Lecture 7:

Shading Languages

(and mapping shader programs to GPU processor cores)

Visual Computing Systems
CMU 15-869, Fall 2013

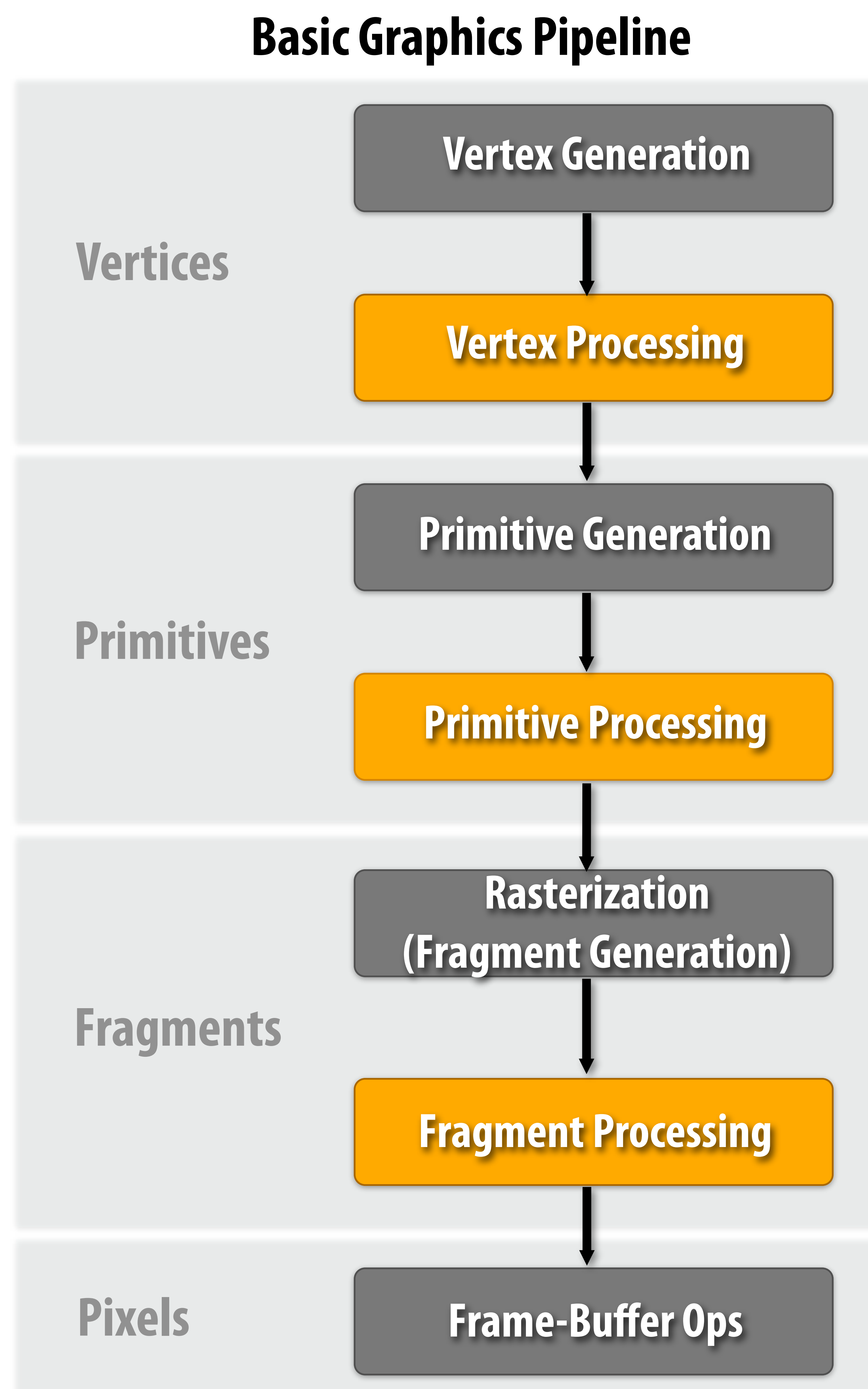
The course so far

So far in this course: focus has been on non-programmable parts of the graphics pipeline

- Geometry processing operations
- Visibility (coverage, occlusion)
- Texturing

I've said very little about about materials, lights, etc.

