CgFX in Python
Programmable Shaders in a Scripting Language

Jonas Jonsson

Examensarbete för 20 p, Institutionen för datavetenskap, Naturvetenskapliga fakulteten, Lunds universitet

Thesis for a diploma in computer science, 20 credit points, Department of Computer Science, Faculty of Science, Lund University

CgFX in Python

Programmable Shaders in a Scripting Language

Abstract

Real-time graphics is one of the fastest growing areas in computer graphics today, where modern graphics hardware can produce almost cinematic rendering. Shaders provide a new level of control over the rendering pipeline.

This master's thesis is about combining real-time graphics programmability with scripting languages. This is accomplished by using Nvidia's effect format CgFX and the programming language Python. This approach enables high level scene descriptions and geometric abstractions. CgFX is a file format encapsulating rendering conditions to achieve a certain visual appearance.

Python is an interpreted, high-level, object-oriented, open-source programming language, used for scripting and gluing existing components together. The dynamic bindings and semantics of the language significantly simplify connecting the parts of a graphics renderer.

This report describes the result; a Python effect class and the way it can be used in a Python renderer. This thesis has also involved a thorough study of programmable real-time computer graphics, therefore a considerable part of the report is a result of this study.

CgFX i Python

Shaderprogrammering i skriptspråk

Sammanfattning

Realtidsgrafik är idag ett av de snabbast växande områdena inom datorgrafik. Moderna grafikkort kan uppnå renderingsresultat som är jämförbara med specialeffekter i filmbranschen.

Detta examensarbete behandlar kombinationen programmerbar realtidsgrafik och skriptspråk. För detta ändamål har Nvidias effektformat CgFX använts tillsammans med programmeringsspråket Python, vilket ger möjlighet att beskriva scener och geometri på en hög abstraktionsnivå. CgFX är ett filformat som beskriver effekter för att uppnå olika visuella resultat i en scen.

Python är ett tolkat, objektorienterat högnivåspråk som finns att tillgå som öppen källkod. Språket används ofta som ett skriptspråk och för att sammanfoga komponenter i större projekt. Dynamiken i språket gör att det med fördel kan användas för att underlätta sammanfogning av ett renderingssystem.

Den här rapporten beskriver resultatet, en Pythonklass som fungerar som ett verktyg för att rendera effekter i Python. Examensarbetet är även en studie av programmerbar realtidsgrafik och en betydande del av rapporten är ett resultat av den studien.

Table of Contents

1. Introduction

Real-time computer graphics has come a long way in recent years, and is now standing on the threshold of a new age. The hardware in our everyday computers has not just become faster, but even capable of bringing cinematic effects within reach. The gap is still large, but with the rate at which computer graphics are developing it will most certainly be decreased in the foreseeable future.

This thesis is about real-time shaders, effects programming, and scripting languages. Scripting languages in real-time computer graphics may sound unfamiliar to many, but this is about to change. Modern applications displaying advanced computer graphics are often written in low-level languages, as execution speed is the main priority. As the complexity of the graphics and the programs increases, development time becomes an important issue. This fact has forced many developers to integrate higher-level languages into their applications. Development speed is traded off against execution speed.

1.1 Background

Real-time graphics is a commonly studied area in computer science. However, the concepts of shaders are relatively unfamiliar to many, so this report tries to give a fairly extensive overview. Shaders provide full control over real time rendering, and they can be programmed with languages that resemble the high-level languages we are used to when programming regular applications. Cg, C for graphics, is a new high-level shading language, developed by Nvidia. The language is platform and API independent. It is used by a large number of companies working with real-time graphics. Cg also provides a way of encapsulating Cg shaders with other rendering conditions to achieve a certain appearance. This format is called CgFX which stands for Cg Effect.

Python is a high-level, object-oriented scripting language, offering a way to glue components written in other languages. This makes python a well suited part of computer graphics programming. It provides a high level control over large projects such as a renderer. Python is a scripting language, this makes it slightly slower compared to low-level languages, but since much of the rendering is done in hardware, the performance is acceptable. The Python language is easy to learn, and success in working with it is independent of any prior experience with other programming languages. Combining Python with CgFX would be a powerful and simple way to program real-time graphics.

1.2 Objectives

The main goal of this thesis is to examine the possibility of applying advanced effects to objects in a scene which is set up in Python. The Effects will be handled and accessed though high level abstraction called scene descriptions. For this purpose Nvidia's CgFX format is used, since it provides an efficient encapsulation for most parameters needed to render an effect. The format is hardware and platform independent, which means it can be used together with both leading real-time API's; OpenGL and DirectX.

Since Python is an interpreted language and the chosen effect format, CgFX, can be compiled at runtime, another goal is to make the applied effect editable at runtime. This would eliminate the edit-compile-run-quit cycle and therefore make effect editing much easier. The purpose of this report is to describe the way the implementation is done and how it is used. The description does not cover every detail, but enough to be able to use the effect interface to render simple effects. In addition to these goals, the report is also intended to give an overview to what shaders are and what can be done with them. The theory behind a number of shader programs is discussed briefly, both on vertex and pixel levels.

1.3 Constraints

Shader programming is a very large area in computer graphics. Applying an arbitrary shader to a program requires a lot of experience and work. This is because when writing a shader for a program, the program has to be modified and adapted to the shader. One of the main goals of this thesis was to avoid this, because it makes shader programming in Python extremely difficult. This is why CgFX is chosen, since it supplies an application-independent format, and even other features as different hardware profiles. This makes the effect format very portable. So instead of focusing on shader programming, this thesis is more focused on applying effects. When choosing CgFX, some restrictions follow. CgFX is designed to be applicable on objects, giving each object a particular appearance. If an effect should be able to interact with the surrounding geometry or with user input, the effect must be programmed for this, therefore making it application dependent. This thesis will mainly focus on shaders that are application-independent. Another constraint is the fact that CgFX is still in beta stage, this means that the existing documentation is very sparse. The official documentation only mentions the way CgFX is intended to be used, but does not explain anything in detail. This fact has made CgFX programming very hard, and converting existing Cg shaders to CgFX, complicated.

1.4 The Outline of This Report

After this chapter follows an introduction to shaders, explaining concepts, usage and what shaders really are. Short examples are given to illustrate what a shading language could look like and the differences between shader programming and regular programming. Then, a chapter is dedicated to effects and the CgFX format, describing how it can be used, what types of applications that can take advantage of CgFX, and a brief review of the syntax. Furthermore, there is a short section describing how effects are compiled and a word about the runtime system. The Cg language is also discussed and its resemblance to other high level languages. The following chapter briefly describes the Python programming language and some of the Python libraries. The next three sections describe the Python effect interface. A top-down approach is used, starting at a high level describing eight different effect types and scene descriptions. Subsequently the usage of the Python effect interface is explained, and finally the way the effect interface provided by Nvidia is wrapped to be used with Python is described. Thereafter follows discussion of the results and conclusions. Here, different approaches are considered, including future extensions and what could have been done if Nvidia's effect interface had not been in beta stage. The last chapter is a short summary.

2. Introduction to Shaders

2.1 Concepts

The word *shader* is somewhat misleading, because it applies on two rather different concepts: pixel shaders and vertex shaders. While a pixel shader is a fairly accurate name for something that colors a pixel, a vertex shader should rather be called a vertex processor or a vertex program. In this report, the word “shader” will be used more as a common name for shaders as a concept, but when referring to an actual program or code, the word “program” will be used.

2.2 What is a Shader?

A shader is a common name for an algorithm as a part of a rendering pipeline. When the algorithm affects vertices, it is called a vertex shader, and when it affects pixels, it is called a pixel shader. The algorithm controls the way each vertex or pixel is treated down the rendering pipeline.

A typical task for a vertex program is to perform transformations, lighting calculations, manipulating texture coordinates or altering normals. The data produced by a vertex program can be passed to another vertex shader program, or to a pixel program.

The pixel program, or fragment program computes the colour of each fragment in the rasterized data from the vertex program (fragment is a more accurate nomenclature than pixel since a pixel refers to the actual pixel on a screen, while the term fragment is the state required potentially to update a particular pixel.[1]). This means that when a vertex program is finished, it passes transformed data through a rasterizer which interpolates the positions and colours from the vertex program, and performs additional computations on every fragment.

2.3 Historical Background

When it comes down to the very core of computer graphics, things are very simple, it is just a matter of putting the right color at the right point on the screen. The things to be displayed are described geometrically, which to some extent is a simplification of the real world. The geometry consists of vertices, and each vertex has to undergo a number of steps in a rendering pipeline before it is put to screen. This pipeline includes such things as transformations, lightning calculations, and clipping. The steps in the pipeline can vary enormously in complexity, from simple wire frame rendering to advanced effects like skin and hair.

In a way, graphics programmers have always been able to do whatever they wanted, but detail and effects come at the cost of speed. Therefore, the most advanced graphics have been dedicated to the movie industry, where images can be produced long before they are used. This is called off-line rendering, and a single frame can take hours or even days to complete, depending on the complexity of the scene and

the power of the computer. Several images are then put together and displayed as an animation. Because time is not of the essence, only the software algorithms and the creativity of the artist restrict the visual appearance of an image or a frame.

Processing vertices in real-time is a very time consuming activity for any regular CPU. Real-time rendering was until the mid 1990's only possible using supercomputers. A typical example is a traffic flight simulator. In the mid 1990's things changed dramatically when the first 3D accelerators appeared. These so-called accelerators are quite simply graphics cards with a built in processor. This processor is called a GPU (Graphics Processing Unit) and is designed for doing one thing: calculating the steps in a rendering pipeline.

Something that is implemented in hardware cannot be changed, so the pipeline was fixed, which meant in giving up control over the rendering in exchange for speed. This has been the way real-time computer graphics has looked since the mid 1990's. The only real improvement has mostly been in performance, i.e. the amount of vertices and pixels drawn per second. But this is about to change; the new GPU's are not just faster, but like common CPU's, they are also programmable. This means that the GPU can run a program that is written in software, using an assembly language or a high level language if only the program can be compiled for the graphics processor. The fixed functions are no longer the limit when rendering a scene in real-time, now any function or algorithm can be implemented and executed.

The first shader languages were assembly languages, in which a programmer specified instructions directly to the graphics processor. This offers full control, but as with regular CPU assembly programs, they become long and difficult to read and understand. This lead to the development of real-time high level shading languages, and Cg is one of them.

2.4 Computer Graphics Hardware

A Brief History

Computer graphics cards are sometimes divided into four generations, each marking a significant step in GPU development.

