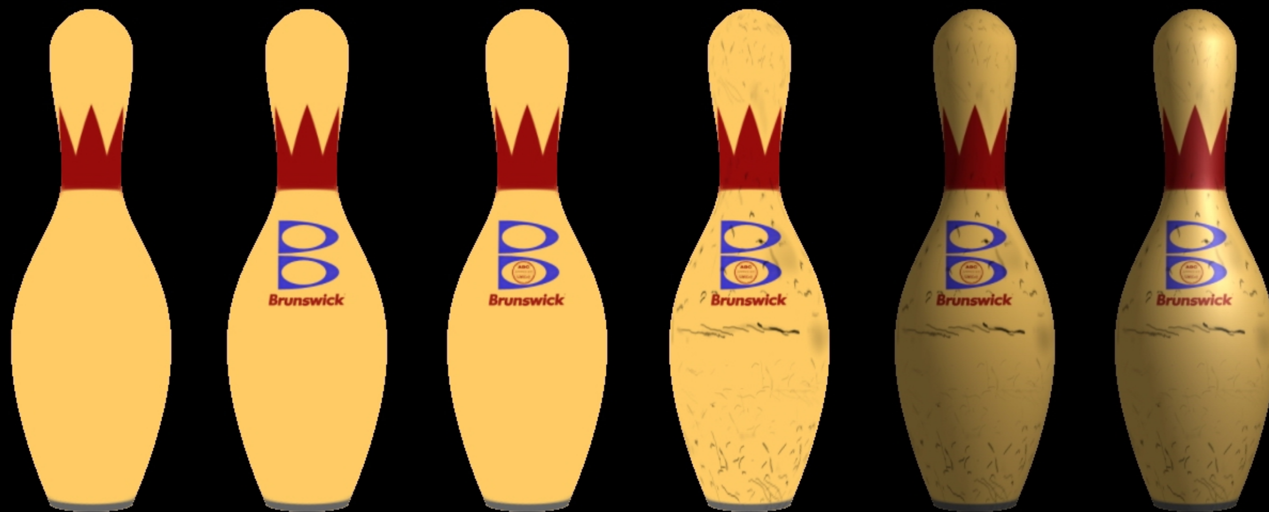
Complex Multipass Shading On Programmable Graphics Hardware



Eric Chan
Stanford University

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

Outline

Overview of Stanford shading system

- Language features
- Compiler architecture

Recent work

- Back ends for DX9-class GPUs
- 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

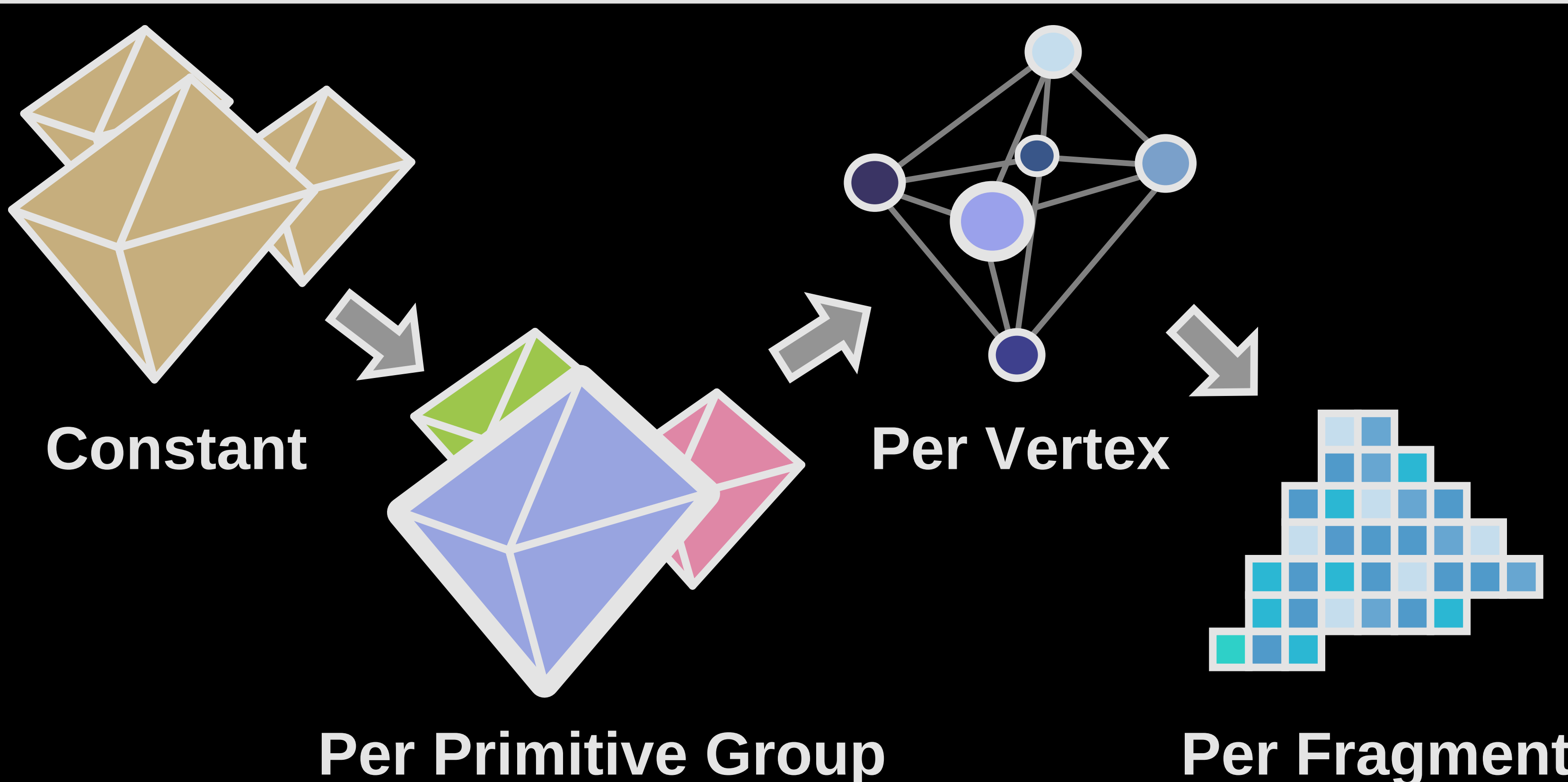
Many features inspired by RenderMan:

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

Model:

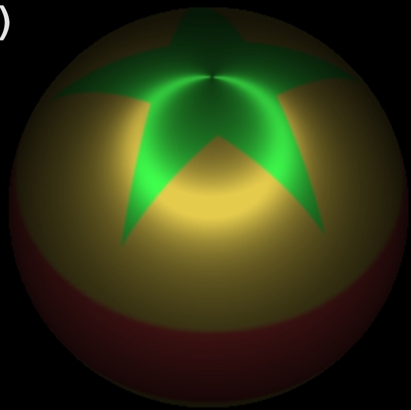
- Single programmable pipeline with multiple computation frequencies

Multiple Computation Frequencies



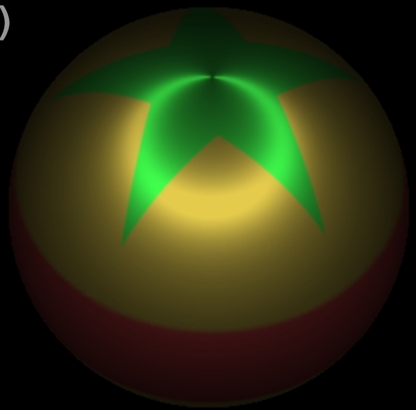
Evaluated less often	Evaluated more often
More complex math	Simpler math
Floating point	Fixed point

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);
}
```

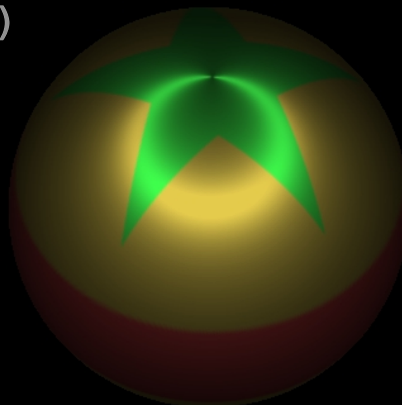
Surface and Light Shaders



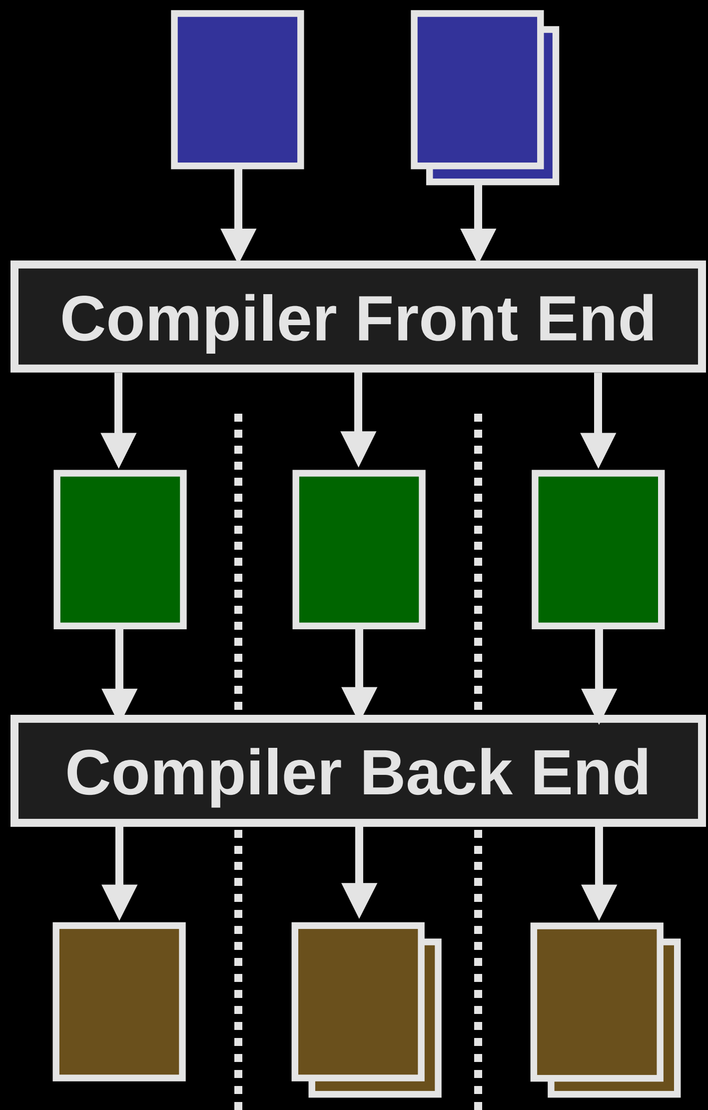
```
surface shader float4
anisotropic_ball (texref anisotex, texref star)
{
    // generate texture coordinates
    perlight float4 uv = { center(dot(B, E)),
                           center(dot(B, L)),
                           0, 1 };

