to a space called *Clip Space* is performed. The coordinate space is called Clip Space because it has properties that facilitate removing (clipping) the parts of the geometry that are not visible (located outside the view frustrum). Prior to rendering, a transformation from Clip Space to *Screen Space* is made. Screen Space is the 2D coordinate system used for referencing actual pixels on the screen.

**Tangent Space**

*Tangent Space* is a special coordinate space that is used in the calculations for various surface effects like *Normal mapping* (see figure 2.3) and *Parallax mapping* (Kaneko et al., 2005). The coordinate space is defined on the surface of an object (see figure 2.4), with the y-axis of the coordinate space pointing along the surface normal and the x and z-axes tangential to the surface.

*Fig 2.4* 2D view of coordinate spaces commonly used in real-time rendering.
2.2 3D Rendering Hardware

The purpose of 3D rendering hardware is to offload the CPU of work related to the 3D rendering pipeline. The first available hardware solutions could only offload a relatively small part of the pipeline, but with the evolution the hardware has been able to handle more and more parts of the pipeline, thus leaving the CPU free to use more processing power on other tasks.

2.2.1 The Evolution of 3D Rendering Hardware

The evolution of 3D rendering hardware has gone very fast compared to that of other computer hardware (Comba, 2003), and has far surpassed Moore’s Law. This can be explained partly by the big demand from the entertainment industry. Video game sales have in fact exceeded the film industry’s annual box office. Another explanation is that the task of rendering computer images is highly parallelizable, where for example much of the work done for each pixel can be handled independent of other pixels. This makes it easy to increase the power of a chipset by just adding more parallel processing power.

The first publicly available consumer level 3D hardware accelerated graphics cards appeared on the market in the mid 90’s. Their capacity was limited but it was still a big breakthrough for the computer graphics technology that meant that the era of 3D applications as pure software solutions was over and true real-time 3D applications and games were finally made a reality. It should be noted that special graphics hardware already had been available before as integrated hardware-software solutions by companies such as Silicon Graphics Inc. These were the first hardware solutions for handling vertex transformation and texture mapping etc. but were by no means available for the general public because of their high costs.

The evolution of 3D rendering hardware can be grouped into four generations where each generation marks a significant change in hardware rendering technology (Comba, 2003).

The first generation of 3D rendering hardware was around during the later half of the 90’s. These chips were able to offload the general CPU of the rasterization part of the 3D pipeline, but transformation and lighting calculations still had to be done by the CPU. The hardware basically supported texturing and Gouraud shading, and some chips also supported multitexturing which is the process of blending two or more textures together using one of a fixed set of available blending functions. The typical chipsets from this generation were the NVIDIA TNT, ATI Rage and the 3Dfx Voodoo series, and had around 10 million transistors.

The second generation of 3D rendering chipsets was introduced late in 1999, and was in addition to rasterization also able to handle transformation and lighting calculations which meant that even that part of the 3D pipeline could be offloaded from the general CPU. Some of these cards also supported storing geometry data in their own on-chip memory which meant that it was only necessary to transfer geometry data once to the card (instead of for each frame). The vertices were uploaded as coordinates in object space, and chipsets would then transform the vertices for each frame and all they needed to do this was an update of transformation matrices from the CPU, which meant that traffic between the CPU and the GPU could be lowered drastically. The chips also offered an extended set of fixed functions for blending textures and coloring pixels such as signed math operations along with support for new features like cube map textures. Main chipsets from this
generation were NVIDIA GeForce 256, ATI Radeon 7500 and S3 Savage3D, with a transistor count of around 25 million.

The third generation of rendering hardware was introduced 2001, and finally opened the door to programmability. These chipsets offered a programming model for vertex computations instead of just extending the set of fixed configurations. The vertex programs were limited to a maximum of 128 instructions and 96 parameters and had no access to textures. A very limited form of programming per-pixel operations was also introduced, with an instruction count limit of around 12 instructions and with further restrictions on the instructions used such as inability of dependant texture accesses etc. Program flow control such as branching was not supported. Typical chipsets from this generation were NVIDIA GeForce3 and Geforce4 and ATI Radeon 8500. The transistor count was around 60 million.

Today’s generation of graphics cards offers a much more generalized form of programming both vertex and fragment shaders, and in addition to these two the newest chipsets also offer a completely new type of programmable processing, called geometry shaders. The geometry shaders are executed in between vertex and fragment processing, directly after or as an extension to the primitive assembly stage, and are used to generate new primitives. Processing units can be freely allocated for any of the three processing types to balance the processing power for a specific type of rendering load. Texture accessing is allowed in all program types and there are no restrictions on dependant texture access. There is practically no longer a limit of the instruction count for either program type or for parameter count. An example of a modern chipset is the NVIDIA GeForce 8 series, which offers a limit of 65536 instructions per program and 16x4096 parameters. The transistor count of a typical chipset from this series (GeForce 8800 GTX) is 681 million.

2.2.2 The Hardware Rendering Pipeline

The parts of the 3D rendering pipeline that today’s hardware can off-load from the CPU is the geometry and rasterizer stages. The structure of the pipeline is still very much the same as the general description in section 2.1, with the difference that the last two conceptual stages of the pipeline has been realized by an actual physical pipeline system implemented in hardware.

Geometry data is sent from the application to the hardware in the form of vertex data (position coordinates, color, texture coordinates etc) and primitive data (an indexed ordering of the vertices to form primitives like lines or triangles). The vertices then goes through a vertex processing stage that handles all per-vertex operations and finally outputs the vertices transformed into clipping space. After the vertex processing, primitives are assembled from the transformed vertices and are then passed on to the geometry processing stage. During geometry processing new vertices can be generated and vertices are connected to form primitives like lines or triangles. Next, the primitives are clipped, culled and their coordinates projected and finally transformed into screen space. The rasterizer translates the primitives into actual pixel fragments and interpolate vertex data over the primitive so that each fragment get its own set of texture coordinates etc. Each fragment from the rasterizing stage then goes through a fragment processing stage that handles all per-fragment operations and finally outputs color and depth information for the fragment (a pixel). The depth information is then used for visibility testing such as the depth buffer test. If the fragment passes the visibility test the pixel of the respective fragment is finally shaded with the color information from the fragment processing stage.
The old rendering hardware based on the Fixed Function Pipeline structure offered no geometry processing (primitives were created directly from the ordered list of vertices that entered the pipeline) and a very limited functionality in the vertex and fragment processing stages, with a fixed set of functions that could be activated or deactivated from the application, and texture access was not available during vertex processing.

In the programmable pipeline of today’s hardware the vertex and fragment processing stages have been replaced by programmable vertex and fragment processors respectively that can execute generic programs called shaders, and texture access is now also supported during vertex processing. In addition, a new processing stage called geometry processing has been introduced, which is able to generate new primitives from the ones delivered in the pipeline. The rest of the pipeline practically remains the same, which means that clipping, culling, projection, frame buffer operations etc. still operate in a fixed set of modes that are controllable from the application (see figure 2.5).

The programmability not only offers complete freedom over effect design (as opposed to the old fixed set of functions), but also takes the quality and amount of detail possible to a new level due to the introduction of pixel processing control. Lighting and other effects can now be calculated for each pixel individually instead of interpolating the result of vertex computations. The geometry processing stage also makes it possible to for example tessellate geometry (e.g. generate primitives to approximate curved surfaces etc.) in hardware. The generalized form of programming that the new architecture offers even opens new opportunities for using the hardware for generic computations, which can be especially beneficial for highly parallelizable computations such as physics simulation.

Geometry processing is a new feature, which is not yet supported by all common high level shading languages (it is currently only supported in the languages Cg and HLSL, as of DirectX 10). The work in this thesis focuses solely on vertex and fragment processing, and we will therefore only concentrate on these two for the remainder of this chapter.

The architecture of the programmable pipeline is designed with high parallelizability in mind, so that vendors will be able to produce faster graphics hardware by adding more parallel processing power for each new generation of hardware. This is reflected in some restrictions on the operations possible in the programming model. It is for example not possible to access information about other vertices during vertex processing of a given vertex or information about other fragments than the current during fragment processing.
**Vertex Programs**

A vertex program is executed once for each vertex in the geometry. The programs can be specified as a completely general sequence of operations, and current hardware supports general purpose programmability, so the vertex programs can perform a wide range of general computations. The most important task of the vertex programs is to handle transformation of vertex coordinates, but they can be used for a multitude of different computations and purposes. Some of the tasks commonly handled by vertex programs are vertex transformation, normal transformation and normalization, texture coordinate generations and transformation, lighting calculations etc.

There are three different sources of input to the vertex programs:

- **Vertex attributes**
- **Uniforms**
- **Samplers**

**Vertex attributes** are values associated with each vertex in the geometry (see figure 2.6). Examples of commonly used vertex attributes are position coordinates (usually in object space), normal vector and texture coordinates.

**Uniforms** (application constants) are a limited set of registers with read-only access from within the vertex program. These registers can be set from within the application and are meant to be used for storing parameters that remain constant over the course of rendering a set of primitives. Common use of these register includes storing of transformation matrices, parameters specific to the current object being rendered like material constants, and also scene parameters like light information, fog density etc.

The vertex programs also have read-only access to texture memory. Texture memory is accessed with special registers called **Samplers**, which are bound to a specific texture by the application before executing a shader. Access to the texture memory, which was originally meant for storing images, has more obvious usage in fragment programs, but texture memory can be used for storing virtually any kind of information as long as it can be encoded as an image. Curves and functions can for example be stored as one-dimensional images by storing amplitude as light intensity, where each pixel in the image represent a sampling point on the curve/function. Evaluating functions by just making a texture sample in the vertex program can be much more efficient than actually evaluating the function in the program, especially for complex functions.

---

**Fig 2.6** The input and output of the Vertex and Fragment Processors.
When the vertex program is done it will always output the coordinates of its vertex. The coordinates are then passed on to fixed processes in the hardware that handle perspective division, viewport mapping and operations that require knowledge of several vertices at a time like primitive assembly, clipping etc. In addition to vertex coordinates the vertex program can also generate other output values like a transformed normal, computed light intensity etc that upon program termination are passed on to the rasterization stage of the pipeline.

**Fragment Programs**

A fragment program is executed for each fragment (pixel) that is rendered. The architecture of the vertex and fragment processors is generally the same. The instructions available for the fragment programs and the level of general purpose programmability are therefore generally the same as for the fragment vertex programs.

The purpose of fragment programs is to calculate color and depth for each fragment. These values are then passed on to fixed processes that handle depth buffer comparison and fragment buffer blending and updating. Common tasks performed in a fragment program are texture application, color blending (screen blending is not supported), lighting calculations and fog calculations.

There are three different sources of input to the fragment programs:

- *Varying variables*
- *Uniforms*
- *Samplers*

The resulting output from each processed vertex in a triangle to be rendered is interpolated during the rasterization stage of the pipeline so that each fragment in the triangle gets its own set of interpolated values, called *Varying variables*. This set of per-fragment-interpolated values is available with read-only access from within the fragment program. Examples of typical values that can be interpolated are position coordinates, normal, light intensity and color.

*Uniforms* are the same that are available to vertex programs (see description above).

As the vertex programs, the fragment programs also have access to texture memory, through *Samplers*. Textures can have several different applications in fragment programs. The most common usage of texture sampling is for the effect of wrapping images over 3D objects by using interpolated texture coordinates for sampling colors from a texture image. Another common usage of texture memory is for storing normal vector perturbations in an image, called a normal map. The x, y and z components of a normal in a spot on the image can be stored as red, green and blue intensity respectively for the corresponding pixel. By sampling these textures a normal can be obtained for each fragment and can be used in subsequent lighting calculations to simulate details in the surface of the rendered object.
2.3 Shader Languages

With the new technology of programmable 3D rendering hardware, the need of means for describing the programs follow. This has given rise to several issues, and the way of programming the GPUs has come a long way since the introduction of the first programmable 3D rendering hardware.

2.3.1 Assembler

In the beginning the only way of programming the GPUs was by using assembler which is a low-level hardware dependant language. The well known issues of using assembler are the same for GPU programming as for any other kind of hardware. Assembler is cumbersome and time consuming to use and one of the biggest problems with using assembler is that different hardware has different instruction sets, which means that a program written for a specific GPU might not necessarily work for another. The advantage of using assembler over any other language though is that it gives total control of the hardware and therefore makes it possible to optimize code down to every instruction to be executed.

In an effort to standardize the programming model, Microsoft introduced several virtual shader machines with their release of DirectX 8. Each of these machines roughly resembled different kinds of hardware architectures at that time, and more were introduced as new architectures were released. Programs still had to be written with an assembly language, but had greater hardware independency since they were written for these virtual machines. It was then up to the GPU vendors to implement drivers that could translate the virtual machine code into their own hardware specific machine code.

Once the technology evolves though, new and more effective instructions will be available, and to take advantage of these the old GPU programs must be rewritten. The task of writing optimized assembler code also grows more and more complex as instruction sets grow richer and new hardware features cause efficiency to be dependant on the ordering and combinations of some instructions etc. All these issues with the assembly language created a great demand for easier ways of programming the GPUs and for better hardware independency and compiler automated optimizing.

The solution to the problems with programming shaders in assembler have been to introduce high-level languages, which offer hardware independence, simplicity, improved readability and the possibility of compiler optimization. Several solutions have been proposed with different advantages.

2.3.2 Cg

Cg is a high-level shading language for GPU programming that is developed by the NVIDIA Corporation (NVIDIA Corporation, Cg 2.0) (Fernando, 2003). Cg stands for “C for graphics” and as the name implies it is very similar to the C programming language, but has also been influenced by earlier shading languages like RenderMan (Upstill, 1990) and has adopted some ideas of modern languages like C++ and Java as well. The language has features that make it easy to write programs to be compiled to highly optimized GPU code, such as support for native data types for vectors and matrices, as well as built-in functions for context-specific operations such as matrix multiplication, dot product etc. which are often supported directly by the hardware. With Cg, the issue of differing hardware architectures is solved by offering different profiles in the Cg compiler that allow
for generation of code for different hardware profiles. The same GPU program can then be compiled into executable code for different hardware architectures. One problem with this solution is that an application will have to be shipped with several variations of each GPU program, one for each architecture type it will support.

2.3.3 HLSL
With the release of DirectX 9, Microsoft Corporation introduced a shader language called HLSL (Microsoft Corporation, HLSL) (St-Laurent, 2005). The language was created in cooperation with NVIDIA and is practically identical to Cg. But the way in which it is implemented is quite different. Instead of offering different hardware profiles during the compilation, the HLSL code is compiled into an intermediate and general binary representation than can be stored and later sent to a vendor driver for translation into hardware specific code before executing. In this way shaders can be compiled in advance and only one binary representation of each GPU program needs to be shipped with an application because it gets translated to executable code on the target machine. One problem with this technique could be that structural information in the program can be lost during the compilation of the HLSL program into the intermediate assembler-like representation, that could otherwise be used for further optimization and utilization of specific instructions in a given hardware architecture if the compilation would be made directly from HLSL into the final executable form.

2.3.4 GLSL
3DLabs has proposed another high-level shading language called GLSL (OpenGL.org, OpenGL Shading Language) (Rost, 2006), to be used by applications written for the OpenGL API. The language is very similar to Cg and HLSL but also has several features that facilitate integration with other parts of the OpenGL API. The OpenGL implementation of GLSL differs from the HLSL implementation in that there is no intermediate storage of shaders, but shaders are always compiled directly from GLSL to specific hardware architecture by vendor drivers on the target machine. This means more work for the driver developers, but 3DLabs tries to facilitate the process by distributing a free GLSL parser. This implementation of the compilation process makes it possible for even more optimization than if an intermediate format would have been used as in HLSL, but for the price of a more expensive compilation process each time a new shader is prepared for usage.
CHAPTER 3

Related Work

3.1 Visual Programming

As opposed to a textual programming language, a visual programming language (VPL) lets the user specify programs by manipulating program elements in a graphical interface. The logic and flow of a program can for example be defined with different visual expressions, spatial arrangements of visual components, graphical symbols defining relationships and communication between different program elements etc. Visual programming has several application areas: general-purpose programming, scientific visualization languages, database programming, image processing, user-interface generation and more.

3.1.1 Dataflow Programming

One interesting and very successful area of application of visual programming is in the area of dataflow programming. In dataflow programming languages, data is promoted to become the main concept behind a program, as opposed to the majority of programming languages which uses the imperative programming model, where the focus is on describing a program as a series of operations which are connected by dataflow channels. In dataflow programming each operation is considered a “black box”, carrying out its task as soon as all of its inputs are ready, as opposed to the imperative programming model where the instructions are clearly specified and executed as they are reached in a linear ordered fashion. One of the main advantages with the dataflow programming model is that parallel processing comes very natural, if the input of several operations is ready at the same time, the operations can be performed simultaneously.

One problem with the non-linear fashion of dataflow programs is that it makes them difficult to express through textual means. It is much easier to get an overview and understand the dependencies of a program if it is presented visually as a graph, where operations are represented by nodes, and edges specify dataflow between the operations. This makes these languages perfect candidates for using visual programming interfaces.

