Lecture 5:

Programmability

EEC 277, Graphics Architecture
John Owens, UC Davis, Winter 2017

Transition in Graphics Systems

- Distant past
  - Fixed-function pipelines
  - Feature-based interfaces
- Recent/Now
  - "Limited" programmability: programmable shading, SPMD programming model
  - Assembly-language interfaces → shading languages → programming systems
- Now/Emerging
  - General programmability: programmable graphics

Review: the rendering equation *

[Kajiya 86]

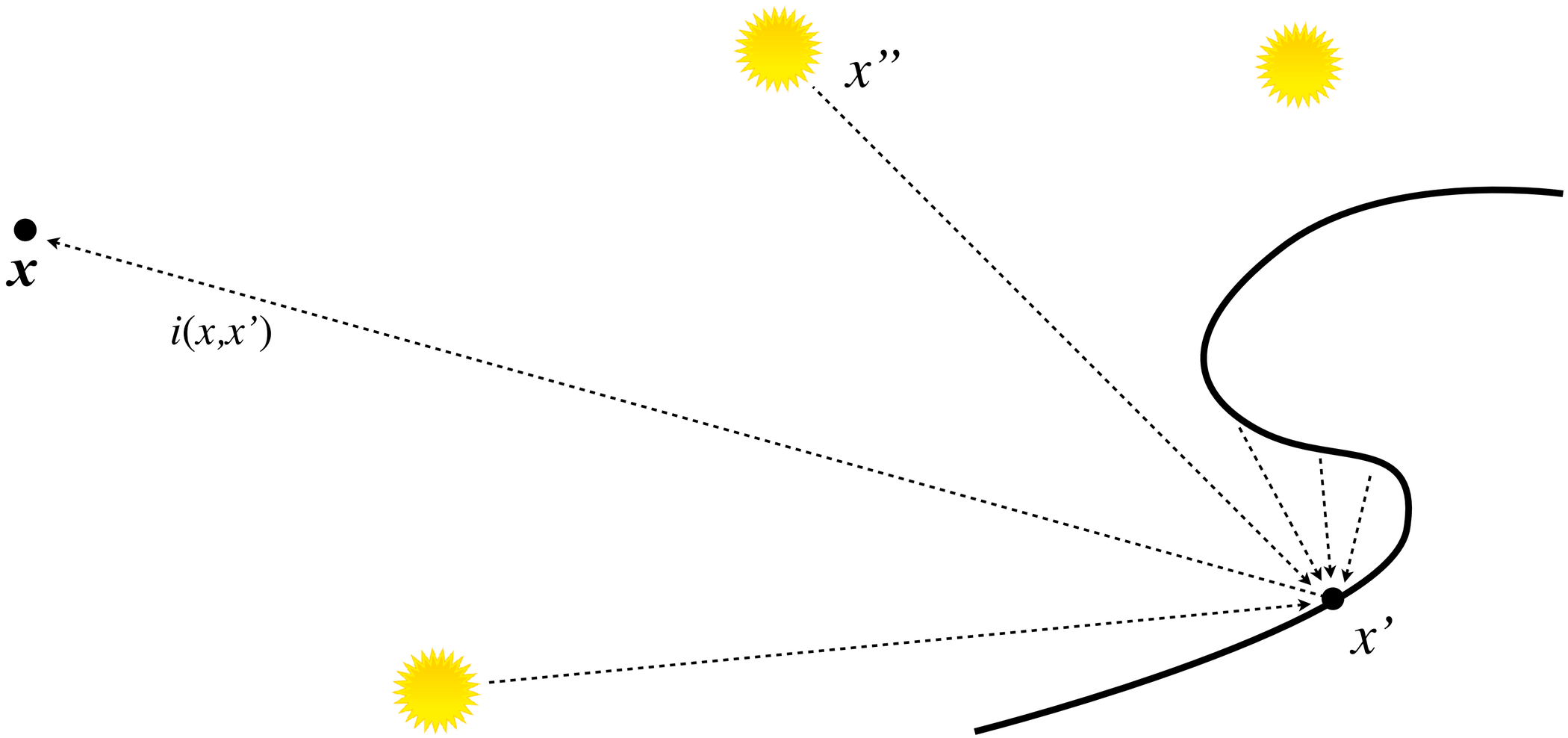
$$i(x,x') = v(x,x')\left[l(x,x') + \int r(x,x',x'')i(x',x'')\,dx''\right]$$

$i(x,x')$ = Radiance (energy along a ray) from point x′ in direction of point x

$v(x,x')$ = Binary visibility function (1 if ray from x′ reaches x, 0 otherwise)

$l(x,x')$ = Radiance emitted from x′ in direction of x (if x′ is an emitter)

$r(x,x',x'')$ = BRDF: fraction of energy arriving at x′ from x″ that is reflected in direction of x

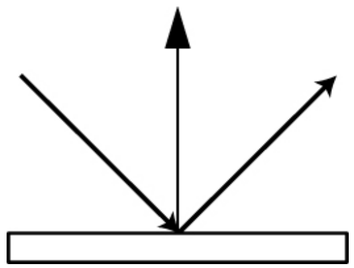


* Note: using notation from Hanrahan 90

Categories of reflection functions

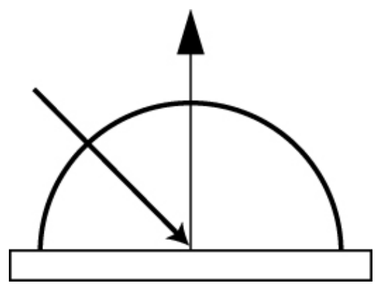
- **Ideal specular**

  Perfect mirror



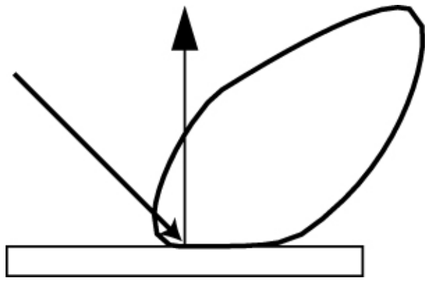
- **Ideal diffuse**

  Uniform reflection in all directions



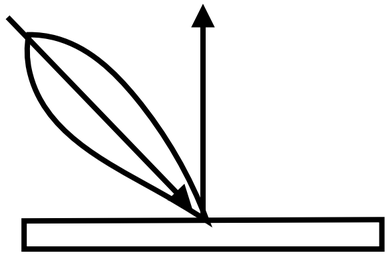
- **Glossy specular**

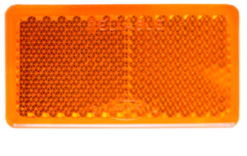
  Majority of light distributed in reflection direction



- **Retro-reflective**

  Reflects light back toward source





Diagrams illustrate how incoming light energy from given direction is reflected in various directions.

Materials: diffuse

[Slide credit: Matt Pharr]

Materials: plastic

[Slide credit: Matt Pharr]

Materials: red semi-gloss paint

[Slide credit: Matt Pharr]

Materials: mirror

[Slide credit: Matt Pharr]

Materials: gold

[Slide credit: Matt Pharr]

Materials

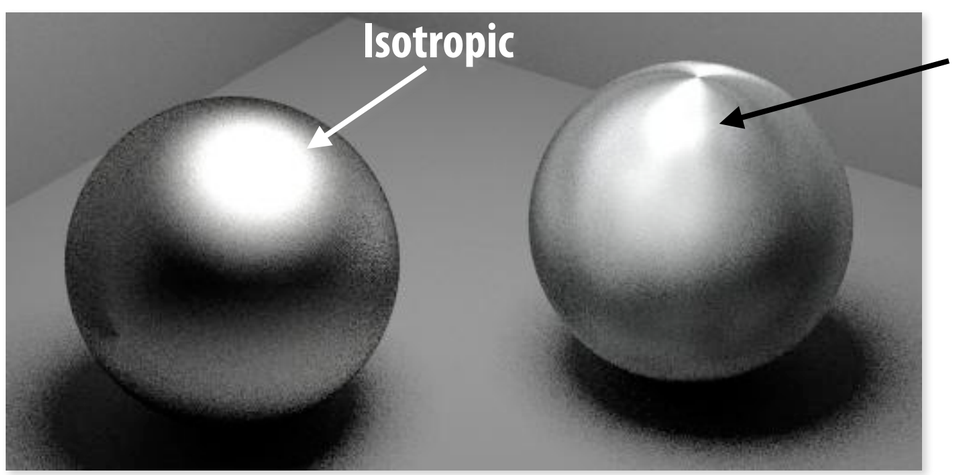
[Image credit: Jakob et al. 2014]

More complex materials

[Images from Lafortune et al. 97]

Fresnel reflection: reflectance is a function of viewing angle (notice higher reflectance near grazing angles)

[Images from Westin et al. 92]



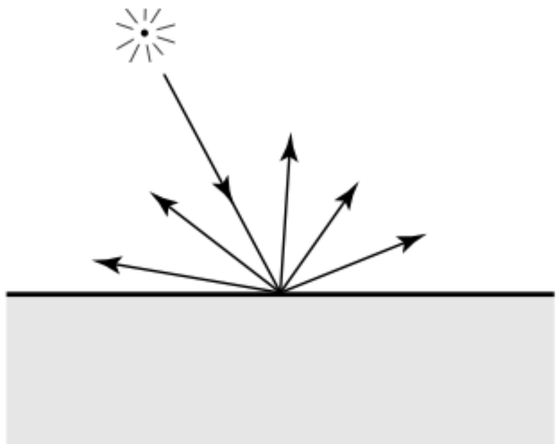
Anisotropic reflection: reflectance depends on azimuthal angle (e.g., oriented microfacets in brushed steel)

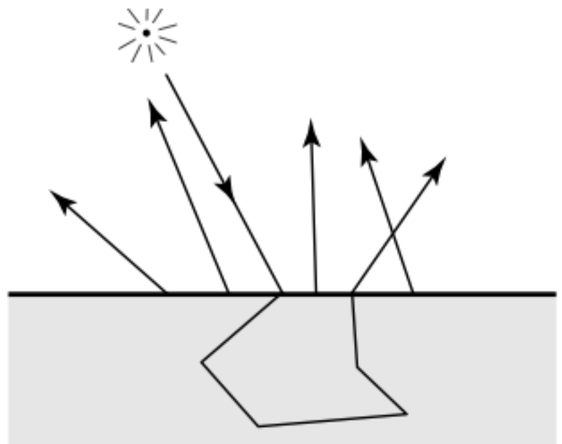
Subsurface scattering materials

[Wann Jensen et al. 2001]

BRDF

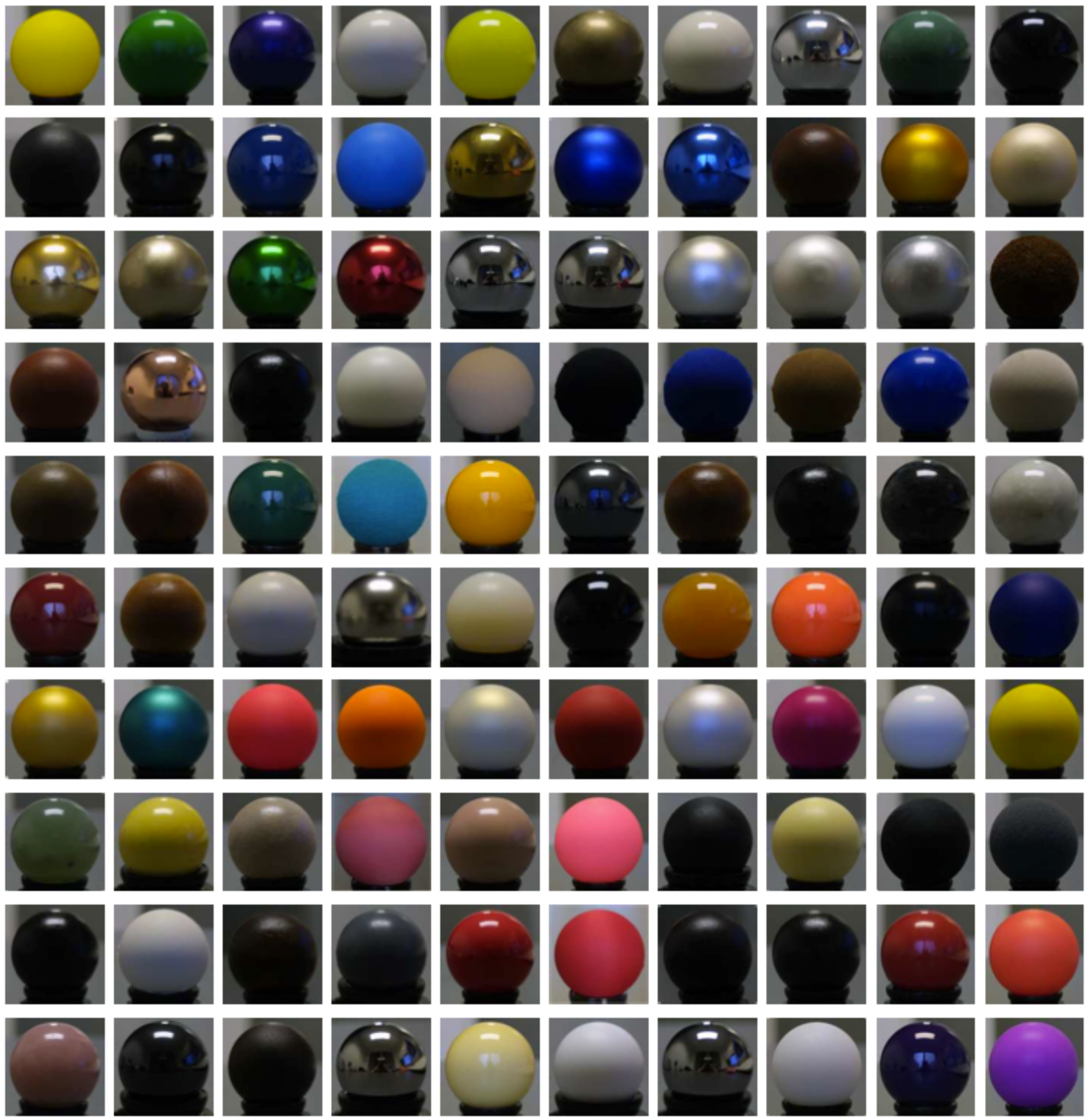
BSSRDF





- **Account for scattering inside surface**
- **Light exits surface from different location it enters**
  - **Very important to appearance of translucent materials (e.g., skin, foliage, marble)**

More materials



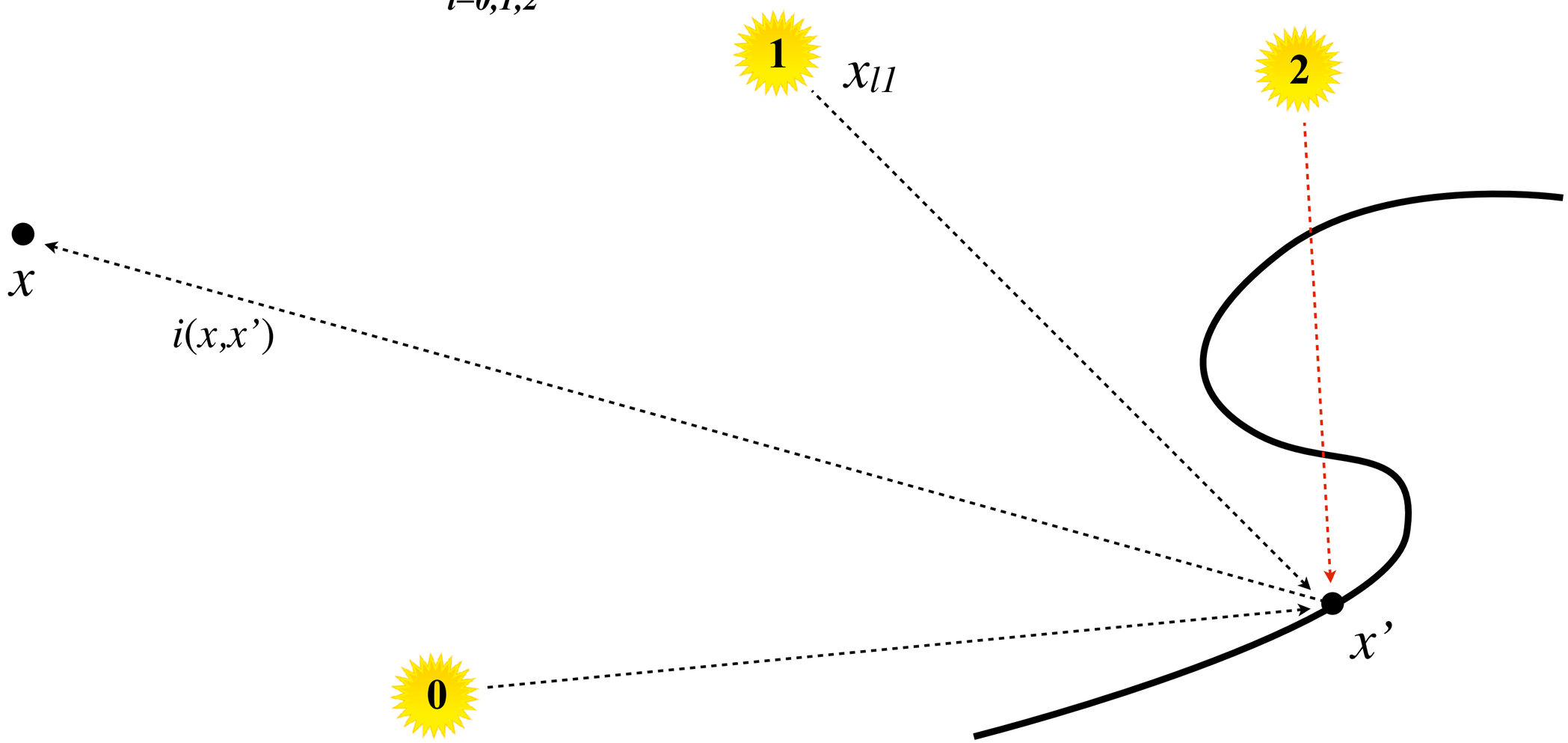
Tabulated BRDFs

Images from Matusik et al. SIGGRAPH 2003

Simplification of the rendering equation

- All light sources are point sources (light i emits from point x_{li})
- Lights emit equally in all directions: radiance from light i: $i(x', x_{l_i}) = L_i$
- Direct illumination only: illumination of x′ comes directly from light sources

