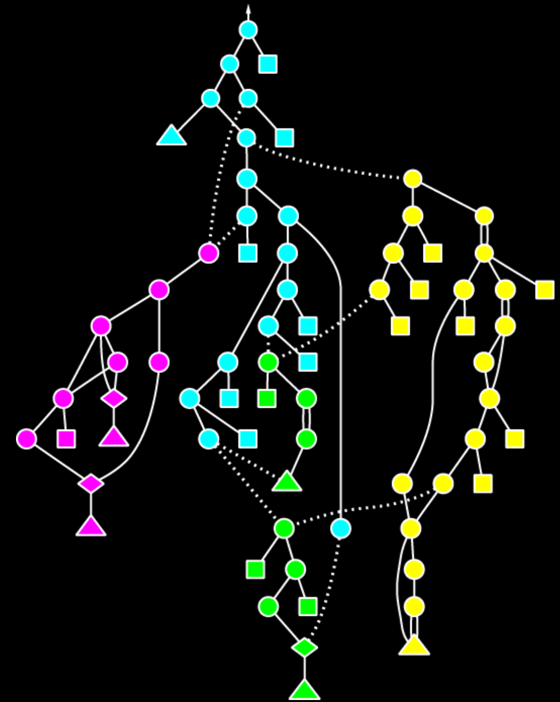
Efficient Partitioning of Fragment Shaders for Programmable Graphics Hardware

Eric Chan
Ren Ng
Pradeep Sen
Kekoa Proudfoot
Pat Hanrahan

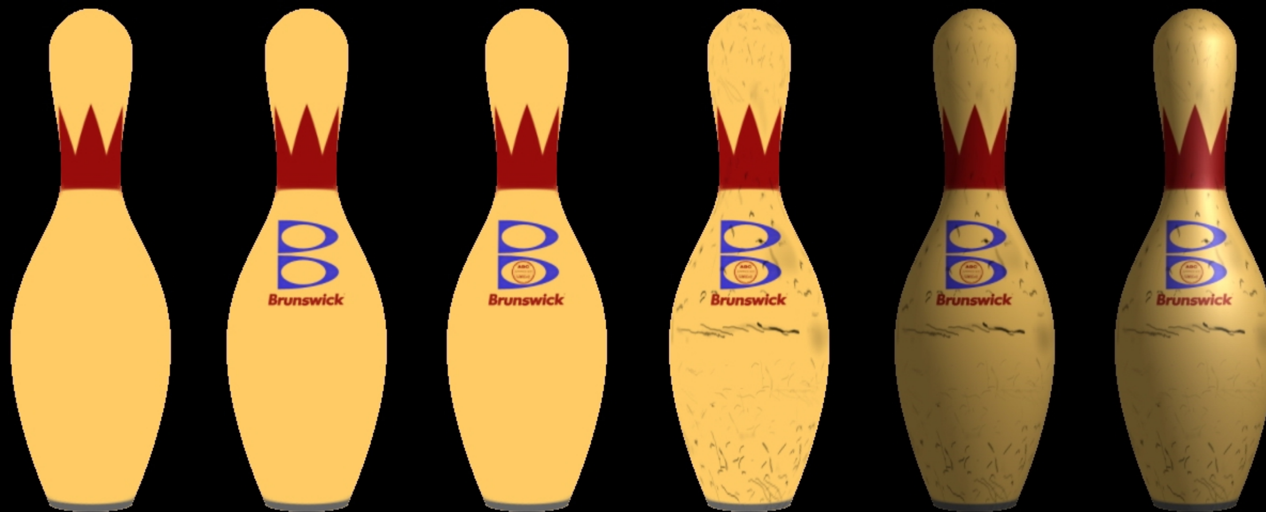
Computer Graphics Lab
Stanford University



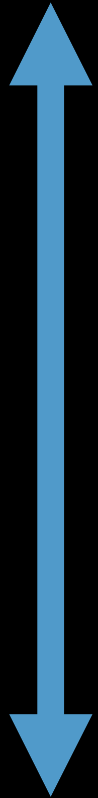
Real-Time Shading Languages

Common goal:

- Make programmable hardware features accessible via high-level language
- Easy to write large and complex shaders



Shading Language Differences



Higher-level languages

- Example: RenderMan, Stanford shading language
- Surface and light shaders
- Virtualizes hardware resources

Lower-level languages

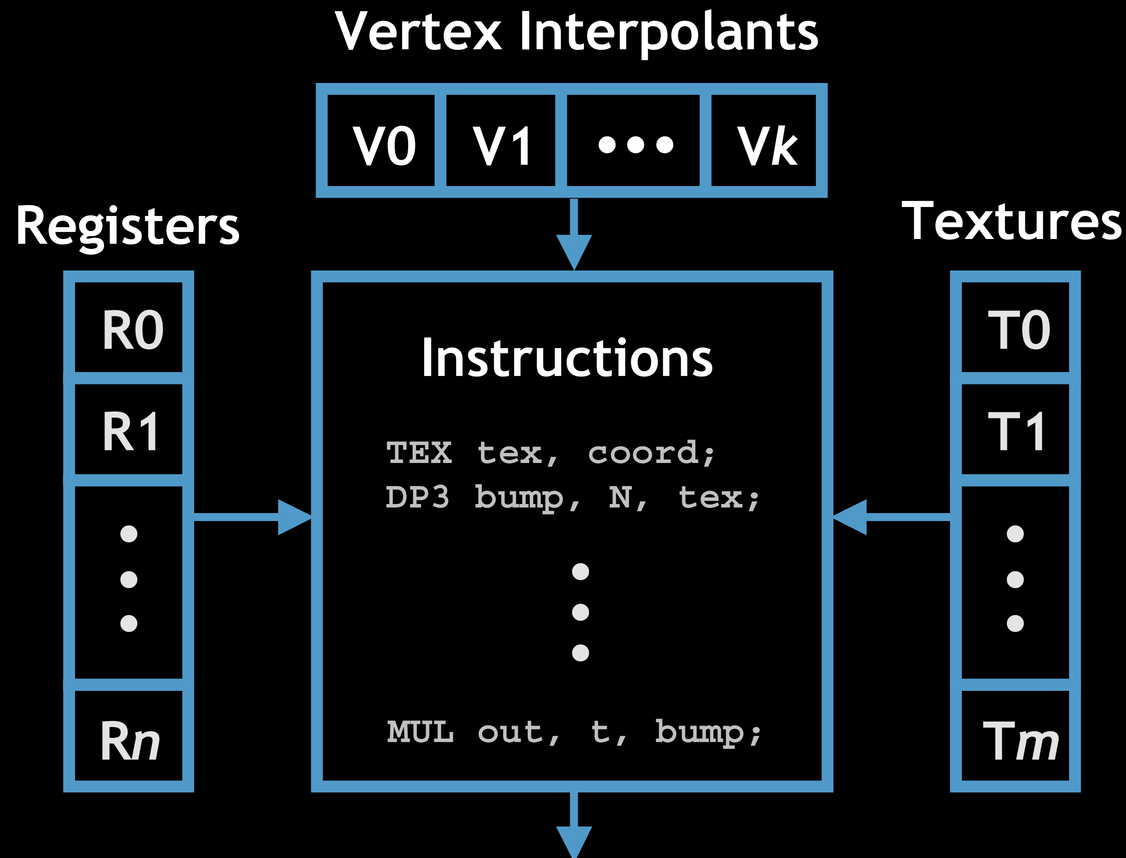
- Examples: NVIDIA's Cg, 3D Labs' OpenGL 2.0 proposal
- Separate vertex and fragment programs
- No virtualization of hardware resources

Key design question:

- How are hardware resource limits handled?

Resource Limits Are Troublesome

- Every GPU has resource limits
- Sufficiently large shaders can't be mapped to one pass!



Virtualization Is Key

"... the upcoming high level languages MUST NOT have fixed, queried resource limits if they are going to reach their full potential ..."

"... drivers must have the right and responsibility to multipass arbitrarily complex inputs to hardware with smaller limits."

— John Carmack
id Software

June 2002

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**

Problem:

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

Previous Approaches

Peercy et al. used pattern-matching on shade trees:

- Handles arbitrarily large shaders for *non-programmable* hardware
- But, doesn't work well for programmable hardware!

