An XML-based Visual Shading Language for Vertex and Fragment Shaders

Frank Goetz*
University of Paderborn
Germany

Ralf Borau†
University of Paderborn
Germany

Gitta Domik‡
University of Paderborn
Germany

Abstract

This paper presents a new system for the visual development of complex vertex and fragment shaders. The system makes usage of the advantages of visual programming languages. The core of the system is a Java program. With this program users can develop and test dataflow diagrams that describe the functionality of OpenGL ARB vertex and fragment programs. To get a graphical feedback the system is able to display rendered and shaded scenes immediately. The rendering of these three dimensional scenes will be done with the common Java-OpenGL binding GL4Java. Further on Cg4Java, a new binding for connecting Java to NVIDIA’s High-Level Shading Language C for graphics (Cg) was developed. For an easy and fast way of verification, and for the possibility of a future integration into XML based 3D formats like Extensible 3D (X3D), the whole topology of the dataflow diagrams will be stored in XML. The concept of shader programming with dataflow diagrams and the architecture of our system will be described in detail. For a better understanding two examples will be presented and explained.


Keywords: vertex shader, fragment shader, dataflow programming, visual programming, shading languages, Extensible 3D (X3D), Extensible Markup Language (XML)

1 Introduction

In the last years there have been interesting developments in the area of real time rendering of three dimensional graphics with graphics accelerator boards. The rendering quality and power of current graphics accelerator boards were significantly improved. The restricted graphics pipeline with fixed shading and illumination models was extended by programmable units (called vertex and fragment processor). These units make it possible to render more specialized graphics effects like e.g. per-pixel lighting, bump and environment mapping. Also complex illumination models or motion blurring can be implemented in vertex or fragment programs. Today these programs are being developed with so-called High-Level Shading Languages (HLSL). These HLSLs are usually comparable to procedural programming languages like C, extended by new vector and matrix data types. Compared to assembler code, such shading languages offer more comfort (e.g. with the support of functions) and have a better legibility.

The development of such programs with assembler is a very hard and laborious work. So the demand for programming languages that deliver the same functionality as high level languages rises extremely. These languages should deliver more comfort and a higher abstraction level. Previous developments in this area go into a similar direction. Shaders can be programmed with a procedural programming language. Afterwards a compiler generates a vertex or fragment program. As mentioned before these languages are called High-Level Shading Languages. Examples are Microsoft’s High-Level Shader Language [Microsoft 2003], NVIDIA’s C for graphics (Cg) [Mark et al. 2003] and the OpenGL Shading Language (GLslang) [Kessenich et al. 2003]. The development with HLSLs is not only positive. There are also some points that have to be criticised:

• At the moment there are no possibilities for debugging and output available. This makes the search for errors very hard and time-consuming.
• Even first programming practices are very hard without having enhanced programming skills.
• In spite of the fact of using high programming languages, detailed knowledge about the architecture is necessary.
• The abstraction level of HLSLs is very low; this means that the languages are hardware oriented. A better way would be if HLSLs would take care of the simulated process (e.g. seen easily by programming the illumination).
• Until now there are only a few integrations into object oriented visualization and graphics APIs.

Our goal was to develop a system for vertex and fragment programs that eliminate some of the weaknesses mentioned before.

Extensible 3D (X3D) is developed by the Web3D Consortium and should replace the Virtual Reality Modelling Format (VRML) in the near future. At the moment VRML is the standard for 3D graphics in the WWW. But X3D is distinctly more than a description language. The scene graph is not fixed anymore to one specific language. Rather it is specified as an object model. Besides the integrated node types of VRML97 a lot of new ones are supported. In X3D the scene data can be stored in the Extensible Markup Language (XML) or in the old VRML notation, thereby supporting completely the predecessor standard. An additional improvement is an Application Programming

1*e-mail: frank.goetz@uni-paderborn.de
1†e-mail: borau@uni-paderborn.de
1‡e-mail: domik@uni-paderborn.de
Flow Units

A parallel hardware rendering system was developed. It consists of 1995. Because of the high demand of computational power a earlier approach in this area is the PixelFlow system [Lastra et al. calculate interactive scenes in movie quality and in real time. An future which direction it will follow.