In the beginning, graphs were only used as a static means for viewing dataflow programs that were already designed textually, and were only meant as an aid for the scheduling of the program on a dataflow machine. In the 1970s a man named Davis proposed what is arguably the first visual dataflow language, called Data-Driven Nets (DDN) (Davis, 1974). In DDN a program was represented by a cyclic dataflow graph, with data items flowing along the arcs which represented FIFO queues. The system was fairly simple and was not intended to be used for directly programming in. Since then the technology has evolved (Johnston, 2004), and today there are many dataflow visual programming languages (DFVPLs) available. LabView (National Instruments, NI – LabView), or Laboratory Virtual Instrumentation Engineering Workbench, is a well-known DFVPL which was developed in 1986 and is used for data analysis in laboratories. LabView has a library of
predefined functions that the user can use. To create a program, the user selects some functions which appear as boxes in the user interface, and can then specify dataflow between these functions by connecting the boxes with arcs. Another example is NL (Harvey, 1993) which is a general-purpose language which is fully based on the dataflow execution model. The system has several interesting features for flow control, and comes with a supported programming environment and a visual debugger.

### 3.1.2 Other Applications of Visual Programming

OpenDX (OpenDX.org, OpenDX Home Page) (Thompson, 2004) is a general data visualization environment which was developed in the beginning of the 1990s (see figure 3.1). The system has three visual programming support components for creating data visualizations. The first programming component is a graphical program editor which lets the user create visual programs by selecting program components from a component library, designating the order of their application and supplying parameters values that the components require. Everything is done with a point and click interface where components are placed on a canvas and then linked together to specify data communications between them. The second programming interface allows the user to implement any type of data transformations, and encapsulate it as a module with specific inputs, user defined parameters, and outputs that will be generated by the implemented function. The module can then be used in combination with other modules to create data visualizations. The third programming component gives the user control over the calculations performed by the visualization programs, based on a dataflow driven client-server execution model. This facility allows the user to trace data flow during program execution, and allows the visualization to be divided into sub-components that can be distributed over several workstations or multiple processing elements of super computers.

![Fig 3.1 Screenshots of OpenDX™ (from http://www.opendx.org)](image)

AVS (AVS, AVS Software for Developers) is an example of another comprehensive and versatile data visualization tool that was introduced in 1989. The system has evolved since then and is today used by both non-programmers and experienced developers, for analyzing and presenting scientific data in areas like medical, telecommunications and environmental research.

Visual programming tools are common in the offline rendering industry. Content creation applications like 3D Studio Max (Autodesk, Autodesk 3ds Max) and Maya (Autodesk, Autodesk Maya) use graph-based visual programming interfaces in various parts of their product suites. Hypershade is a part of Maya that allows the user to design materials and effects with a graph-based user interface. The interface lets the user select from a predefined set of nodes and connect them in a network to describe an effect.
3.2 Visual Shader Programming

3.2.1 Shade Trees
The earliest work that has been done on representing graphical effects with graphs was presented as early as in 1984 in the Siggraph paper Shade Trees (Cook, 1984).

The trend at the time was to use fixed models of light reflection to which all surfaces in a scene must conform. The models could be parameterized with different appearance parameters but the shading equation itself remained fixed. This made very little allowance for the extremely diverse ways in which objects interact with light in a scene. As opposed to this old technique Cook’s report describes a new modular approach based on the key observation that no single shading model is appropriate for describing all surfaces. For some surfaces it might be sufficient with simple models while other might need relatively complex calculations for describing them accurately.

Cook proposes a language for describing surfaces with a set of basic operations, such as vector normalization and dot product, and with the syntactic means for ordering these operations in a tree (see figure 3.2). Each node in the tree represents an operation, has zero or more appearance parameters as input and produces one or more appearance parameters as output. The output from one node can then be linked as input to another node, to express that the result of the operation represented by the first node should be used as input to the operation represented by the second node.

![Fig 3.2 Shade tree for copper (from Shade Trees, Cook 1984)](image)

For example a “diffuse” node, with the purpose of calculating diffuse light reflection, could have a surface normal and a light vector as input and an intensity value as output. In addition to shade trees that describe surfaces Cook also proposes similar trees for describing lighting and atmospherical effects respectively. Shaders from each category can then be combined to describe a scene. This will make the system even more flexible as for example one light tree can be grafted onto several different shade trees.

3.2.2 Building Block Shaders
In the report Building block shaders (Abraham, 1990), Abram et al. continue the work of Cook by proposing a graphical user interface for the creation of shade trees, as opposed to the pure syntactical approach proposed by Cook.

The proposed system had one high level interface and one low level interface. The high level interface consisted of a graphical editor which allowed the users to construct complex
shaders by connecting shading elements in a network. In the low level interface, shading elements were programmed in a standard programming language and compiled into modules that could be linked either at runtime or compile time. Each shading element were declared with a set of tabs that defined points of connection, or entry points of the module, and networks of the elements were constructed by linking tabs from one element with tabs from another. In effect each link in the shading network represented a subroutine call, and the execution of the network was analogous to the execution of an interpreted language.

The system had support for automatic type checking and offered a preview window in which the resulting effect could be seen applied to a sphere. Although the system was based on standard programming languages for designing its shader elements and its shader networks were interpreted rather than compiled, it was still a big step in the evolution of graphical shader graph editors.

3.2.3 An XML-Based Visual Shading Language

F. Goetz et al. presents a graph based system for developing effects for vertex and fragment shaders in the paper An XML-based Visual Shading Language for Vertex and Fragment Shaders (Goetz, 2004). The system is a natural extension of the ideas presented in the report Building block shaders by Abram et al, adapted to today’s 3D hardware technology.

The system allows the user to define shaders through the creation of dataflow diagrams in a graphical user interface similar to the one presented earlier by Abram et al. The system seems to be geared towards web graphics as it outputs shaders in an XML format for the possibility of future integration into XML based 3D formats like Extensible 3D (X3D).

The dataflow diagrams in the system consist of nodes that are functional units and edges that represent data that flows from output variables to input variables of each node. There are six different kinds of nodes that can be used in the system, as shown in table 3.1.

<table>
<thead>
<tr>
<th>Node Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator</td>
<td>Generates output for one variable. Corresponds to Cg operators.</td>
</tr>
<tr>
<td>Function</td>
<td>Several input, one output. Corresponds to Cg lightweight functions.</td>
</tr>
<tr>
<td>Constant</td>
<td>No input. One output that delivers constant value.</td>
</tr>
<tr>
<td>Diagram</td>
<td>Encapsulate a diagram of nodes into one node.</td>
</tr>
<tr>
<td>OpenGL State</td>
<td>Gives access to OpenGL program variables and vertex attributes.</td>
</tr>
<tr>
<td>Constructor</td>
<td>Used to combine vector components to build a new vector.</td>
</tr>
</tbody>
</table>

Table 3.1 Node types in the XML-Based Visual Shading Language.

The dataflow diagrams of the created vertex and shader programs are exported to XML files or DOM trees. These can later be transformed into the Cg programming language by using an Extensive Stylesheet Language Transformation (XSLT) processor.

The abstraction level of the system is relatively low. Nodes represent more or less atomic operations as they map one-to-one with functions in the Cg language (see 2.3.2 Cg). The system performs type checking but does not offer correction of types by for example automatic type conversion, nor does it aid in automatic conversion between spaces etc.
3.2.4 Abstract Shade Trees

In contrast to the work presented by Goetz et al. a more abstract approach was presented in the paper Abstract Shade Trees (McGuire, 2006). This paper proposes an abstract system that hides all programming related information from the user.

The system allows the user to specify dataflow between abstract modules by dragging and clicking connections from one module to the other. The difference from earlier systems is that at most one connection is allowed between two modules, instead of connecting specific outputs from one module with an input of another module. In effect this just allows the user to specify that there is a dataflow between two modules, but no details on the dataflow itself can be specified (see figure 3.3).

![Fig 3.3 Examples of Abstract Shade Trees (from Abstract Shade Trees, McGuire 2006)](image)

Each module in the system is defined with a set of input and output components, although these are not shown to the user. These components are declared with a rich set of types that describe semantics, dimensionality, transformation space, length (normalized or not) etc. When the user specifies a dataflow between two components the system will automatically try to match each output component from the first module with an input component of the second module. The matching is done by comparing the type declarations of the input and output components respectively, to find and connect the ones that are closest semantically. During the matching of the components the system handles conversion between spaces, data types etc. automatically by using a table of predefined conversion modules from one type to another. It will sometimes be necessary to traverse the conversion table more than once and therefore perform conversions in several steps before a matching between two components with differing declarations can be made. Ultimately the components that are closest to each other semantically will be connected. In order to avoid matching components that differ too much in their declarations a limit for the number of table iterations (search depth) is set.

The system generates code in the GLSL programming language from the diagram that the user creates. The diagrams are much more compact in this system than in earlier systems as more details are hidden, but this might also makes them more difficult to understand as there is no way of knowing what real data connections are made and what conversions are performed in the process. Debugging shaders created in the system would also be difficult for the same reason.

3.2.5 Interactive Shader Development

The paper Interactive shader development (Jensen, 2007) proposes new ideas for how to remove several programming difficulties in visual shader development. Graphs are constructed by connecting node slots much like in the systems proposed by Goetz and
Abraham, but the Jensens’s system has a more comprehensive typing system (referred to as intelligent variables in the paper) that not only specifies data type but also which mathematical space (object, world, eye and tangent space) a value is defined in. When constructing shader graphs, the system only allows connections between matching data types but still allows connections between slots declared with different spaces. If connections are made between two slots with different space declarations, the system will automatically insert transformation nodes that will transform the value to the correct space before it reaches the other slot. Thanks to this feature the user will not have to worry about different mathematical spaces when creating shaders.

The paper also discusses two ways of optimizing the shaders. The first one involves the placement of each code fragment either on the applications side (for functions based on constant values only, whose results can be passed as a constant to the shader), in the vertex program or in the fragment program. The other optimization process involves inserting as few transformation nodes as possible and to insert them in places that are most cost-efficient.

Jensen also proposes ideas for improving workflow such as the possibility of grouping a set of nodes together into one group node, and a preview feature at each node that show the shader result of that node’s subtree.

### 3.2.6 Commercial Tools

The graph-based approach for designing real-time shaders is still quite new but there are already a few commercially available applications that are based on this technique. Shaderworks (Mad Software, Shaderworks.com) is a stand-alone system for graph-based real-time shader development that started out as an open source project. The system was bought up by Activision in 2005 to be used as an in-house tool, and the system is since then no longer available for download. There are still some tutorials and screenshots available on their homepage from which some information on the systems can be drawn. The system creates shaders in HLSL language and exports effects in the DirectX 9 Effect Files (.FX) format. The system uses separate graphs for describing vertex and fragment shaders, and supports effects with multiple passes. The system performs type checking and the slots in the nodes of the graphs are color-coded by type to aid the user in making valid connections. From the available information there seems to be no support for automatic conversion neither between types nor between transformation spaces. Shaderworks also had a SDK that facilitated integration with third party products. No information is available on details of the SDK implementation other than that it relies on callback methods for constant value, mesh and texture updating.

The 3D engine package Unreal Engine™ (Epic Games, Unreal Technology) comes with a graph-based editor for designing materials called Visual Material Editor. Although there is almost no publicly available information on the tool, a screenshot was available online, but has now been removed, that showed a graph that describes a material with normal mapping and parallax effects. The system seems to be based on material description only, with lighting and shadowing etc. handled separately by the engine. The root node is a material node with a set of input slots for diffuse, emissive and specular color, opacity, surface normal etc. These slots can be connected to other nodes for texture sampling, bump offset calculation etc to describe the appearance of the material. Each node also has a preview window – the material node for example has a view that shows the material applied to an object, the texture sample nodes show a preview of the texture image etc. No
information is available on the format of the shaders that the system exports, nor about how the shaders are integrated with the engine.

Another game engine that has a graph-based interface for shader creation is Project Offset (Offset Software, Project Offset). The engine is still under development and as in the case of Unreal Engine the information available about the shader creation tool is very sparse. Some videos and screenshots have been released though that show some glimpses of the tools’ functionality. Nodes come with a set of slots that can be connected to slots in other nodes and some nodes (e.g. a texture sampling node) shows a preview of the texture image in the same manner as the Visual Material Editor for Unreal Engine.

Tools that are designed to work with specific engines are often adapted to work well and integrate seamlessly with their own systems for lighting and shadowing etc. As a result they can be difficult to reuse and integrate with third party products.

### 3.2.7 Non Graph-based Tools

FX Composer (NVIDIA Corporation, FX Composer) and RenderMonkey (AMD, ATI Developer: RenderMonkey™) are two freely available and commonly used tools that aid in the development of vertex and fragment shaders. None of these applications use a graph-based approach for shader design, but they are two commonly used tools for shader development which makes them interesting for this project for a comparison of alternative approaches.

Both tools produce shaders in the form of Effect Files (St-Laurent, 2005). Effect Files are in essence an extension of high level shading languages that in addition to source code also contain rendering flags, references to textures, separation of the source code into multiple passes and parameters that describe how to set up an effect for rendering. The tools offer an editor interface with syntax highlighting where the user can edit effect files directly and also have a preview window that shows the current effect applied to a 3D object. Program variables, parameters and rendering flags can be modified in a graphical user interface while the preview window is continuously updated to show the effect of the changes.

These two applications are mainly aimed at programmers since shaders are written by hand, but they also offer templates for different kinds of effects which makes it possible for non-programmers to create and customize effects as well. The graphical user interface also makes it possible for non-programmers to tweak parameters for an effect that a programmer previously has created.
CHAPTER 4

Design Considerations

4.1 Overview

After having studied the work presented in the previous chapter, it has become evident that the development of shaders with of a graph-based visual programming interface is a promising technique. The technique is based on ideas that were presented as early as during the 1980s, and was proven successful in the off-line rendering industry. Even though much of the work that has been done in the area has not been aimed directly at the real-time technology of today’s 3D rendering hardware, the work can easily be applied to it due to the similarities between real-time shader programming and earlier off-line rendering environments such as RenderMan.

Although work has been done in the area, there are still several aspects of the technology are in need of more research. The idea is to incorporate the (in my opinion) best aspects of previous work into the design of the own system, and also try to make improvements by introducing some new ideas.

Most of the implementations of visual shader programming that are available today are designed to work with specific engines, which makes them difficult to use with other engines or applications. One of the goals of our system will be to keep it general, without ties to any specific engine or even to any specific shading language. This will have a big impact on system design, and for such a system to still be useful we will have to investigate the possibilities of creating a general interface for integrating the produced shaders with third party applications.

Another big goal with the system will be to make shader development for users that might not have experience in shader programming, which means that we will need to investigate ways of making the system easy to use for this user group.

A third goal with the system is that it shall be possible to be used as an aid in learning shader programming. This will impose its own set of requirements on the system design.

4.2 Target Users

There are three major groups of users of the system that has to be taken into account in the design of the system.

The first group is artists, since the over all goal with the technology is to give artists more influence over shader design. It will be assumed that they have a basic knowledge of 3D graphical concepts such as diffuse and specular color, normal, reflection and refraction etc.
The second user group is shader programmers. The technology of shaders evolves quickly; new shader language features and algorithms are introduced frequently which means that some nodes might have to be re-implemented and new ones created, for which a shader programmer will be needed. It is also the intention that even shader programmers shall find this system useful, as it will hopefully be more efficient than writing code by hand and since it will help in organizing code in bigger projects.

The third user group is students. The requirements that this group imposes depends on the level of the course that they are taking. The tool should be able to be used both in learning about shading in general, and to be used for aiding the process of learning shader programming.

4.3 Abstraction

Since one of the main goals of the system is to make it possible for artists with no shader programming experience to create shaders, possible ways of making the process easier by various levels of abstraction is an important part of the system design. Artists involved in 3D art development usually have a wide knowledge of common 3D graphics concepts like lighting coefficients, transparency and normal vectors but often lack knowledge of space transformation, vector normalization, and of advanced concepts like view direction vectors etc.

As described in the last chapter, the paper by McGuire (McGuire, 2006) (see 3.2.4 Abstract Shade Trees) presents an extreme idea of abstraction, which in my opinion makes it difficult to know what is actually being done. Because of the focus on ease of use and not so much on performance, the technique will probably also only be useful for non-programmers. The work presented by Goetz (Goetz, 2004) (see 3.2.3 An XML-Based Visual Shading Language) on the other hand offers very little abstraction, which gives the user much more control but also makes it difficult to use for users without programming experience. A technique for handling type conversions in a general way in combination with an interface where data flow is clearer could make the tool both for non-programmers and for programmers or other users in need of more control. The system presented by Jensen (Jensen, 2007) (see 3.2.5 Interactive shader development) seems to have a better balance where the user has more control but where the system still handles space transformations automatically. It would be interesting to investigate possibilities of extending the automatic conversion between spaces to be able to handle any form of type conversion (float to integer, mathematical spaces, color spaces, normalized/non-normalized vectors) and to make it possible for more advanced users to easily extend the system with new types and conversion rules. We will investigate possibilities of implementing such features in our system (see 5.4.1 Automatic Type Conversion).