Proudfoot et al. used custom compiler back ends:

- Targets programmable hardware
- Uses hardware resources efficiently
- But, doesn't split shaders into multiple passes

Contributions

"Recursive Dominator Split" (RDS) algorithm:

1. Efficiently targets *programmable* graphics hardware
2. Supports arbitrarily large shaders
3. Supports hardware with different constraints
4. Supports hardware with different performance characteristics
5. Integrates smoothly with a real shading system

Outline

1. Stanford shading system overview
2. Problem statement and assumptions
3. Algorithm: Recursive Dominator Split (RDS)
4. Results

Shading System Overview (1/5)

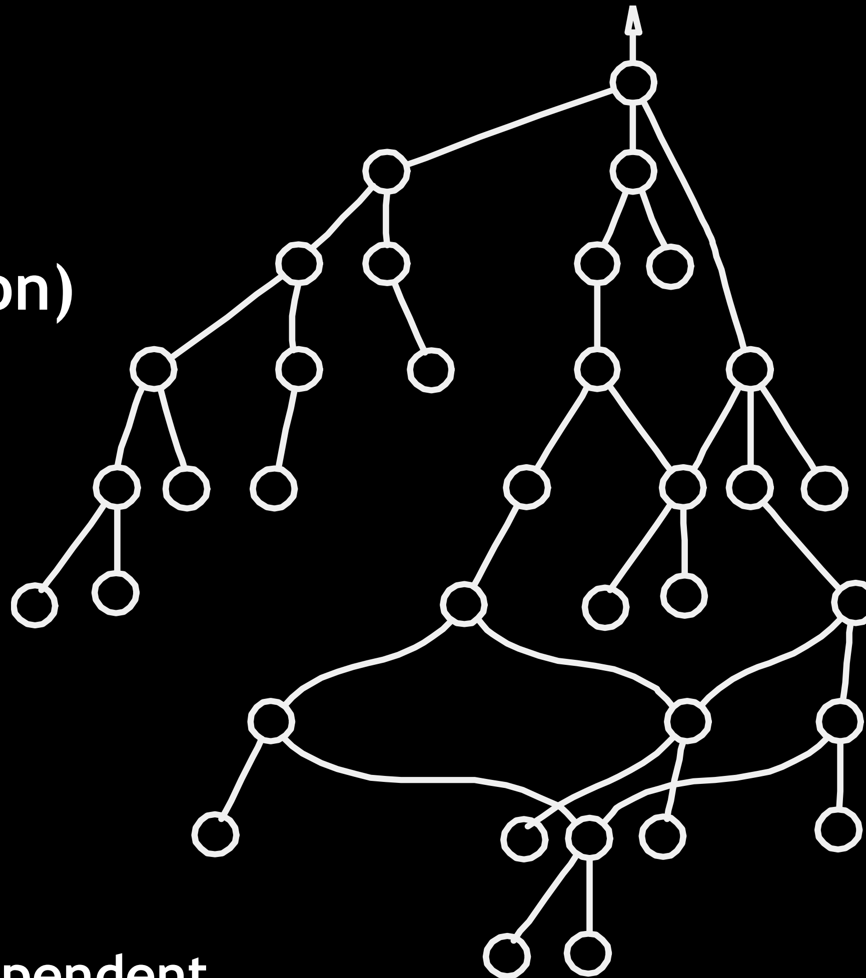
Source Code

```
// 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;
}
```

Shading System Overview (2/5)

Intermediate Representation

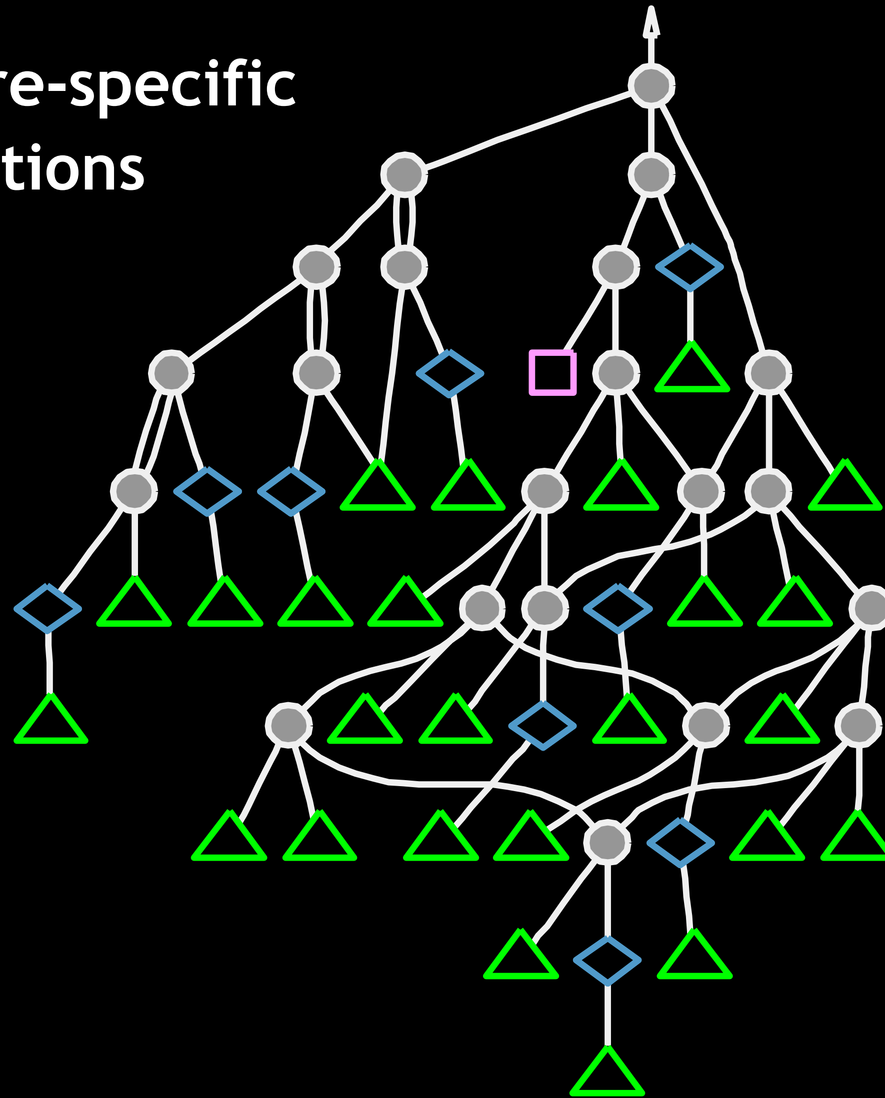
(fragment portion)



○ Hardware-independent, high-level language operator

Shading System Overview (3/5)

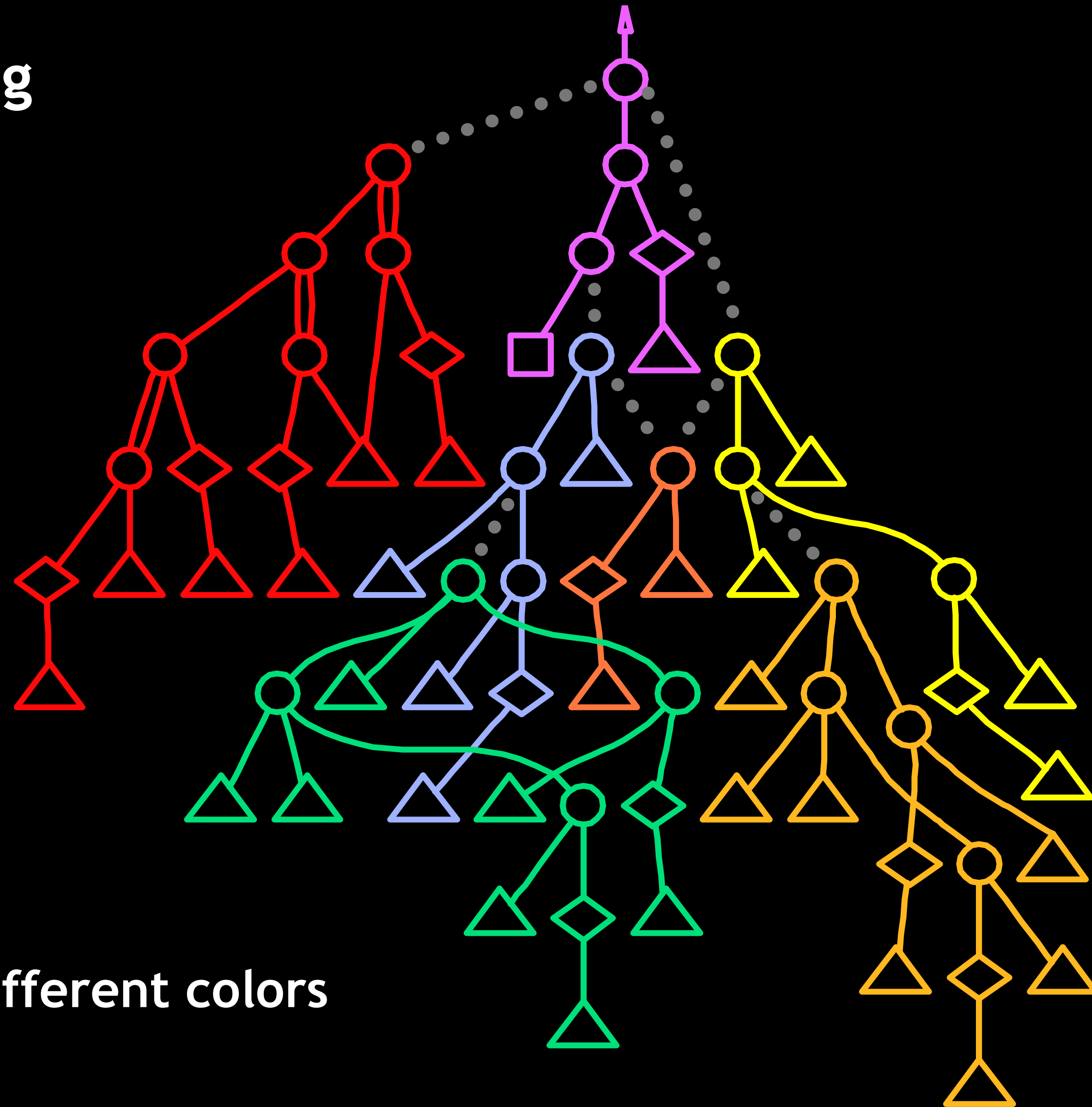
DAG of hardware-specific fragment operations



- Arithmetic op
- Texture op
- Vertex input
- Constant

Shading System Overview (4/5)

Pass partitioning using RDS



Passes shown as different colors

Shading System Overview (5/5)

