Shading Languages for Graphics Hardware

Bill Mark and Kekoa Proudfoot

Stanford University

Collaborators:
Pat Hanrahan, Svetoslav Tzvetkov, Pradeep Sen, Ren Ng

Sponsors:
ATI, NVIDIA, SGI, SONY, Sun, 3dfx, DARPA

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

Motivation

Current generation of hardware is very capable

- Vector-based vertex processing
- Multiple textures per pass
- Advanced texture combining operations
- Multiple passes for complex effects

Downside: Hardware is difficult to program

- Programming can be like writing microcode
- Splitting computations into multiple passes is time-consuming
- Functionality varies between chipsets

Real-time shading languages

Implement a shading language with two objectives:

- Higher-level programming interface
- Portability across platforms and chipsets

Borrow ideas from off-line rendering systems

- e.g. PIXAR's PhotoRealistic RenderMan

Tailor language to programmable graphics HW

- Language must obey limitations of hardware

Topics

Survey of real-time shading systems

Overview of Stanford system

Demo

Compiler technology in Stanford system

- Vertex programs
- Register combiners / Pixel shaders

Concluding comments

Questions

Related systems

Quake 3 Arena (id Software)

- ftp://ftp.idsoftware.com/idstuff/quake3/tools/Q3Ashader_manual.doc

Interactive Multi-Pass Programmable Shading

- Peercy, et al., SIGGRAPH 00
- http://reality.sgi.com/olano/papers/

McCool's SMASH API

- http://www.cgl.uwaterloo.ca/Projects/rendering/Papers/smash.pdf

Quake 3 Arena: shader scripts

Linear chain of image compositing stages

```
textures/testsurface
{
    {
        map textures/basecolor.tga
        rgbgen identity
    }
    {
        map $lightmap
        rgbgen identity
        blendfunc filter
    }
}
```

Stage 1: base color

Stage 2: lightmap

Stages are mapped to rendering passes

Vertex colors, positions, and texcoords can be generated/modified using builtin functions

Peercy *et al*.: overview

Two languages implemented:

- Simple Shading Language for standard OpenGL
- RenderMan for extended OpenGL

System based on a SIMD processor abstraction:

- Graphics hardware = SIMD processor
- One rendering pass = SIMD instruction

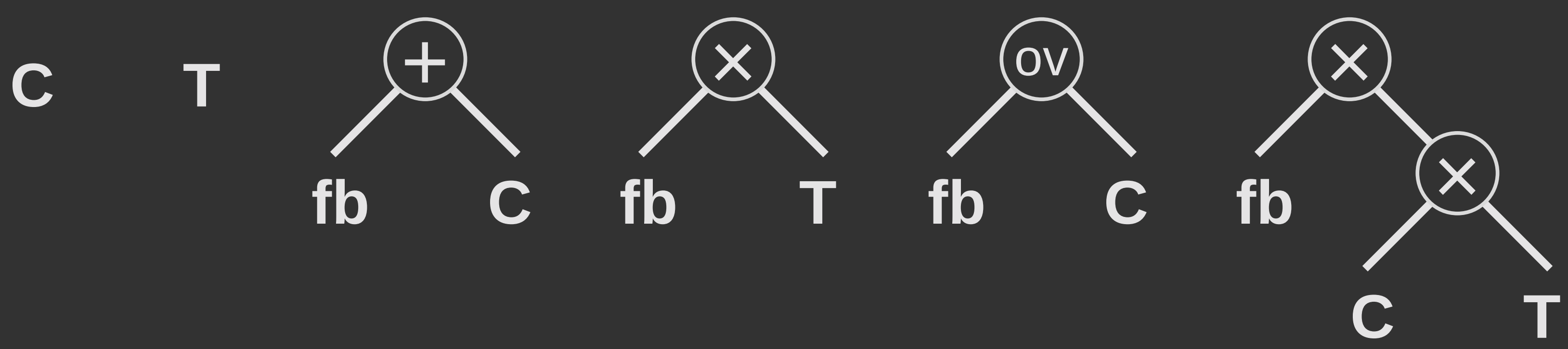
Computations are compiled to many simple passes

Peercy *et al.*: pass generation

A tool called *iburg* maps computations to passes

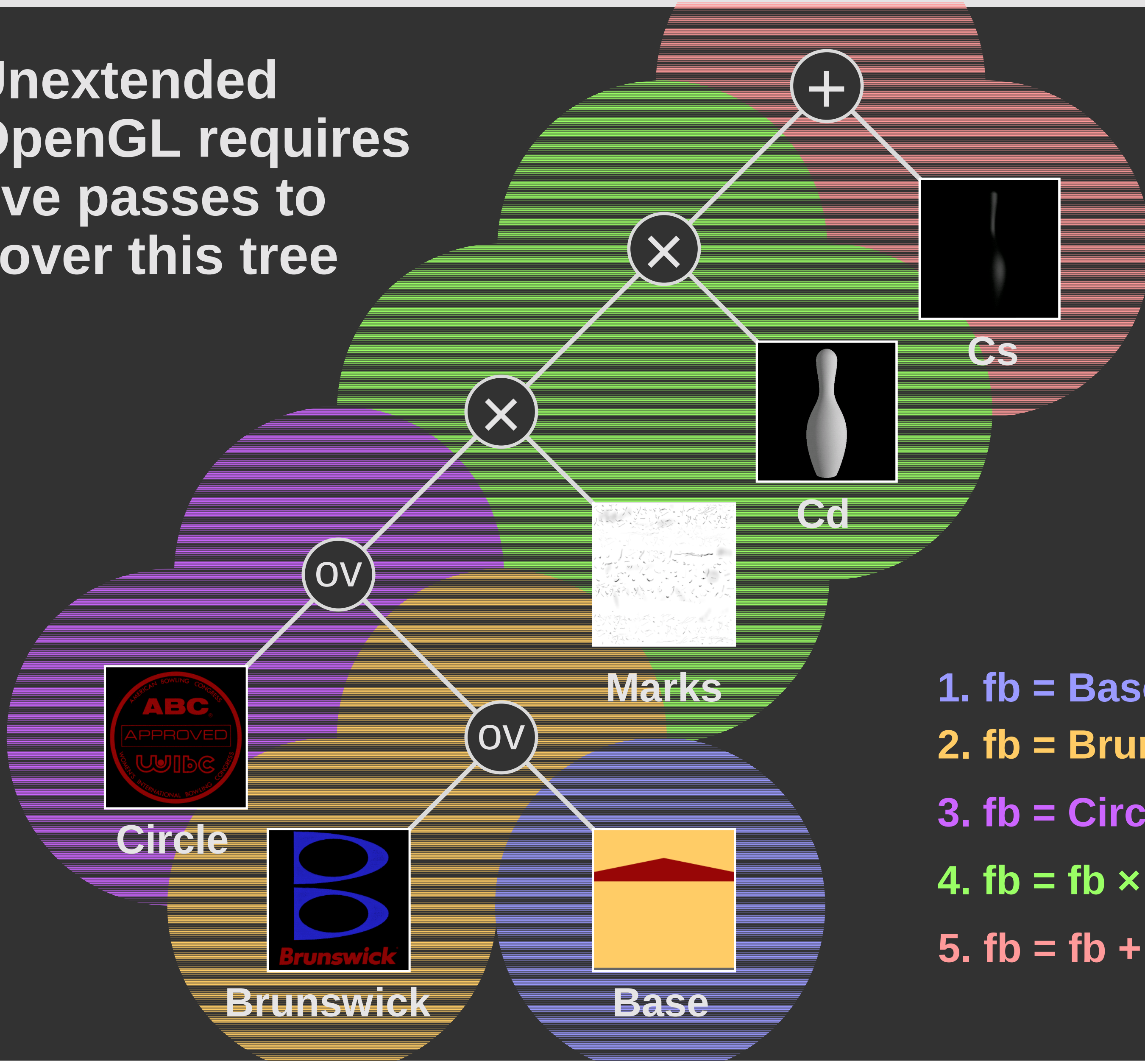
- Express computation as one or more trees
- Express possible passes as rules
- Dynamic programming optimally covers trees given rules

Some example OpenGL 1.1 rules:



Peercy *et al.*: *iburg* example

Unextended OpenGL requires five passes to cover this tree



1. fb = Base
2. fb = Bruns over fb
3. fb = Circle over fb
4. fb = fb × Marks × Cd
5. fb = fb + Cs

Project goals

1. Provide a shading language as an abstraction layer between programmer and graphics hardware
2. Explore how current hardware may be used to implement shading language abstractions
3. Investigate new hardware architectures optimized for programmable shading
4. Create new interactive applications based on shading languages

Design philosophy

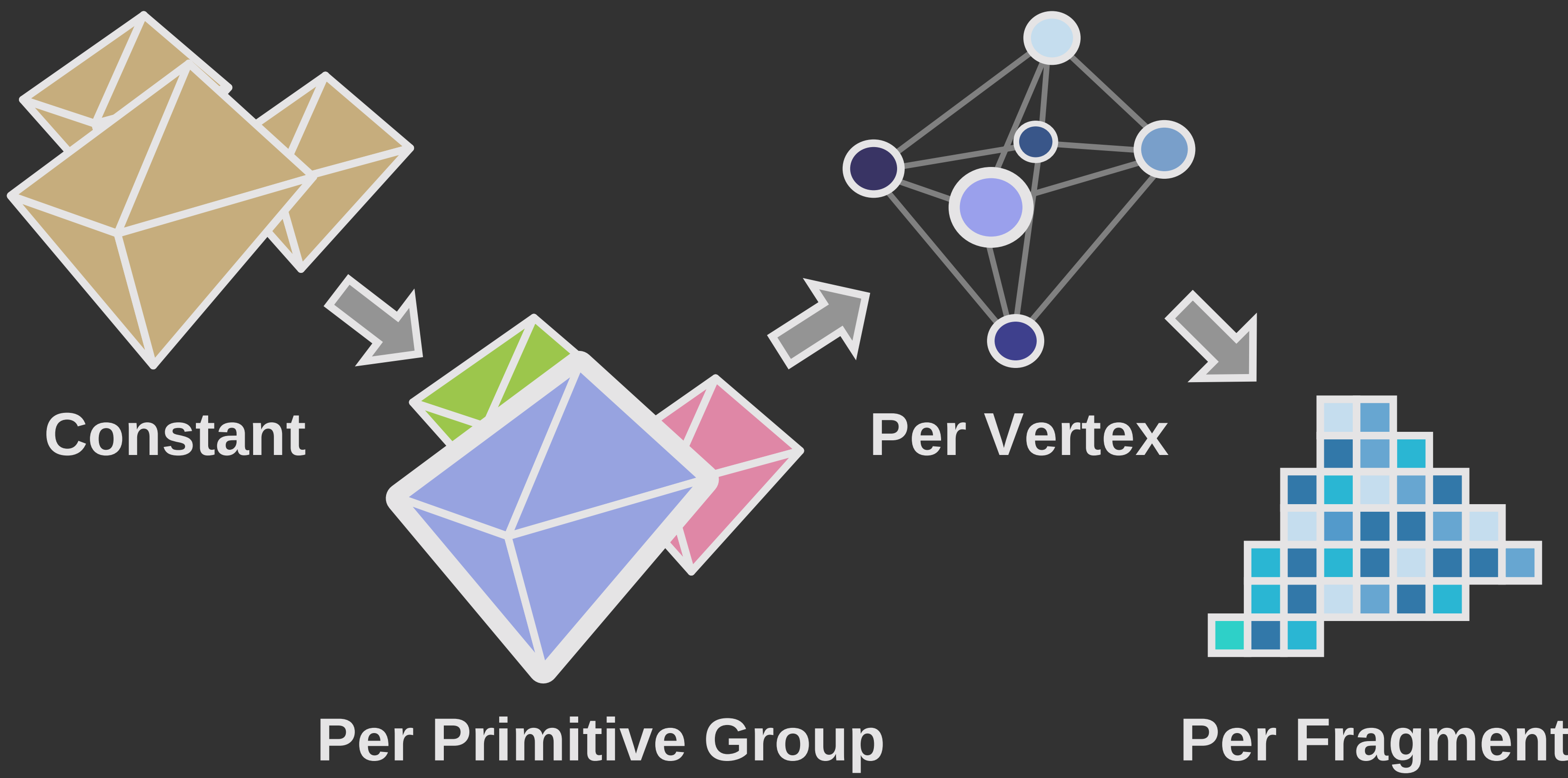
Primary goals of current shading system

- Make hardware easy-to-use
- Make shaders portable across platforms

System only abstracts what hardware can do:

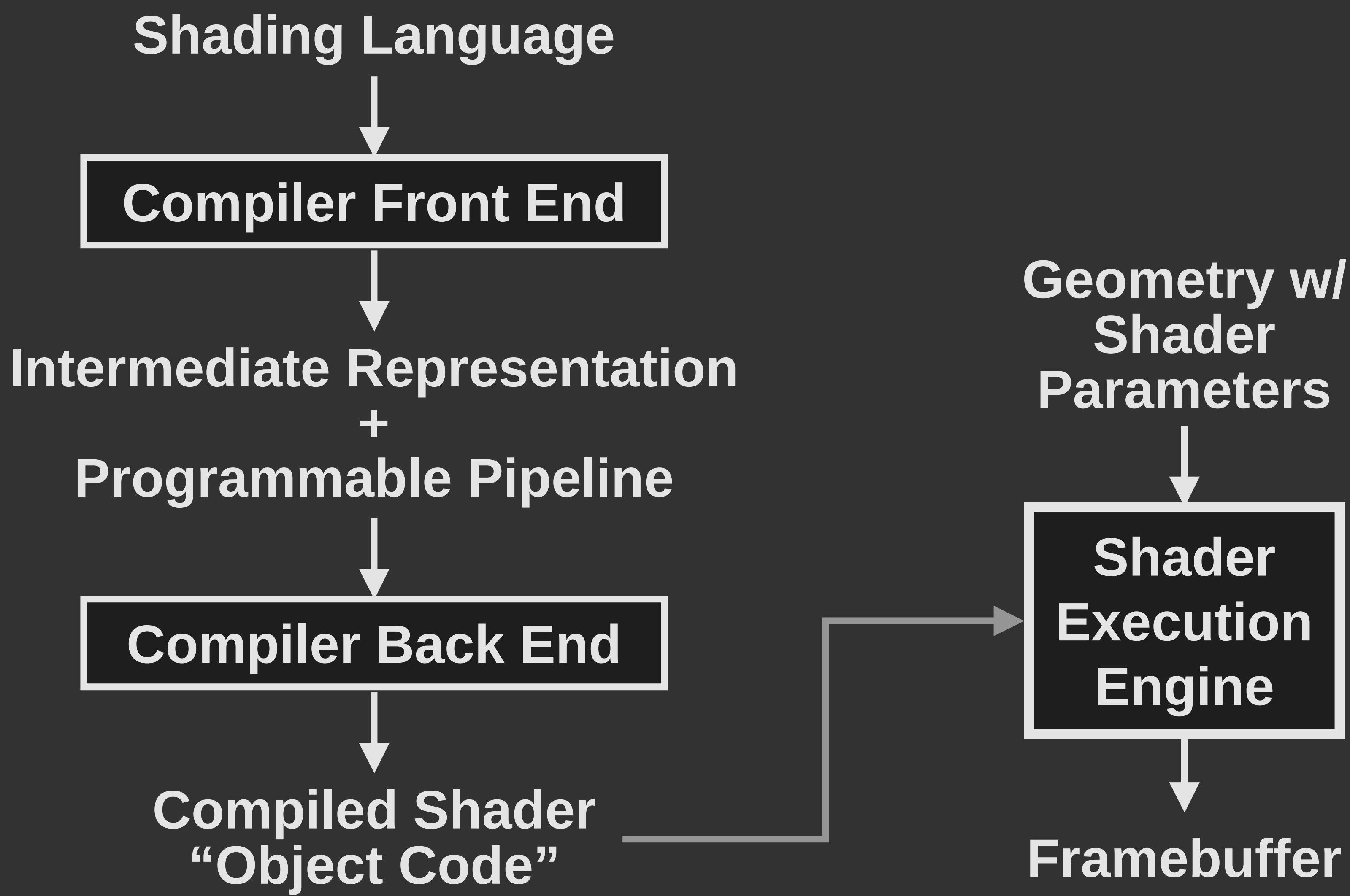
- The system does not perform magic!!!