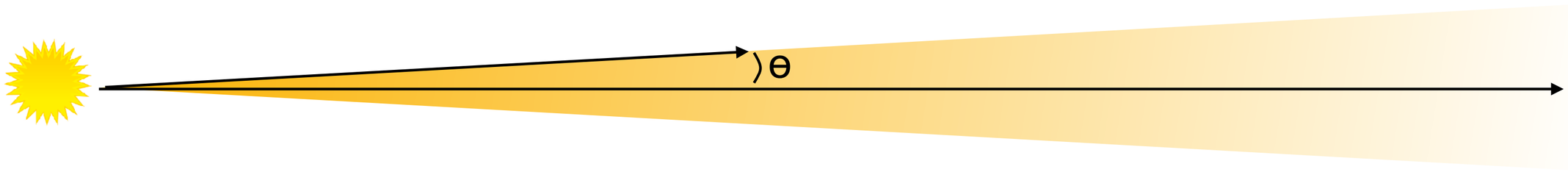
$$i(x,x') = \sum_{i=0,1,2} L_i v(x', x_{l_i}) r(x, x', x_{l_i})$$



More light types

- **Attenuated omnidirectional point light**
  (emits equally in all directions, intensity falls off with distance: $1/R^2$ falloff)

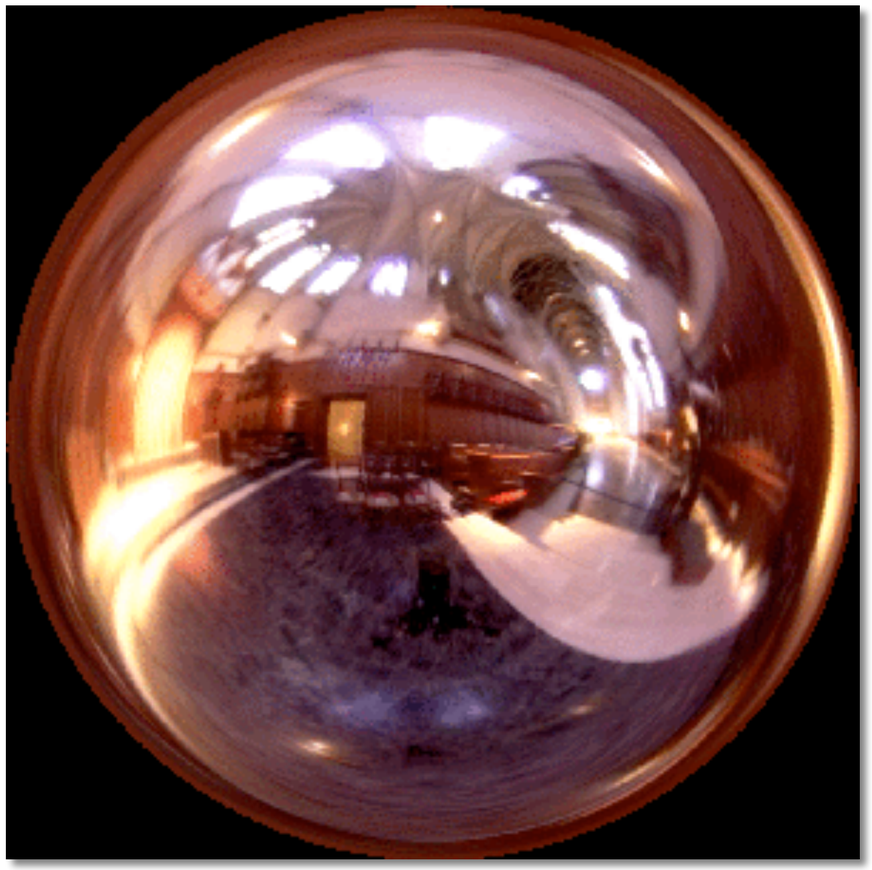
- **Spot light**
  (does not emit equally in all directions)



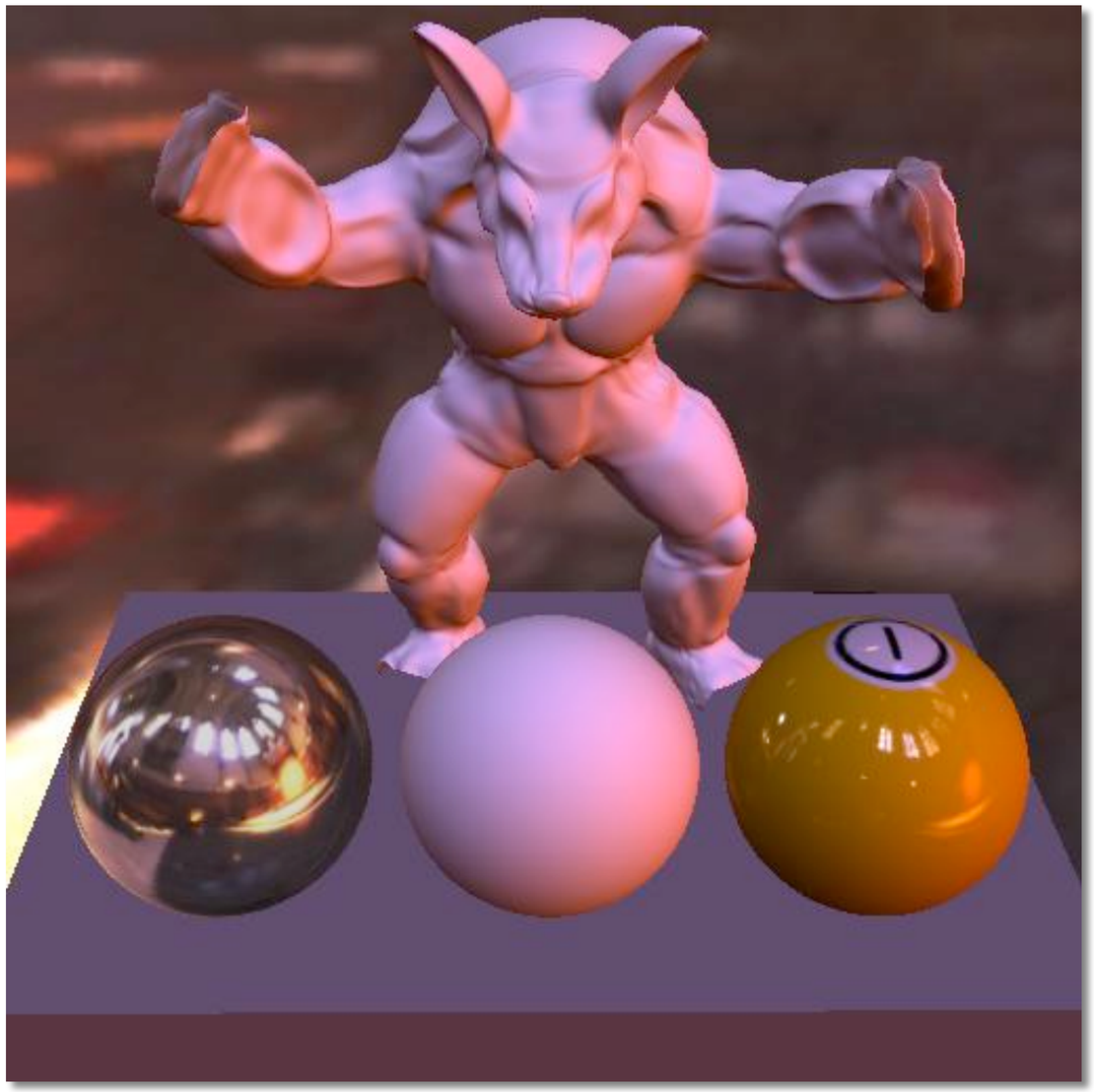
More sophisticated lights

- **Environment light**

  (not a point light source: defines incoming light from all directions)



Environment Map
(Grace Cathedral)



Rendering using environment map
(pool balls have varying material properties)
[Ramamoorthi et al. 2001]

Parameterized materials and lighting in early OpenGL

(prior to programmable shading)

- `glLight(light_id, parameter_id, parameter_value)`
  - 10 parameters (e.g., ambient/diffuse/specular color, position, direction, attenuation coefficient)

- `glMaterial(face, parameter_id, parameter_value)`
  - Face specifies front or back facing geometry
  - Parameter examples (ambient/diffuse/specular reflectance, shininess)
  - Material value could be modulated by texture data

- Parameterized shading function evaluated at each vertex
  - Summation over all enabled lights
  - Resulting per-vertex color modulated by result of texturing

Siggraph ’84 and Pixar/Lucasfilm

“Plants, Fractals, and Formal Languages”, Alvy Ray Smith

“Chap—A SIMD Graphics Processor”, Levinthal and Porter

“Distributed Ray Tracing”, Cook, Porter, and Carpenter

“The A-Buffer”, Loren Carpenter

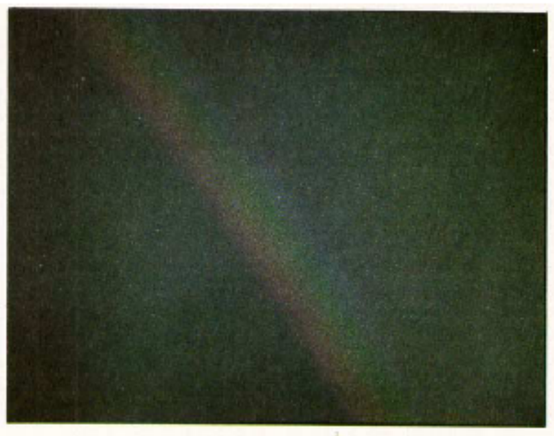
“Shade Trees”, Rob Cook

“Compositing Digital Images”, Porter and Duff

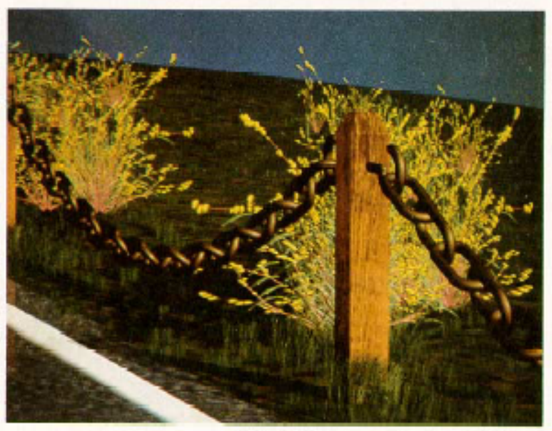
Road to Point Reyes

Foreground = *FrgdGrass* **over** *Rock* **over** *Fence* **over** *Shadow* **over** *BkgdGrass*;

Hillside = *Plant* **over** *GlossyRoad* **over** *Hill*;



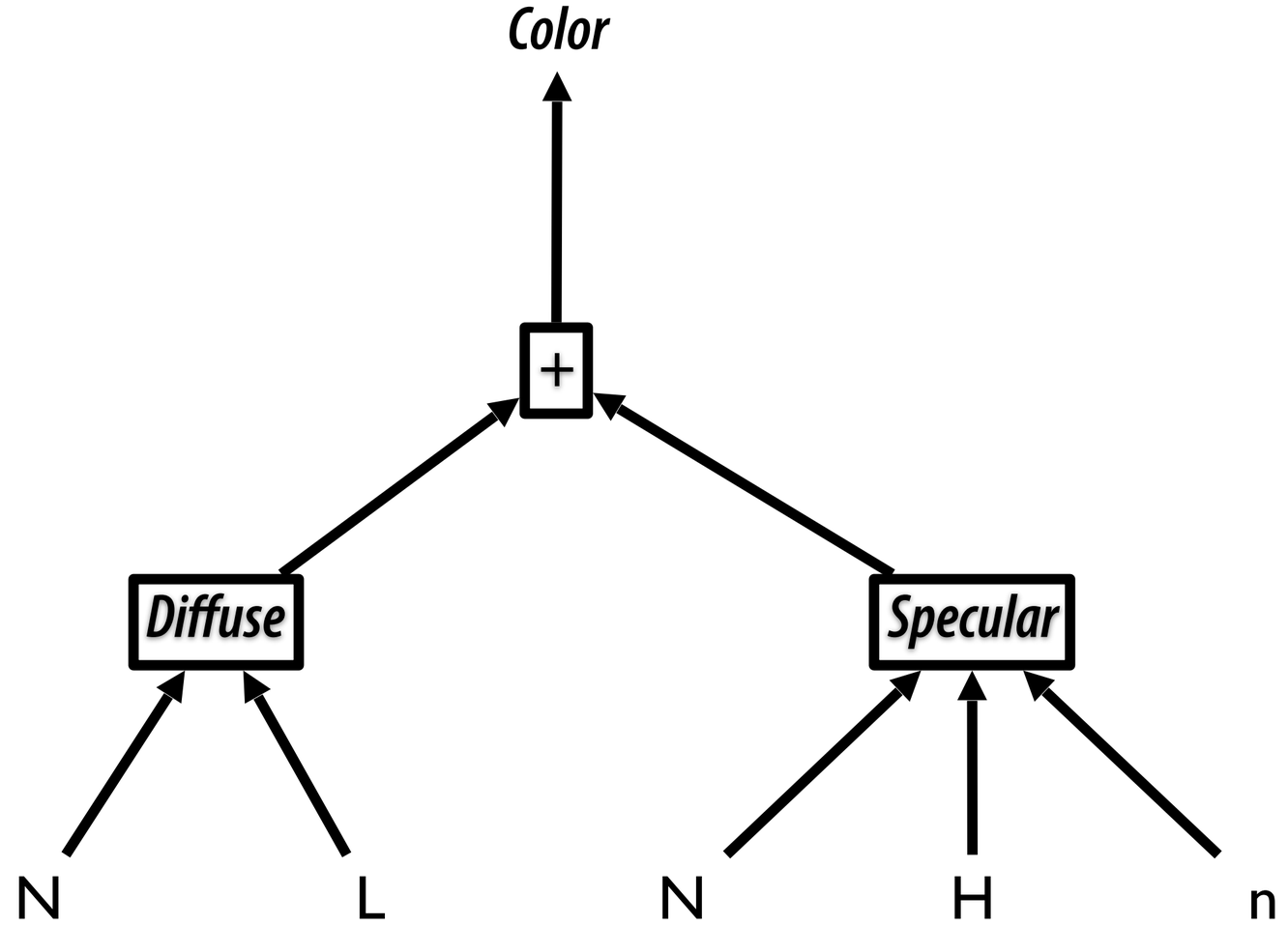
Background = *Rainbow* **plus** *Darkbow* **over** *Mountains* **over** *Sky*;



Pt.Reyes = *Foreground* **over** *Hillside* **over** *Background*.

[Lucasfilm, "Road to Point Reyes"]

Shade Trees



Cook: Arbitrary Trees

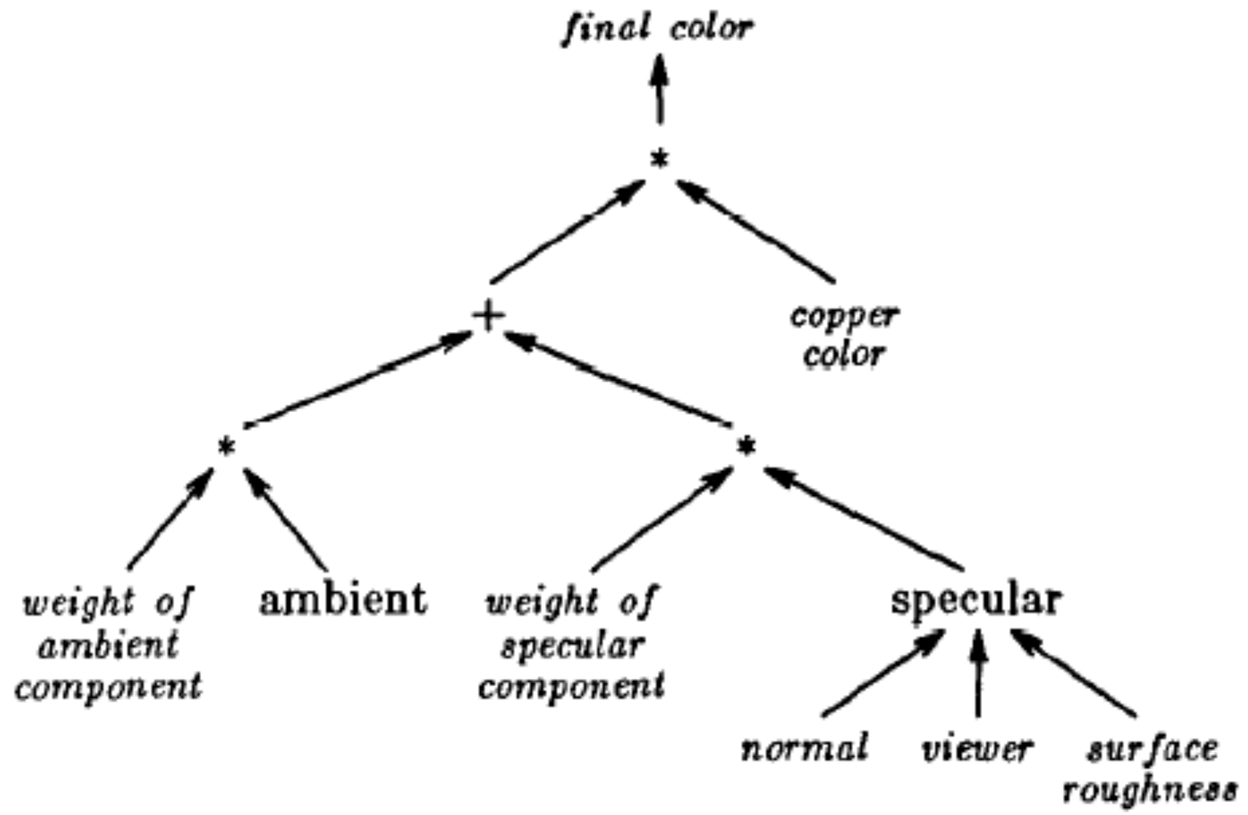


Figure 1a. Shade tree for copper.

