Real-Time Shading Languages

Dalibor Mitrović*

Institute of Computer Graphics and Algorithms
Vienna University of Technology

Abstract

Implementation of real-time shading languages in consumer level hardware started in late 90s. The first graphics processor to support procedural shading was NVIDIA's TNT. It offered register combiners a simple but powerful mechanism, that enabled real time shading in hardware. The shading language used at the beginning were very assembly-like but with the new generations of graphics processors they became more flexible and were replaced by high level shading languages. State of the art is NVIDIA's Cg, as the name implies it is a shading language with a C-like syntax specially designed for the creation of shaders.

Keywords: Real-Time Shading Languages, RTSL, HLSL, Cg

1 Introduction

In this paper we want to discuss Real-Time Shading Languages. As an introduction we will start with a definition of what procedural shading means. Discussion of Hardware that first supported real-time procedural shading starting with NVIDIA’s GeForce 256 is to follow. Furthermore we will compare NVIDIA’s chipsets with ATI’s and give a general overview of the development of the so called Graphics Processing Units (GPUs) focusing on their capabilities in the sector of real-time shading language support. What is procedural shading and how does it work? In this work we will follow Olano’s and Lastra’s definition [8]:

In procedural shading, a user (someone other than a system designer) creates a short procedure, called a shader, to determine the final color for each point on a surface.

The shader deals with all aspects of the pixel’s final appearance. Color and the interaction of light with the surface are managed by the shader. Shaders take input parameters like the surface’s normals, light directions and colors. The simple but powerful Register Combiners [3] were programmable in a proprietary assembly like language. They accepted the RGB and alpha values of textures and were able to manipulate two of them in each step. This was done on a per-pixel basis. Possible calculations included

signed addition, signed multiplication, and dot products. There also is a mux computation that can be used to conditionally select the output. For a detailed explanation see NVIDIA’s documents on the register combiner technology [3]. The operations could be performed on a wide variety of operands including texture fragment colors, polygon colors, and even results of the calculations performed by other combiners. Figure 1 gives an example of what could be achieved with the use of NVIDIA’s Register Combiners. Particularly Figure 1 shows an sphere with a simple geometry (few polygons) that looks far better tessellated than it is in reality. This effect is achieved through the use of bump mapping, that is entirely computed with the register combiners.

The next step in development was the GeForce 2 which was the first GPU to support per-pixel lighting. It introduced primitives called texture elements (texels) as components of a lighting equation. This approach required a new texture the so called normal map, which contains per-pixel normals that, together with texel information, were used to calculate the intensity and direction of light.

Figure 1: This image shows the result of bump mapping with the use of register combiners. With the bump mapping the sphere looks smoother than it is. This can be noticed at the edges to the background.
for each rendered pixel [4]. This technology was called NVIDIA Shading Rasterizer or short NSR and employed the register combiner at the low level which means it was no architectural improvement but marketing.

A real architectural step forward was the GeForce 3 (short GF3) with its Vertex Shader. The Vertex Shader was able to execute programs with a length of up to 128 instructions. With this instructions 16 data entries per-vertex could be manipulated. The instructions itself are very simple, but therefore also easily understandable. Loops, jumps or conditional branches were not supported. See the following DirectX 8 vertex shader for illustration:

```
vs.1.1
mov oT0, v7
mov oT1, v8
dp4 oPos.x, c1, v0
dp4 oPos.y, c2, v0
dp4 oPos.z, c3, v0
dp4 oPos.w, c4, v0
mul oD0, c0.x, v5
```

ATI’s Radeon was introduced to compete with the GeForce 2. It offered a feature named The Pixel Tapestry which in fact is the same as NSR from a technological point of view. The Radeon 8500 was released after the GeForce 3 and offered a Vertex Shader much like the GF3. ATI’s pixel shader was an improved duplicate of the GF3 feature. In contrast to the GF3 that could execute 12 instructions the Radeon supported up to 22 instructions in a pixel shader program. The next chronological step was the introduction of the GeForce 4 (GF 4) and ATI’s Radeon 9700. Note that this was only a chronological step, the GF4 was an accelerated GF3 without any architectural changes whereas the Radeon 9700 offered not only twice the amount of vertex shaders (in comparison to the GF4, but the four units were much more sophisticated per se. The Radeon 9700 supports vertex programs up to 256 instructions, though more instructions can be executed due to looping, subroutines etc. [7]. But the reason why this GPU has to be considered a new generation in graphics processors, is that Radeon 9700’s pixel shader operates on floating point color values. To see what difference floating point precision makes have a look at Figure 2 [12]. A pixel shader can have up to 64 ALU instructions and 32 texture instructions [7]. NVIDIA’s answer is the GeForceFX (GFX). Its vertex shader handles 256 instructions in saved programs, the maximum number of instructions that can be executed per shader is 65536. The pixel shader allows 1024 instructions per rendering pass [12]. Besides that NVIDIA offers a profile for CG that fully supports the GFX’s capabilities.