The first generation of GPU’s is capable of rasterizing pre-transformed triangles and applying one or two textures. Updating pixels on screen is done completely by the GPU, but it leaves all the vertex transformations to the CPU. This is a clear limitation because the CPU power restricts the amount of vertices in a scene. This generation includes GPU’s such as Nvidia TNT2, ATI Rage and 3dfx Voodoo 3.[1]

The second generation completely relieves the CPU from vertex transformations. However, the hardware uses fixed-function vertex transformation which is configurable rather than programmable. The GPU has a wider set of math and texturing functions, but it is still limited. This generation includes Nvidia GeForce 256, GeForce 2, ATI 7500 and S3 Savage3D.[1]

The third generation is the first to really support programmable vertex transformations. In addition to the conventional, fixed function lighting and transformations, these GPU's offer a set of vertex processing instructions. The pixel

processing configurability is increased substantially, but it cannot be considered truly programmable. This generation includes Nvidia GeForce3 and GeForce4, Microsoft Xbox, and ATI Radeon 8500.[1]

The fourth and present generation includes GeForce FX and ATI Radeon cards. The GPU offloads all vertex and pixel operations from the CPU. These cards are fully programmable on both pixel and vertex level. Applications use Microsoft's DirectX 9 or various OpenGL extensions to reach this programmability. [1]

The Fixed Function Hardware Pipeline

A pipeline is an effective way of using a computer chip for calculating vertices and pixels; it uses the parallelism in the architecture of GPU's. The pipeline consists of several different operations that have to be performed on every vertex or pixel. Each stage receives its information from the previous stage in the pipeline. Because all stages are executed simultaneously, a large number of vertices or pixels can be computed much faster than using a non-pipelining approach.

The first stage in the pipeline is vertex transformation; a number of math operations are performed on each vertex. This includes transforming coordinates, generating texture coordinates, and setting colour, based on material and lights. When completed, this stage sends screen-transformed, lit and textured vertices to the next stage.

The next stage, primitive assembly and rasterization, assembles the vertices into geometric primitives. In other words triangles, lines or points, depending on the vertex' batching information. These geometric primitives are tested against clipping and culling conditions defined by the programmer and the view frustum. The remaining primitives are finally rasterized into fragments, with each fragment representing a corresponding pixel-sized area on the screen/frame buffer. Each fragment, potentially a pixel, has a location, depth value, colour, and texture coordinates. These fragments are sent to the next stage, fragment texturing and coloring. Here the final colour of the fragment is interpolated based on the information from earlier stages in the pipeline. Depth, colour, and texture coordinates are taken into account.

The final stage performs per-fragment raster operations. This is where fragments are eliminated through depth test, scissor test, alpha test, and stencil test. If all tests succeed, the fragment is blended with the corresponding pixel's value and written to the frame buffer, if not, the fragment is discarded without updating the pixel in the frame buffer.

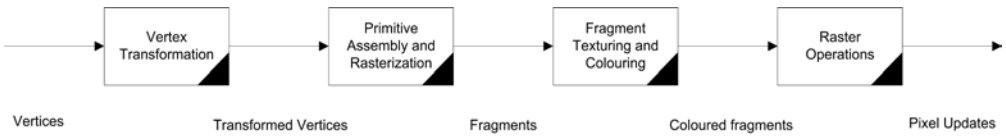

Figure 1. The Fixed Function Hardware Pipeline.

The Programmable Hardware Pipeline

The programmable part of a graphics processor consists of two parts: a vertex processor and a fragment processor. These processors are an alternative step in the fixed pipeline. Figure 2 shows where these programmable parts of the pipeline are located.

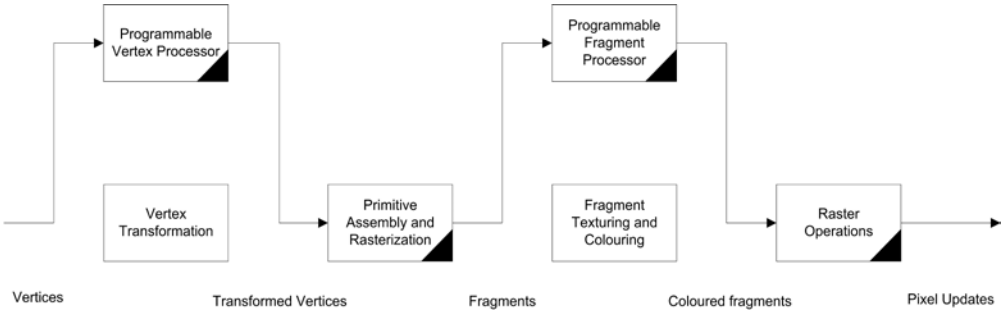

Figure 2. The Programmable Graphics Pipeline.

The programmable vertex processor operates in a number of registers, much like a common processor. There are three different types of registers: Input registers, temporary registers, and output registers. When a new vertex is about to be processed, the vertex' data such as color, position, texture coordinates etc, are loaded into the input registers. A new instruction is fetched from the instruction memory, where the compiled program resides, and the temporary registers are used for intermediate storage of values. This is repeated until there are no more instructions. The resulting data are written to the output registers, which are passed on to rasterization.

The programmable fragment processor retrieves interpolated fragments from the rasterized vertices produced by the vertex processor. The values are stored in input registers and instructions are fetched from the instruction memory. The fragments can, in addition to the vertices in the vertex processor, undergo a number of texture operations. The final colour and depth value are written to the output registers, and these are passed on to the pixel update tests described above, and maybe written to the frame buffer.

2.5 CPU versus GPU

A CPU, or Central Processing Unit, is a general purpose processor, designed for running all types of applications. It might be fast, but since it is not specialized, a thing like processing millions of vertices per second is impossible. On the contrary, a GPU, or Graphics Processing Unit, is designed for just one task, processing vertices and fragments. With this design, tens of millions of vertices and billions of fragments can be processed each second. But on the other hand, a GPU cannot execute general purpose programs. It can only do what it is designed for. This is why a GPU cannot be programmed with a general purpose language, like C++ or Java. A language designed specific for programming a GPU is needed, like Cg, HLSL or GPU assembly.

3. The Cg Language and the CgFX Format

3.1 The Cg Language

Background

Cg is a high level shading language inspired by several concepts in the graphics industry. While Cg is a relatively new language, shading languages have existed since the 1980's. The early shading languages were designed to be used with offline rendering software, such as Pixar's RenderMan, creating photo-realistic effects for movies. Many of the concepts in these languages can also be found in Cg, but adapted to real-time rendering.

The Cg syntax is almost identical to the syntax of the C programming language, only a small number of keywords and types differ because of the different architecture of the graphics processor. Finally, the programmable GPU's as well as the APIs (Application Programming Interface) for 3D programming have influenced the design of Cg. The hardware sets the limits of what can be done with Cg, and the APIs restricts the way the hardware can be used.

Cg is said to be syntactically identical to Microsoft's High Level Shading Language (HLSL). This is because the languages are developed simultaneously and cooperatively between Microsoft and Nvidia. This is discussed in chapter 8.

The Features and Constraints of the Cg Language

Unlike general-purpose languages as C, Cg has no pointer handling, memory allocation, or file input/output operations. However, conditional statements, unbound loops, and functions are supported, but these features are limited by the architecture of today's GPU's. All current graphics hardware has a limited instruction set for shaders, making many features impossible to support. No cards today support dynamic conditions or unbound loops, but since this is not a limitation in the design of the language, they will most likely be supported in future hardware. Cg natively supports data types like vectors and matrices since most graphics operations involve vector or matrix operations. Graphics hardware often support these kinds of operations directly in hardware, so it is natural to have such functions closely integrated into the language. Cg has a simple library called Standard Library that has functions like `reflect` which computes a reflection vector.

The Data Flow Model

In contrast to general-purpose languages, Cg does not operate on an arbitrary set of data or memory. Cg is used to perform a number of operations on a set of vertices or fragments. The vertex or pixel program is executed for each vertex or pixel. The processing of one vertex has no impact on any of the other vertices, so the execution of a program is isolated from the rest of the processed data.

A Small Cg Example

This short, but fully functional Cg program has one very simple task: it changes the colour for each vertex to red.

```
struct vertexout {
            float4 position : POSITION;
            float4 color    : COLOR;
};

vertexout  VertexRed(float2 position : POSITION)
{
            vertexout OUT;
            OUT.position = float4(position, 0, 1);
            OUT.color    = float4(1.0,0.0,0.0,1.0); //Set vertex colour
to red
            return OUT;
}
```

The structure `vertexout` contains the data that is to be sent to the pixel shader. The program creates a new `vertexout` structure and sets the members. When calling `return`, the data is passed on to the pixel shader. This example contains no pixel shader, since one is not needed for this simple example. It can be rendered using the fixed function pixel shader.

3.2 The CgFX Format

The CgFX format integrates vertex and pixel shaders with render states to render objects. It supports Cg programs, assembly language programs and fixed-function shaders. Effects provide a way to combine different shaders to produce unique render conditions.

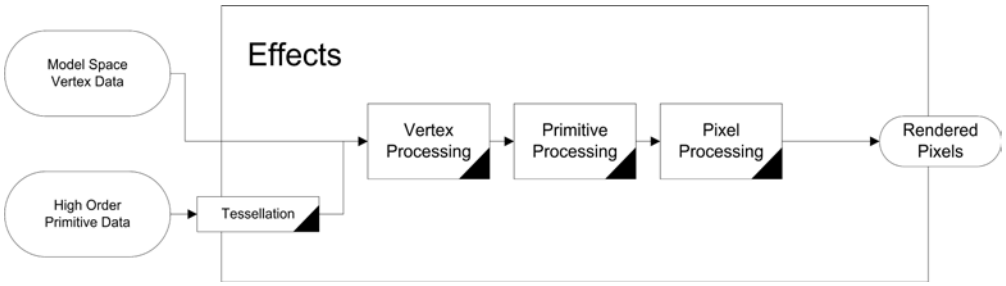

Figure 3. The effect concept. This shows how effects are supposed to manage the graphics pipeline.

The CgFX Runtime

Unlike Cg shader programs, an effect contains more than just shading code. The effect file has to be analyzed using a parser. When parsing an effect, the parser makes sure the syntax is correct, if it is not, the parsing fails and an error message can be read from the effect parser. If it succeeds, the parameters are read into memory and the shader code is compiled with the Cg compiler. When an effect is successfully parsed and compiled, it can be used via the ICgFXEffect interface.