No mention of programmable GPUs



Review: the rendering equation * [Kajiya 86]

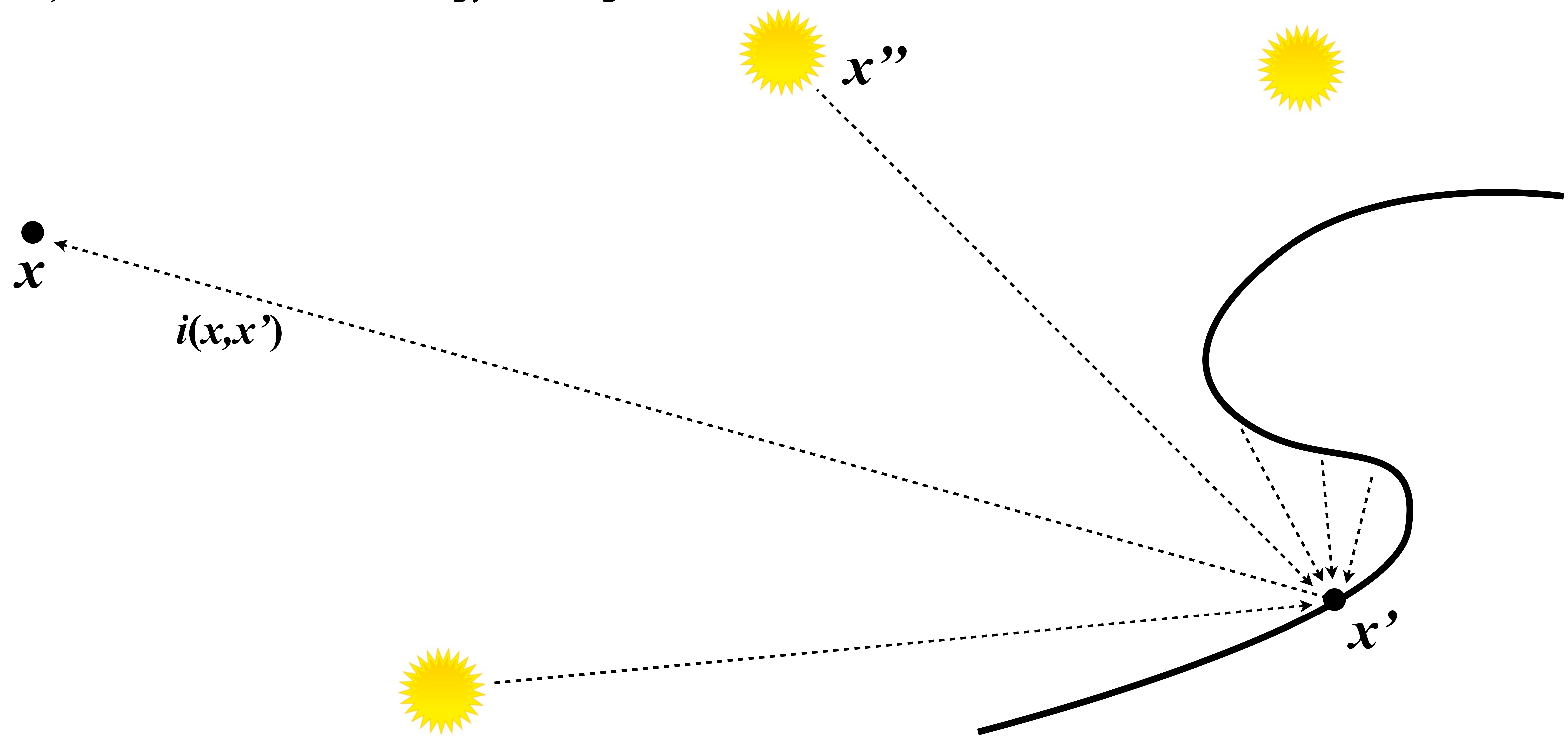
$$i(x,x') = v(x,x')\left[l(x,x') + \int r(x,x',x'')\,i(x',x'')\,dx''\right]$$