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
```

Hardware-specific object code

Multipass Partitioning Problem (MPP)

Definitions:

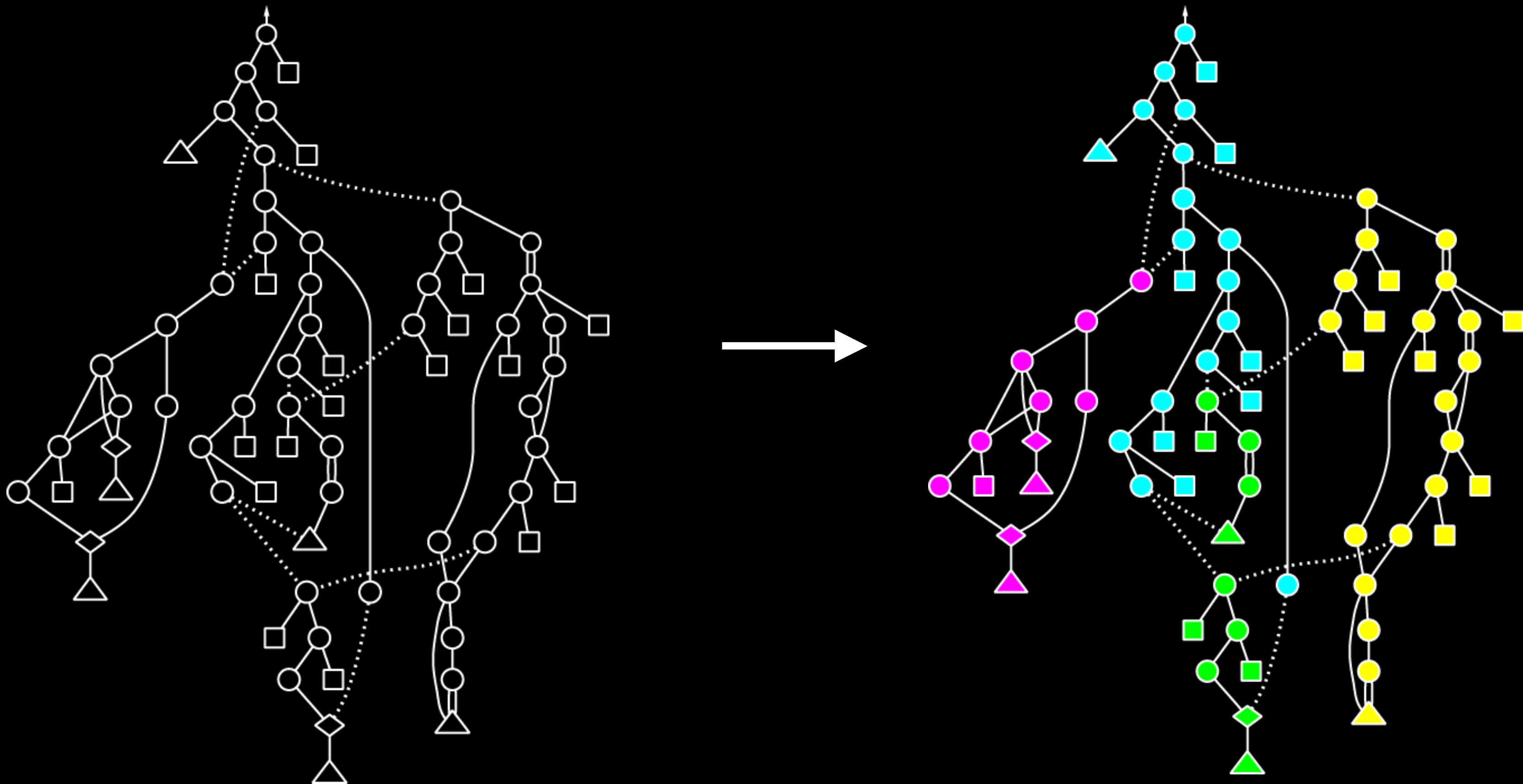
- Each way of splitting a shader is a partition.
- A cost model evaluates the cost of partitions.
- A partition is valid if each pass satisfies all constraints.

Task:

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

MPP Is Hard

- Abstract problem: graph partitioning
- Falls into class of "NP optimization" problems



Assumptions

1. We only consider fragment shaders
2. Shaders are represented as a single directed acyclic graph (DAG)
3. Only 1 output value per pass
4. Hardware preserves intermediate values at full precision
5. Hardware provides base set of resources to support multipass rendering

Recursive Dominator Split (RDS)

Overview:

	Objectives	Techniques
1.	Minimize number of passes	Greedy bottom-up merging
2.	Identify cases when we can avoid save / recompute decisions	Top-down traversal through a *partial dominator tree*
3.	Make save / recompute decisions when necessary	Search over nodes that are *multiply-referenced*

Greedy Merging Intuition

Assumption:

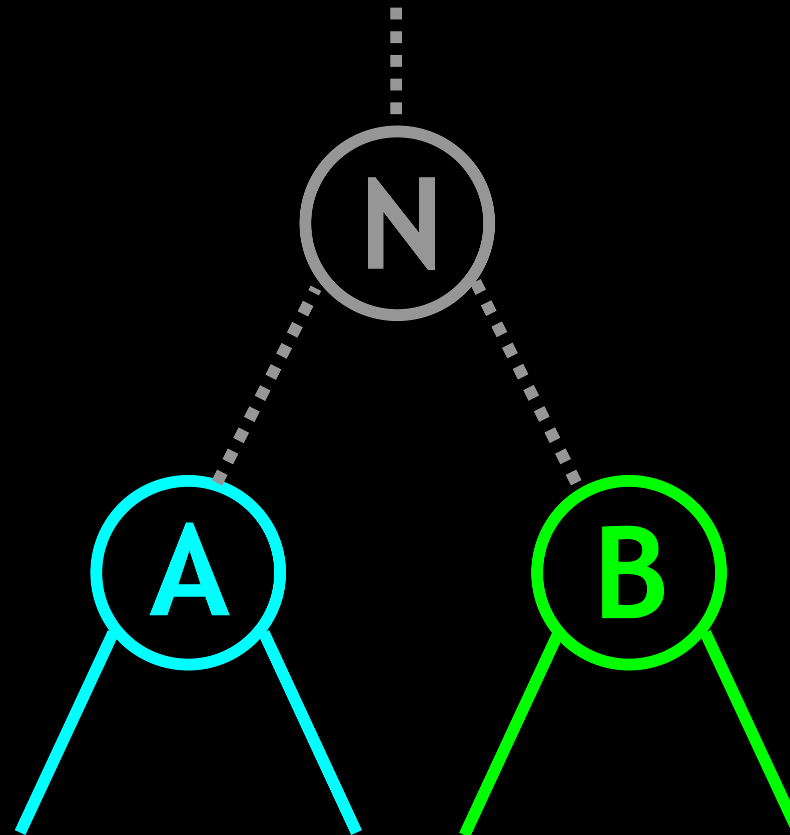
- Extra passes are expensive (e.g. save to texture)
- This overhead dominates the overall cost

Idea:

- Use *minimum passes* to approximate *minimum cost*
- Merge as many nodes into as few passes as possible
- Proceed bottom-up because of dependency on children

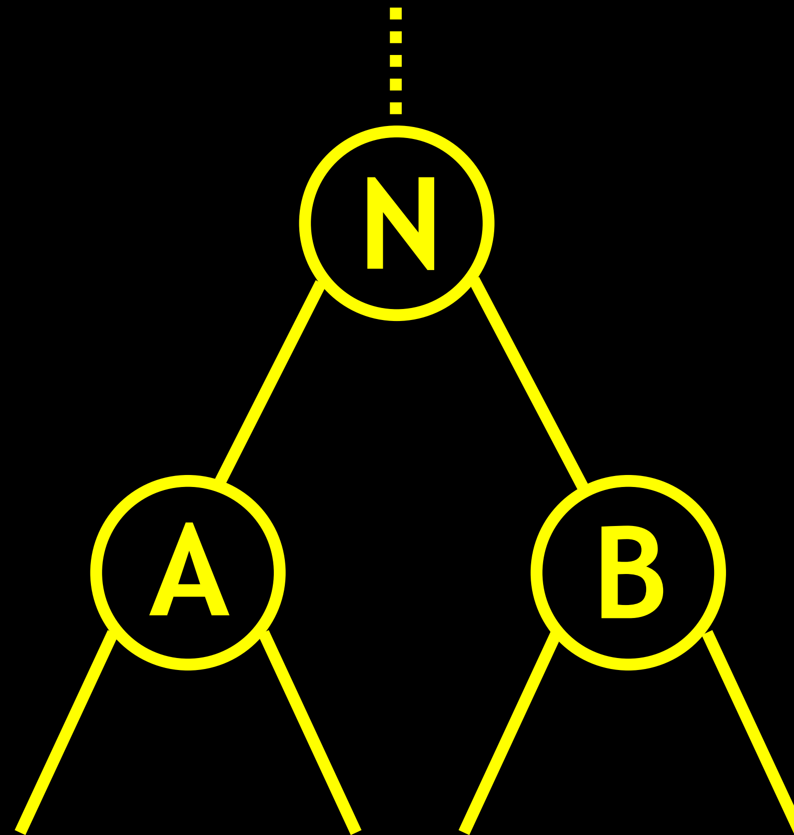