2 Related Work

Over many years three dimensional graphics systems only used fixed parametric functions to calculate shading and illumination models [Olano et al. 2002]. With Procedural shaders the user is able to produce individual high quality graphics effects. These are programs (shaders) that, for example, will be executed instead of a fixed illumination model to calculate a color value. By using these programs the user has the possibility to control the representation of their 3d models on their own. The programming language is used to develop these shader programs is called shading language.

An early example of a shading language is part of the RenderMan Interface developed by Pixar [Hanrahan and Lawson 1990]. The RenderMan Interface is a standard interface that can be used by modellers to convert their scene information to high quality rendering systems. It offers the possibility to insert self developed shaders at five different stages in the rendering pipeline [RenderMan 2000]. These shaders can be used to simulate new light sources (Light Source Shaders), reflection and refraction characteristics of a surface (Surface Shaders) or changes of the surface (Displacement Shaders). To write those shaders the system offers a C-similar programming language (RenderMan Interface Shading Language). The processing of color and vector data is thereby a substantial component. A large advantage is that for the addition of new graphics effects neither the interface nor the renderer has to be changed. Since these shaders are executed per-pixel, usually a software rendering systems needs minutes or hours, depending on the complexity of the scene, for the computation of only one picture. However, this kind of computation can be used to produce pictures and animations in photo-realistic quality. The company Pixar is particularly well-known for its computer-generated motion picture films (e.g. Toy Story or Finding Nemo), which were produced with their high quality render X3D. At the moment there are two proposals that differ not very much from each other [Paris and Couch 2003][Carvalho 2003]. Both are about to implement shader nodes that consist of a field that contains the source code of a shader. Both propose to support the OpenGL Shading Language, Microsoft’s High-Level Shader Language and NVIDIA’s Cg. The outcome of these proposals lies in the hands of the consortium and it will be seen in the near future which direction it will follow.

2.2 Usage of Vertex and Fragment Shaders

At the moment there are two predominant APIs for real time 3D graphics. On one hand there is Direct3D, part of the DirectX multimedia interface for the operating system Windows [Microsoft 2003]. On the other hand there is OpenGL, a low-level interface for graphics hardware that is available for all common operating systems [Segal and Akeley 1993].

Integrated in Direct3D is the so-called High Level Shading Language. This is a cooperation development of Microsoft and NVIDIA. This shading language is similar to Cg. The shader technology of DirectX is up-to-date. The current versions (3.0) of vertex and pixel shaders allow extensive programs and dynamical control structures. Starting with version 1.5 OpenGL supports vertex and fragment programs as standard ARB (Architecture Review Board) extensions. By finalizing the OpenGL Shading Language (GLSLang) [Kessenich et al. 2003], an HLSL is included in OpenGL, too.

2.3 C for graphics

In mid-2002 the graphics accelerator board manufacturer NVIDIA released the first version of the programming language C for graphics (Cg) [NVIDIA 2003] [Mark et al. 2003].

As the name Cg indicates, there are many affinities to the programming language C. Compared to other shading languages like RenderMan, Cg was not implemented as an application oriented programming language. Cg is more a general auxiliary equipment for the exploitation of hardware resources. An important property of Cg is its portability. The central tool is the Cg compiler. With the Cg Compiler vertex or fragment programs written in Cg can be compiled for a distinguished target platform. Target platforms can be ARB programs, Direct3D shaders or manufacturer specific OpenGL extensions.

2.4 Visual Programming

A very practical variant of visual programming systems are dataflow oriented systems. They are based on a dataflow model [Sharp 1992]. This dataflow model basically consists of two components: nodes that represent functional units or operators, and directed edges that represent dependencies between nodes regarding the data. Incoming edges represent input data necessary for the execution. In contrast to control flow oriented systems the execution order does not depend on the command counter, but depends on the individual nodes.

There are two possibilities for the execution:
1. Demand-Driven: the data that is necessary for execution is requested from nodes which are connected through incoming edges.
2. Data-Driven: an execution will be started only if all data is available.

