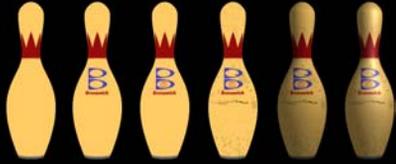


Arbitrarily Complex Shading On DirectX 9 Graphics Hardware



Eric Chan
Stanford University

<http://graphics.stanford.edu/projects/shading/>

Outline

Overview of Stanford shading system

- Language features
- Compiler architecture

Recent work

- DirectX 9 back ends
- General multipass support

Comparison to other shading languages

Motivation

Research project began in 1999

Problem:

- Graphics hardware tough to program because of low-level, non-portable interfaces

Solution:

- Shading languages give users high-level access to programmable features

Project Goals

1. Implement real-time shading language (RTSL)
2. Support a variety of hardware
3. Generate efficient code
4. Investigate future hardware features

RTSL Language Features

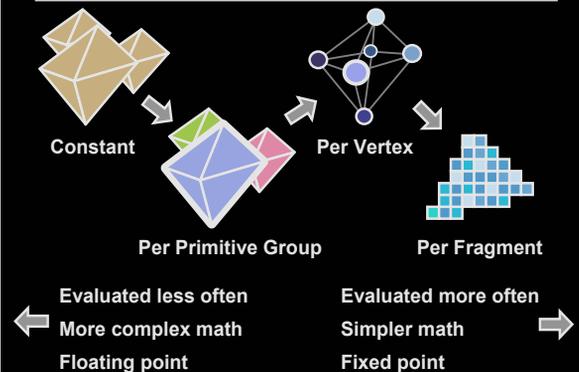
Many features inspired by RenderMan:

- C-like syntax
- Data types and operators for graphics
- Surface and light shaders

Model:

- Single programmable pipeline with multiple computation frequencies

Multiple Computation Frequencies



Shading Language Example

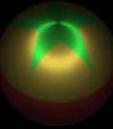
```

surface shader float4
anisotropic_ball (texref anisotex, texref star)
{
    // generate texture coordinates
    perlight float4 uv = { center(dot(B, E)),
                          center(dot(B, L)),
                          0, 1 };

    // compute reflection coefficient
    perlight float4 fd = max(dot(N, L), 0);
    perlight float4 fr = fd * texture(anisotex, uv);

    // compute amount of reflected light
    float4 lightcolor = 0.2 * Ca + integrate(Cl * fr);

    // modulate reflected light color
    float4 uv_base = { center(Pobj[2]), center(Pobj[0]),
                     0, 1 };
    return lightcolor * texture(star, uv_base);
}
    
```



Surface and Light Shaders

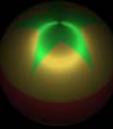
```

surface shader float4
anisotropic_ball (texref anisotex, texref star)
{
    // generate texture coordinates
    perlight float4 uv = { center(dot(B, E)),
                          center(dot(B, L)),
                          0, 1 };

    // compute reflection coefficient
    perlight float4 fd = max(dot(N, L), 0);
    perlight float4 fr = fd * texture(anisotex, uv);

    // compute amount of reflected light
    float4 lightcolor = 0.2 * Ca + integrate(Cl * fr);

    // modulate reflected light color
    float4 uv_base = { center(Pobj[2]), center(Pobj[0]),
                     0, 1 };
    return lightcolor * texture(star, uv_base);
}
    
```



Computation Frequency Analysis

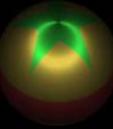
```

surface shader float4
anisotropic_ball (texref anisotex, texref star)
{
    // generate texture coordinates
    perlight float4 uv = { center(dot(B, E)),
                          center(dot(B, L)),
                          0, 1 };

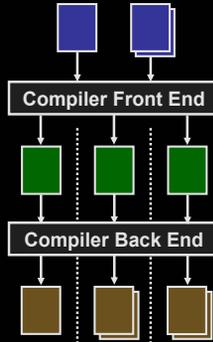
    // compute reflection coefficient
    perlight float4 fd = max(dot(N, L), 0);
    perlight float4 fr = fd * texture(anisotex, uv);

    // compute amount of reflected light
    float4 lightcolor = 0.2 * Ca + integrate(Cl * fr);

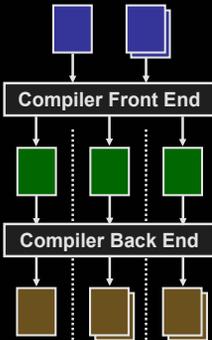
    // modulate reflected light color
    float4 uv_base = { center(Pobj[2]), center(Pobj[0]),
                     0, 1 };
    return lightcolor * texture(star, uv_base);
}
    
```



