Lecture 15:
The Design of Shading Languages

Visual Computing Systems
Stanford CS348K, Spring 2020
3D graphics pipeline so far

So far in this section of the course: our focus has been on non-programmable parts of the graphics pipeline
- Geometry processing operations
- Visibility (coverage, occlusion)
- Texturing

I’ve said very little about materials, lights, etc.

And hardly mentioned programmable GPUs
Many different materials in the world
Materials: diffuse
Materials: plastic
Materials: red semi-gloss paint
Materials: mirror
Materials: gold
Materials

[Image credit: Jakob et al. 2014]
More complex materials

Fresnel reflection: surface reflectance is a function of viewing angle
(notice higher reflectance near grazing angles)

Anisotropic reflection: reflectance depends on azimuthal angle
(e.g., oriented microfacets in brushed steel)
Subsurface scattering materials

- Account for scattering inside surface
- Light exits surface from different location it enters
  - Very important to appearance of translucent materials (e.g., skin, foliage, marble)

[Wann Jensen et al. 2001]
More materials

Images from Matusik et al. SIGGRAPH 2003

Tabulated BRDFs
The rendering equation

An equation that describes how much light reflects off a surface in a given direction, as a function of all the light arriving at the surface from all directions.
The rendering equation *

\[ i(x, x') = v(x, x')[l(x, x') + \int r(x, x', x'')i(x', x'')\,dx''] \]

- \( i(x, x') \) = Radiance (light energy along a ray) from point \( x' \) in direction of point \( x \)
- \( v(x, x') \) = Binary visibility function (1 if ray from \( x' \) reaches \( x \), 0 otherwise)
- \( l(x, x') \) = Radiance emitted from \( x' \) in direction of \( x \) (if \( x' \) is an emitter)
- \( r(x, x', x'') \) = BRDF: fraction of energy arriving at \( x' \) from \( x'' \) that is reflected in direction of \( x \)

* Note: using notation from Hanrahan 90 (to match reading)
The rendering equation (simplified)

- All light sources are point light sources (light $i$ emits from point $x_{li}$)
- Assume lights emit equally in all directions: radiance from light $i$: $i(x', x_{li}) = L_i$
- Direct illumination only: illumination of $x'$ comes directly from light sources

$$i(x, x') = \sum_{i=0,1,2} L_i v(x', x_{li}) r(x, x', x_{li})$$
Categories of reflection functions: $r(x,x',x'')$

- **Ideal specular**
  - Perfect mirror
- **Ideal diffuse**
  - Uniform reflection in all directions
- **Glossy specular**
  - Majority of light distributed in reflection direction
- **Retro-reflective**
  - Reflects light back toward source

Diagrams illustrate how incoming light energy from given direction is reflected in various directions.

[Slide credit: Stanford 348b / Pat Hanrahan]
Many different types of lights in the world
Types of lights

- **Attenuated omnidirectional point light**
  (emits equally in all directions, intensity falls off with distance: $1/R^2$ falloff)

- **Spot light**
  (does not emit equally in all directions)
More sophisticated lights

- Environment light
  (not a point light source: defines incoming light from all directions)

Environment Map
(Grace cathedral)

Rendering using environment map
(pool balls have varying material properties)
[Ramamoorthi et al. 2001]
Environment map

Image credit: USC High-Resolution Light Probe Image Gallery
Early graphics APIs: parameterized materials and lighting (prior to programmable shading)

- `glLight(light_id, parameter_id, parameter_value)`
  - 10 parameters (e.g., ambient/diffuse/specular color, position, direction, attenuation coefficient)

- `glMaterial(face, parameter_id, parameter_value)`
  - Face specifies front or back facing geometry
  - Parameter examples (ambient/diffuse/specular reflectance, shininess)
  - Material value could be modulated by texture data

Problem: can’t represent all the material and light types in the world with just a few parameters!
Precursor to shading languages: shade trees

[Cook 84]

Albedo texture: (multiple textures used to define surface albedo at each point)

Diffuse reflectance (N dot L)

Specular reflectance

Material: specular reflection coeff

Note: shade tree is a declarative abstraction!
Shading languages

- Goal: support description of a diverse set of materials and lighting conditions

- Idea: allow application to extend graphics pipeline by providing a programmatic definition of shading function logic
Tension: flexibility vs. performance