Merging Example: 2 kids

Node N with kids A and B already assigned to passes



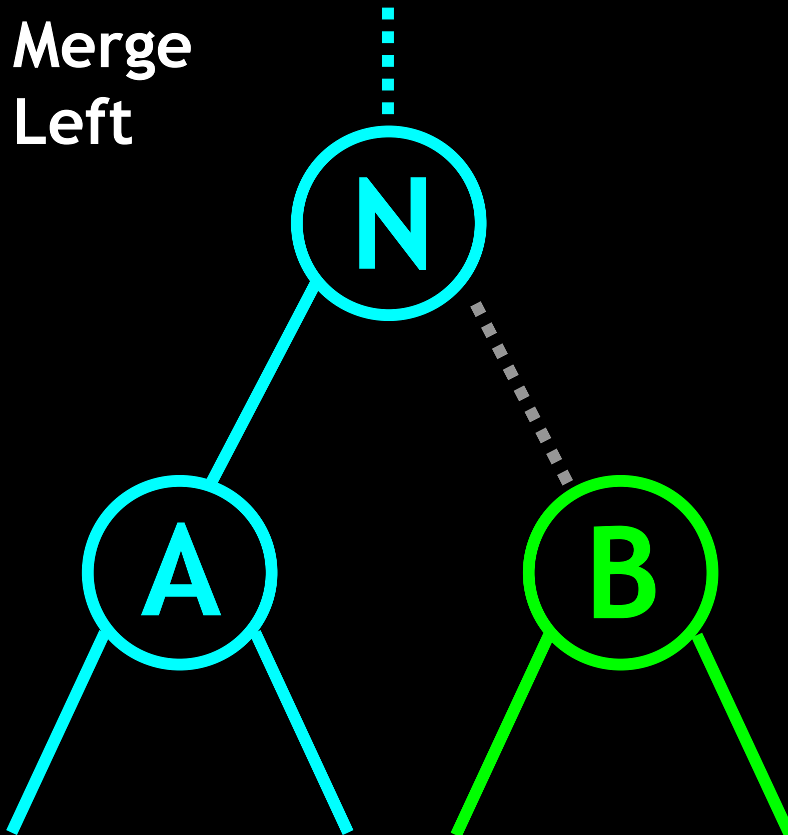
Merge All (1 pass)

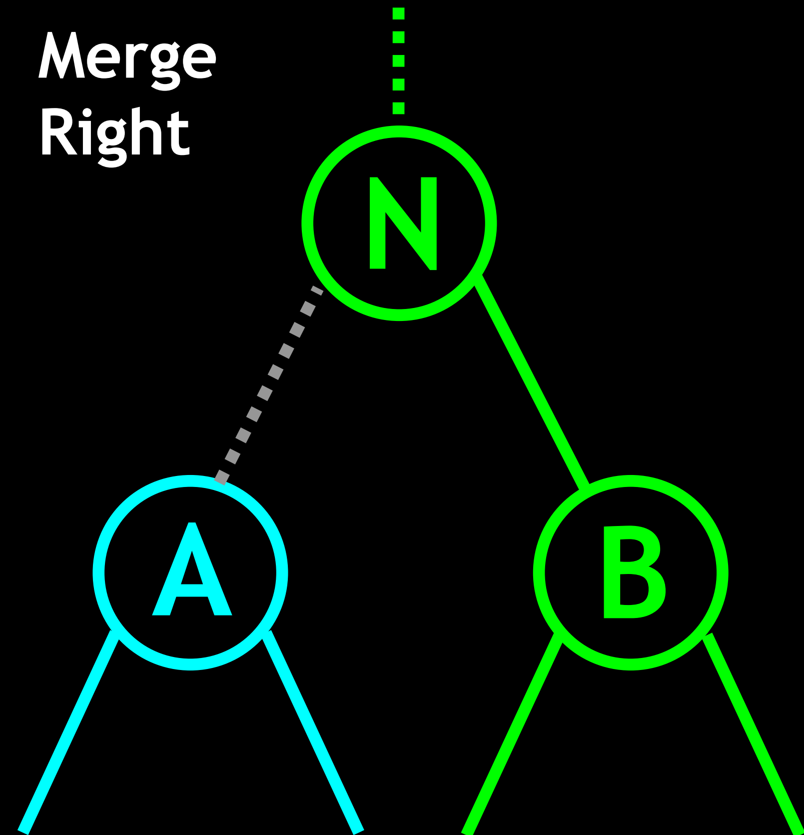
First, try to place all nodes in the same pass



Merge Left / Right (2 passes)

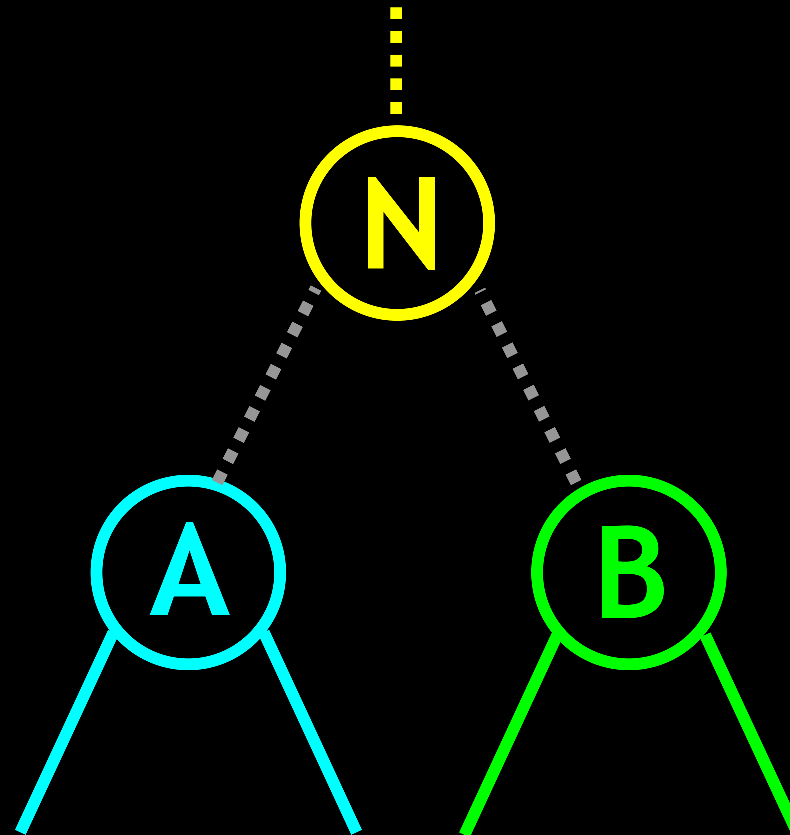
If that doesn't work, try Merge Left and Merge Right





Merge None (3 passes)

If merge not possible, must start a new pass at N



Merging Decisions

What if both Merge Left and Merge Right are possible?

Intuition:

- Pick the merge most likely to require fewer passes later in the partitioning process

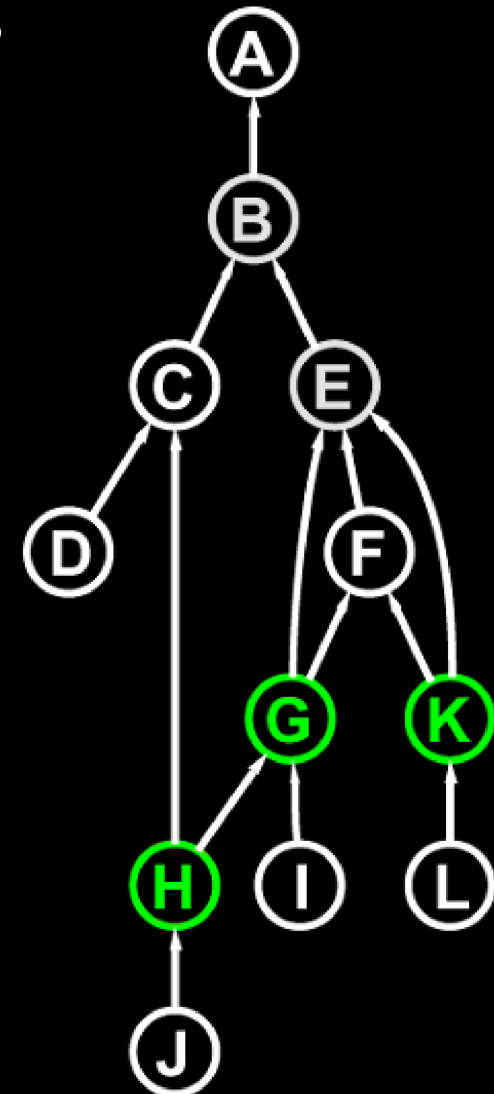
Heuristic:

- Pick the merge that consumes the *fewest* resources
- Break resource ties arbitrarily

Save vs. Recompute

What about multiply-referenced nodes?

- A <u>multiply-referenced</u> (MR) node has two or more parents
- A MR node can be
  1. merged
  2. saved in a separate pass
  3. recomputed (duplicated)
- Save has a cost (extra pass)
- Recompute has a cost (extra ops)



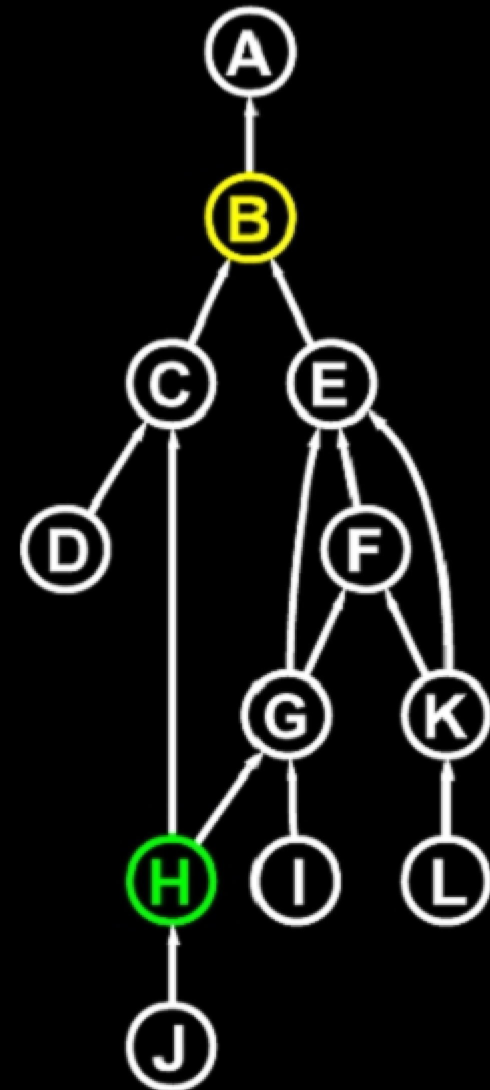
Immediate Dominators

Concept borrowed from compilers:

- The immediate dominator B of a node A is the "closest" node to A such that all paths from the root to A go through B

Example:

- idom(H) = B



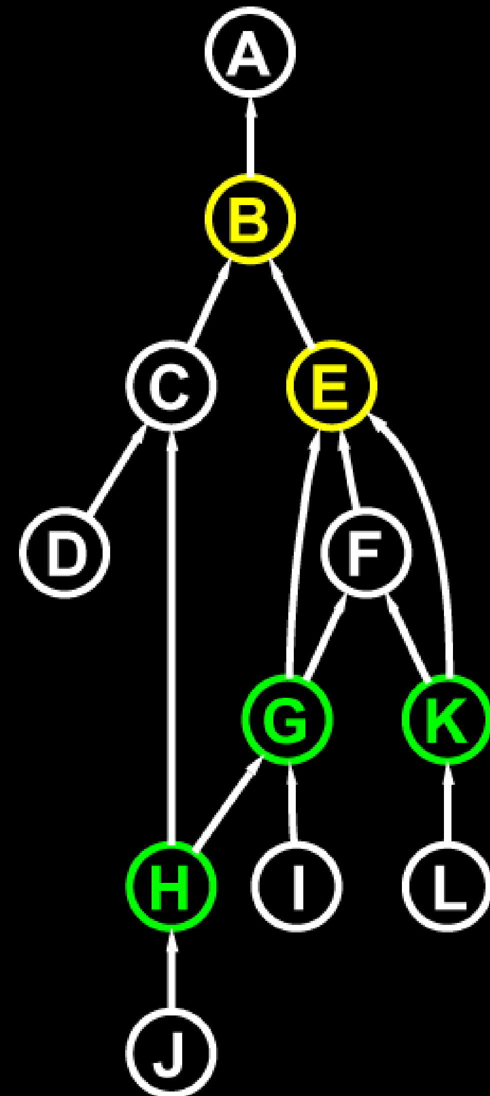
More Immediate Dominators

Concept borrowed from compilers:

- The immediate dominator B of a node A is the “closest” node to A such that all paths from the root to A go through B

Examples:

- idom(H) = B
- idom(G) = E