System Overview

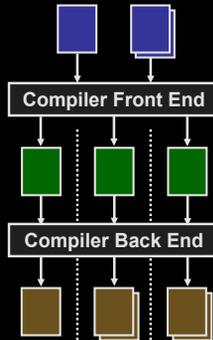


System Overview



Shading language abstraction:
Surface and light shaders

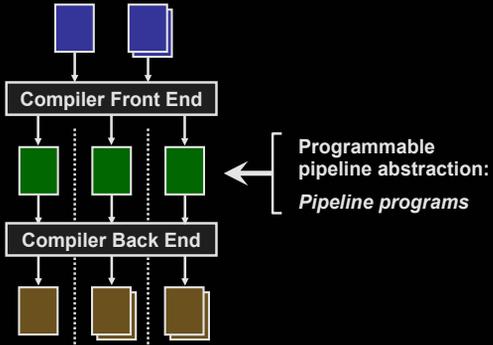
System Overview



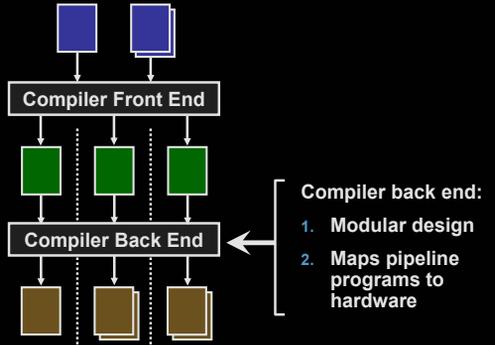
Compiler front end:

1. Combines surface and light shaders
2. Maps shaders to intermediate abstraction

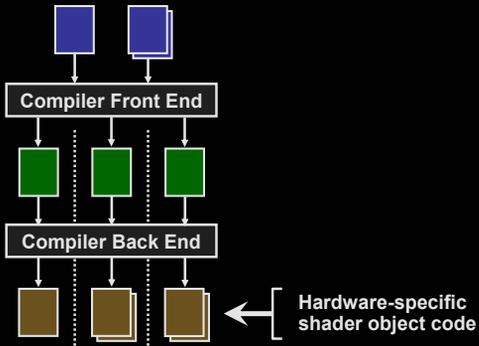
System Overview



System Overview



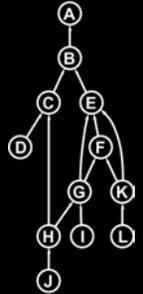
System Overview



Single Compiler Front End

Simplified analysis:

- No data-dependent loops or conditionals
- All functions inlined
- All shading computations reduced to one directed acyclic graph (DAG)



Retargetable Compiler Back End

Two goals:

- Provide support for many hardware platforms
- Virtualize hardware resources

Back End Modules

Host processor:

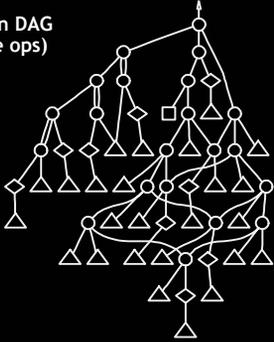
- C code with external compiler
- Internal x86 assembler

Hardware:

- Multipass OpenGL with extensions
- NVIDIA vertex programs
- NVIDIA register combiners
- ATI vertex and fragment shaders
- Stanford Imagine processor
- *DirectX 9 GPUs ...*