Cook: Arbitrary Trees

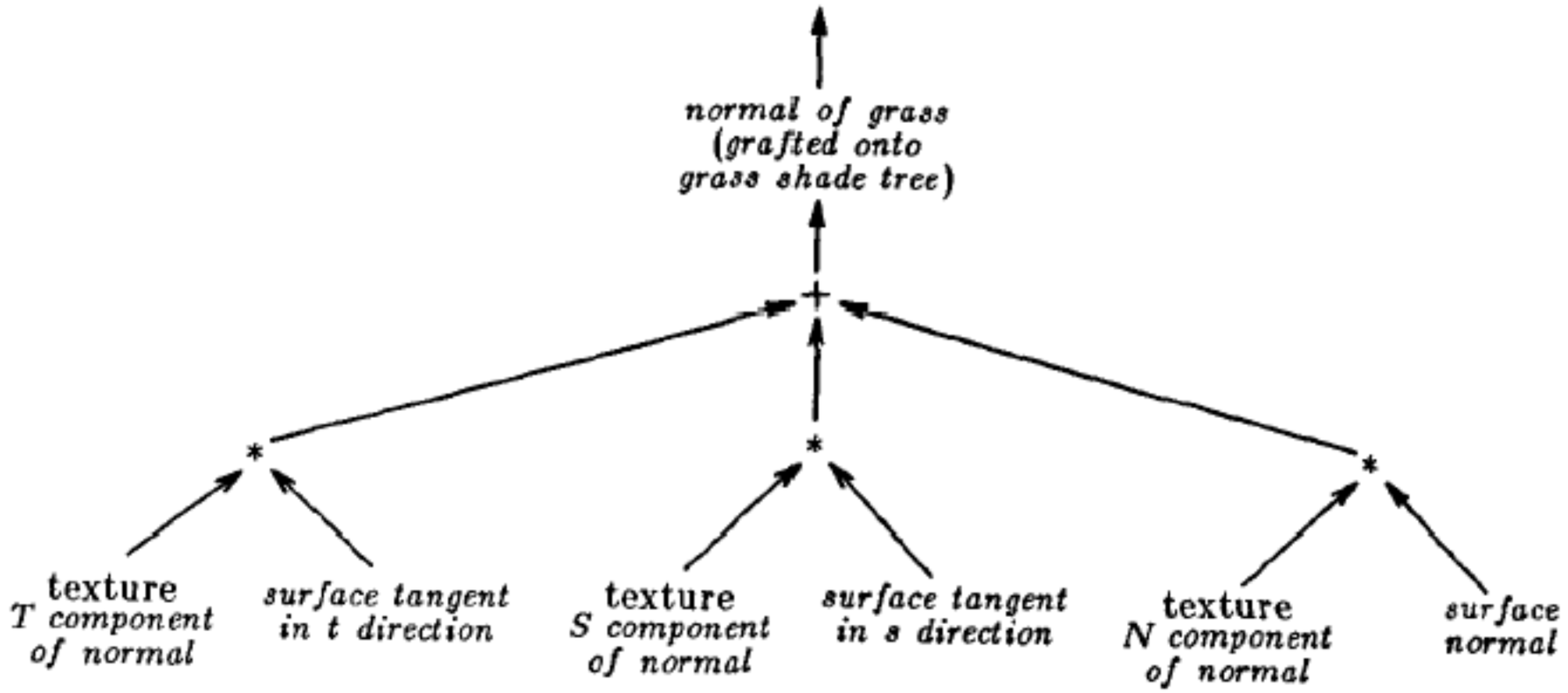
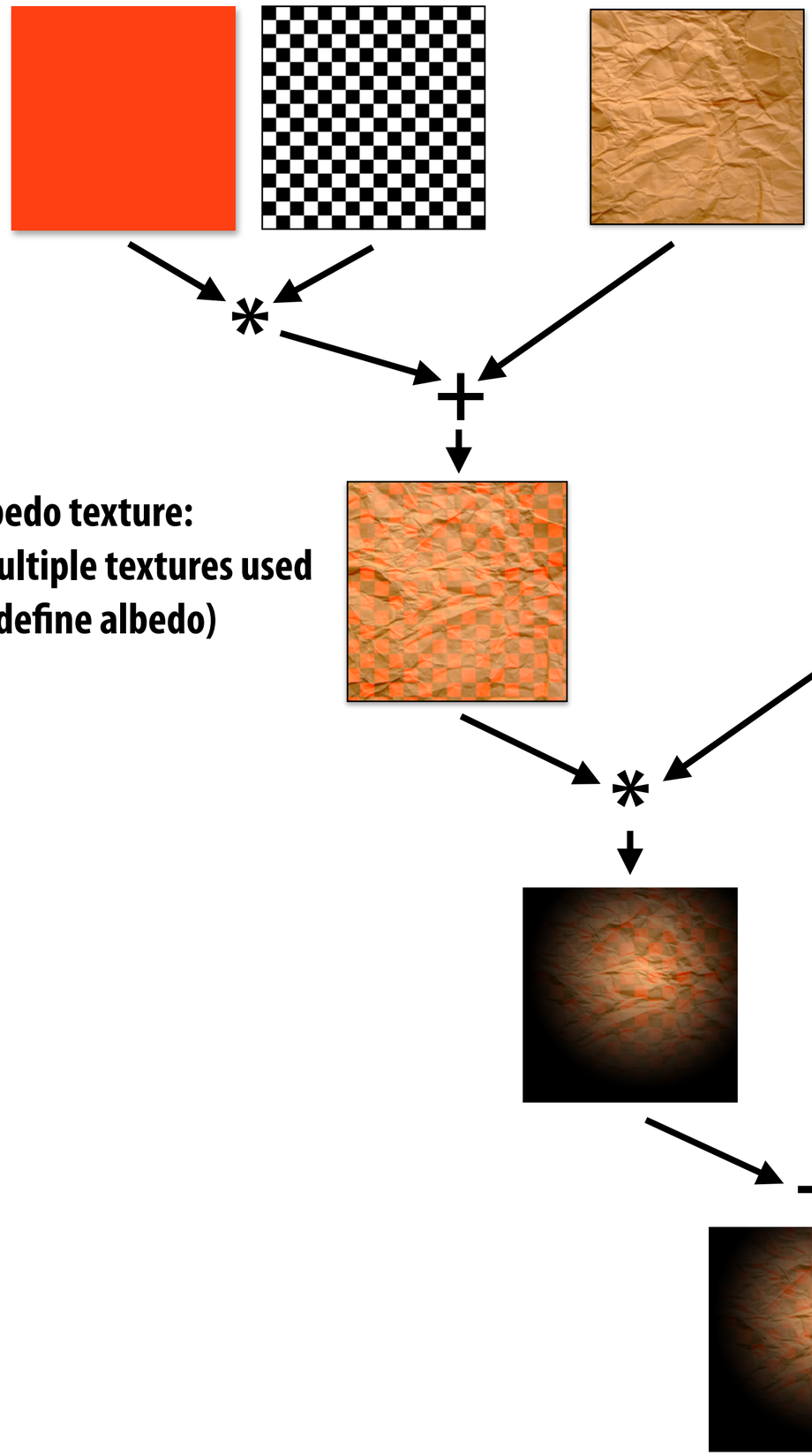


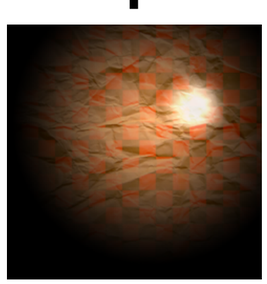
Figure 1c. Textured grass normal.

Precursor to shading languages: shade trees

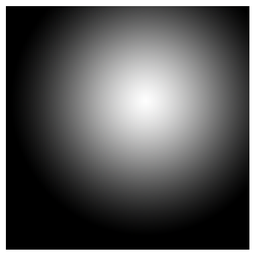
[Cook 84]



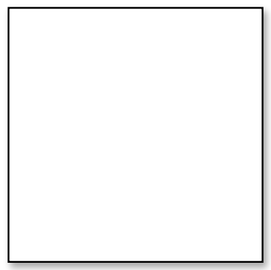
Albedo texture:
(multiple textures used to define albedo)



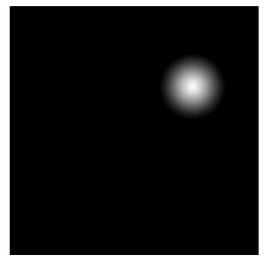
Diffuse reflectance (N dot L)



Material:
specular reflection coeff



Specular reflectance



Note: shade tree is a declarative abstraction!

Cook Approach

- Shader:
  - Basic operations: dot products, norms, etc.
- Operations organized into trees
- Separated light source specification, surface reflectance, and atmospheric effects
- Multiple trees: shade trees, light trees, atmosphere trees, displacement maps
- Simple language

“An Image Synthesizer”, Ken Perlin

Extended Cook shading (Siggraph ‘85)

Shading as postprocess after visibility

- Difficult to do light transport algorithms

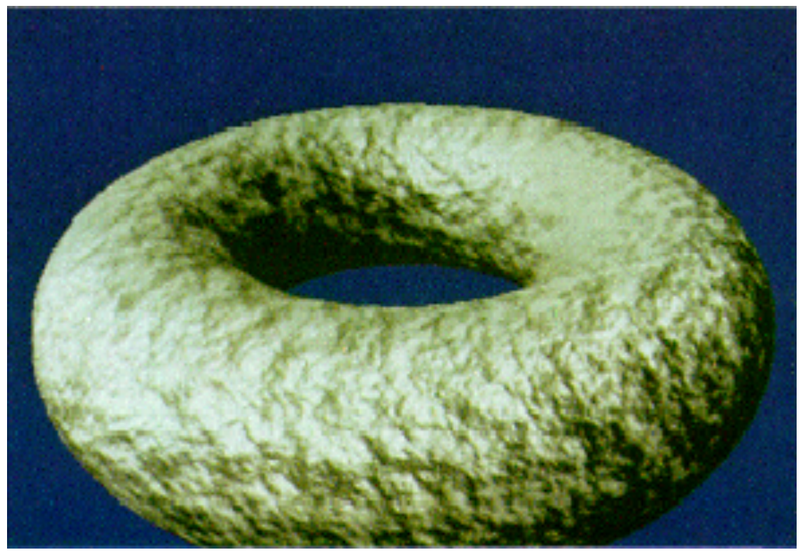
“Pixel Stream Editor”—more general control flow, language

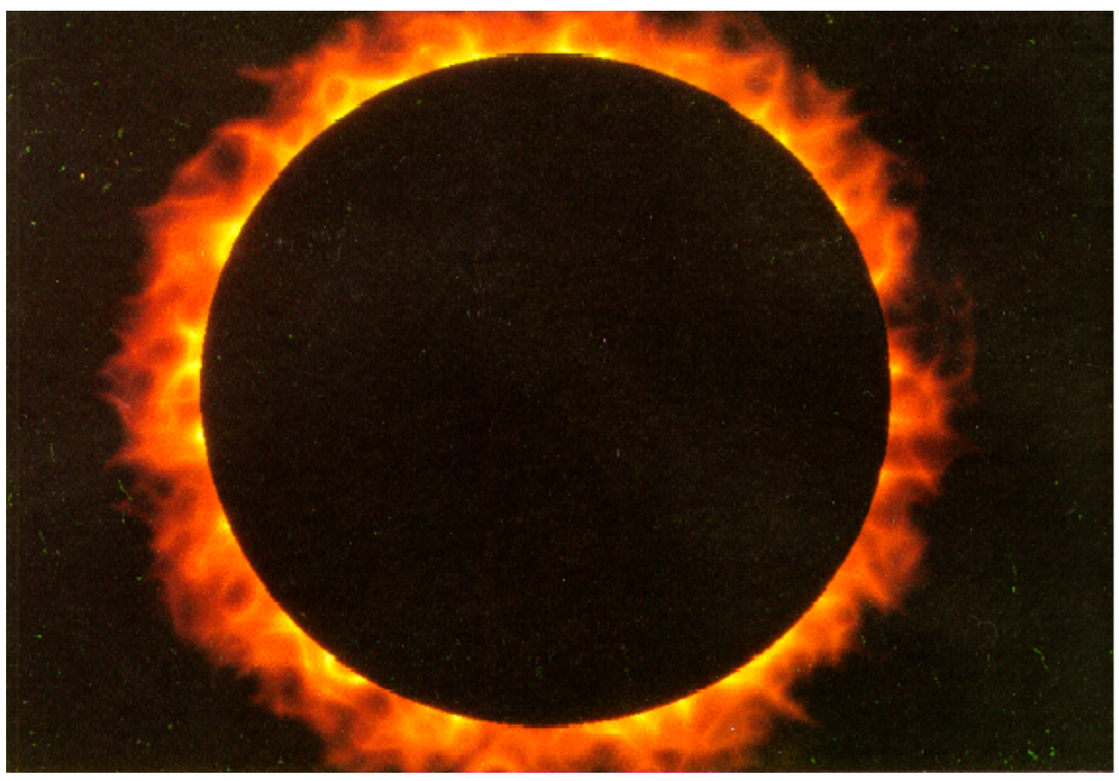
- SIMD operations on pixels
- Conditionals
- Loops
- Function definitions
- Logical operators

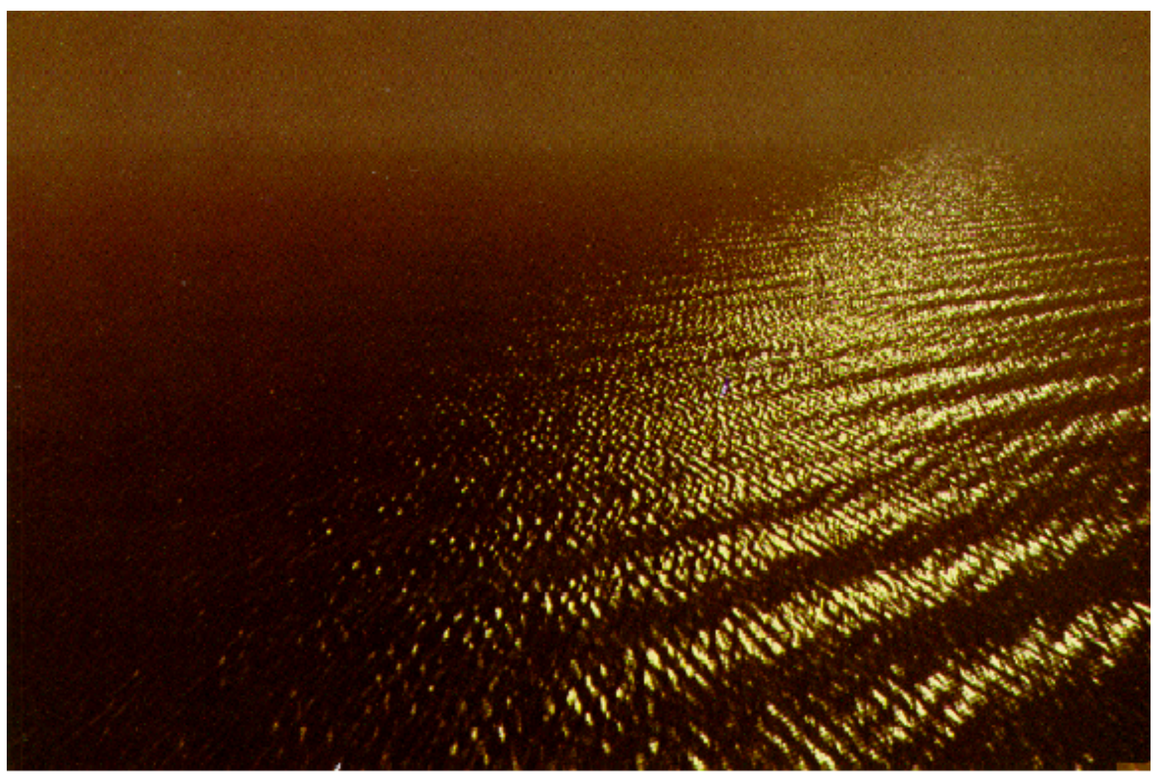
Requires multipass

Also introduces noise

Images from Perlin







Contributions

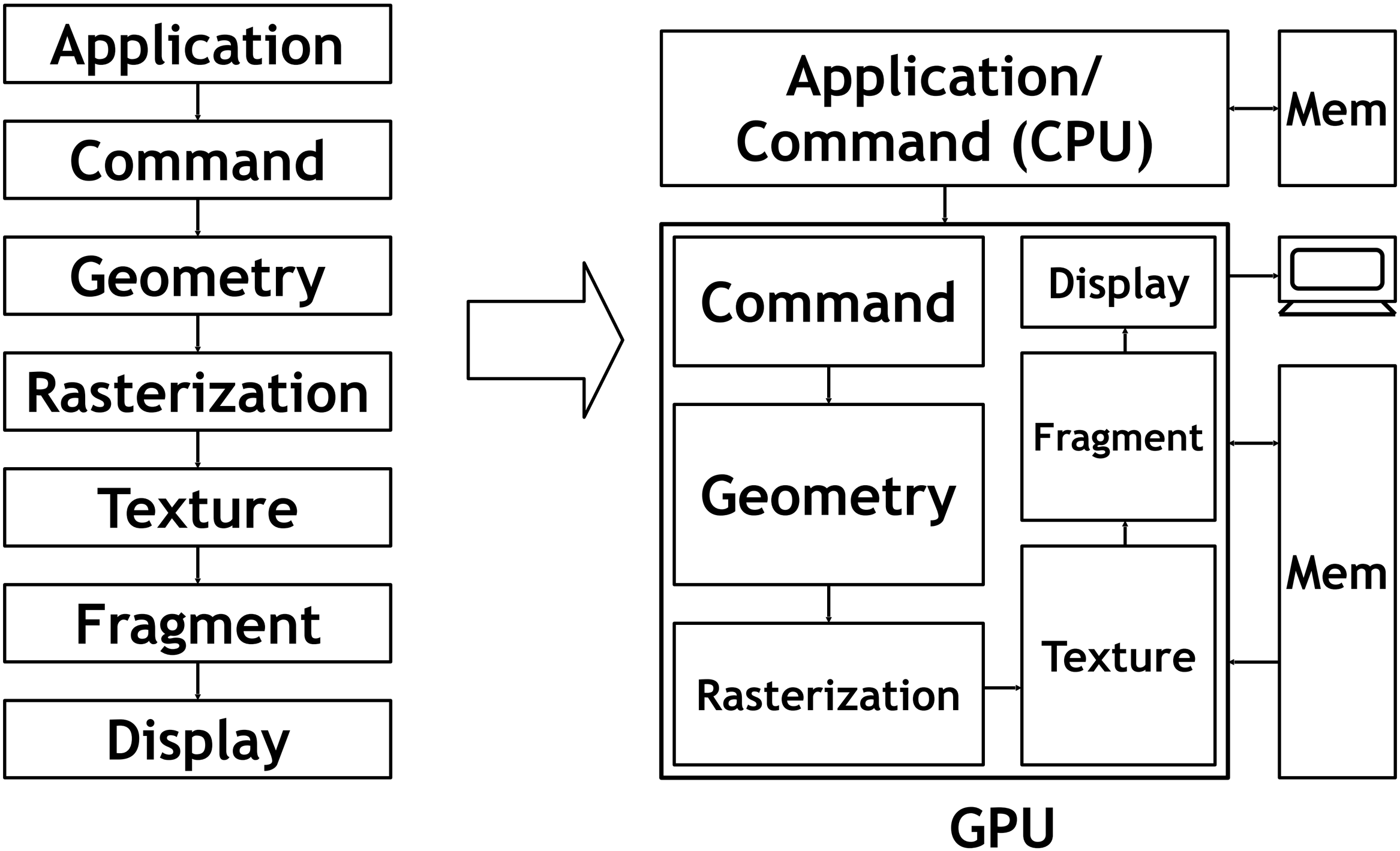
- Cook: Separates / modularizes factors that determine shading
  - Geometric
  - Material
  - Environmental
- Perlin: Language definition
- Leads to . . .