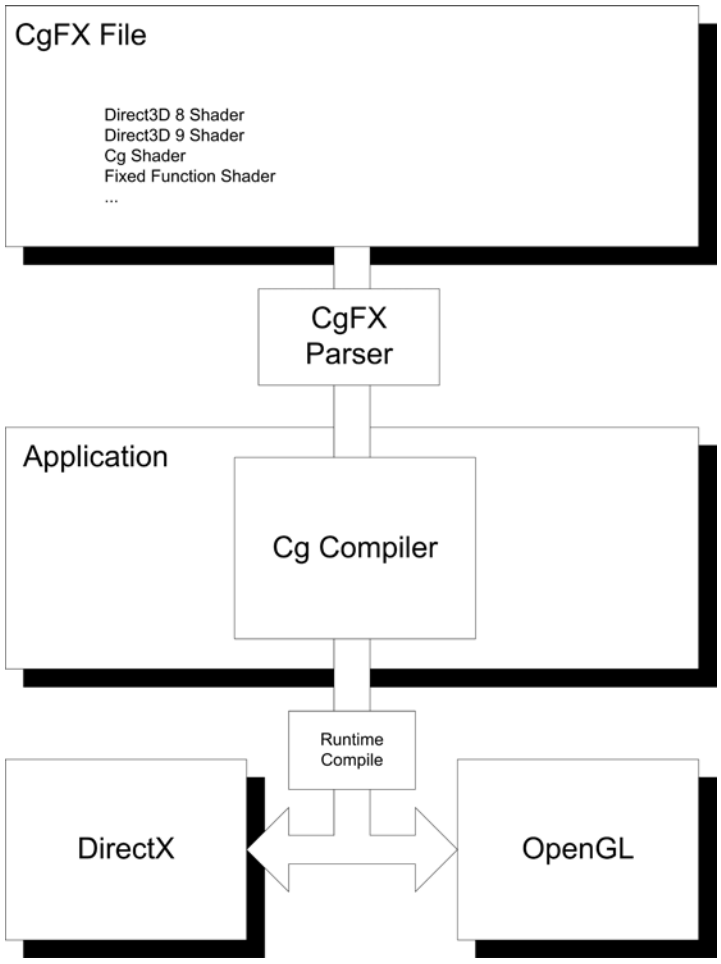

Figure 4. The way CgFX is intended to be used with an application.

The Effect Structure

This scheme is an example of how an effect can be structured. The order of the contents is irrelevant but the passes have to be inside the technique declarations. An effect contains all information needed to apply a certain appearance to an object. The different concepts are explained below.

```
variables
    .
    .
    .

input/ output structures
    .
    .

vextex shader

pixel shader

technique1
             pass1

             pass2

technique2
             pass1

             pass2

             pass3
```

Variables and Structures

The global variables of an effect are the means by which the application communicates with the effect. These can be simple types like floats or integers, but also more complex types like matrices, textures and structures. Matrices are used when transforming vertices, and when computing lighting vectors etc. These must be supplied by the application, and setting them is done in much the same way as setting simple variables. Textures are set with pointers, pointing to a location in the video memory, telling the effect where to get image info.

The structures are handled somewhat differently because they are often used for sending vertex data between the application and the effect, and transformed vertex data from the vertex program to the rasterizer. If for example the structure used for vertex input contains binormal and tangent data, these must be provided by the application. Thus, the vertex format of the application must match the structure in its effect.

Pixel and Vertex Programs

The vertex or pixel programs contained in the effect can be written in Cg or in assembly language. The programs are compiled when the effect is compiled. Any type of shader program can be specified, as long as the effect can fall back on a simpler program if validation fails. Compiler arguments are set in a Pass block.

Techniques

An effect contains one or more techniques, each describing a way to achieve the desired result. Because different GPU's have different functionality, a single Cg program is not sufficient to cover all hardware platforms. However, by using techniques, a different shader program can be specified for each type of hardware. If the GPU does not support one version of fragment shaders it can fall back to a simpler program, or a program describing a fixed-function shader. [1] For example:

```
effect effectName
{
    technique PixelShaderVersion1.4
    {
       ...
       Cg program
       ...
    };

    technique PixelShaderVersion1.0
    {...};

    technique FixedFunction
    {...};
}
```

Passes

Objects are rendered using passes, and a technique can contain one or more passes. Each pass performs one rendering pass, using a vertex shader or/and a pixel shader. A pass can accordingly contain both a vertex shader and a pixel shader together with a set of render states. A render state is typically alpha blending, depth buffer writes or texturing. A pass can contain assembly language shader programs, Cg programs or fixed function processing. [1]

A pass uses the shader programs specified in the effect. The usage of a shader is declared as follows:

```
VertexShader = compile vs_1_1 MyVS(arguments);
```

MyVS is the shader to be compiled, the vs_1_1 argument states which profile to be used when compiling the effect. The different vertex profiles are vs_1_1, vs_2_0 and vs_2_x.

vs_1_1 is the first version, only capable of compiling simple and short programs.

vs_2_0 adds a lot more functionality such as static flow-control conditions. vs_2_x introduces simple dynamic flow control and the possibility to add more instructions. [4]

For pixel shaders the following compile arguments can be specified: ps_1_1, ps_1_2, ps_1_3, ps_1_4, ps_2_0, and ps_2_x. The earliest versions, ps_1_0 – ps_1_4, are very basic, they support no form of flow-control. ps_2_0 added macro instructions like cosine and other math functions, as well as higher instruction count. The latest version adds static flow-control such as if-end-endif, and very simple dynamic flow-control.

4. Python

This chapter is a short description of the Python programming language, how it is used, what the benefits are and how it is used together with C++.

4.1 The Python Language

Python is a high-level, interpreted, object-oriented, open source programming language. It is designed to optimize development speed. Although Python is a completely general-purpose language, it is often called an object-oriented scripting language, as it is commonly used to “glue” software components in an application.

Python was invented in the early 1990's, and originally designed as a scripting language for the Amoeba operating system. Today it is used by over a hundred thousands programmers and engineers in large variety of tasks.[5]

Python is optimized for development speed, not execution speed. The interpreter handles details like type declarations and memory management. Because the language is interpreted, the build procedure of lower-level languages is completely avoided. The simple and clear syntax of the language also contributes to the development speed, because it is easy to use, read, and understand. Because Python is hardware-neutral and operating system-neutral, Python scripts will run on most platforms without being modified. However, there exist non-portable extensions, but the core language is platform-independent. As most scripting languages, Python can be extended with scripts written in other languages. This includes both other scripting languages, and low-level languages like Java, C, and C++. [5]
At the Department of Computer Graphics, a Python renderer is used as a base for the programming assignments. This enables geometric abstractions on scene descriptions on a very high level. A focus is placed on teaching basic computer graphic theory, without having to deal with specific implementation issues. The concept of scene descriptions plays a key role in this thesis when describing how to apply an effect to a scene. Details about scene descriptions and how it is used can be found in chapter 5.

Python is, compared to languages often used in computer graphics, slow. But due to the possibility to call C and C++ libraries, performance-critical operations such as rendering, can be done at a lower level. This, combined with the fact that hardware rendering today is very fast, makes the speed acceptable.

4.2 SWIG - Simplified Wrapper and Interface Generator

Almost every scripting language has some sort of support for running lower level code like C and C++. SWIG is an interface compiler, generating wrapper code that scripting languages (this could be a variety of languages, but here it is assumed that Python is used) need to access methods in a C/C++ library. It is used in this project to make the CgFX Interface accessible to Python.

SWIG uses an interface header file, which can use C++ header files to generate wrapper code. This code can thereafter be compiled to libraries that can be used from Python. In this project it is compiled to a Dynamically Linked Library (DLL). To make the generated module more easily importable from Python, SWIG also generates a Python file that serves as an interface to the DLL.

4.3 Numerical Python

Numerical Python (NumPy) is an open-source extension to the Python language. The extension supplies a way to efficiently handle large arrays of data. These arrays can have any number of dimensions, where a two-dimensional array can be used as matrices from linear algebra. The NumPy extension is written in C for the simple reason that native Python is too slow to perform complex operations on large arrays. NumPy is inspired by numeric languages like Basis and MATLAB, but requires no knowledge of these languages. NumPy can be used with PyOpenGL to handle matrix math and passing matrices to the underlying C-layer. [10]

Because CgFX uses a lot of pre-calculated matrices, NumPy is used in the Python effect framework to handle this math. NumPy is also required for passing matrix data to the PyCgFXEffect module.

4.4 The PyOpenGL library

PyOpenGL is a cross platform, open source OpenGL library for Python. It is created using the SWIG interface compiler. This makes almost the entire OpenGL library available directly from Python. PyOpenGL supports OpenGL v1.1. However, additional functionality can be obtained using various OpenGL extensions.[9]

The main difference between using PyOpenGL and native OpenGL is the way some functions are called. Modifications are made to compensate for the problem with handling data passed between C and Python. For example:

An OpenGL call to a method:

```
void glGenTextures (GLsizei n, GLuint *textures );
```

A PyOpenGL call to a method:

```
glGenTextures  ( n ) •  textures
```

This is because of the differences between C and Python. Python cannot return values as arguments which is a common way to return multiple values in C. However, most of the functions are identical in the way they are called.

5. Scene Descriptions and the Effect Class

This chapter gives a short description of the Effect class and the effects that were intended to be used with the Python effect interface. Implementing these effects was a part of the goal for this thesis, and some effects were implemented successfully and some were not. A review of which effects were successfully implemented can be found in chapter 8.

5.1 The Effect Class

The effect is constructed using the PyCgFXEffect module, and it is the interface through which effect rendering is done in python. The purpose of this class is to handle all types of effect rendering through one simple interface. This is done using dynamic parameter handling inside the effect class. A Simple Effect class capable of rendering many effects, including effects with textures and environmental mapping can be found in Appendix A.

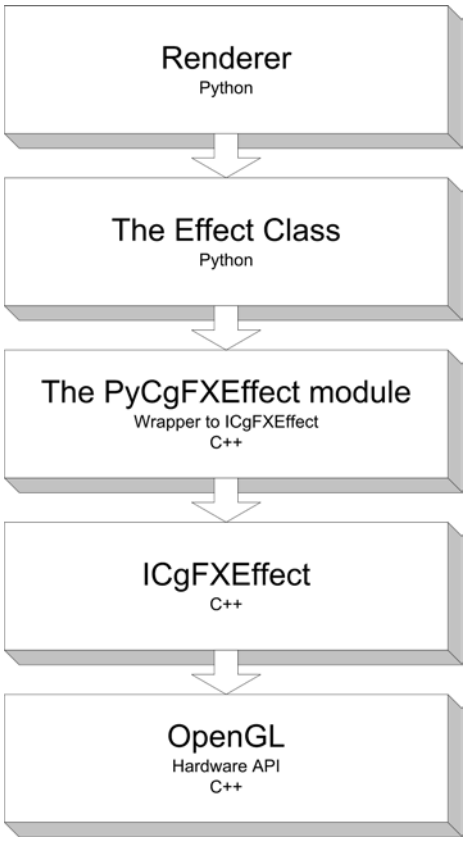

Figure 5. The way the Effect class interacts with the hardware and a renderer.