- idom(K) = E



Using Immediate Dominators

Basic idea:

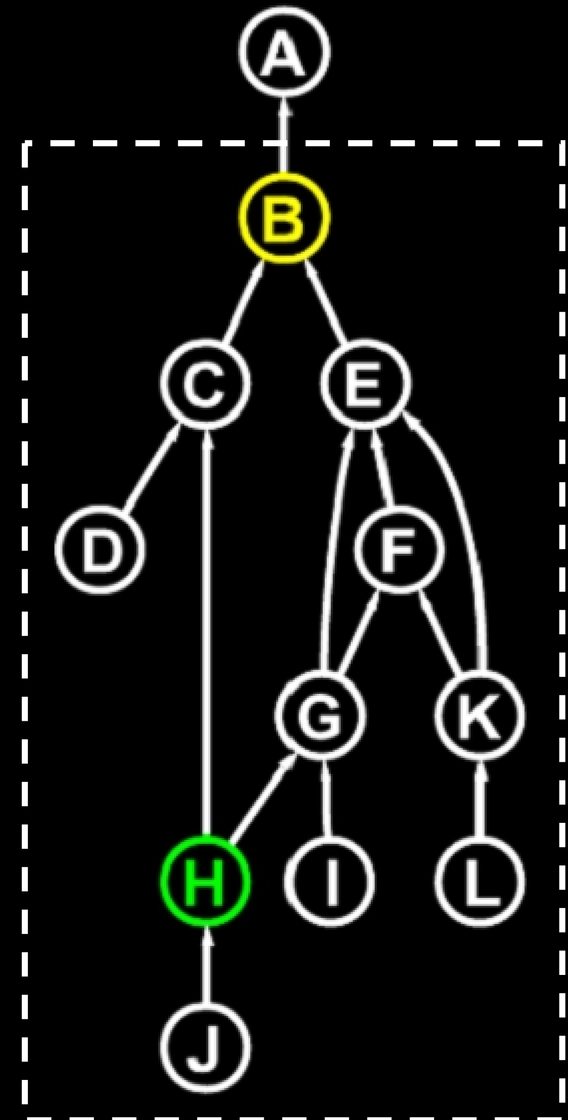
- Try to merge a multiply-referenced node with its immediate dominator

Why?

- Identifies "small enough" regions
- Avoids save / recompute cost!

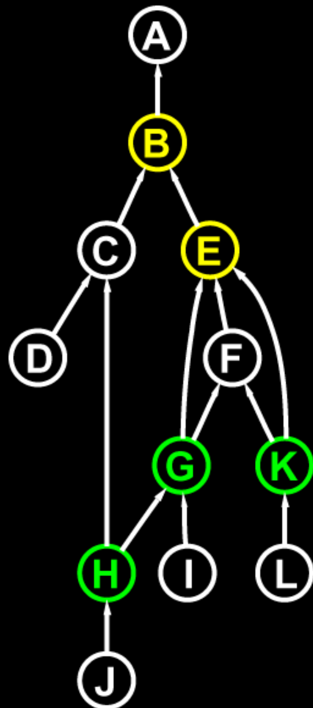
Mechanism:

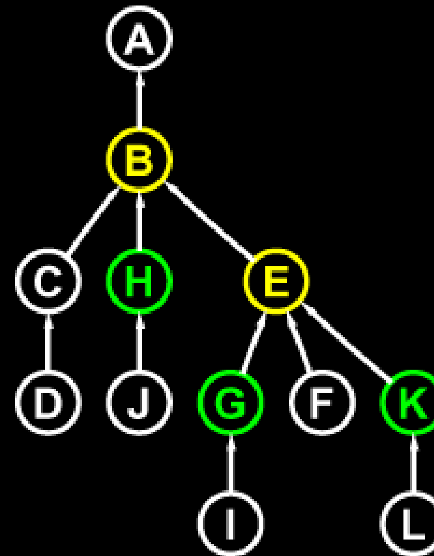
- Need a convenient data structure to identify these regions



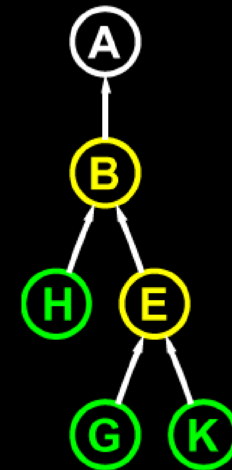
Partial Dominator Tree (PDT)

- In a dominator tree, the parent of each node is its immediate dominator.
- A partial dominator tree (PDT) is constructed from a normal dominator tree by keeping only the root, MR nodes, and their immediate dominators.





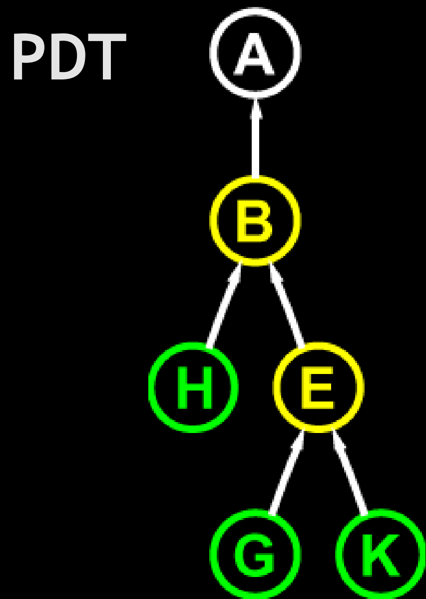
Dominator Tree

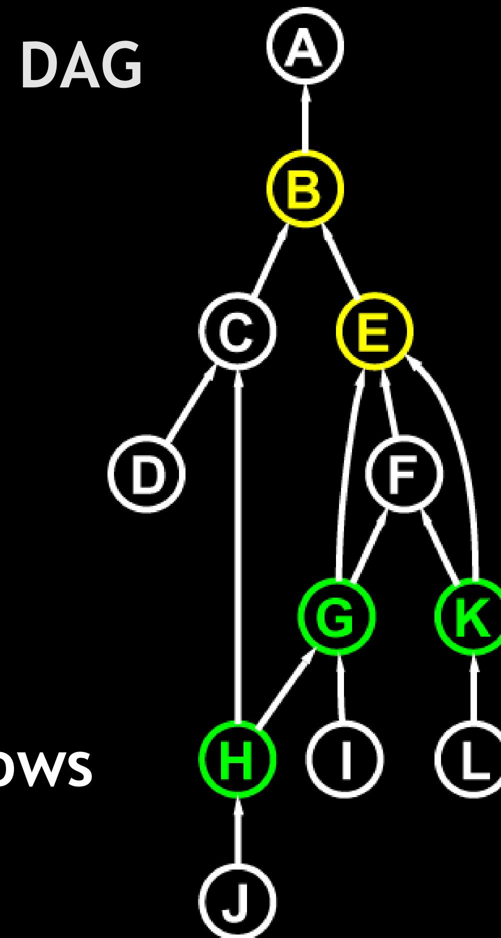


PDT

Recursive Subdivision Using PDT

Use a top-down traversal through the PDT

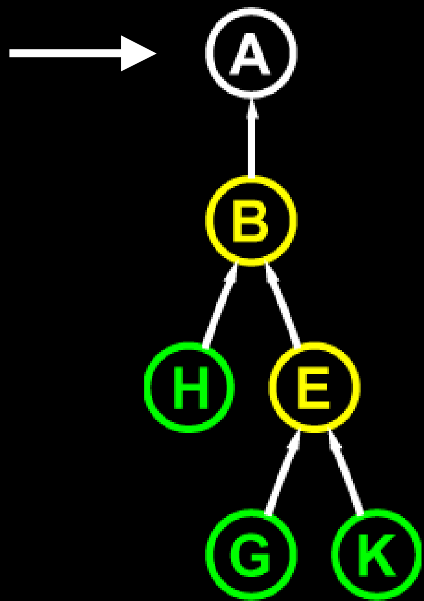


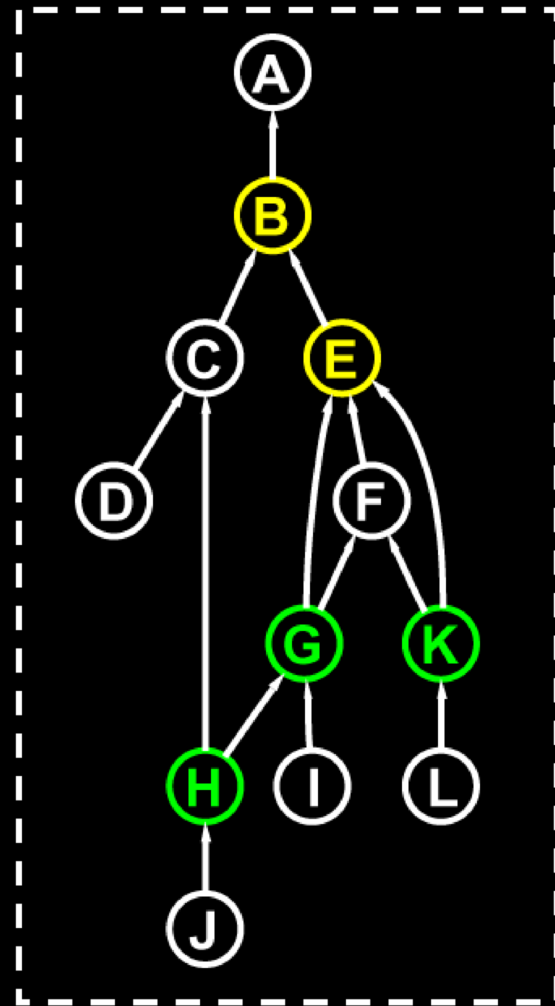


Example: suppose hardware allows only 10 operations per pass

Recursive Subdivision (1/3)

Start with node A. Does the whole DAG fit?

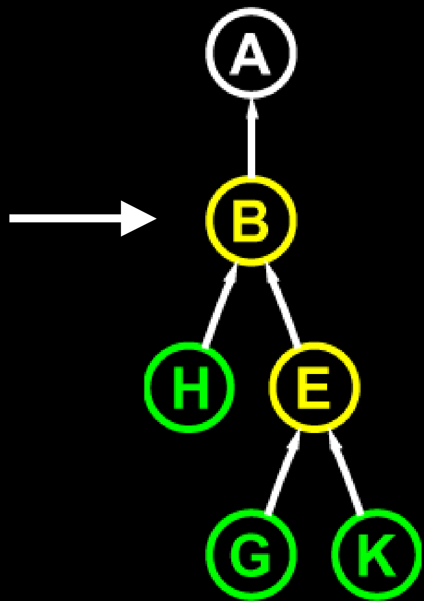




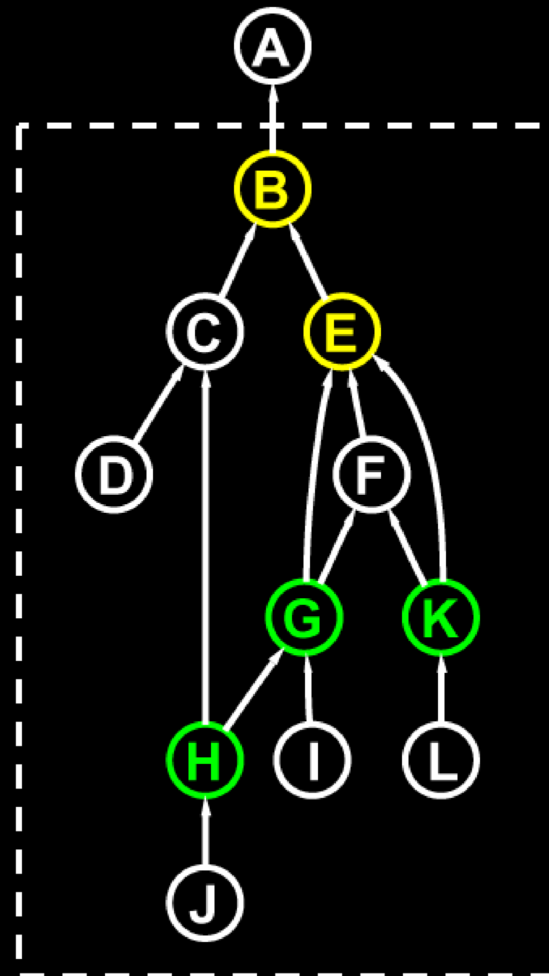
No — fails validity check

Recursive Subdivision (2/3)

Try B next. Does subregion(B) fit?

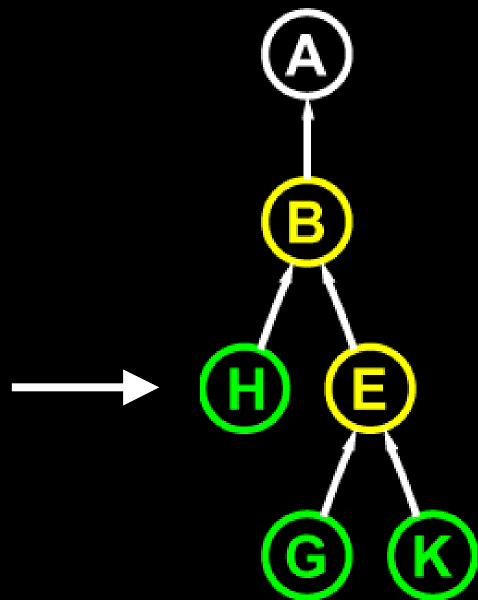


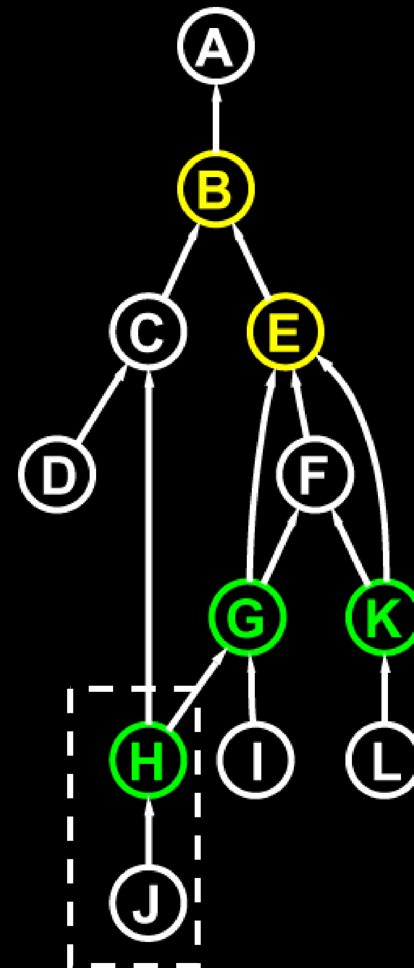
No — fails validity check



Recursive Subdivision (3/3)

Try H next (not E, because E depends on H).





Yes — 2 nodes fits under the limit.

Question: save or recompute at H?