RenderMan—Hanrahan/Lawson

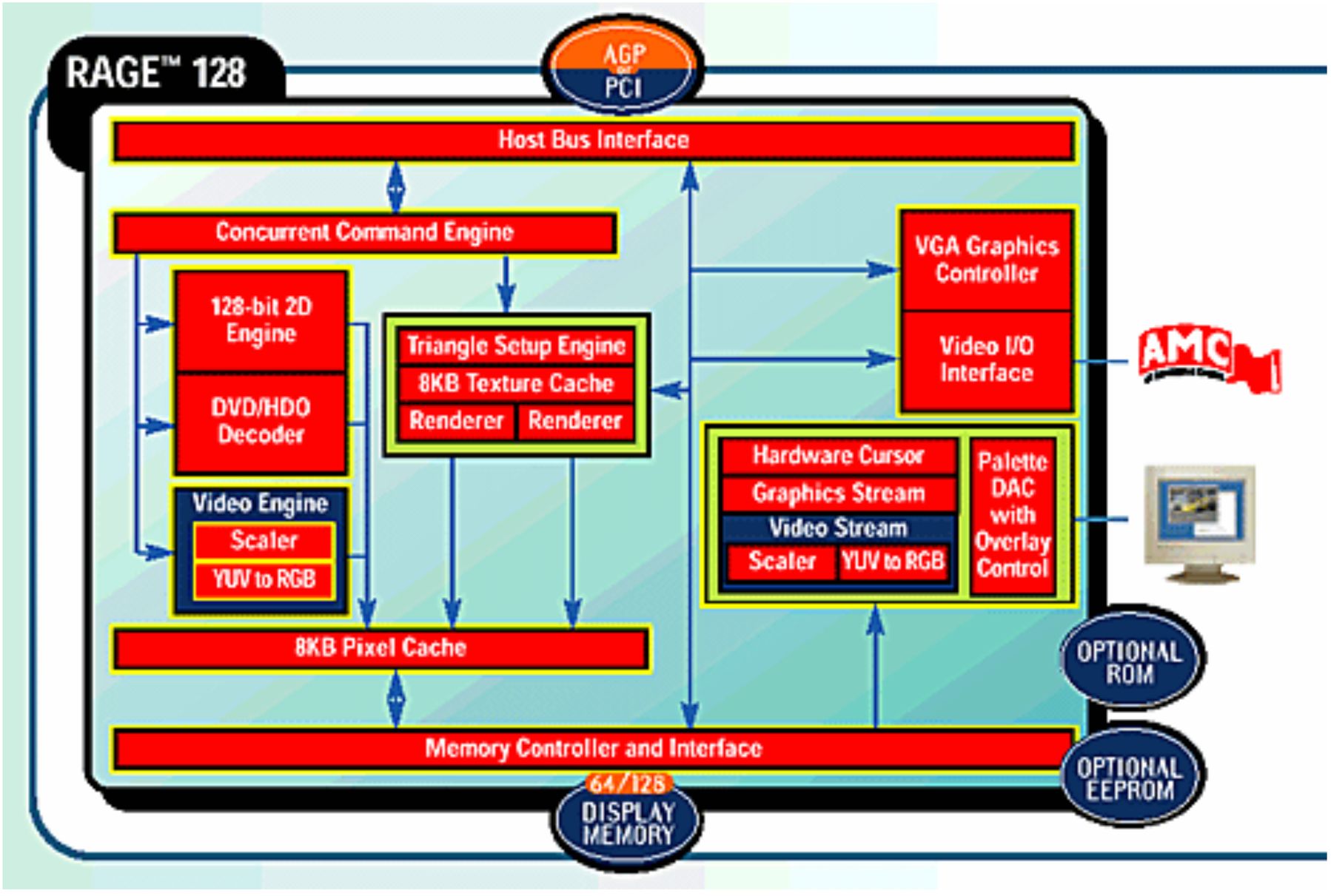
1. Abstract shading model based on optics for either global/ local illumination
2. Define interface between rendering program and shading modules
3. High-level language

Three main kinds of shaders—light source, surface reflectance, volume

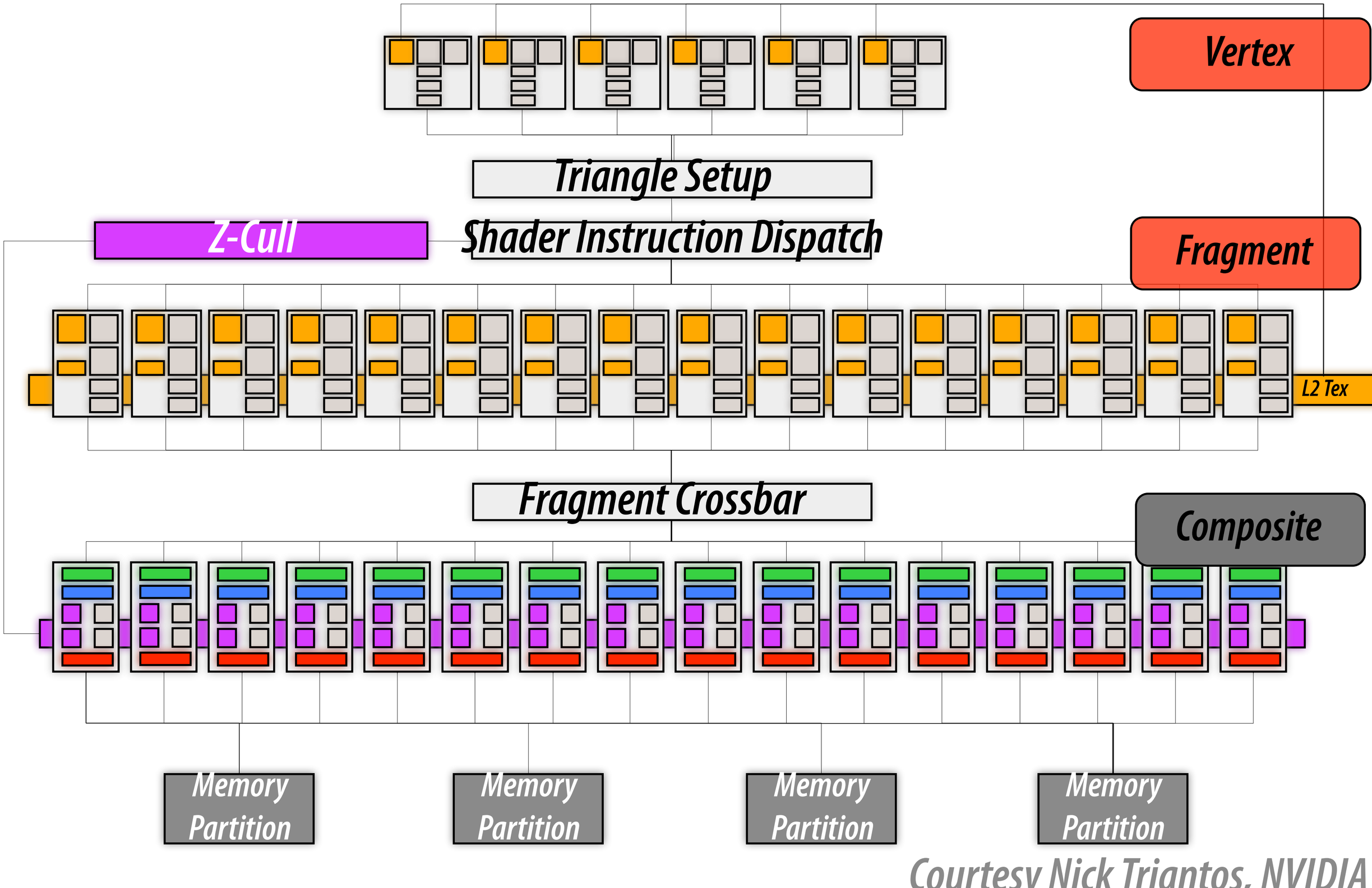
Graphics Hardware—Task Parallel



Rage 128

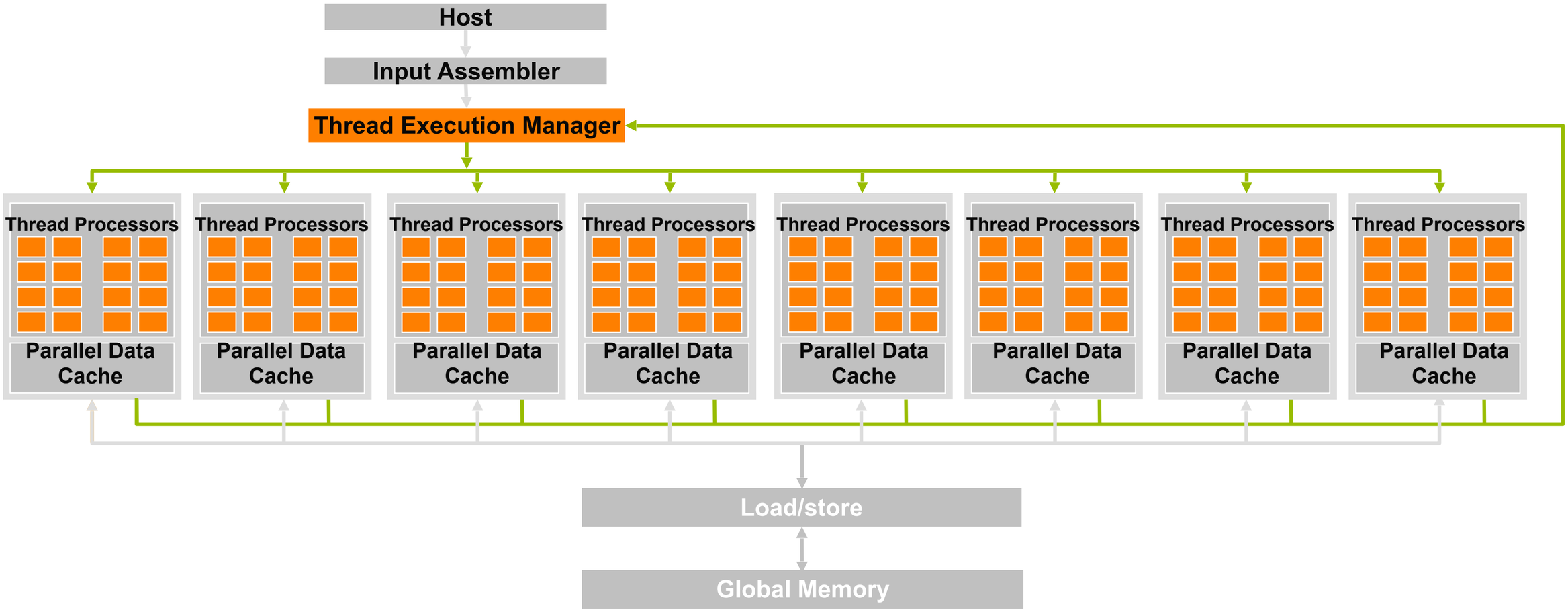

RAGE™ 128
AGP
PCI
Host Bus Interface
Concurrent Command Engine
128-bit 2D Engine
DVD/HDO Decoder
Video Engine
Scaler
YUV to RGB
Triangle Setup Engine
8KB Texture Cache
Renderer
Renderer
8KB Pixel Cache
VGA Graphics Controller
Video I/O Interface
AMC
Hardware Cursor
Graphics Stream
Video Stream
Scaler
YUV to RGB
Palette DAC with Overlay Control
OPTIONAL ROM
OPTIONAL EEPROM
Memory Controller and Interface
64/128
DISPLAY MEMORY

NV GF6800: Programmable Shading



Courtesy Nick Triantos, NVIDIA

GeForce 8800 GPU Computing



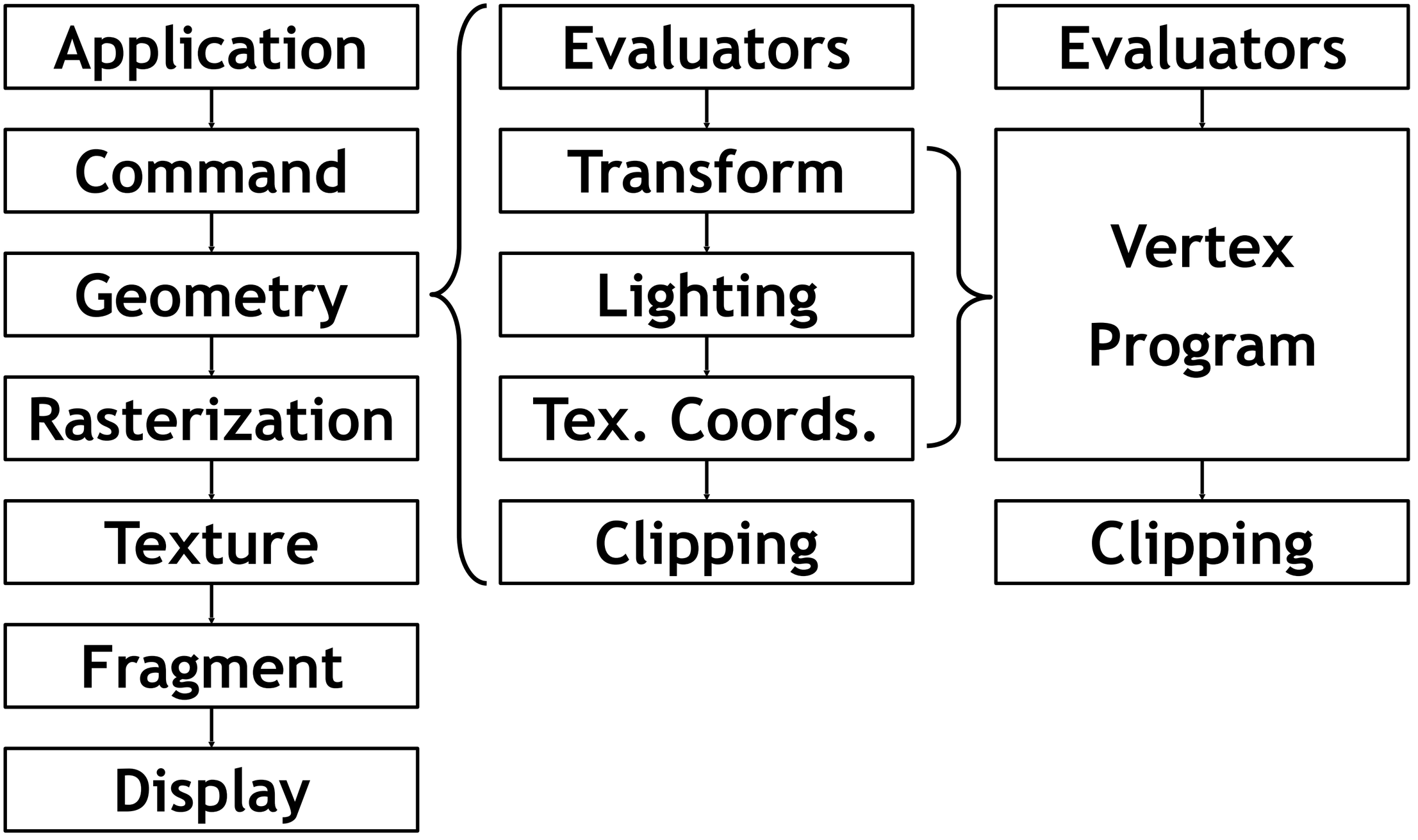
[courtesy of Ian Buck, NVIDIA]

Tension: flexibility vs. performance

- Graphics pipeline provides highly optimized implementations of specific visibility operations
  - Examples: clipping, culling, rasterization, z-buffering
  - Highly optimized implementations on a few canonical data structures (triangles, fragments, and pixels)
  - (We will see how much) the implementation of these functions was deeply intertwined with overall pipeline scheduling/parallelization decisions

- Impractical for rendering system to constrain application to use a single parametric model for surface definitions, lighting, and shading
  - Must allow applications to define these behaviors programmatically
  - Shading language is the interface between application-defined surface, lighting, material reflectance functions and the graphics pipeline

Vertex Programs in the Graphics Pipeline



Hardest Thing To Get Used To

- When you write a vertex or fragment program,
  - *You Write One Program.*
  - *It Runs On Every Vertex/Fragment.*
- Programming model is “SPMD” (Single Program, Multiple Data)