Figure 5 shows how the Effect class is supposed to interact with a renderer an the underlying hardware. The ICgFXEffect interface uses OpenGL or Direct3D to interact with the hardware, in this case OpenGL. PyCgFXEffect is a wrapper for ICgFXEffect to make it accessible from Python. The Effect class is implemented using the PyCgFXEffect module. OpenGL is also accessed from the effect class and the renderer. These calls to OpenGL are done via PyOpenGL.

The PyCgFXEffect module is described in chapter 6, and the wrapping of the ICgFXEffect interface in chapter 7.

Scene Descriptions

A scene description is a high level abstraction of a scene. It is used to describe objects, appearances, cameras, lights, etc. The purpose of the effect class is to use it as a part of a scene description.

When using the Effect class together with a scene description, the constructor of the class is used to set the effect and all parameters needed to render a particular appearance. The class always takes the effect filename as an argument, but this is the only required argument, since the rest of the arguments depend on the parameters of the specified effect. Most effects has default values for these parameters, so when supplying only the filename, the effect is rendered using its default appearance.

The Effect Class Structure

The goal of the Effect class is to keep it general and at the same time as simple as possible. As mentioned above, the constructor takes an arbitrary number of parameters for initializing an effect. In addition to the filename, other parameters can be specified using their global variable names from the effect file(see chapter 3). The parameters left out are set to default values. If the set parameter does not exist in the effect, an error is generated.

The `__setattr__` and `__getattr__` methods are special methods used for making the parameters of the effect editable through dot notation. For example, setting a vector at runtime could be done like this: `effect.lightPos = (10, 0, 20, 1).` Similarly, values can be fetched using for example: `light = effect.lightPos`

```
class Effect

            def __init__(self, initialization parameters)  #constructor

            def __setattr__(self, paramname, value)

            def __getattr__(self, paramname)

            def begin(self)

            def pass(self, pass number)

            def end(self)
```

5.2 Scene Examples

A scene description for a renderer can look like this:

```
Cylinder( transform = Scale(( 1, 1.5, 1)),
          appearance = Material(ambient   = ( 0.11, 0.11, 0.29 ),
                                diffuse   = ( 0.37, 0.38, 0.84 ),
                                specular  = white,
                                shininess = 20.0 ))
```

A material is defined and used as the appearance of the cylinder.
The Effect class is intended to be used as an appearance. For example, a scene description to render the cylinder using an effect called “goochy2.fx”:

```
Cylinder( transform = Scale((1, 1.5, 1)),
          appearance = Effect((effectfile = "goochy2.fx",
                                warmColor = red),
                                coolColor = blue),
                                specExpon = 10.0))
```

All the effects below will be described along with an example of a scene description.

5.3 The Interface to a Renderer

When used together with a Python renderer, the interaction with the Effect class is done using the methods `Begin()`, `Pass(pass_number)`, and `End()`. These three methods control which geometry to render and what pass to apply. When calling `Begin()` all render states and shader data is saved by the Effect class.

Effects are rendering are performed using passes, see chapter 3. The `Begin()` method sets all matrices that are not set by the renderer itself. If the matrices are not set when `Begin()` is called, the current model view and projection view matrices are used. The method also returns the number of passes used in the current effect.

To perform an actual pass, the method `Pass(pass_number)` is called with the pass number as an argument.
When all passes are done, `End()` is called to restore all render states and shaders.
This is an example of what a rendering loop can look like in a renderer:

```
passes = effect.begin()
for p in range(passes):
            effect.pass(p)
            draw_geometry()
effect.end()
```

5.4 Effect Examples

This section contains descriptions of a number of effect types. The theory behind them is not covered in detail, since many of the effect types involve quite extensive theories. All effects are described along with a scene description. The scene

description gives a more specific example of an effect that falls in the range of possible effects for that effect type.

Per-Fragment Lighting

Standard real-time lighting, or Gouraud shading, is dependent on the detail of the underlying mesh. The light is computed for each vertex and the interpolated when rasterizing each triangle.

To achieve better results, lighting calculations have to be performed on a per-fragment basis. This means that the lighting function is evaluated for every pixel on the object and therefore gives a much more accurate implementation of lighting. Both per-vertex lighting and per-fragment lighting are based on Phong shading, so the only difference is the level at which the calculations are performed. [1]

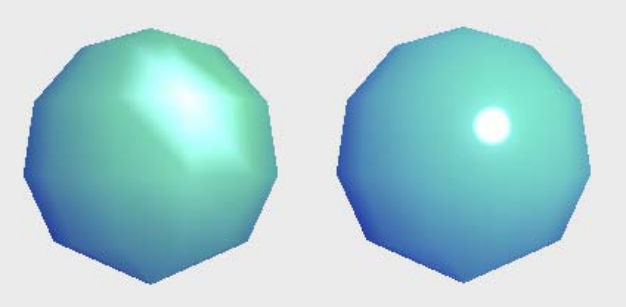

Figure 6. Per-vertex and per-fragment lighting.

```
aBall = Ball(transform =
                MoveTo(( 0.0, 0.0, 0.0))*
                Scale(( 1.0, 1.0, 1.0)),
            appearance = Effect("pixellighting.fx",
                diffuse = (0.5,0.5,1.0),
                specular = (1.0,1.0,1.0),
                power = 10.0))
```

Hemispheric Lighting

The hemispheric lighting algorithm approximates an area light. Area lights are the light reflected by the surrounding environment. Because the surroundings can be complex, accurately calculating this light is a very extensive procedure. To avoid this, the light reflected is approximated by two directional lights, a sky light and a ground light. The colors of these lights are set to match the surroundings, giving a realistic illusion of light reflected by the environment. The hemispheric light component is calculated using a linear interpolation. The Cg function `lerp(a, b, s)` linearly interpolates between a and b, such that the return value is a when s is 0, and b when s is 1. [4]

```
Hemi = lerp(groundColor,
             skyColor, (dot(Normal, dirFromSky) + 1) / 2);
```

Scene Description:

```
aSkull = Skull( transform =
                    MoveTo(( 1.0, 0.0, 0.0))*
                    Scale(( 1.0, 1.0, 1.0)),
                appearance = Effect("hemispheric.fx",
                    groundColor = (0.6,0.3,0.3),
                    skyColor = (0.6,0.6,1.0)))
```

Figure 7. Hemispheric Lighting

Bump Mapping

Bump mapping is a large and advanced topic in computer graphics. The goal is to give the illusion of high level details on objects in a scene, without increasing the number of polygons used in the objects. The technique is a per-fragment technique, and all lighting calculations are performed on every pixel. There are several ways of implementing bump mapping, and one of these will briefly be described here.

Instead of using a high level tessellation, bump mapping uses an extra texture. This texture represents the normals on the surface in tangent space. The normals are encoded as colors, where the RGB values for the texture corresponds to the direction of the normal. Because the normals are in tangent space, the normal texture can be applied to any type of object in a scene. The light reflected from each pixel on the surface is modified with consideration to the normal at that pixel.

Scene Description:

```
aBumpWall = Box( transform =
                     MoveTo(( 0.0, 0.0, -5.0))*
                     Scale(( 5.0, 5.0, 0.1)),
                 appearance = BumpEffect("DiffuseBumpCg.fx",
                     bumpheight = 1.0,
                     normalmap = "LU_logo.dds",
                     basetexture = "wall.png"))
```

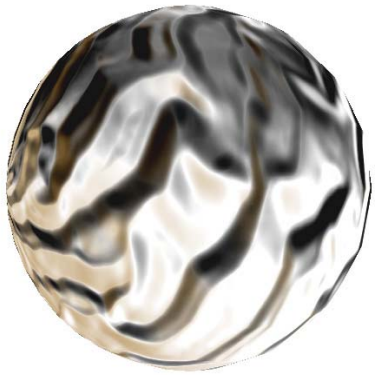

Figure 8. A bump mapped sphere.

Cartoon/Cell Shading

Most shading techniques try to resemble the real world as much as possible. Cartoon shading is an example of a technique that does not. This effect renders the scene as if it were taken from a cartoon movie or a comic book.

The idea behind cartoon shading is to convert the diffuse color spectrum to just a few shades of color. This is done by using a step function instead of the continuous function commonly used when computing lighting colors. A step function can be done using one-dimensional textures to look up what color to draw:

```
diffuseLighting = tex1D(diffuseTex1D, diffuseLighting);
```

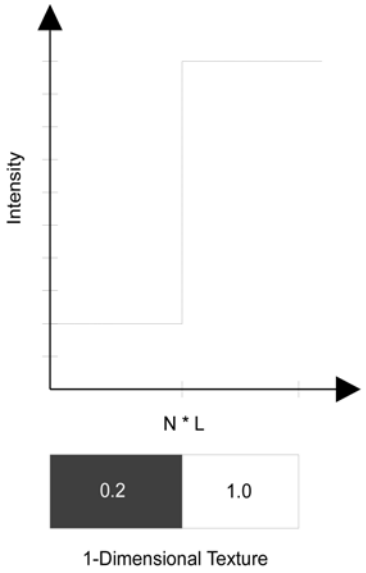

Figure 9. A 1-Dimensional Texture used to represent a step function. N is the normal vector and L is the lighting vector.

To further enhance the look of a cartoon movie, an edge-detection function to draw black or dark outlines around the objects can be used.

Scene Description:

```
aTeapot = Teapot(transform =
                     MoveTo(( 0.0, 0.0, 0.0))*
                     Scale(( 1.0, 1.0, 1.0)),
                 appearance = Effect("Cartoon.fx",
                     stepDiffuse = "sTexDiffuse.dds",
                     stepOutline = "sTex02",
                     diffuseColor = (1.0,0.0,0.0),
                     edgeColor = (0.0,0.0,0.0)))
```

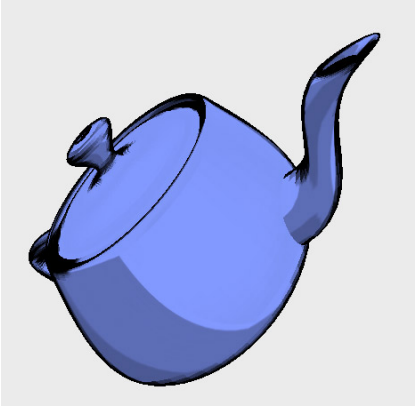

Figure 10. A cartoon shaded teapot.

Environment mapping

Environment mapping is a technique which simulates an object reflecting its surroundings. It can be used to create effects like reflection, refraction, dispersion, and the Fresnel effect. All these effects are achieved using cube textures. A cube texture is not one but six different textures, each representing a view along the positive and negative x, y, and z axes. The cube texture can be rendered using six cameras, or pregenerated to increase performance. A pregenerated texture does not reflect the actual surroundings, yet it still gives a good illusion of a chrome-like material.