$i(x,x')$ = 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 (to match reading)

Example reflection functions

Ideal Specular

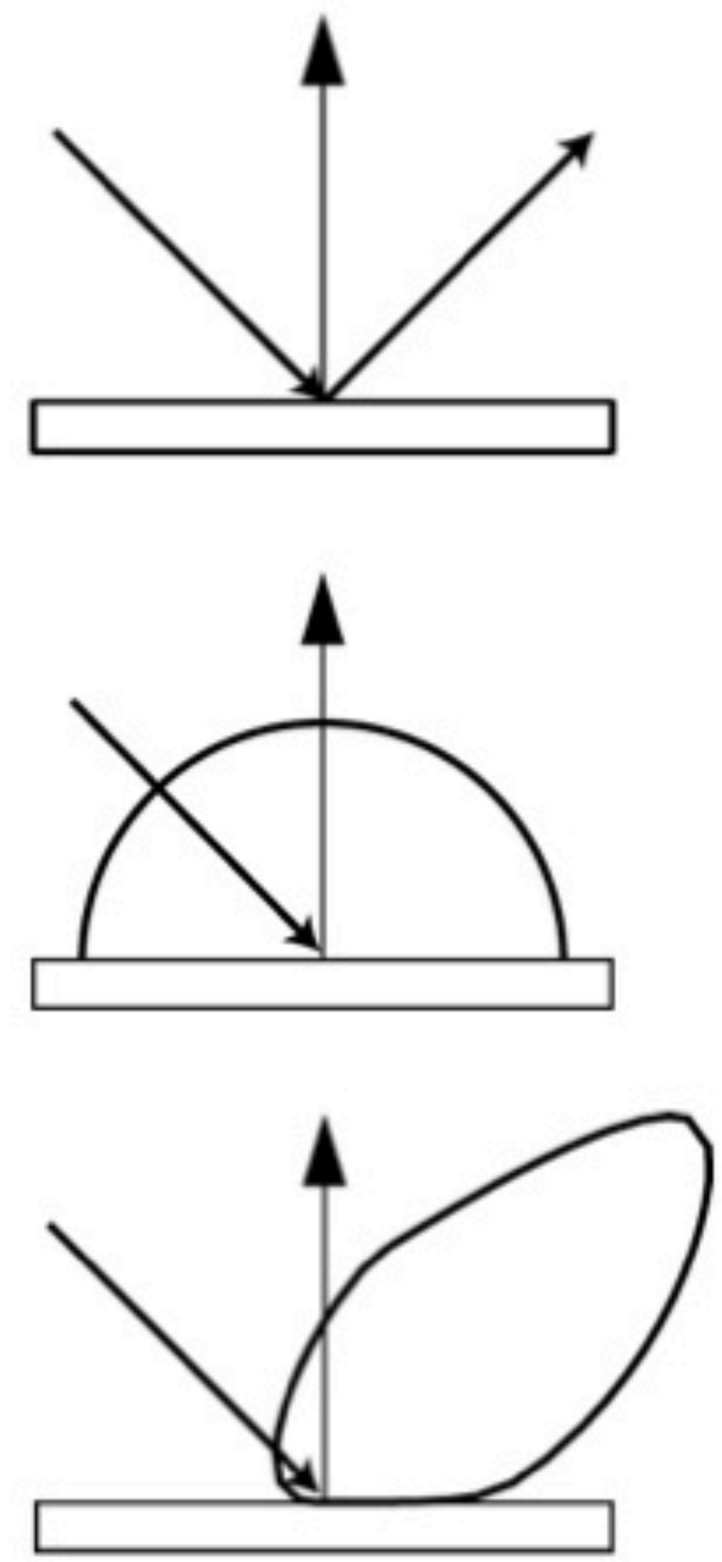
- Reflection Law
- Mirror

Ideal Diffuse

- Lambert's Law
- Matte

Specular

- Glossy