Sample glsl Vertex Shader

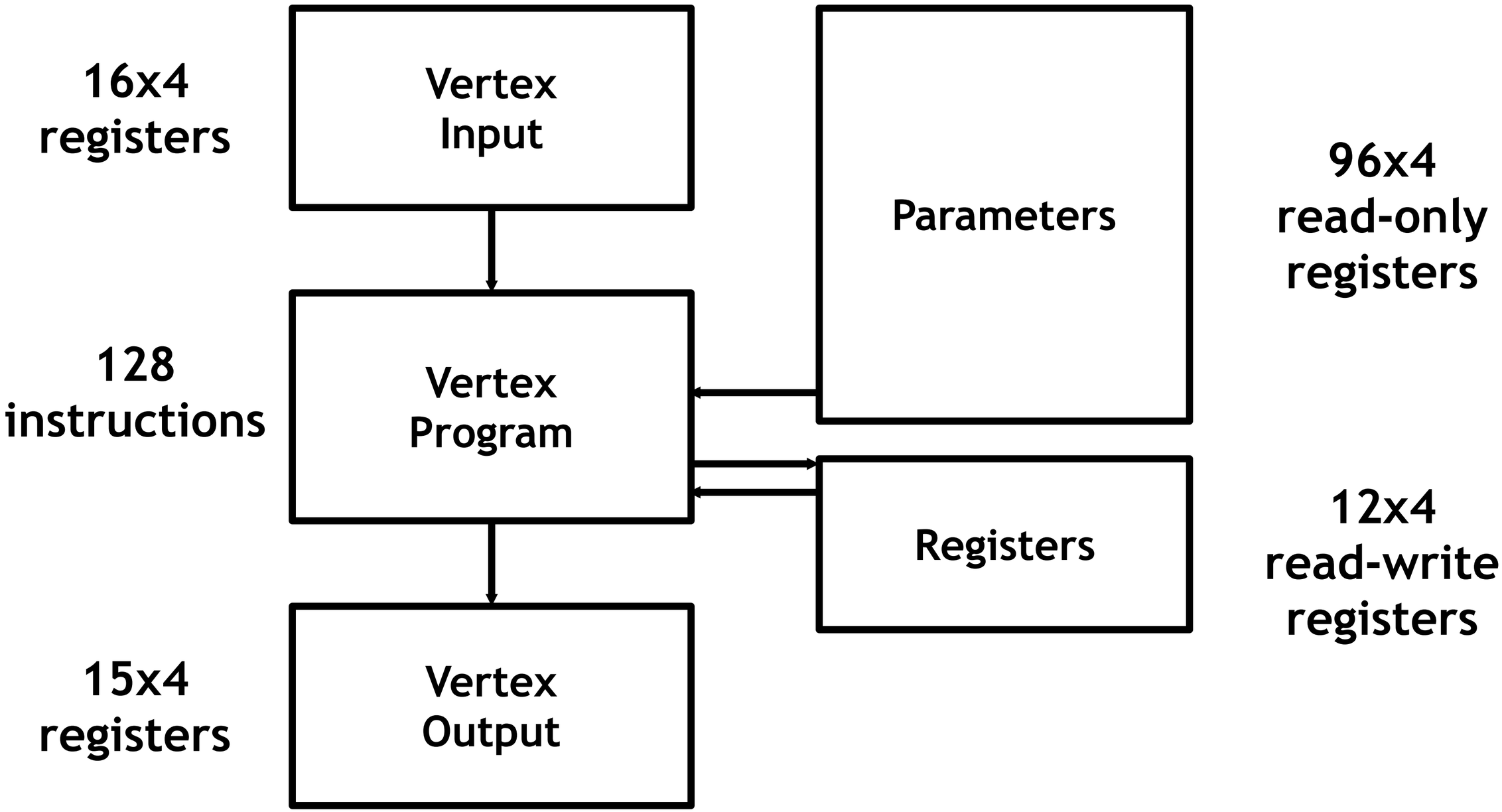
```
layout (std140) uniform Matrices {
    mat4 projModelViewMatrix;
    mat3 normalMatrix;
};

in vec3 position, normal;
in vec2 texCoord;

out VertexData {
    vec2 texCoord;
    vec3 normal;
} VertexOut;

void main() {
    VertexOut.texCoord = texCoord;
    VertexOut.normal = normalize(normalMatrix * normal);
    gl_Position = projModelViewMatrix * vec4(position, 1.0);
}
```

http://www.lighthouse3d.com/tutorials/glsl-core-tutorial/vertex-shader/

Vertex Programs

- Restrict processing to make it easier to parallelize
- Restrictions
  - Avoid dependencies and (weird) ordering constraints
  - Avoid 1 to n expansions or n to 1 reductions
  - Restrictions create independent tasks
- Disallows
  - Evaluation and tessellation
  - Primitive assembly (dependency)
  - Clipping and culling
  - These tricker cases handled in special-purpose ways

Vertex Program Architecture (VS1.0)



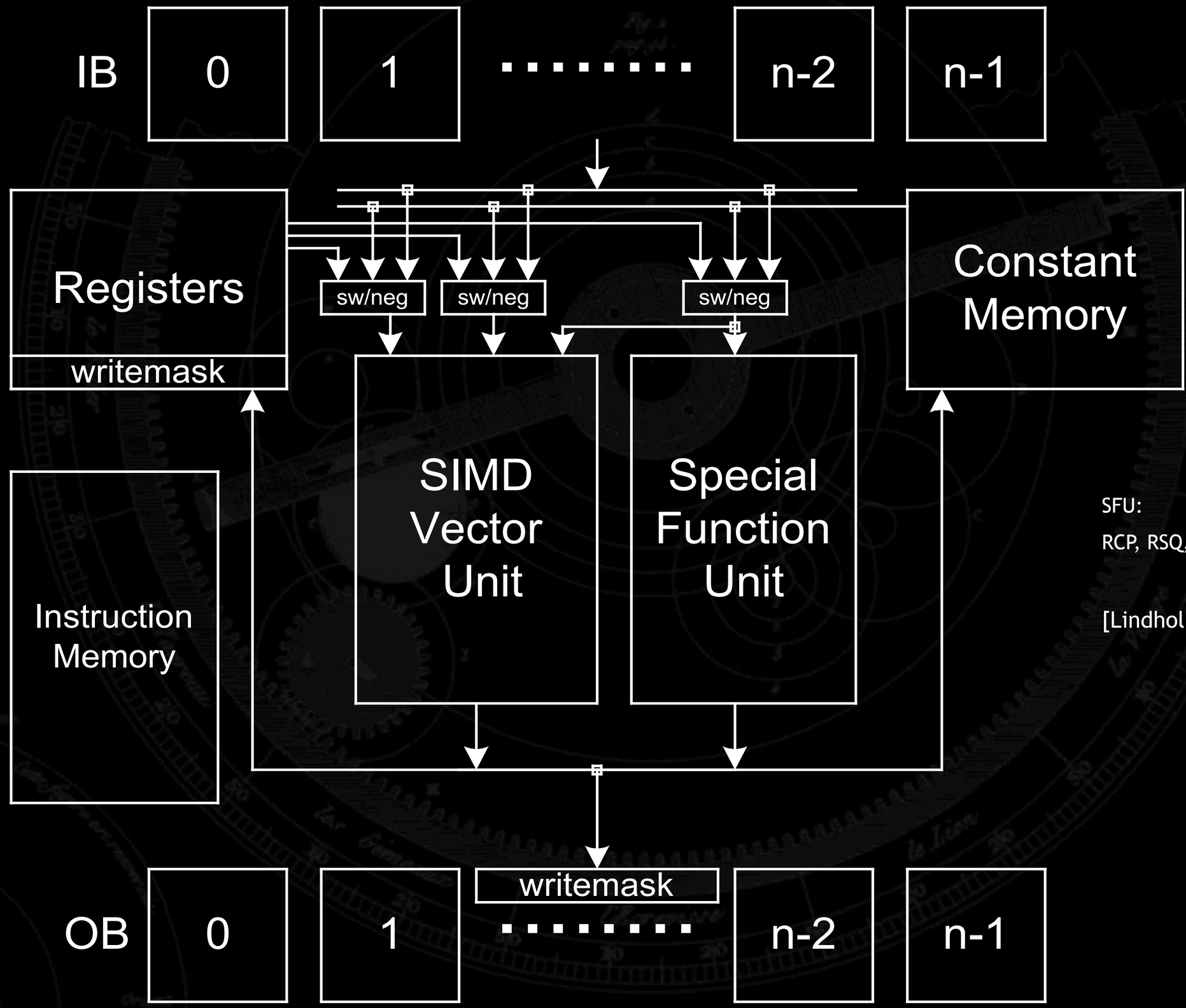
Vertex Input Registers

Attribute Register	Conventional per-vertex Parameter	Conventional Cmd	Conventional Mapping
0	vertex position	glVertex	x,y,z,w
1	vertex weights	glVertexWeightEXT	w,0,0,1
2	normal	glNormal	x,y,z,1
3	Primary color	glColor	r,g,b,a
4	secondary color	glSecondaryColorEXT	r,g,b,1
5	Fog coordinate	glFogCoordEXT	fc,0,0,1
6			
7			
8	Texture coord 0	glMultiTexCoord	s,t,r,q
9	Texture coord 1	glMultiTexCoord	s,t,r,q
10	Texture coord 2	glMultiTexCoord	s,t,r,q
11	Texture coord 3	glMultiTexCoord	s,t,r,q
12	Texture coord 4	glMultiTexCoord	s,t,r,q
13	Texture coord 5	glMultiTexCoord	s,t,r,q
14	Texture coord 6	glMultiTexCoord	s,t,r,q
15	Texture coord 7	glMultiTexCoord	s,t,r,q

Vertex Output Registers

Register Name	*Description*	*Component Interpretation*
o[HPOS]	Homogeneous clip space position	(x,y,z,w)
o[COL0]	Primary color (front-facing)	(r,g,b,a)
o[COL1]	Secondary color (front-facing)	(r,g,b,a)
o[BFC0]	Back-facing primary color	(r,g,b,a)
o[BFC1]	Back-facing secondary color	(r,g,b,a)
o[FOGC]	Fog coordinate	(f,*,*,*)
o[PSIZ]	Point size	(p,*,*,*)
o[TEX0]	Texture coordinate set 0	(s,t,r,q)
o[TEX1]	Texture coordinate set 1	(s,t,r,q)
o[TEX2]	Texture coordinate set 2	(s,t,r,q)
o[TEX3]	Texture coordinate set 3	(s,t,r,q)
o[TEX4]	Texture coordinate set 4	(s,t,r,q)
o[TEX5]	Texture coordinate set 5	(s,t,r,q)
o[TEX6]	Texture coordinate set 6	(s,t,r,q)
o[TEX7]	Texture coordinate set 7	(s,t,r,q)

HW Block Diagram



SFU:

RCP, RSQ, LOG, EXP, LIT

[Lindholm et al]

Instruction Set Summary (VS 1.0)

- 4-way SIMD (like SSE)
- Swizzle/negate on all sources
- Write-mask on all destinations
- MOV/MUL/ADD/MAD: 50%
- DP3/DP4: 40%
- LIT implements Blinn lighting model
- Limited addressing mechanism
- No branches or conditionals

Program Examples

Vector Cross Product

```
# |  i      j      k    | into R2.
# | R0.x   R0.y   R0.z |
# | R1.x   R1.y   R1.z |
MUL R2, R0.zxyw, R1.yzxw;       // swizzle
MAD R2, R0.yzxw, R1.zxyw, -R2; // swizzle, negation
```

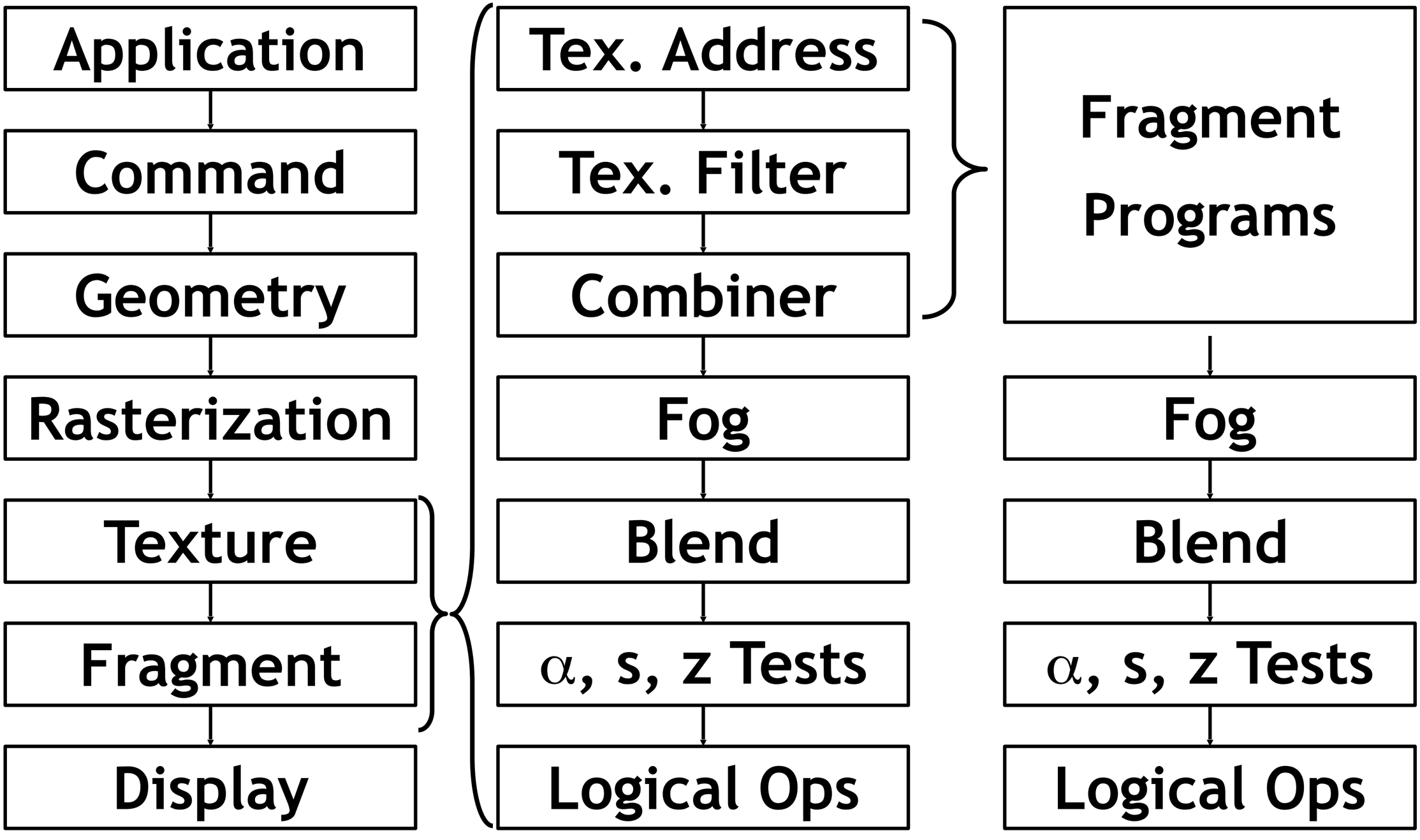
Vector Normalize

```
# R1 = (nx,ny,nz)
#
# R0.xyz = normalize(R1)
# R0.w   = 1/sqrt(nx*nx + ny*ny + nz*nz)
DP3 R0.w, R1, R1;
RSQ R0.w, R0.w;                 // write-mask
MUL R0.xyz, R1, R0.w;           // promotion
```

Simple Graphics Pipeline

```
# c[0-3]  = Mat;          c[4-7]  = Mat^{-T}
# c[32]   = L;            c[33]   = H
# c[35].x = Md * Ld;      c[35].y = Ma * La
# c[36]   = Ms;           c[38].x = s
DP4   o[HPOS].x, c[0], v[OPOS];       # Transform position.
DP4   o[HPOS].y, c[1], v[OPOS];
DP4   o[HPOS].z, c[2], v[OPOS];
DP4   o[HPOS].w, c[3], v[OPOS];
DP3   R0.x, c[4], v[NRML];            # Transform normal.
DP3   R0.y, c[5], v[NRML];
DP3   R0.z, c[6], v[NRML];
DP3   R1.x, c[32], R0;                # R1.x = L DOT N
DP3   R1.y, c[33], R0;                # R1.y = H DOT N
MOV   R1.w, c[38].x;                  # R1.w = s
LIT   R2, R1;                         # Compute lighting
MAD   R3, c[35].x, R2.y, c[35].y;     # diffuse + ambient
MAD   o[COL0].xyz, c[36], R2.z, R3;   # + specular
END
```