Multiple computation frequencies

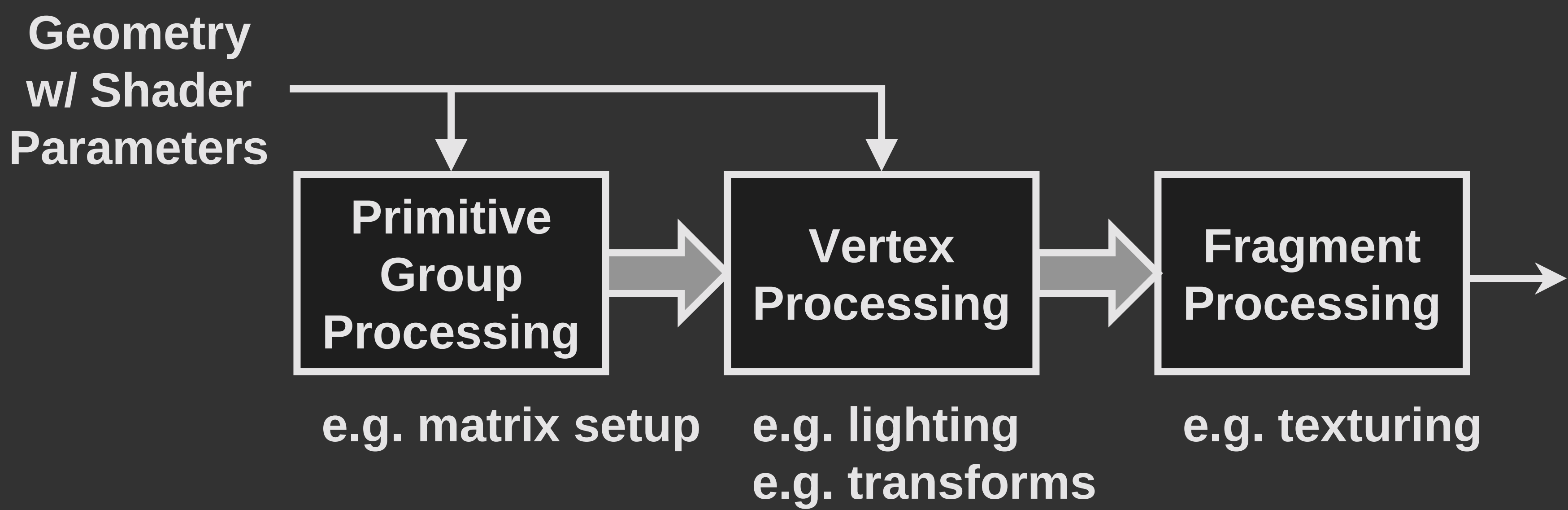


Evaluated less often | Evaluated more often
More complex operations | Simpler operations
Floating point | Fixed point

System overview



Programmable pipeline abstraction



Intermediate abstraction between language and HW

Virtualization of hardware:

- No operation count limits
- No temporary storage limits
- Note: DX8/OpenGL do not have these properties

Restrictions

Hardware limitations

- No data-dependent loops or conditionals
- No random read/write memory accesses
- No fully-general dependent texturing

Optional operators

- Not all hardware supports every operation

No fully-orthogonal “float” type

- Fragment values are currently fixed point
- Fragment “float” is largest range possible on hardware
- Currently either [0,1] or [-1,1]

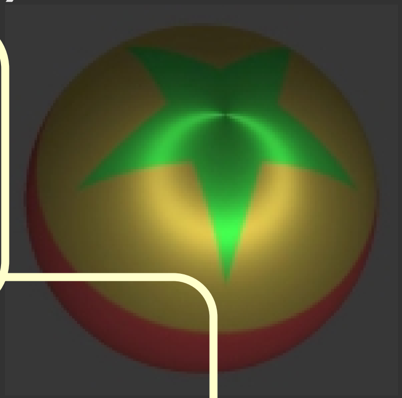
Anisotropic ball example

```
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);
}
```



Shading language

Features

- Simple C-like language
- Scalar, vector, and matrix operations
- Separate surface and light shaders
- Easy specification of computation frequencies
- Designed to be easily analyzed and optimized
- Deformation shaders are coming soon

See also:

- Conference Proceedings
- Project Web Site (URL at end of talk)

API

Two APIs implemented

- Vertex array interface
- Immediate-mode interface

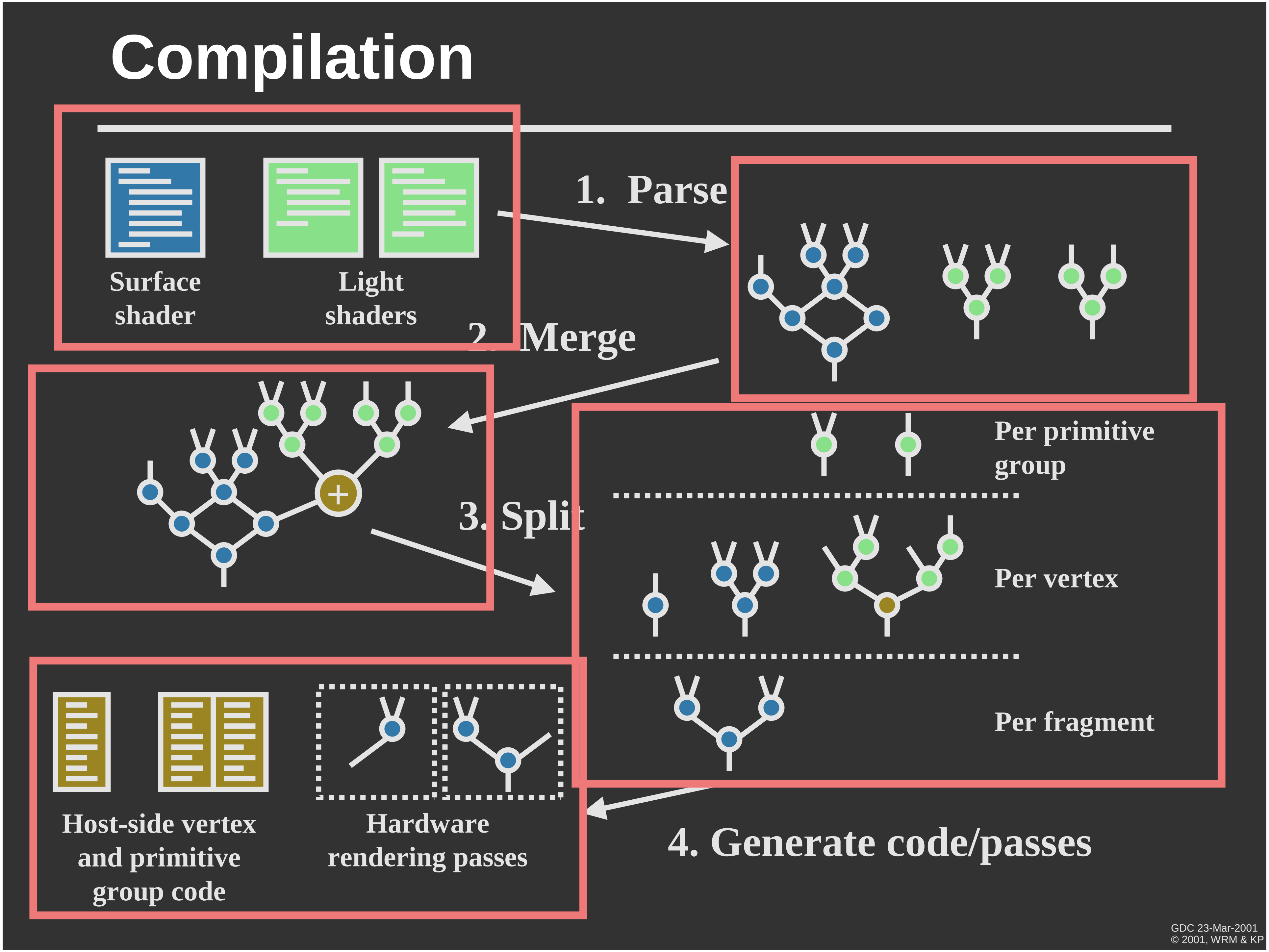
Both extend standard interfaces with:

- Shader specification routines
- Routines for specifying arbitrary params

APIs hide multiple rendering passes

- Must buffer data if multiple passes required

Demo



Compilation
Surface shader
Light shaders
1. Parse
2. Merge
3. Split
Per primitive group
Per vertex
Per fragment
Host-side vertex and primitive group code
Hardware rendering passes
4. Generate code/passes
GDC 23-Mar-2001
© 2001, WRM & KP

System generates many types of code

Host Processor

- C code or X86 code
- Use as fallback when there's no vertex HW

Vertex Programs

Multi-pass OpenGL 1.2

- Uses Multi-texture
- Very Portable
- Virtualizes HW

Register Combiners

Vertex-program architecture

- Access to just one vertex at a time
- Instructions operate on four 32-bit floats (like SSE2)
- Instruction set:
  - Mostly RISC-like
  - Some special instructions for graphics – e.g. LIT
  - No branches
- Can negate and/or swizzle any instruction operand
- Two types of registers – read/write, and read-only

Step 1: Instruction selection

Choose instructions from templates

C = cross(A, B) → MUL C, A.zxyw, B.yzxw
MAD C, A.yzxw, B.zxyw, -C

Optimizations (pre- and post- template)

- MUL, ADD → MAD
- Match patterns for LIT, DST
- Combine scalar ops into vector instructions
- etc.

Step 2: Allocate read/write registers

Adapt standard algorithms to this architecture

- Determine set of live values at each instruction
- Construct interference graph
  - Scalars treated same as vectors for interference
- Use greedy graph-coloring algorithm
  - Allocate “hardest” variables first
  - Put up to four unrelated scalars in a register
  - Careful about outputs from DST, LOG
- More details in our SIGGRAPH paper

Step 3: Allocate “constant” registers

1. Assign “primitive group” variables to constant registers
2. Assign true constants to constant registers
   A. Rank constants by number of unique values e.g. {1,2,3,4} ranks higher than {1,1,1,2}
   B. Assign highest-ranked constants first
   C. Try to reuse components from previously-allocated values

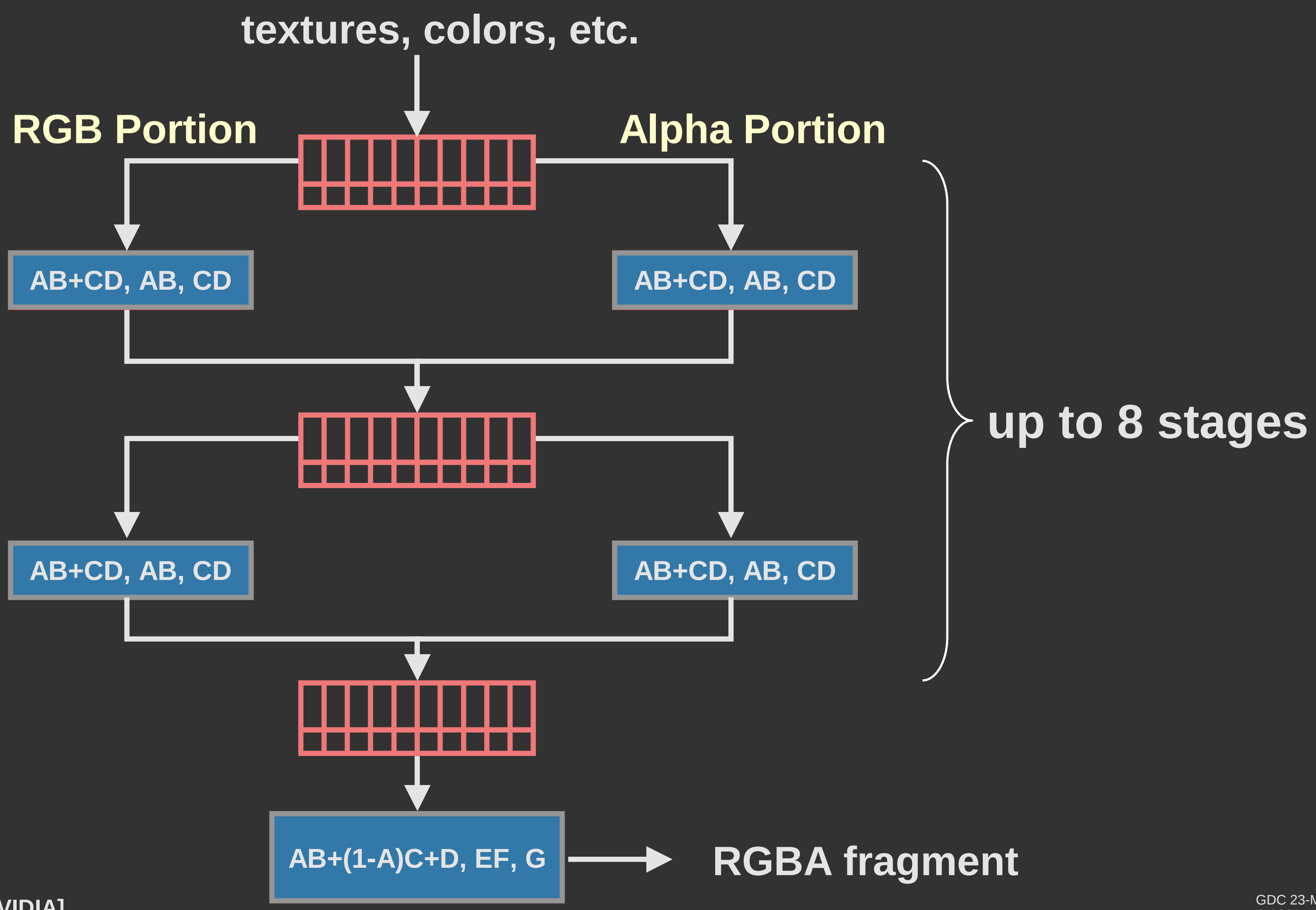
Generated code is efficient

Example: surface with local, specular light

- Compiler-generated code
  - 44 instructions
- Hand-written code (from NVIDIA template)
  - 38 instructions

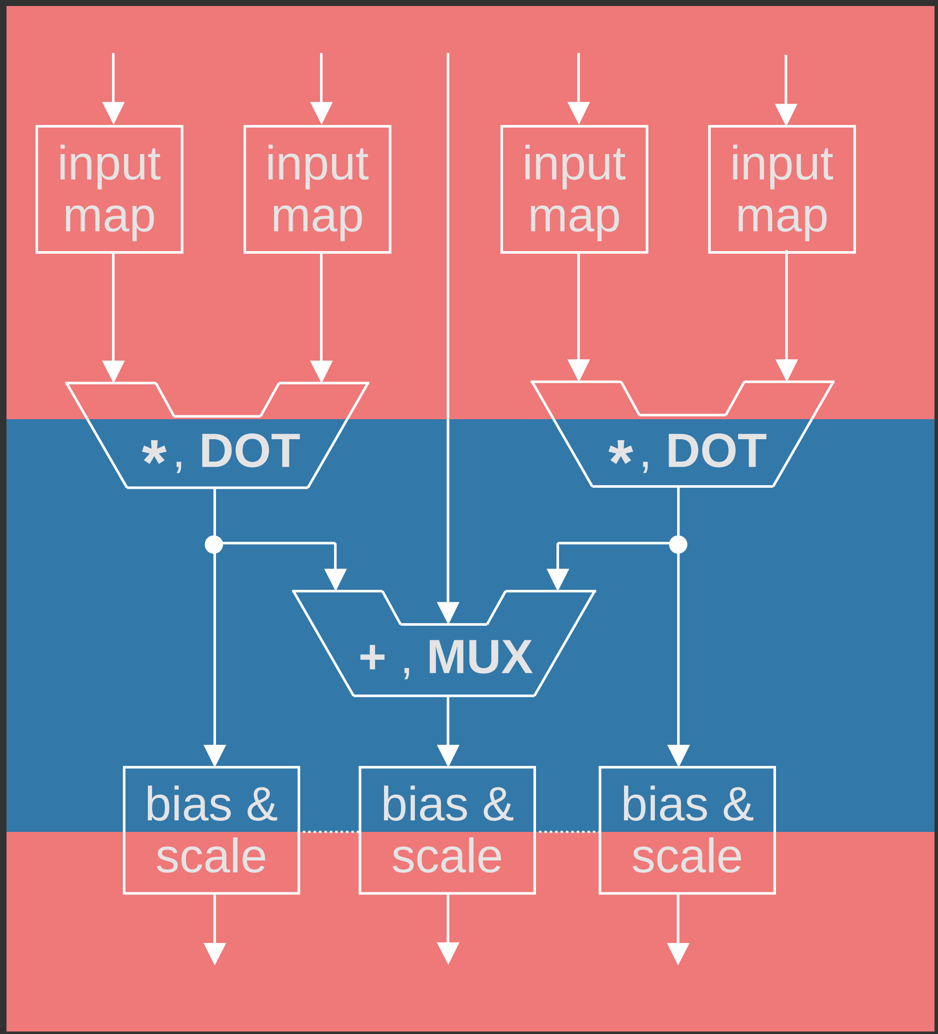
Register combiner pipeline

Similar to a VLIW architecture



[slide modified from NVIDIA]