2.1 Programmable Graphics Hardware

Because of the immense calculation effort, the wish for programmable graphics hardware is natural. A long term goal is to calculate interactive scenes in movie quality and in real time. An earlier approach in this area is the PixelFlow system [Lastra et al. 1995]. Because of the high demand of computational power a parallel hardware rendering system was developed. It consists of independent rendering units (Flow Units) that contain a geometry unit and a rasterizer unit [Eyles et al. 1997]. Each unit is able to generate pictures. These will be connected via two networks. By using the geometry network, the geometry units are able to communicate with each other. For the calculation of a picture, the geometry information of a scene will be segmented and distributed. Each Flow Unit generates its own picture based on this data. As a result these pictures will be combined via an image composition network to one picture. The available shading language is called pfman and has a close resemblance to the RenderMan Shading Language and is for this reason very similar to it [Olano and Lastra 1998].
2.5 Visual Programming and Shading

Shade Trees [Cook 1984] have been a concept already known since 1984. In this concept shaders are visualized and developed as big tree structures. Nodes inside a tree represent easy and complex operations like vector and matrix operations. Operands are the results of the corresponding branches. Instead of fixed reflection and shading models this concept has the benefit of a higher flexibility. In the year 1990 Abram and Whitted presented an implementation of Shade Trees [Abram and Whitted 1990]. This is a shader development system that consists of two parts. The first high level part consists of blocks delivered by the system with constant functionality. To program a shader different blocks can be connected to each other. This network-like environment is comparable to the dataflow model. This part of the system can easily be used by a user without any knowledge of programming languages. The second part of the system is a common programming language for the development of new blocks that are first compiled and afterwards can be integrated in the system.

3 Application

In this chapter architecture and basic concepts of our visual and dataflow oriented programming system will be presented. The main program was developed in Java (Figure 1). The graphical user interface is programmed with the Swing API. With different toolbars the user can select nodes of different types to integrate them in the current dataflow diagram. These toolbars can be selectively shown on the desktop. If more than one vertex or fragment program is needed, then a fast selection is made possible by choosing the corresponding tabulator in the diagram editor. Already existing diagrams can also be loaded or saved using import or export functions. The connection of individual input and output variables of particular nodes can be managed by consecutively selecting the nodes that have to be linked. The positioning of nodes and links is executed semi-automatically and requires in principal no explicit postprocessing. The construction of dataflow diagrams, node types, data types, XML structure of the diagrams, the transformation of the data flow diagram to a Cg program and the visual representation of the shader program will be described in the next paragraphs of this chapter.

3.1 Dataflow Program

The visual representation of a dataflow program is a dataflow diagram. The nodes of a dataflow program consist of input and output variables. Directed edges connect exactly one output variable with one input variable. The edges are called data channels. During the program execution data passes through these edges. Nodes without input variables are sources. These nodes can have a constant value or can be value generators. Nodes without output variables are targets. Nodes themselves can again be dataflow programs. Their data sources build the input variables of the node and according to this the data targets build the output variables. Each output variable can be connected with different input variables, but an input variable can only be connected with one edge. All input variables have to be connected, except the visual programming system delivers default values. Cycles are not allowed.

A few advantages of a dataflow model are [Schiffer 1998]:
• well ordered illustration of the real world
• easy to learn, e.g. no variables and dynamic data structures have to be used
• top-down splitting is possible
• visualization of the program execution is possible in an easy and fast way

3.1.1 Node Types

A vertex or fragment program is a dataflow program that consists of nodes and edges. While nodes are functional units with input and output variables, edges represent data that flows from output variables to input variables.

The following node types are available:
• **Operator nodes** generate values for exactly one output variable. These correspond to the operators that are available in Cg. On one hand, there are mathematical operators, like addition and multiplication, for vector or elementary data types. On the other hand, swizzle operators (a term used in Cg [NVIDIA 2003]) can be used to move, change or leave out vector components.
• **Function nodes** can have a lot of input parameters (normally only a few), but only one output variable. They are light weighted functions that belong to the language range of Cg, as part of the Cg standard library. These are divided in the categories mathematical, geometric and texture.
• **Constant nodes** have no input variables and one output variable that delivers a constant value.
• **Diagram nodes** deliver the possibility to encapsulate distinguished parts of the program into a node. Internally they are constructed as dataflow diagrams in the same way as the main program and will also be edited visually. Moreover they can be saved as files and be reused as parts of other shaders. A collection of frequently required functionality that were realised in diagram nodes enhances and accelerates the development of new shaders. Vertex and fragment programs are particularly also diagrams but can certainly not be integrated as nodes in other diagrams.
• **OpenGL state nodes** are special nodes that give access to vertex attributes and OpenGL program variables. These nodes are visually represented as constant nodes. However vertex attributes will be updated for every vertex.
• **Constructor nodes** are used to combine different vector components to build a new vector.
3.2 Vertex and Fragment Program Generation

The visual program (vertex or fragment program) will be transformed in different languages and conditions before its execution. The different steps are (Figure 3):

1. In the first step the topology of the program will be exported to an XML file or a DOM tree.
2. By using an Extensive Stylesheet Language Transformation (XSLT) processor a transformation of the program into the Cg programming language will be performed.
3. The generation of a runnable program and the loading of a corresponding processor can be done with resources of the Cg runtime.

