Stanford
Real-Time Programmable Shading Project

Kekoa Proudfoot
Stanford University

Collaborators:
Pat Hanrahan, Bill Mark, Phillip Slusallek,
Svetoslav Tzvetkov

Sponsors:
3dfx, NVIDIA, SGI, Sun

Web page:
http://graphics.stanford.edu/projects/shading/
Motivation

- **Toy Story**
  - Disney

- **Binary Neutron Star Collision**
  - David Bock

- **Quake 3 Arena**
  - id Software

- **Bump and Shadow Mapping**
  - NVIDIA
Hardware trends

Increasing hardware functionality
- Multiple textures
- Advanced texture combining operations

Increasing fill rate
- Multiple rendering passes

But...
- Programming graphics hardware is like writing microcode
- Decomposing computations into multiple passes is time-consuming
- Functionality varies between chipsets
Higher-level hardware abstractions

Problem:

Current hardware abstractions (e.g. OpenGL) use a configurable pipeline model that is too low-level

Solution:

Use a shading language as a higher-level hardware abstraction
Hardware abstractions

Hardware abstractions:

- Provide a standard interface
- Simplify underlying complexities
- Hide differences in implementations
- Help to define hardware behavior
- Drive new architectures

Hardware abstractions make hardware easier or harder to use
Project goals

- Provide a shading language as an abstraction layer between programmer and graphics hardware
- Explore how current hardware may be used to implement shading language abstractions
- Investigate new hardware architectures optimized for programmable shading
- Create new interactive applications based on shading languages
Our first system

Arbitrary expressions of:

- Constant colors, lit materials, and textures
- Operators: +, *, and over

Expressions compiled to multiple rendering passes

// pool ball shader

// material properties (to configure lighting model)
material Diffuse { diffuse .5 .5 .5; specular 0 0 0; ... }
material Specular { diffuse 0 0 0; specular 1 1 1; ... }

// texture declaration (to configure a texture object)
texture POOLONE { image "one.ppm"; transform { ... }; ... }

// shader definition
shader poolball { Diffuse * POOLONE + Specular; }

Limitations of pure multipass

Problem:

- Pure multipass rendering only allows fragment programmability
- Today’s fragment operations are limited: fixed point, simple set of operators
- Fragment lighting and texture coordinate generation can be very expensive
- No support for future programmable vertex hardware (e.g. DirectX 8)

Solution:

- Add vertex and primitive-group programmability
Multiple computation frequencies

- Constant
- Per Primitive Group
- Per Vertex
- Per Fragment

- Evaluated less often
- More complex operations
- Floating point

- Evaluated more often
- Simpler operations
- Fixed point
System overview

Shading Language

Compiler Front End

Intermediate Representation

+ Programmable Pipeline

Compiler Back End

Compiled Shader “Object Code”

Shader Parameters

Shader Execution Engine

Framebuffer
Anisotropic ball example

surface shader floatv
anisotropic_ball (texref anisotex, texref star)
{
    // generate texture coordinates
    perlight floatv uv = { center(dot(B, E)),
                         center(dot(B, L)),
                         0, 1 };  

    // compute reflection coefficient
    perlight floatv fd = max(dot(N, L), 0);
    perlight floatv fr = fd * texture(anisotex, uv);

    // compute amount of reflected light
    floatv lightcolor = 0.2 * Ca + integrate(Cl * fr);  

    // modulate reflected light color
    floatv uv_base = { center(Pobj[2]), center(Pobj[0]),
                      0, 1 };  
    return lightcolor * texture(star, uv_base);  
}
Managing computation frequencies

Given: a system with multiple computation frequencies

How to specify how often to compute something?

Two methods:
- Explicit specification with type modifiers
- Automatic propagation
Computation frequency modifiers

Four type modifiers allow explicit specification:

constant
perbegin
vertex
fragment

Specification by assignment:

fragment float y = x;

Specification by type cast:

float y = (fragment float)x;
Automatic propagation

Computation frequencies propagate from operation inputs to operation outputs

All shader inputs have default frequencies
Surfaces and lights

From RenderMan:

Separate surfaces and lights by defining two kinds of shaders
A linear integrate operator

RenderMan combines surfaces and lights using illuminance

- Implicit loop over lights
- Unrestricted combining of computed light values

We define a linear integrate() operator

\[
\text{integrate}(a + b) = \text{integrate}(a) + \text{integrate}(b)
\]

If \( k \) is the same for every light:

\[
\text{integrate}(k \times a) = k \times \text{integrate}(a)
\]

Restricted to combining light values using addition
Perlight expressions

The `integrate()` operator evaluates a perlight expression once for every light, summing the results.

Three built-in perlight globals

- \( L \) light vector
- \( H \) halfangle vector
- \( C_l \) light color

Expressions of perlight values are themselves perlight.
Integrate example

Anisotropic ball example:

\[
\begin{align*}
\text{perlight floatv } fd &= \max(\text{dot}(N, L), 0); \\
\text{perlight floatv } fr &= fd \times \text{texture}(\ldots); \\
\text{floatv } \text{lightcolor} &= \ldots + \text{integrate}(Cl \times fr);
\end{align*}
\]

With two lights, expands to:

\[
\begin{align*}
\text{floatv } fd_0 &= \max(\text{dot}(N, L_0), 0); \\
\text{floatv } fr_0 &= fd_0 \times \text{texture}(\ldots); \\
\text{floatv } fd_1 &= \max(\text{dot}(N, L_1), 0); \\
\text{floatv } fr_1 &= fd_1 \times \text{texture}(\ldots); \\
\text{floatv } \text{lightcolor} &= \ldots + Cl_0 \times fr_0 + Cl_1 \times fr_1;
\end{align*}
\]
Integrate optimization