- Graphics pipeline provides highly optimized implementations of specific visibility operations
  - Examples: clipping, culling, rasterization, z-buffering
  - Highly optimized implementations for a few canonical data structures (triangles, fragments, and pixels)
  - Recall how much the implementation of these functions (including specialized hardware) was deeply intertwined with overall pipeline scheduling/parallelization decisions

- Impractical for rendering system to constrain application to use a single parametric model for surface definitions, lighting, and shading
  - Must allow applications to define these behaviors programmatically
  - Shading language is the interface between application-defined surface, lighting, material reflectance functions and the graphics pipeline
Renderman shading language (RSL)

[Hanrahan and Lawson 90]

- High-level, domain-specific language for describing shading
  - Domain: describing propagation of light through scene

- Developed as interface to extend Pixar’s Renderman renderer

Key question:
what is the structure of the problem we are trying to write programs for?
Why a programming language?

Abstract
A shading language provides a means to extend the shading and lighting formulae used by a rendering system. This paper discusses the design of a new shading language based on previous work of Cook and Perlin. This language has various types of shaders for light sources and surface reflectances, point and color data types, control flow constructs that support the casting of outgoing and the integration of incident light, a clearly specified interface to the rendering system using global state variables, and a host of useful built-in functions. The design issues and their impact on the implementation are also discussed.


Additional Keywords and Phrases: Shading language, little language, illumination, lighting, rendering

1. Introduction
The appearance of objects in computer generated imagery, whether they be realistic or artistic looking, depends both on their shape and shading. The shape of an object arises from the geometry of its surfaces and their position with respect to the camera. The shade or color of an object depends on its illumination environment and its optical properties. In this paper the term shading refers to the combination of light, shade (as in shadows), texture and color that determine the appearance of an object. Many remarkable pictures can be created with objects having a simple shape and complex shading. A well-designed, modular rendering program provides clean interfaces between the geometric processing, which involves transformation, hidden surface removal, etc., and the optical processing, which involves the propagating and filtering of light. This paper describes a language for programming shading computations, and hence, extending the types of materials and light sources available to a rendering system. In Bentley's terminology it would be called a "little" language[3], since, because it is based on a simple subset of C, it easy to parse and implement, but, because it has many high-level features that customize it for shading and lighting calculations, it is easy to use.

Two major aspects of shading are the specification of surface reflectance and light source distribution functions. The earliest surface reflectance models have terms for ambient, diffuse and specular reflection. More recent research has added anisotropic scattering terms[14, 16, 21] and made explicit wavelength and polarization effects. Although not nearly as well publicized, many improvements have also been made in light source description. The earliest light source models consisted of distant or point light sources. Verbeck and Greenberg[29] introduced a general framework for describing light sources which includes attaching them to geometric primitives and specifying their intensity distribution as a function of direction and wavelength.

Light sources and surface reflectance functions are inherently local processes. However, many lighting effects arise because light rays traveling from light to surface are blocked by intervening surfaces or because light arriving at a surface comes indirectly via another surface. Turner Whitted termed these effects global illumination processes. Kajiya introduced the general light transport equation, which he aptly termed the rendering equation, and showed how all these techniques are tied together[15]. Most recent research in shading and lighting calculations is now being focussed on making these global illumination algorithms efficient. Note that global and local illumination processes are independent aspects of the general illumination process.

In parallel to the development of specific shading models is the development of shading systems. Most current systems implement a single parameterized shading model. There are several problems with this approach. First, there is little agreement on what this shading model should be. Almost every rendering system written has used a slightly different shading model. Second, it seems unlikely that a single parameterized model could ever be sufficient. As mentioned earlier, the development of shading models is an active area of research and new material models are continually being developed. Shading also involves many tricks, one major example being texture mapping, and the use of rendering tricks is completely open ended. Furthermore, the surface reflectance models of simple and composite materials are phenomenologically based and not derivable from first principles. Shading models that capture the effects of applying varnish or lac-
RSL programming model

- Structures shading computations using two types of functions: surface shaders and light shaders
  - a.k.a idea of surfaces and lights are built in and central to the language

- Structure of shaders corresponds to structure of the rendering equation:
  - Surface shaders integrate incoming light and compute the reflectance (from a surface point) in the direction of the camera
  - Light shaders compute emitted light in the direction of surface point
Key RSL abstractions