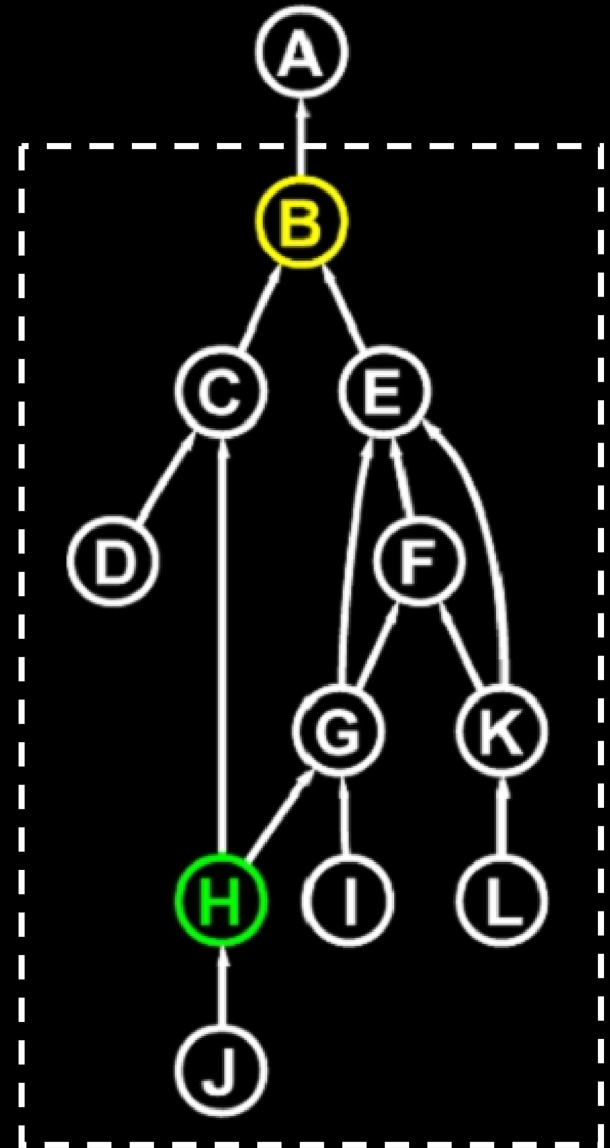
Save vs. Recompute

Problem:

- Not always possible to merge up to the immediate dominator

Options:

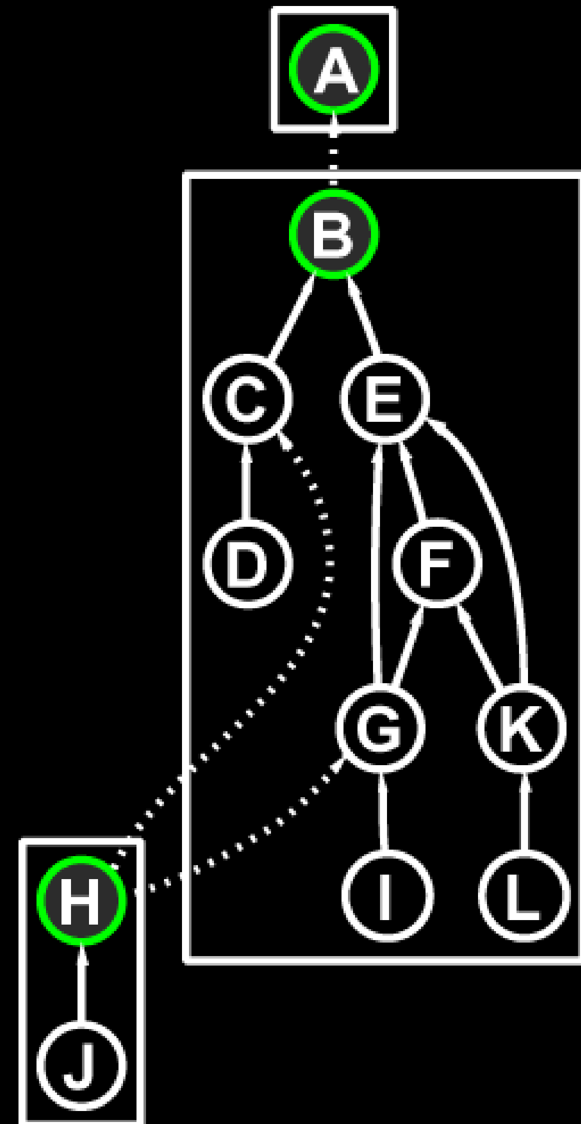
- Save H to its own pass
- Recompute H



Case 1: Save Subregion(H)

Partition #1:

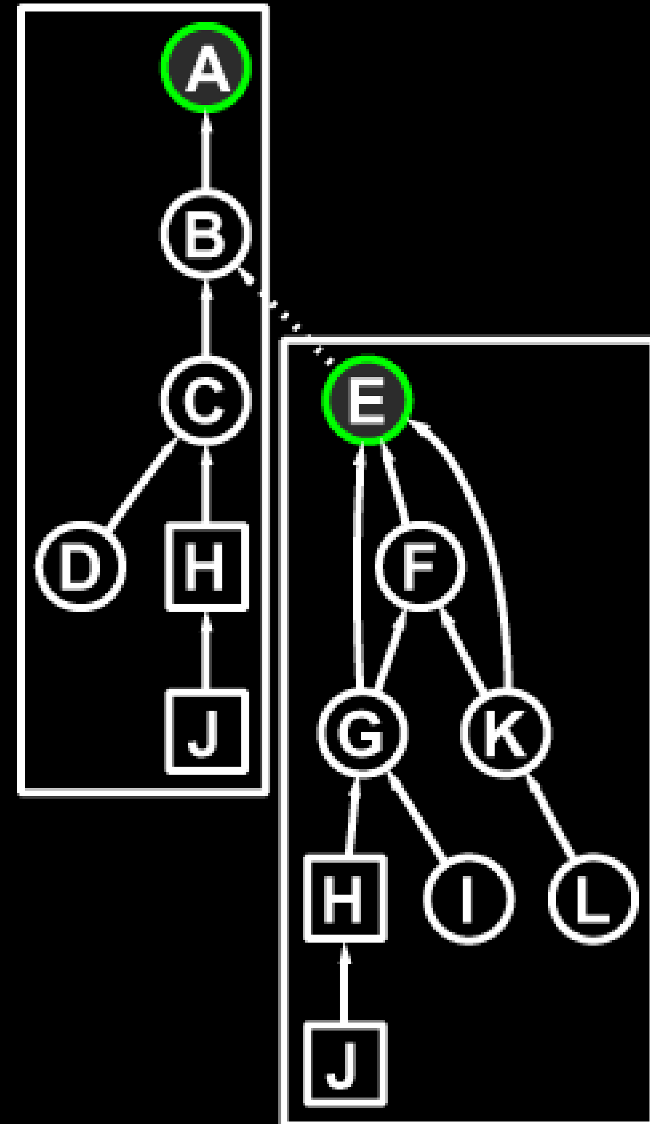
- 3 passes
- 14 operations
  (including 2 restores)



Case 2: Recompute Subregion(H)

Partition #2:

- 2 passes
- 15 operations
  (including 1 restore)



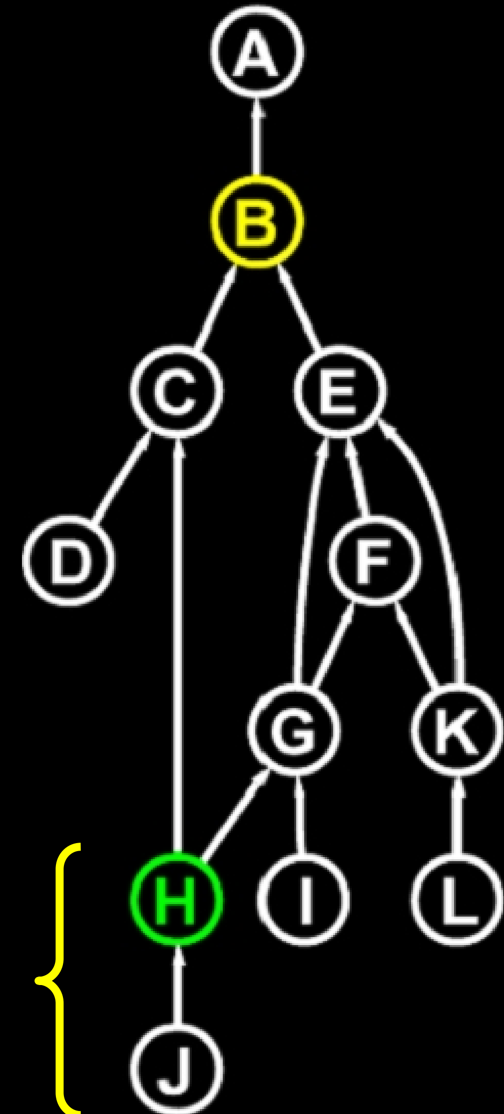
Recompute Heuristic

Intuition:

- Save if a subregion is almost "full"
- Recompute if a subregion is almost "empty"

Simple heuristic:

- Recompute iff the consumption of each resource is less than one-half the maximum allowed



Complexity

Basic analysis:

- Let N = the size of the DAG
- Greedy merging requires $O(N)$ time
- Assume validity check is $g(N)$

Algorithm has complexity $O(N \cdot g(N))$

Validity check can be expensive:

- Code generation, resource allocation, ...
- Recompile from scratch: $g(N) = O(N)$

Early Observations

Greedy merging is easy, fast, and works well

- Merge heuristic is good enough

Save vs. recompute is tricky

- Recompute heuristic is not good enough
- Hard to predict how a decision will affect result

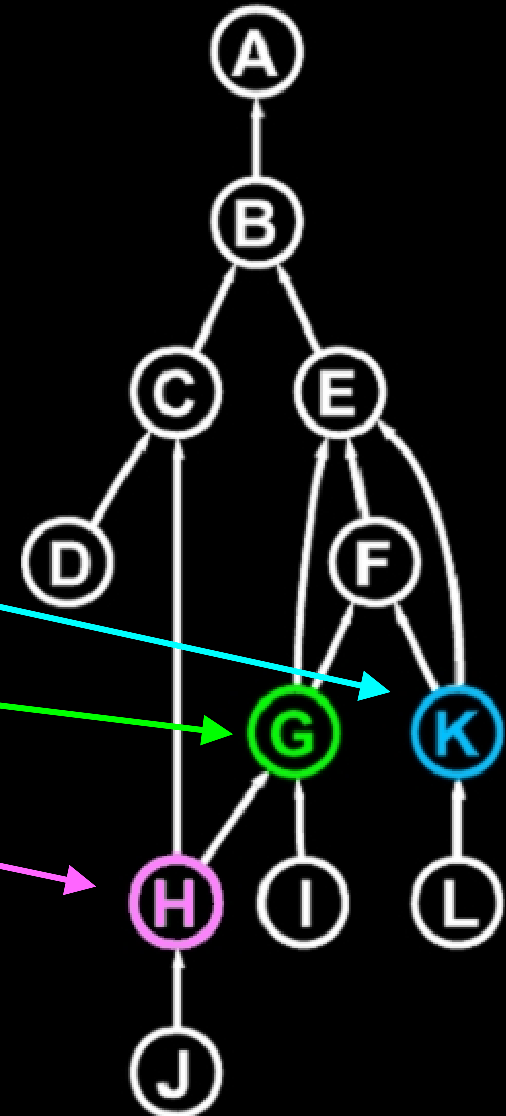
Another approach:

- Search over save / recompute decisions to obtain better results

Search Overview

Walk bottom-up through MR nodes:

- Decisions at visited nodes are fixed
- Try both save / recompute at current node