3.2.1 Control Flow

Control structures like if and while play, at the current state of the shader development with vertex and fragment programs, only a minor role. Despite the fact they are supported in Cg our momentarily decision was not to transmit elements of this categories. Currently they can only be supported by an ARB program if they will be unrolled by the Cg compiler. Only the profiles of NVIDIA extensions allow dynamic branches and loops with OpenGL at runtime. The support of dynamical control structures is planed for future versions.

3.2.2 XML Data Structure

For a possible integration of visual shading languages into X3D XML was choosen as a basic data structure for storing shaders. During the storage all topology information, specifically the structure of the dataflow program and the position of the node elements will be transferred to an XML file.

The conversion of a shader, stored in XML, into Cg will be done with an Extensive Stylesheet Language (XSL) transformation. XSL allows the transformation of XML documents in arbitrary text formats. Therefore an XSLT processor is necessary. This procedure has two fundamental advantages:

1. Without any big effort the transformation can be made with an X3D viewer or a graphics system, independently from the underlying operating system and programming language that was used to realize such a system. Only an XSLT processor implementation that is available for all common programming languages is necessary.

2. A possible change to a new target platform (Cg) is very easy. By exchanging the XSL instructions other HLSLs (e.g. OpenGL Shader Language (GLslang) or Microsoft High-Level Shader Language) can be supported at a later point, because the topology of the dataflow diagram contains an insignificant amount of Cg dependent elements.

3.2.3 XML Export and Transformation

In this paragraph the transition from the visual representation to the execution of the program will be described. The first step is the export to XML. The topology, the connection of the input and output variables of the different nodes, has to be saved. In the next step (the processing with XSL) certain requirements that have to be fulfilled will be provided. The idea is to save the directed graph as a list of nodes and edges. In a straightforward way this strategy does not work because of the inadequacy of the transformation language XSL.

Nodes will be exported as pre ordered tags to XML. The data dependencies, respectively the incoming edges will be stored as childs of the node element. This enables the XSLT processor to pass the whole graph. Source nodes are marked clearly with their Name and ID. Details of the XML data structure will be explained later.

![Figure 3. Transformation overview.](image-url)
3.2.4 Cg Compilation

If the XSLT processor has produced a correct transformation of the XML file a runnable vertex or fragment program in Cg source code will exist. By using the Cg compiler (Cgc) this program can be compiled to an executable program. The Cg runtime takes care of the loading into the graphics processor and the activation.

3.3 Realization of a preview in Java

For a fast development a test environment inside the development toolkit is necessary. There has to be the possibility to construct and render a three dimensional OpenGL scene inside the Java application. Furthermore the functionality for a direct binding of the vertex and fragment programs has to be integrated in our system. To solve these problems advantage of GL4Java and a new self developed interface called Cg4Java (Figure 4) was taken.

3.3.1 GL4Java

The OpenGL context and the three dimensional test scenes have to be generated. Therefore the GL4Java package was used. GL4Java brings the whole OpenGL functionality of the OpenGL API v1.3 to Java. This is done by the usage of the Java Native Interface (JNI). Via JNI the Java functionality will be bound to C++ Libraries. JauSoft provides C++ libraries that establish a binding to the native OpenGL libraries for different platforms like Microsoft Windows, Apple Macintosh, Linux and different UNIX derivates.

3.3.2 Cg4Java

The first version of C for graphics (Cg) was released in July 2002 by NVIDIA. At this time Cg was the only High-Level Shading Language for developing vertex and fragment programs that can be used with OpenGL. Furthermore Cg delivers a maximum of flexibility to support for example Microsoft’s Direct3D shaders in a later version of our system.