    // 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

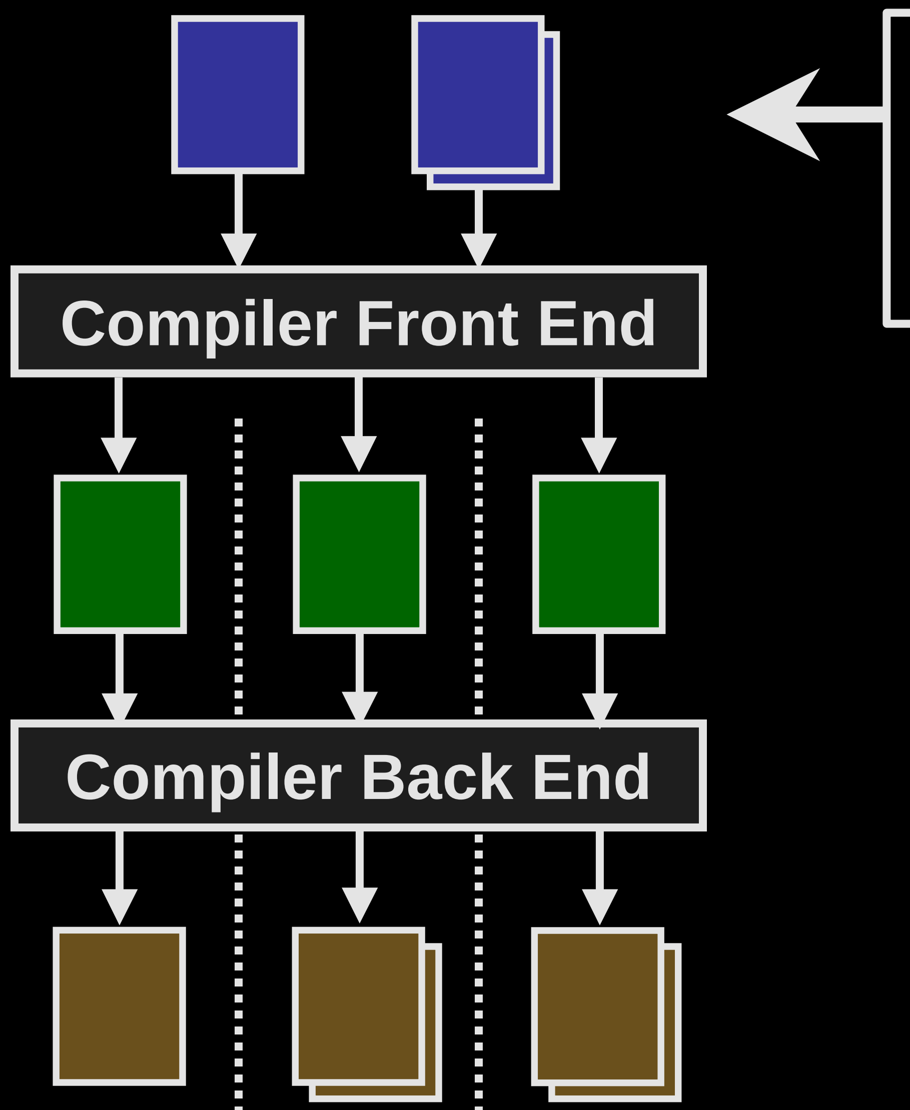
```
surface shader float4
anisotropic_ball (texref anisotex, texref star)
{
    // generate texture coordinates
    perlight float4 uv = { center(dot(B, E)),
                           center(dot(B, L)),
                           0, 1 };

    // 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



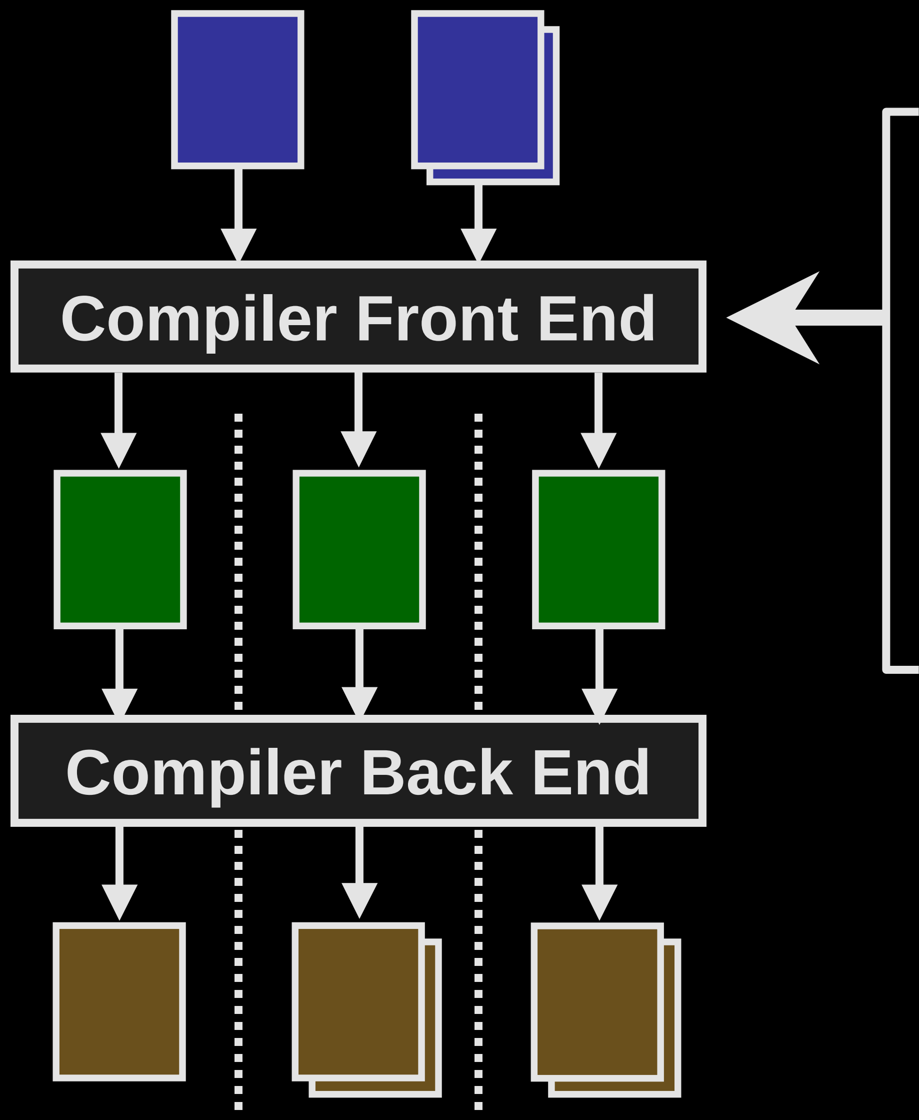
System Overview



Shading language abstraction:

Surface and light shaders

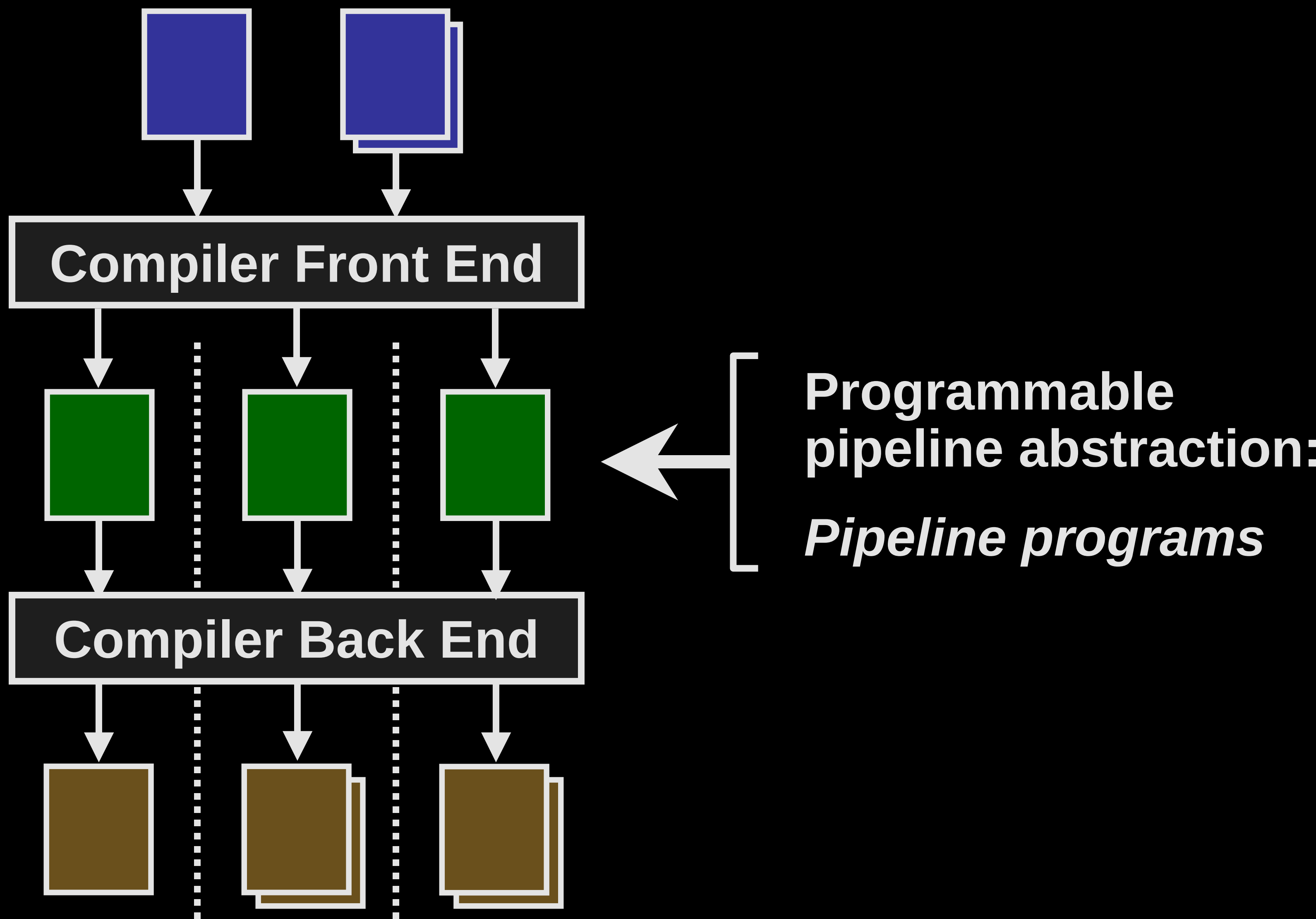
System Overview



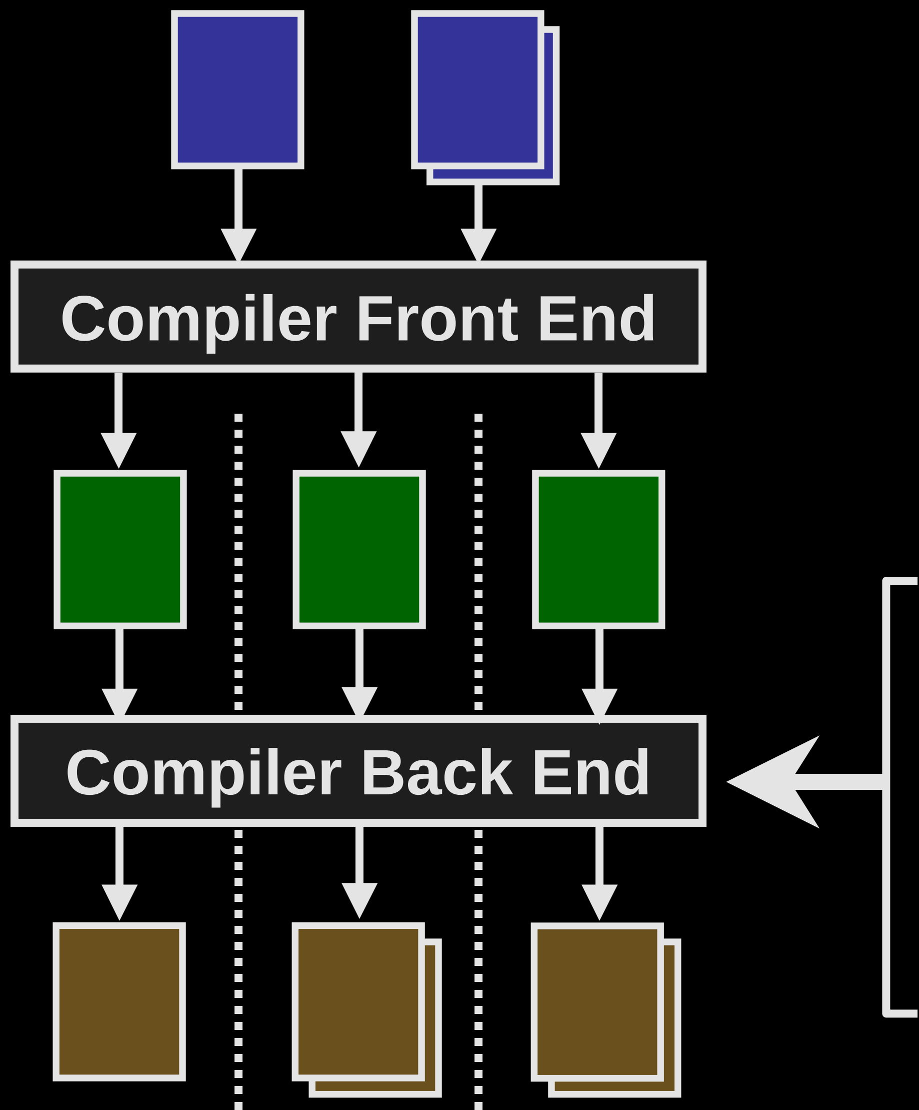
Compiler front end:

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

System Overview



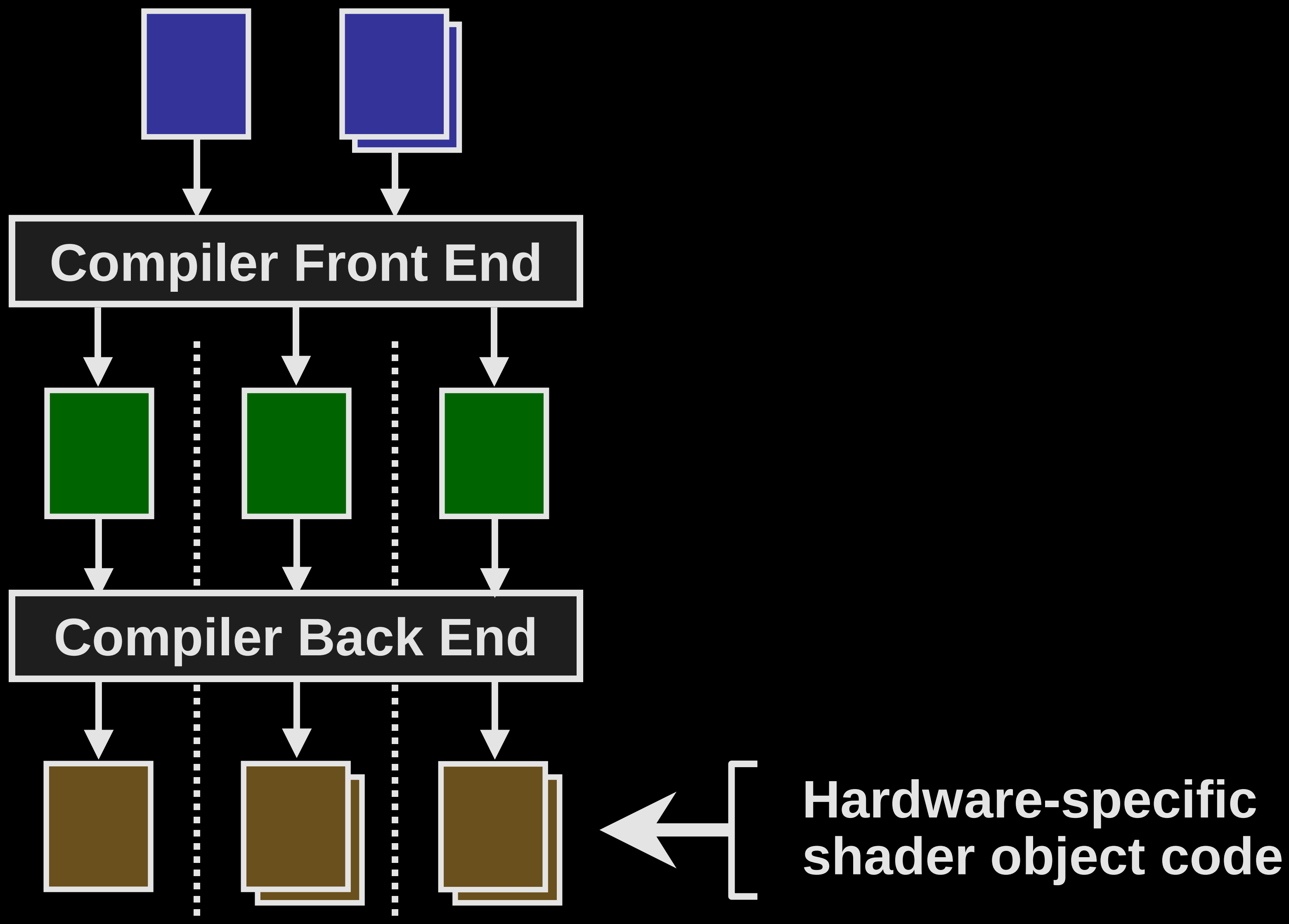
System Overview



Compiler back end:

1. Modular design
2. Maps pipeline programs to hardware

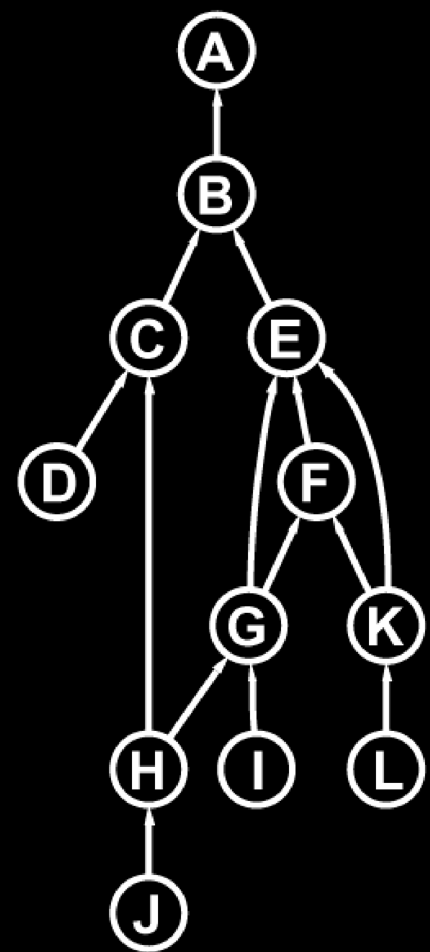
System Overview