Fragment Shaders in the Graphics Pipeline

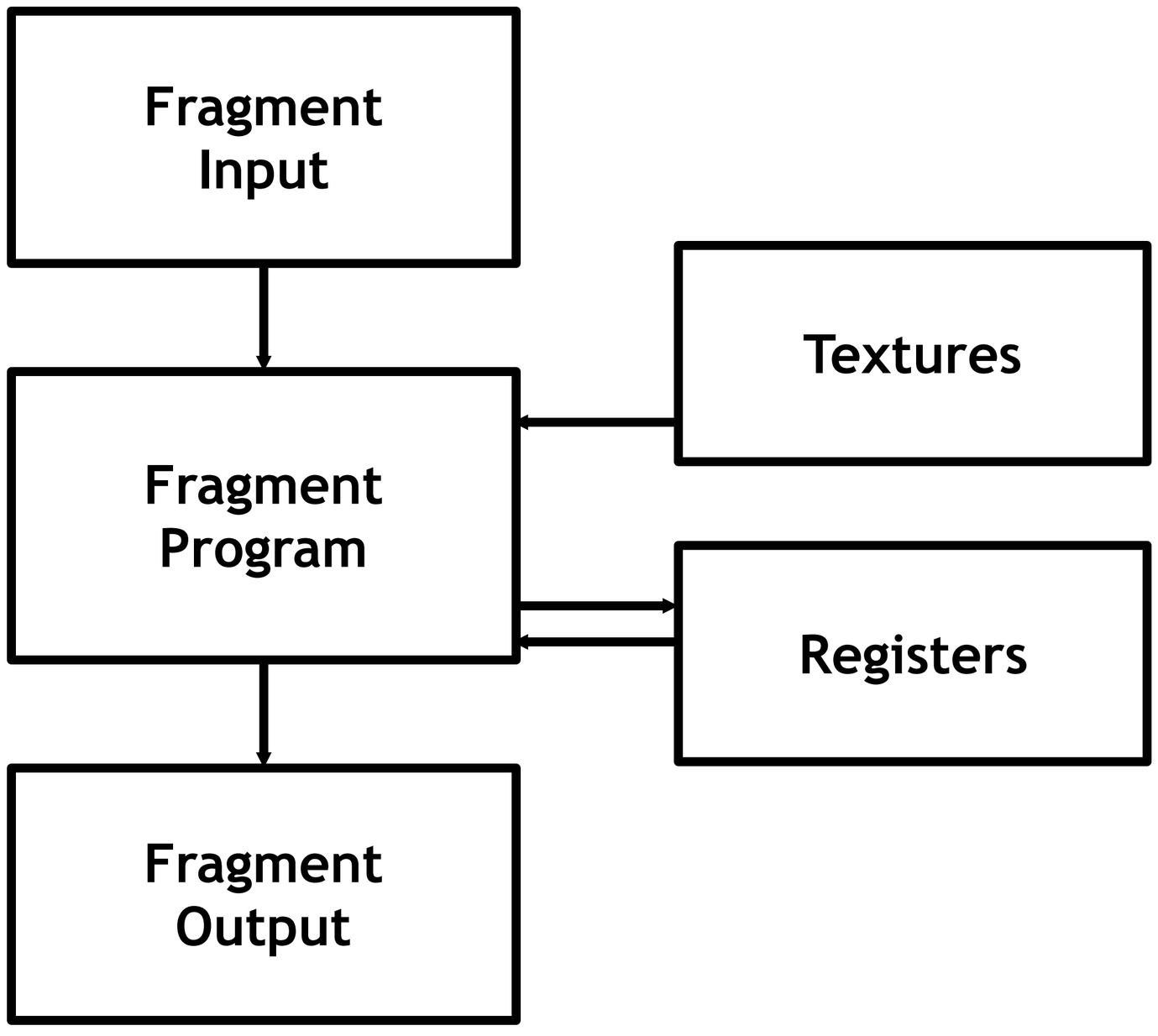


Sample glsl Fragment Shader

```
varying vec3 normal;
varying vec3 vertex_to_light_vector;

void main()
{
    // Defining The Material Colors
    const vec4 AmbientColor = vec4(0.1, 0.0, 0.0, 1.0);
    const vec4 DiffuseColor = vec4(1.0, 0.0, 0.0, 1.0);

    // Scaling The Input Vector To Length 1
    vec3 normalized_normal = normalize(normal);
    vec3 normalized_vertex_to_light_vector = normalize(vertex_to_light_vector);

    // Calculating The Diffuse Term And Clamping It To [0;1]
    float DiffuseTerm = clamp(dot(normal, vertex_to_light_vector), 0.0, 1.0);

    // Calculating The Final Color
    gl_FragColor = AmbientColor + DiffuseColor * DiffuseTerm;
}
```

http://nehe.gamedev.net/article/glsl_an_introduction/25007/

Fragment Program Architecture



Fragment Input Registers

Register Name	*Description*	*Component Interpretation*	*Range/ Precision*
v0	Diffuse color	(r,g,b,a)	0–1
v1	Specular color	(r,g,b,a)	0–1
t0	Texture coordinate set 0	(s,t,r,q)	-1..1
		...	
tn	Texture coordinate set n	(s,t,r,q)	-1..1

Fragment Output Registers

Register Name	*Description*	*Component Interpretation*	*Range/ Precision*
r0	Color	(r,g,b,a)	0..1
r5.r	Depth	(z)	0..1

Shading typically has very high arithmetic intensity

```
sampler mySamp;
Texture2D<float3> myTex;
float3 ks;
float  shinyExp;
float3 lightDir;
float3 viewDir;

float4 phongShader(float3 norm, float2 uv)
{
  float result;
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  float spec = dot(viewDir, 2 * dot(-lightDir, norm) * norm + lightDir);
  result = kd * clamp(dot(lightDir, norm), 0.0, 1.0);
  result += ks * exp(spec, shinyExp);
  return float4(result, 1.0);
}
```

Image credit: http://caig.cs.nctu.edu.tw/course/CG2007

3 scalar float operations + 1 exp()
8 float3 operations + 1 clamp()
1 texture access

Vertex processing often has higher arithmetic intensity than fragment processing (less use of texturing)

Doom 3 Shader (lighting.fp)

- **Phong lighting model:**
  - **Diffuse + specular + ambient**
  - **(Kd*Id*(N.L)) + Ks*Is*(N.H)s)+ (Ka*Ia)**
- **Doom:**
  - **Final lit color = fragment color * light * (diffuse + specular)**
  - **cf * ((fall * proj * (N.L)) * ((Fd*Kd) + (2Ks*Fs*(N.H)s)))**
- **At right: Bump, diffuse, specular, normalization cube, falloff, light projection**

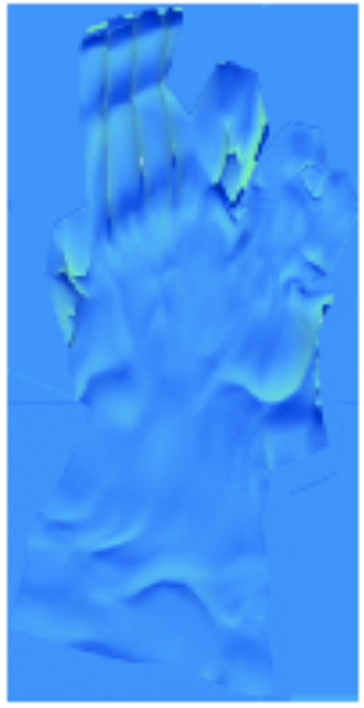


Fig.4 - Bump texture for the hand

Fig.5 - Diffuse texture for the hand

Fig.6 – Specular texture for the hand

Fig.7 – Normalization cube texture

Fig.8 – Light falloff texture

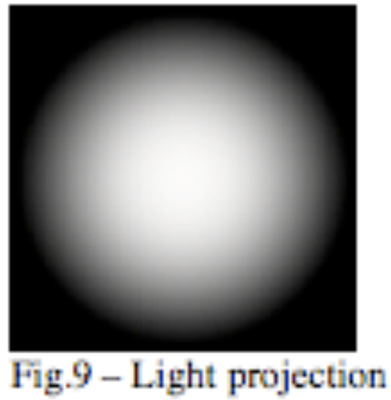


Fig.9 – Light projection texture

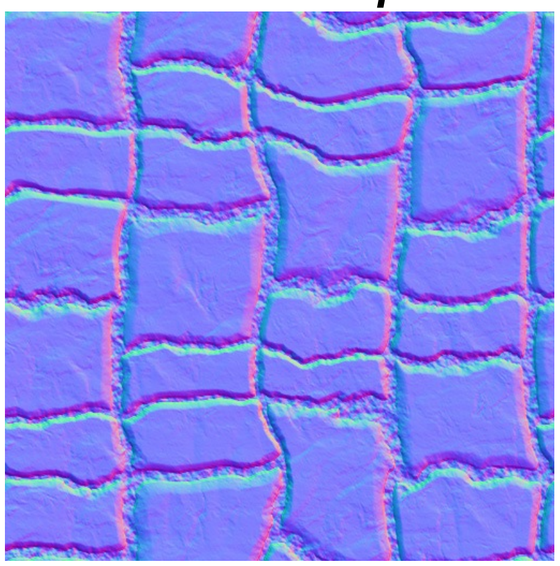
[Pınar Muyan-Özçelik and Ryan Zhou]

Fragment Shader Example

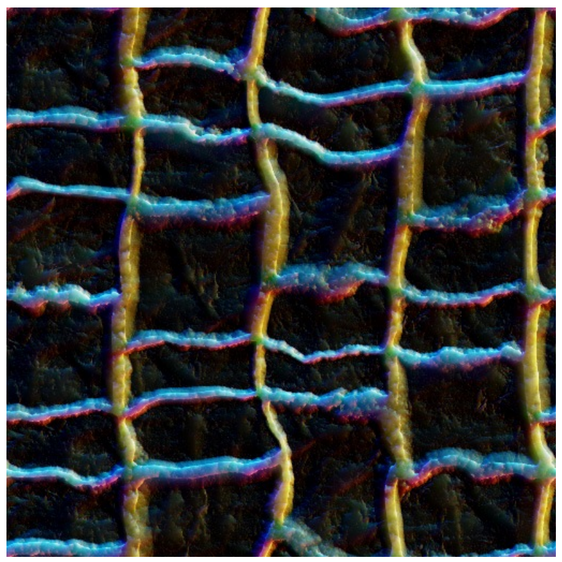
- Per-pixel (tangent space) lighting
- Bump mapping
- Parallax mapping
- Horizon mapping
- Blinn specular (exponent = 60)

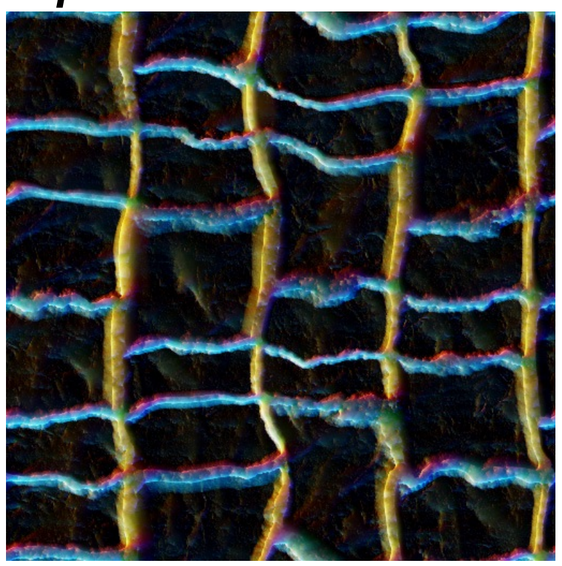
Diffuse Map

Normal Map



Horizon Maps



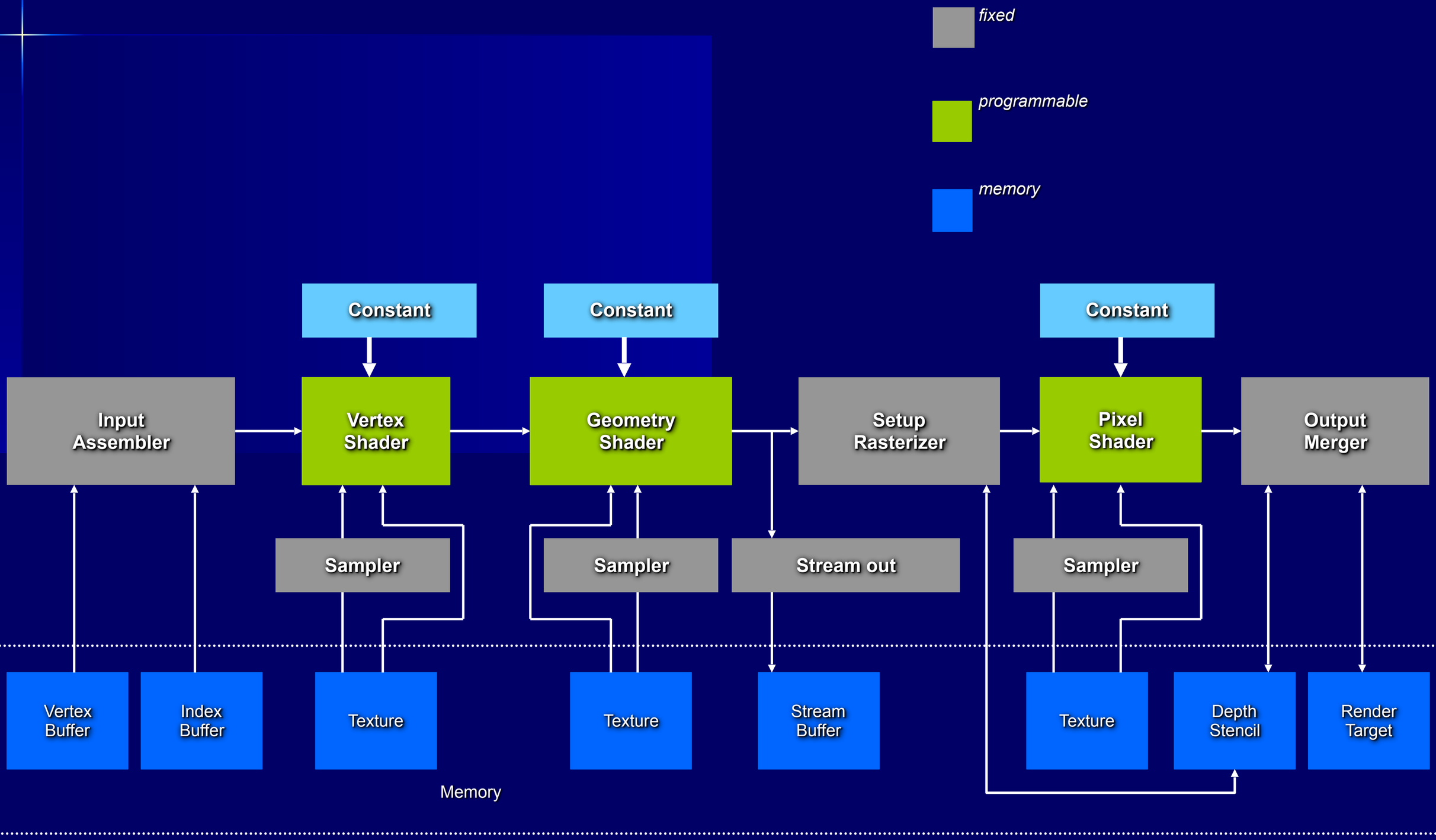


[C4 Engine screenshot, Eric Lengyel]

Fragment Shader Example

```
!!ARBfp1.0
OPTION NV_fragment_program2;

ATTRIB      prim_texcoord = fragment.texcoord;
ATTRIB      camera_dir = fragment.texcoord[1];
ATTRIB      light_dir = fragment.texcoord[2];

PARAM diffuse_params = program.env[13];
PARAM specular_params = program.env[14];
PARAM parallax_scale = program.env[19];

TEMP  temp, colr, vdir, hdir;
TEMP  bump, plax, nrml, diff, spec;

# Normalize direction to camera V
NRMH  vdir.xyz, camera_dir;

# Sample normal map with original texcoords
TEX   bump, prim_texcoord, texture[3], 2D;
MAD   bump, bump, 2.0, -1.0;

# Perform parallax texcoord offset
MUL   plax.xy, bump.w, parallax_scale;
DP3   temp.z, bump, vdir;
MAX   temp.z, temp.z, 0.5;
RCP   temp.z, temp.z;
MUL   plax.xy, plax, temp.z;
MAD   plax.xy, plax, vdir, prim_texcoord;

# Sample normal map at new texcoords
TEX   nrml, plax, texture[3], 2D;
MAD   nrml.xyz, nrml, 2.0, -1.0;

# Calculate max(N·L,0) for diffuse shading
DP3_SAT     colr.rgb, nrml, light_dir;

# Sample primary texture map at new texcoords
TEX   diff, plax, texture[2], 2D;

# Modulate texture color by N*L
MUL   colr.rgb, colr, diff;

# Calculate halfway vector H
ADDH  hdir.xyz, light_dir, vdir;
NRMH  hdir.xyz, hdir;

# Calculate max(N·H,0)^m for specular shading
DP3_SAT     temp.x, nrml, hdir;
POW   spec.rgb, temp.x, specular_params.w;

# Modulate diffuse and specular components by product of material
# color and light color (calculated ahead of time). Then add.
MUL   colr.rgb, colr, diffuse_params;
MAD   colr.rgb, spec, specular_params, colr;

# Apply horizon mapping...
TEMP  hrz1, hrz2, wgt1, wgt2;

# Read directional weights from special cube texture
TEX   temp, light_dir, texture[4], CUBE;
MAD   temp, temp, 2.0, -1.0;
MOV_SAT     wgt1, temp;
MOV_SAT     wgt2, -temp;

# Sample horizon maps
TEX   hrz1, plax, texture[5], 2D;
TEX   hrz2, plax, texture[6], 2D;

# Calculate smooth horizon transition
DP4   temp.x, hrz1, wgt1;
DP4   temp.y, hrz2, wgt2;
MOV   temp.z, light_dir.z;
MOV   temp.w, 1.0;
DP4_SAT     temp.x, temp, {-8.0, -8.0, 8.0, 1.0};

# Modulate final color by horizon shadow
MUL   result.color.rgb, colr, temp.x;
END
```

DirectX 10 Pipeline



Direct3D 11 Pipeline

Input Assembler

Vertex Shader

Hull Shader

Tessellator

Domain Shader

Geometry Shader

Rasterizer

Pixel Shader

Output Merger

Stream Output

Direct3D 10 pipeline

Plus

Three new stages for Tessellation

Plus

Compute Shader

Data Structure

Compute Shader

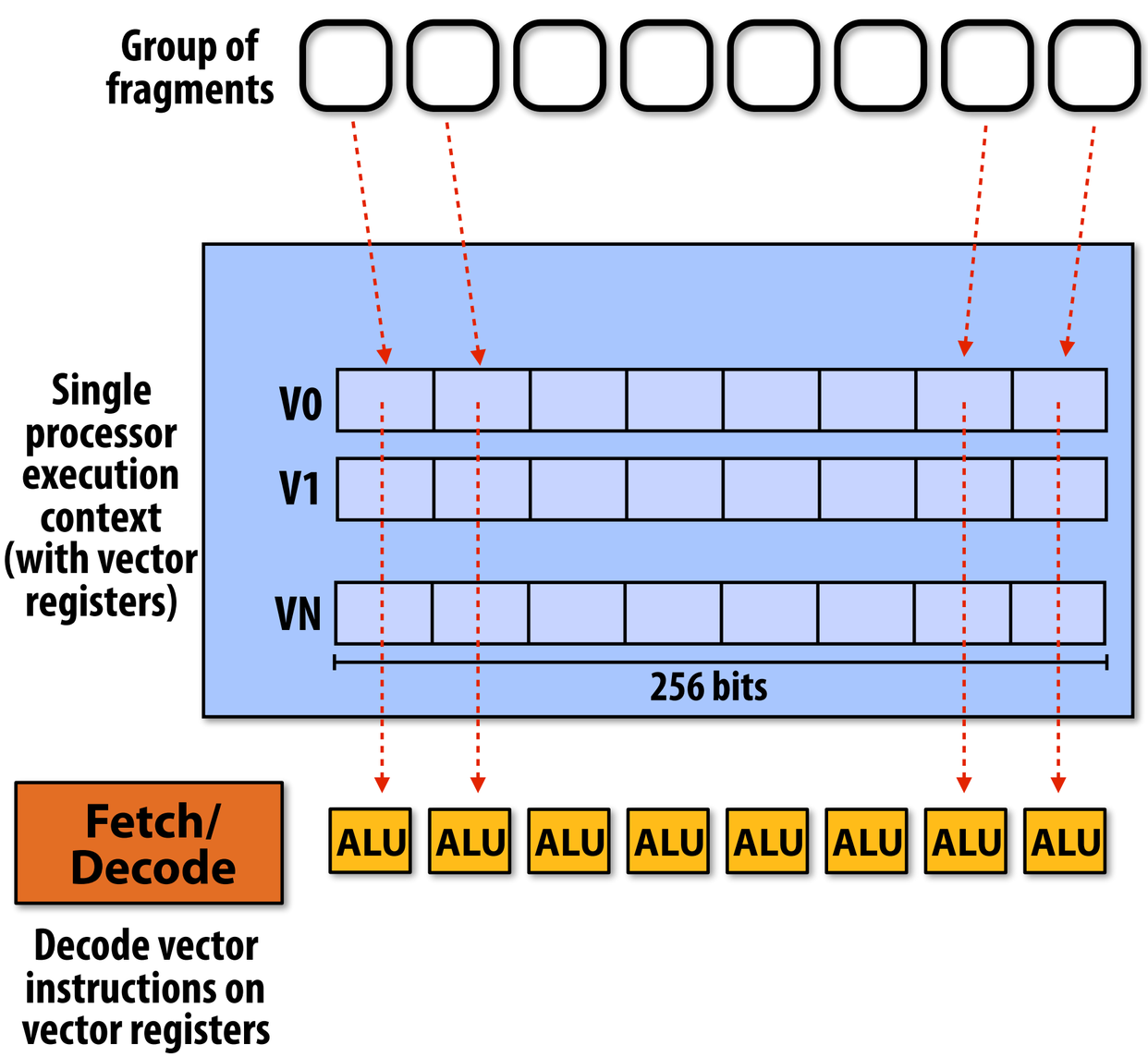
Efficiently mapping shading computations to GPU hardware