- Directional diffuse



Slide credit Pat Hanrahan

Example materials

Plastic   Metal   Matte

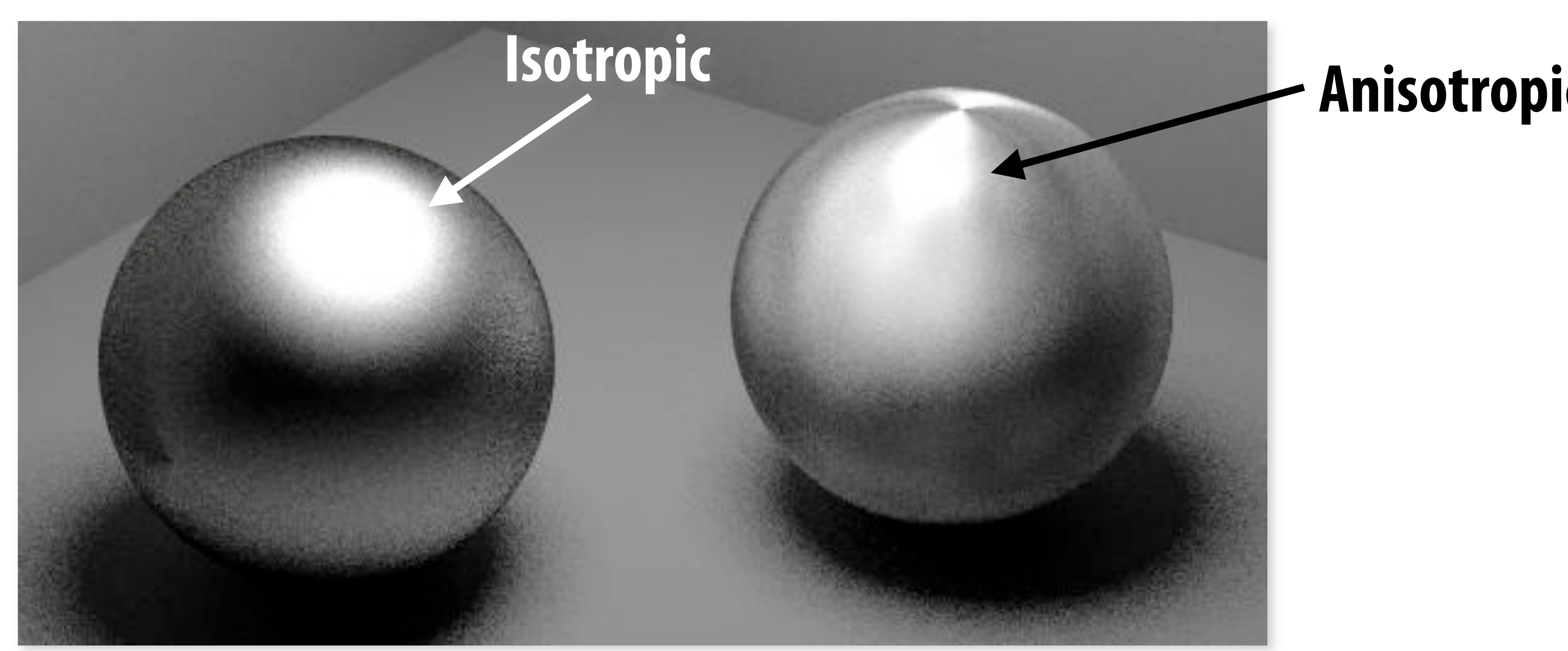
Slide credit Pat Hanrahan
Images from Advanced Renderman [Apodaca and Gritz]

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]