To compile and activate Cg programs needed for the rendering of a three dimensional scene, an interface comparable to GL4Java by using the Java Native Interface (JNI) was developed. This new interface was named Cg4Java and makes all required function calls of the Cg runtime library available to our Java application. The Cg runtime library mainly consists of three parts. The core contains functions for loading, compiling and managing of Cg programs. The other two parts contain API specified functions for the activation of programs under Direct3D or OpenGL and to transfer the required parameters. These parts are oriented at the programming paradigms of the two graphics APIs. The Java interface of the Cg runtime library allows calling nearly every function of the core and the OpenGL part via the JNI. With Cg4Java a Cg context can be generated, Cg programs can be compiled and loaded in such a Cg context. Finally they can be loaded in an appropriated processor and afterwards activated.

3.4 Advantages of the Visual Programming System

For ordinary (often required) applications the user should have the possibility to choose between different types of templates. These templates are stencils with an integrated basic structure. An example could be e.g. a vertex template that already transforms the vertex position and normal into camera coordinates since the calculation of the illumination normally is done in the camera coordinate system. Furthermore, the calculation of the color value could be done previously and then only the illumination model has to be chosen. This could have been integrated into the structure as an empty diagram node. Other effects like per-pixel lighting need a special kind of calculation and a broadcasting of the data from the vertex to the fragment program. Also this could have been pre implemented in a per-pixel template.

Further unique advantages for the user are:
- Because of the control of the editor and the visual assistance the generation of wrong syntax is not possible, this guarantees a high reliability.
- Default values and templates make it possible that only correct (compilable) programs will be generated.
- Absence of variables that produce any kind of errors.

3.5 Relationship to X3D

The usage of this shader development system within a complex 3D graphics API is an important goal. X3D as a modern and open system seemed to be a good decision for this. Our aim is not to develop a new X3D standard. We want only to deliver new ideas and concepts for a future integration of vertex and fragment programs into X3D. Therefore we implemented an XML-based extension and different demo shaders as proof of concept. For the integration one should be able to define new proprietary X3D node types.

The data format that was developed for the visual system should be seen as an alternative approach for integrating shaders in an XML-based 3D data format. The proposals that were submitted to the X3D Programmable Shaders Working Group want to integrate shaders through storing the source string of a shader program written in HLSL as an attribute of an XML element. In contrast to those proposals our format is able to express the shader itself in XML.
4 Examples

In this paragraph first a complex shader program will be explained in detail. Afterwards we will shortly take a look at the different textual code representation of an additional shader example. With these two examples we want to explain how our dataflow diagrams work and how vertex and fragment programs will be described and represented in our system.

As a detailed example the Fresnel effect (Figure 5) was chosen. For the realtime simulation of the Fresnel effect two programs (a vertex and fragment program) with different diagram nodes are needed. First the vertex program is described and then a deeper look into the fragment program will be taken. This example is based on the chromatic dispersion shader from [Fernando and Kilgard 2003]. As a second example a shader that simulates a simple diffuse illumination is presented.

4.1 A Detailed Explanation of the Fresnel RGB Shader

The Fresnel formula is an approximation of the so called Fresnel effect. It describes the fact that at a shallow angle of view (angle between surface and viewing direction) relatively more light is reflected than at an obtuse angle. This means if you look straight onto a water surface you can clearly look through the water. In contrast to this, if you look under a shallow angle onto a water surface then you will see primarily the reflecting background or sky.

4.1.1 Vertex Program of the Fresnel RGB Shader

Figure 6 shows the main structure of the vertex shader. On the left side you see the connectors that represent data coming from the application. The first five are standard vertex attributes. They are automatically bound by OpenGL. As additional binding there are two three-component vectors called fresnelBSP and etaRatio. fresnelBSP consists of three variables that belong to the Fresnel equation. etaRatio consists of three different refraction quotients for the colors red, green and blue taking into account the different refraction behaviour for light at different wavelengths. For testing purposes these bindings are stored with default values. These can be used within our test environment. The connectors on the right side represent the data that will be transferred to the fragment program. In general the data flows from the left side to the right.

Figure 6. fresnelRGB_vertex dataflow diagram.
The vertex program consists of five diagram nodes (toEyeCoord, projection, calculate_I_N, fresnelReflect, rgbRefraction), one function node and five swizzle operator nodes.

The diagram node toEyeCoord transforms the vertex position and vertex normal into eye coordinates. Figure 7 gives a detailed look inside this node. As you can see, internal diagrams are in general very similar to diagrams of the main program. Both position and normal are multiplied by an OpenGL state matrix. The multiplication nodes get the matrix data from OpenGL state nodes. They deliver the ModelView matrix for multiplication with the position and the inverted transposed ModelView for the normal. The output of the multiplication nodes are connected with the export anchors.