RGB register combiner



[-1,1] range

[-2,2] range

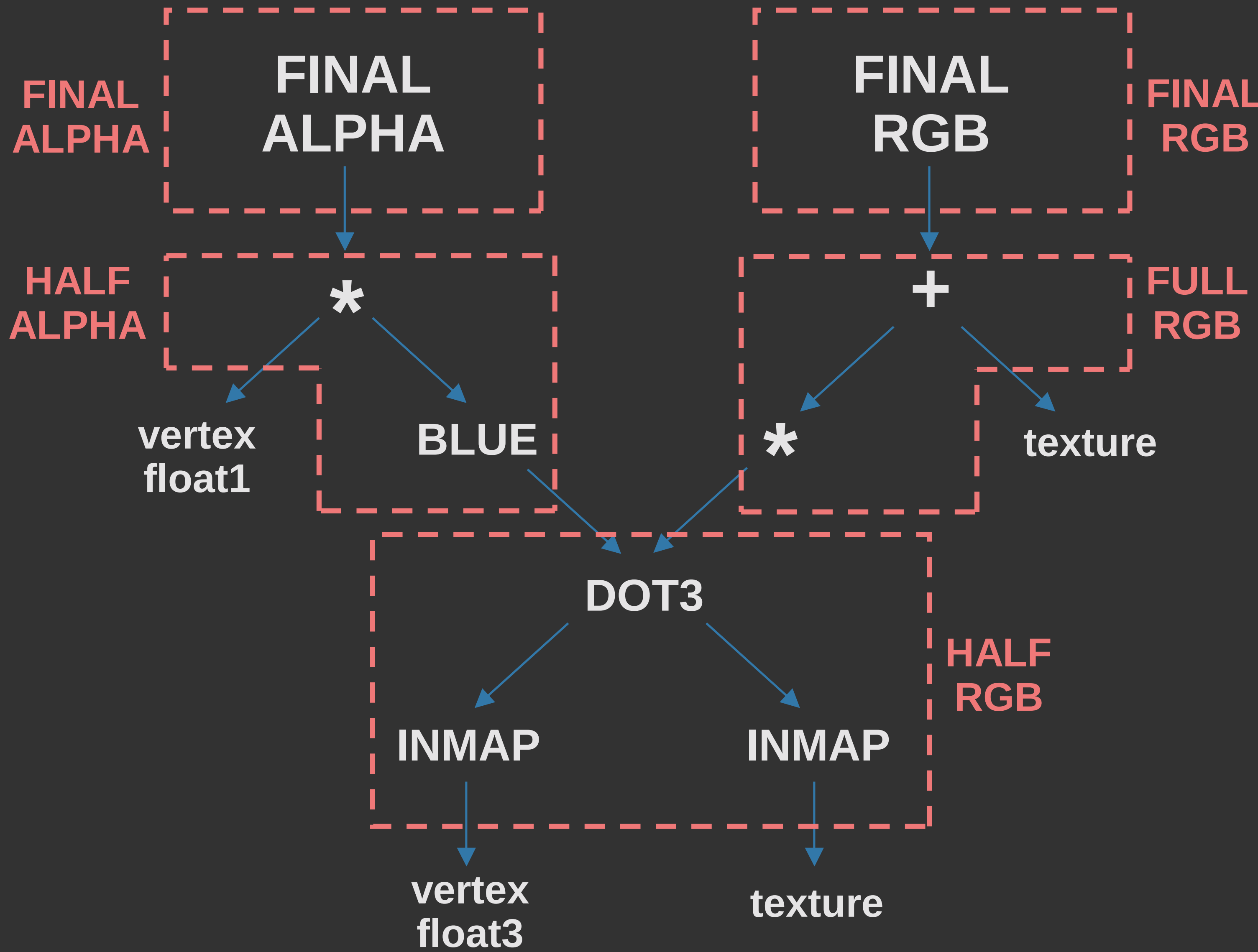
[-1,1] range

Note: DX8 pixel shader instructions are similar, but slightly less powerful

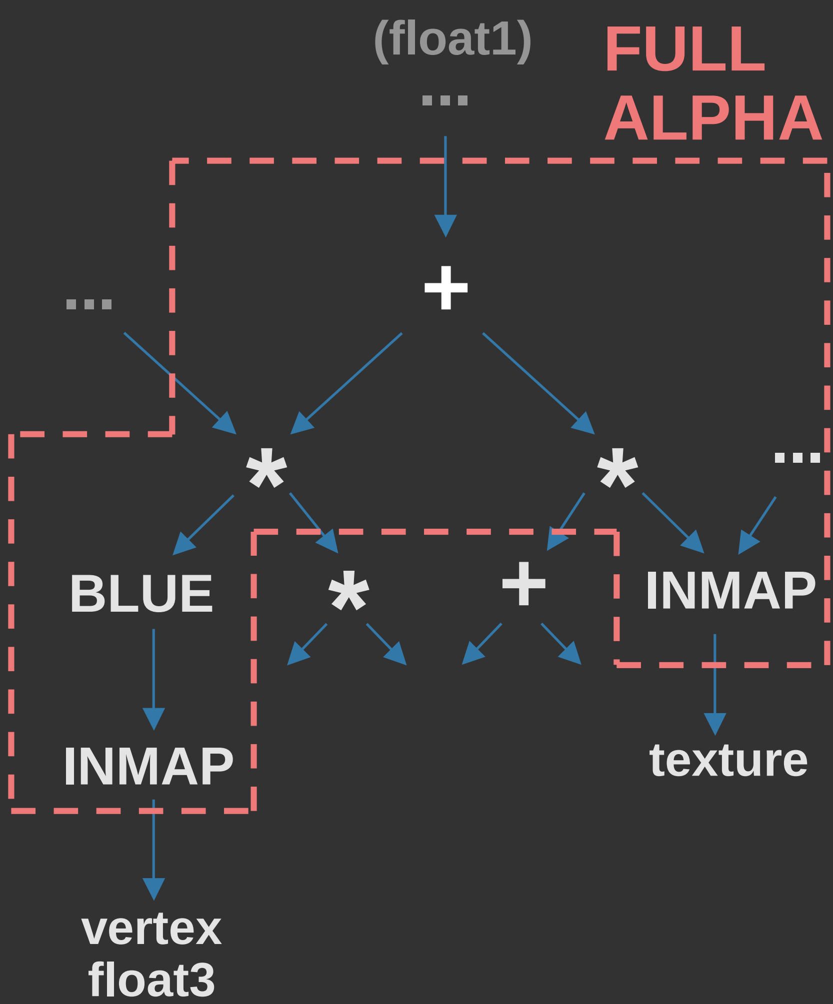
Rewrite DAG to use basic HW ops

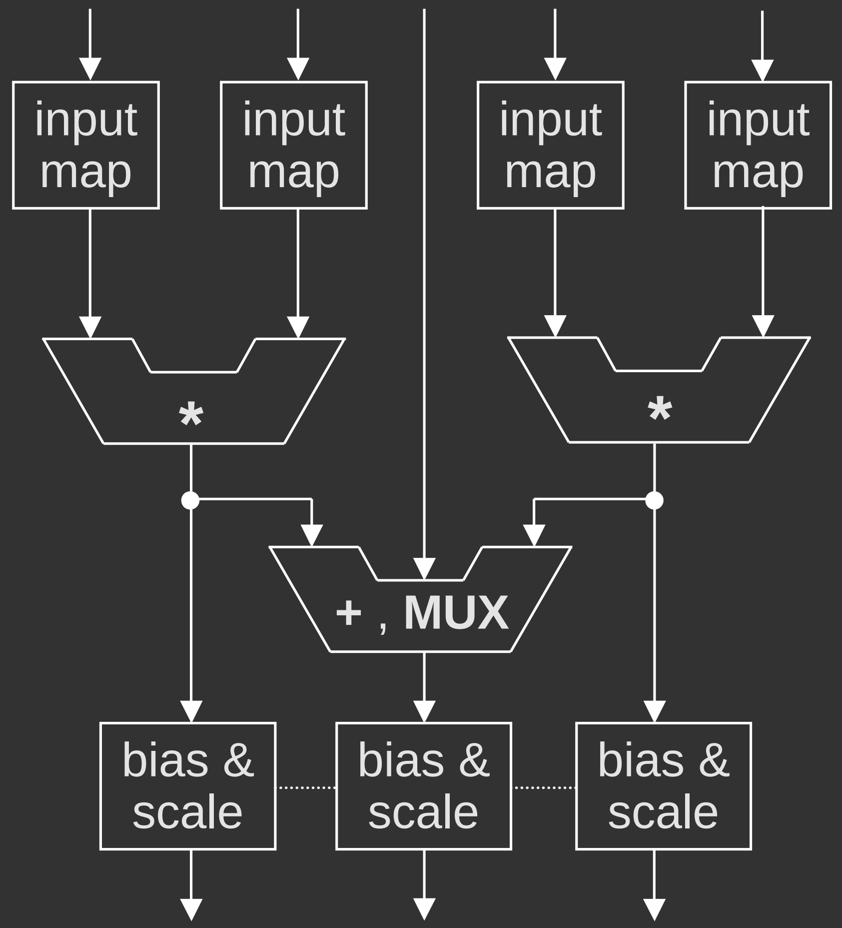
From...		To...
float4 * float4	→	{ float3 * float3 float1 * float1
X **DOT** Y	→	**BLUE (** X **DOT3** Y **)**
2 * (X - 0.5)	→	**INMAP(X, expand_normal)**