Reflection is the most basic effect, in which a reflection vector is computed for each vertex. The reflection vector is then used to look up texture coordinates in the cube map. The Cg function `reflect` is used to compute the reflection vector:

```
R = reflect(I, N);
```

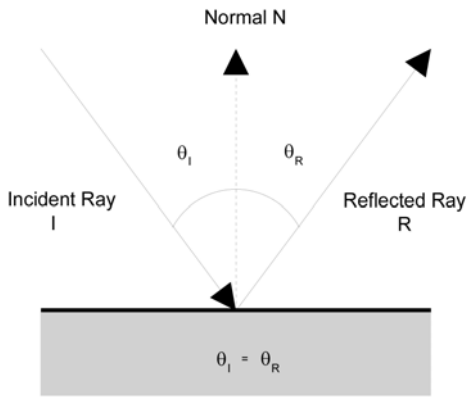

Figure 11. The Reflection Vector.

Figure 12. An environment mapped teapot.

To simulate refraction in transparent materials like glass and water, Snell's law can be used to compute a refraction vector. The angle at of the refraction vector depends on the index of refraction of the material. A material's index of refraction measures how the material affects the speed of light. Water has a relatively low refractive index while diamonds have a very high value. The refraction vector is used to lookup texture coordinates in the cube map. The refraction vector can by computed in Cg using this function: [1]

```
T = refract(I, N, nRatio);
```

Here, I is the incident vector, N is the normal and nRatio is the ratio between the indices of refraction of the two materials.

Most transparent materials both refract and reflect the incoming light. Whether it is refracted or reflected depends on the angle which the light reaches the surface of the material. This effect is called the Fresnel effect. The equations required to compute an accurate model of the optics involved are complicated. However, approximations to these equations can be made which give sufficiently good results. This equation can be used to compute the reflection factor: [1]

```
reflectFactor = fresnelBias +
     fresnelScale * pow(1 + dot(I, N), fresnelPower);
```

Dispersion is the phenomenon where refracted light spreads into a rainbow, like light passing through a prism. The dispersion occurs because the speed of light passing through a material also depends on the wavelength of the light. The effect can be simulated using several cube textures, each representing a particular color. These textures are then applied using an offset.

Procedural Textures

Procedurally generated textures are created using conditions and mathematical functions. These textures are often volume textures which have a third dimension. Procedurally created textures can be used to create effects like wood, water, volumetric fog, noise, and lightning bolts. Texture generation is demanding for the hardware and large textures cannot be generated at runtime, and only the fourth generation of graphics hardware supports this type of texture operations. The wood texture on the teapot in Figure 12 uses this noise function to generate the texture:

```
float4 GenerateNoise(float3 Pos : POSITION) : COLOR
{
    float4 Noise = (float4)0;

    for (int i = 1; i < 256; i += i)
    {
        Noise.r += abs(noise(Pos * 500 * i)) / i;
        Noise.g += abs(noise((Pos + 1)* 500 * i)) / i;
        Noise.b += abs(noise((Pos + 2) * 500 * i)) / i;
        Noise.a += abs(noise((Pos + 3) * 500 * i)) / i;
    }

    return Noise;
}
```

Figure 13. A procedurally generated wood texture.

Vertex Displacement

Displacing the positions of the vertices is easy to do in a vertex program since the transformation of vertices is a step required in every vertex program. An object can be deformed using various functions to achieve effects like ripples, inflated objects, and twisting. Other applications include interpolation between different key frames, to fill the gap between the frames. A vertex program could produce a very simple animation by interpolating between only two key frames.

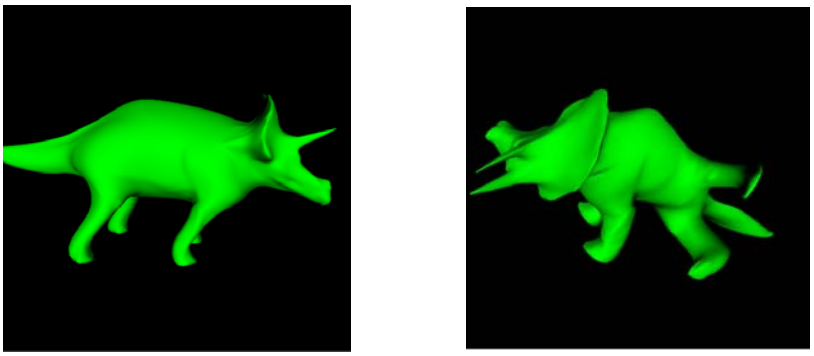

Figure 4. Vertex Displacement

Vertex Blending

A commonly used vertex displacement technique is called *vertex blending* or *vertex skinning*. Instead of having a key frame for every pose of a character or shape of an object, this technique uses a basic pose and a large set of matrices describing the rotation, scaling and translation of subsets of the basic mesh. These matrix calculations are much less demanding than providing a mesh for every single key frame. [1]

Each of the matrices in the set has a weighting factor which indicates how much that matrix affects each vertex. The weighting factor ranges from 0 to 100 percent. Only a few of the matrices have positive weighting factors for each vertex, these matrices are called a *bone set*. Each vertex has a bone set, and the sum of the weights of the matrices in this set is always 100 percent.

When rendering the model, each vertex is multiplied with all matrices and their weights in its bone subset. The sum of the results is called the skinned vertex position, and it is the position at which the vertex is placed.

The scene description, in this example, the Effect class is subclassed to take another mesh as an argument:

```
aDolphin = Mesh( meshfile = "dolphin.x",
                 transform =
                     MoveTo(( 0.0, 0.0, 0.0))*
                     Scale(( 1.0, 1.0, 1.0)),
                 appearance = BlendEffect("vertexblend.fx", blendToMesh=
                                                    "dolphin2.x"))
```

Particle Systems

Another application for vertex displacement is particle systems. Here, every vertex is treated as a point, or *particle*, moving according to a specified function. With a large number of particles, effects like smoke, sparkles and snow can be implemented.
For example, a particle following a projectile motion can be described using this equation in Cg which is computed every frame:

```
finalPos = initPos + initVel * time + 0.5 * acceleration * time * time;
```

6. The PyCgFXEffect module

This chapter describes how to use the PyCgFXEffect module in a Python program. It explains how to initialize, load, and render an effect in a PyOpenGL environment.

6.1 Creating and Loading an Effect

The PyCgFXEffect module can be used as a generic Python class, it can be instantiated and the constructor has no additional arguments:

```
from PyCgFXEffect import *
pyeffect = PyCgFXEffect()
```

When loading an effect, a filename for the effect to be loaded must be supplied.

```
pyeffect.PyCgFXCreateEffectFromFileA(filename, 0)
print errors
```

By calling this method, the effect is compiled by the CgFX runtime. Any errors are returned by the method as a string. The last argument is not yet used by CgFX, so it is set to 0. If no errors where found during compilation, the compiled effect is loaded into memory.

Before the effect can be used, a technique must be validated. Validation is a way to find out if the current hardware can run the effect. As described in chapter 3, an effect can have several techniques. A technique describes one possible way of achieving the desired result. If no technique is validated, the hardware is not capable of running the effect. The validation procedure is done by first setting a technique, and then checking whether it is validated:

```
pyeffect.SetTechnique(techniqueNum)
pyeffect.Validate()
```

The Validate method returns true or false.
If a technique is validated, the effect is ready for rendering.

6.2 Descriptions

All effect parameters are handled using descriptions. This is a way to control and change the variables in an effect, including things such as textures, colors, light positions, and matrices.

Effect Description

A compiled effect has an effect description, which can be obtained using the following method:

```
effectDesc = pyeffect.GetDesc()
```

The returned data is a simple class containing three members, these members are:

`effectDesc.Parameters`	The number of parameters
`effectDesc.Techniques`	The number of techniques
`effectDesc.Annotations`	The number of annotations

Effect Parameter Descriptions

The parameters in an effect are handles to the global variables in an effect file. These variables are described using parameter descriptions, which there are as many of as there are parameters in the effect. Effect parameters do not have to be set when the effect is compiled, default values are given. If no default value is specified in the effect file, the value is set to zero, or null in case of a string. A parameter description is a class with the following members:

`Name`	The Unique name of the variable.
`Index`	This index is used when setting parameter values, see below.
`Semantic` parameter.	A short text string describing the intended usage for the For example, a `DiffuseMap` parameter is intended to be used as a diffuse texture.
`Type`	The variable type. Can for example be `CgFXPT_FLOAT` or `CgFXPT_TEXTURE`. See Appendix B For a complete listing.
`Annotations`	The number of annotations for this parameter.

A parameter description can be obtained using the following method:

```
paramDesc = pyeffect.GetParameterDesc(index)
```

The index argument corresponds to the index of the parameter description. A convenient way of handling the parameters to an effect is using a python tuple:

```
for i in range(0, effectDesc.Parameters):
            parameterList.append( pyeffect.GetParameterDesc(i) )
```

A member in the `parameterList` can then be accessed using dot notation. For example, this prints the semantics of an effect:

```
for param in parameterList:
            print param.Semantic
```

Effect parameters can be set at any time during initialization or rendering.

Annotations

Annotations are user specified data that can be attached to a parameter. The information is not crucial and the application can choose whether to use them or not. Annotations are handled in a similar way to parameter descriptions. The members are:

`Name` The name of the annotation.

`Index` The index of the annotation. A parameter description can have several annotations, starting at 0. Use this index to distinguish them.

`Value` The value of the annotation, to read the data correctly, the type of the annotation must first be checked.

`Type` The possible data types for an annotation are the same as for a parameter.

Use this method to get an annotation:

```
annotation = pyeffect.GetAnnotationDesc(parameter.Index,AnnotationIndex)
```

Here, the `parameter.Index` is the parameter description that contains the annotation. Annotations can be used to read user-specified texture filenames from an effect.
If the parameter is known to be a texture, the filename can be obtained like this:

```
for i in range(0,parameter.Annotations):
            annotation = pyeffect.GetAnnotationDesc(parameter.Index, i)
            if annotation.Name = "File":
                        filename = annotation.Value
```

6.3 Setting Parameters

To change the appearance of the effect, the values of the parameters can be changed. The parameters themselves have no information about the values of the parameter, since the parameter descriptions are merely handles to the values. The current values have to be fetched from the effect. All parameters can be changed at runtime, but they cannot be changed inside a `Begin()` - `End()` block.

Setting and Getting Basic Parameters

Setting values like vectors or floats are done through various methods in the effect. Methods for setting values are listed in appendix B. Here are some examples of how to use these methods:

Setting and getting a float:

```
pyeffect.SetFloat(parameter.Index, specPower)
```

```
specPower = pyeffect.GetFloat(parameter.Index)
```

Setting and getting a vector:

```
pyeffect.SetVector(parameter.Index, lightPos, 4)
lightPos = pyeffect.GetVector(parameter.Index)
```

Here, `lightPos` is a Python tuple with length 4, containing values for a light position.