To project the transformed position vector onto the screen the output $P$ of the node toEyeCoord is connected with the diagram node projection. It multiplies its input value by the projection matrix and the output value will be written to the output position register.

The diagram node calculate_I_N calculates the lookAt vector $I$ (this one points from the vertex to the camera) and the normalized normal vector $N$. Important here is the transformation of a four-component into a three-component vector. $I$ and $N$ are input values for three different nodes. The first one of these three nodes is the reflect function node. This reflects $I$ at the surface (with normal $N$) and delivers the result via a swizzle operator node in the texture coordinate register 0. This time the swizzle operators only enlarge the three-component vectors to four-component vectors. This is done by copying the z-component of a vector to the w-component (homogenous component).

In the fresnelReflect diagram node the Fresnel equation will be calculated. The equation calculates the amount of light that will be reflected by the surface (the rest will be refracted). The Fresnel equation (approximation) for the reflection looks like this:

$$\text{bias} + \text{scale} \times (1 + \sqrt{N}) \times \text{power}$$

The components bias ($B$), scale ($S$) and power ($P$) will be stored in the $x$-, $y$- and $z$-component of the input vector fresnelBSP. The formular will be executed in the diagram node fresnelReflection. The reflection value will be written into the color register (all four vector components get the same value).

The diagram node rgbRefraction calculates for every color component a different refraction vector. The equation quotient consists of the components of the etaRatio vector. etaRatio will be divided by some swizzle operator nodes and the individual components will be send together with $I$ and $N$ to the function node refract that calculates the light refraction. The results will be sent to the output variables of TRed, TGreeen and TBlue. Finally the result vectors will be written into the texture coordination registers one, two and three of the vertex program.
4.1.2 Fragment Program of the Fresnel RGB Shader

Input values to the fragment program (Figure 8) are the interpolated values of color and texture coordinates.

A short reminder: The color register contains the reflection portion; the texture coordinates contain the vectors for the reflection, refraction (red), refraction (blue) and refraction (green).

The fragment program needs two components for the calculation of the pixel color: the reflected and the refracted light. Like in normal environment mapping the reflected light will be looked up in the cube map. To do this the texture coordinate 0 \( \text{IN.texture0} \) will be reduced to a three-component vector by using a swizzle operator node. This vector will be submitted to the function node \( \text{texCube} \). The second input value for this node is the \( \text{cubeMap} \) variable (a reference to the cube map). The color of the refracted light portion will be calculated with the diagram node \( \text{texCubeRGB} \). Its input values the three refracted light rays and the \( \text{cubeMap} \) variable.

A look inside the diagram node can be seen in Figure 9. For all three colors the cube map will be accessed, but only one color component will be transported via a swizzle operator node. The different color components will be composed to a new color via a constructor node and the output value will be extended to a four-component vector.

Finally the two light portions have to be combined in the fragment program. This will be done with the function node \( \text{lerp} \) that calculates \( r \times \text{color}_{\text{reflect}} + (1-r) \times \text{color}_{\text{refract}} \).

Figure 5 shows the high quality output this shader produces.

4.2 Textual Code Representations of a Diffuse Shader

In the previous section the dataflow diagrams of a complex shader were described. Next an overview of the appearances of the different textual files and their code structures that will be generated by our program will be shown. These steps have to be processed until a three dimensional scene can be shown on the screen. To make this paragraph as comprehensive as possible only a very short program that calculates a diffuse illumination is used.

The result of this shader and the main dataflow program can be seen in Figure 10 and Figure 11, respectively.

First a short description of the assembler program is given. The words TEMP, ATTRIB, PARAM define different variables (Figure 12: lines 10-16). The binding is done with ATTRIB for the vertex attributes and with PARAM for other program...
parameters like the OpenGL matrixes. First a multiplication of the vertex position and the Modelview matrix is executed (Figure 12: lines 17-20) to convert the vertex position into camera coordinates. The ADD command calculates the vector from the vertex position to the light source (Figure 12: line 21). After that the vector will be normalized (Figure 12: lines 22-24) and the vertex normal will be transformed into camera coordinates (Figure 12: lines 25-27). The scalar product of the L vector and the normal delivers the diffuse value (Figure 12: line 28). The diffuse value will be written in the output register of the primary color (Figure 12: line 29). The last rows (Figure 12: lines 30-33) of the program are necessary for the projection of the vertex onto the screen.