Map ops to partial combiners using top-down algorithm



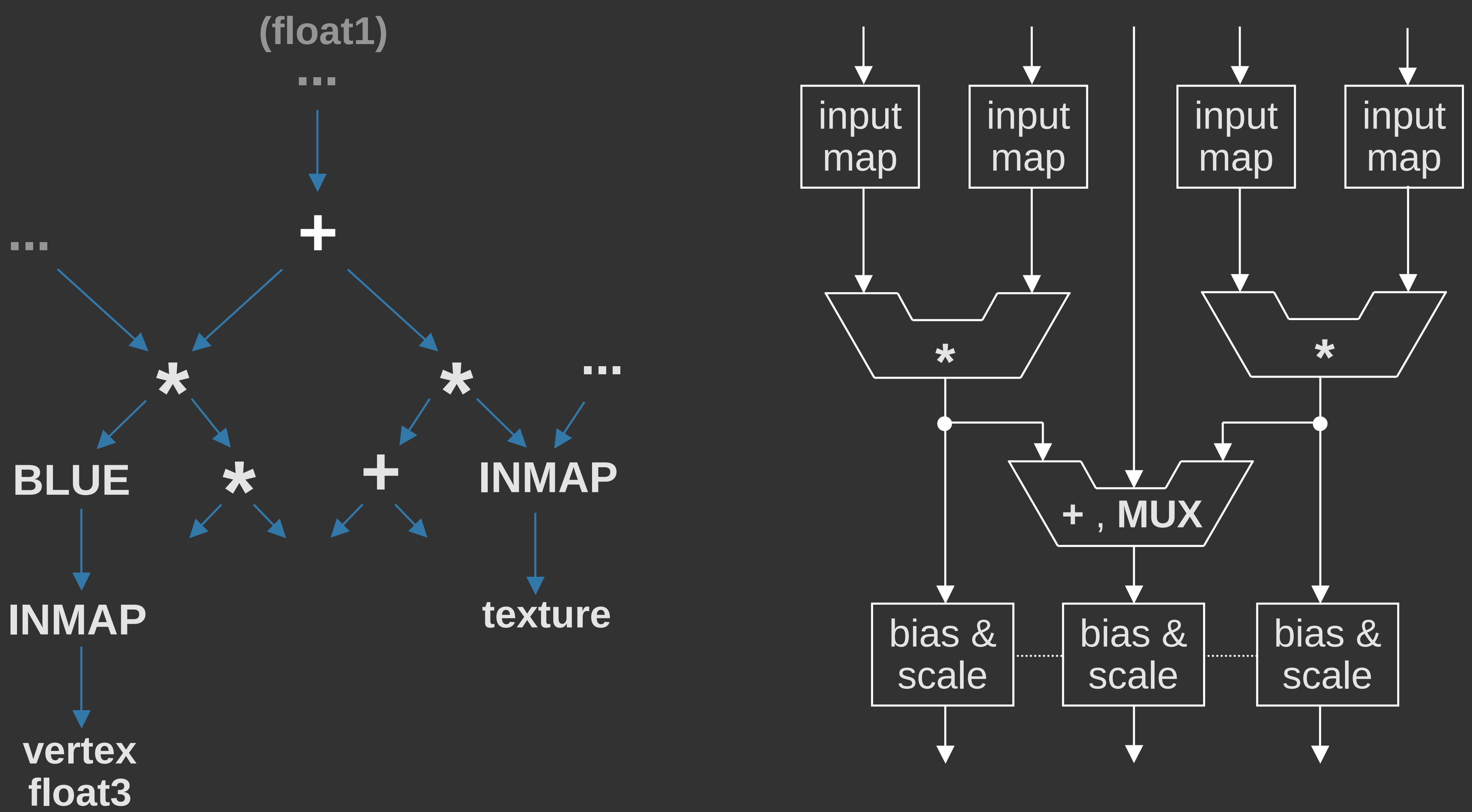
Mapping ops to a partial combiner





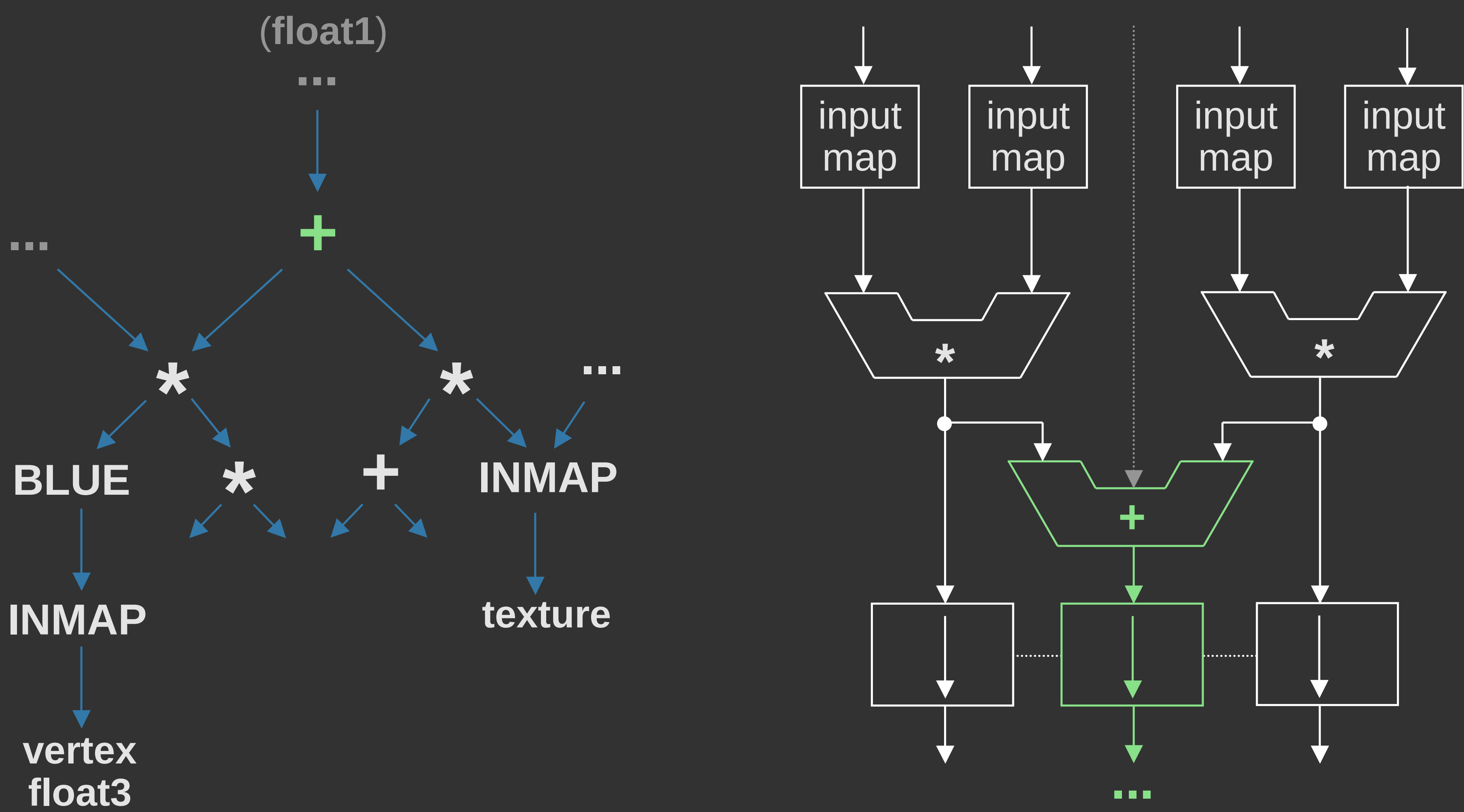
ALPHA COMBINER

Mapping ops to a partial combiner

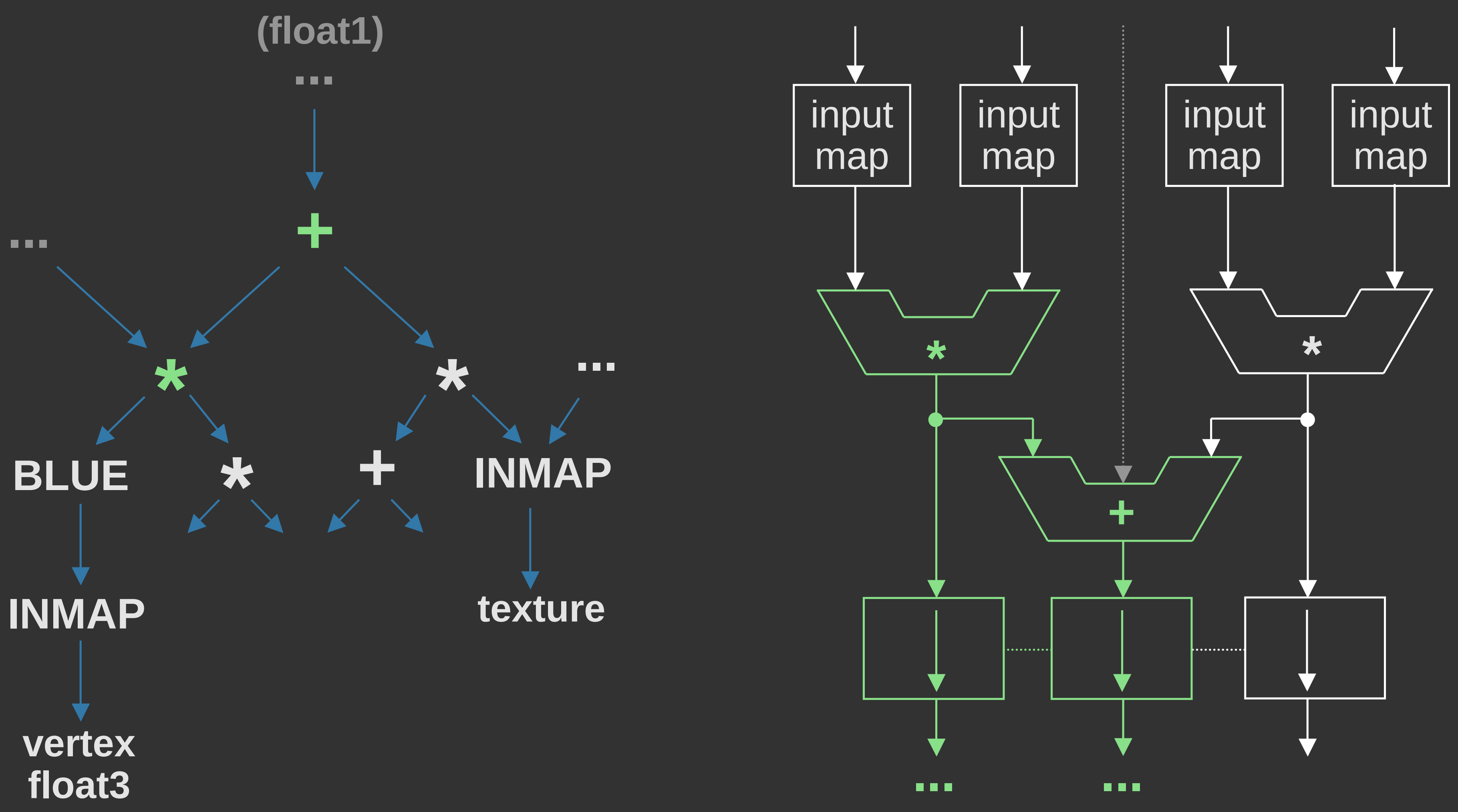


ALPHA COMBINER

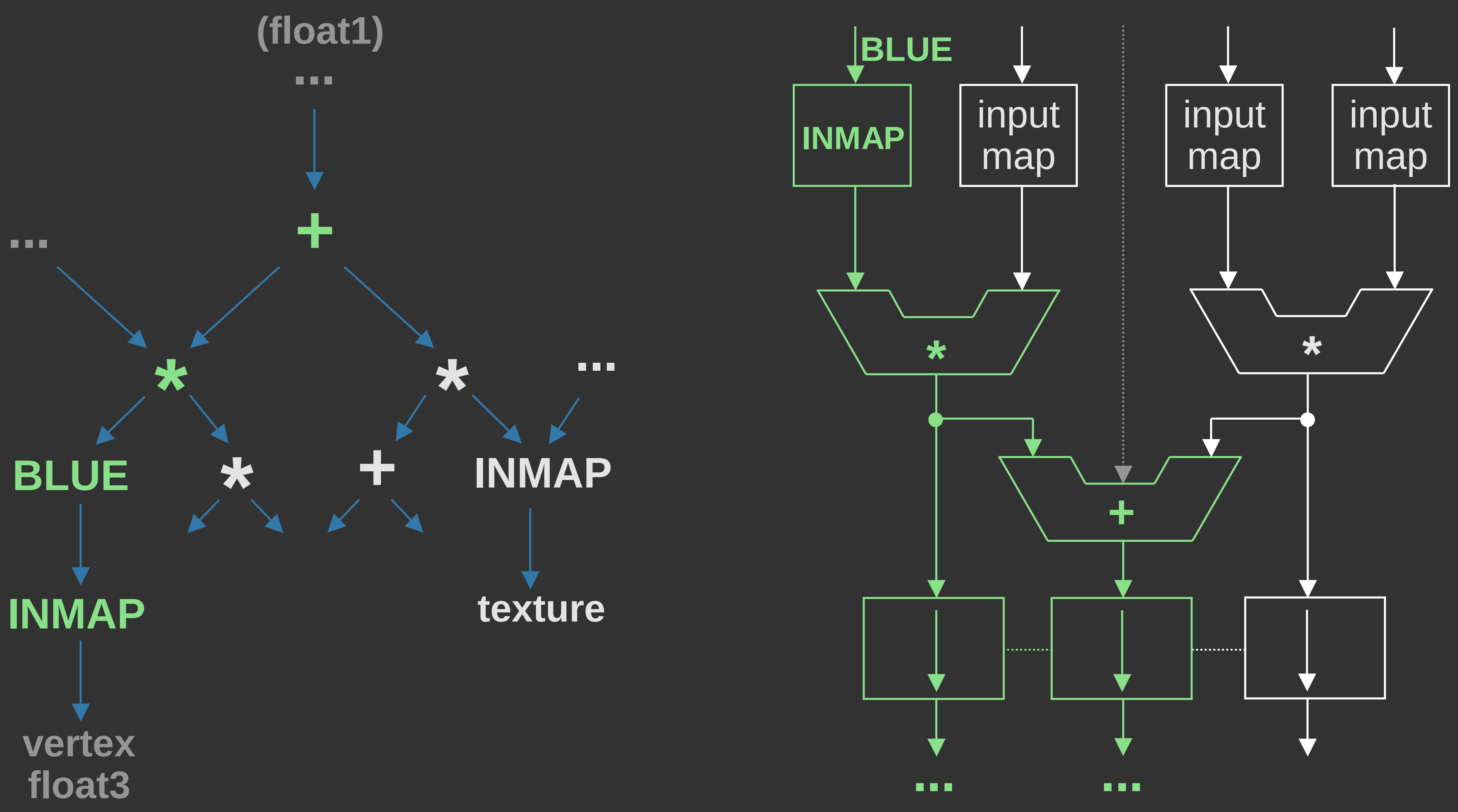
Mapping ops to a partial combiner



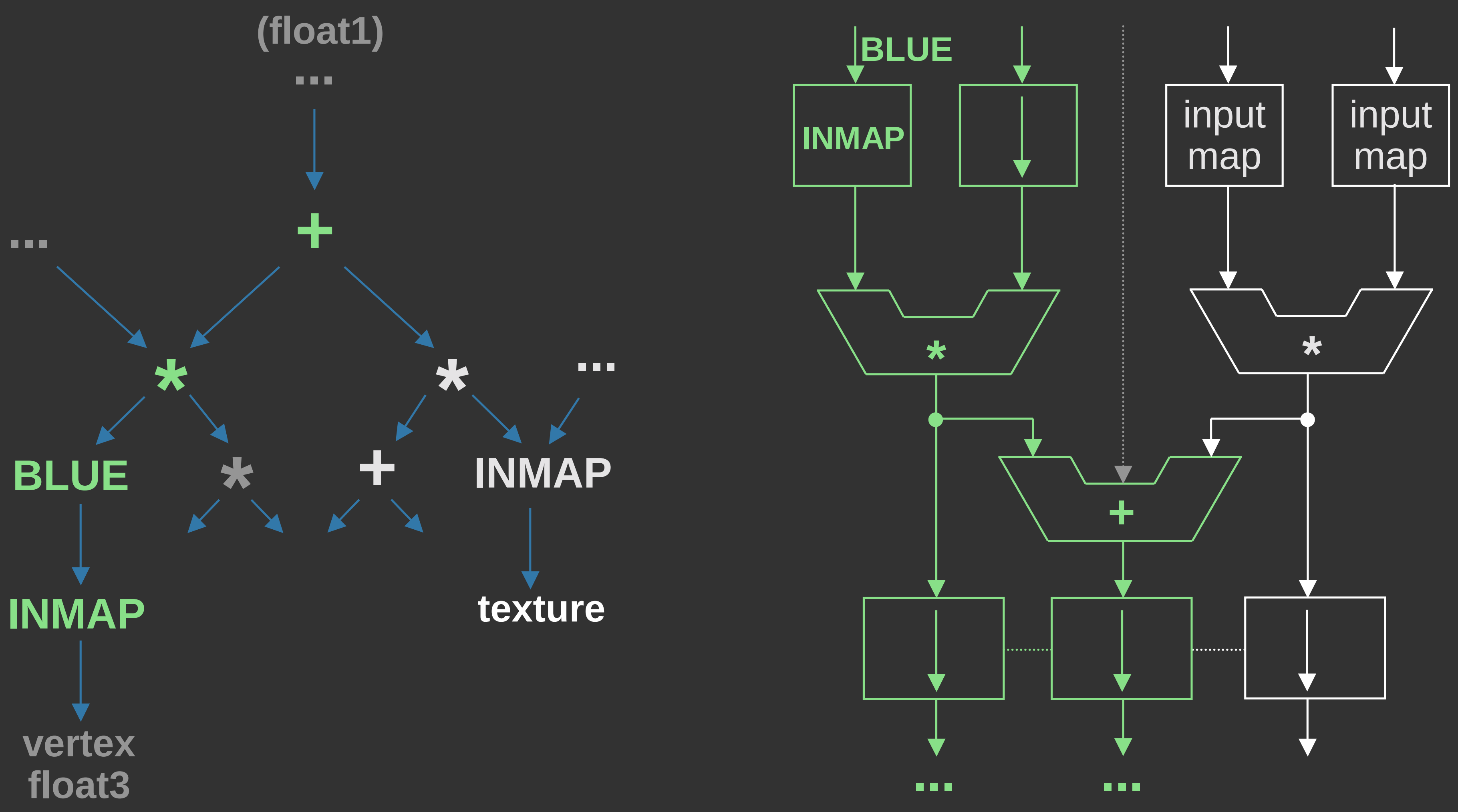
Mapping ops to a partial combiner



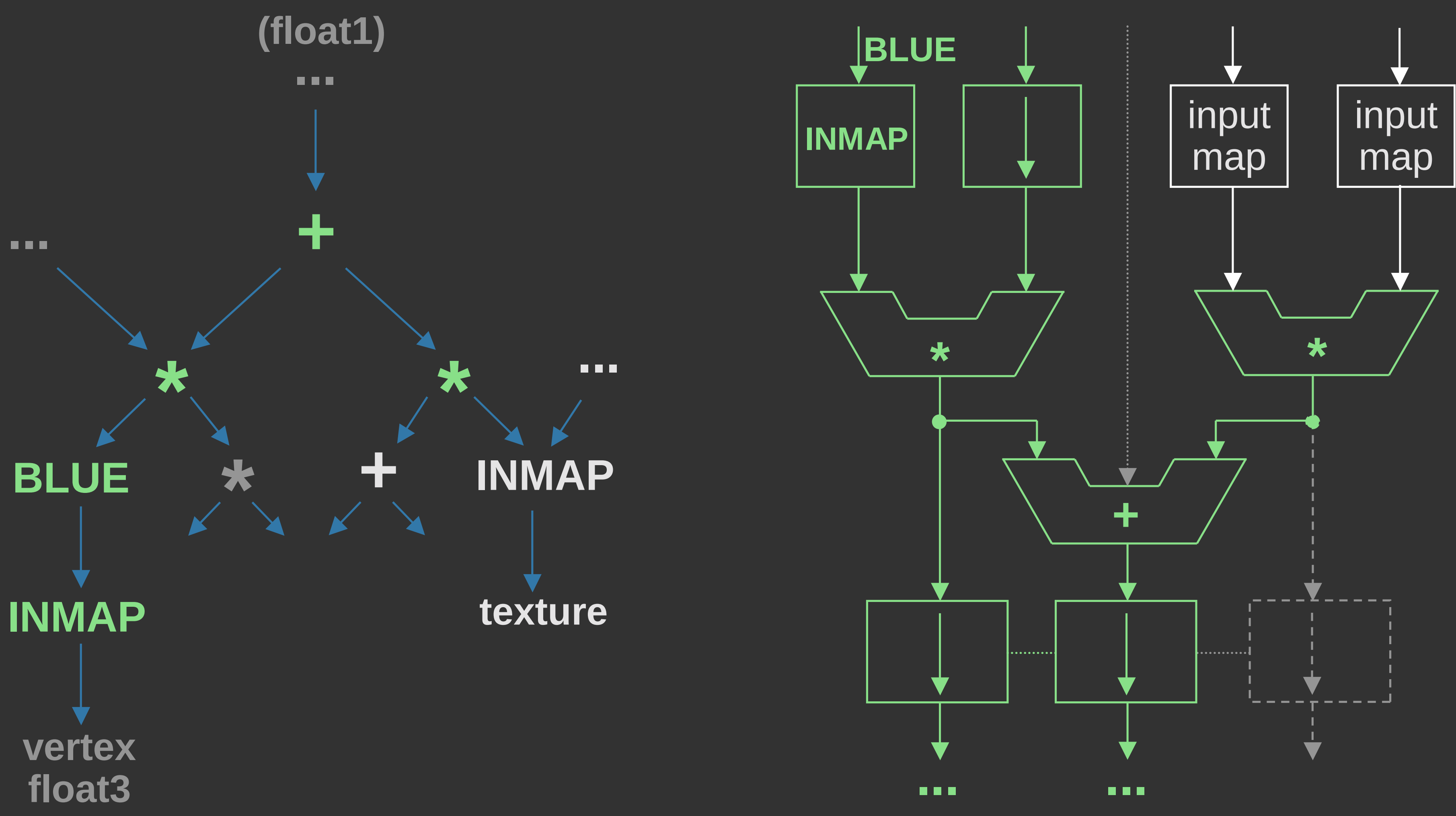