Setting Matrices

All matrices needed for rendering a scene have to be provided by the application. Some matrices have to be set every frame, and some can be set only once, for example if using a fixed camera. An effect might need more than the two matrices used in OpenGL, the model view matrix and the projection matrix. These additional matrices have to be computed by the application, and then passed to the effect. The reason for this is to save a lot of computational work for the vertex program, because it would have to do several matrix operations for each vertex.

Matrices are handled using NumPy's arrays. This is the same way matrices in PyOpenGL are handled, so conversions and operations are very easily performed. Some matrices can be set directly, and some must first be computed. This is a simple case, illustrating how to find and set the model view matrix:

```
for parameter in parameterList:
     if parameter.Semantic == "ModelView":
             modelView = glGetFloatv(GL_MODELVIEW_MATRIX)
             pyeffect.SetMatrix(parameter.Index, modelView, 4, 4)
```

The first argument is the parameter description describing the model view matrix, the second is the matrix to be passed to the effect, and the last two arguments are the dimensions of the matrix. Finding or keeping track of which parameters correspond to the right matrix can be done in many different ways, the example above is a simple but ineffective way.

Setting Textures

Any texture used in OpenGL can be used by an effect. Since textures reside in video memory, the effect only has to know which texture to use, or where to read from the video memory. Setting a texture is a single call to the effect:

```
pyeffect.SetTexture(parameter.Index, texture)
```

where `texture` is a standard OpenGL texture.

Loading and setting a texture with Pynv_dds

The DDS file format (DirectDraw Surface) is an image format designed to be used for textures. The format can hold information about alpha channels, mip-maps,

compression, and whether the texture is a cube map or a volume map. It was originally a format used by Microsoft's DirectX API, but Nvidia provides software for loading these textures in OpenGL. The software is called nv_dds and has been wrapped using SWIG, and the module is called Pynv_dds. The following illustrates how to load and set a DDS texture.

First, the module has to be imported:

```
from Pynv_dds import *
```

Pynv_dds contains a class called CDDSImage, which handles the loading of images:

```
ddsimage = CDDSImage()

texture = glGenTextures(1)
ddsimage.load("default_color.dds")
glBindTexture(GL_TEXTURE_2D, texture)
ddsimage.upload_texture2D()
pyeffect.SetTexture(parameter.Index, texture)
```

6.4 Rendering

When rendering an effect, the fixed function pipeline is replaced by the vertex and fragment programs specified in the effect file. All render states are automatically set by the effect. The rendering is done using passes, and with each pass the geometry to be rendered has to be drawn. The passes handles setting render states and shaders. Before performing the passes, the Begin method, which returns the number of passes, has to be called:

```
numPasses = pyeffect.Begin(flags)
```

The flags argument is not yet implemented in CgFX, so this value is ignored. The purpose of the flags argument is to specify whether the effect shall remember what render states and shaders were set before calling End, but this does not seem to be implemented yet, see chapter 8 for comments on bugs.

The next step is to iterate over the number of passes, each pass draw the geometry that is to be rendered. This can be done using a simple for-loop:

```
passes = pyeffect.Begin()

for p in range(passes):
    pyeffect.Pass(p)

    DrawGeometry()

pyeffect.End()
```

Calling End tells the effect that all rendering using this effect is done.

7. Wrapping the ICgFXEffect Interface

This chapter explains how the ICgFXEffect Interface is accessed from Python.

7.1 The Effect Interface

The effect interface, ICgFXEffect, is a part of the Nvidia Cg Toolkit. It is designed to integrate effects into an application. The interface consists of a C++ header file with an accompanying DLL library. Thus, the actual implementation of the interface is hidden, but can be accessed through the header file. The header file is also sufficient for wrapping the interface.

To be able to use the interface from Python, some modifications had to be done, as the interface could not directly be wrapped with SWIG. This is because the differences between C++ and Python in the way arguments and types are handled. This chapter explains the modifications and measures that has been made to be able to wrap the interface. A header file called PyCgFXEffect.h is used as a layer between the actual interface and the wrapper. This header file is used to create the Python module.

The ICgFXEffect.h file, and the PyCgFXEffect.h file, used for wrapping, can be found in appendix B resp. appendix C.

7.2 Converting Types

A common problem when wrapping code written in C++ to be made accessible from python are the different ways the languages handles data types. C++ uses static data types, meaning that they must be defined before compilation. The data types include floats, integers, booleans and characters. Data can also be grouped together as structures or classes. At runtime, the type information is lost, so there is no way of checking what data type a block of memory is. On the other hand, Python handles data types dynamically, checking types at runtime. All data types in Python are objects, these objects contains information about what type they are or to what class they belong.

SWIG automatically handles basic data types, but structures and pointers have to be manually converted using typemaps. When generating code, SWIG inserts these typemaps when needed. This is an example of a typemap which converts a Python tuple to a float[4] array:

```
%typemap(python,in) float[4] (float temp[4]) {
if (PyTuple_Check($source)) {
       if (!PyArg_ParseTuple($source,"ffff", temp, temp+1, temp+2, temp+3)) {
            PyErr_SetString(PyExc_TypeError,"expected a 4-element tuple of
floats");
              return NULL;
       }
       $target = temp;
} else {
       PyErr_SetString(PyExc_TypeError,"expected a tuple.");
       return NULL;
```

```
        }
    }
```

The typemap handles all method calls from Python were an argument in the C++ code is a float[4] vector. The $source variable is a Python object containing the data passed as an argument to the method in Python. `PyTuple_Check($source)` checks whether $source really is a Python tuple. If it is, the data is parsed from the object, or else an error is raised.

7.3 Using Structures

Many of the parameters in CgFX are handled using structures. To be able to access the members of these data structures from Python, they have to be converted to classes with public variables. For example this structure used in ICgFXEffect:

```
typedef struct _CgFXEffect_DESC{
            UINT Parameters;
            UINT Techniques;
            UINT Functions;
} CgFXEFFECT_DESC;
```

is changed into a class in PyCgFXEffect:

```
class PyCgFXEFFECT_DESC
{
public:
            UINT Parameters;          // Number of parameters
            UINT Techniques;          // Number of techniques
            UINT Functions;           // Number of functions
};
```

The purpose is to gain access to the members as they are accessed in C++. For example:

```
Desc = effect.GetDesc()
print Desc.Parameters
```

Some of the members of these structures have been changed because Python and C++ do not handle characters the same way.

7.4 Methods Returning Data as Arguments

A common way of handling multiple return values in C++ is by passing a pointer to the variable or data as an argument to a function. For example, when a new interface is created in ICgFXEffect, a pointer to the interface pointer is passed as an argument to this function:

```
HRESULT CgFXCreateEffectFromFileA(
        LPCSTR              pSrcFile,
        DWORD               Flags,
        ICgFXEffect**       ppEffect,
        const char**        ppCompilationErrors);
```

The method fills the `ppEffect` pointer with data. However, this would require Python to send a reference to an object. Since Python doesn't handle references, this is impossible, but there are workarounds. The only thing that really has to be returned is the error string. The ICgFXEffect instance is handled as a member variable inside the PyCgFXEffect. Therefore, the method can be written like this:

```
const char* PyICgFXEffect::PyCgFXCreateEffectFromFileA(
        char* pSrcFile,
        DWORD Flags)
{
       const char* errors;
       CgFXCreateEffectFromFileA((LPCSTR)pSrcFile,
                                 Flags,
                                 &m_pICgFXEffect,
                                 &errors);
return errors;
}
```

8. Discussion and Conclusions

8.1 The Result

The primary goal of this thesis was to examine the possibility of applying effects to a Python scene. This has been successfully implemented by wrapping the CgFX format so that it could be used together with Python. However, there was not enough time to make the Effect class completely general. The one that is supplied in appendix A shows that effect rendering in Python is possible, but it does not handle all parameters and cases. To achieve the goal of this thesis, an extensive study of shader programming, the Cg language, CgFX, and the way Python interacts with other languages has been done. This report is a result of this study.

8.1 What Types of Effects Were Renderable?

A secondary goal of this thesis was to implement an Effect class which is able to render all of the effect types in chapter 5. However, time showed to be the largest hindrance, because some of the effects demand extensive modifications to the application, or in this case, the Effect class.

Cartoon shading, environment mapping, and bump mapping where implemented successfully and is renderable using the Effect class.
The Hemispheric lighting was written for the Direct3D version of the effect format. There was not enough time to convert it. Vertex displacement was not implemented since no effects capable of doing this were supplied with the Cg Toolkit. Vertex displacement also requires a slightly different approach when handling the vertex streams from the application. An effect can be created from existing Cg programs, but again, there was not enough time to do this.

Per-fragment lighting and procedural textures were impossible to implement with the current version of CgFX. This is because CgFX does not support pixel shader version 2.0 in OpenGL, which is required to render these effects.

8.2 The Beta Version of CgFX

As mentioned several times before Nvidia has not released an official version of CgFX. The Interface for using CgFX is buggy, and lacks documentation. A lot of time was spent on trying to figure out the way the interface works. However, since Nvidia's CgFX format and Microsoft’s effect format is supposed to be identical, some documentation could be found for the Microsoft version of the format. But the formats are not exactly identical, most effects written in Microsoft’s format does not compile without adjusting them to CgFX. This is mostly differences between the two shader languages; HLSL and Cg. For example the way a function is called can differ in the order or number of arguments. The CgFX format itself lacks some of the abilities that Microsoft’s format has, for example; to specify meshes, background colour and defining values with a `#define` keyword. There are no significant differences, but several small differences will make it quite hard to locate errors

since the error output from the compiler is not nearly as good as for example most C++ compilers are.

There are also differences in the way the effects are integrated into an application. Microsoft's interface, the ID3DXEffect, is more complete compared to ICgFXEffect. It is obvious in this example: The pass description for a CgFX effect contains no information on what data to send to the vertex shader. For example, it does not tell whether the application should send binormal and tangent data when rendering an effect which need this data. The result is that the application must be more specifically designed for particular effect.

8.3 Alternative Approaches

The effect approach was chosen early in the thesis, but this is only one way of integrating shaders into Python or any scripting language. Microsoft's Direct3D9 effect format could have been wrapped using SWIG, but since this requires wrapping a large part of the Direct3D9 API, it is a much harder task to do. Another approach is wrapping the Cg interface to work with Python, this is already done and a version called PyCg can be found at [http://www.csit.fsu.edu/~mason/?section=projects:personal:pycg].

The difference in wrapping Cg instead of CgFX is that more control over the Cg language is possible. An application would be able to handle the rendering more effectively, but at the same time the shader programs would be more specific to the application.