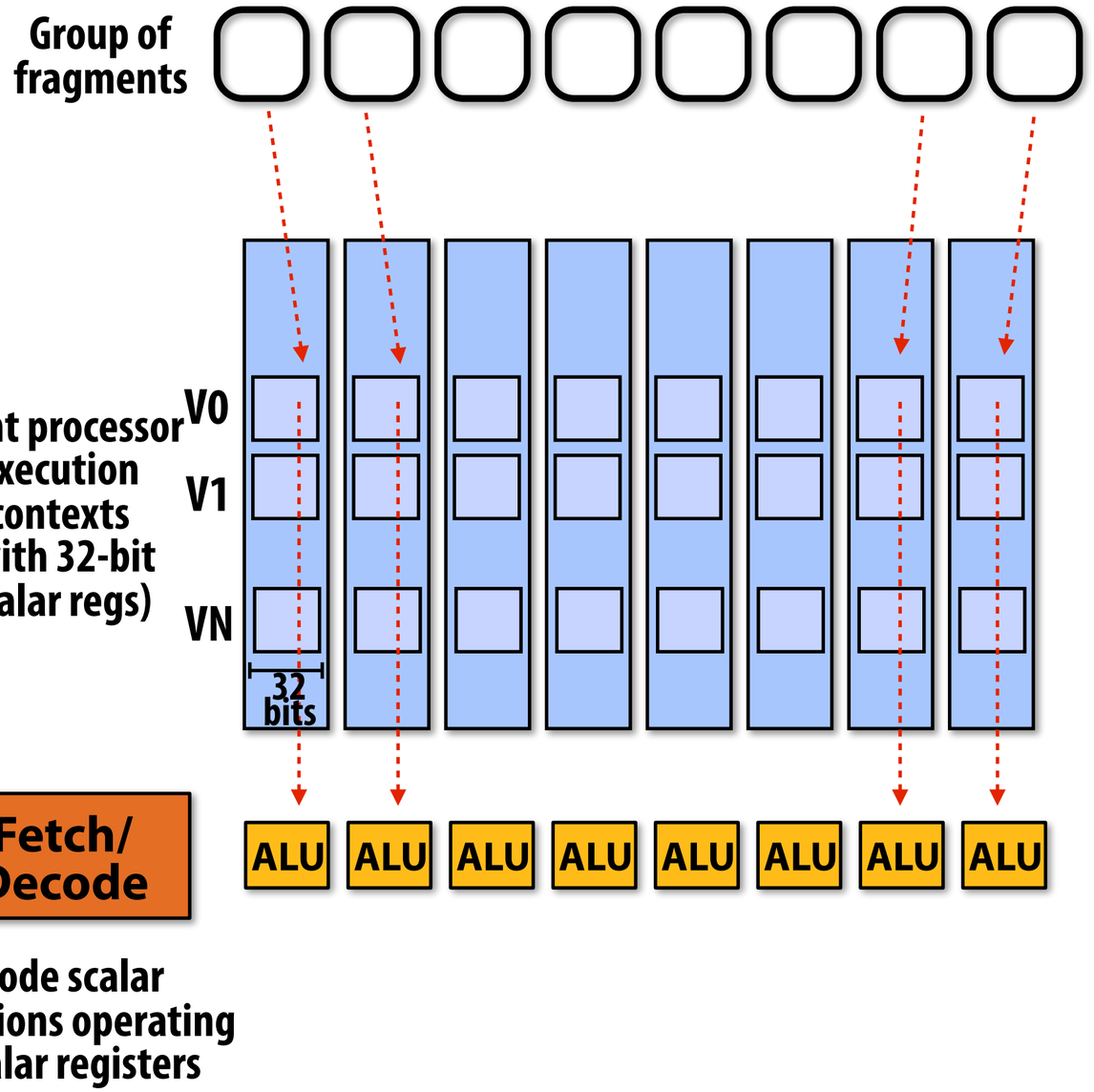
Review: fictitious throughput processor

- Processor decodes one instruction per clock
- Instruction controls all eight SIMD execution units
  - SIMD = "single instruction multiple data"
- "Explicit" SIMD:
  - Vector instructions manipulate contents of 8x32-bit (256 bit) vector registers
  - Execution is all within one hardware execution context
- "Implicit" SIMD (SPMD, "SIMT"):
  - Hardware executes eight unique execution contexts in "lockstep"
  - Program binary contains scalar instructions manipulating 32-bit registers

Mapping fragments to execution units:

Map fragments to "vector lanes" within one execution context (explicit SIMD parallelism) or to unique contexts that share an instruction stream (parallelization by hardware)





GLSL/HLSL shading languages employ a SPMD programming model

- SPMD = single program, multiple data
  - Programming model used in writing GPU shader programs
    - What's the program?
    - What's the data?
  - Also adopted by CUDA, Intel's ISPC

- How do we implement a SPMD program on SIMD hardware?

Example 1: shader with a conditional

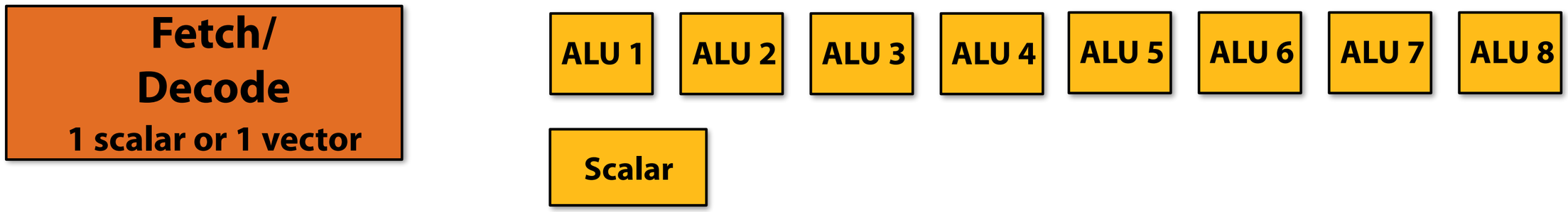
```
sampler mySamp;
Texture2D<float3> myTex;

float4 fragmentShader(float3 norm, float2 st, float4 frontColor, float4 backColor)
{
  float4 tmp;
  if (norm[2] < 0)  // sidedness check (direction of Z component of normal)
  {
     tmp = backColor;
  }
  else
  {
     tmp = frontColor;
     tmp *= myTex.sample(mySamp, st);
  }
  return tmp;
}
```

Example 2: predicate is uniform expression

```
sampler mySamp;
Texture2D<float3> myTex;
float myParam;       // uniform value
float myLoopBound;

float4 fragmentShader(float3 norm, float2 st, float4 frontColor, float4 backColor)
{
   float4 tmp;
   if (myParam < 0.5)
   {
      float scale = myParam * myParam;
      tmp = scale * frontColor;
   }
   else
   {
      tmp = backColor;
   }
   return tmp;
}
```

Notice:
predicate is uniform expression
(same result for all fragments)

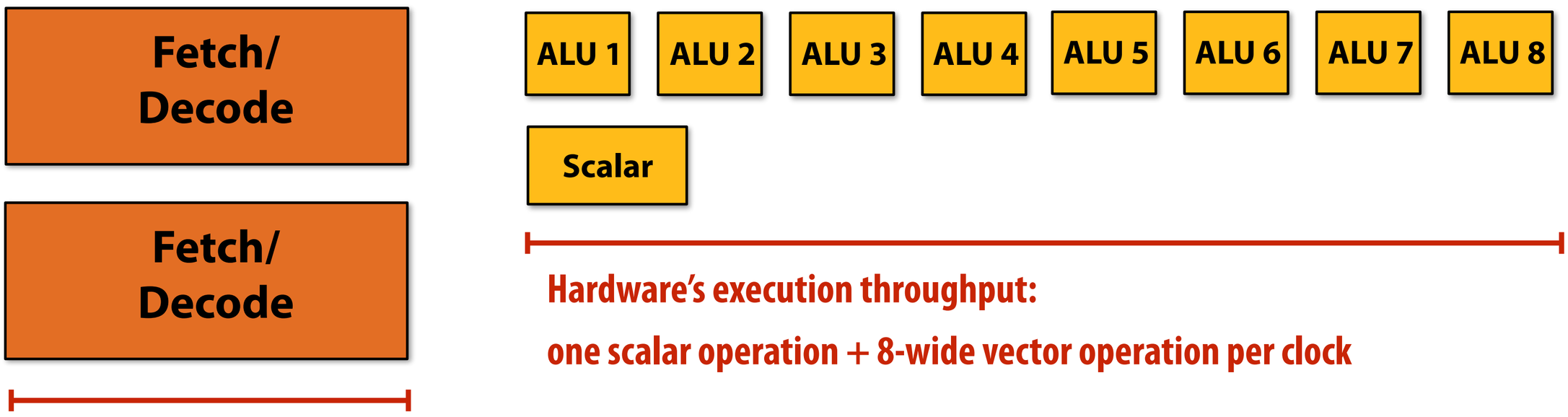
Improved efficiency: processor executes uniform instructions using scalar execution unit



- Decodes one instruction per clock
- Instruction broadcast to all eight execution units
- Instructions manipulate contents of 32-bit (scalar) registers
  - e.g., floating point or integer operations
- Logic shared across all "vector lanes" need only be performed once (not repeated by every vector ALU)
  - Scalar logic identified at compile time (compiler generates different instructions)

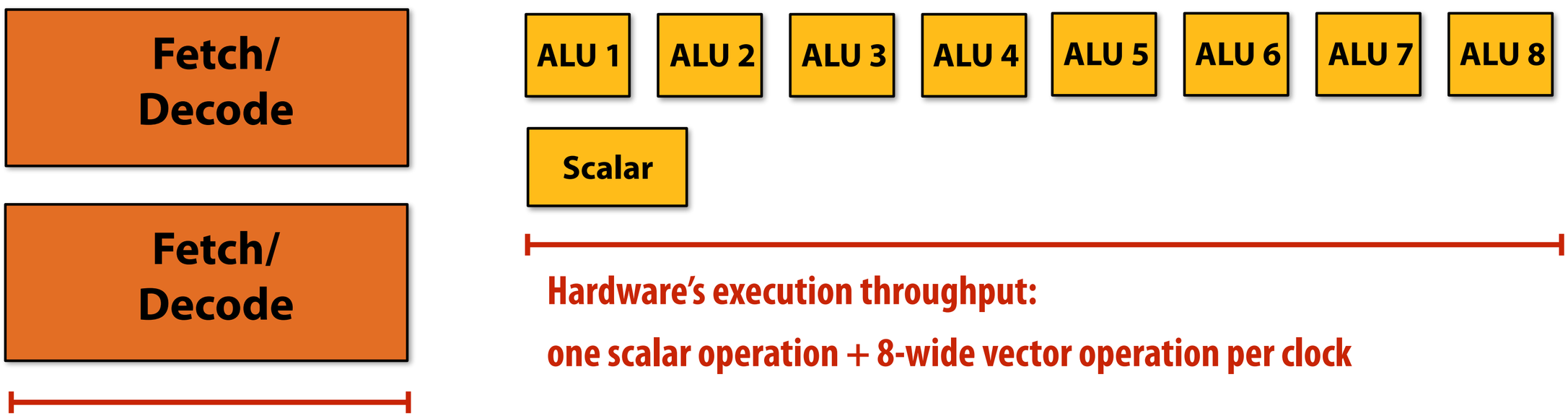
```
float3 lightDir[MAX_NUM_LIGHTS];
int numLights;
float4 multiLightFragShader(float3 norm, float4 surfaceColor)
{
   float4 outputColor;
   for (int i=0; i<num_lights; i++) {
     outputColor += surfaceColor * clamp(0.0, 1.0, dot(norm, lightDir[i]));
   }
}
```

Improving the fictitious throughput processor



- **Now decode two instructions per clock**
  - How should we organize the processor to execute those instructions?

Three possible organizations

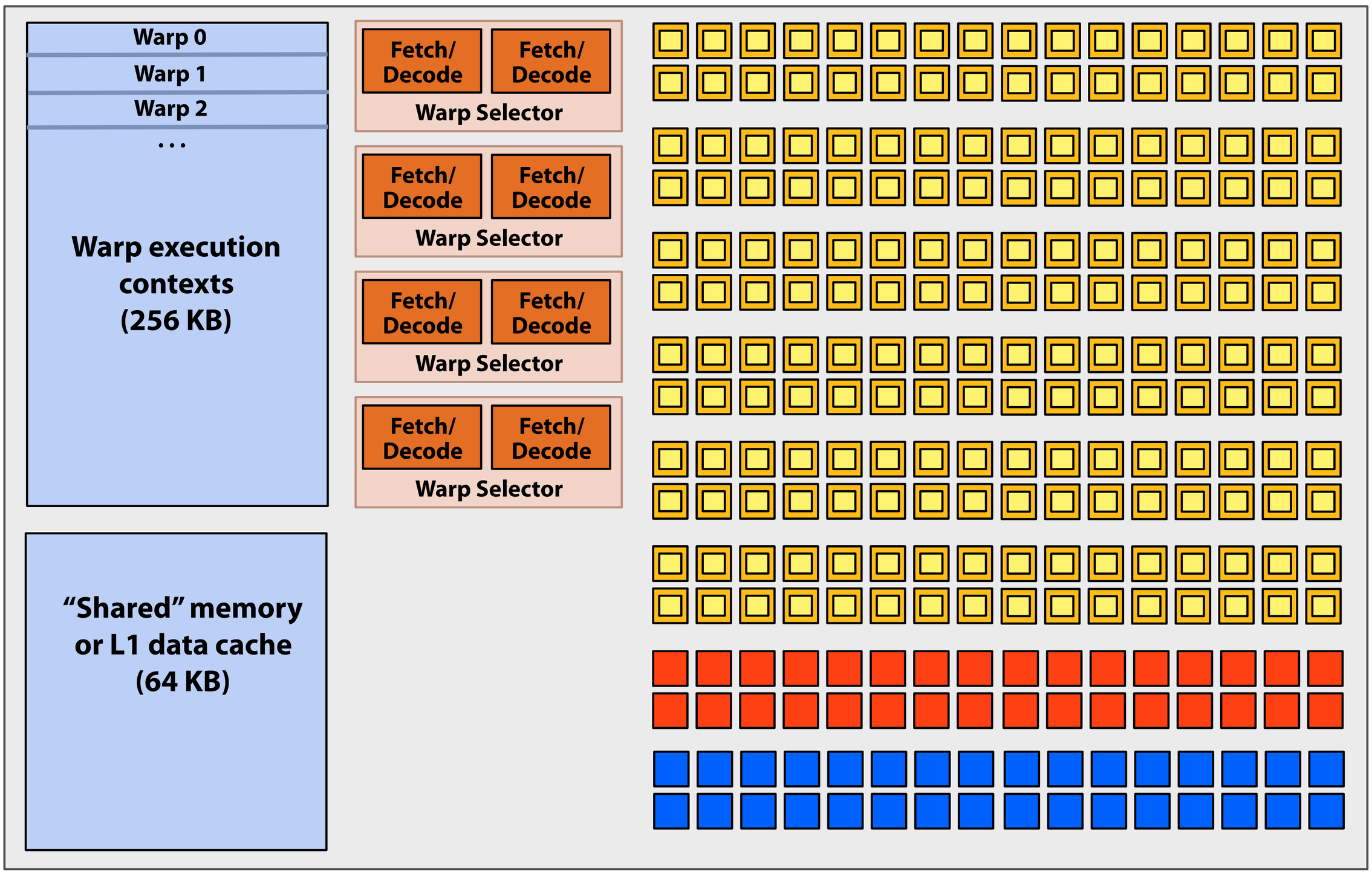


- **Execute two instructions (one scalar, one vector) from same execution context**
  - One execution context can fully utilize the processor's resources, but requires instruction-level-parallelism in instruction stream
- **Execute unique instructions in two different execution contexts**
  - Processor needs two runnable execution contexts (twice as much parallel work must be available)
  - But no ILP in any instruction stream is required to run machine at full throughput
- **Execute two SIMD operations in parallel (e.g., two 4-wide operations)**
  - Significant change: must modify how ALUs are controlled: no longer 8-wide SIMD
  - Instructions could be from same execution context (ILP) or two different ones

NVIDIA GTX 680 (2012)

NVIDIA Kepler GK104 architecture SMX unit (one "core")

Core executes two independent instructions from four warps in a clock (eight total instructions / clock)