- **Shaders**
  - **Surface shaders:**
    - Define surface reflection function (BRDF)
    - Integrate contribution of light from all light sources
  - **Light shaders:** define directional distribution of energy emitted from lights

- **Light shader’s illuminate construct**

- **Surface shader’s illuminance loop**
  - integrates reflectance over all lights

- **Also:**
  - First-class color and point abstract data types
  - First-class texture sampling functions
Recall: rendering equation

\[ i(x, x') = v(x, x') \left[ l(x, x') + \int r(x, x', x'') i(x', x'') \, dx'' \right] \]

Surface shader integrates contribution to reflection from all lights

Light shader computes \( i(x', x'') \)
(accessed as \( L \) in RSL surface shader illuminance loop)
RSL surface shaders

Executed once per point to shade

Key abstraction: illuminance loop — iterate over illumination sources (but no explicit enumeration of sources: surface definition is agnostic to what lights are linked)

illuminance (position, axis, angle)
{
}

Example: computing diffuse reflectance

surface diffuseMaterial(color Kd)
{
    Ci = 0;

    // integrate light from all lights (over hemisphere of directions)
    illuminance (P, Nn, PI/2)
    {
        Ci += Kd * Cl * (Nn . normalize(L));
    }
}

Surface shader computes Ci

L = Vector from light position (recall light_pos argument to light shader’s illuminate) to surface position being shaded (see P argument to illuminance)

Cl = Value computed by light shader
RSL light shaders

Key abstraction: illuminate block

```cpp
illuminate (light_pos, axis, angle)
{
    // recall: L is RSL built-in light shader variable
    // that is vector from light to surface point
    Cl = my_light_color / (L . L)
}
```

Example: attenuating spot-light (no area fall off)

```cpp
illuminate (light_pos, axis, angle)
{
    // recall: L is RSL built-in light shader variable
    // that is vector from light to surface point
    Cl = my_light_color / (L . L)
}
```
Shading objects in RSL

Shaders are closures:
Shading function code +
values of shader parameters

---

**Surface shader object**

- compiled code (e.g., plastic material)
- current transforms
- bound parameters
  - $kd = 0.5$
  - $ks = 0.3$

**Light shader objects**

- compiled code (spotlight)
  - current transforms
  - bound parameters
    - intensity = 0.75
    - color = (1.0, 1.0, 0.5)
    - position = (5,5,10)
    - axis = (1.0, 1.0, 0)
    - angle = 35

- compiled code (point light)
  - current transforms
  - bound parameters
    - position = (5,5,5)
    - intensity = 0.75
    - color = (1.0, 1.0, 0.5)

- compiled code (point light)
  - current transforms
  - bound parameters
    - position = (20,20,100)
    - intensity = 0.5
    - color = (0.0, 0.0, 1.0)
GPU shading languages today: e.g., HLSL

HLSL shader program example: defines logic of fragment processing stage

```hlsl
sampler mySampler;
Texture2D<float3> myTex;
float3 lightDir[4];

diffuseShader(float3 norm, float2 uv)
{
    float3 kd = myTex.Sample(mySampler, uv);
    float3 lighting = 0.0;
    for (int i=0; i<4; i++)
        lighting += clamp(dot(-lightDir, norm), 0.0, 1.0);
    return float4(lighting * kd, 1.0);
}
```

Note: Imperative abstraction for defining logic within a shader!
Short discussion

Cg: A system for programming graphics hardware in a C-like language

William R. Mark*       R. Steven Glanville†       Kurt Akeley†       Mark J. Kilgard†
The University of Texas at Austin*       NVIDIA Corporation†

This paper discusses the Cg programmer interfaces (i.e. Cg language and APIs) and the high-level Cg system architecture. We focus on describing the key design choices that we faced and on explaining why we made the decisions we did, rather than providing a language tutorial or describing the system’s detailed implementation and internal architecture. More information about the Cg language is available in the language specification [NVIDIA Corp. 2003a] and tutorial [Fernando and Kilgard 2003].
Clear articulation of goals/constraints

The language and system design was guided by a handful of high-level goals:

- **Ease of programming.**
  Programming in assembly language is slow and painful, and discourages the rapid experimentation with ideas and the easy reuse of code that the off-line rendering community has already shown to be crucial for shader design.