8.4 Future Improvements

The single largest improvement is to write a more general Effect class, and fully integrate it into the Python rendering system used at the Department of Computer Graphics. When, or if, an official version of CgFX is released, adjusting the whole Python effect interface to adapt to the new version would be a natural improvement.

Acknowledgements

I would like to thank my supervisor Lennart Ohlsson for guiding me through this work. I would also like to thank Calle Lejdfors for helping me out with the tricky parts of Python, SWIG and OpenGL.

References

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

[2] Wolfgang F. Engel, (2002), *Direct3D ShaderX*, Worldware Publishing, Inc.

[3] Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner, (1999), OpenGL Programming Guide: The Official Guide to Learning OpenGL, Third Edition,

[4] Kris Gray, (2003), *Directx9 Programmable Graphics Pipeline*, Microsoft Press/Microsoft Corporation.

[5] Mark Lutz, (2001), *Programming Python. Second Edition*, O'Reilly & Associates, Inc.

[6] *The SWIG Homepag*e [http://www.swig.org].

[7] *The Nvidia Developer Website* [http://developer.nvidia.com].

[8] *The Cg Language Development Forum* [http://www.cgshaders.org].

[9] *The PyOpenGL website* [http://pyopengl.sourceforge.net].

[10] *The NumPy Online Manua*l [http://www.pfdubois.com/numpy/numpy.pdf].

Appendix A: The Effect Class

This is the source code for a simple effect class.

```
from OpenGL.GL import *
from OpenGL.GLUT import *
from OpenGL.GLU import *
from OpenGL.GL.ARB.texture_cube_map import *
import sys
from Numeric import *
from LinearAlgebra import *
from PyICgFXEffect import *
from Pynv_dds import *
from types import *


class Effect:

	# contains the matrices that should be set by Begin()
	parametersToSetAtBeginDict = {}
	# A dict which manages the parameter descriptions in the effect
	parameterDict = {}


	def __init__(self, filename= 'default.fx', **kwords):
		self.effect = PyICgFXEffect()
		self.parameterDict = {}
		self.matWorld = identity(4,typecode=Float)

		#create the effect
		errors = self.effect.PyCgFXCreateEffectFromFileA(filename, 0)
		print errors

		self.effect.PyCgFXSetDevice('OpenGL',0)

		effectDesc = self.effect.GetDesc()

		# iterate over the number of parameters in the effect and save them in the parameterDict
		for i in range(0,effectDesc.Parameters):
			pDesc =  self.effect.GetParameterDesc(i)
			if pDesc.Semantic in [ 'WorldIT', 'World', 'ViewIT', 'WorldViewProjection' ]:
				self.parametersToSetAtBeginDict[pDesc.Name] = pDesc
			self.parameterDict[ pDesc.Name ] = pDesc

		self.effect.SetTechnique(0)
		self.effect.Validate()

		self.ddsimage = CDDSImage()

		# Set standard values
		for attr,val in kwords.items():
			setattr(self,attr,val)


	def __setattr__( self, attr, val ):
		if not self.parameterDict.has_key(attr):
			self.__dict__[attr] = val
			return

		#extract the description to use for setting the value
		pDesc = self.parameterDict[attr]

		# Is it a matrix?
		# Only matrices and textures are set in this version
		if pDesc.Type == CgFXPT_FLOAT:
			if self.parametersToSetAtBeginDict.has_key(attr):
				del self.parametersToSetAtBeginDict[attr]

			self.effect.SetMatrix(pDesc.Index, val, 4, 4)

		# A texture
		elif pDesc.Type == CgFXPT_TEXTURE:

			texture = glGenTextures(1)

			# Load the texture
			self.ddsimage.load(val)

			if self.ddsimage.is_cubemap():
				glBindTexture(GL_TEXTURE_CUBE_MAP_ARB, texture)
				self.ddsimage.upload_textureCubemap()
			elif self.ddsimage.is_volume():
				glBindTexture(GL_TEXTURE_3D, texture)
				self.ddsimage.upload_texture3D()
			elif self.ddsimage.get_height() == 1:
				self.ddsimage.upload_texture1D()
			else:
				glBindTexture(GL_TEXTURE_2D, texture)
				self.ddsimage.upload_texture2D()
```

```
                        self.effect.SetTexture(pDesc.Index, texture)


def __getatt__( self, attr):

        pDesc = self.parameterDict[attr]
        if pDesc.Type == CgFXPT_FLOAT:
                return self.effect.GetMatrix(pDesc.Index, 4, 4)


def Begin(self):

        # Set those matrices which are not already set
        for p in self.parametersToSetAtBeginDict.values():

                # Get the matrices from opengl
                matView = glGetFloatv(GL_MODELVIEW_MATRIX)
                matProjection = glGetFloatv(GL_PROJECTION_MATRIX)

                if p.Semantic == 'World':
                        transp = transpose( self.matWorld )
                        self.effect.SetMatrix(p.Index,  transp, 4, 4)

                if p.Semantic == 'WorldIT':
                        inv = inverse(self.matWorld)
                        self.effect.SetMatrix(p.Index, inv, 4, 4)

                if p.Semantic == 'WorldViewProjection':
                        wv = matrixmultiply( self.matWorld,matView )
                        wvp = matrixmultiply(wv, matProjection )
                        transposewvp = transpose(wvp)
                        self.effect.SetMatrix(p.Index,  transposewvp, 4, 4)

                if p.Semantic == 'ViewIT':
                        inv = inverse(matView)
                        self.effect.SetMatrix(p.Index,  inv, 4, 4)

        return self.effect.Begin(0)

def Pass(self, p):
        self.effect.Pass(p)

def End(self):
        self.effect.End()
```

Appendix B: PyCgFXEffect.h

```
#include  <stdio.h>
#include "ICgFXEffect.h"
#include "Python.h"
#include "Numeric/arrayobject.h"



#define CgFXPT_UNKNOWN 0
#define CgFXPT_BOOL 1
#define CgFXPT_INT 2
#define CgFXPT_FLOAT 3
#define CgFXPT_DWORD 4
#define CgFXPT_STRING 5

#define CgFXPT_TEXTURE 6
#define CgFXPT_CUBETEXTURE 7
#define CgFXPT_VOLUMETEXTURE 8
#define CgFXPT_VERTEXSHADER 9
#define CgFXPT_PIXELSHADER 10

// Complex type
#define CgFXPT_STRUCT 11

// force 32-bit size enum
#define CgFXPT_FORCE_DWORD      = 0x7fffffff


class PyCgFXEFFECT_DESC
{
public:
        UINT Parameters;      // Number of parameters
        UINT Techniques;      // Number of techniques
        UINT Functions;       // Number of functions
};

class PyCgFXPARAMETER_DESC
{
public:
        LPCSTR Name;                                    // Parameter name.
        UINT Index;                                     // Parameter index

        // Usage
        LPCSTR Semantic;                                // Semantic meaning

        // Type
        UINT Type;                                      // Parameter type
        UINT Dimension[CGFX_MAX_DIMENSIONS];            // Elements in array
        UINT Bytes;                                     // Total size in bytes

        // Annotations
        UINT Annotations;                               // Number of annotations.
};

class PyCgFXANNOTATION_DESC
{
public:
   LPCSTR Name;                                         // Annotation name
   UINT Index;                                          // Annotation index (cast to LPCSTR)
   LPCSTR Value;                                        // Ska egentligen vara LPCVOID

   // Type
   UINT Type;                                           // Annotation type
   UINT Dimension[CGFX_MAX_DIMENSIONS];                 // Elements in array
   UINT Bytes;                                          // Total size in bytes

};

class PyCgFXTECHNIQUE_DESC
{
public:
   LPCSTR Name;                                         // Technique name
   UINT Index;                                          // Technique index (cast to LPCSTR)

   UINT Properties;                                     // Number of properties
   UINT Passes;                                         // Number of passes

};

class PyCgFXPASS_DESC
{
public:
   LPCSTR Name;                                         // Pass name
   UINT Index;                                          // Pass index (cast to LPCSTR)

};
```

```
class PyICgFXEffect
{
public:
        PyICgFXEffect();
        ~PyICgFXEffect(void);

        const char* PyCgFXCreateEffectFromFileA(
                char*           pSrcFile,
                DWORD           Flags);

        const char* PyCgFXCreateEffect(
                char*           pSrcData,
                DWORD           Flags);

        const char* PyCgFXCreateEffectCompiler(
                char*           pSrcData,
                DWORD           Flags);

        const char* PyCgFXCreateEffectCompilerFromFileA(
                char*           pSrcFile,
                DWORD           Flags);

        const char* CompileEffect(DWORD Flags);

        void            PyCgFXSetDevice(const char* pDeviceName, DWORD pDevice);
        void            PyCgFXFreeDevice(const char* pDeviceName, DWORD pDevice);
        const char*     PyCgFXGetErrors();
        void            PyCgFXRelease();

        void            SetTechnique(int pTechnique);
        char*           GetTechnique();

        // Validate current technique:
        bool            Validate();

        // Returns the number of passes in pPasses:
        int             Begin(DWORD Flags);
        void            Pass(int passNum);
        void            End();

        // Descriptions
        /* Return description of effect */
        PyCgFXEFFECT_DESC GetDesc();
        /* Return description of the named/indexed parameter */
        PyCgFXPARAMETER_DESC GetParameterDesc(int pParameter);
        /* Return description of the named/indexed annotation */
        PyCgFXANNOTATION_DESC GetAnnotationDesc(int pParameter, int pAnnotation);
        /* Return description of the named/indexed technique */
        PyCgFXTECHNIQUE_DESC GetTechniqueDesc(int pTechnique);
        /* Return description of pass for given technique */
        PyCgFXPASS_DESC GetPassDesc(int pTechnique, int pPass);

        ULONG AddRef();
        ULONG Release();

        // Get/Set Parameters

        void  SetFloat(int pName, float f);
        float GetFloat(int pName);

        void  SetVector(UINT pName, const float* pVector, UINT vecSize);

        void  SetMatrix(UINT pName, const float* pMatrix, UINT nRows, UINT nCols);

        void  SetDword(int pName, DWORD dw);
        DWORD GetDword(int pName);

        void  SetBoolValue(int pName, bool bvalue);
        bool  GetBoolValue(int pName);
        void  SetString(int pName, LPCSTR pString);
        char* GetString(int pName);
        void  SetTexture(int pName, DWORD textureHandle);
        DWORD GetTexture(int pName);

        void  SetVertexShader(int pName, DWORD vsHandle);
        DWORD GetVertexShader(int pName);

        void  SetPixelShader(int pName, DWORD psHandle);
        DWORD GetPixelShader(int pName);

        void OnLostDevice();
        void OnResetDevice();

private:
        ICgFXEffect*            m_pICgFXEffect;
        ICgFXEffectCompiler*    m_pICgFXEffectCompiler;
};
```