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:

1. Virtualize hardware resources
2. Provide support for many hardware platforms

Virtualization

Primitive group back ends:

- Always executed on host — no virtualization needed

Vertex back ends:

- Can fall back to host (e.g. x86 assembly) if needed

Fragment back ends:

- Use multipass
- Works well for non-programmable hardware
- Harder for programmable hardware

Back End Modules

Host processor:

- C code with external compiler
- Internal x86 assembler

Hardware:

- Multipass OpenGL 1.2 with extensions
- NVIDIA vertex programs
- NVIDIA register combiners
- ATI vertex and fragment shaders
- Stanford Imagine processor
- *DX9-like GPUs* …

Demo

OpenGL 1.2 (multipass)

- Constructing the RenderMan Bowling Pin

NVIDIA nv2x (register combiners)

- Textbook Strike
- Animated Fish
- Volume Rendering

Summary

Programmable hardware

- Very capable, but hard to use
- Need a shading language interface

Stanford shading system

- Shading language designed for hardware
- Programmable pipeline abstraction
- Retargetable compiler back end
- Runs in real-time on today's hardware

Outline

Overview of Stanford shading system

- Language features
- Compiler architecture

Recent work

- Back ends for DX9-class GPUs
- General multipass support

Comparison to other shading languages

Coming Soon: DirectX 9

Increased fragment programmability:

- Similar to current vertex programs
- More complex operators
- Floating-point support

Already supported in our shading system:

- Updated language exposes new hardware features
- New back ends target DX9 hardware

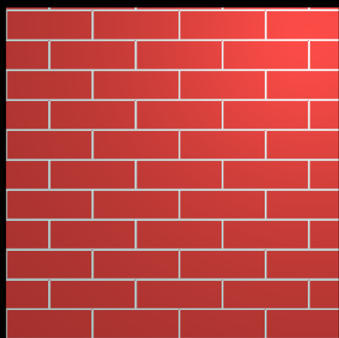
Images

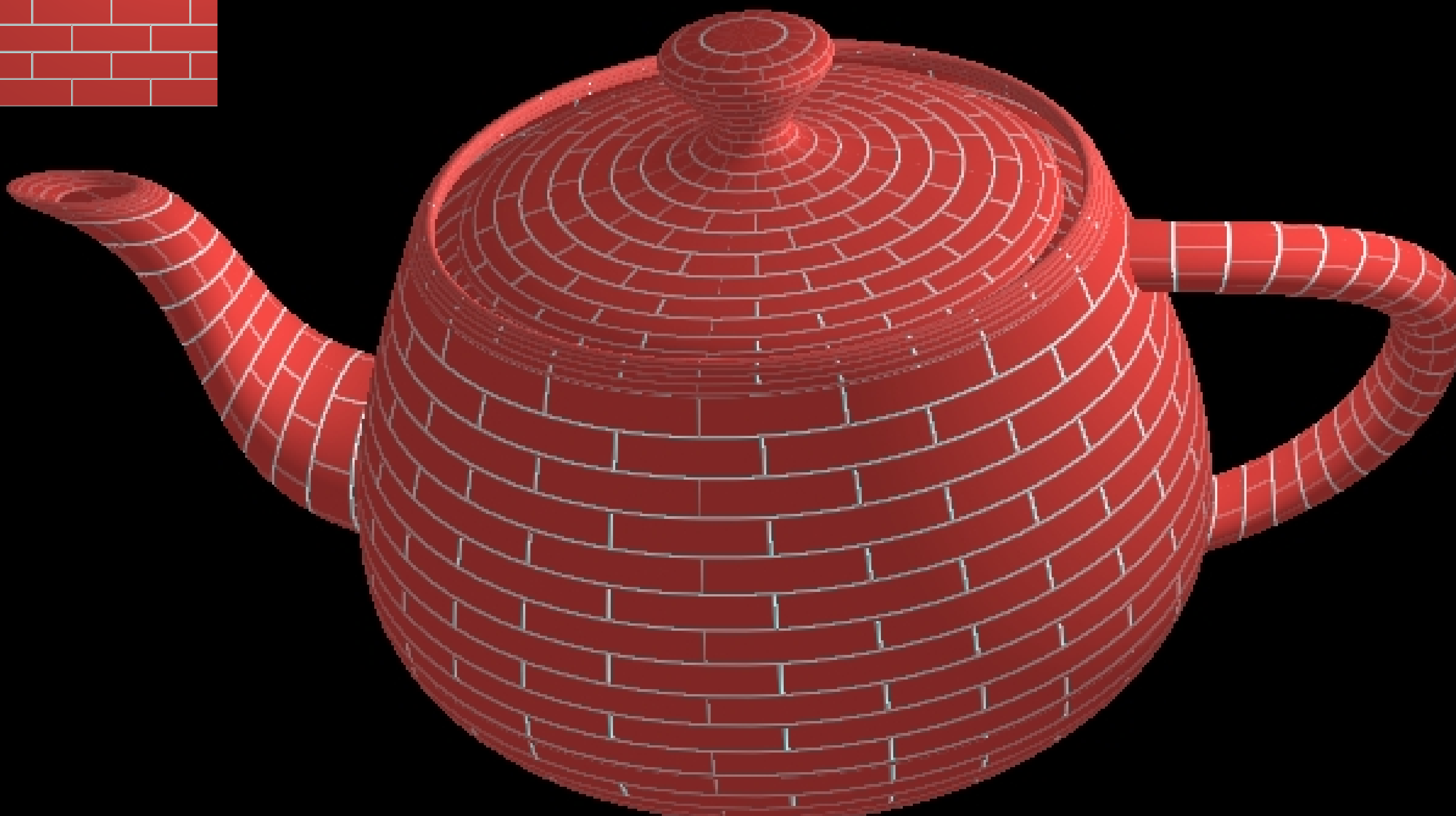
Produced using:

- OpenGL + NV_fragment_program
- NVIDIA NV30 emulation driver

Shaders consist entirely of fragment computations