Next the XML description (Figure 13) of the before presented assembler program (Figure 12) will be explained. Root elements of a XML shader document can be either VertexProgram or FragmentProgram. The input and output structures describe the input and output variables of the main program. The description is stored externally in an XML file.

Node tags describe the nodes that are inside the dataflow diagram. The attribute type corresponds to the different node types like diagram, function, operator, constructor, constant or state. Attributes name and id together deliver an unambiguous key inside an internal diagram or the main program. The attribute filename is optional and shows the storage location of the diagram. If no filename is provided then the diagram has to be defined as child of the Node element. Optional attributes like x and y indicate the component position in the visualization of the dataflow diagram. Data elements describe the incoming data connections and attribute names that indicate the input variable.

Connection elements contain the concrete connection information. With the attributes source and source_id the linked node will be identified. The attribute output identifies the output variable of the node. Export and Import elements are similar to connections, but they only link the import and export interfaces of the actual diagram. They will be handled separately because if they are used as input variable and output variable they do not have a node they are connected to. They are necessary because source and target nodes do not exist at all.

Of course such programs in XML look very complicated (e.g. more complicated than a Cg source code), but if they are visualized as a dataflow diagram they become very clear and comprehensible. Compared to the assembler program the size of the Cg program (Figure 14) is very big. The reason for this is mainly that for every output parameter a variable has to be declared. The number at the end of a variable name is an unambiguous id, so there are no conflicts with the variable names if an identical function name will be used for more than one time.
<?xml version="1.0" encoding="UTF-8"?>
<VertexProgram name="simple_diffuse">
  <InputStruct name="IN" type="v_in_ARB_small"/>
  <OutputStruct name="OUT" type="v_out_ARB_small"/>
  <Export name="OUT.position">
    <Connection source="projection" sourceid="00" output="p" x="0.2990654"/>
  </Export>
  <Export name="OUT.color0">
    <Connection source="simple_diffuse" sourceid="00" output="diffuse_color" x="0.5"/>
  </Export>
  <Node type="diagram" name="projection" id="00" filename="projection.xcp" x="504" y="35">
    <Data name="position">
      <Connection source="toEyeCoord" sourceid="00" output="p" x="0.5"/>
    </Data>
  </Node>
  <Node type="diagram" name="simple_diffuse" id="00" filename="simple_diffuse.xcp" x="488" y="214">
    <Data name="position">
      <Connection source="toEyeCoord" sourceid="00" output="p" x="0.43650794"/>
    </Data>
    <Data name="normal">
      <Connection source="toEyeCoord" sourceid="00" output="n" x="0.3483871"/>
    </Data>
    <Data name="light_position">
      <Connection source="light_0_osition" sourceid="00" output="position" x="0.5"/>
    </Data>
  </Node>
  <Node type="diagram" name="toEyeCoord" id="00" filename="toEyeCoord.xcp" x="170" y="102">
    <Data name="position">
      <Import name="IN.position" x="0.70869565"/>
    </Data>
    <Data name="normal">
      <Import name="IN.normal" x="0.5"/>
    </Data>
  </Node>
  <Node type="state" name="light_0_osition" value="glstate.light[0].position" id="00" x="236" y="326"/>
</VertexProgram>

struct v_in_ARB_small {
  float4 position : POSITION;
  float4 normal : NORMAL;
  float4 color : COLOR0;
  float4 texture0 : TEXCOORD0;
  float4 texture1 : TEXCOORD1;
};

struct v_out_ARB_small {
  float4 position : POSITION;
  float4 color0 : COLOR0;
  float4 texture0 : TEXCOORD0;
  float4 texture1 : TEXCOORD1;
};

void projectionN40001700( out float4 p, in float4 position) {
  float4 mulN40003400;
  mulN40003400 = mul(glstate.matrix.projection,position);
  p = mulN40003400;
}

void simple_diffuseN40004F00( out float4 diffuse_color, in float4 position, in float4 normal, in float4 light_position) {
  float4 opN4000D800;
  opN4000D800 = light_position-position;
  float3 opN4000C749;
  opN4000C749 = opN4000D800.xyz;
  float3 opN4000B800;
  opN4000B800 = normal.xyz;
  float3 normalizeN4000A900;
  normalizeN4000A900 = normalize(opN4000C749);
  float dotN40009300;
  dotN40009300 = dot(normalizeN4000A900,opN4000B800);
  float4 opN40008200;
  opN40008200 = dotN40009300.xxxx;
  diffuse_color = opN40008200;
}