A linear integrate allows an optimized light sum:

```c
floatv Kd = texture(...);  // non-perlight
floatv NdotL = dot(N,L);   // perlight
floatv color = integrate(Kd * dot(N,L));
```

No optimization:

```c
Kd * dot(N,L_0) + Kd * dot(N,L_1)
```

2 fragment multiplies
1 fragment add

Factor out non-perlight term:

```c
Kd * (dot(N,L_0) + dot(N,L_1))
```

1 fragment multiply
1 vertex add
Vertex and fragment lights

Vertex lights
- Return a per-vertex light color

Fragment lights
- Return a per-fragment light color
- Usually involves a projective texture

Automatic propagation of computation frequencies allows vertex and/or fragment integrate() as appropriate

Sort lights by computation frequency to optimize
Operators and types

Seven basic types:

float, floatv, clampf, clampfv, matrix, bool, texref

<table>
<thead>
<tr>
<th>Primitive group ops</th>
<th>Vertex ops</th>
<th>Fragment ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>cross product, matrix generation, matrix multiply, sin, cos</td>
<td>divide, compare ops, clamp, dot, length, min, max, normalize, pow, reflect, select, sqrt, vector index, scalar join</td>
<td>add, subtract, multiply, blend</td>
</tr>
<tr>
<td>+ vertex ops</td>
<td>+ fragment ops</td>
<td>Fragment only: texture lookups</td>
</tr>
<tr>
<td>+ fragment ops</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See our web page for details
Builtin global variables

For surfaces:

- \( P \) - surface position
- \( P_{obj} \) - surface position (object space)
- \( N, T, B \) - normal, tangent, binormal vectors
- \( E \) - eye vector
- \( Ca \) - global ambient light color
- \( L \) - light vector
- \( H \) - halfangle vector
- \( Cl \) - light color

For lights:

- \( S \) - surface vector (light space)
- \( S_{dist} \) - distance to surface

Inspired by RenderMan
Language constraints

Language is constrained to promote SIMD parallelism across vertices and fragments.

No conditional or loop statements

- A `select` operator enables conditional expressions
- A `repeat(n)` construct could be added to enable limited forms of looping

No explicit communication between data elements
Demo
Programmable pipeline abstraction

Three programmable pipeline stages

Intermediate level abstraction between language and OpenGL

Hardware independent:
- No operation count limits
- No temporary storage limits
- Abstract types: float, clampf
Intermediate representation

A compiler intermediate representation is used to specify programmable pipeline programs

- Same operators and types as language
- Surfaces and lights are combined
- Builtin globals are expanded
- Function calls are inlined
- Constants are fully simplified
Front end compilation

Three steps:

- Parse shader input file, inlining globals and functions, simplifying constants
- Join surface/light shaders together to make a single pipeline program
- Determine computation frequencies and split pipeline program accordingly

Result:

- Three-part pipeline program, one part for each programmable pipeline stage
Back end compilation

Goal: Produce an executable version of a pipeline program

Hardware mappings by computation frequency:

- Primitive group → Host processor
- Vertex → Host processor
- Fragment → Multiple rendering passes

When hardware becomes available: move host-side computations to graphics processor
Two techniques for generating code:

- External C compiler
- Internal x86 code generator
Fragment computations are mapped to multiple rendering passes using *lburg*

- Define rules corresponding to particular configurations of portions of the fragment pipeline
- Dynamic programming optimally covers trees given rules
- Additional rules are enabled if necessary GL extensions are present
- Despite optimal cover, *lburg* isn’t perfect

*lburg* is from Fraser and Hanson, *A Retargetable C Compiler: Design and Implementation*
We abstract the OpenGL pipeline as implementing two kinds of operations:

\[
\begin{align*}
\text{fb} &= \left\{ \begin{array}{c} C \\ T \\ C \ op \ T \end{array} \right\} \left[ \begin{array}{c} \text{op} \ T \\ \text{op} \ \text{fb} \end{array} \right] \\
\text{T} &= \text{fb} \\
\end{align*}
\]

\(\text{op}\) is one of: +, −, ×, blend

\text{fb} = \text{framebuffer color}
\text{C} = \text{triangle color}
\text{T} = \text{texture}

All of our \textit{lburg} rules are derived from this abstraction.
Pass generation example

Unextended OpenGL requires five passes to cover this tree

\[ fb = Base \]
\[ fb = Bruns \text{ over } fb \]
\[ fb = \text{Circle over } fb \]
\[ fb = fb \times \text{Marks} \times \text{Cd} \]
\[ fb = fb + Cs \]
API

Primary differences compared to current APIs:

- Support for compiling pipeline programs
- Support for arbitrary per-primitive-group and per-vertex parameters
- Hidden multipass rendering

Our system provides immediate mode and vertex array interfaces

- Both result in buffers filled with primitive group and vertex data to be processed and rendered
Review

Multiple computation frequencies
- Support for computing values at different rates allows for a much broader set of operations and types with reasonable cost

Shading language abstraction
- User-level abstraction layer
- Type system for multiple computation frequencies
- Linear integrate operator

Hardware-independent programmable pipeline
- Intermediate abstraction layer to separate language from hardware
Take-home message

Real-time programmable shading
- Can be implemented today
- Makes complicated hardware easy to use
- Simplifies multipass rendering
- Hides hardware dependencies
- Will drive future generations of graphics hardware

Real-time programmable shading is the next big thing!!!