- **Portability.**
  We wanted programs to be portable across hardware from different companies, across hardware generations (for DX8-class hardware or better), across operating systems (Windows, Linux, and MacOS X), and across major 3D APIs (OpenGL [Segal and Akeley 2002] and DirectX [Microsoft Corp. 2002a]). Our goal of portability across APIs was largely motivated by the fact that GPU programs, and especially “shader” programs, are often best thought of as art assets – they are associated more closely with the 3D scene model than they are with the actual application code. As a result, a particular GPU program is often used by multiple applications (e.g. content-creation tools), and on different platforms (e.g. PCs and entertainment consoles).

- **Complete support for hardware functionality.**
  We believed that developers would be reluctant to use a high-level language if it blocked access to functionality that was available in assembly language.

- **Performance.**
  End users and developers pay close attention to the performance of graphics systems. Our goal was to design a language and system architecture that could provide performance equal to, or better than, typical hand-written GPU assembly code. We focused primarily on interactive applications.

- **Minimal interference with application data.**
  When designing any system layered between applications and the graphics hardware, it is tempting to have the system manage the scene data because doing so facilitates resource virtualization and certain global optimizations. Toolkits such as SGI’s Performer [Rohlf and Helman 1994] and Electronic Arts’s EAGL [Lalonde and Schenk 2002] are examples of software layers that successfully manage scene data, but their success depends on both their domain-specificity and on the willingness of application developers to organize their code in conforming ways. We wanted Cg to be usable in existing applications, without the need for substantial reorganization. And we wanted Cg to be applicable to a wide variety of interactive and non-interactive application categories. Past experience suggests that these goals are best achieved by avoiding management of scene data.

- **Support for non-shading uses of the GPU.**
  Graphics processors are rapidly becoming sufficiently flexible that they can be used for tasks other than programmable transformation and shading (e.g. [Boltz et al. 2003]). We wanted to design a language that could support these new uses of GPUs.

Some of these goals are in partial conflict with each other. In cases of conflict, the goals of high performance and support for hardware functionality took precedence, as long as doing so did not fundamentally compromise the ease-of-use advantage of programming in a high-level language.

Often system designers must preserve substantial compatibility with old system interfaces (e.g. OpenGL is similar to IRIS GL). In our case, that was a non-goal because most pre-existing high level shader code (e.g. RenderMan shaders) must be modified anyway to achieve real-time performance on today’s graphics architectures.
4 Key Design Decisions

4.1 A “general-purpose language”, not a domain-specific “shading language”

Computer scientists have long debated the merits of domain-specific languages vs. general-purpose languages. We faced the same choice – should we design a language specifically tailored for shading computations, or a more general-purpose language intended to expose the fundamental capabilities of programmable graphics architectures?

4.2 A program for each pipeline stage

The user-programmable processors in today’s graphics architectures use a stream-processing model [Herwitz and Pomerene 1960; Stephens 1997; Kapasi et al. 2002], as shown earlier in Figure 2. In this model, a processor reads one element of data from an input stream, executes a program (stream kernel) that operates on this data, and writes one element of data to an output stream.

Choosing a programming model to layer on top of this stream-processing architecture was a major design question. We initially considered two major alternatives. The first, illustrated by RTSL and to a lesser extent by RenderMan, is to require that the user write a single program, with some auxiliary mechanism for specifying whether particular computations should be performed on the vertex processor or the fragment processor. The second, illustrated by the assembly-level interfaces in OpenGL and Direct3D, is to use two separate programs. In both cases, the programs consume an element of data from one stream, and write an element of data to another stream.

The unified vertex/fragment program model has a number of advantages. It encapsulates all of the computations for a shader in one piece of code, a feature that is particularly comfortable for programmers who are already familiar with RenderMan. It
Cg vs. RSL

- RSL is a language for describing shading computations in a renderer

- Cg is a language for high-performance programming on graphics hardware
  - Expose what the hardware does (while preserving productivity as much as possible)
  - Note: CUDA follows the same philosophy
Shading language design questions

- Design issue: programmer convenience vs. application scope
  - Should we adopt high-level (graphics-specific) or low-level (more general and flexible) abstractions?
  - e.g., Should graphics concepts such as materials and lights be first-class primitives in the programming model?