The next generation of GPU’s was introduced with the Radeon 9800. It is the first GPU to theoretically support infinite shader programs, in practice the GPU does not offer enough performance for infinite shader programs [12]. The technique that enables this theoretically infinite shaders is called F–Buffer and was presented by William R. Mark and Kekoa Proudfoot in their paper The F–Buffer: A Rasterization–Order FIFO Buffer for

---

**Figure 2:** This Figure illustrates the effect of floating point precision color. On the left side one can see the same scene in 32-bit color and on the right side one can see the increase in color and brightness dynamics due to the usage of 128/bit floating point color precision.

---

### 2 Related Work

The presentation of all important related work that lead to Real-Time Shading Languages is far beyond the scope of this paper. That is why we will concentrate on two systems that seem important to us. RenderMan because most readers will know at least one feature film entirely produced using RenderMan (Toy Story). PixelFlow because it was the first system whose shading language worked in real-time.

#### 2.1 RenderMan

Talking about RenderMan requires an explanation of what RenderMan is.

The RenderMan Interface is a standard interface between modeling programs and rendering programs capable of producing photorealistic quality images [10].

Because of the interface’s limited nature an extension mechanism is required. This mechanism is offered by The RenderMan Shading Language. According to Pixar:

The RenderMan Shading Language is a programming language for extending the predefined functionality of the RenderMan Interface [10].

The RenderMan Shading Language is a strongly typed C-like language that supports the following types:

- Floats, used for all scalar including integer calculations,
Colors, implemented as an abstract data type,

Points, Vectors, and Normals, stored as triplets \((x,y,z)\) of floats,

Transformation Matrices, required to transform points and vectors,

Strings, used to name external objects,

Arrays, 1D all basic types supported.

A library of mathematical functions is provided with the shading language. Besides the standard functions like \(\sin(), \cos(), \sqrt()\) it includes functions for interpolation and computing derivatives. Not only math functions are built-in, there are Geometric, Color, Matrix and several other libraries. (See the RenderMan interface specification [10] for the full list.) An example shader is provided below, it implements a simple turbulence procedural texture:

```plaintext
surface
turbulence (float Kd=.8, Ka=.2){
  float a, scale, sum;
  float IdotN;
  point M;
  M = transform(marble,P);
  scale = 1;
  sum = 0;
  a = sqrt(area(M));
  while( a < scale ) {
    sum += scale * float noise(M/scale);
    scale *= 0.5;
  }
  Oi= sum;
  Ci = Cs * Oi * (Ka + Kd * I.N * I.N / (I.I * N.N) );
}
```

A major problem with RenderMan Shading Language is that there is no implementation of the RI (RenderMan Interface) to support real-time shading. One has to note that such a real-time implementation is possible for the future, considering the increase in processing power over the years.

### 2.2 PixelFlow

The PixelFlow System is a SIMD graphics multicomputer that supports procedural shading using a shading language named pfname similar to Pixar’s RenderMan shading language [8]. This system is the first one that used a shading language and was able to render at real-time. In fact PixelFlow could render images at 30 frames per second. This is not much now, but in 1998 it was a significant improvement in photorealistic rendering. Although pfname is very similar to the RenderMan Shading Language there are differences. According to Olano and Lastra [8]:

These differences are

1. the introduction of a fixed-point data type,
2. the use of arrays for points and vectors,
3. the introduction of transformation attributes,
4. the explicit listing of all shader parameters, and
5. the ability to link external functions.

The first two changes were introduced in order to increase speed and efficiency. The fixed-point format used in pfname allowed higher precision than the floating-point (namely the 32-bit IEEE single precision) format which offered a greater range. An example for the need of the fixed-point format is given with the Mandelbrot fractal shader, that needs high precision but only in the range of \([-2,2]\) (Figure 3). The fifth difference is due to the fact that pfname is compiled and not interpreted. A pfname shader is compiled in a two step process. The compiler produces C++ source code which is compiled by a C++ compiler to produce the object file for the shader. In contrast to most RenderMan implementations, pfname supports calling C++ functions in the shading language as well as calling shader functions in C++, because function definitions and calls in pfname are linked directly to C++ function definitions and calls which is only true for functions using the types the shading language supports.