Appendix C: ICgFXEffect.h

```
/*********************************************************************NVMH3****
File:  ICgFXEffect.h

Copyright NVIDIA Corporation 2002
TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, THIS SOFTWARE IS PROVIDED
*AS IS* AND NVIDIA AND ITS SUPPLIERS DISCLAIM ALL WARRANTIES, EITHER EXPRESS
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS FOR A PARTICULAR PURPOSE.  IN NO EVENT SHALL NVIDIA OR ITS SUPPLIERS
BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES
WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS,
BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS)
ARISING OUT OF THE USE OF OR INABILITY TO USE THIS SOFTWARE, EVEN IF NVIDIA HAS
BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.


******************************************************************************/
#ifndef __ICGFX_EFFECT_H
#define __ICGFX_EFFECT_H

#include "cgfx_stddefs.h"
#define CGFX_MAX_DIMENSIONS 4

////////////////////////////////////////////////////////////////////////
// Types ///////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

typedef enum _CgFXPARAMETERTYPE
{
        CgFXPT_UNKNOWN                                                  = 0,
        CgFXPT_BOOL,
        CgFXPT_INT,
        CgFXPT_FLOAT,
        CgFXPT_DWORD,
         CgFXPT_STRING,

         // Ref counted objects; Can be accessed as objects or as strings
         CgFXPT_TEXTURE,
         CgFXPT_CUBETEXTURE,
         CgFXPT_VOLUMETEXTURE,
         CgFXPT_VERTEXSHADER,
          CgFXPT_PIXELSHADER,

         // Complex type
         CgFXPT_STRUCT,

         // force 32-bit size enum
        CgFXPT_FORCE_DWORD      = 0x7fffffff

} CgFXPARAMETERTYPE;

typedef struct _CgFXEffect_DESC
{
        UINT Parameters;     // Number of parameters
        UINT Techniques;     // Number of techniques
        UINT Functions;      // Number of functions
} CgFXEFFECT_DESC;

typedef struct _CgFXPARAMETER_DESC
{
        LPCSTR Name;                                    // Parameter name.
        LPCSTR Index;                                   // Parameter index (cast to LPCSTR)

        // Usage
        LPCSTR Semantic;                                // Semantic meaning

        // Type
        UINT Type;                                      // Parameter type
        UINT Dimension[CGFX_MAX_DIMENSIONS];            // Elements in array
        UINT Bytes;                                     // Total size in bytes

        // Annotations
        UINT Annotations;                               // Number of annotations.
} CgFXPARAMETER_DESC;

typedef struct _CgFXANNOTATION_DESC
{
        LPCSTR Name;                                     // Annotation name
        LPCSTR Index;                                    // Annotation index (cast to LPCSTR)
        LPCVOID Value;                                  // Annotation value (cast to LPCVOID)

          // Type
        CgFXPARAMETERTYPE Type;                          // Annotation type
        UINT Dimension[CGFX_MAX_DIMENSIONS];            // Elements in array
        UINT Bytes;                                     // Total size in bytes

} CgFXANNOTATION_DESC;

typedef struct _CgFXTECHNIQUE_DESC
{
```

```
    LPCSTR Name;                        // Technique name
    LPCSTR Index;                       // Technique index (cast to LPCSTR)

    UINT Properties;                    // Number of properties
    UINT Passes;                        // Number of passes

} CgFXTECHNIQUE_DESC;

typedef struct _CgFXPASS_DESC
{
        LPCSTR Name;                    // Pass name
        LPCSTR Index;                   // Pass index (cast to LPCSTR)

} CgFXPASS_DESC;

typedef enum _CgFXMode
{
        CgFX_Unknown,
        CgFX_OpenGL,
        CgFX_Direct3D8,
        CgFX_Direct3D9
} CgFXMode;

//////////////////////////////////////////////////////////////////////////
// Base Effect ///////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
// base interface. extended by Effect and EffectCompiler.
//////////////////////////////////////////////////////////////////////////
struct ICgFXBaseEffect
{
        virtual ULONG AddRef() = 0;
        virtual ULONG Release() = 0;

         // Descriptions
        /* Return description of effect */
        virtual HRESULT GetDesc(CgFXEFFECT_DESC* pDesc) = 0;
        /* Return description of the named/indexed parameter */
        virtual HRESULT GetParameterDesc(LPCSTR pParameter, CgFXPARAMETER_DESC* pDesc) = 0;
        /* Return description of the named/indexed annotation */
        virtual HRESULT GetAnnotationDesc(LPCSTR pParameter, LPCSTR pAnnotation, CgFXANNOTATION_DESC* pDesc) = 0;
        /* Return description of the named/indexed technique */
        virtual HRESULT GetTechniqueDesc(LPCSTR pTechnique, CgFXTECHNIQUE_DESC* pDesc) = 0;
        /* Return description of pass for given technique */
        virtual HRESULT GetPassDesc(LPCSTR pTechnique, LPCSTR pPass, CgFXPASS_DESC* pDesc) = 0;

        // Get/Set Parameter
        virtual HRESULT SetValue(LPCSTR pName, LPCVOID pData, UINT Bytes) = 0;
        virtual HRESULT GetValue(LPCSTR pName, LPVOID pData, UINT Bytes) = 0;
        /*
        virtual HRESULT SetValueTranspose(LPCSTR pName, LPCVOID pData, UINT Bytes) = 0;
        virtual HRESULT GetValueTranspose(LPCSTR pName, LPVOID pData, UINT Bytes) = 0;
        */

        virtual HRESULT SetFloat(LPCSTR pName, FLOAT f) = 0;
        virtual HRESULT GetFloat(LPCSTR pName, FLOAT* f) = 0;
        virtual HRESULT SetVector(LPCSTR pName, const float* pVector, UINT vecSize) = 0;
        virtual HRESULT GetVector(LPCSTR pName, float* pVector, UINT* vecSize) = 0;
        virtual HRESULT SetMatrix(LPCSTR pName, const float* pMatrix, UINT nRows, UINT nCols) = 0;
        virtual HRESULT GetMatrix(LPCSTR pName, float* pMatrix, UINT* nRows, UINT* nCols) = 0;
        //virtual HRESULT SetMatrixTranspose(LPCSTR pName, const float* pMatrix, UINT nRows, UINT nCols) = 0;
        //virtual HRESULT GetMatrixTranspose(LPCSTR pName, float* pMatrix, UINT* nRows, UINT* nCols) = 0;

        virtual HRESULT SetDword(LPCSTR pName, DWORD dw) = 0;
        virtual HRESULT GetDword(LPCSTR pName, DWORD* dw) = 0;

        virtual HRESULT SetBoolValue(LPCSTR pName, bool bvalue) = 0;
        virtual HRESULT GetBoolValue(LPCSTR pName, bool* bvalue) = 0;


        virtual HRESULT SetString(LPCSTR pName, LPCSTR pString) = 0;
        virtual HRESULT GetString(LPCSTR pName, LPCSTR* ppString) = 0;
        //virtual HRESULT SetTexture1D(LPCSTR pName, DWORD textureHandle) = 0;
        //virtual HRESULT GetTexture1D(LPCSTR pName, DWORD* textureHandle) = 0;
        virtual HRESULT SetTexture(LPCSTR pName, DWORD textureHandle) =0;
        virtual HRESULT GetTexture(LPCSTR pName, DWORD* textureHandle) = 0;
        /*
        virtual HRESULT SetCubeTexture(LPCSTR pName, DWORD textureHandle) =0;
        virtual HRESULT GetCubeTexture(LPCSTR pName, DWORD* textureHandle) = 0;
        virtual HRESULT SetVolumeTexture(LPCSTR pName, DWORD textureHandle) = 0;
        virtual HRESULT GetVolumeTexture(LPCSTR pName, DWORD* textureHandle) = 0;
        */

        virtual HRESULT SetVertexShader(LPCSTR pName, DWORD vsHandle) = 0;
        virtual HRESULT GetVertexShader(LPCSTR pName, DWORD* vsHandle) = 0;

        virtual HRESULT SetPixelShader(LPCSTR pName, DWORD psHandle) = 0;
        virtual HRESULT GetPixelShader(LPCSTR pName, DWORD* psHandle) = 0;
};

struct ICgFXEffect : public ICgFXBaseEffect
{
        // Set/get current technique:
        virtual HRESULT SetTechnique(LPCSTR pTechnique) = 0;
        virtual HRESULT GetTechnique(LPCSTR* ppTechnique) = 0;
        // Validate current technique:
        virtual HRESULT Validate() = 0;
```

```
        // Returns the number of passes in pPasses:
        virtual HRESULT Begin(UINT* pPasses, DWORD Flags) = 0;
        virtual HRESULT Pass(UINT passNum) = 0;
        virtual HRESULT End() = 0;

    virtual HRESULT CloneEffect(ICgFXEffect** ppNewEffect) = 0;
        /*
        virtual HRESULT FindNextValidTechnique(LPCSTR pTechnique, CgFXTECHNIQUE_DESC* pDesc) = 0;
        */

        virtual HRESULT GetDevice(LPVOID* ppDevice) = 0;
        virtual HRESULT OnLostDevice() = 0;
        virtual HRESULT OnResetDevice() = 0;

};

struct ICgFXEffectCompiler : public ICgFXBaseEffect
{
    // Compilation
    virtual HRESULT CompileEffect(DWORD Flags, ICgFXEffect** ppEffect, const char** ppCompilationErrors) = 0;
};

HRESULT CgFXCreateEffect(
    LPCSTR              pSrcData,
    DWORD               Flags,
    ICgFXEffect**       ppEffect,
    const char**         ppCompilationErrors);

HRESULT CgFXCreateEffectFromFileA(
    LPCSTR              pSrcFile,
    DWORD               Flags,
    ICgFXEffect**       ppEffect,
    const char**         ppCompilationErrors);

HRESULT CgFXCreateEffectCompiler(
    LPCSTR              pSrcData,
    DWORD               Flags,
    IcgFXEffectCompiler**   ppEffectCompiler,
    const char**        ppCompilationErrors);

HRESULT CgFXCreateEffectCompilerFromFileA(
    LPCSTR              pSrcFile,
    DWORD               Flags,
    IcgFXEffectCompiler**   ppEffectCompiler,
    const char**        ppCompilationErrors);

HRESULT CgFXSetDevice(const char* pDeviceName,LPVOID pDevice);
HRESULT CgFXFreeDevice(const char* pDeviceName, LPVOID pDevice);
HRESULT CgFXGetErrors(const char** ppErrors);
HRESULT CgFXRelease();
#endif
```