- Apply heuristic to unvisited nodes



Unvisited: use heuristic

Current MR node

Decision fixed
(recompute)

- Increases running time by a linear factor

Implementation

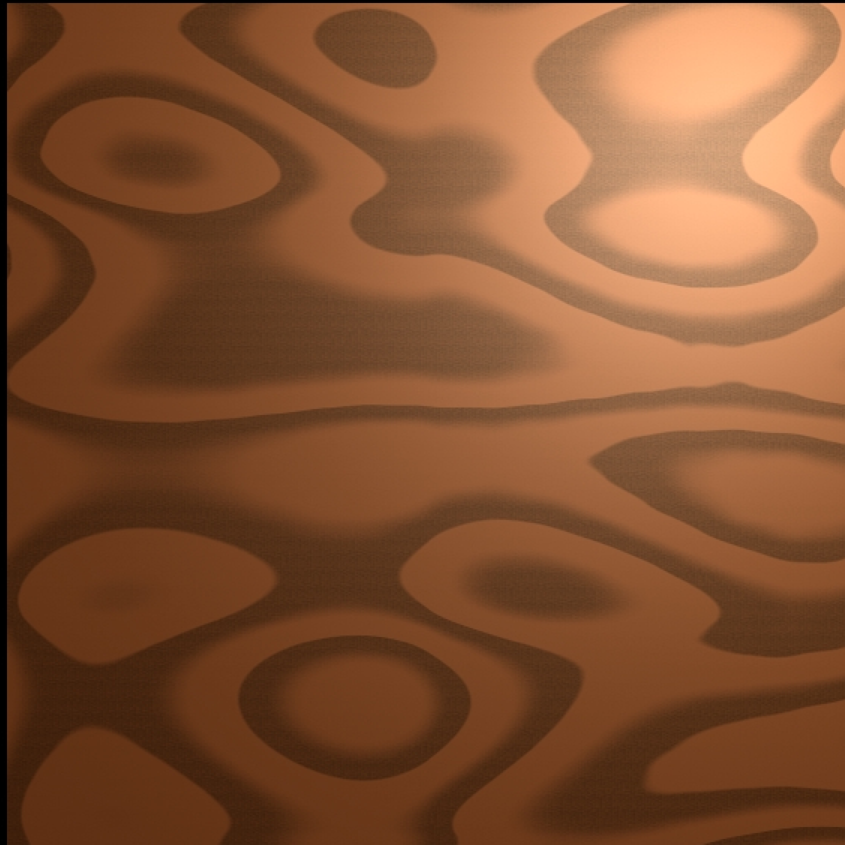
Compiler back ends for the Stanford shading system:

- ATI R200 (Radeon 8500)
- ATI R300 (Radeon 9700)
- NVIDIA NV30 (software driver)

new back ends: added since original paper

Images

Procedural wood surface (credit: Larry Gritz)



50 passes on Radeon 8500

7 passes on Radeon 9700

1 pass on NV30

Limited by instructions

Image generated using NV30 software driver

Images

Procedural flame shader (credit: Bill Mark)

20 passes on Radeon 8500

3 passes on Radeon 9700

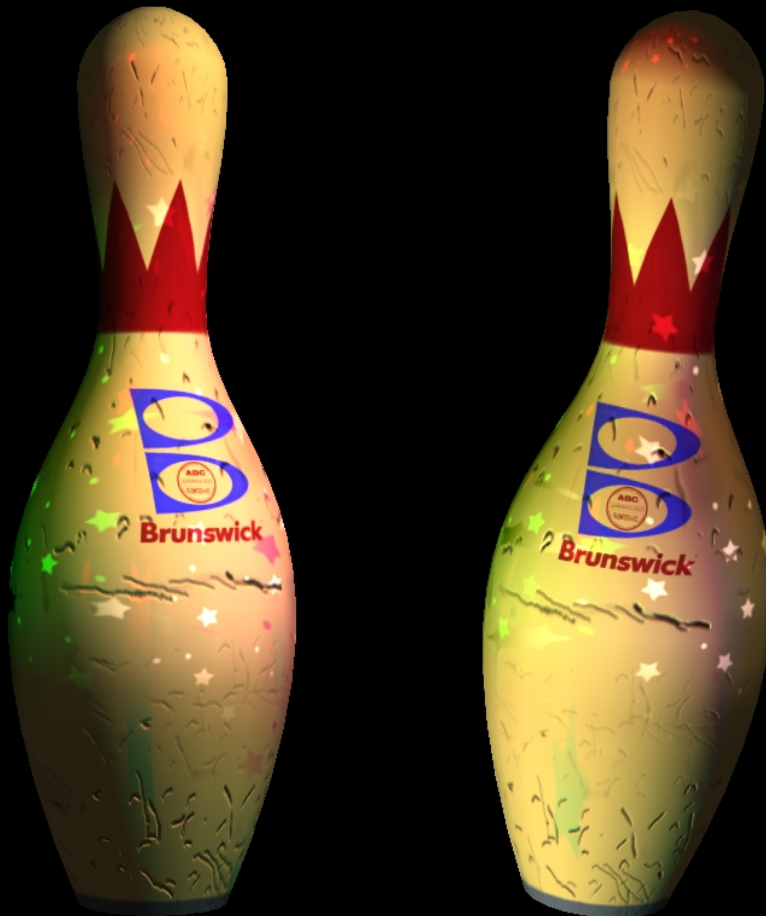
1 pass on NV30

Limited by instructions

Image courtesy of Bill Mark

Images

RenderMan bowling pin + projected textured lights



7 passes on Radeon 8500

5 passes on Radeon 9700

5 passes on NV30

Limited by interpolants!

Images generated using
ATI Radeon 9700

Cost Models

Simple linear model: cost = $c_p\, p + c_t\, t + c_i\, i$

- p = # of passes
- t = # of texture fetches
- i = # of instructions

Different kinds of cost:

- c_p is a *per-pass* cost
- c_t and c_i are *per-fragment* costs

Measured cost model for ATI Radeon 8500:

- cost = 15.7p + 1.3t + i;

Results (Bowling Pin)

Cost model: cost = 15*p* + 5*t* + *i*

	Optimal		RDS	
Architecture	Cost	Passes	Cost	Passes
6 ops	309	11	309	11
4 registers	125	2	131	2
4 textures	171	4	171	4
4 interpolants	269	9	269	9
6_o / 4_r / 4_t / 4_v	310	11	310	11