Procedural Textures

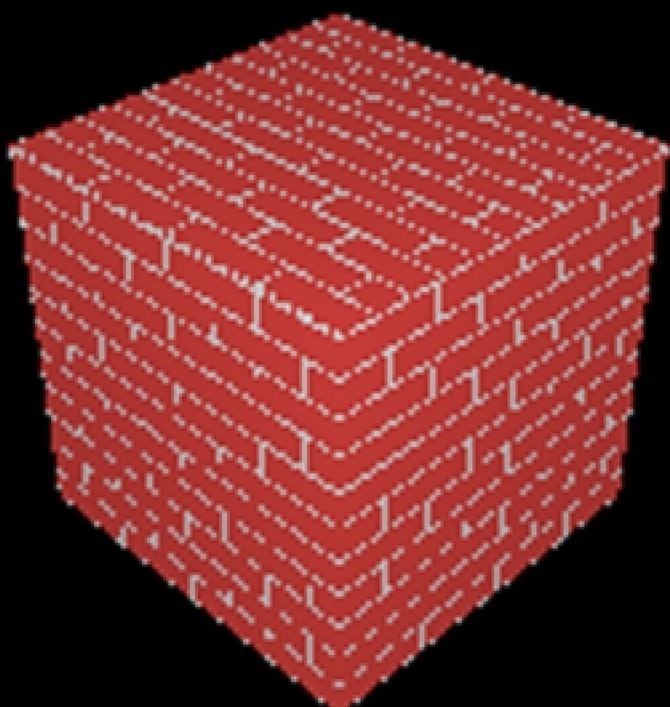




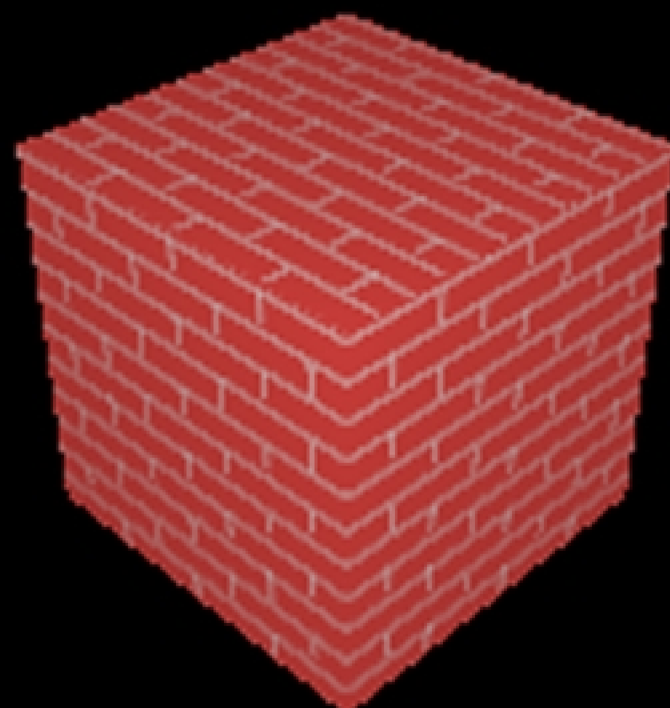
Procedural Anti-Aliasing

Use new screen-space derivative operators

Aliased (45 ops)

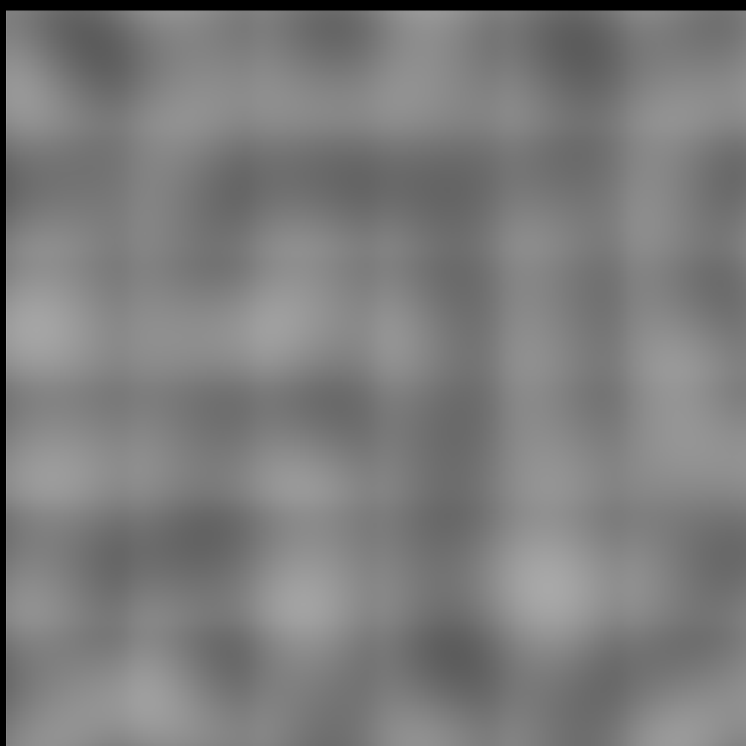


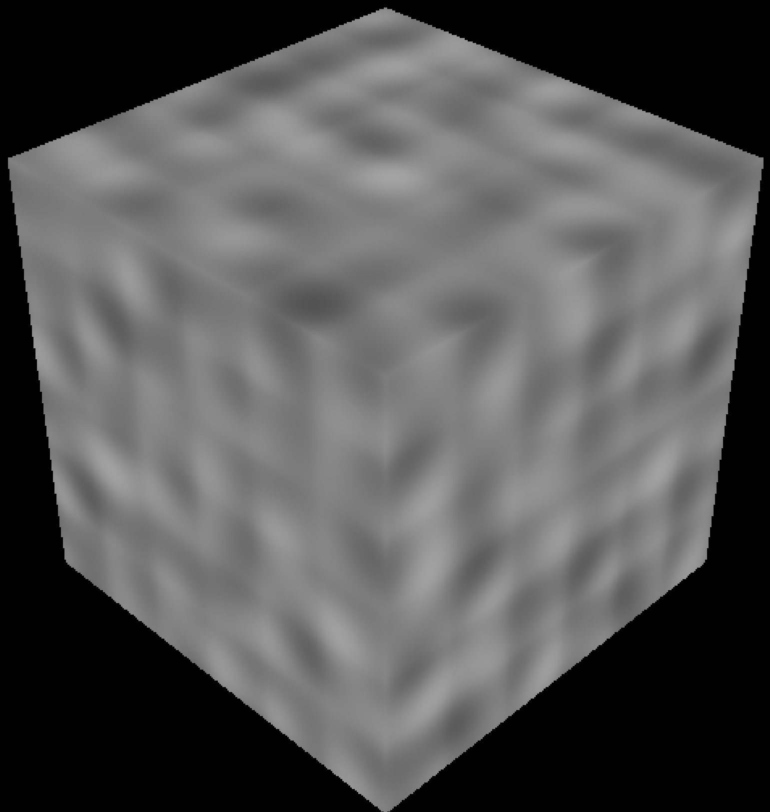
Anti-aliased (74 ops)



Procedural Noise + Solid Textures

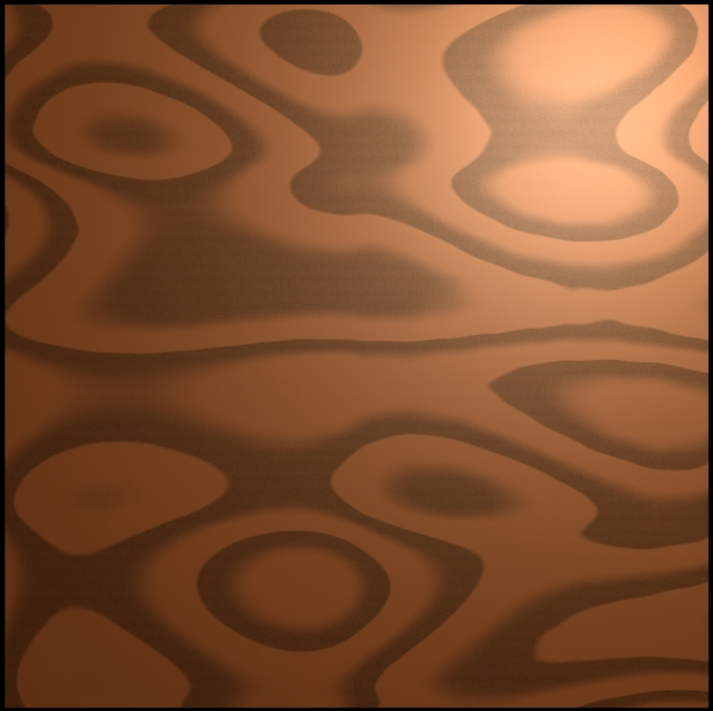
- Perlin’s original noise implementation
- Lots of computation and texture lookups (48 ops)





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?

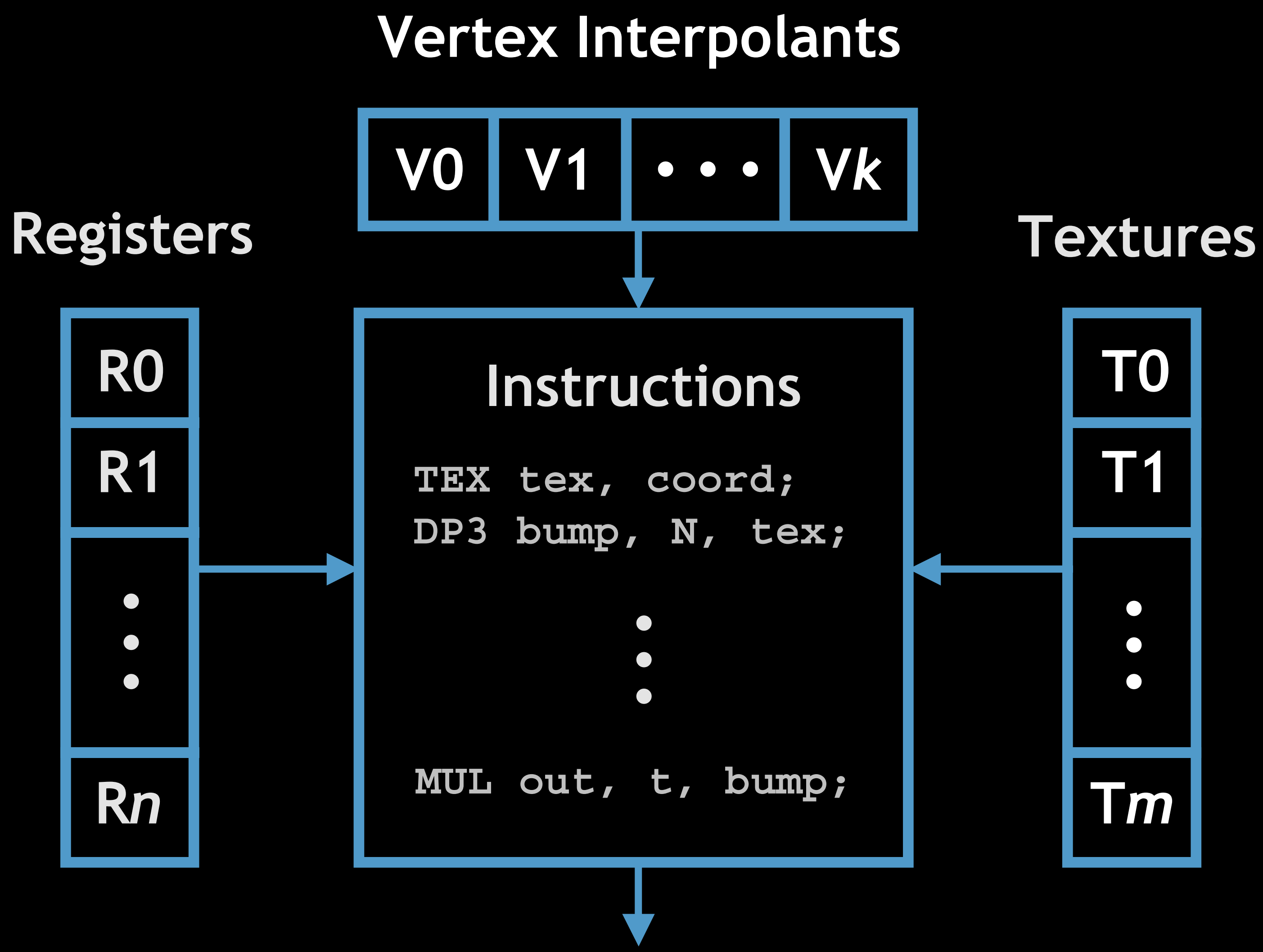
Shading system abstraction:

- 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

- Eric Chan, Ren Ng, Pradeep Sen, Kekoa Proudfoot, Pat Hanrahan

Recursive Dominator Split (RDS)

1. Pass split example
2. Problem statement
3. Algorithm overview
4. Demo

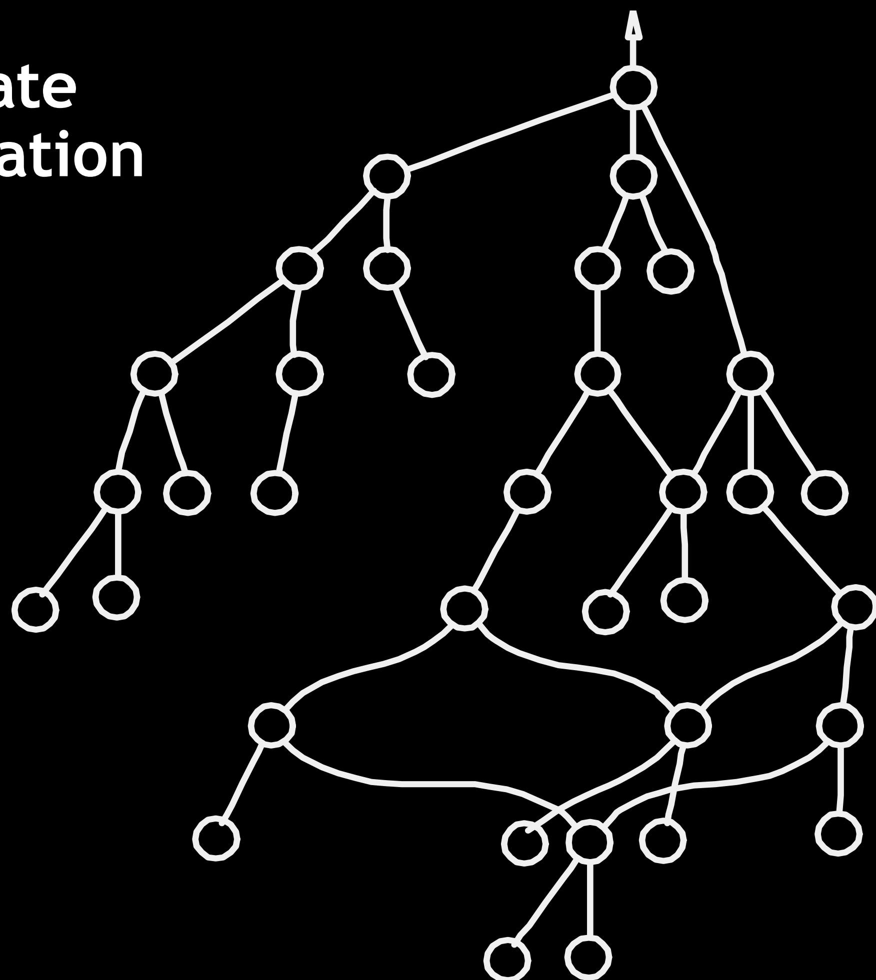
Pass Split Example (1 of 5)

Source code