Mapping ops to a partial combiner



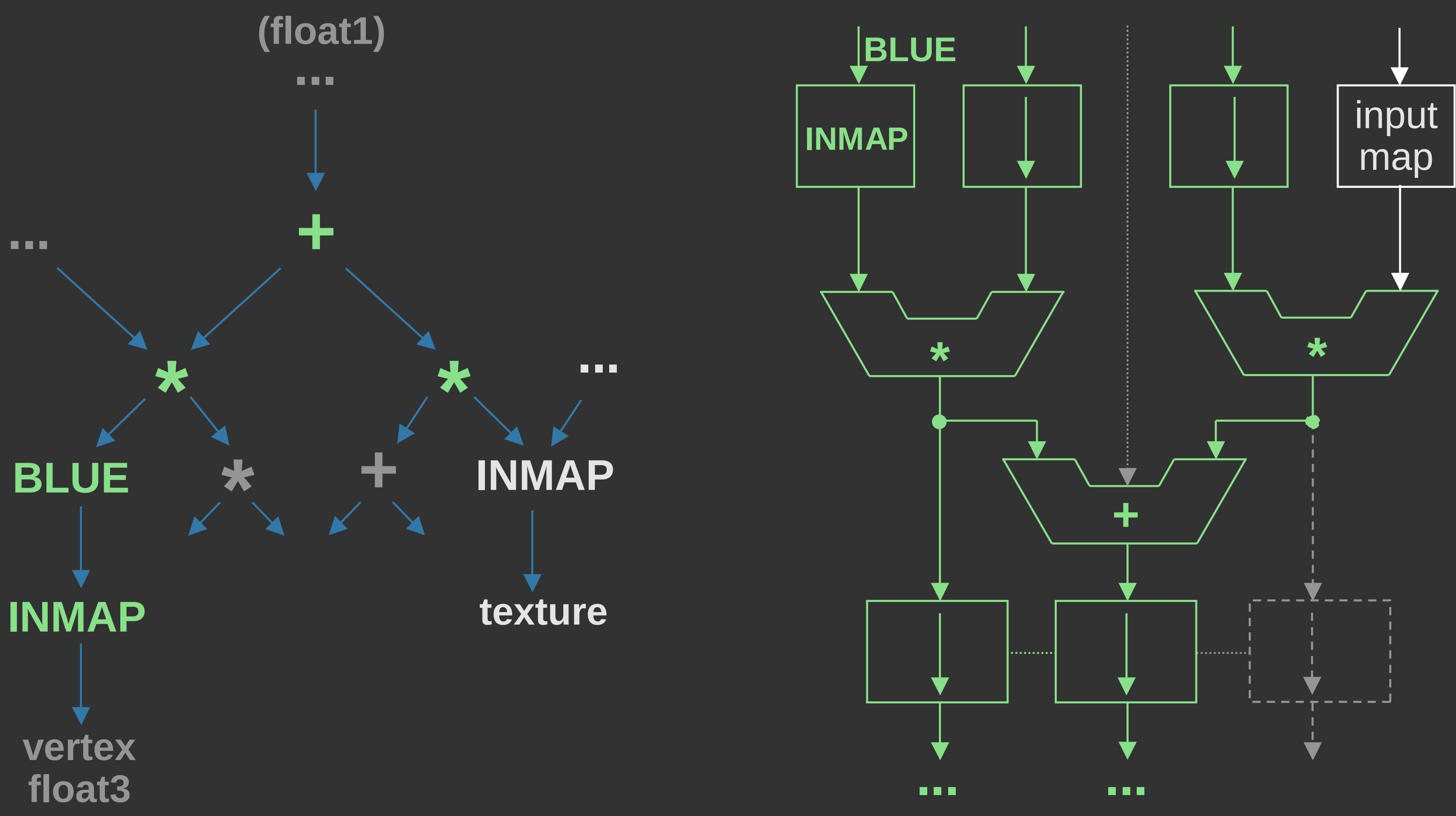
Mapping ops to a partial combiner



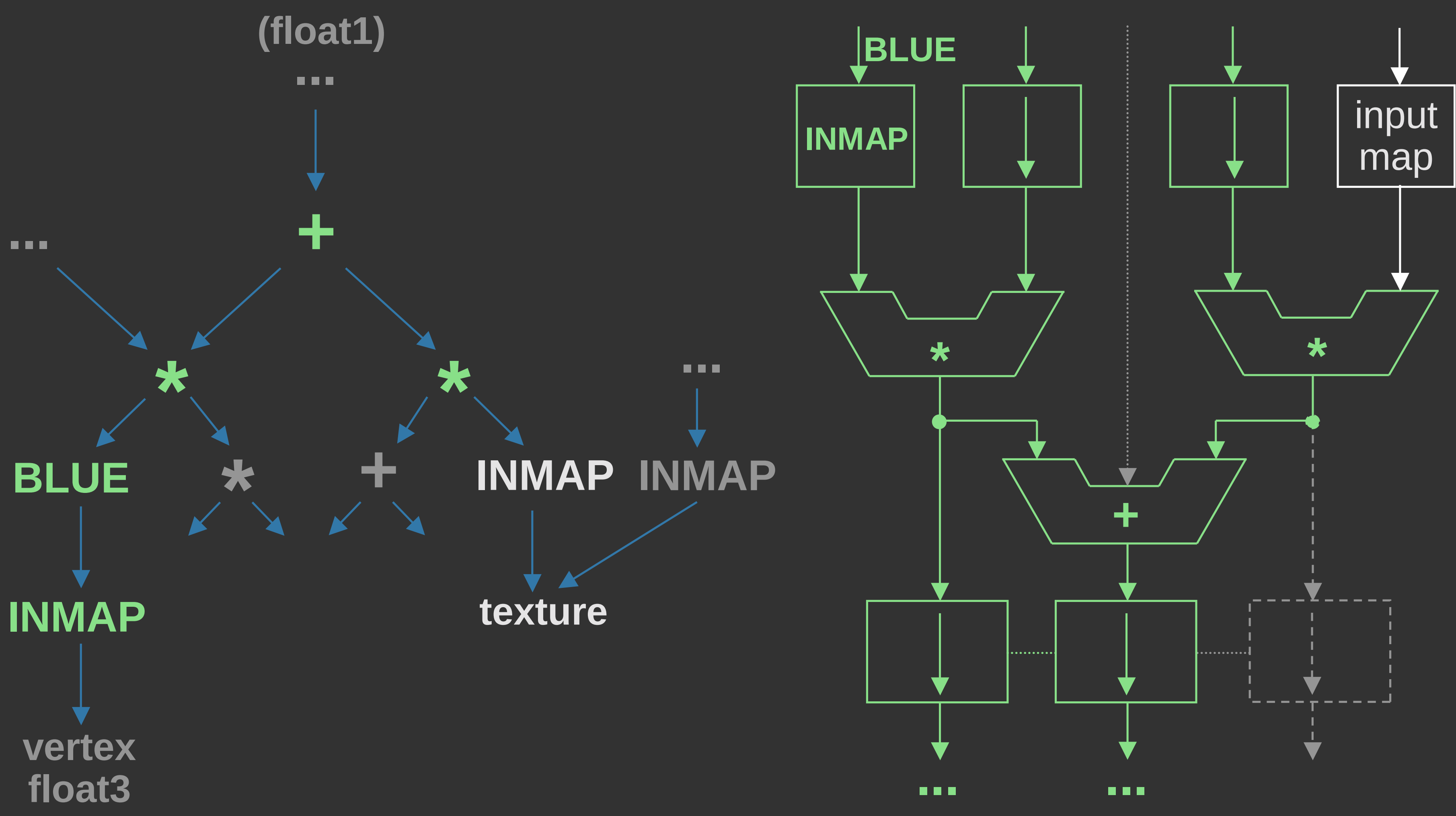
Mapping ops to a partial combiner



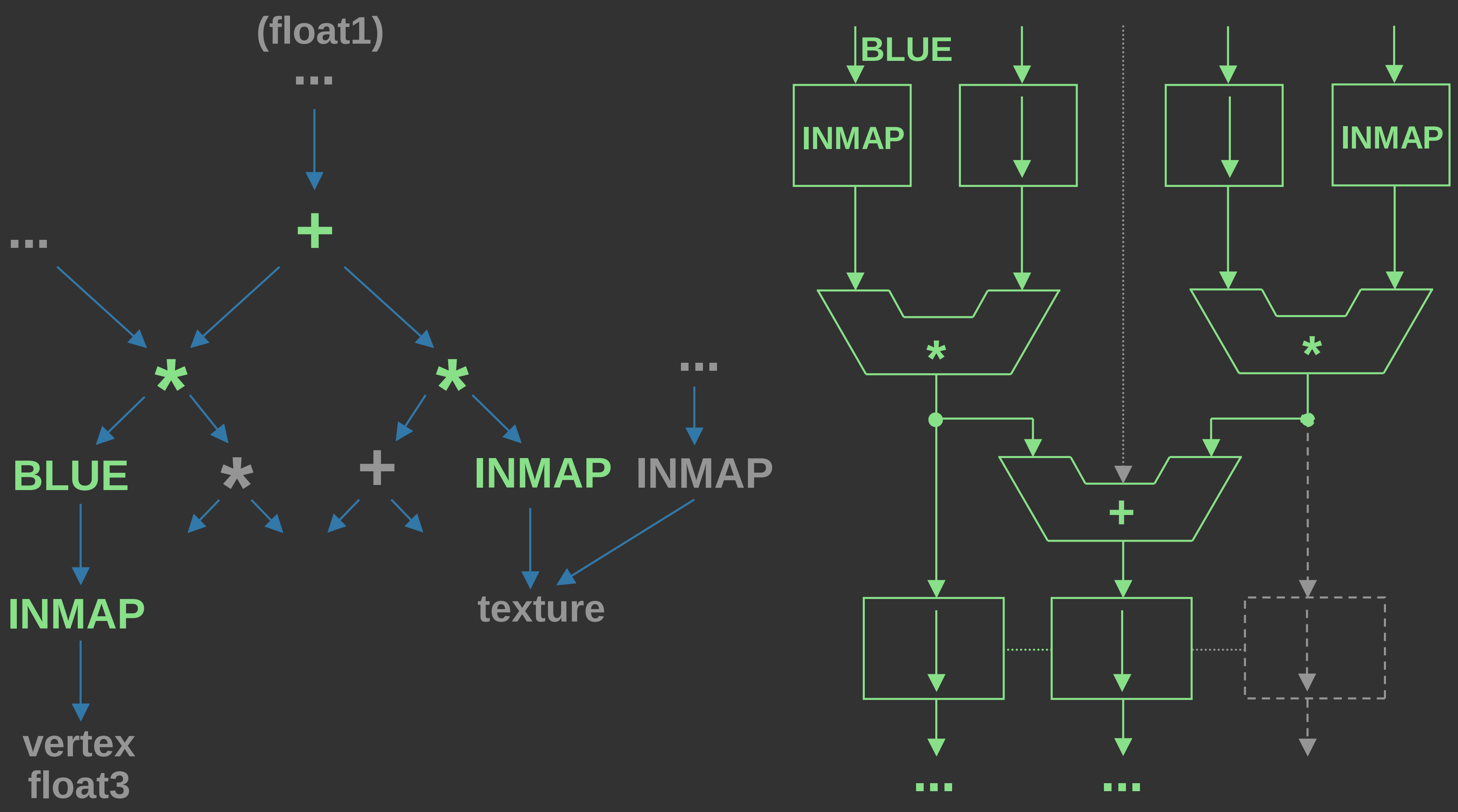
Mapping ops to a partial combiner



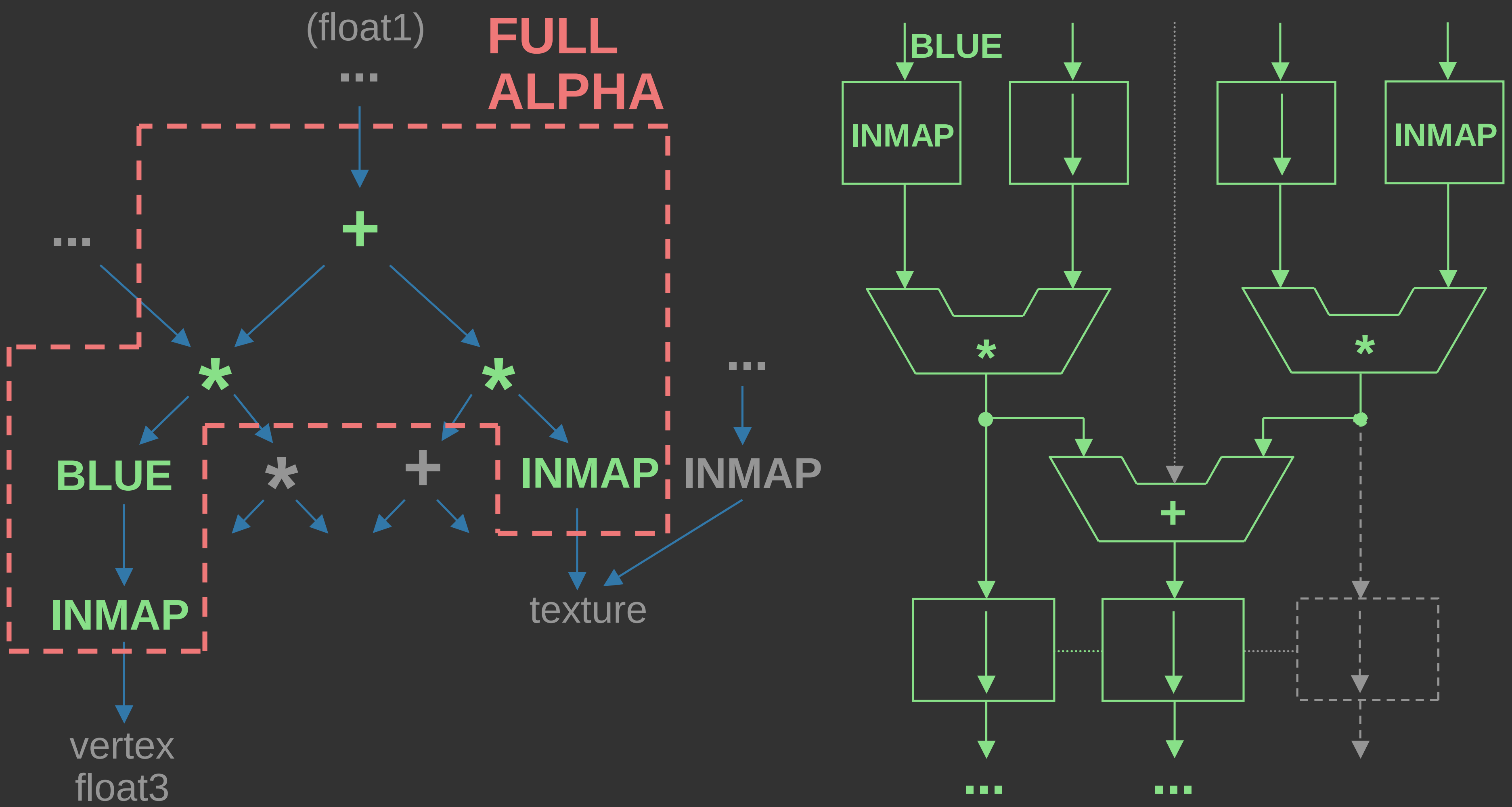
Mapping ops to a partial combiner



Mapping ops to a partial combiner



Mapping ops to a partial combiner



Allocate DAG inputs to registers

DAG inputs consist of:

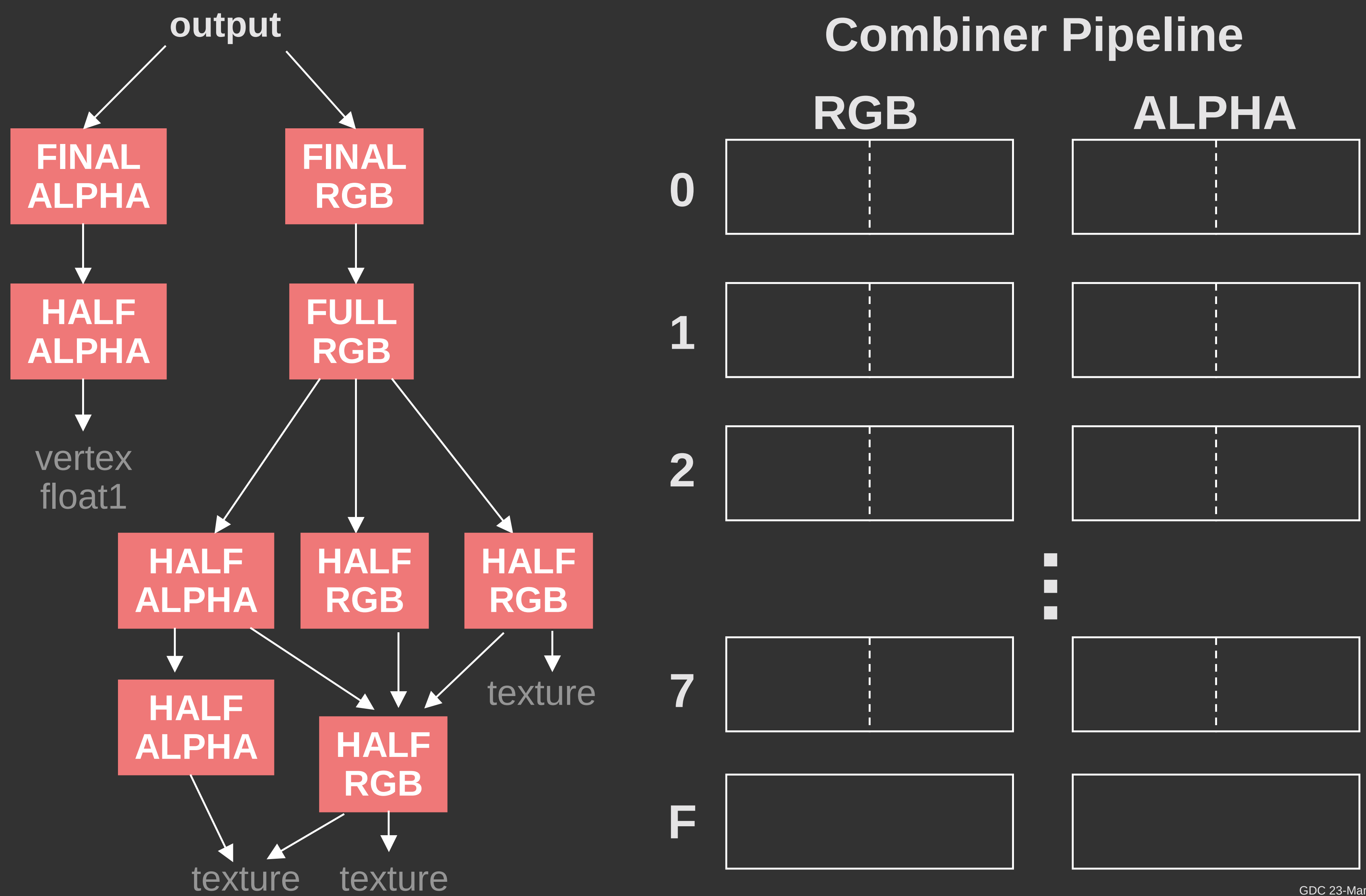
- textures
- interpolants from vertex values
- constants and “primitive group” values

Use a greedy algorithm -- do “hardest” cases first

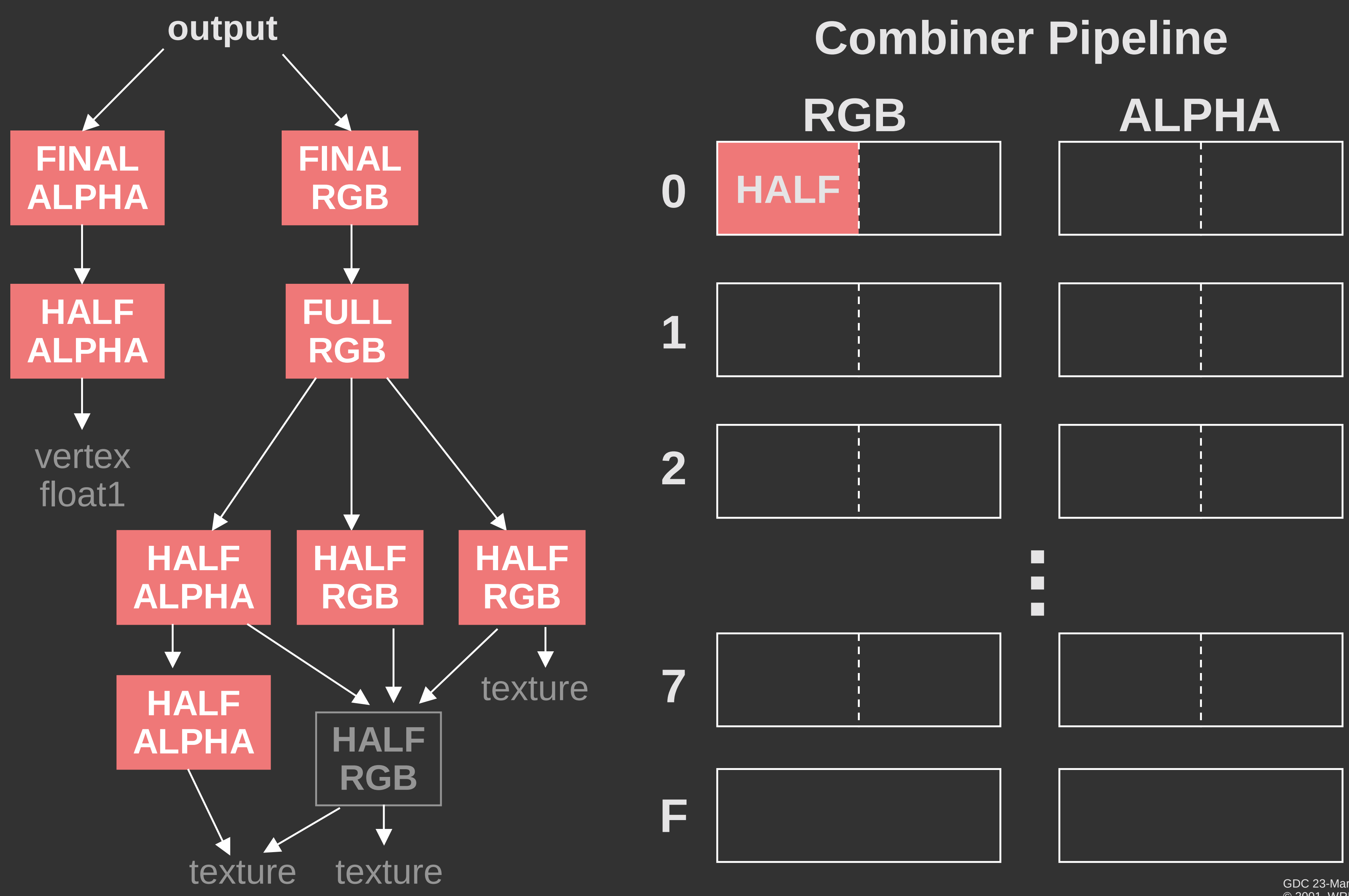
Some capabilites:

- pack unrelated 3-vector and scalar into RGBA
- put scalar in RGB
- use PASSTHRU texture for vertex interpolants

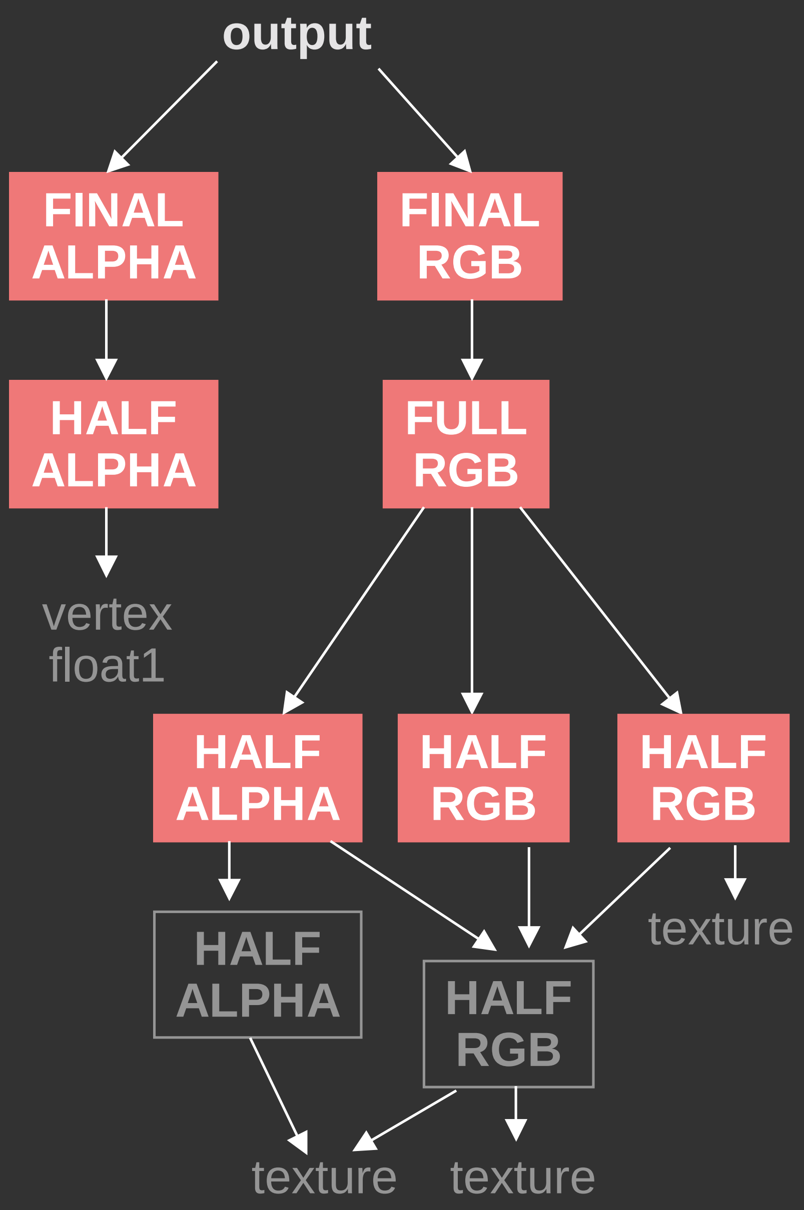
Map partial combiners to pipeline using bottom-up algorithm



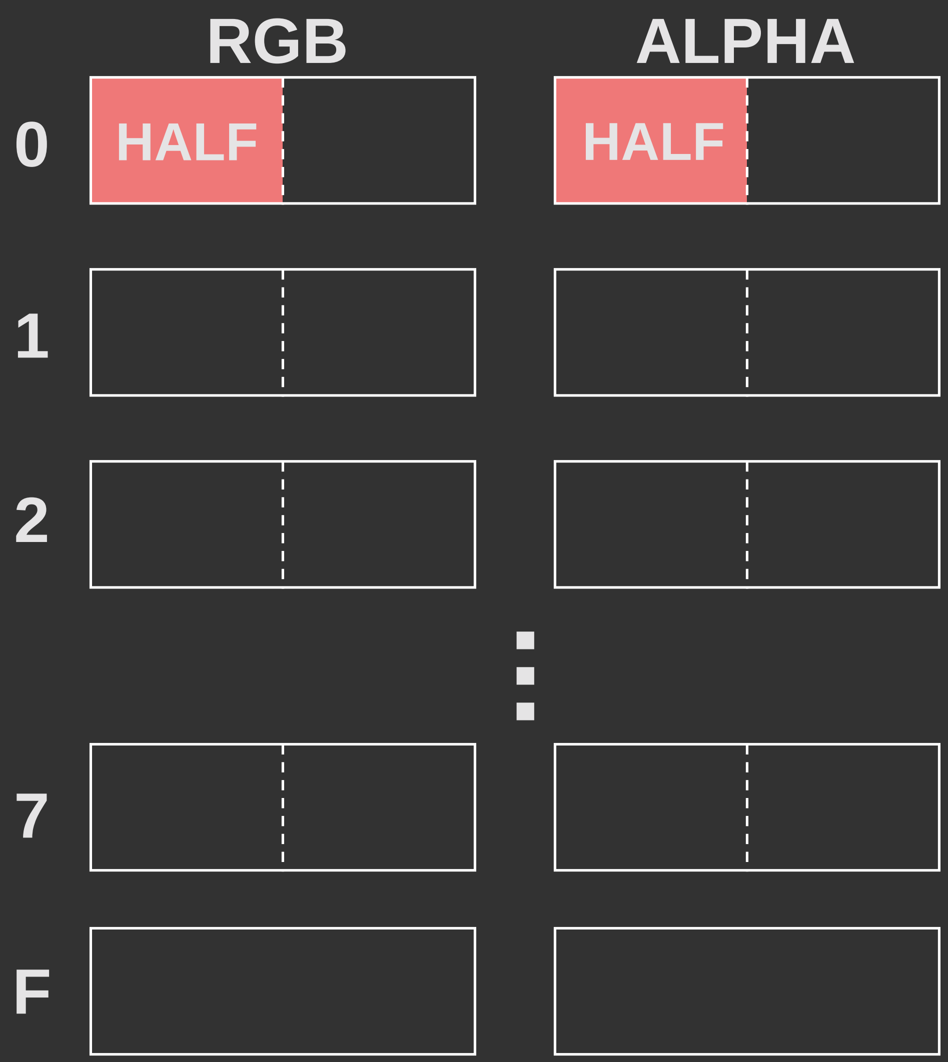
Map partial combiners to pipeline using bottom-up algorithm