To make it easier for the user to create graphs we should try to remove as much complex details as possible from the graphs. One natural way of making the system easier to use is to create nodes that handle several tasks (like a complete lighting node, an environment mapping node etc) so that the user can create shaders by connecting various concepts rather than defining the actual calculations to be made. Inevitably the risk of performing redundant calculations will rise as the nodes grow more complex, as nodes might share some of the same calculations. Some nodes will still also need to share information about advanced concepts like view direction vector etc. that might be difficult for less experienced users to grasp. It would therefore be interesting if there could be a way for nodes to communicate some of the more complex information without the user having to
handle it manually by creating graph connections. We will try to solve these problems in the design of our system (see 5.4.3 Globals).

4.4 Application Integration

In order for our shader creation program to be useful in projects where the shaders are to be passed on and used in for example a game engine, it is vital that the shaders can easily be integrated with a third party system. In an interview with the lead rendering programmer of a big computer game company in Sweden it was confirmed that this is one of the major problems with existing tools today. Either the tools are designed to work with a specific game engine, or they are too general to make use of engine-specific features.

The commercial tools based on this visual shader programming that exist today are mostly add-ons for commercial rendering engines. The problem with these tools is that they are specialized for their respective engines and therefore might be difficult to reuse for third part engines. There have been some projects that have strived to create stand-alone tools for visual shader development, but they have either been bought up for in-house use or have been shut down. A stand-alone tool would be preferable, since it could be reused for a wider range of projects. The problem with these tools is that their usefulness is highly dependent on the ease of integrating their output with third party engines, and this is an area that is still in need of research. We will discuss ways of facilitating integration with third party systems in section 5.5.

4.5 Educational Aid

Programming real-time shaders is a complex task, even in high-level shading languages like Cg, HLSL and GLSL. The learning curve involved is high, even for people with experience in other programming areas. The hope is that our system can be used to make the process of learning shader programming easier.

For students completely new to shader programming, it would be good if the system could offer a way of interacting with different shading components in an abstract way in order to give a feel for the different elements of shader development and for what the limitations and possibilities of the technology are.

If it is possible for the user to write source code in the system it could also be used in more advanced courses. As opposed to having to write complete shader programs and set them up by also having to create a DirectX or OpenGL program that can run them, our system could offer a “sandbox” environment where the application-setup part would already be handled so that the user can focus on simply writing shader source code.

4.6 Key Functionality

The technology of shader programming is evolving at a fast pace and new algorithms and effects are introduced daily, and it would be impossible to foresee and build in all needed functionality in advance. For our shader development system to be useful it therefore has to be easy to extend with new functionality, otherwise the system would age quickly and it would be difficult to compete with the difficult but flexible development process of writing code by hand.

It is important that the system will not be restricted to working with a specific shading language. There are several big shading languages in use today for different platforms, and
these languages are constantly evolving as new hardware features are introduced. If the system would be tied to a specific language it will not be useful in a wide range of projects and might risk being rendered obsolete as the languages evolve. The design of the system therefore needs to handle shading language implementations in an abstract and general way.

The system will need two user interfaces, one for shader design and one for extending the system by customizing and implementing new nodes. For effect design the system shall have a clean and inspiring GUI with the main focus on an interface for editing dataflow diagrams. It is important that the visual feedback of the system is good, especially since the purpose of the system is to create visual effects. It must be possible to preview the effects in the systems as they are designed. The interface for extending the system with new nodes should be as light and clean as possible. It should also impose minimal restrictions on which shader language is used so that we will not be tied to a specific language that might die out in the near future.

For debugging purposes and to be used as an aid in learning shader development it shall be possible to view the source code of the shaders that are generated by the system. The source code shall be commented automatically where possible, and be syntax-highlighted to improve readability.

### 4.7 System Requirements

There will be an opportunity to test the system on a group of students as part of a laboratory assignment for a course in advanced computer graphics. If the testing of the system is successful, it is possible that it will also be used as an educational aid in other graphics courses at the university in the future. The computers that are used in these courses run Windows XP™, have RADEON X300 graphics cards (which support Shader Model 2.0) and have support for OpenGL version 1.5. Care has to be taken to make sure that the minimum requirements of our system will fit within these limitations.
CHAPTER 5

System Design

5.1 Overview
The system uses dataflow graphs for representing shaders, where each node of the graph will represent operations and the connections between nodes will represent dataflow between the operations. Nodes can have several input slots as in most of the previously proposed systems, but as opposed to only having one output slot our system allow nodes to have several output slots. The reason for this is that operations sometimes produce several output or variations of the same output.

Because of the requirements of customizability and extendibility the nodes and their functionality are completely defined by an external node implementation interface. This makes it possible for users to design and create new node types. The idea of using an external form of defining nodes has been proven useful in different forms in related work (see chapter 3). The interface we will implement will be based on XML. Note that XML was also used in the report An XML-Based Visual Shading Language (Goetz, 2004) presented in chapter 3, but was there used in a different way. XML will here be used for describing the appearance and functionality of the individual nodes, along with fragments of source code in a shading language. This interface is more like the node implementation interface described in Abstract Shade Trees (McGuire, 2006), but based on XML and with the big difference of slots not being hidden, but instead an important part of the graph interface.

The system offers the possibility of previewing the shaders as they are created in the system and the rendering engine (a custom engine made for this system) that is used for rendering the preview is based on OpenGL. Because of this we have chosen to use GLSL (the native shading language of OpenGL) during the development of the system. One of the goals of the system is to make it as flexible as possible, and we have therefore made sure that the design of the system will not make it dependant on any specific shading language, so support for other shading languages could be added in the future.

The remainder of this chapter has been divided into four main sections that will cover different aspects of the system design. We will start by describing the core of the system, which is the way in which the system uses graphs to describe shaders. We will then continue with describing the design of the user interface, which consists of a graphical user interface that is used for creating and previewing shaders, and a textual interface that is used for customizing the system by creating new node types etc. The third section covers the design decisions that have been made to make it as easy as possible for users without experience in shader programming to use the system. The final section will describe the aspects of the system design that will facilitate integration of the shaders with third party applications (e.g. game engines).
5.2 Shader Graphs

5.2.1 Graph Structure
Each graph node represents a specific function or set of operations that will be carried out on a set of input parameters, and will produce a resulting set of output values. Each node will have a set of input slots; one for each input parameter of its function, and it will also have a set of output slots; one for each output value of its function. Nodes can be connected to form graphs by connecting an output slot from one node to an input slot of another node, specifying that the output value of the first function will be used as input to the second function.

Connections in the shader graph represent dataflow from one function (node) to another and the graphs will therefore be directed graphs, where connections only are allowed from an output slot to an input slot. The outputs of one function can be reused as input for several functions which means that one output slot can be connected to several input slots, but it will only make sense to specify one value for each input parameter of a function which means that each input slot can have only one incoming connection. Further more loops are not allowed in the graphs. The allowed subset of graphs formed by these constraints is DAGs (Directed Acyclic Graphs).

5.2.2 Node Types
The graph nodes can be separated into three different groups: input nodes, function nodes and output nodes.

Input nodes are nodes that represent some kind of input to the shader. Since these just have an output slot and no input slots they appear as sources of dataflow in the graphs. There are different kinds of input node types for different kinds of input, like constant values, vertex attributes, shader parameters (see 5.5.3 Shader Parameterization) etc.

Function nodes specify the actual operations to be performed by the shader, such as texture lookups, color blending, mathematical operations, geometry operations and much more. The operations they represent are carried out on the input parameters of their respective
nodes, and the output slots represent the resulting output of the operations. The system comes with a set of ready function node classes that perform a variety of standard shader-related operations. When constructing graphs in the system a function node is created as an instance based on one of these node classes. Since extendibility is an important aspect of the system it will be possible to create new function node classes and to customize old classes in the system through a separate user interface. This interface is covered in more detail in section 5.3.3 Node Implementation Interface.

Finally, output nodes are nodes that represent shader output of some kind. These nodes only have input slots, one for each output parameter they represent. The Pixel Output node for example represents final fragment shader output and has input slots for the final shader output parameters color, alpha and depth for each pixel fragment. Since the output nodes have no output slots and therefore no further outgoing connections they will be the dataflow targets of their graphs. A shader is not limited to use only one output node, several output nodes could be used to specify different aspects of the shader. One could for example use one output node for pixel output and a second one for handling vertex transformation output, both with separate subgraphs for describing their respective operations.

5.2.3 Data Types

Each input and output slot of the nodes in the graph is assigned a data type that specifies the kind of data that can be connected to it. The system has a set of built-in data types, but the user can also add custom data types (see 5.4.2 Custom Data Types). Most of the built-in data types are the same as in the GLSL shading language. Table 5.1 shows a list of some common built-in data types.

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>Floating-point scalar</td>
</tr>
<tr>
<td>int</td>
<td>Integer value</td>
</tr>
<tr>
<td>vec2</td>
<td>Two-component floating-point vector</td>
</tr>
<tr>
<td>vec3</td>
<td>Three-component floating-point vector</td>
</tr>
<tr>
<td>vec4</td>
<td>Four-component floating-point vector</td>
</tr>
<tr>
<td>sampler2D</td>
<td>2D texture sampler</td>
</tr>
<tr>
<td>samplerCube</td>
<td>Cube map sampler</td>
</tr>
<tr>
<td>color</td>
<td>An alias for the vec3 data type, used for storing RGB colors</td>
</tr>
</tbody>
</table>

Table 5.1 Examples of built-in data types

For a shader graph to be valid, only slots with the same data type can be connected. The user interface will still allow two slots of different types to be connected if there is a way to convert data between the two types. In this case the graph will be altered automatically prior to compilation by insertion of conversion nodes so that the graph remains valid (see 5.4.1 Automatic Type Conversion)
5.2.4 Shader Graph Example

Figure 5.2 shows an example of a shader graph describing a texturing effect where the colors from two different textures are mixed together. The shader uses several different inputs represented by the separate input nodes of the graph: two samplers (textures), a vertex attribute \textit{TexCoord} which is a two-dimensional image coordinate stored with each vertex in the geometry, and finally a constant float value 0.5. The \textit{Texture2D} nodes sample the color of a texture at a specific coordinate. The two \textit{Texture2D} nodes in this graph will sample at the same coordinates (given by the vertex attribute \textit{TexCoord}) but from different textures (\textit{Sampler1} and \textit{Sampler2}). The resulting colors of these two samples are then coupled to the two color input slots of a \textit{Mix} node, along with the float constant 0.5 coupled to the \textit{Balance} input slot. The \textit{Mix} node performs the following calculation:

\[
\text{ColorMix} = \text{Color1} \times (1.0 - \text{Balance}) + \text{Color2} \times \text{Balance}
\]

Hence, a balance value of 0.0 will output \text{Color1}, a balance value of 1.0 will output \text{Color2} and a balance value between 0.0 and 1.0 will output a linear interpolation between the two color values. Since the value 0.5 is used as balance, the resulting output from the \textit{Mix} node of our graph will be an average of the two input colors. The result of the mixing is finally coupled to the \textit{Color} slot of the \textit{Output} node, and will be the color finally written to the pixels during rendering.

Figure 5.1 shows the actual graph as maintained and seen by the system, but it is represented differently in the user interface, as explained in the section 5.3.2 Graph Interface.
5.3 User Interface

5.3.1 GUI Overview
The graphical user interface is based on standard Windows™ controls, and consists of a main application window divided into sub-windows that focus on different tasks of the shader development process (see figure 5.3).

![Graphical User Interface](image)

**Fig 5.3 Graphical User Interface**

1. Explorer Window
This window is divided into two sections. The top section shows a hierarchical overview of the current shader and its components. The components that are shown here are vertex attributes and samplers that have been declared in the shader. The bottom section of the window shows details about the shader component that is currently selected in the top section of the window. This bottom section is meant both for viewing and for altering parameters of the selected shader component.

2. Output Window
The output window is meant to show different forms of text output from the system. The window has two tabs, showing the generated vertex and fragment shader source code respectively. The source code is syntax highlighted for better readability. The highlighting system recognizes symbols of from the GLSL language, but can easily be extended to support other languages in the future. The plan is to add more tabs to the output window.
in the future, for compiler messages etc. Compiler errors are currently shown with message box popups.

3. Preview Window
The preview window shows a rendered 3D view of the current shader applied to an object. The user can pan, zoom and rotate the view to see the shader in action from different viewpoints. The system offers some built-in standard objects such as a cube, sphere, torus etc that can be used for viewing the shader, but external objects can also be loaded from files. The 3D object file format currently supported in the system is RTF-Files.

If the shader graph cannot be compiled into a valid shader at a given time (e.g. because of an illegal connection present in the graph etc.) the preview window will show a wireframe model (only the edges of each triangle in the geometry is drawn) of the 3D object, using a default white shader.

4. Graph Window
The dataflow diagram window is the most important part of the graphical user interface. In this view the dataflow graphs that define the shaders are constructed. For more information on this window see the following section.

5.3.2 Graph Interface
This is the interface that the user will use to design the actual shader graphs. The interface consists of a background canvas, on top of which the user can place nodes and draw connections between the nodes.

A node is created by clicking somewhere on the background canvas, and then selecting a node type from an appearing menu. A node is represented by a box with a title bar displaying the node name (see figure 5.4). The input parameters of the node are listed in the top left corner of the box, and the output parameters are listed in the bottom right corner. The connection slots of each node are marked with small arrows. Each input parameter has a connection slot directly in front of it, in the left margin of the box, and the output parameters each has a connection slot placed directly to the right of them, in the right margin of the box.

![Fig 5.4 A node in the Graph Interface.](image)

To connect an output slot of one node with an input slot of another node, the user simply presses the mouse button over the output slot, moves the cursor to the input slot to connect it to, and then releases the mouse button. Edges are represented by lines drawn between the two slots they connect.
As a result of the placement of the input and output slots on the left and right sides respectively, input nodes are naturally placed on the left side of the graph, and output nodes on the far right, so that the flow of the graph goes from left to right as shown in figure 5.5.

To simplify the graphs and to keep them more structured and compact a design decision was made to allow values for the node input parameters to be entered beside each parameter directly in the node box, as opposed to having to construct for example a constant value node and then connect that node to the input slot of the input parameter. To specify a value for an input parameter the user first clicks on the parameter. A menu will then appear in which the user can decide whether to use a constant value, a vertex attribute, an external variable (see 5.5.4 Externals) or a shader parameter (see 5.5.3 Shader Parameterization) to define the input parameter. The user can also decide to leave the parameter undefined in which case a default constant value will be used for the parameter.

If the constant value option is selected, a control will appear beside the input parameter, which the user can use to specify a value for the parameter. Different controls are used for different types of input parameters. For floating point parameters a number field is used, and for color parameters a special color control is used. If the user instead selects a vertex attribute, an external variable or a shader parameter, this item will be displayed beside the input parameter instead of the constant value control. The user can also always use the output of another node as input to the input parameter by simply drawing a connection to the slot, in which case the previously specified value for the parameter will disappear, and the value from the connection will be used instead.

![Fig 5.5 A shader graph in the Graph Interface.](image)

Because of the decision of placing parameter values directly in the node boxes, constants, vertex attributes, shader parameters and externals will never need to be represented by separate node boxes in the user interface, they will just appear as symbols or controls in the nodes that use them. Note that this is just a feature of the user interface, the underlying graph maintained by the system will still have to have a node for each constant value, vertex attribute etc. Figure 5.5 shows how the shader graph from figure 5.2 would be represented in the user interface.
5.3.3 Node Implementation Interface

To satisfy the requirements of extendibility and customizability a design decision to keep the implementation of the function node classes completely open was made early on. All functional node classes are defined completely by XML-files which are kept in a sub directory in the program directory, one XML-file for each node class. To add a new class all that is needed is to declare it in a new XML-file and place the file in the node class sub directory. On startup the system will search the node class sub directory and update its list of available node classes that can be used when constructing the shader graphs.

The XML-files define the appearance (name etc.), input and output slots and a fragment of shader source code describing the implementation of the functionality of the node. The input and output slots are declared by input and output tags respectively (see fig 5.6), one tag for each slot. These tags have 2 parameters; the name of the slot and the data type of the parameter associated with the slot. For the input slots it is also possible to define a default value for the slot by writing the default value as the body of the tag (as for the Level slot in fig 5.6). The default value of an input slot will be used if no value is specified for the slot when constructing the graph. If no default value is specified for an input slot in the declaration, the default value for the data type associated with the slot will be used.

The body tag holds a fragment of source code that implements the functionality of the node. The system does not parse the source code; it is just treated as a string of text. The interface imposes only two restrictions on the shading language used. Symbols that could be mistaken for XML-tags are not allowed, which would otherwise break up the structure of the XML-document, and the $-character is reserved for special use. The purpose of the source code fragment is to implement the function to be carried out on the input parameters for the node, and to write the results to its output parameters. To be able to do this there has to be a generic way of referencing to these parameters in the source code, and this is accomplished by writing the name of a parameter (as declared in the input and output tags) prefixed by a $-character.