24_o / 8_r / 8_t / 8_v	179	5	184	5
128_o / 12_r / 16_t / 12_v	145	3	145	3

Results (Wood)

Cost model: cost = 15*p* + 5*t* + *i*

Architecture	Optimal		RDS	
	Cost	Passes	Cost	Passes
6 ops	–	–	2685	91
4 registers	–	–	1462	32
4 textures	378	2	378	2
4 interpolants	374	2	374	2
$6_o / 4_r / 4_t / 4_v$	–	–	2681	92
$24_o / 8_r / 8_t / 8_v$	–	–	937	17
$128_o / 12_r / 16_t / 12_v$	396	3	396	3

Future Work

- Support for branching and loops
- Support for multiple outputs
- Support for vertex shaders
- Faster algorithms (use incremental techniques)

Summary

Contribution:

- RDS virtualizes GPU resources by splitting arbitrarily large fragment shaders into multiple passes

Virtualization today:

- Multipass rendering + pass-splitting algorithm

Virtualization tomorrow:

- Combination of software + hardware techniques
- GPU-assisted mechanisms (like CPU register spilling)

Virtualization Is Key

Why virtualization is critical:

- All GPUs have resource limits
- Program size isn't always the limiting factor!
- Forward compatibility is useful

Future of programmable GPUs:

- Not just procedural shading
- Large scientific computations, physical simulations
- Programs shouldn't be held back by resource limits!

Acknowledgments

Stanford graphics architecture group

Bill Mark (NVIDIA)

Monica Lam (Stanford compilers/OS group)

Sponsors

- DARPA, ATI, NVIDIA, Sony, Sun

Hardware, drivers, and bug fixes

- Matt Papakipos, Mark Kilgard, Nick Triantos, Pat Brown
- James Percy, Bob Drebin, Evan Hart, Jeff Royle