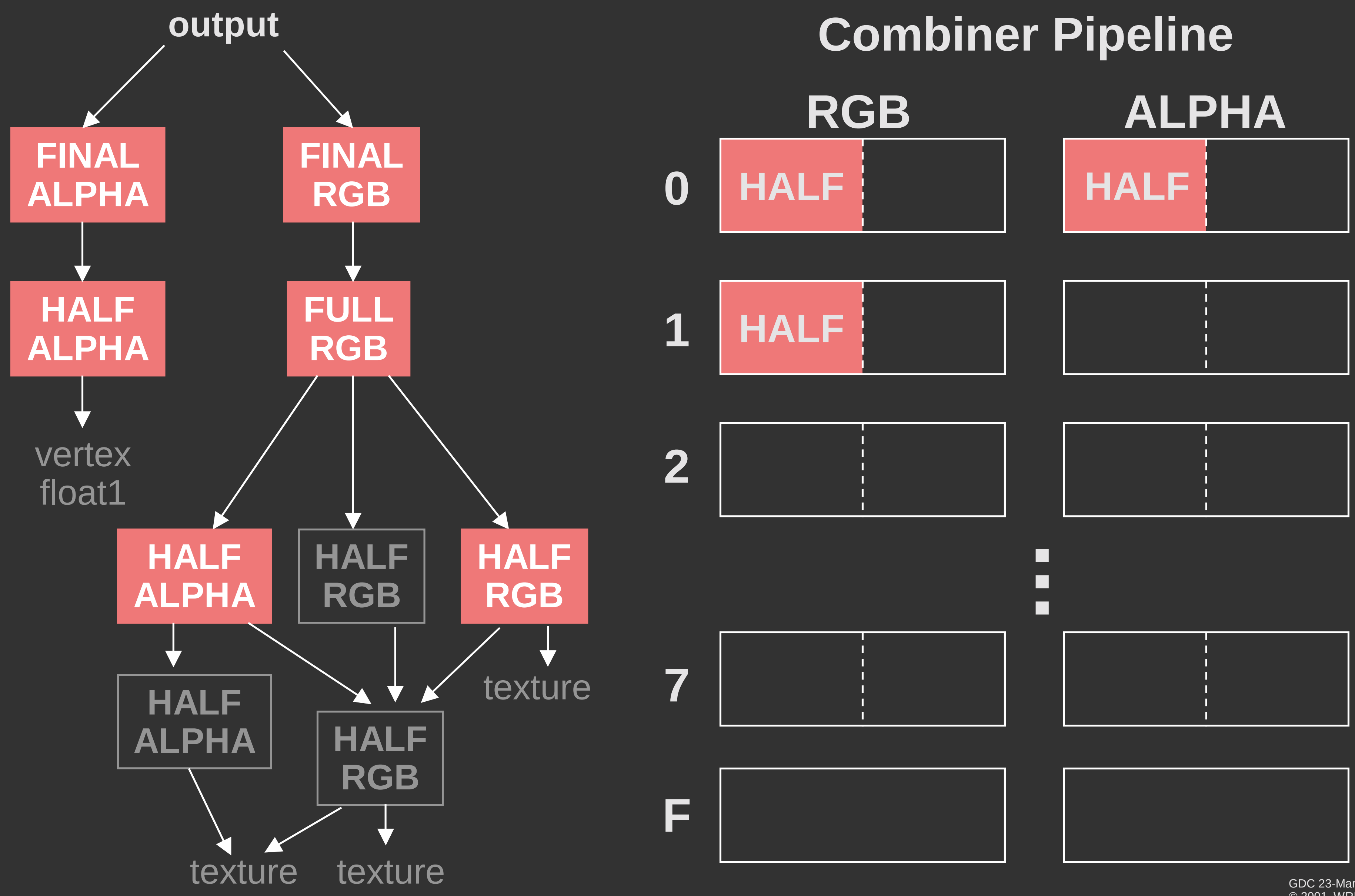
Map partial combiners to pipeline using bottom-up algorithm



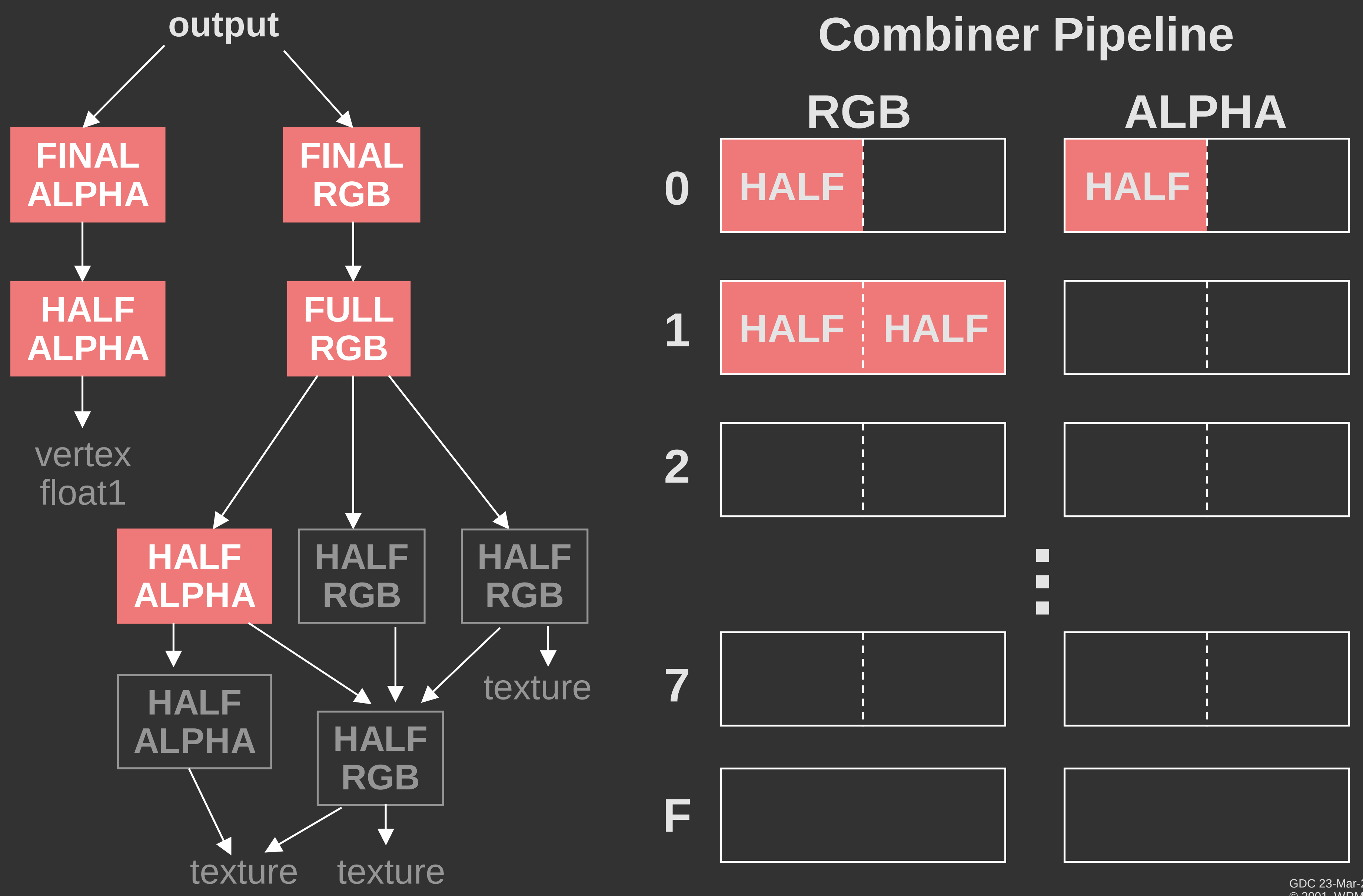
Combiner Pipeline



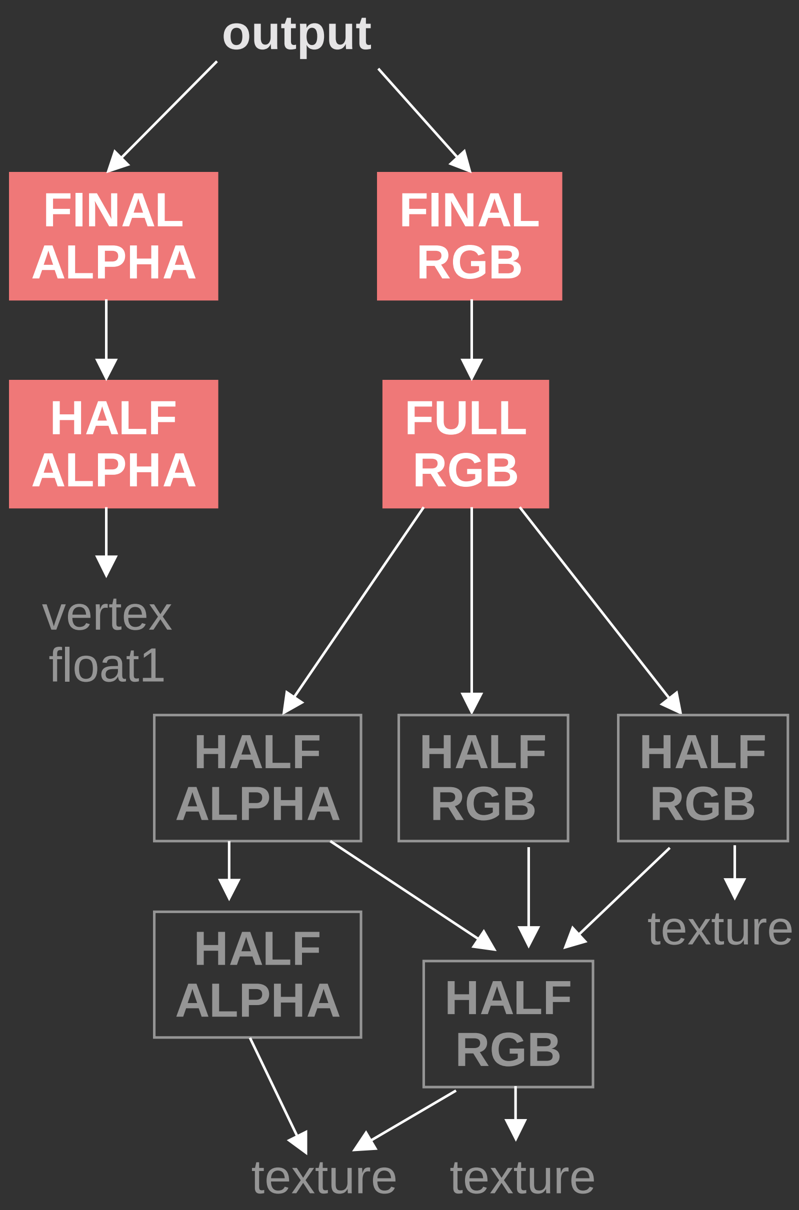
Map partial combiners to pipeline using bottom-up algorithm