When a shader graph is compiled, the source code fragments from each node in the graph are assembled into one program and all words prefixed with a $-character are replaced with unique variable names generated by the system. $-references of two connected slots from different fragments are replaced by the same uniquely generated symbol to link the dataflow from the first fragment to the second through the use of a common variable. To avoid collisions between temporary variables (collision here meaning having the same name) of different fragments, even these variables has to be prefixed with a $-character. If a word prefixed with a $-character is found which does not correspond to an input or

```xml
<node-class>
  <title>Saturation</title>
  <input name="Color" type="color" />
  <input name="Level" type="float">1.0</input>
  <output name="SatColor" type="color" />
  <body>
    // Saturation
    float $lum = dot(vec3(0.3,0.59,0.11), $Color);
    vec3 $SatColor = mix(vec3($lum,$lum,$lum), $Color, $Level);
  </body>
</node-class>
```

Fig 5.6 Implementation of a Node Class.
output parameter the system will generate a unique variable name and replace all occurrences of the $-prefixed word with this new variable name. The compilation process is described in more detail in 6.4 Compilation Process.

The decision to base the interface on XML was made because it is a widely used format that most users already are familiar with. Because of its hierarchical structure the interface can also easily be extended with additional parameters and sub-trees as the system evolves and more features are added, which satisfies the requirements of building a system that is open for evolution.

5.4 Abstraction Features

The system has been equipped with a set of abstraction features which when combined become very powerful. The first feature, globals, makes it possible for an advanced user to set up default evaluations of complex concepts so that the ordinary user will not have to deal with these. The second feature, automatic type conversion, makes it possible to specify for the system how to convert data from one type to another, so that type conversion can be handled automatically by the system. The last feature makes it possible to create custom data types that can hold semantic or spatial information, which in combination with the type conversion feature makes it possible to for example automatically handle transformations between different coordinate spaces.

5.4.1 Automatic Type Conversion

To convert a value from one data type to another it would be possible to implement a function node with one input node of the first type, an output node of the second type, and with source code that converts the value of the input parameter and writes it to the output parameter. An example of an implementation of such a node is shown in figure 5.7.

![Fig 5.7 A function node that implements data type conversion](image)

Although this is a possible way of handling data type conversion, it will still require that the user has knowledge of the data types of each slot, and it will be time consuming to insert conversion nodes by hand everywhere they are needed. It would be much more efficient if the system could handle data type conversion automatically and even better if it could be handled without having to clutter the graphs with conversion nodes.

To solve the problem the system has been equipped with a sub-system for automatic type conversion which has been inspired by ideas from the paper Abstract Shade Trees (McGuire, 2006) (see 3.2.4 Abstract Shade Trees). Our system maintains a data conversion graph where nodes represent data types and edges represent a possible conversion between the two data types. If a path exists between two nodes in the graph, there is a possible conversion between the two respective types, either directly or indirectly (in several conversion steps). The graph is set up in a special XML-file with a syntax that is similar to
that of node implementation. This interface is called a type library, and contains both definitions of custom data types (covered in section 5.4.2 Custom Data Types) and rules for conversion between these data types. Figure 5.8 shows an example of how the conversion discussed above can be implemented in this new interface.

![Fig 5.8 An excerpt of a type library, showing the implementation of a conversion from the float data type to the int data type.](image)

Each conv-block in the type library specifies a conversion between two data types (an edge in the data conversion graph). The type-tag specifies which data types the conversion is for. The penalty-tag specifies the complexity of the conversion and results in a weight on the corresponding edge in the data conversion graph. The body-tag specifies the implementation of the type conversion in a shader language (GLSL). The source code fragment is written exactly as in the node implementation interface (see section 5.3.3 Node Implementation Interface), but here the node will always have one input slot called “from” (the value to convert) and an output slot called “to” (the variable to write the converted value to).

The weight (penalty) to be used for each conversion edge should be an estimate of how complex the conversion is. One could for example specify the weight as the number of instructions needed to perform the conversions. By using weights it will be possible for the system to find an optimal conversion path between two data types.

When using the automatic conversion system two slots of different types can be connected in the user interface. During compilation the shader graph will be traversed and conversion nodes (using the implementation specified in the type library) will be inserted where needed (but not visible to the user). If a connection has been made between two slots in the shader graph, but there is no path between their respective data types in the conversion graph, an error will be generated during compilation.

Thanks to the automatic type conversion system the user will not have to worry about different data types, the shader development process will be much easier and more efficient, and the graphs will not be cluttered with (visible) conversion nodes. As will be shown in the next section, the conversion system will be even more powerful when combined with custom data types, which will make it possible to not just handle conversions between raw data types but between other type semantics like coordinate spaces etc.
5.4.2 Custom Data Types

To make it possible for the system to automatically handle complex tasks like transformations between different coordinate spaces etc. one might argue that it would need more information about a parameter than just its raw data type. For a vector it would for example need to know in which coordinate space it is defined, whether it is a directional or a position vector (coordinate space transformations are easier for directional vectors than for position vectors), and if it is a directional vector it would be interesting to know if it is normalized or not etc. A way to implement this would be to extend the type definitions of the system so that a parameter would have to be declared with storage type, semantic, coordinate space, length (unit/arbitrary) etc. And then to add logic in the system for handling the different type attributes separately, like converting the coordinate space part of a type (matrix transformations), or the length part (normalization) conversions etc.

The problem is that for example conversion between two coordinate spaces is handled differently based on semantics (“direction” or a “position”), storage type (2-component or 3-component vector) etc. so we end up having to look at combinations of type attributes as separate data types anyway. Another problem is that there will be much redundancy in this type definition system since not all type attributes makes sense with all values; a color has for example no coordinate space. A third problem with this system would be that it would be relatively stiff, it would be difficult to add new type attributes without interfering with old ones, as the conversion routines would end up quite complex having to check for different combinations of type attributes. The solution proposed here, and implemented in the system, is much simpler and yet very powerful.

In addition to the standard built-in data types of the system (see section 5.2.3 Data Types), the user can create additional data types through the use of type libraries. A type library is a special XML-file that contains definitions of custom data types and information on how to perform conversions between different types. The custom types can not define a completely new type of data storage; they still have to use built-in data types for storage, so in essence they are little more than aliases for the standard data types. Custom types are defined by a type name and a reference to a built-in data type to use for storage. Figure 5.9 shows an excerpt of a type library file that defines some custom data types.

```xml
<type-lib>
  ...
  <alias-type name="opos" super="vec3" />
  <alias-type name="wpos" super="vec3" />
  <alias-type name="e pos" super="vec3" />
  <alias-type name="tpos" super="vec3" />
  <alias-type name="odir" super="vec3" />
  <alias-type name="wdir" super="vec3" />
  <alias-type name="edir" super="vec3" />
  <alias-type name="tdir" super="vec3" />
  ...
</type-lib>
```

Fig 5.9 An excerpt of a type library, showing the declaration of some custom data types.
The types shown here are all based on the built-in data type \textit{vec3}, which is a three-component floating-point vector. The type names $o\textit{pos}$, $w\textit{pos}$, $e\textit{pos}$ and $t\textit{pos}$ stand for positional vectors expressed in the object, world, eye and tangent spaces respectively. The type names $o\textit{dir}$, $w\textit{dir}$, $e\textit{dir}$ and $t\textit{dir}$ denote directional vectors in the different coordinate spaces and the $o\textit{normal}$, $w\textit{normal}$, $e\textit{normal}$ and $t\textit{normal}$ types represent normalized (length 1.0) directional vectors.

These new types can be used as any other data type for example in slot definitions for functional nodes, vertex attribute declarations etc. and will all result in the usage use of $\textit{vec3}$ variables in the final generated shader program. Although all these types are based on the same standard data type for storage, they are still treated as completely separate types and can therefore not be mixed directly in graph connections etc.

The system has no more knowledge of these new types other than the form of storage they use. The actual characteristics of the new types and relationships between them are defined completely by specifying conversion rules (see section \ref{type:conversion}). Figure \ref{fig:5.10} shows some conversion rules for some of our new vector types.

\begin{figure}[h]
\begin{verbatim}
<type-lib>
...
/*!-- normalizing -->
<conv>
  <type from="o\textit{dir}" to="o\textit{normal}" />
  <type from="w\textit{dir}" to="w\textit{normal}" />
  <type from="e\textit{dir}" to="e\textit{normal}" />
  <type from="t\textit{dir}" to="t\textit{normal}" />
  <penalty>3</penalty>
  <body>
    vec3 $to = normalize($from);
  </body>
</conv>

/*!-- Object Space to World Space -->
<conv>
  <type from="o\textit{dir}" to="w\textit{dir}" />
  <type from="o\textit{normal}" to="w\textit{normal}" />
  <penalty>10</penalty>
  <extern name="worldmtx" />
  <body>
    // Object-space to World-space rotation
    vec3 $to = mat3($worldmtx) * $from;
  </body>
</conv>
...
</type-lib>
\end{verbatim}
\caption{Conversion rules for custom vector data types.}
\end{figure}

The first conversion rule defines how to convert arbitrary length directional vectors to normalized directional vectors. Note that the same rule can be applied to conversions between several pairs of types by just inserting several \texttt{type}-tags. Imagine that we have a functional node that performs some kind of geometrical operation in world space (mirroring a vector for example) and which outputs the result as a non-normalized vector.
and that we want to use this vector in lighting calculations performed by another node. If the operations of the lighting node would require the input vector to be normalized (which is usually the case) this node would have an input slot of type \texttt{wnormal} which would not be directly compatible with the vector we would like to use. With the first rule in figure 5.10 in effect, the system would allow the two slots to be connected, because it would know how to perform a conversion between the two. In effect we have with this first rule added support for automatic vector normalization to our system.

The second conversion rule in figure 5.10 specifies how to convert a directional vector (normalized or non-normalized) in object space to world space. The conversion is implemented by multiplying the vector with a world transformation matrix. This is the commonly used matrix which is used for defining and objects position and orientation in world space (see section 2.1.3 Coordinate Spaces). The world matrix is declared as an \texttt{external} variable, this is a way of accessing application variables from within the shader and is explained in more detail in section 5.5.4 Externals. Imagine that we again want to use a node that performs lighting calculations in world space (because lighting information as light position, direction etc are specified in world space by our application) and that the node would need a surface normal to perform its calculations. A common way of defining the surface normal for an object is to include it as a vertex attribute in the geometry, expressed in object space. Since the normal is expressed in object space it could not be used directly as input to the lighting node. With our new type system the normal would be declared as \texttt{onormal} and the input slot of the lighting node would be declared as \texttt{wnormal}.

With the second conversion rule of figure 5.10 in effect the system would allow these two to be connected, and would result in an invisible conversion node being inserted between them to perform the conversion specified in the type library. By adding more conversion rules between our new vector types we could abstract the concept of coordinate spaces totally from the shader graphs, so that a user would not have to worry about performing transformation between coordinate spaces.

The powerful combination of custom data types and automatic conversion rules makes it possible to add support for abstraction of virtually any form of type attribute or concept in our system. The solution is very general and flexible since no rules or concepts have to be hard coded into the system, all functionality is specified in simple XML-files. The system is also very light and solves the redundancy issues of the alternative solution discussed earlier.

5.4.3 Globals

After having started implementing parts of the system it soon became clear that the communication between nodes offered by slot-connections alone was not enough for handling more complex effects in a general and efficient manner. To illustrate the problem we will consider an example.

Imagine that we have a node that performs an environment mapping effect. Environment mapping is a way of simulating a mirror-like surface by using a texture image for representing the surroundings of an object. Our node implements the Cube Environment Mapping algorithm, where the surroundings are represented by a cube map (a 6-face texture, one for each face of a cube). To decide the color of a given fragment, the algorithm takes the viewing directing vector (the vector from the eye/camera to the current fragment to be rendered), mirrors this vector by the surface normal at the fragment, and then uses the resulting vector to sample the cube map that represents the surroundings. Figure 5.11 depicts the algorithm in two dimensions.
Our node takes two inputs: a cube map sampler and a surface normal vector. To be able to perform its calculations it will also need the view direction vector. How will we get this view vector?

The view direction vector can be calculated by subtracting the fragment position from the eye position. We could make these calculations in the implementation of the environment mapping node, by using (an interpolated version of) the position vertex attribute to get the position of the fragment, and to subtract this from the eye position (given by the world-to-view transformation matrix), and then normalize the result to get the view direction vector. But there are two problems with this solution. First, if the view direction vector would also be needed in other nodes in the shader graph, these calculations would have to be performed for each node, instead of calculating the view vector once and then using that value in all nodes. Second, how can we be sure that the fragment position is given by the position vertex attribute? Imagine that the shader graph also had a node that performs vertex transformations of some sort, let us say that we for example are implementing a water shader that has a node that causes a wave deformation on the geometry and also has an environment mapping node for simulating a reflection of the environment. In this case the transformed position should be used when calculating the view direction vector, not the position vertex attribute.

One solution could be to force the user to specify the view vector as a third input to our environment mapping node, but this would mean that any shader graph that would use environment mapping would always also need extra nodes for calculating the view vector. There are several examples of similar components that are needed in various common shader tasks, and they are usually concepts that can be difficult for an inexperienced user to grasp. It would mean much more work for the user if these kinds of details would have to be handled manually, the graphs would quickly grow much more complex and the base requirements for the system as being a tool for making shader development easier and accessible to users without the knowledge of these kinds of concepts would not be met. The system would be much easier to use if these values somehow could be handled automatically by the system.
The solution to the problem that was finally decided upon was to introduce a new concept that we call **globals**. Globals are a way of supplying a default implementation of commonly used data, like for example the view direction vector described above. A global is defined by a name, a data type and a fragment of source code that implements the default way of calculating the data. Globals are defined in special files called **Global Libraries**, which uses a format that is similar to the format of the conversion rules of the **Type Libraries**. The user can add or customize globals simply by editing these library files.

All nodes in a shader graph can read from globals, and if a global is read in a shader graph, the source code from its definition will automatically be included in the final generated shader source. If another shader node in the same graph also uses the same global, there will be no need to insert the source code that define the global again, instead the value of the first calculation can just be reused for this second node. So, data that is commonly used can be defined in **Global Libraries** along with source code for calculating them, and by referring to these globals in the node implementations instead of re-implementing them in each node, we can make sure that they are not calculated more than once in a shader program.

Let us get back to our example. If we add a global to our library of globals, let us call it `viewdir`, and define it by specifying a source code fragment that calculates the view direction vector based on the calculations explained above, we could then refer to this global in the implementation of the environment mapping node and all other nodes that need the view direction vector, instead of inserting the view direction vector calculations in each node implementation. Since a global will only need to be evaluated once, even if it is referred to by several nodes in a shader graph, this means that the direction vector calculations would only be executed once for the whole graph. This solves the first of the two problems mentioned above.

What about the second problem? What if the shader graph also had a node that performed vertex transformations of some sort? Our environment mapping node would still rely on the `viewdir` global for getting the view direction vector, and if the implementation of the `viewdir` global based its calculation on the position vertex attribute, it would generate an invalid view direction vector.

The solution to the problem is to make it possible for shader nodes to also write to globals and thereby make a temporary redefinition of them for the current graph, but a global is only allowed to be written to by at most one node in a shader graph (otherwise several definitions of a global would be available). If a node in a shader graph writes to a global, the default definition of that global specified in the global library will be discarded (and therefore never included in the final generated shader source) and the value written to this global will be used when other nodes in the graph refers to it. So, the default source code specified for a global in the global library will only be used if that global is used in a shader graph but never written to by any of the nodes in the graph.

Let us apply this new feature to the environment mapping example. We add a new global to our global library, which we will call `position`, and as default implementation of this global we specify source code that simply copies the value of the position vertex attribute. We then change the implementation of the `viewdir` global, so that it retrieves its position parameter (see the environment mapping algorithm above) from the position global instead of the position vertex attribute. So, the `viewdir` global will now be dependent on the
Finally, we will also make the geometry transformation node write to the position global, to override its default implementation. As a result, the environment mapping node will obtain the view direction vector from the viewdir global, which in turn will base its calculations on the position global, which in our example will be written to and redefined by the transformation node. If the transformation node had not been present in the graph, the default definition of the position global would be used, which just copies the position vertex attribute. By using globals it is possible to hide and automatize calculations of complex or commonly used data so that the graphs get simpler and easier to work with. In this way the user can focus on the more important parts of the effects, without having to worry about all implementation details.

5.5 Application Integration

In order for our shader creation program to be useful in projects where the shaders are to be passed on and used in for example a game engine, it is vital that the shaders can easily be integrated with a third party system. The following sections will describe how the system has been designed to facilitate integration with third party applications.

5.5.1 Deferred Compilation

Most general shader development tools (e.g. RenderMonkey, FX Composer) exports shaders as pure source code in some shader language, or as Effect Files (St-Laurent, 2005). Effect files contain much more information than just the actual source code. They can contain rendering flags (flags for color blending, texture wrapping modes etc.), references to texture files, separation of the source code into multiple sub-programs called passes (so that an effect can be rendered in several layers), and other parameters that describe how to set up an effect for rendering.

One problem with storing shaders in pure source code or as Effect Files is that they will be rather inflexible. A game engine will often need several versions of the same shader, with variations for handling different lighting or environmental conditions, with or without vertex blending etc. This can be solved in Effect Files by writing different code paths that handle different lighting conditions etc. The application can then decide which paths to use and which sections to skip by specifying different flags when compiling the effect files. Although this is a possible solution, it is still very inflexible, since all possible uses of the shader have to be determined before-hand, when it is written. This solution will also make the shader programs pretty complex to maintain since variable names etc will have to match for different combinations of code paths.