void toEyeCoordN4000EC00( out float4 p, out float4 n, in float4 position, in float4 normal) {
  float4 mulN40012E46;
  mulN40012E46 = mul(glstate.matrix.modelview[0],position);
  float4 mulN40011A00;
  mulN40011A00 = mul(glstate.matrix.invtrans.modelview[0],normal);
  p = mulN40012E46;
  n = mulN40011A00;
}

v_out_ARB_small main(v_in_ARB_small IN) {
  v_out_ARB_small OUT;
  float4 pN4000EC00;
  float4 nN4000EC00;
  toEyeCoordN4000EC00(pN4000EC00, nN4000EC00, IN.position, IN.normal);
  float4 diffuse_colorN40004F00;
  simple_diffuseN40004F00(diffuse_colorN40004F00, pN4000EC00, nN4000EC00, glstate.light[0].position);
  float4 pN40001700;
  projectionN40001700(pN40001700, pN4000EC00);
  OUT.position = pN40001700;
  OUT.color0 = diffuse_colorN40004F00;
  return OUT;
}

Figure 13: XML representation of the simple_diffuse shader.

Figure 14: Cg program of the simple_diffuse shader.
5 Conclusion and Outlook

The system that was built to develop shaders can be characterized in the following way:

- ease of use to develop shaders
- minimum of programming errors
- makes complete usage of the hardware
- one system for different operating systems
- support of other HLSLS is possible, not only Cg
- possibility of an integration into X3D

The system described here allows users with only basic knowledge in the area of vertex and fragment shader programming to achieve good results. In contrast to the low level programming with assembler code and also in contrast to the programming with HLSLs this can be done in an easy and fast way. Usually it is not a very easy and often time-consuming to integrate HLSL functionality into an existing operating system and into a development environment. This effort was reduced by combining all necessary functionality in one system.

Possible programming errors were dramatically reduced through the aspect of using a visual programming language. It is nearly impossible to generate a program that is not compilable. By investing more work in this part of our visual programming system it would be possible to generate only working programs.

Particularly vertex and fragment programs are often utilized in an unusual way. Texture registers are used for the data transport to the fragment processor (this means normalized values have to be adapted in a corresponding way to map them into a range of values between zero to one). Also textures are used as lookup tables to get a faster and direct data access to the memory of the graphics accelerator board. A practical utilization of such effects is not possible at a high abstraction level. Therefore we oriented ourselves according to the paradigm of Cg. This paradigm of Cg is: completely make usage of the hardware and do not worry about the requirements of the applications. A small help in this direction would be the categorization of the diagram nodes that are made available by the system. They should be ordered in groups of the same functionality (e.g. illumination or surface material). The categorization would additionally enable a faster way of finding required diagrams.

While using NVIDIA's Cg in combination with Java, Swing, GL4Java and Cg4Java our system is platform independent. Another consideration was to include only a minimum of HLSL specific elements in the XML description of our shader. So a later support of Microsoft's High-Level Shader Language or the OpenGL Shading Language can be achieved with little effort. Also the supported language elements will be imported from an XML file. For a support of a new HLSL this XML file has to be adapted in the right way. Additionally the transformation has to be adapted with XSLT.

At the moment a correct integration into the X3D format can not be delivered, because of the lack of extension possibilities in X3D. In our case additionally node types (momentary not allowed) that deliver support for our shader integration have to be bound into an appropriate (user defined) X3D profile. This node types can only be integrated by changing the standard of the X3D format. A possible solution for this is described in “Advanced Extension Mechanisms for X3D to Define, Implement and Integrate New First-Class Nodes, Components, and Profiles” [Rukzio 2003].

References

  http://www.cryogenographics.com/shading/.
  ShaderSpecV1.051.pdf
  Real-Time Shading. A K Peter, Natick, Massachusetts.
- RUKZIO, E., 2003. A generic extension mechanism for x3d to define, implement and integrate new first-class nodes, components, and profiles.
  http://www-mmt.inf.tu-dresden.de/web3d2003/Position
  Tech. rep., Silicon Graphics Computer Systems, Mountain View, CA, USA.