### 3 High Level Shading Languages

To know why High Level Shading Languages (HLSL) are considered to be important we think it is necessary to have a look at their advantages. The list below is a list of advantages we have identified. Note that some of these are not fully independent one from an other.

1. Independence of the underlying graphics architecture,
2. the implementation details are hidden,
3. faster development of shaders,
4. the languages are easier to learn,
5. the possibility of producing the same effects as before with less effort, and

6. cost decrease.

From a scientific point of view independence of the underlying graphics architecture and the hidden implementation details are the most important advantages. But the others should not be underestimated, because of their importance for the future development of high level shading languages. Cost decrease gives impulses for the use of HLSL and with use the need for improvement and further development arises.

From a consumer’s point of view point 5 is most important, because it will increase the number and quality of visual effects to be found in graphics software.

3.1 The Stanford Shading System

The Stanford Shading System is a procedural shading system consisting of three parts. First, a new programmable pipeline abstraction, that is based on previous work. It combines and extends elements from systems published before. Second, a new shading language (short RTSL, not that we will use RTSL as an abbreviation for the Stanford Shading System). Third, a retargetable compiler back end. Due to the use of a set of interchangeable modules the compiler maps the authors’ abstractions to various different graphics accelerators, including those with vertex and fragment programmability.

System Overview. Figure 4 shows a block diagram of the system. The authors describe the principal components of the system as follows [11]:

- **Shading language and compiler front end.** Shaders in RTSL are used to describe shading computations. A compiler front end maps the shading language to an intermediate pipeline program representation.

- **Programmable pipeline abstraction.** An intermediate abstraction layer provides a generic interface to hardware programmability to hide hardware details and to simplify compiler front ends. It consists of a computational model (the programmable pipeline) and a means for specifying computations (pipeline programs). Pipeline programs are divided into pieces by computation frequency.

- **Retargetable compiler back end.** A modular, retargetable compiler back end maps pipeline programs to shader object code. There are back-end modules for different stages and for different hardware.

![Figure 4: Block diagram of the Stanford Shading System.](image)

The systems components:

- **Shader object code.** Compiled shaders are used to configure hardware during rendering. Shader object code separates the compile-time and render-time halves of the system.

- **Shader execution engine.** A shader execution engine controls the rendering of geometric primitives using the shader object code. The application may attach shader parameters to groups of primitives and to vertices. These parameters are processed to compute surface positions and surface colors.

- **Graphics hardware.** Shader execution modules rely on graphics hardware for most shading computations, although the host CPU may be used for some computations.

The system runs on top of OpenGL. It utilizes a sequential approach to rendering of objects. After a shader is written and compiled it is bound to one or more objects. In contrast to shaders, objects are rendered immediately after specification.

The programmable pipeline abstraction (see Figure 5) is considered to be the central element of the system. It was introduced to enable a simplified mapping of the shading language to hardware. It provides a computational model...
that describes what and how values are computed. Furthermore it defines parts of the language syntax.

The programmable pipeline renders objects’ positions and color intensities. The position is required for depth buffering. The abstraction contains two kinds of stages: programmable stages and fixed-function stages.

The programmable stages are associated with four computation frequencies, shown in Figure 6. Constant expressions are once evaluated at compile time. Primitive groups are defined as the geometry within an OpenGL Begin/End block; vertices are defined by the OpenGL Vertex command; and fragments are defined by the screen-space sampling grid. The computation frequencies are used to enable a tradeoff between costly high-precision floating-point computations at a coarse level of detail and many simple low-precision fixed-point computations at a fine level of detail. The default computation frequencies are inferred by a two rule set. The compiler is in no way flexible with respect to selecting computation frequencies. Users can always predict what frequency will be chosen.

Programmable stages are ordered by frequency starting with the least-frequent first. Each stage computes an output stream given an input stream of individual objects. These input objects can be application specific parameters and output streams of previous stages.

Fixed-function stages implement parts of the graphics pipeline that can not be programmed (e.g., stages that assemble vertices into primitives).

Programmable stages are driven by directed acyclic graphs (DAGs) that specify how to compute the stage’s output stream. The pipeline programs are not restricted in length, number of inputs, parameters and outputs. The programs are conceptually rendered in a single pass, but due to hardware restrictions large and complex shaders can be rendered using a multipass approach.