```
// bowling pin, based on RenderMan bowling pin
surface shader floatv
bowling_pin (texref pinbase, texref bruns, texref marks, floatv uv)
{
    // generate texture coordinates
    floatv uv_wrap = { uv[0], 10 * Pobj[1], 0, 1 };
    floatv uv_label = { 10 * Pobj[0], 10 * Pobj[1], 0, 1 };
    // texture transformation matrices
    matrix t_base = invert(translate(0, -7.5, 0) * scale(0.667, 15, 1));
    matrix t_bruns = invert(translate(-2.6, -2.8, 0) * scale(5.2, 5.2, 1));
    matrix t_marks = invert(translate(2.0, 7.5, 0) * scale(4, -15, 1));
    // per-vertex scalar used to select front half of pin
    float front = select(Pobj[2] >= 0, 1, 0);
    // lookup texture colors
    floatv Base = texture(pinbase, t_base * uv_wrap);
    floatv Bruns = front * texture(bruns, t_bruns * uv_label);
    floatv Marks = texture(marks, t_marks * uv_wrap);
    // compute lighting
    floatv Cd = lightmodel_diffuse({ 0.4, 0.4, 0.4, 1 }, { 0.5, 0.5, 0.5, 1 });
    floatv Cs = lightmodel_specular({ 0.35, 0.35, 0.35, 1 }, { 0, 0, 0, 0 }, 20);
    // compute surface color
    return (Bruns over Base) * (Marks * Cd) + Cs;
}
```

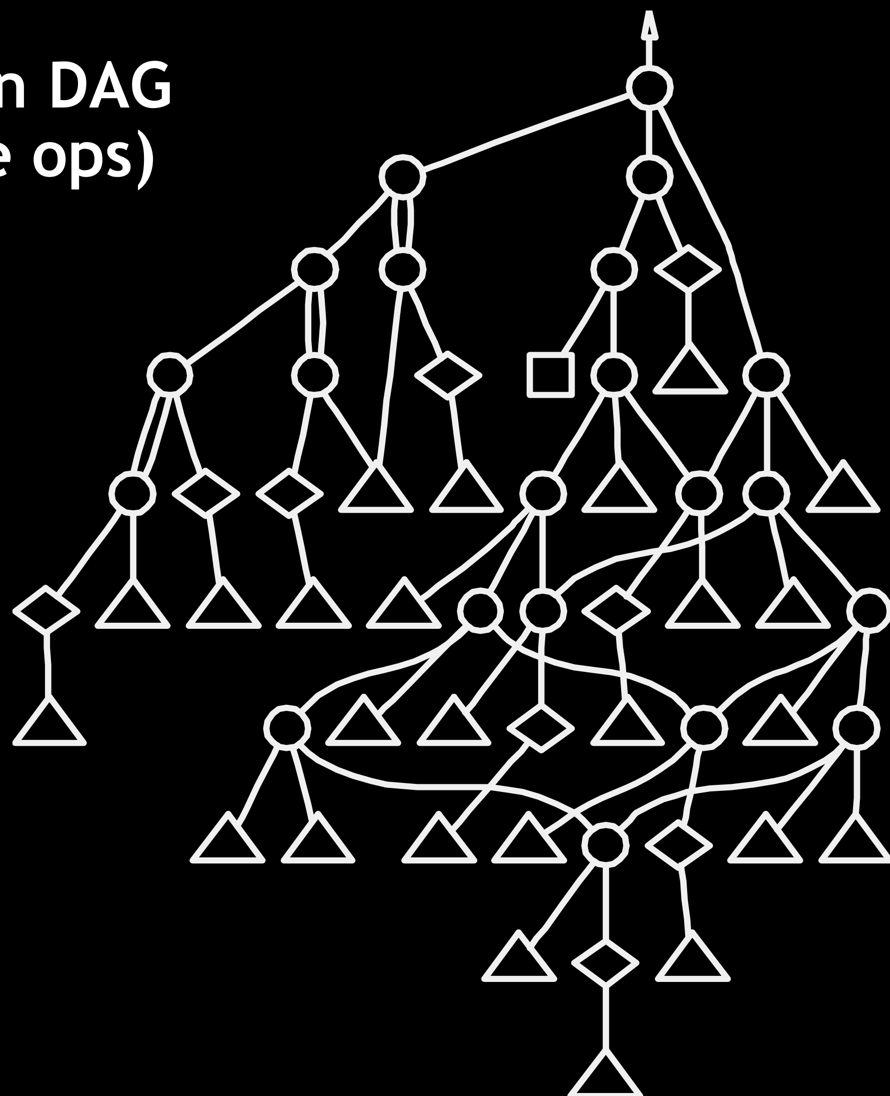
Pass Split Example (2 of 5)

Intermediate
Representation



Pass Split Example (3 of 5)

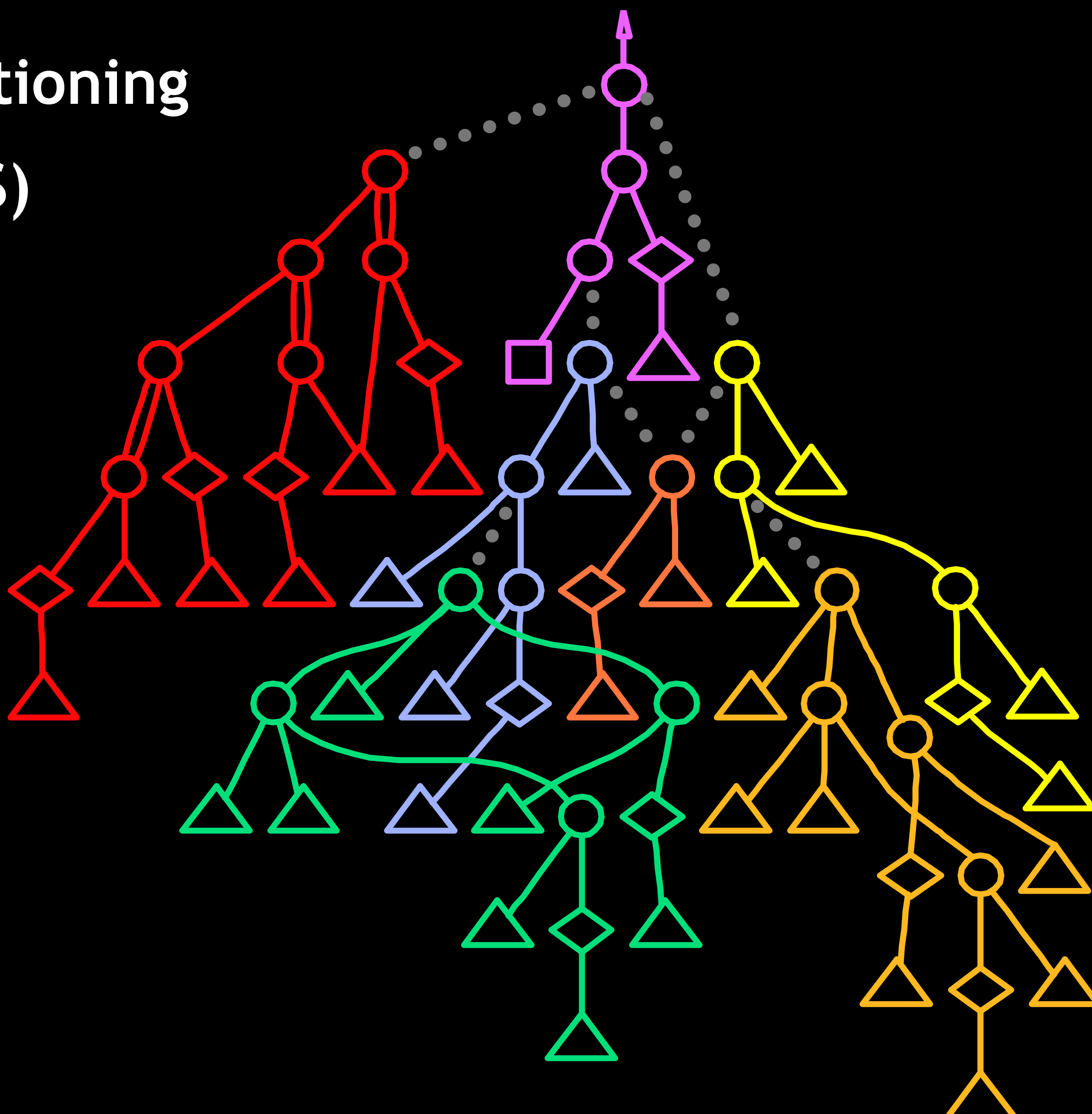
Instruction DAG
(hardware ops)



Pass Split Example (4 of 5)

Pass partitioning

(using RDS)



Pass Split Example (5 of 5)

Code generation