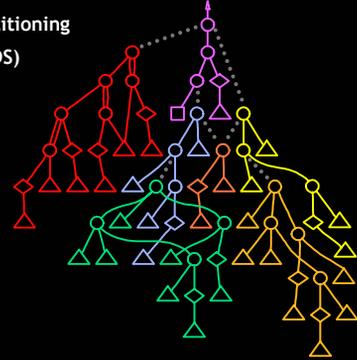
Fragment Compiler Overview (3)

Instruction DAG
(hardware ops)



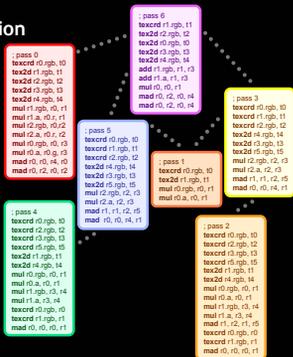
Fragment Compiler Overview (4)

Pass partitioning
(using RDS)



Fragment Compiler Overview (5)

Code generation



Images

Produced using:

- OpenGL + NV_fragment_program
- NVIDIA "Buzz" emulation driver

Shaders consist entirely of fragment computations

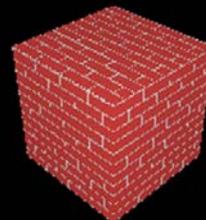
Procedural Textures



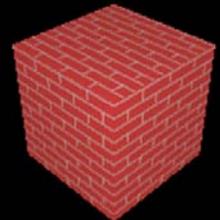
Procedural Anti-Aliasing

Use new screen-space derivative operators

Aliased (45 ops)

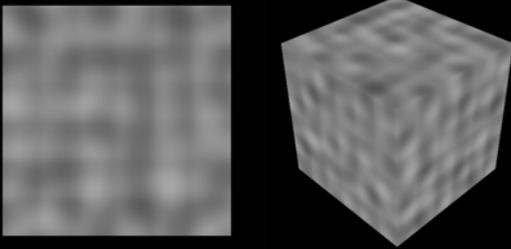


Anti-aliased (74 ops)



Procedural Noise + Solid Textures

- Perlin's original noise implementation
- Lots of computation and texture lookups (48 ops)



Wood Surface

- Originally a RenderMan shader by Larry Gritz
- See RenderMan Repository online
- Uses noise function 3 times
- 207 ops



Wood Surface



Wood Surface



What About Really Big Shaders?

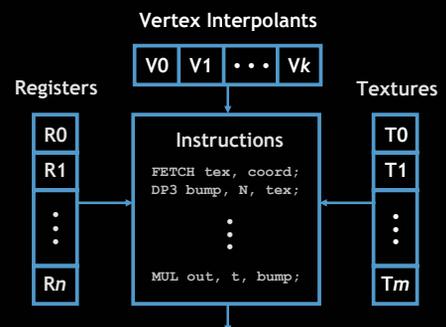
Shading system abstraction:

- Conceptually, one rendering pass
- Internally splits shaders into passes if needed

Why multipass?

- Easy to write large shaders using high-level languages
- Large computations are important, even if too slow to run in real-time on today's hardware
- Hardware more programmable, but still has resource limits

Resource Constraints Example



Virtualization Using Multipass

Basic idea:

- Split shaders into multiple passes; each pass satisfies all resource constraints.
- Intermediate results *saved* to texture memory and *restored* in later passes.
- Requires floating-point!

Problem:

- There are many ways to split a shader. Which one renders the fastest?

Pass Split Algorithm

Goals:

- Support arbitrarily large shaders
- Efficiently target programmable hardware

Support:

- Hardware with different resource constraints
- Hardware with different performance behavior

HWWS 2002 paper [Chan et al.]