Thanks to the way in which shaders are designed in our system we will have a substantial amount of structural information about the shader program itself. The source code is divided into independent sub-sections represented by the nodes of the shader graphs, and the dependencies between the sections are clearly specified by the edges in the graph. It would be a waste to throw this structural information away by compiling the shader graph and storing the generated shader code. Instead, the system will store the graphs and leave the compiling to be done by the application that will use the shader. This way it will be possible for the application to switch or insert nodes into the shader graph prior to compiling and using the shader, so that it can be adjusted to implement a specific lighting model or environmental phenomenon. A fog node could for example be inserted just prior to the output node, or the lighting node in the graph could be switched to a node implementing the lighting conditions of a specific scene.
Thanks to this way of storing shaders it would also be possible to omit lighting entirely from the shader design, and instead only concentrate on surface shading by using an output node in the shader graph with input slots for different surface parameters like diffuse color, specular color, normal etc. and then let the application insert nodes for specific lighting conditions prior to compiling and using the shader. This makes it possible to reuse the same shader for a variety of conditions, and it will not be necessary to know all possible applications of a shader during shader design.

To make this solution possible, we will have to create an API (Application Programming Interface) that applications can use to load, compile and set up the shaders created in our system.

### 5.5.2 API

To make it possible to integrate shaders created in the system with a third party application the system comes with an API called SGAPI (Shader Graph Application Programming Interface). The API implements functionality for loading, traversing, altering and compiling the shader graphs created in our system. When a shader graph is compiled using the API, a vertex program and a fragment program will be generated. Along with the shader program source code there will also be an information structure that holds information about what vertex attributes etc. that need to be set up to run the shader. For more information on the compilation process and the generated information structure see section 6.4 Compilation Process.

### 5.5.3 Shader Parameterization

The scenes of today’s games are often very complex and are comprised of a wide variety of materials and effects. During rendering only one shader program can be active at a time so in order to use different effects in a scene the active shader program has to be switched several times during the rendering of a frame. Switching shader program is a time consuming process and if a unique shader would be used for each unique material in the scene the game could slow down to a stutter because of the performance losses incurred by the frequent switching of shaders. To remedy this problem the shader programs can use input parameters for customizing their behavior so that the same shader can use for example different textures, colors and lighting parameters etc. The same shader can then be used for rendering a group of materials with similar characteristics by just altering its input parameters. In this manner, a specific material in a scene can be described as a reference to a shader program to use and a set of input parameters to use with that program. Before rendering a frame, all materials can be grouped together by which shader programs they use. The rendering can then be done one material group at a time, within which a parameter change is enough for switching between its materials. In this way the number of necessary shader program switches will be kept at a minimum, resulting in a higher overall performance of the rendering system.

Parameterization of shader programs is possible through the use of uniform variables (see section 2.2.2 The Hardware Rendering Pipeline) which are shader program variables that can be written to by the application and which have read-only access from within the shader programs. The uniform variables declared in a shader program can be enumerated from the application, and written to prior to drawing polygons with a specific material.

To support parameterization in our shader graph program we introduce a new shader component type called **shader parameters**, in addition to the previous components like vertex attributes and constants. A parameter is declared with a name and a data type and appears in
the explorer window along with the other shader components. When a parameter has been declared it can be accessed in the shader graph by using special parameter nodes. This node type works exactly like other input nodes, like for example attribute nodes, but instead of referring to a vertex attribute they will refer to a shader parameter. When the shader graph is compiled these parameter nodes will be represented by uniform variables in the final shader program.

When using the shaders in an application (e.g. game engine), a material could for example be stored as an XML-file that has a reference to the shader file to use, and a set of name-value pairs for all the parameters of the shader. To use the material, first the material file has to be read. Then the shader referred to by the material file can be loaded and compiled through the Shader Graph API (if the shader has not already been loaded by another material). Finally, the material parameters can be matched against the uniform variable parameters of the shader and these can be set to the respective values specified in the material file.

5.5.4 Externals

In order to integrate shaders with for example a game engine, there is often much more information other than material parameters that needs to be communicated between the application and the shader. Information about lighting, environment phenomenon parameters like for example fog density and color, transformation matrices etc. will have to be accessible from within the shader. To support this in our system, a new type of input node called externals have been added. As opposed to other input nodes like parameter nodes and vertex attribute nodes, an external node is used to access an application variable (a transformation matrix, light position vector, fog color etc.).

The external parameters available to the shaders will vary completely from engine to engine, and we will therefore need a general interface for declaring application variables so that our system can be used with different engines. For this purpose, the user can declare externals in simple XML-files called External Libraries, similar to the Type Libraries used for defining custom types. Figure 5.12 shows an example of the format used in the External Library files.

Externals can be used as any other type of input (shader parameters, attributes) in the shader graphs. To use an external as input to a node the user clicks the input slot to connect it to, chooses “Externals” from the appearing menu, and can then choose from the library of external variables declared in the loaded External Libraries. Externals can also be used in the implementation of nodes or conversion rules (see figure 5.10 for an example of how externals can be used in conversion rules).

```
<extern-lib>
  ...
  <extern name="worldmtx" type="mat4x4" />
  <extern name="viewmtx" type="mat4x4" />
  ...
</extern-lib>
```

Fig 5.12 Example of an External Library.
When the shader graph is compiled the external variables will be implemented as uniform variables (see 2.2.2 The Hardware Rendering Pipeline for a description of uniform variables) in the generated shader code. The information structure generated with the source code will contain a list of all externals and the name of their respective uniform variables in the source code so that they can be set up correctly from an application before running the shader.

The rendering engine used for previewing the shaders inside our application only supports a limited set of external variables (some of which are shown in figure 5.12). For external variables that it does not recognize it can do no more than apply a standard value depending on the data type. This is a problem since shaders using external variables not known to the system (which should be the case for any engine-specific variables) will not be rendered properly. To solve this problem we would have to implement a plug-in interface in the future so that external rendering engines could be used for rendering the previews in the system. Another alternative would be to add a user interface for controlling external variables manually during preview rendering in the system.
CHAPTER 6

Implementation

6.1 Overview

The system has been developed in C++ using Microsoft Visual Studio 2005. The application core and user interface are implemented using the MFC (Microsoft Foundation Classes) library, which is a commonly used C++ library that wraps the Windows API and also offers an object-oriented framework for application programming.

To make the system easier to maintain it has been divided into several subsystems that handle separate tasks and with interfaces in between them that are as clean as possible. This makes it possible to substitute the implementation of a subsystem independently of the other parts of the system. A system of this level of complexity would otherwise be very difficult to extend if all parts were strongly coupled together. The main subsystems of the application are graph representation, GUI, compilation and 3D rendering.

6.2 Graph Representation

The graphs in the system are implemented by a Graph class which holds a collection of node objects. There are several different node types that are used in the graphs and these are implemented with separate classes, but all classes implement a common Node interface. The common Node interface exposes methods for enumerating and accessing the slots of the node, for reading attributes such as node name etc. and for compiling the node (for more information on compilation see 6.4 Compilation Process). Thanks to the common interface implemented by all node types the interface towards other subsystems is unified and clean. The graph can be traversed and connections handled by just using the unified Node interface.

The system has two main groups of node types that are represented by the classes InputNode and FunctionNode respectively. The InputNode class implements the base functionality for nodes with just one output slot (i.e. input nodes, see 5.2.2 Node Types) and the FunctionNode implements the base functionality for nodes with several input and output slots. Classes for representing the different types of input nodes, like ConstantNode, AttributeNode, SamplerNode etc., all derive from the InputNode class. ScriptedNode inherits from the FunctionNode and implements nodes whose functionality is declared in XML-files (see section 5.3.3 Node Implementation Interface).

Connections can only be made between an output slots and an input slot, and since each output slot can be connected to several input slots, but each input slot only can have a connection from one output slot, the connections are stored per input slot, as a pointer to the output slot it is connected to. In other words, the connections are stored in the
opposite direction of the dataflow of the graph. If connections were stored the other way around the output slots would have to have lists of the input slots that they are connected to, which would be more complex and less efficient. This also means that the graphs can only be traversed from top to bottom (root to leaves), since the parents store the connections to their children.

### 6.3 Graphical User Interface

The user interface is implemented with the MFC library, and are mostly based on standard Windows™ controls, except for the graph window which has been implemented from the ground. Some of the standard controls used have also been customized and extended with features, such as the Output Window which is based on the Rich Edit control but have been extended to support syntax highlighting for better readability.

The dataflow graph window is constructed as a parent window with a child window for representing each node. Since the nodes are separate windows they handle their own drawing and can be customized independently of the parent window. The parent window paints the background of the graph, and also draws the connections of the graph (as Bezier curves) on top of the background. All the node windows implement an interface of methods with which the parent window can locate the input and output slots of the nodes, so that it knows the coordinates for start and destination of the connection lines to be drawn. All drawing is made using the GDI+ (Graphics Device Interface Plus) library which is a standard object-oriented library for handling drawing operations in Windows™. Thanks to the nodes being implemented as actual windows, drag-and-drop repositioning etc of the nodes can be handled as standard window functionality by the operative system, as opposed to us having to implement these features ourselves. All we have to do is redraw the background when we are notified that a node window has been moved, to update the connection lines.

![Diagram over Graph Classes.](image)
6.4 Compilation Process

The task of the compilation process of our system is to take a shader graph and generate source code for a final shader program, expressed in some shading language. This source code can then be used as any other shader program and be compiled and linked into a final executable shader by compiler and linker tools available for the specific shading language. The implemented compilation process consists of several steps:

- Node Compilation
- Global Resolving
- Type Conversion
- Program Selection (Vertex vs. Fragment program)
- Fragment Ordering
- Symbol Generation
- Source Code Generation

6.4.1 Intermediate Program Representation

During the compilation process an intermediate representation of the shader program is constructed, consisting of a collection of source code fragments and a set of program variables that bind the source code fragments together into a tree-like structure. We will call this structure a fragment tree. It is common for compilers to operate on tree-like intermediate representations of programs, but since our program does no actual parsing of the source code we have no knowledge of the actual operations performed. Instead we are forced to see a whole source code fragment as one operation with a set of input variables and a set of output variables.

Each source code fragment in the fragment tree will often correspond to a node in the original shader graph, but more complex nodes might produce several source code fragments. Additional fragments may also be added by later phases of the compilation process for handling type conversions etc.

In addition to the source code fragments the intermediate representation also maintains a list of program variables. There are six different types of program variables: constants, temporaries, attributes, globals, shader parameters, and externals. Constants are variables that contain a constant value and thus have read-only access. Temporaries are variables that are declared in a source fragment and are either used as temporary variables within the scope of a source code of the fragment, or are used by several fragments for communicating results from one fragment to another next. Attributes represent vertex attribute variables (see 2.2.2 The Hardware Rendering Pipeline). Globals represent variables that are defined in global libraries (see 5.4.3 Globals for more information). Shader parameters represent variables that shall be visible from outside the shader, and are used externally to customize the shader for rendering different materials (see 5.5.3 Shader Parameterization). Externals represent application variables that need to be accessed from the shader (see 5.5.4 Externals). Each program variable is declared as either belonging to a shader fragment that created it, or as being part of the global scope (attributes, globals, parameters and externals always belong to the global scope).

Each fragment has a set of input and output variables that are referred to by textual symbols prefixed with a ‘$’-character in its source code. These references serve as place
holders that will be substituted by the name of real program variables when all fragments are linked together to form the final shader program. When a source code fragment is created, each of its input and output variables are linked to real program variables. The input variables can be linked to program variables created by other fragments, or to program variables of the global scope (externals, attributes etc). New program variables have to be created and linked to the output variables of the fragment.

The actual tree-structure of the intermediate program representation is formed through the use of variables. Two fragments are linked together if an output variable of the first fragment and an input variable of the second fragment are linked to the same program variable, effectively making the second fragment read from a variable that has been written to by the first fragment. The source code fragments can be seen as nodes, and the variables as directed edges with the flow going from the fragment created the variable to a fragment that uses the variable.

6.4.2 Node Compilation

Since one of the main requirements of the system is to keep an open and easily extendable design, no assumptions of the limitations of the functionality of the shader graph nodes have been made. New types of nodes might be introduced in the future that offer other type of functionality than blindly performing operations based on input and generating output. There might for example be conditional nodes that based on some condition outputs one of the input nodes (a kind of switch node), nodes that alter other types of environmental attributes of the shader etc. To make this possible, all nodes are responsible for implementing their own compilation.

The compilation pass is initiated by asking each output node in the shader graph to compile itself through the common Node interface that all nodes implements. From there it is up to the implementation of the output node how the compilation proceeds. A typical node starts by asking each of the nodes connected to its input slots to compile themselves (through the common Node interface) and then proceeds by compiling itself, in effect causing a depth-first-search of the graph. But then again, all nodes need not follow this scheme, an imaginational conditional node (as described above) might for example choose to only compile the subtree of one of its input slots, so that the remaining nodes connected to the other input slots never might be visited.

During node compilation each node adds source code fragments program variables to the fragment tree. When a node is finished it updates a global map that associates node output slots with real program variables by adding entries for associations between all the output slots of the current node and the program variables they have been represented by in the fragment tree. This map can then be used by other nodes that need to know which program variables to get their input from.

Before a typical node can start compiling itself it traverses all of its input connections and checks the output slots they reach against the global map of slot-to-variable associations. If an output slot is encountered that does not yet exist in the global map this means that the node it belongs to has not yet been compiled, and will therefore be asked to compile itself before continuing. When all input connections have been traversed all dependant nodes will have been compiled and the node can then continue with compiling itself. The first step is to add its source code as a new source code fragment in the fragment tree. It then links the input variables of the source code fragment to the real program variables that were found in the global map during the traversal of the input connections of the node.
When this is done it creates new program variables and links them to the output variables of the source code fragment. Finally the global map of slot-to-variable associations is updated by adding the output slots of the node with associations to their respective program variables.

By checking against the global map of slot-to-variable associations before asking a dependant node to compile itself it is made sure that each node is only visited and compiled once. Nodes could otherwise be compiled several times since the graphs that are allowed are DAGs (Directed Acyclic Graphs); one output slot can be used as input for several nodes (see 5.2.1 Graph Structure).

6.4.3 Global Resolving

During this phase the implementation of the globals (as specified in the global libraries) that are referred to in the shader are added to the intermediate program representation. For more information about globlas, see 5.4.3 Globals. Up until this point, the global variables in the program representation have been nothing more than empty placeholders holding the names of the globals they refer to.

The algorithm used for this step maintains a list of all globals that have been evaluated in the program so far, along with a pointer to the program variable that holds the calculated value of each respective global that has been evaluated. When the algorithm starts this list is empty.

The fragment tree is first traversed, and all globals that has been written to, and therefore have been defined directly in the shader, are added to the list of evaluated globals. Then the fragment tree is traversed again, and each global variable is checked to see whether the corresponding global has yet been evaluated (by checking if it is in the list of globals evaluated so far) If it has already evaluated the global variable is replaced by the variable holding the actual value of that global. If the global has not yet been evaluated, it will be looked up in the currently loaded global libraries. The implementation of the global found in the global library is then added to the intermediate program representation, the global is added to the list of evaluated globals along with the output variable of its implementation. It is possible that the implementation of a global uses references to other globals, so the algorithm traverses the fragment tree again and keeps doing so until all global variables have been replaced by real variables.

6.4.4 Type Conversion

This step of the compilation process makes sure that all variable links in the intermediate program representation are type-valid, i.e. each fragment input variable is linked to a program variable of the same type.

All source code fragments are traversed and each input variable of each fragment is checked against the program variable it is linked to, to see if their types match. If their types do not match the system will try to find a way to convert the program variable into a new program value with the correct type, and link the fragment input variable to this new program variable instead of the old one.

To perform the conversion, the system starts out by searching for an optimal set of conversion rules for performing the conversion. This is done by performing a graph search in the type conversion graph (see 5.4.1 Automatic Type Conversion). The type conversion graph is a directed graph with weighted edges, where each node represents a data type and each edge
represents a possible conversion between the data types it connects. The weight of an edge represents the processing costs involved in performing the corresponding type conversion. The goal of our search algorithm will be to find the path from the node of the source type to the node of the target type with the minimal processing cost. This is a well-known problem and can be solved by the A* (A-star) algorithm. If there is no path between the two nodes the system has no way of performing the desired type conversion, in which case an error message will be outputted and the compilation process will be terminated. If a path is found, we will have an optimal set of conversion rules to apply to perform the type conversion.

The conversion is realized by adding the source code fragments of the conversion rules into the fragment tree, and linking them together in the same order as they appear in the conversion path. The program variable to convert is linked to the input of the first conversion fragment, and the output variable of the last conversion fragment will hold the value converted to the desired type, and can finally be linked to the original fragment input variable for which the conversion was done.

A program variable can be linked to several fragment input variables, and the same type conversion of one variable could therefore be requested by several input variables. To avoid performing the same conversion twice, each program variable has a list of all other program variables that represent converted data type versions of it. Before any type conversion is initiated, this list of already existing converted versions of a variables is traversed, and if a match is found that variable can be used instead of performing the same conversion again.