```
; pass 0
texcrd r0.rgb, t0
tex2d r1.rgb, t1
tex2d r2.rgb, t2
tex2d r3.rgb, t3
tex2d r4.rgb, t4
mul r1.rgb, r0, r1
mul r1.a, r0.r, r1
mul r2.rgb, r0,r2
mul r2.a, r0.r, r2
mul r0.rgb, r0, r3
mul r0.a, r0.g, r3
mad r0, r0, r4, r0
mad r0, r2, r0, r2
```

```
; pass 1
texcrd r0.rgb, t0
tex2d r1.rgb, t1
mul r0.rgb, r0, r1
mul r0.a, r0, r1
```

```
; pass 2
texcrd r0.rgb, t0
texcrd r2.rgb, t2
texcrd r3.rgb, t3
texcrd r5.rgb, t5
tex2d r1.rgb, t1
tex2d r4.rgb, t4
mul r0.rgb, r0, r1
mul r0.a, r0, r1
mul r1.rgb, r3, r4
mul r1.a, r3, r4
mad r1, r2, r1, r5
texcrd r0.rgb, r0
texcrd r1.rgb, r1
mad r0, r0, r0, r1
```

```
; pass 3
texcrd r0.rgb, t0
texcrd r1.rgb, t1
texcrd r2.rgb, t2
tex2d r4.rgb, t4
tex2d r3.rgb, t3
tex2d r5.rgb, t5
mul r2.rgb, r2, r3
mul r2.a, r2, r3
mad r1, r1, r2, r5
mad r0, r0, r4, r1
```

```
; pass 4
texcrd r0.rgb, t0
texcrd r2.rgb, t2
texcrd r3.rgb, t3
texcrd r5.rgb, t5
tex2d r1.rgb, t1
tex2d r4.rgb, t4
mul r0.rgb, r0, r1
mul r0.a, r0, r1
mul r1.rgb, r3, r4
mul r1.a, r3, r4
texcrd r0.rgb, r0
texcrd r1.rgb, r1
mad r0, r0, r0, r1
```

```
; pass 5
texcrd r0.rgb, t0
texcrd r1.rgb, t1
texcrd r2.rgb, t2
tex2d r4.rgb, t4
tex2d r3.rgb, t3
tex2d r5.rgb, t5
mul r2.rgb, r2, r3
mul r2.a, r2, r3
mad r1, r1, r2, r5
mad  r0, r0, r4, r1
```