**Data types.** The system operates with ten data types listed below:

1. scalars to be float or [0,1]-clamped float,
2. 3-vectors containing float or [0,1]-clamped float,
3. 4-vectors containing float or [0,1]-clamped float,
4. 3 \times 3 floating point matrices,
5. 4 \times 4 floating point matrices,
6. booleans, and a special
7. texture reference type.

The texture reference type allows access to textures through the OpenGL texture naming mechanism. All of these data types are abstract data types, i.e., they have well-defined semantics but not necessarily a well-defined storage representation.

**The Operators.** The operators support standard transform, lighting and texturing operations. Among supported operations are: basic arithmetic; scalar, vector and matrix manipulation; comparisons and selection of two values based on a third boolean value. For a full list see the RTSL paper [11].

Use of textures is supported through to the OpenGL texture naming mechanism. This means a texture to be used in a shader program needs to be passed to the program as input from OpenGL.

Complex operations that cannot be expressed efficiently across a variety of hardware are supported through canned functions. Namely two canned functions are implemented (bumpspec and bumpdiff) to facilitate bump mapping.

Because not every hardware platform supports every operator at every computation frequency there is a table-driven restriction mechanism. The language compiler determines at run-time which operators are available thus it can provide conditional compilation directives to allow multiple code versions.

**Shading Language.** The shading language is loosely based on the RenderMan Shading Language. Though there are noteworthy differences. Features that are not currently supported by mainstream graphics hardware (like loops and conditionals) are omitted. As well as features the authors felt not essential for the exploration of the compilation and architectural issues they wanted to research. This category includes atmosphere, imaging, and transformation shaders and a complete library of built-in functions. Authors plan to add features over time.
What distinguishes this shading language from the previous work is, that is easily analyzed and optimized by a compiler. Analysis is important because it allows to infer several kinds of information that users would otherwise have to specify explicitly. Optimization is important because of performance issues. Shaders have to work as fast as possible in a real-time environment.

Read what the authors have to say on Analysis and Optimization [11]:

Four aspects of the language help make it easy to analyze and optimize:

- **Function inlining.** We explicitly inline all functions and delay the analysis and optimization of each inlined function until after the function has been inlined. This allows the compiler to specialize each function to its calling context.

- **Combined compilation of surfaces and lights.** We compile surfaces and lights together, and we delay analysis and optimization until after surfaces and lights have been combined. This allows us to perform analysis and optimization across surface and light shaders.

- **No data-dependent loops and branches.** The lack of support for data-dependent loops and branches in hardware means we do not support these features in RTSL. This considerably simplifies the analyses we must perform.

- **No random access to memory.** The lack of hardware support for random read/write access to memory likewise allows us to eliminate that feature from the shading language. In particular, this removes the possibility of pointer aliasing, again simplifying the analyses we must perform.

These properties make it possible to represent a complete shader program as a DAG. This allows very straightforward analysis and optimization.

**Retargetable Compiler Back End.** In order to support a wide range of hardware the compiler is implemented to be modular. This allows new hardware to be supported by providing a compiler module to map the pipeline abstraction to shader code that can be processed by this particular piece of hardware and a render-time module which is necessary for configuration and utilization during rendering.

Two techniques are used to support arbitrarily-complex computations. First, multipass methods are used if a single hardware pass is unable to execute the entire fragment portion of a pipeline program. Second, the host processor is used if available vertex-processing resources are insufficient.
Figure 8: The Cg compiler. The Cg compiler produces output based on a specified profile. Each profile represents an API (like shown above) or target GPU (not shown). Due to the modularity of the profile mechanism shaders can take advantage of innovations yet to come with new APIs or new hardware.

is compiled due to dependencies if the Cg Runtime is not used. Furthermore the Cg Runtime API facilitates management of input parameters.

Profiles. As profiles are an important concept of Cg we want to quote the authors:

Compilation of a Cg program, a top-level function, always occurs in the context of a compilation profile. The profile specifies whether certain optional language features are supported. These optional language features include certain control constructs and standard library functions. The compilation profile also defines the precision of the float, half, and fixed data types, and specifies whether the fixed and sampler* data types are fully or only partially supported. The choice of a compilation profile is made externally to the language, by using a compiler command-line switch, for example [2].

The profile restrictions are only valid for the top-level function that is being compiled and for the variables and functions that it references, no matter if they are referenced directly or indirectly. This way the Cg code can contain functions that are used in other profiles. e.g code written for a Radeon GPU can be placed in the same Cg file that is used for the GeForce. This enhances portability of code. Each profile has to come with an accurate specification of its characteristics and restrictions. The restrictions are limited by the core Cg specification which requires certain minimum requirements to be met by all profiles. Figure 8 illustrates how the Cg compiler works in combination with profiles.