### 6.4.5 Vertex vs. Fragment Program

As described in 2.2.2 The Hardware Rendering Pipeline, shaders are constructed as a pair of two programs; one for specifying the processing to be done per vertex (vertex shader), and the other for specifying the operations to be carried out per pixel (fragment shader). So far in the compilation process we have not made a distinction between the two, but handled all source code fragments in a uniform way. The task of the Context Selection phase is to separate the source code fragments into the two different program contexts, and to create necessary links between the two programs. The algorithm used by our system for selecting program context is based on the optimization ideas presented in the paper Interactive Shader Development (Jensen, 2007).

When a source code fragment is added to the fragment tree it is also flagged as belonging to the vertex, pixel, or generic context. If a fragment is flagged as belonging to the vertex context this it can only be executed as part of the vertex program, the pixel context means it can only be executed as part of the fragment program, and generic means it can be executed in either one of the two. It is up to the different node types of the shader graph to set the context of the source code fragments that they add to the fragment tree. Nodes whose functionality is specified via the XML-interface (see 5.3.3 Node Implementation Interface) can be used for either context (vertex, pixel or generic), which one to use can be specified with a special tag in the XML-interface. Like the source code fragments, program variables also have a context flag specifying which context they belong to.

In the fragment tree (the intermediate program representation, see above) it is perfectly possible to specify that a fragment belonging to the pixel context should use a program variable that has been created and written to by a fragment in the vertex context. But since the two fragments actually are parts of two different programs, the pixel and vertex shader
program respectively, this means that the data these two fragments share will have to be communicated between the two programs. The means offered by today’s shading languages for accomplish this is varying variables. As described in 2.2.2 The Hardware Rendering Pipeline, varying variables are variables that are written to by the vertex shader, and which are then interpolated over the fragments of each line or triangle that is drawn, and then presented as input to the fragment shader that is run for each fragment. It is not allowed to specify that data should be communicated in the opposite direction (meaning that a fragment in the vertex context should use output from a fragment in the pixel context). Data could never be communicated from the fragment shader to the vertex shader since the vertex shader is executed before the fragment shader, and a connection of this type in the fragment tree would result in an error message during compilation. (It would actually be possible to communicate data from the fragment shader to the vertex shader if rendering was done in several passes, but this is not supported in our system. In these cases the fragment shader could write their output to texture memory during the first pass, and the texture memory could then be read by the vertex shader during the second pass).

The context selection phase is executed in two passes. The task of the first pass is to choose a context for all fragments that is flagged with the generic context (being executable in either context). The algorithm used for this does a depth-first-search traversal of the fragment tree and for each fragment with generic context that is visited a decision as to which context to include it in. The pixel context is selected for a fragment only if it is directly or indirectly dependant on data from another fragment in the pixel context, and otherwise vertex context is chosen. This will make sure that as much processing as possible is done in the vertex shader which will be cheaper since the vertex program normally is executed less frequently than the fragment shader. At the same time we avoid making vertex fragments depending on pixel fragments.

The second pass creates varying variables for data that need to be communicated from the vertex program to the pixel program. For each fragment in the fragment tree that belongs to the pixel context, it is checked if any of the variables it uses are flagged as belonging to the vertex context. If such a variable is found, a new varying variable is added to the fragment tree, a source code fragment with a single line copying the value of the original variable to the varying variable is added to the tree. Finally, the variable in the source code fragment in the pixel context is substituted by the varying variable. It is assumed that a copying of one variable to the other is made with the syntax “A = B;” which is the case of all common high level shading languages today (this could easily be extended to a more general implementation with some kind of language-profile system where a variable copying expression among others could be specified for each language profile).

One thing that has to be considered is that not just any value is suited for being interpolated as a varying variable. An interpolation between two normalized vectors will for example result in a non-normalized vector. This is solved by making it possible to specify a corresponding “interpolation type” for each data type, which will specify the type that a given data type is transformed into when it is interpolated. If we for example have a data type \textit{wnormal} which represents a normalized vector in world space, we could specify that the interpolation type of \textit{wnormal} is \textit{wdir}, which is a directional (non-normalized) vector in world space. When the system creates a varying variable link it checks if there is an interpolation type specified for the data type of the variable, and if so declares the new varying variable with this interpolation type.
6.4.6 Fragment Ordering

Up until now the representation that we have of our shader program (the fragment tree) has all the shader fragments ordered in a tree structure. Before the shader programs can be outputted the fragments have to be ordered in a linear fashion, this is the task of the Fragment Ordering phase of the compilation process.

The fragment tree form a directed acyclic graph where the source code fragments are nodes and the program variables are edges. A fragment is dependent on (and will therefore have to be executed after) another fragment if the first fragment uses a variable that is created by the second. Seen as a graph this means that node B has to be executed after node A, if there is a path from A to B in the graph. This is a well known problem that can be solved with topological sorting algorithms with linear time complexity. The nodes are ordered by such an algorithm and are then separated into two lists, one with the fragments of the pixel context and the other with the fragments of the vertex context.

6.4.7 Symbol Generation

The program variable objects that are created as part of the intermediate program representation each represents a unique variable in the final shader program, but they have not yet been given symbol names to be used in the final generated source code. Each variable is given a name upon creation, but there is no check for collisions between these variable names so they are just to be regarded as a name base or as a hint of what to use as final name. During the Symbol Generation phase all variables will be given a unique name.

This phase of the compilation process will in part be dependant on the shading language used, since care has to be taken to not give variable names that are illegal or reserved for special use in the shading language. So far the system only has support for GLSL, but the system could easily be extended to support a shading language profile system, in which case this phase would have to be part of the implementation of the respective shading language profiles.

The algorithm used for symbol generation in the prototype is as follows: The list of program variables are traversed, and if their name hint has yet not been encountered it is used as final symbol for the variable. If the hint has already been encountered, new symbols are generated by appending the name hint with an increasing number until a unique symbol is found (see figure 6.2).

```
[ ] -> S
[List of variables] -> V
for each variable v in V
  v.hint -> v.name
  0 -> i
  while v.name in S
    i + 1 -> i
    v.hint + i -> v.name
  add v.name to S
```

*Fig 6.2* Pseudo code for the symbol generation algorithm.

To improve the readability the symbols are also given a prefix that shows the variable type. Attribute variables are prefixed with “a_”, constants with “c_”, shader parameters with “p_” etc.
GLSL has built-in support for integrating shaders with standard OpenGL states, which makes it easier for applications written for the Fixed Function Pipeline to start supporting shaders. Various OpenGL states are available with read-only access from within the shader and appear as variables with reserved names prefixed with “gl_”. This is taken advantage of by the implementation of our system through support for standard OpenGL vertex attributes, which facilitates setting up a shader for execution from within an application. The system has a list of names for standard vertex attributes paired with their respective GLSL reserved symbol counterpart. The names of attribute variables declared in the program are checked against this list, and if a match is found, the attribute variable is substituted for a local variable with the corresponding GLSL symbol. So creating an attribute variable called “POSITION” will result in it appearing as “gl_Position” in the final shader code, “COLOR0” automatically gets translated to “gl_PrimaryColor” etc. Again, this is something very specific to the GLSL language, but these kinds of language specializations can easily be switched with some language profile system in the future, thanks to the clear separation and independence of the different phases of the compilation process.

6.4.8 Source Code Generation

This is the final phase of the compilation process. When this phase is reached, the shader program is represented by two lists of source code fragments; one list for the vertex shader and the other for the fragment shader. We also have a list of all program variables, each with a unique name ready to be used in the source code. The only thing remaining is to output the final source code.

As is the case with the Symbol Generation phase, some aspects of this phase will be dependant of the shading language used, so similar but different implementations for this phase will be necessary for supporting several shading languages.

The process of outputting the shader source code is pretty straight forward. First all declarations of global variables (uniforms and vertex attributes) and constants are outputted, since these declarations should appear before the main program body in GLSL. The program variable list is traversed and a row declaring each global variable that is visited is appended to the shader program source code.

The next step is to output the actual source code of the program body. Each fragment list is traversed from the first fragment to the last. For each fragment that is visited, all the variable references in its source code are replaced by the unique names of the variables they refer to, and then the source code is appended to the shader program. When all fragments have been visited and outputted, the shader programs are finished.

Along with the shader program a data structure is also generated which holds information about the vertex attributes, externals and shader parameters (such as name, type, symbol used in the source code etc.) so that these variables can be set up properly by the application that is going to use the shader.

6.5 Rendering

The 3D rendering performed in the preview window of the prototype system is done by a custom engine that is based on OpenGL. The rendering engine uses the API described in 5.5.2 API in combination with the OpenGL API to setup vertex attributes, shader parameters and externals correctly before rendering.
Shaders are fully supported in OpenGL 2.0, but not in OpenGL 1.5 which is installed on the computers to be used for evaluating the system (see 4.7 System Requirements). It is still possible to use shaders on OpenGL 1.5 installations if the graphics driver on the system supports the necessary OpenGL extensions `GL_ARB_shader_objects` and `GL_ARB_vertex_program` (OpenGL.org, OpenGL Extension Registry), which is the case with the computers to be used for the evaluation of the system. To make the system as compatible as possible it has been implemented to use OpenGL 2.0 functionality if possible, or otherwise simulate the shader functionality of OpenGL 2.0 through the use of the above mentioned extensions if available.
CHAPTER 7

The Resulting System

7.1 Overview
The resulting system is a general framework for visual shader programming with a set of general tools for editing graphs, previewing shaders, browsing the generated shader source code and more. We will in this chapter focus on the user experience of the system. Other aspects like advantages and possible issues with the abstraction features, application integration etc. will be discussed in chapter 9.

We will start by giving a demonstration of how the system is used by going through some typical usage scenarios, and will then show some examples of shaders that have been created in the system. The generated source code for the shaders in this chapter is available in Appendix C.

7.2 System Usage Example
We start with an empty shader project. At this point the Graph Window is completely empty and the Preview Window only shows a wireframe (non-solid) representation of the currently loaded 3D object (a simple cube is always loaded by default). The wireframe view is always used when there is no valid shader available (either because the graph is empty or because it is invalid in some way).

The first thing we will do is to create an Output Node. All graphs need an output node that specifies the final output of the shader. To create the node we right-click in the Graph Window and select the node type to create from the appearing menu (see figure 7.1).

![Fig 7.1 Creating a graph node.](image-url)
We then create another node called *Phong*, which implements *Phong*’s lighting model (*Phong*, 1974). This node has five input nodes: *Diffuse*, *Specular*, *Ambient* and *Shininess* correspond to the four surface parameters of the *Phong* equation, and the *Normal* input parameter which specifies the surface normal parameter of the equation (see figure 7.2).

![Fig 7.2 The Phong and Output nodes.](image)

We will start by using the normal vectors stored at the vertices of the geometry as the normal vector for the lighting calculations. Before we can use a vertex attribute (vertex normal, texture coordinates etc) in our graph a vertex attribute declaration has to be created. This can either be done manually (by specifying attribute name, data type etc.) or we can copy the attribute definitions from an existing 3D object. We will here show how to do the latter.

![Fig 7.3 Declaring Vertex Attributes.](image)

We right-click on “*Attributes*” in the explorer window and choose “*Import Attributes from Mesh*...”. This opens a dialog window (see figure 7.3), which shows a list of the vertex attributes that are defined for the 3D object that is currently loaded in the Preview Window. Our cube object (which is the one loaded by default) has five different vertex attributes stored at each vertex, but we are only interested in *NORMAL* and *TEXCOORD0*. To add these two vertex attribute declarations into our project we mark them in the list and press “*Import*”. These two vertex attributes now appear in the “*Attributes*” subtree of the Explorer Window and can from now on be used as input in our shader graph.
We specify that the *NORMAL* vertex attribute should be used as input for the *Normal* input parameter of the *Phong* node by clicking on the *Normal* input slot and then choosing our attribute from the appearing menu. If no values are specified for the other input parameters of the *Phong* node, a set of default values will be used, but we decide to specify some of these parameters ourselves. To specify a constant value for a parameter directly, you click on it and then select “*Constant*” from the appearing menu. A control will then appear beside the input slot, which can then be used to specify the value of the parameter (see figure 7.4).

![Fig 7.4 Setting up node parameters.](image)

Finally we specify that the resulting color from the calculations of our *Phong* node shall be used as the final output from the shader, by connecting the output slot of the *Phong* node with the input slot of the *Output* node (see figure 7.5).

![Fig 7.5 Creating a connection.](image)

To preview the shader that we have created so far we press the *Compile* button on the toolbar. The system will generate a shader program from our graph, and will output the shader source code in the *Output Window* in the bottom of the application window. The *Preview Window* will also be updated to show the new generated shader. We will now have a simple shader which can draw plain and single-colored surfaces that are lit with Phong’s lighting model (see figure 7.6).
To make the shader a bit more interesting we will now add a Normal Mapping effect. Normal mapping is a way of creating the illusion of a highly detailed mesh by using a low polygon mesh and a special form of textures called Normal Maps. A Normal Map is an image that instead of color values stores a surface normal for each pixel, where the x, y and z components of the normal is encoded as the red, green and blue components respectively. Instead of using interpolated vertex normals in the lighting equations of the shader, the normal is calculated by sampling a normal map for each fragment that is rendered. The result can be very effective and gives the illusion of the object having a highly detailed surface.

We start by adding a new sampler (texture reader) to our project. This is done by clicking on Samplers in the Explorer Window and then selecting “New Sampler…” from the appearing menu. This will open a new dialog window where the user is asked to specify a name for the sampler, which type of sampler it is, and an image file that the sampler will read from (see figure 7.7). We create a 2D image sampler, call it “MyNormalMap” and specify a special image file that contains normal vectors encoded as color values for each pixel (as explained above). The system will still treat this image as any ordinary 2D image file.

We then add a new node of the type NormalMap to our graph. This node samples a color from a texture and decodes the color into a vector. It has two input parameters; Texture which specifies which sampler to use and Coords which specifies at what coordinates in the texture to do the sample. It has only one output, Vector, which is the resulting vector that is decoded from the color obtained from the texture sample.

**Fig 7.6 Phong shader applied to different 3D objects.**

**Fig 7.7 Creating a Sampler.**
We specify our newly created sampler as the input of the Texture parameter, and the Vertex Attribute TEXCOORD0 which are evenly spaced two-dimensional coordinates that are specified at each vertex in the geometry of the 3D objects we use. Finally we use the resulting decoded vector as input to the Normal parameter of the Phong node (instead of the previous Vertex Attribute NORMAL) by connecting the two slots (see figure 7.8).

![Fig 7.8 The shader graph of our Phong + Normal Map shader.](image)

As a result, each point of the surface will now get a unique normal vector (as opposed to the previously interpolated vertex normals), obtained from our special image file (see figure 7.9).

![Fig 7.9 Phong + Normal Map shader.](image)

Finally, we will show how to wrap an image (texture) over the object surface instead of using a constant color. We add a new sampler to our project which uses an ordinary image (as opposed to the vector encoded image used earlier). We call this new sampler MyTexture. We also add a new node of the type 2DTexture, which implements a plain texture sample from a 2D texture. We specify our new sampler and the TEXCOORD0 Vertex Attribute as input to this node, and connect the resulting color output of the 2DTexture node to the Diffuse parameter of the Phong node. Now the diffuse color component of the Phong equation will be obtained from our texture instead of using a plain color constant (see figure 7.10).
Fig 7.10 Phong + Normal Map + Texture Map shader.
7.3 Examples of Shaders Created in the System

7.3.1 Environment Mapping Shader

This shader is an implementation of the environment mapping algorithm that was covered in section 5.4.3. It calculates the reflection vector at each point on the object surface, and uses this vector to sample a cube map texture. The same cube map texture has also been applied as background in the scene (see figure 7.11).

![Figure 7.11 Environment mapping shader applied to a sphere.](The cube map texture was downloaded from http://www.humus.ca)
7.3.2 Glass Shader

This shader simulates a glass material. It calculates the reflection and refraction vectors at each point on the object surface by using a surface normal that is obtained from sampling a normal map. The reflection and refraction vectors are then used to sample a cube map texture two times, to retrieve the color of reflected and refracted light respectively. The obtained colors are then mixed (with proportions that are dependant on the steepness of the reflection angle) to determine the final color (see figure 7.12).

Fig 7.12 Normal Mapped Glass shader applied to a torus knot.
(The Cube Map texture was downloaded from http://www.humus.ca)
CHAPTER 8

Evaluation

8.1 Introduction
To get feedback on the system design and to see how visual shader programming would be perceived by actual users, the system was tested on a group of 30 students attending a course in computer graphics (Focus Group, Course DH2413). The test was done as a laboratory assignment where the students were asked to create various shaders in the system. The students were also asked to fill out a form about their thoughts and impressions of the system.

8.2 User Group
All students had previous programming experience and were familiar with concepts of computer graphics, some (but not all students) also had some earlier experience of shader programming. This is a quite good representation of the targeted user group since the system is meant to be used by both shader programmers and users without shader programming experience (artists etc.). The inclusion of students without earlier experience in shader programming will also make it possible to get an indication of whether it might be possible to use the system as a tool for aiding in learning shader programming.

It would have been interesting to also try the system on users without any kind of programming experience at all, which often is the case with artists. On the other hand, one might argue that general programming experience should not have that big of an influence on the ability to designing graphs, and that it on the contrary probably would be more of an advantage to have a deeper understanding of 3D graphics concepts and experience from graph-tools from other graphics related applications, as is the case with most 3D artists.