- Design issue: preserving high performance
  - Abstractions must permit wide data-parallel implementation of fragment shader stage (to utilize many programmable cores)
  - Abstractions must permit use of fixed-function hardware for key shading operations (e.g., texture filtering)
Mapping shader programs to GPUs
Mapping fragments to execution units:

Map fragments to “vector lanes” within one execution context (explicit SIMD parallelism) or to unique contexts that share an instruction stream (parallelization by hardware).
GLSL/HLSL shading languages adopt a SPMD programming model

- **SPMD** = single program, multiple data
  - Programming model used in writing GPU shader programs
    - What’s the program?
    - What’s the data?
  - Also adopted by CUDA and ISPC

- How do we implement a SPMD program on SIMD hardware?
Example 1: shader with a conditional

```cpp
sampler mySamp;
Texture2D<float3> myTex;

float4 fragmentShader(float3 norm, float2 st, float4 frontColor, float4 backColor)
{
    float4 tmp;
    if (norm[2] < 0) // sidedness check (direction of Z component of normal)
    {
        tmp = backColor;
    }
    else
    {
        tmp = frontColor;
        tmp *= myTex.sample(mySamp, st);
    }
    return tmp;
}
```
Example 2: predicate is uniform expression

```cpp
sampler mySamp;
Texture2D<float3> myTex;
float myParam;  // uniform value
float myLoopBound;

float4 fragmentShader(float3 norm, float2 st, float4 frontColor, float4 backColor)
{
    float4 tmp;
    if (myParam < 0.5)
    {
        float scale = myParam * myParam;
        tmp = scale * frontColor;
    }
    else
    {
        tmp = backColor;
    }
    return tmp;
}
```

Notice: predicate is uniform expression (same result for all fragments)
This is one NVIDIA Pascal GP104 streaming multi-processor (SM) unit

- Instructions operate on 32 pieces of data at a time (instruction streams called “warps”).
- Different instructions from up to four warps can be executed simultaneously (simultaneous multi-threading)
- Up to 64 warps are interleaved on the SM (interleaved multi-threading)
- Over 2,048 fragments/vertices/etc can be processed concurrently by a core

= SIMD functional unit, control shared across 32 units (1 MUL-ADD per clock)
= load/store
= SIMD special function unit (sin, cos, etc.)
Context size / latency-hiding trade-off

- Notice there is 256 KB of storage for execution context on this SM
- Max 64 warps = 64 x 32 = max 2048 CUDA threads
- So there’s 128 bytes of “register” storage per thread (32 x 32-bit words)
- So if a shader program, or CUDA thread requires more storage, fewer threads can run at once
  - e.g., 128 registers of storage → only max 16 warps = 512 CUDA threads per SM

Implication: if shaders that have small execution contexts, chip can provide more latency hiding

Implication: shader compilers perform analysis on shaders to statically bound max number of registers needed during execution
  - Inability to compute tight bound can harm performance
Interleaved multi-threading

Consider a processor with:

- Two execution contexts (can execute one instruction per clock)
- Memory latency is 50 cycles
- Running a shader then does 10 math ops, then a load, then another 10 math ops...

How many threads needed for 100% utilization? (six!)
What if program did 50 math ops for every load? (two!)

\[ \text{ALU executing T0 at this time} \]
\[ \text{ALU executing T1 at this time} \]
Storing execution contexts

On-chip storage of execution contexts a finite resource.

Context storage (or L1 cache)
Many small contexts (high latency hiding ability)

Small execution contexts → many execution contexts per core
Four large contexts (low latency hiding ability)

Large execution contexts $\rightarrow$ fewer execution contexts per core
What is the major source of data access in a shader?

Texture access
Shading often has very high arithmetic intensity

```c
sampler mySamp;
Texture2D<float3> myTex;
float3 ks;
float  shinyExp;
float3 lightDir;
float3 viewDir;

float4 phongShader(float3 norm, float2 uv)
{
    float result;
    float3 kd;
    kd = myTex.Sample(mySamp, uv);
    float spec = dot(viewDir, 2 * dot(-lightDir, norm) * norm + lightDir);
    result = kd * clamp(dot(lightDir, norm), 0.0, 1.0);
    result += ks * exp(spec, shinyExp);
    return float4(result, 1.0);
}
```

3 scalar float operations + 1 exp()
8 float3 operations + 1 clamp()
1 texture access

Vertex processing often has even higher arithmetic intensity than fragment processing (less use of texturing)