Data Types. Cg supports six basic data types:

1. float 32-bit IEEE floating point number,
2. half 16-bit IEEE-like floating point number,
3. int 32-bit integer,
4. fixed 12-bit fixed-point number,
5. bool,
6. sampler* a texture handler.

The texture handler comes in six variants: sampler, sampler1D, sampler2D, sampler3D, samplerCUBE, and samplerRECT. The int type may be treated as float or be not usable depending on the used profile. Cg has built-in vector types that are based on the basic types. Syntax for declaration looks like this:

float4 vector1;
//4-component vector of floats
bool2 vector2;
//2-component vector of bools

Matrices are supported up to a size of four-by-four and are declared similar to vectors:

float4x4 matrix1;
float2x3 matrix2;

Beside the more basic types Cg also supports structures and arrays. Structures are supported the same way they are supported in C. These structures are needed for the data flow into and from the shader. One could say, that shaders written in Cg communicate with the programs they are called by, through the above mentioned structures. These have to be specified in the shader at implementation time. Arrays are declared just as in C. Arrays must be defined using array syntax because Cg does not support pointers. A proper array definition is shown below (example from [2]):

// Declare a function // that accepts an array // of five skinning matrices.
returnType foo(float4x4 mymatrix[5]) {/* ... */};

The most important difference between arrays in Cg and C is, that in Cg arrays are a first-class type. i.e., arrays are handled by value not by reference.

Operators. Cg supports all the standard C arithmetic operators (+, -, *, /) and allows their use on scalars as well as on vectors. Matrix operands are not supported. Cg includes the mul() operator which is used to multiply matrices and vectors. To be precise mul() is used to multiply matrices by vectors and matrices by matrices. For a detailed overview see the Cg users manual [2] (page 22).
In contrast to C the boolean operator logical AND, logical OR and logical negation consume and produce booleans. Unlike C AND and OR cannot be used to shorten evaluation because both sides are evaluated. The boolean operators can be applied to vectors where they are applied in an elementwise fashion producing a vector of booleans the size of the input vectors.

The so-called swizzle operator allows permutation of a vector’s components to form a new vector. The new vector is not restricted to be the same size as the input vector, it can be smaller (elements from the input vector can be omitted) or be bigger (elements from the input vector can be repeated). One can even use the swizzle operator to form new vectors out of scalars. See the example below, which was taken from the Cg users manual [2]:

\[ b.xxx \text{ yields } \text{float3}(b, b, b) \]

Cg has a write mask operator that can be used to manipulate selectively a vector’s components.

Control Flow. Cg comes with the following control structure:

1. Function calls and the return statement
2. if/else
3. while
4. for

The conditional expressions of this structures are to be of type bool.

In profiles other than vs2_8 and pv30, while and for are only supported if the compiler can determine the iteration count at compile time. The mentioned profiles vs2_8 and pv30 support branch instructions, that is why they offer full support for loops. return may only be the last statement in a function.

Recursion is not allowed.

Cg does not use switch, case or default. But these keywords are reserved for future use, as are all other C and C++ keywords, although it seems unlikely at the moment that keywords dealing with file input/output or memory allocation will be used in the near future.

A Cg Shader. To illustrate Cg’s possibilities have a look at the sample below. This shader implements phong lighting [9]. The structure appin is used to define the inputs of the application, vertout is used to define the shader’s output. It is provided by NVIDIA in the Cg SDK. See Figure 9 for this shader’s output.

```c
// define inputs from application
struct appin
{
    float4 Position : POSITION;
    float4 Normal : NORMAL;
};

// define outputs from vertex shader
struct vertout
{
    float4 HPosition : POSITION;
    float4 Color0 : COLOR0;
};

vertout main(
    appin IN,
    uniform float4x4 ModelViewProj,
    uniform float4x4 ModelViewIT,
    uniform float4 LightVec)
{
    vertout OUT;
    // transform vertex position
    // into homogeneous clip-space
    OUT.HPosition = mul(ModelViewProj, IN.Position);
    // transform normal
    // from model-space to view-space
    float3 normalVec = normalize(mul(ModelViewIT, IN.Normal).xyz);
    // store normalized light vector
    float3 lightVec = normalize(LightVec.xyz);
    // calculate half angle vector
    float3 eyeVec = float3(0.0, 0.0, 1.0);
    float3 halfVec = normalize(lightVec + eyeVec);
    // calculate diffuse component
    float diffuse = dot(normalVec, lightVec);
    // calculate specular component
    float specular = dot(normalVec, halfVec);
    // Use the lit function to compute
    // lighting vector from diffuse and
    // specular values
    float4 lighting = lit(diffuse, specular, 32);
    // blue diffuse material
    float3 diffuseMaterial = float3(0.0, 0.0, 1.0);
    // white specular material
    float3 specularMaterial = float3(1.0, 1.0, 1.0);
    // combine diffuse and specular
    // contributions and output
    // final vertex color
    OUT.Color0.rgb = lighting.y * diffuseMaterial + lighting.z * specularMaterial;
    OUT.Color0.a = 1.0;
    return OUT;
}
```