8.3 About the Laboratory Assignment
The laboratory assignment (see Appendix B) consisted of 4 obligatory exercise, and a fifth that was voluntary. The students either worked individually or in pairs. In the first exercise, the user learned the basics of the system like creating nodes, defining vertex attributes etc. and resulted in the creation of a very simple single-node graph. The second exercise introduced textures and the construction of graphs by connecting several nodes, and resulted in a shader that applied a blending of two textures to an object. The third exercise covered various lighting operations, and the fourth exercise covered normal mapping. In the fifth exercise the user was asked to implement an own node using the node implementation interface (see 5.3.3 Node Implementation Interface). Since this exercise required some experience in shader programming it was voluntary.
The overall impression of the system was very positive. All students managed to complete the exercises within the 2 hour timeframe. Although some students needed a bit of help (the two color inputs of the mixing node in exercise 2 were for example in some cases confused with each other), most of the students managed to complete the exercises independently.

Several students chose to also do the last exercise about the node implementation interface. This exercise took more time to finish than the others, partly because it simply involved more work but also because the system still lacks good debugging features. The system also has to be restarted to update changes made in the implementation of a node. Although this is a big problem, it is not caused by a weakness in the system design itself, but rather just a case of missing practical features. The workflow could be improved substantially by just adding an interface for editing node implementations directly in the graphical user interface, and allowing for instant refreshing of node implementations from within the application. Despite the lack of commodity features in the node implementation interface, all but one student who tried the last exercise also completed it (even a student who had no earlier experience of shader programming managed to implement the node correctly). One student even came up with an own idea and created a node that implemented an artistic “cartoon” effect.

The students that were the most positive were those who had earlier experience from shader programming, since they knew the work involved for creating shaders by hand they appreciated the more efficient form of shader development.

### 8.4 Evaluation Form

During the last session the students were asked to fill out a form about their impressions of the prototype. The form (see Appendix A) contains questions about background experience, opinions on different aspects of the system and finally some space for own comments. Not all 30 students took part in the evaluation since the form was not handed out during the first session, but in total 16 students answered the form. We will here present the results of the form evaluation.

#### 1. Previous experience

The first question of the form was regarding previous experience. It showed that all students had experience of programming in C++ or Java, that about two thirds of them had experience of shader programming and that all had at least some experience of 3D modeling.

#### 2. It feels natural to design shaders as graphs.

This question is very important since it shows the overall impression of the technique of designing shaders as graphs. As shown in the diagram (see fig 8.1) the response was very positive. From this we can draw the conclusion that graphs are an intuitive way of expressing shader functionality.

---

**Fig 8.1 Form question 2.**
3. It is easy to understand the meaning of the nodes in the program.

This is a question that was meant to give an indication of how the layout of the node windows was perceived, and whether the functionality of the nodes was clear or not. As explained in 5.3.2 Graph Interface, a design decision was made to embed leaf nodes in the window of their parents to give a more structured and compact view. It seems as if the layout seemed logical to most users (see fig 8.2).

After having talked to the students it became clear that the node title alone might not always be enough to fully understand the functionality of a node and its parameters. One student suggested the inclusion of a help-button in each node, which when pressed could show a short description of the node and its functionality in a popup window. This seems like a great idea and will be implemented in the future.

4. It is easy to understand the meaning of the connections in the program.

This question was meant to show if it was easy to grasp the concept of dataflow in the graph, if it was clear what slots were input and output etc. As indicated by the diagram (see fig 8.3), it seems as if the concept of connections was not difficult to grasp. After having observed the students during the session, there seemed to be no confusion between input and output slots. No attempts to create illegal connections like connecting two input slots were spotted.

5. The system has good (visual) feedback.

As with all systems for creation of visual material visual feedback is very important. Most users seem content with the feedback level of the system (see fig 8.4), but some seemed a little bit hesitant (a third of the students marked nr 5).

After having observed the students working with the program and having helped a few of them debugging their graphs, it was clear that it was sometimes easy to confuse one sampler with another in shaders with several samplers. All that distinguish one sampler from another in a graph is the names they have been given by the user. This is an example of lack of visual feedback, and the problem has been remedied in other systems (e.g. Visual Material Editor for Unreal Engine®, see 3.2.6 Commercial Tools) by adding
a preview-window displaying the texture in nodes dealing with samplers. This seems like a good solution and has here been proven to be valuable. It will be implemented in the future.

6. Shaders can be created quickly in the program.

This was something that everyone totally agreed on (see fig 8.5). Most impressed were those who had earlier experience of shader programming and therefore knew the amount of time necessary for writing and debugging shaders by hand.

7. I think it is possible for users without shader programming experience to use the program.

This is a very interesting question since the main purpose of the system is to allow users without shader programming experience to create shaders. Although everyone does not seem totally convinced, most students seem to believe the system fulfills this goal (see fig 8.6). However, the relevance of the answers could be discussed since it is still based on opinion. More interesting is the fact that even the students who did not have experience from shader programming managed to complete all tasks in the program, which shows it is in fact possible for users without experience in shader programming to use the system.

8. I think the program can be useful for an experienced shader programmer.

The relevance of this question could be discussed, since none of the students actually were experienced shader programmers. But since it is an interesting aspect of the usefulness of the system, and since many of the students were experienced in other areas of programming it would be interesting to hear their opinion. About two thirds of the students seem convinced even experienced shader programmers could benefit from the program (see fig 8.7). A survey involving actual experienced shader programmers would have to be conducted in order to get a good answer to this question.
9. Free Comments

The form also had a space for free comments. Some of the received comments are cited here below (translated from Swedish):

"Good initiative! Good tool for understanding shader programming."

“I’m very impressed with the tool which makes it possible to create code through drag&drop. It makes it possible to ‘play’ with effect creation and then learn how to write the source code for them. Very pedagogic.”

"A very quick and easy tool for creating different shaders. It would for sure be useful if it gets bigger and has more available choices. Its strengths are that it is quick and easy to use."

"Great program, looking forward to the final product!"
Discussion

9.1 Generality and Flexibility

There are several shading languages available today for writing shader programs, and they are evolving quickly as new rendering hardware features are introduced. To not be dependant on a specific language and to be able to keep up with the evolution, one important goal of the system design has been to make the system as general and open as possible.

This goal has been met by creating a simple framework with a set of general tools for editing graphs, previewing shaders etc., and by leaving all actual language features like data type definitions, node implementations etc. to be specified externally. This makes it possible to alter the implementation of the visual shading language itself independently of the application. The user can customize or create new nodes, and can even design a new visual shading language by defining data types, conversion rules and abstraction features, all through the external XML-interface offered by the system.

Since the shader language implementations of the graph nodes never have to be parsed in the compilation process, but are just linked together by substitution of symbols in the source code blocks, the system poses few restrictions on the shader language used. The system could easily be extended to work with other languages than GLSL, the shader language used during this project.

9.2 Abstraction Features

The system has been armed with several features for making it easier for users without shader programming experience to use the system.

The globals system (see 5.4.3 Globals) offers a way of defining default ways of calculating various data, so that node implementations can refer to these definitions rather than having to implement them itself, or alternatively having to forcing the user to specify these calculations in the graphs. This makes it possible to hide complex implementation details from the user, and also helps reusing the results of calculations between nodes.

As explained in section 5.4.2 Custom Data Types, the custom type and conversion rule features make it possible to automatically handle conversions between not only physical data types, but between concepts like coordinate spaces, normalization of vectors and distinctions between data.

The testing of the system by a group of students (see chapter 8) assignment showed that it is in fact possible for users without previous experience in shader programming to create
shaders in the system. But, since all students had general programming experience, no conclusions can be drawn about users with no form of programming experience at all, which might be the case with 3D artists. One might on the other hand argue that 3D artists probably would have more experience of using graphs, which is common in other art applications, but a survey involving these users would have to be made before we could draw any such conclusions.

9.3 Application Integration

We have investigated several ways of facilitating integration of the shaders created in the system with third party applications like game engines.

The most important design choice that was made to support application integration was to move graph compilation from our system to the application using the shaders (see 5.5.1 Deferred Compilation). Thanks to the clear separation of functionality in to nodes and dependencies between nodes in the form of edges, the visual shader development process will generate much more structural information about a shader than ordinary textual shader programming. When a shader graph is compiled into shader program source code, all structural information about the various components and their dependencies given by the graph is lost. With access to the graph representation of a shader it will be much easier for an application to adopt the shader to different lighting and environmental conditions, before it is compiled and then used. To make this possible, our system stores the shader graphs in their non-compiled form, and these can later be loaded, altered and compiled in a third party application by using an API that comes with our system (see 5.5.2 API).

Another application integration feature that has been implemented in our system is an interface for exposing application variables to shaders created in our system (see 5.5.4 Externals). This makes it possible for the shaders in the system to access different application variables (lighting data, transformation matrices etc.).

We have also investigated ways of adding support for shader parameterization in our system, which will make it possible to reuse a shader for different materials (see 5.5.3 Shader Parameterization).

To use our system with a third party engine, we could start by creating an External Library (see 5.5.4 Externals) with the application variables that the engine offers, so that the shaders can start accessing engine variables. Depending on how much the engine will need to interact and alter the shaders for different rendering conditions we could also create some custom nodes that represent different material types that are supported by the engine. By using these engine-specific nodes as root nodes in the shader graphs, it will be easier for the engine to identify and switch different parts of the shader graph to adopt the shader for various lighting, environmental or performance conditions etc.

The Globals system (see 5.4.3 Globals) will also facilitate integration and alteration of the shaders. If the engine for example would need to use a shader for a mesh that uses vertex blending (Akenine-Möller, 2002), which alters vertex positions, the engine could simply insert a node in to the graph that performs the blending operations and which writes the resulting vertex position to the global variable position, effectively overriding the default definition of this global which is to simply copy the value of the position attribute of each vertex. Since the other nodes refer to this global, their calculations will still be correct (e.g. in calculation of view direction vectors for lighting calculations etc.). Note that the position global is not something that is built into the system, but is a good example of how
the global library should be used. The user is free to define any kind of globals that might be need.

Unfortunately, there has not been enough time for testing the integration features of our system with a commercial engine. For now, the only engine that it has been tested on is an own engine that is used for rendering the preview window of our system. This engine uses our API for setting up the shaders for rendering and for exposing engine variables to the shaders. To get a good evaluation of the possibilities of our integration solutions the system would have to be tested with a more complex engine. The features for shader parameterization and for exposing application have just been an investigation of ways to apply already existing concepts to visual shader programming, the ideas themselves have already been proven useful in normal shader development. The aspect that would be most interesting to investigate in such a test would be the possibilities offered by making the structural information of the shaders available to the engine.

9.4 Educational Aid

The learning curve of shader programming is very high. Students need not only learn a completely new programming language, but also have to learn about various shading concepts like samplers, vertex attributes etc. To use the shaders, the application side of setting up a shader for rendering is yet another area that has to be studied.

Instead of having to cope with all the details involved in constructing a complete shader program at once, and having to setup the shader correctly using an application programming interface like OpenGL or DirectX, our system makes it possible for a student to learn about various shader concepts and the possibilities of shader development before they need to know how to write shader programs by hand. The abstraction features of the system make it possible for the students to quickly get their hands on the technology and start experimenting and learning about different shader concepts. Since the source code for the created shaders is shown in the user interface, the user can always see how the effects they create are implemented in the shader language.

The system also makes it possible for the students to start implementing parts of a shader, like for example a lighting model, by creating own nodes. In this way, the students can start learning programming in the shader language without having to construct complete shader programs.

The feedback from the students that tested the system was very positive. In fact there was not much negative feedback, which might pose some questions. Since the test was done as part of a mandatory course assignment, this might have affected the students and made them feel inclined to rather give positive than negative feedback because they were being graded on the course. They might also have felt less inclined to give negative feedback because the developer of the system was present.

9.5 Performance

How does the performance of shaders created in our system compare to those written by hand? The answer is that it depends highly on the level of abstraction that is used. If only nodes that implement atomic operations would be used in constructing a graph, and no automatic type conversions, we would have the same control as when writing shaders by hand.
The way in which the source code of each graph node is linked together to form a shader program normally leads to a program with more temporary variables than when writing source code by hand, since temporary variables are generated for almost every node output. This should not have an impact on the performance of the final shader though. The merging of variables, or register allocation (Appel, 2002), is a common compilation process that will remove redundant variables when the shader source code is compiled using the compiler of the specific shader language.

As nodes start getting more complex and implement several instructions, there is risk of having the same calculations being performed by several nodes, instead of doing the calculation once and reuse the calculated value for all nodes that need it. The globals system helps avoiding this (see 5.4.3 Globals), and can reduce much redundant calculations if used wisely.

Another performance problem can arise when using the automatic type conversion system, when mixing nodes that perform calculations in different coordinate spaces. Our system has no built-in knowledge of different mathematical spaces etc., but the space in which the calculations of a node is performed can be defined by using different data types for vectors in different spaces. Although the system for automatic type conversion will always make sure that types (and therefore coordinate spaces) are converted correctly between the nodes, a more efficient solution might be possible if for example some of the nodes instead could have been implemented to perform their calculations in the same space as other nodes in the graph, in which case less conversion nodes would have to be inserted. A possible solution to this problem could be to create different variants of a node for handling its calculations based on different input types (e.g. different vector data types with different coordinate spaces etc.), and to then let the system try every combination of variants of these nodes in a graph to find the graph that has the least number of needed conversion nodes. The running time of such an exhausting algorithm would be exponential though, and might be infeasible. P. D. E. Jensen discusses a similar problem (Jensen, 2007) involving finding an optimal way of inserting space transformation nodes into a shader graph. As he points out, the problem is very complex, but he also proposes a simpler greedy algorithm that performs well. It could be possible to use a similar greedy algorithm for solving our problem. The algorithm can traverse the shader graph from the root node (nodes with no outgoing connections) to the leaf nodes (nodes with no incoming connections) and for each node that is visited which offers variants with different data types, we could choose the variant that would need the least number of conversion nodes to be inserted between this node and its neighbors. This would not necessarily generate an over all optimal solution, but has the potential of generating more efficient solutions than our system as it exists today. Such a solution would have to be investigated further before any real conclusions about its practicality could be drawn.
CHAPTER 10

Conclusions

10.1 Conclusions
In this report we have presented the design and implementation of a system for developing real-time shaders with visual programming.

To keep up with the fast evolution of real-time shader technology, we have focused on an open and general system design. This has led to a system consisting of a framework with a set of general tools for editing graphs, previewing shaders etc., and with all actual language features specified externally. This makes it possible to alter the implementation of the visual shading language itself independently of the application. The user can even design a new visual shading language by defining own operations, data types and rules.

One of the main goals of this project has been to find ways of making it possible for users without experience in shader programming to be able to develop shaders. We have investigated ways of making this possible with visual shader programming, and presented ideas for how the system can hide complex details from the user and handle data type conversion and space transformations etc. in a general and flexible way. These features have been implemented in our system, and a testing of the system on a group of users showed that it is in fact possible for users without earlier experience in shader programming to create shaders in our system.

We have presented several features for facilitating the integration of the shaders created in our system with third party applications, like for example game engines, and have discussed how the visual programming approach could be taken advantage of for customizing shaders from within an application (e.g. a game engine).

The learning curve of shader programming is very high since it is a complex area that not only involves a completely new programming language, but also theory about the rendering pipeline and various hardware-related real-time rendering concepts. We have shown how our system can be used as an aid in the learning process, by allowing students to quickly get their hands on the technology and start experimenting and learning about different shader concepts without having to master the programming language. The students can learn from the source code that is automatically generated while constructing shaders in the system, and can start implement and test small parts of a shader without having to write and setup whole shader programs.

Although our system for visual shader programming offers a much easier and faster form of shader development than the ordinary shader programming, and makes it possible for users without shader programming knowledge to create shaders, it can under some circumstances generate less optimal shaders. We have pointed out some possible...
performance issues that might arise from using different abstraction features, and discussed possible ways of addressing these problems.

### 10.2 Future Research

It would be very interesting to investigate how the area of automatic shader simplification could benefit from visual shader programming. While advanced and complex shaders have the potential to generate beautiful and realistic scenes, they also consume a substantial amount of processing power. Often in real-time rendering, image quality has to be compromised in favor of speed to be able to maintain interactive frame rates. A trick that is often used to increase frame rates while preserving image quality is to focus processing power on the parts of a scene that are in the user's focus. By using high quality shaders in the parts of the scene that is close to the observer or otherwise important visually, and cheaper shaders for distant objects and areas out of focus, an impression of a high quality image can be obtained at higher frame rates.

To make this work we will need different versions of each shader, with different levels of quality. If these were written by hand, this would mean much extra work and that maybe twice or three times as much shader source code would have to be maintained. Also, to make changes to a shader, the same changes might have to be made in several places, in every version of a shader program. To solve these problems research is being done on automatic simplification of shader programs, so that cheaper versions of a shader can be generated programmatically. The problem is that making a system understand hand-written source code is very complex, since pure source code is just a list of instructions and lacks a higher level of structural information that is necessary for making decisions for what parts to skip or replace by cheaper solutions etc.