```
; pass 6
texcrd r1.rgb, t1
tex2d r2.rgb, t2
tex2d r0.rgb, t0
tex2d r3.rgb, t3
tex2d r4.rgb, t4
add r1.rgb, r1, r3
add r1.a, r1, r3
mul r0, r0, r1
mad r0, r2, r0, r4
mad r0, r2, r0, r4
```

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:

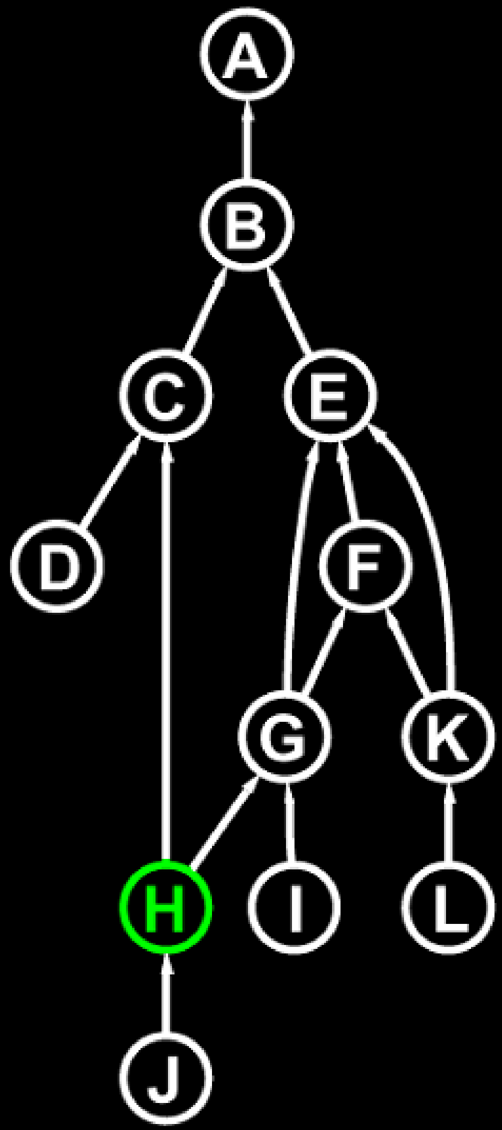
- Given a DAG and a cost model, find a valid partition with the lowest cost

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
- multiple projected texture lights

Hardware

- ATI Radeon 9700 (R300)

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

- Back ends for DX9-class GPUs
- General multipass support

Comparison to other shading languages

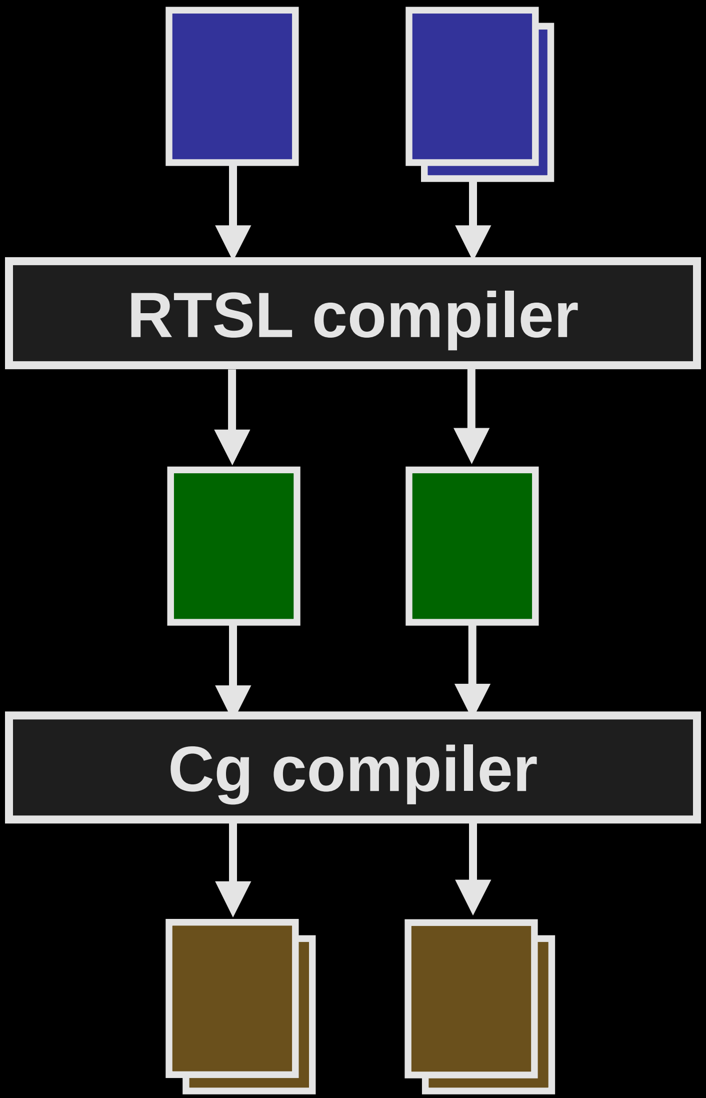
Comparison To Other Languages

Compared to Cg:

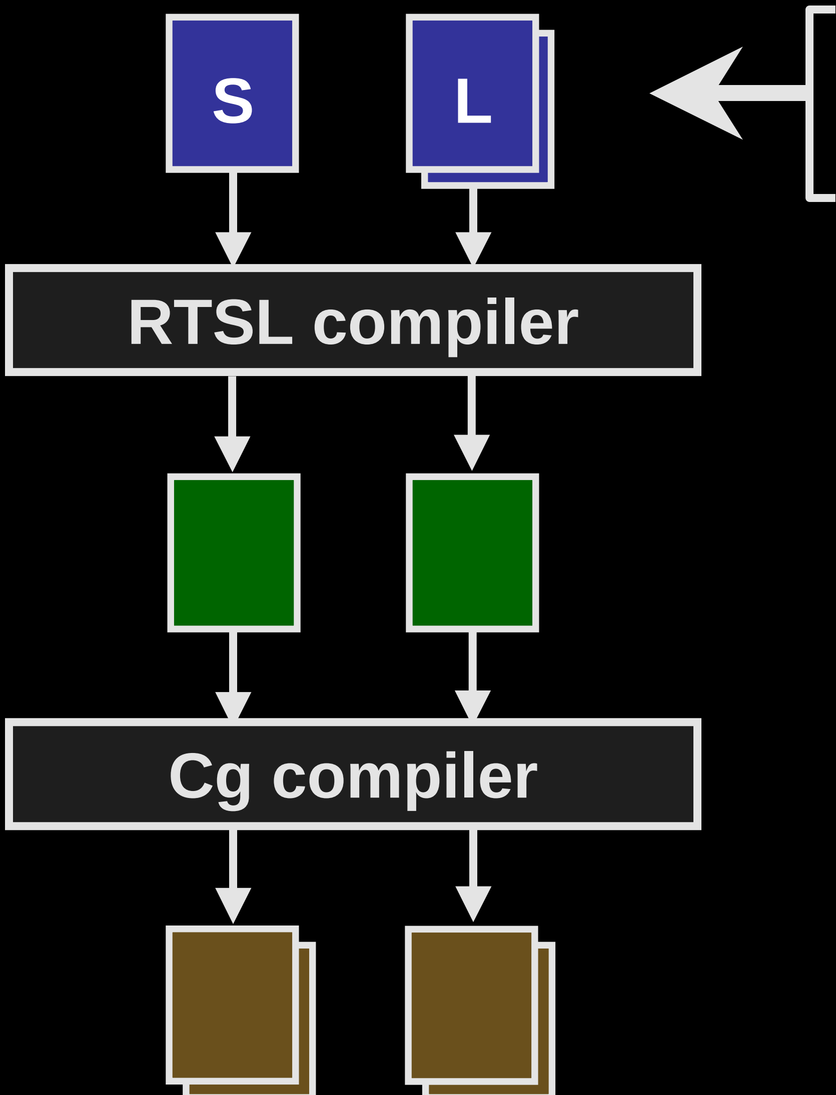
1. Surface and light shaders
2. Single pipeline program split by computation frequency
3. Hides multipass

RTSL provides higher-level abstraction than Cg

RTSL to Cg

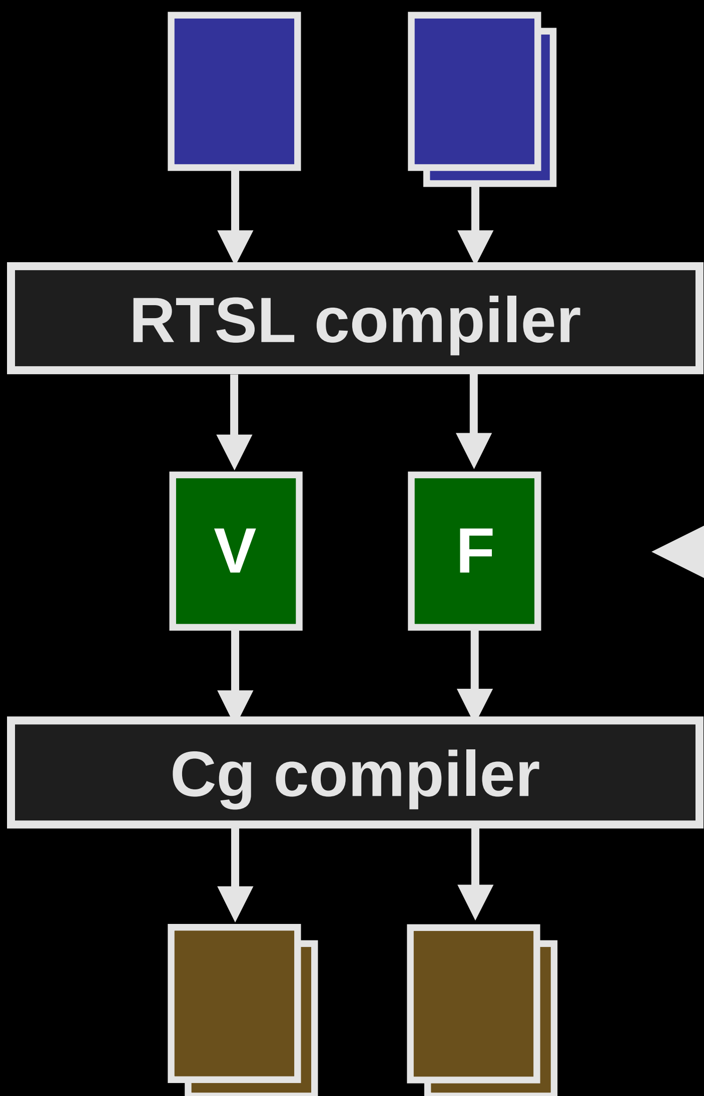


RTSL to Cg



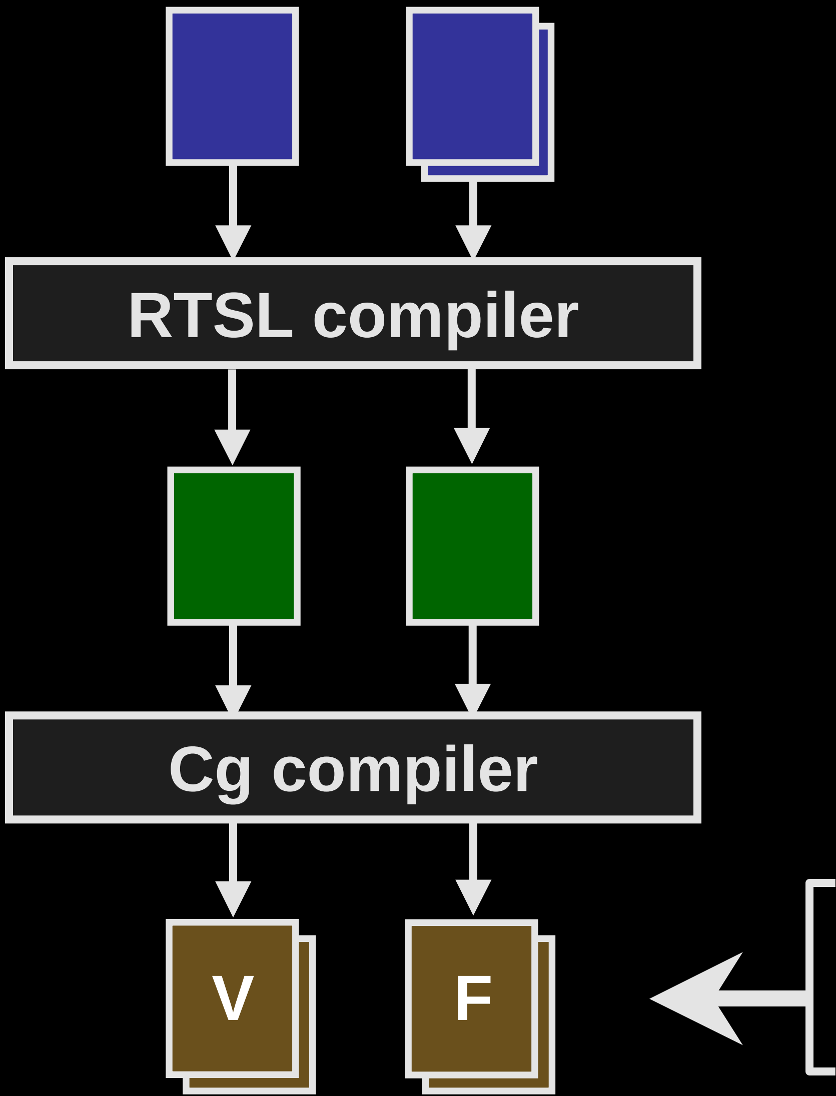
- Surface shader
- Light shaders

RTSL to Cg



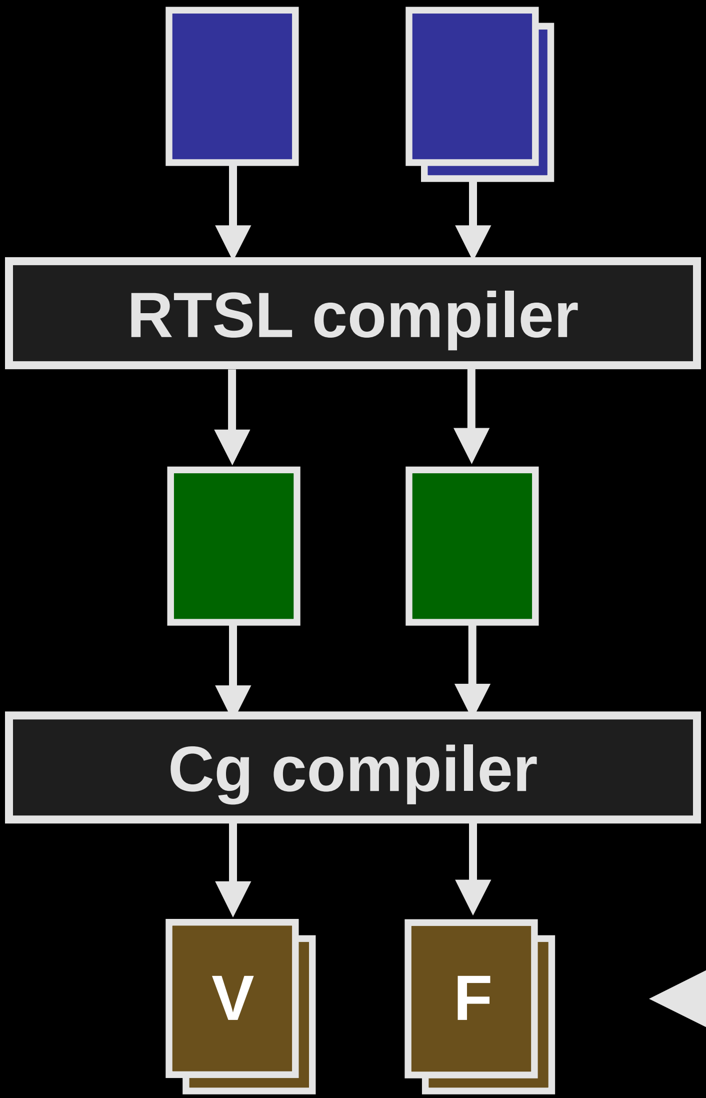
- Cg vertex program
- Cg fragment program

RTSL to Cg (OpenGL)



- GL vertex program
- GL fragment program

RTSL to Cg (DirectX)



- DirectX vertex shader
- DirectX pixel shader

Summary

Current system:

- Next-generation 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/