Recursive Dominator Split (RDS)

1. Algorithm overview
2. Implementation
3. Demo

Multipass Partitioning Problem

Definitions:

- Each way of splitting a shader is a *partition*
- A *cost model* evaluates the cost of partitions
- A partition is *valid* if each pass satisfies all constraints

Task:

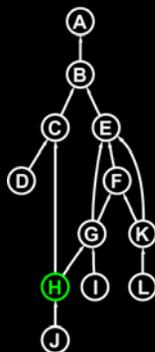
- Given a DAG and a cost model, find a valid partition with the lowest cost

RDS Algorithm Overview

Basic strategy to find best partition:

1. Greedy bottom-up merging for fewer passes
2. Search over multiply-referenced nodes for save vs. recompute

See paper for details



Demo

Shader:

- RenderMan bowling pin
- 1 point light source
- 4 animated projected texture lights

Hardware + Software:

- ATI Radeon 9700 (R300)
- OpenGL + ATI_fragment_program



RDS Remarks

Pros:

- Supports arbitrarily large shaders
- Works on different architectures
- Usually within 5% of optimal (measured by cost)

Cons:

- Doesn't support branching
- Doesn't support multiple outputs

Outline

Overview of Stanford shading system

- Language features
- Compiler architecture

Recent work

- DirectX 9 back ends
- General multipass support

Comparison to other shading languages

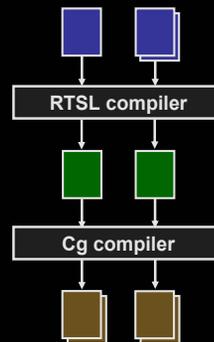
Comparison To Other Languages

Compared to Cg / 3D Labs' OpenGL 2.0 proposal:

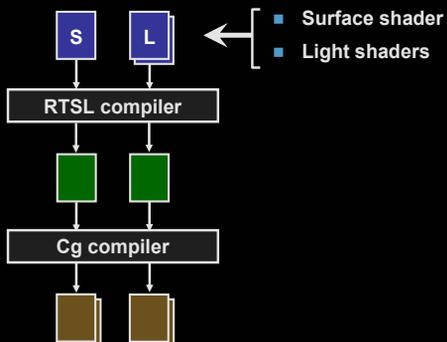
1. Surface and light shaders
2. Single pipeline program split by computation frequency
3. Hides multipass

RTSL provides higher-level abstraction than Cg

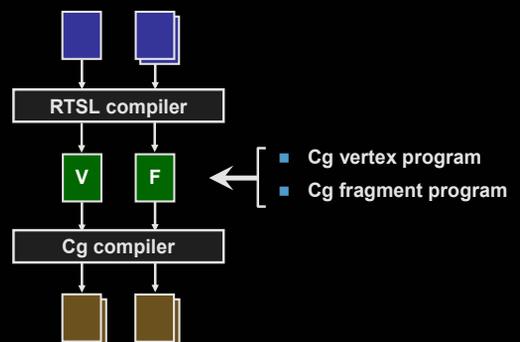
RTSL to Cg



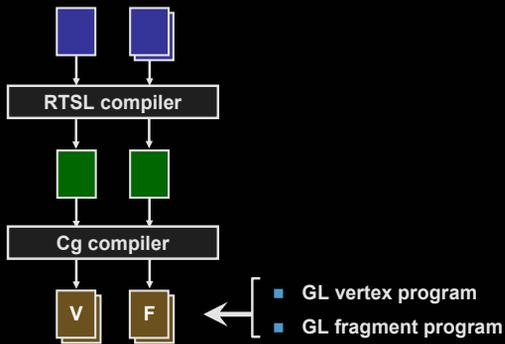
RTSL to Cg



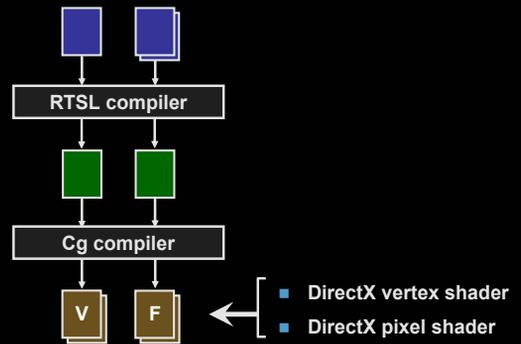
RTSL to Cg



RTSL to Cg (OpenGL)



RTSL to Cg (DirectX)



Summary

Current system:

- DirectX 9 fragment programmability
- Arbitrarily complex shaders via multipass
- Compiles to lower-level languages such as Cg

Final Thoughts

Industry will improve code generators

Co-existence of different types of shading languages

- Higher-level, domain-specific (e.g. RTSL)
- Lower-level, general (e.g. Cg or 3D Labs' OpenGL 2.0 proposal)

Map wild algorithms to the GPU:

- Ray tracing
- Physical simulations (fluid flow, etc.)
- Cryptography

Acknowledgements

Stanford Shading Group & Collaborators

- Kekoa Proudfoot, Bill Mark, Pat Hanrahan, Pradeep Sen, Ren Ng, Svetoslav Tzvetkov, John Owens, Ian Buck, Philipp Slusallek, David Ebert, Marc Levoy

Sponsors

- ATI, NVIDIA, Sony, Sun
- DARPA, DOE

Hardware, drivers, and bug fixes

- Matt Papakipos, Mark Kilgard, Nick Triantos, Pat Brown
- James Percy, Bob Drebin, Evan Hart, Steve Morein, Andrew Gruber, Jason Mitchell

Acknowledgements (Demos)

1. Textbook Strike

- Demo code: Pradeep Sen
- Original scene: Tom Porter
- Animation data: Anselmo Lastra, Lawrence Kestelfoot, Fredrik Fatemi

2. Animated Fish

- Demo code: Ren Ng
- Animation and models: Xiaoyuan Tu, Homan Igehy, Gordon Stoll

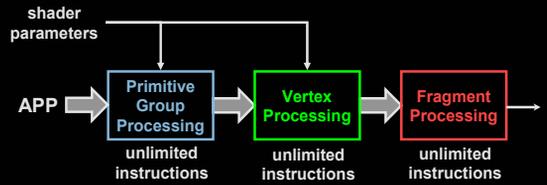
3. Volume Rendering

- Demo code: Ren Ng
- Mouse data: G. A. Johnson, G.P.Cofer, S.L. Gewalt, L.W. Hedlund at Duke Center for In Vivo Microscopy

Questions?

- ericchan@graphics.stanford.edu
- <http://graphics.stanford.edu/projects/shading/>

Programmable Pipeline Abstraction



- Unified framework for all computation frequencies
- Virtualization of hardware resources
- Conceptually only one rendering pass

Compiler Experiences

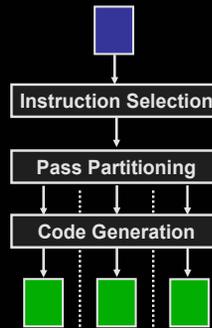
Vertex back ends:

- Similar APIs
- Clean design
- Traditional compiler techniques

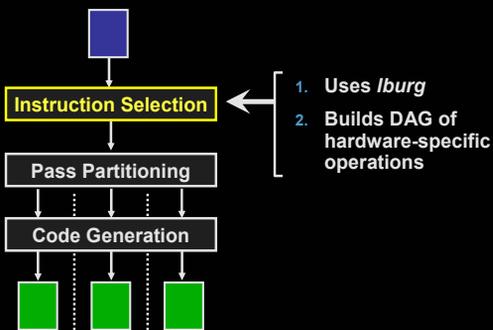
Fragment back ends (up until now):

- Very different APIs
- Hardware not orthogonal
- Compiler more complex
- Still, compilation quality very good

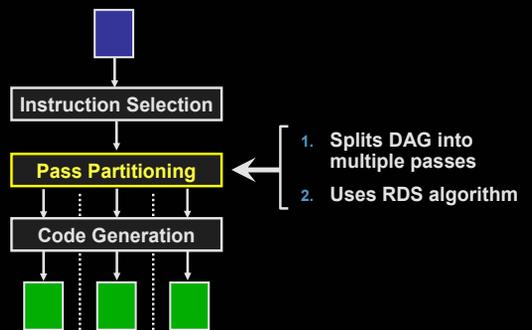
Fragment Compiler Overview



Fragment Compiler Overview



Fragment Compiler Overview



Fragment Compiler Overview