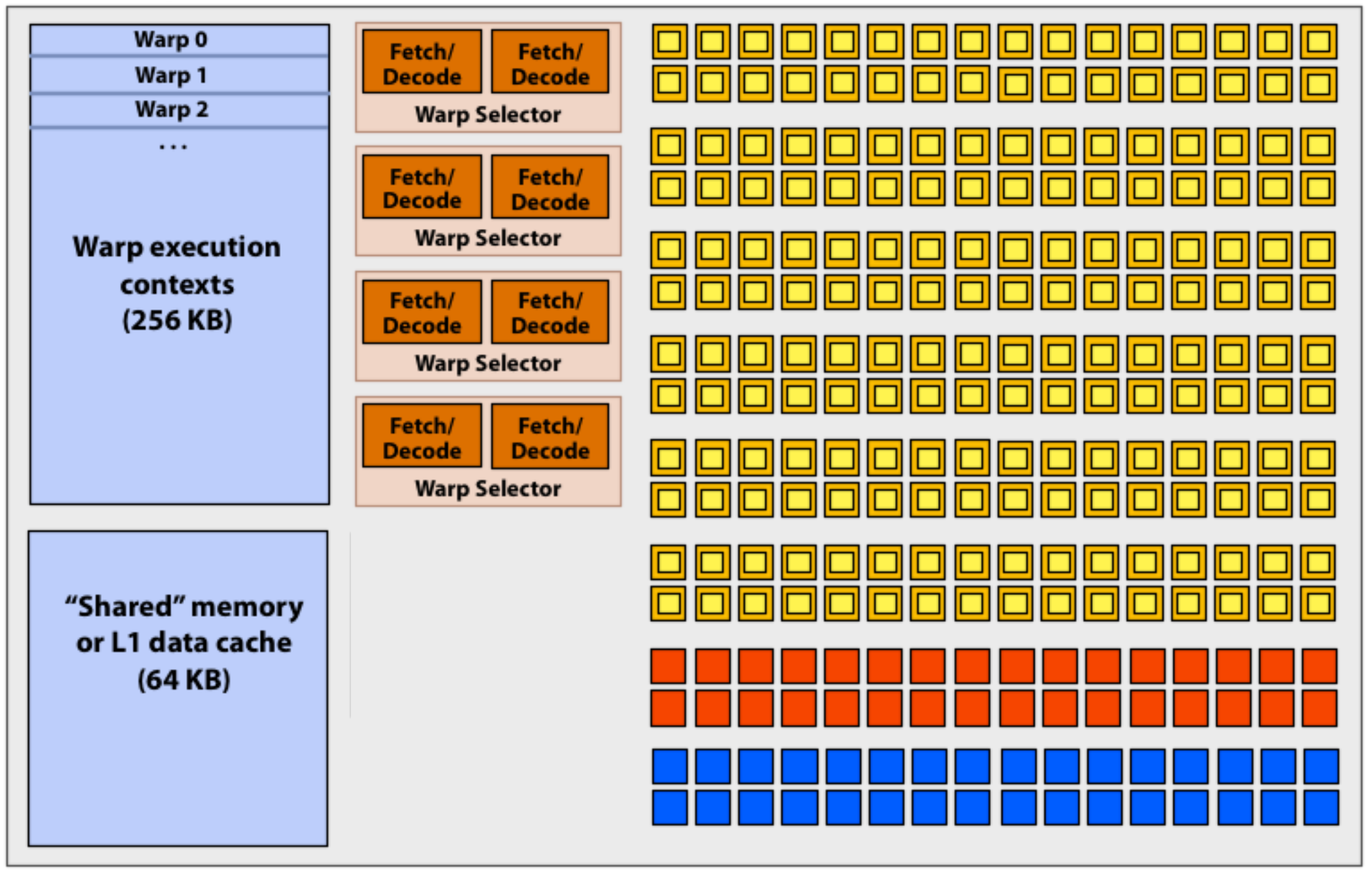
= SIMD function unit, control shared across 32 units (1 MUL-ADD per clock)

= "special" SIMD function unit, control shared across 32 units (operations like sin, cos, exp)

= SIMD load/store unit (handles warp loads/stores, gathers/scatters)

NVIDIA GTX 680 (2012)

NVIDIA Kepler GK104 architecture SMX unit (one "core")



= SIMD function unit,
control shared across 32 units
(1 MUL-ADD per clock)

= "special" SIMD function unit,
control shared across 32 units
(operations like sin, cos, exp)

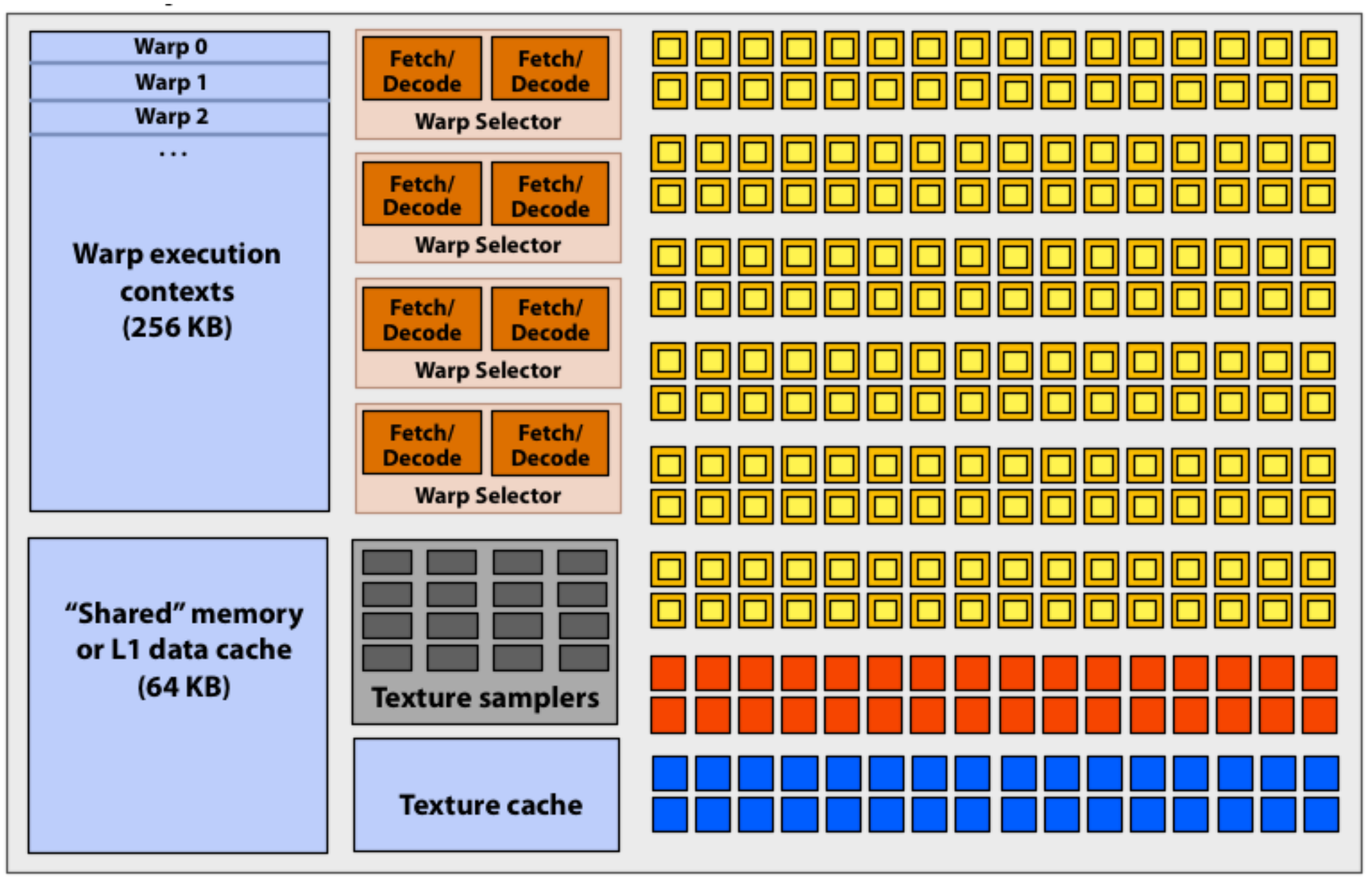
= SIMD load/store unit
(handles warp loads/stores, gathers/scatters)

- SMX core resource limits:
  - Maximum warp execution contexts: 64 (2,048 total CUDA threads)

- Why storage for 64 warp execution contexts if only four can execute at once?
  - Multi-threading to hide memory access latency (in graphics, this is often latency of texture access!)

NVIDIA GTX 680 (2012)

NVIDIA Kepler GK104 architecture SMX unit (one "core")



= SIMD function unit,
control shared across 32 units
(1 MUL-ADD per clock)

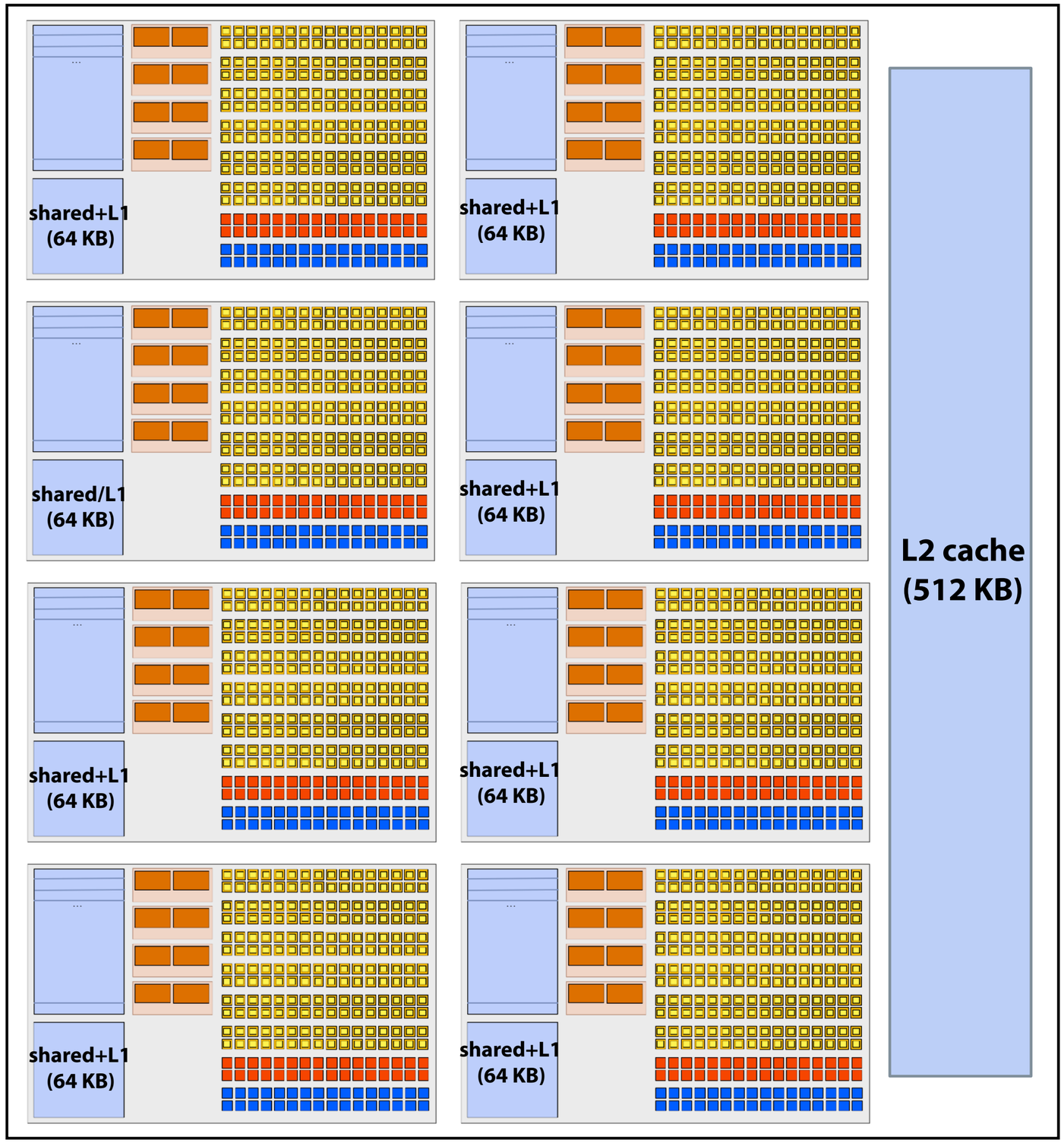
= "special" SIMD function unit,
control shared across 32 units
(operations like sin, cos, exp)

= SIMD load/store unit
(handles warp loads/stores, gathers/scatters)

- SMX programmable core operation each clock:
  - Select up to four runnable warps from up to 64 resident on core (thread-level parallelism)
  - Select up to two runnable instructions per warp (instruction-level parallelism)
  - Execute instructions on available groups of SIMD ALUs, special-function ALUs, or LD/ST units
  - SMX texture unit throughput:
    - 16 filtered texels per clock

NVIDIA GTX 680 (2012)

NVIDIA Kepler GK104 architecture



- 1 GHz clock
- Eight SMX cores per chip
- 8 x 192 = 1,536 SIMD mul-add ALUs = 3 TFLOPs
- Up to 512 interleaved warps per chip (16,384 CUDA threads/chip)
- TDP: 195 watts

192 GB/sec

Memory

256 bit interface

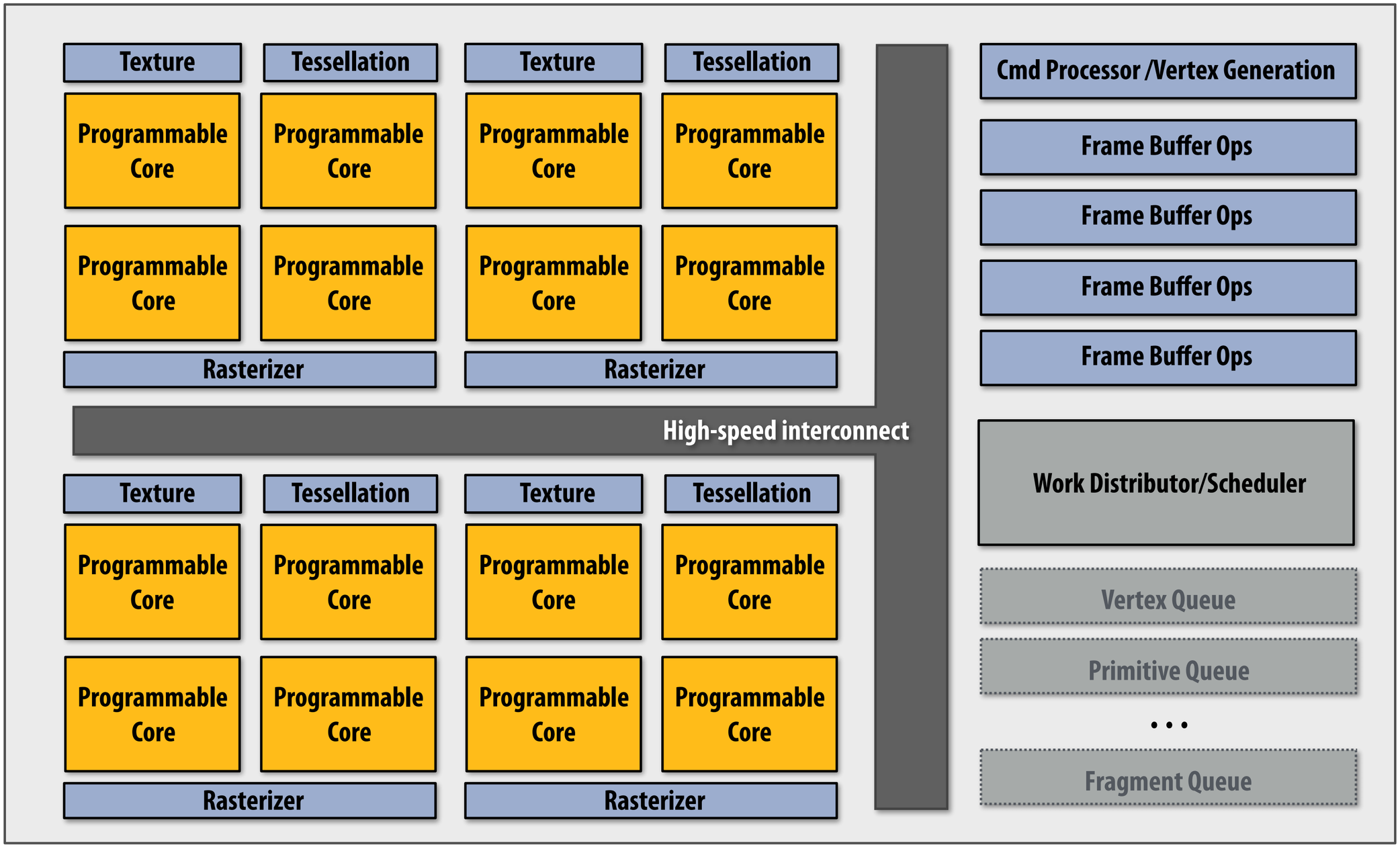
DDR5 DRAM

Shading languages summary

- **Convenient/simple abstraction:**
  - Wide application scope: implement any logic within shader function subject to input/output constraints.
  - Independent per-element SPMD programming model (no loops over elements, no explicit parallelism)
  - Built-in primitives for texture mapping
- **Facilitate high-performance implementation:**
  - SPMD shader programming model exposes parallelism (independent execution per element)
  - Shader programming model exposes texture operations (can be scheduled on specialized HW)
- **GPU implementations:**
  - Wide SIMD execution (shaders feature coherent instruction streams)
  - High degree of multi-threading (multi-threading to avoid stalls despite large texture access latency)
    - e.g., NVIDIA Kepler: 16 times more warps (execution contexts) than can be executed per clock
  - Fixed-function hardware implementation of texture filtering (efficient, performant)
  - High performance implementations of transcendentals (sin, cos, exp) -- common operations in shading

One important thought

Recall: modern GPU is a heterogeneous processor



A unique (odd) aspect of GPU design

- The fixed-function components on a GPU control the operation of the programmable components
  - Fixed-function logic generates work (input assembler, tessellator, rasterizer generate elements)
  - Programmable logic defines how to process generated elements

- Application-programmable logic forms the inner loops of the rendering computation, not the outer loops! ← Think: contrast this design to video decode interfaces on a SoC

- Ongoing debate: can we flip this design around?
  - Maintain efficiency of heterogeneous hardware implementation, but give software control of how pipeline is mapped to hardware resources