Thanks to the way that shaders are designed in visual shader programming, much more structural information is available. The different components of the shader are clearly separated from each other in the form of the separate nodes of the shader graphs, and dependencies between the components are also very clear (as defined by the edges of the graph). This should make it easier to remove or switch parts of a shader to cheaper solutions (e.g. switching a lighting node to a cheaper one, or replacing a sub-tree handling normal vector calculations like normal mapping with a simple normal vector vertex attribute node).

It would also be interesting to investigate ways of adding support for geometry shaders to the visual shader programming environment, so that tessellation of curved surfaces and other geometry generating features also could be designed visually.
References


DONNELLY, W., *Per-Pixel Displacement Mapping with Distance Functions*, GPU Gems 2, Addison-Wesley, p. 123-136, 2005


FOCUS GROUP, Course *DH2413 Advanced Graphics and Interaction*, 2007, KTH

The user group on which the system was tested.


http://www.ni.com/labview/

MAD SOFTWARE INC. *Shaderworks*, Online resource, accessed August 2007
http://www.shaderworks.com


Specifications for the High Level Shading Language for DirectX.

NVIDIA CORPORATION, *Cg 2.0*, Online resource, accessed January 2008
The Cg Toolkit - A compiler for the Cg 2.0 language, runtime libraries, example applications, and extensive documentation.

NVIDIA CORPORATION, *FX Composer*, Online resource, accessed September 2007
http://developer.nvidia.com/object/fx_composer_home.html

http://www.projectoffset.com/

http://www.opendx.org/index2.php

http://www.opengl.org/documentation/gls1

http://www.opengl.org/registry


Appendix A

Evaluation Form (Swedish)

1. Jag har förkunskaper inom följande områden
   - [ ] Programmering i C, Java eller liknande språk.
   - [ ] Shaderprogrammering (GLSL, Cg etc.)
   - [ ] 3D-modellering (3D Studio, Maya etc.)

2. Trädstruktur känns som ett naturligt sätt att beskriva shaders.
   
<table>
<thead>
<tr>
<th>Instämmer inte alls</th>
<th>Instämmer helt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

3. Det är lätt att förstå betydelsen av noderna i programmet.
   
<table>
<thead>
<tr>
<th>Instämmer inte alls</th>
<th>Instämmer helt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

4. Det är lätt att förstå betydelsen av kopplingar mellan noder i programmet.
   
<table>
<thead>
<tr>
<th>Instämmer inte alls</th>
<th>Instämmer helt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

5. Graden av återkoppling (möjligheten att se resultat av handlingar) i programmet är
   
<table>
<thead>
<tr>
<th>Låg</th>
<th>Hög</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

6. Det går snabbt att skapa shaders med hjälp av programmet.
   
<table>
<thead>
<tr>
<th>Instämmer inte alls</th>
<th>Instämmer helt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

7. Jag tror att det är möjligt för icke programmeringskunniga att använda systemet.
   
<table>
<thead>
<tr>
<th>Instämmer inte alls</th>
<th>Instämmer helt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

8. Jag tror att en erfaren shaderprogrammerare kan ha nytta av verktyget.
   
<table>
<thead>
<tr>
<th>Instämmer inte alls</th>
<th>Instämmer helt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

9. Egna synpunkter och kommentarer
Appendix B

Laboratory Assignment

(Originally in Swedish, translated to English)

Preparations

- Download the following file:
  
  http://www.xmnsoftware.com/agi/shaderlabb.zip

- Unzip shaderlabb.zip into a new folder.

- Start sheditor.exe.

- If the program complains about missing DLL-files, this can be fixed by running this program (will not be needed on the computers in Magenta):
  
  http://www.xmnsoftware.com/agi/vcredist_x86.exe

Tip! The preview window can be detached from the main window by double-clicking the title bar or pulling it off (click-drag). It can then be resized independently of the main window.

Exercise 1 – Colors and Vertex Attributes

1. Start by creating an output node for the graph. Nodes are created by right-clicking in the graph window and then selecting the type of node to create from an appearing menu. So far there is only one output node available, which is Output/Output. This node type has one single input parameter, Color, which specifies to color to be written to the pixels during rendering.

2. Compile with the menu command File/Compile or the corresponding button in the toolbar. The input parameter Color or our node has still not been given a specific value, and will therefore be given a default value. The default value for its data type (color) is white, hence the white cube shown in the preview window.

3. Specify a constant value for the Color-parameter. This is accomplished by clicking on the parameter and then choosing Constant from the appearing menu. A small control is now shown to the right of the parameter. There are different types of control for different types of input parameters. Since our parameter is a color value, a color-pick control will be used. Click in the color control to change the color value for our parameter, and then compile the graph to see the result in the preview window.

4. Specify the vertex attribute COLOR0 as input instead of the constant value. Vertex attributes are data that is stored at each vertex in the geometry to be rendered, like for example texture coordinates, normal, color etc. Right-click on Attributes in the Explorer Window and choose Import Attributes from Mesh. A window will appear, showing the vertex attributes that are stored in the geometry that is currently loaded in the Preview Window (the cube object). Select COLOR0 and click Import. COLOR0 will now appear in the list of defined attributes in the Explorer Window. Now that the attribute has been defined, it can
be used as input for a graph node parameter. Click on the Color-parameter in our graph node and choose Attributes/COLOR0, and then compile. The shader will now use the color values stored at each vertex in the geometry to render the object.

5. Do not forget to save your shader before starting the next exercise.

**Exercise 2 - Textures**

1. Start a new effect (File/New) and create an output node and a node of the type Texturing/2DTexture. This new node type samples a color value from a texture image. It has two inputs: Texture specifies which sampler (texture reader) to use and Coords specifies at which coordinates in the texture to do the sample. The node also has two output parameters; Color is the resulting color (RGB) sampled from the texture and Alpha is the alpha-component of the sampled color (1.0 if the texture has no alpha-channel).

2. In order for the 2DTexture node to work it needs a sampler (texture reader) to use. Add a new sampler to the project, either by right-clicking on Samplers in the Explorer Window and choosing New Sampler, or with the menu command Effect/samplers/New Sampler. Specify a name (for example “MyTexture”) and an image file to associate with the sampler by pressing the button “…” (for example textures/face_color.jpg). The type-field shall be 2D. Press Create.

3. Specify the new sampler as input to the Texture parameter of the 2DTexture node in the graph by clicking on the parameter and choosing the new sampler from the appearing menu. Also, define the vertex attribute TEXCOORD0 in the same way as in the previous exercise, and use it as input to the Coords parameter of the 2DTexture node.

4. Finally, use the resulting color of the texture sample as input to the Output node, by clicking on the Color parameter of the 2DTexture node, dragging, and finally releasing the mouse button over the Color-parameter of the Output node. Compile the graph, and your texture image will now be wrapped over the object in the Preview Window. To see how your shader look on other objects, right-click in the Preview Window and choose another model from the sub-menu Mesh.

5. We will now try to put a layer of rust over our texture. Add a new sampler that uses the texture texture/rust_color.png. This image has an alpha-channel that specifies the opacity of the image, with a value of 1.0 for the pixels that are fully opaque, and a value of 0.0 for pixels that are completely transparent. Add a new 2DTexture node to the graph and specify the new sampler and the attribute TEXCOORD0 as its input. Then add a Colors/Mix node to the graph. This node has three inputs: Color1, Color2 and balance, and one output: ColorMix. This node implements the following function:

\[
\text{ColorMix} = \text{Color1} * (1.0 - \text{Balance}) + \text{Color2} * \text{Balance}
\]

Use this node to mix the colors from the two texture nodes, and use the alpha-component of the rust texture sample as input to the Balance parameter. Finally, connect the resulting color mix to the Output node and compile. The Preview Window should now show an object with the same texture as earlier, but with a layer of rust on top of it. (Note: make sure that the texture nodes are connected to the right input parameter of the Mix-node. A balance of 0.0 results in Color1 as output, and 1.0 results in Color2 as output.)
Exercise 3 – Lighting

1. There is a node type Lighting/Phong that implements Phong’s lighting model. Start a new shader and add the nodes Lighting/Phong and Output/PerPixelOutput to the graph. The Phong node has the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diffuse</td>
<td>The material’s diffuse-parameter</td>
</tr>
<tr>
<td>Specular</td>
<td>The material’s specular-parameter</td>
</tr>
<tr>
<td>Shininess</td>
<td>The material’s shininess</td>
</tr>
<tr>
<td>Normal</td>
<td>Surface normal</td>
</tr>
<tr>
<td>LDiffuse</td>
<td>The light’s diffuse-parameter</td>
</tr>
<tr>
<td>LSpecular</td>
<td>The light’s specular-parameter</td>
</tr>
<tr>
<td>LAmbient</td>
<td>The light’s ambient-parameter</td>
</tr>
</tbody>
</table>

2. The only input parameter that is necessary for the Phong node to work is Normal; the rest will work with their respective default values. Specify the vertex attribute NORMAL as input to the Normal-parameter, and then connect the output of the Phong node to the Output node, and then compile. Try rotating the object in the Preview Window by moving the mouse while pressing the left mouse button. Use constant values for the input parameters of the Phong node and experiment with different values, and try the effect on different objects in the Preview Window.

3. Try to use the result from a texture sample as input to the Diffuse-parameter of the Phong node instead of using a constant value, and try adding a layer of rust to the texture in the same way as in Exercise 2. We will now make the rust-covered areas less shiny than the “clean” areas of the texture. Add a texture sampling from the texture textures/rust_specular.png, and use the resulting color as input to the Specular parameter of the Phong node. This texture has dark colors in the same places as there is rust in the rust texture, and is white in the other areas. Note how the rusty areas now do not reflect as much light as the clean areas.

Exercise 4 – Normal Mapping

Create a Phong node and connect it to an Output node. Do a texture sample from one of the textures textures/???_color.jpg and connect it to the Diffuse parameter of the Phong node. Then create a node of the type Texturing/NormalMap. This node samples a color from a texture and converts the R, G and B components of the color to X, Y and Z components of a normal vector. Each texture file textures/???_color.jpg comes with corresponding texture file textures/???_normal.jpg that stores surface normals encoded as colors. Add a sampler for the surface normal texture textures/???_normal.jpg that corresponds to the color texture chosen earlier, and use it as input to the NormalMap node, and use the resulting sampled normal vector as input to the Normal parameter of the Phong node. The surface of the object will now appear to be much more detailed, and will reflect light according to the normals sampled from the surface normal texture instead of the vertex normals.

Exercise 5 (Optional, if time allows) – Implementing an own node

Note! Requires some knowledge of the shading language GLSL.

Make a copy of the file nodes/Color/Greyscale.xml and call it nodes/Colors/Saturation.xml. Then try to make changes in Saturation.xml so that it takes one input Color of the data type color, and a second input parameter Amount of the data type float, and an output parameter SatColor of with type color. Then try to alter the source code in the body-block so that the
node will take the input color, change its color balance and outputs the resulting color as SatColor. The color balance should be calculated so that Amount=0 leads to complete grayscale and Amount=1 leads to no alteration of the color, and a value 0-1 leads to a scale of color strength between these two extremes. You can get some help from looking how the node Colors/Mix (nodes/Colors/Mix.xml) is implemented. The next time the program is started the node Saturate will appear among the other nodes. Try using your new node in a shader graph and look at the generated source code to see how your source code is woven into the shader program.

Note! The error handling in the system is still very poor, so one will not get much more help other than the node just not working if there is any syntax errors in the node file.
Appendix C

Source Code

Phong Shader (fig 7.6, p. 58)

**Vertex Program**

```glsl
// Varying
varying vec3 v_NORMAL_e;

// Body
void main()
{
    // Object-space to Eye-space rotation
    vec3 NORMAL_e = gl_NormalMatrix * gl_Normal;
    v_NORMAL_e = NORMAL_e;
    gl_Position = ftransform();
}
```

**Fragment Program**

```glsl
// Varying
varying vec3 v_NORMAL_e;

// Constants
const vec3 c_Diffuse = vec3(0.702, 0.761, 0.984);
const vec3 c_Specular = vec3(0.180, 0.180, 0.180);
const vec3 c_Ambient = vec3(0.0, 0.0, 0.0);
const float c_Shininess = 40.0;

// Body
void main()
{
    vec3 NORMAL_e_n = normalize(v_NORMAL_e);

    // Phong Lighting Model
    vec3 LDirection = vec3(gl_LightSource[0].position);
    vec3 LHalfVector = vec3(gl_LightSource[0].halfVector);
    vec3 Color =
        (c_Ambient) + // * vec3(gl_LightSource[0].ambient)) +
        (c_Diffuse * vec3(gl_LightSource[0].diffuse) * max(dot(NORMAL_e_n, LDirection), 0.0)) +
        ((c_Specular * vec3(gl_LightSource[0].specular)) * pow(max(dot(NORMAL_e_n, LHalfVector), 0.0), c_Shininess));

    // Final color output
    gl_FragColor = vec4(Color, 1.0);
}
```
Phong + Normal Map + Texture Map Shader
(fig 7.10, p. 60)

Vertex Program

// Attributes
attribute vec3 a_TANGENT;

// Varying
varying vec2 v_TEXCOORD0;
varying vec3 v_NORMAL_e;
varying vec3 v_TANGENT_e;
varying vec3 v_binormal_e;

// Body
void main()
{
  vec3 binormal = cross(gl_Normal, a_TANGENT);

  // Object-space to Eye-space rotation
  vec3 NORMAL_e = gl_NormalMatrix * gl_Normal;

  // Object-space to Eye-space rotation
  vec3 TANGENT_e = gl_NormalMatrix * a_TANGENT;

  // Object-space to Eye-space rotation
  vec3 binormal_e = gl_NormalMatrix * binormal;

  v_TEXCOORD0 = vec2(gl_MultiTexCoord0);
  v_NORMAL_e = NORMAL_e;
  v_TANGENT_e = TANGENT_e;
  v_binormal_e = binormal_e;

  gl_Position = ftransform();
}

Fragment Program

// Parameters
uniform sampler2D p_MyTexture;
uniform sampler2D p_MyNormalMap;

// Varying
varying vec2 v_TEXCOORD0;
varying vec3 v_NORMAL_e;
varying vec3 v_TANGENT_e;
varying vec3 v_binormal_e;

// Constants
const vec3 c_Specular = vec3(0.424,0.424,0.424);
const vec3 c_Ambient = vec3(0.231,0.231,0.231);
const float c_Shininess = 40.0;

// Body
void main()
{
  vec3 binormal_e_n = normalize(v_binormal_e);
  vec3 TANGENT_e_n = normalize(v_TANGENT_e);
  vec3 NORMAL_e_n = normalize(v_NORMAL_e);

  // Tangent Space Normal Map
  vec3 Normal = normalize((vec3(texture2D(p_MyNormalMap, v_TEXCOORD0)) * 2.0) -
      vec3(1,1,1));
// Tangent-space to Eye-space rotation
vec3 Normal_e = (TANGENT_e_n * Normal.x) + (binormal_e_n * Normal.y) + (NORMAL_e_n * Normal.z);

// 2D Texture Look-up
vec4 TexSample = texture2D(p_MyTexture, v_TEXCOORD0);

// Phong Lighting Model
vec3 LDirection = vec3(gl_LightSource[0].position);
vec3 LHalfVector = vec3(gl_LightSource[0].halfVector);
vec3 Color = (c_Ambient) + // * vec3(gl_LightSource[0].ambient)) +
(TexSample.rgb * vec3(gl_LightSource[0].diffuse) * max(dot(Normal_e, LDirection), 0.0)) +
((c_Specular * vec3(gl_LightSource[0].specular)) * pow(max(dot(Normal_e, LHalfVector), 0.0), c_Shininess));

// Final color output
gl_FragColor = vec4(Color, 1.0);

Cube Environment Mapping Shader
(fig 7.11, p. 61)

Vertex Program

// Externals
uniform mat4 e_viewmtx;
uniform mat4 e_worldmtx;

// Varying
varying vec3 v_viewdir_w_n;
varying vec3 v_NORMAL_w;

void main()
{
    vec3 position = vec3(gl_ModelViewMatrix * gl_Vertex);
    vec3 viewdir = position;

    // Eye-space to World-space rotation
    vec3 viewdir_w = viewdir * mat3(e_viewmtx);
    vec3 viewdir_w_n = normalize(viewdir_w);

    // Object-space to World-space rotation
    vec3 NORMAL_w = mat3(e_worldmtx) * gl_Normal;
    v_viewdir_w_n = viewdir_w_n;
    v_NORMAL_w = NORMAL_w;
    gl_Position = ftransform();
}

Fragment Program

// Parameters
uniform samplerCube p_MyCubeMap;

// Varying
varying vec3 v_viewdir_w_n;
varying vec3 v_NORMAL_w;

// Constants
const vec3 c_Vector = vec3(0.0, 0.0, 0.0);

// Body
void main()
{
    vec3 NORMAL_w_n = normalize(v_NORMAL_w);
    vec3 viewdir_w_n_n = normalize(v_viewdir_w_n);

    // Reflection
    vec3 RefDir = reflect(viewdir_w_n_n, NORMAL_w_n);

    // Cube Map
    vec4 TexSample = textureCube(p_MyCubeMap, RefDir);

    // Final color output
    gl_FragColor = vec4(TexSample.rgb, 1.0);
}