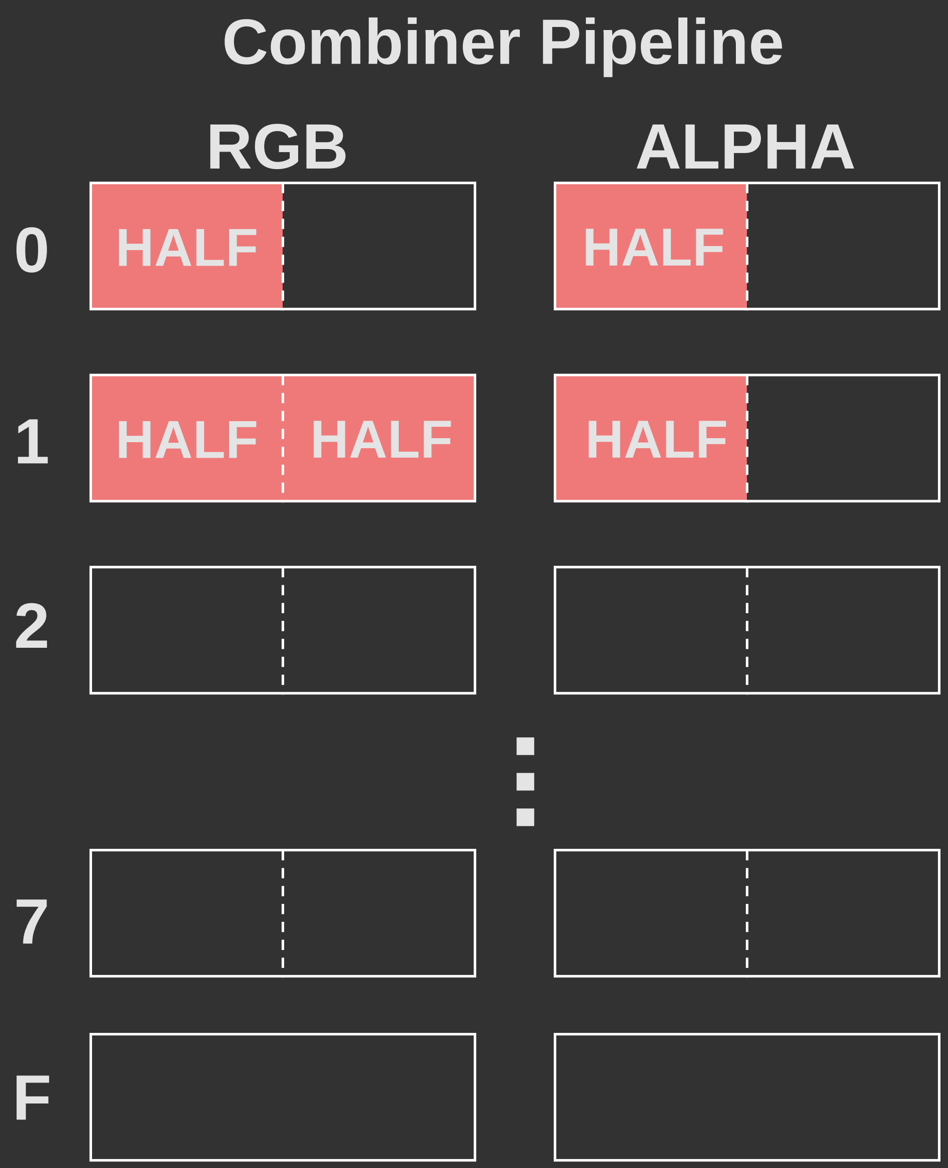


Map partial combiners to pipeline using bottom-up algorithm

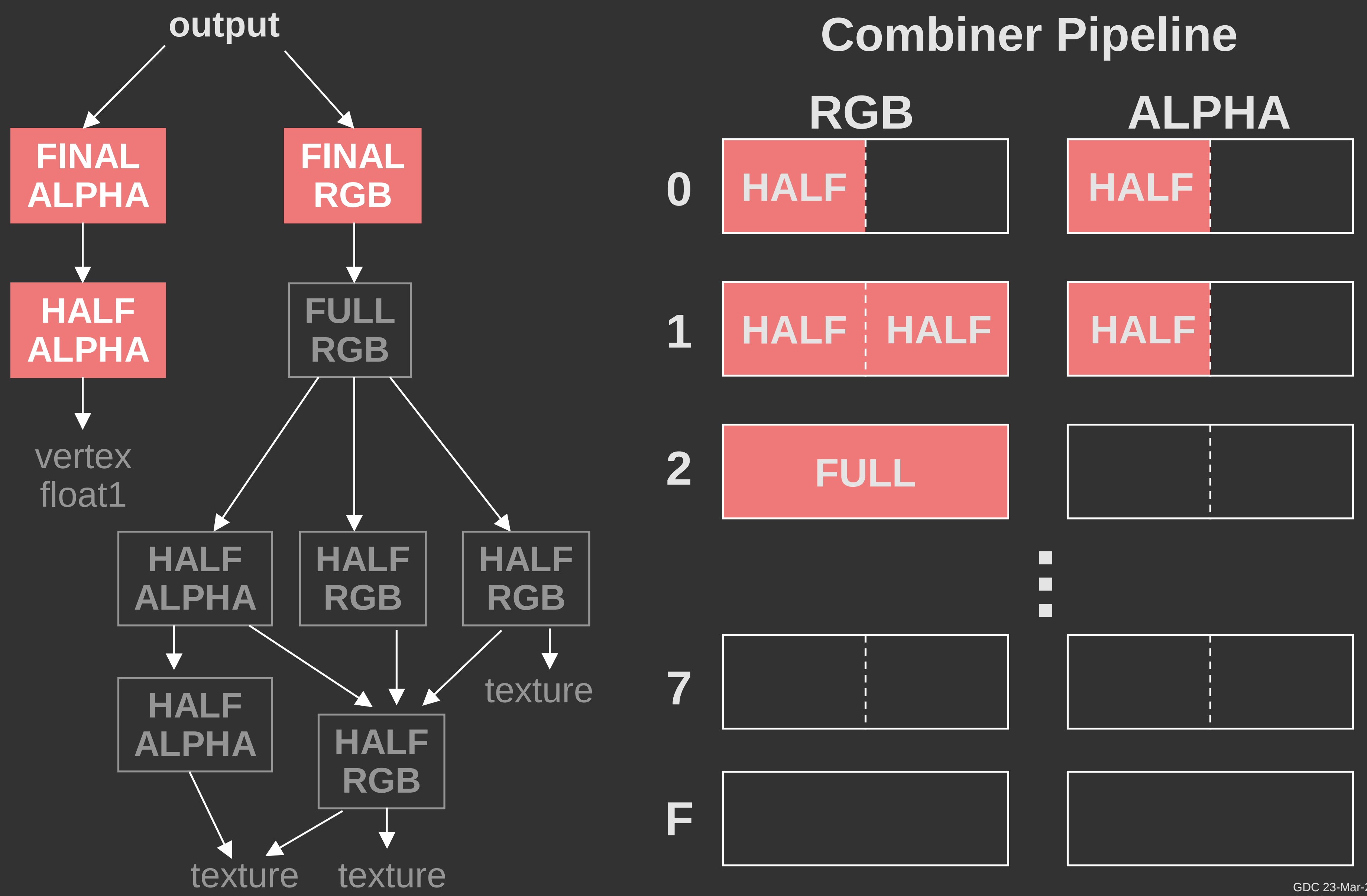


Map partial combiners to pipeline using bottom-up algorithm

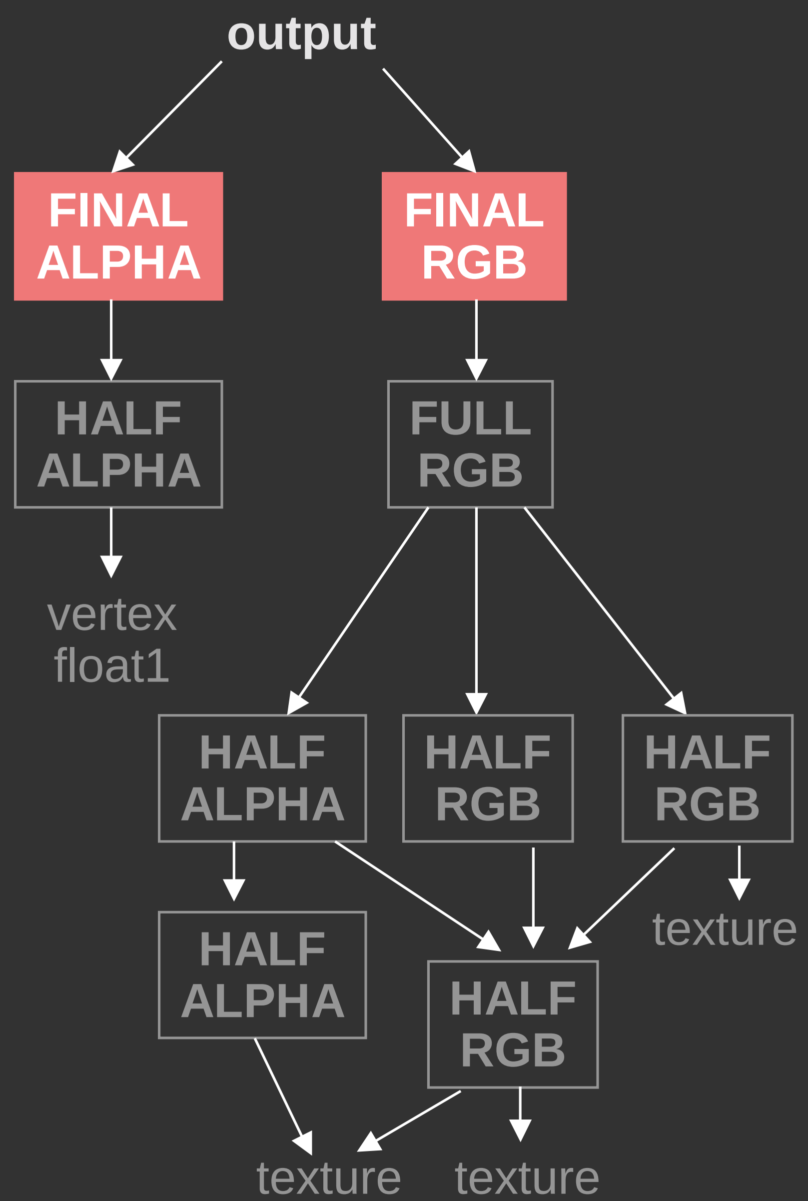


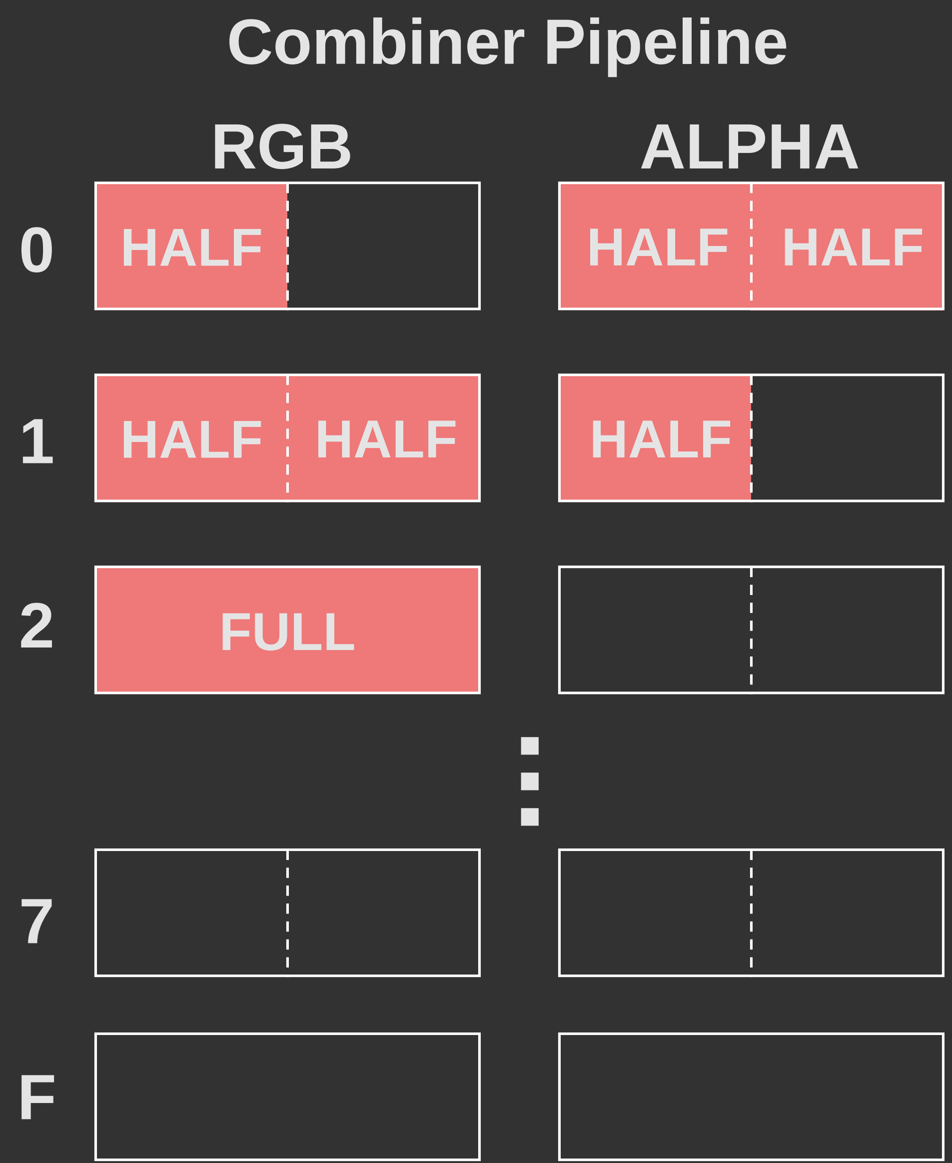


Map partial combiners to pipeline using bottom-up algorithm

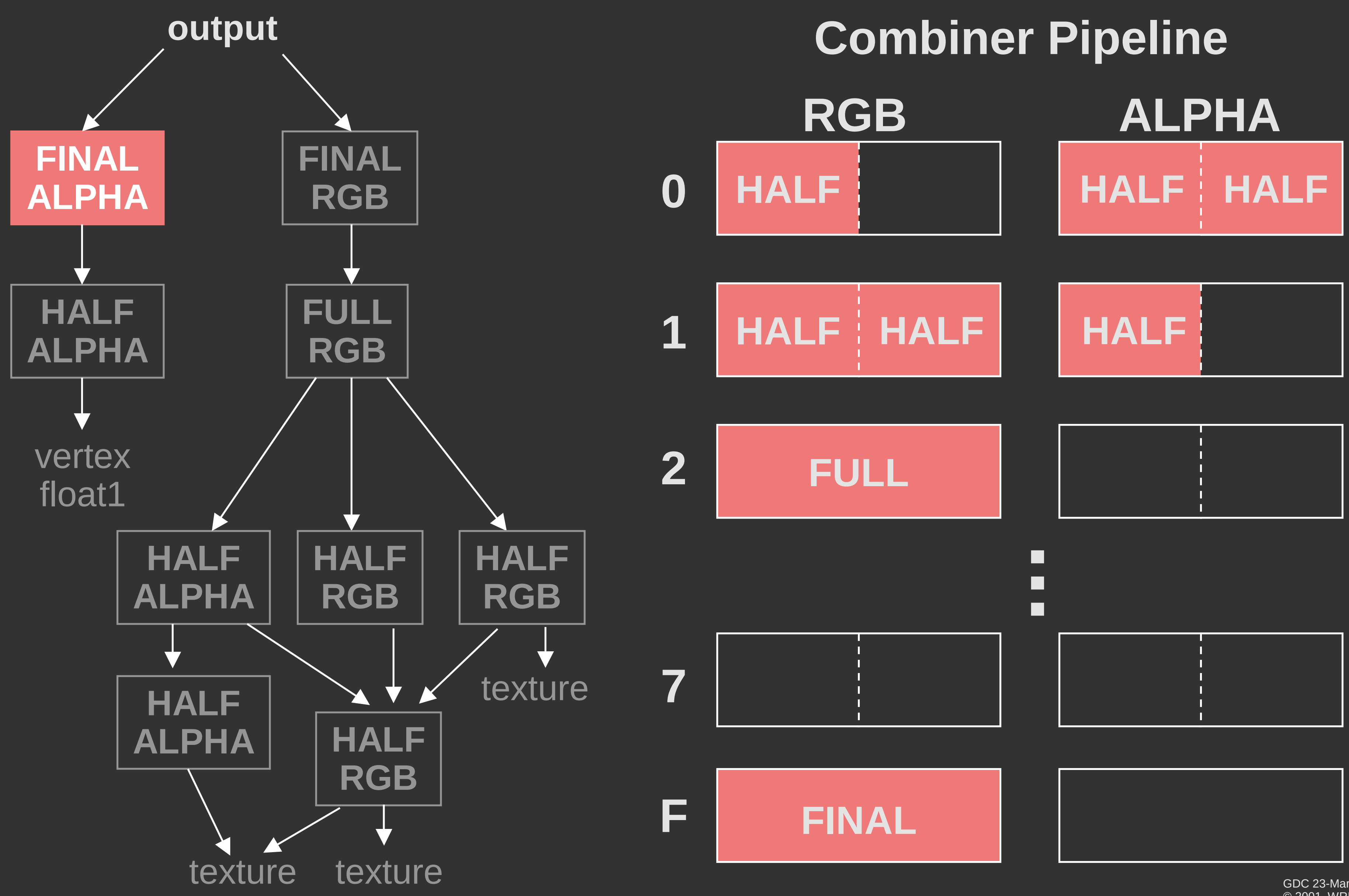


Map partial combiners to pipeline using bottom-up algorithm

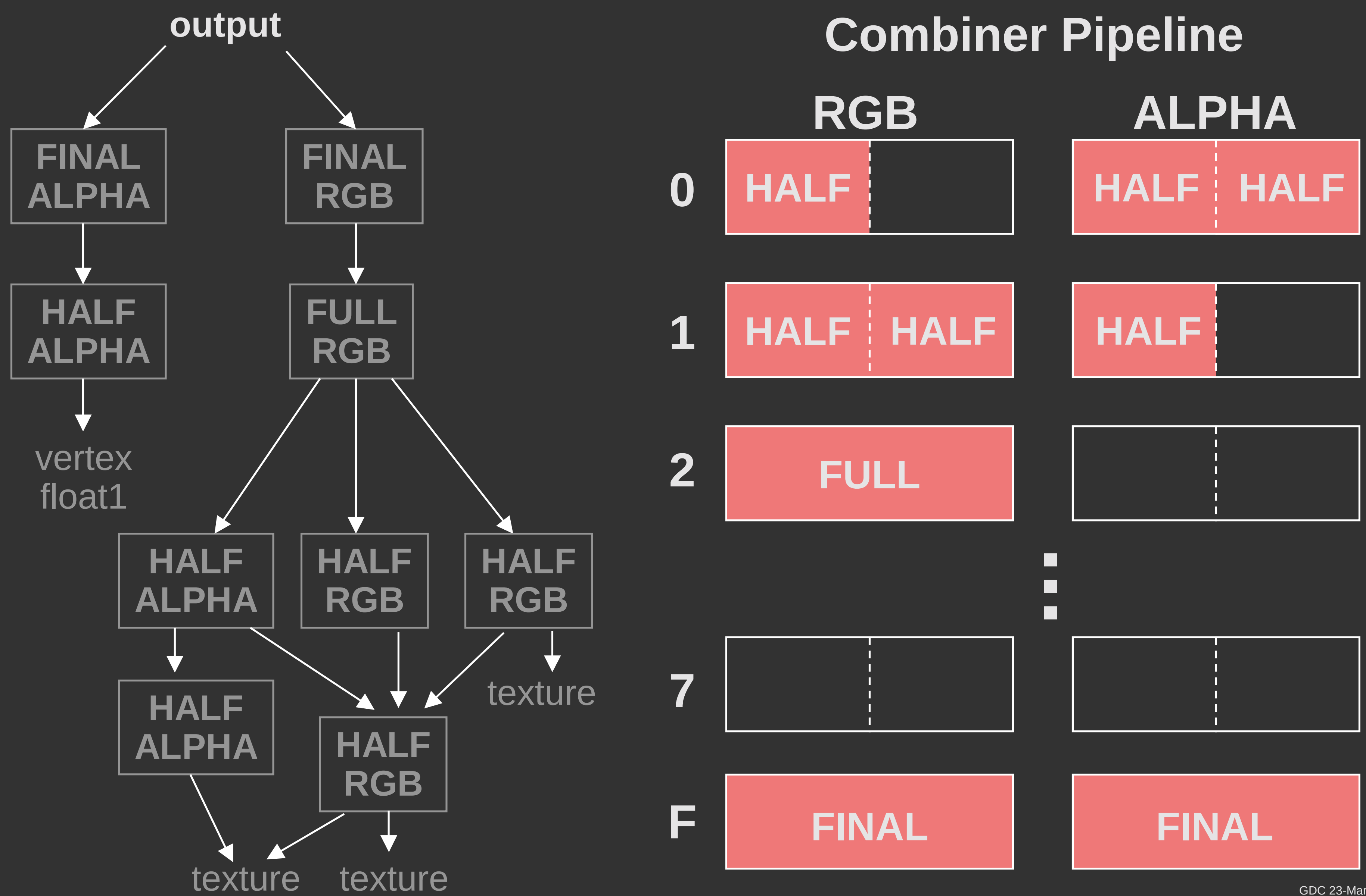




Map partial combiners to pipeline using bottom-up algorithm



Map partial combiners to pipeline using bottom-up algorithm



Compiler generates efficient code

Example: Bowling pin shader

- Initially 8 combiners
- Can reduce to 7 by using compiled code to guide source-code changes
- Can't do any better by hand – this is typical

What the compiler can't do:

- Reorder mathematical operations
- Reorganize textures (e.g. join RGB with A)
- Design algorithms that map well to combiners

RegComb backend limitations

- No texture address ops yet (that's coming).
- No automatic multi-pass
- Compiler doesn't hide varying numerical ranges

High-level comments

Two uses for this technology:

- Prototyping
- Final product

Design decision: Support multi-pass?

- Flexibility/portability vs. complexity/performance
- Problems with partially-transparent surfaces

More High Level Comments

Design decision: When to compile?

- Once
- Runtime: explicit or implicit

System is complex

- Ours: 18 months, ~45,000 lines of source

Summary

Real-time shading languages are powerful & addictive

- A step towards “Toy Story in real time”

Programmable shading can be efficient

- Compiler technology is the key
- Design entire system for real-time hardware

Hardware will continue to improve

- More functionality
- Cleaner architectures
- Higher performance

Thanks to people who helped

System design, coding, and demos

- Svetoslav Tzvetkov, Pat Hanrahan, Pradeep Sen, Ren Ng

Sponsors

- ATI, NVIDIA, SGI, SONY, Sun, 3dfx
- DARPA

Special thanks to

- Matt Papakipos, Mark Kilgard
- David Ebert

More information on the web

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

- Download system
  (binary only, but includes linkable library)
- Draft copy of SIGGRAPH 2001 paper

Questions?