3.3 Differences between the Stanford approach and Cg

In our opinion the most important difference between the Stanford Shading System (RTSL) and Cg is the lack of control structures in the Stanford Shading System. This difference is based on the fact that the RTRTSLas invented prior to Cg. At the time the RTSL was published there was
no hardware support for conditionals, therefore the authors decided to omit this feature.
The second important difference is that Cg shaders can be compiled at runtime or before whereas shaders written in RTSL are compiled on the fly before they are passed to the hardware.
An other difference is that Cg was developed by the NVIDIA Corporation whereas RTSL is an academic project. We think this is the reason why RTSL works on top of OpenGL (an open standard) and why Cg supports Microsoft’s proprietary DirectX API as well as OpenGL.
Two more, but less important differences we have identified are: First, there are slight differences in the supported data types. Second, Stanford Shading System shaders fall back to the use of the host CPU if the GPU does not offer enough computational power for certain operations. Cg shaders do not support this, in fact the profiles effectively prevent such situations.
In general one can say, that the Stanford Shading System and Cg have more in common than it might seem at the first glance. Both systems share various concepts like the modular compiling mechanism, backward compatibility through restriction mechanisms, support for vector and matrix data types, etc. This is not surprising since several concepts are necessary for a high level shading language system to be usable.

3.4 Other Shading Languages

There are two more high level shading languages we want to mention in this work. First is the OpenGL Shading Language also known as glslang, second is Microsoft’s DirectX 9.0 HLSL.
The OpenGL Shading Language is actually two closely related languages, based on ANSI C. One language is used for vertex manipulation the other is used for pixel computations. glslang is built into OpenGL instead of being build on top of it like the Stanford Shading System. For further information on glslang see the OpenGL shading language specification draft [5].

4 Applications

The four major application areas of real time shading languages are

1. Scientific Visualization
2. CAD
3. Feature Films
4. Games

In scientific visualization real time shading is becoming an important technique. It offers for the first time the possibility of real time interaction. The simplest use of programmable shading is to change the appearance from standard Gouraud shading available in visualization software to a shader that will provide normal interpolation for curvature information (See Figure 10)[1]. Simulations can be visualized and manipulated at the same time which enables faster progress and better results. Almost the same arguments apply for the use in CAD environments. Costly prototyping can be reduced through use of high quality visualization.
In entertainment there are two main areas of application for real time shading languages. First, the production of feature films. Second, computer games. Feature films heavily rely on offline rendering at the moment. Modeling and animation are performed first and after that the result is rendered. This process is time consuming. Since time is money the feature film industry will appreciate faster production. With real time shading animators can actually see immediately how a scene will look like in the film. This way they can fix errors faster without the need for offline rendering. Real time shading offers game developers the chance to incorporate new never seen before visual effects into their games. This will result in games with a much higher level of detail and a new level of realism.
5 Summary

Real time shading languages started with the PixelFlow System. With the progress in graphics hardware came along the development of shading languages. First consumer level shading languages where assembly code specific for the hardware platform. The complexity and diversity of the so called GPUs lead to the development of high level shading languages like the Stanford Shading System’s shading language and Cg.

At present one can state that high level shading languages will bring forward the realism in computer games as well as they will boost scientific visualization and CAD. It is not possible to foresee if Cg will become a quasi standard or if an other shading language, e.g., glslang will become the mostly used language. Nevertheless it is sure that high level shading language use and development have just started and that they will become a key technology.

References


[4] The NVIDIA Corporation. Per-Pixel Lighting and Bump Mapping with the NVIDIA Shading Rasterizer


[10] Pixar. The RenderMan Interface Specification Version 3.2