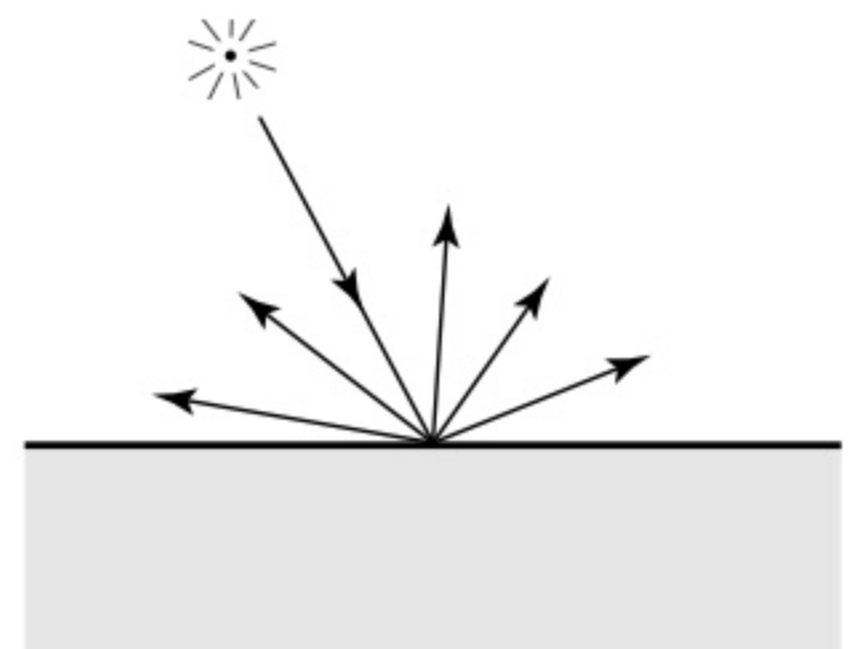
Anisotropic reflection: reflectance depends on azimuthal angle (e.g., oriented microfacets in brushed steel)

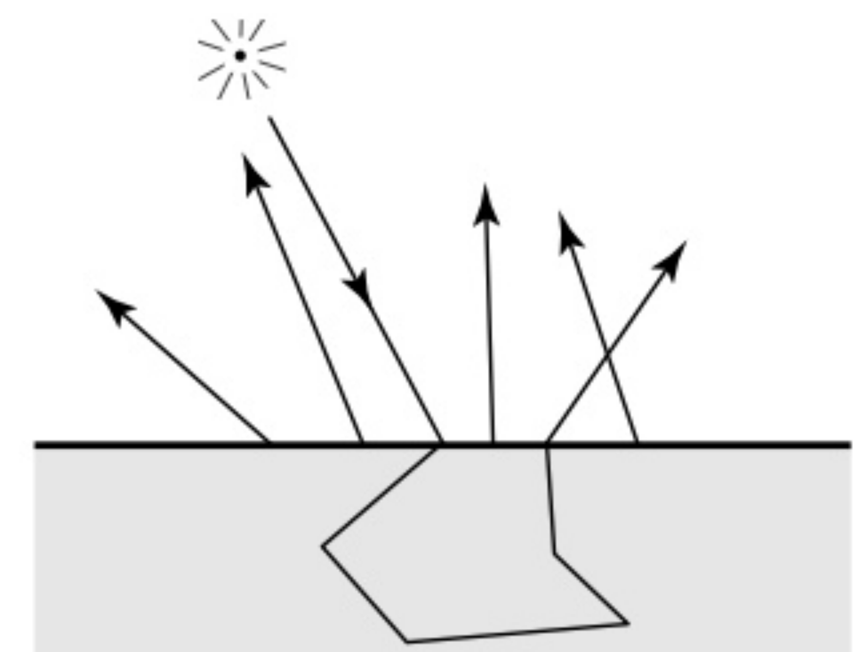
Subsurface scattering

[Wann Jensen et al. 2001]

BRDF

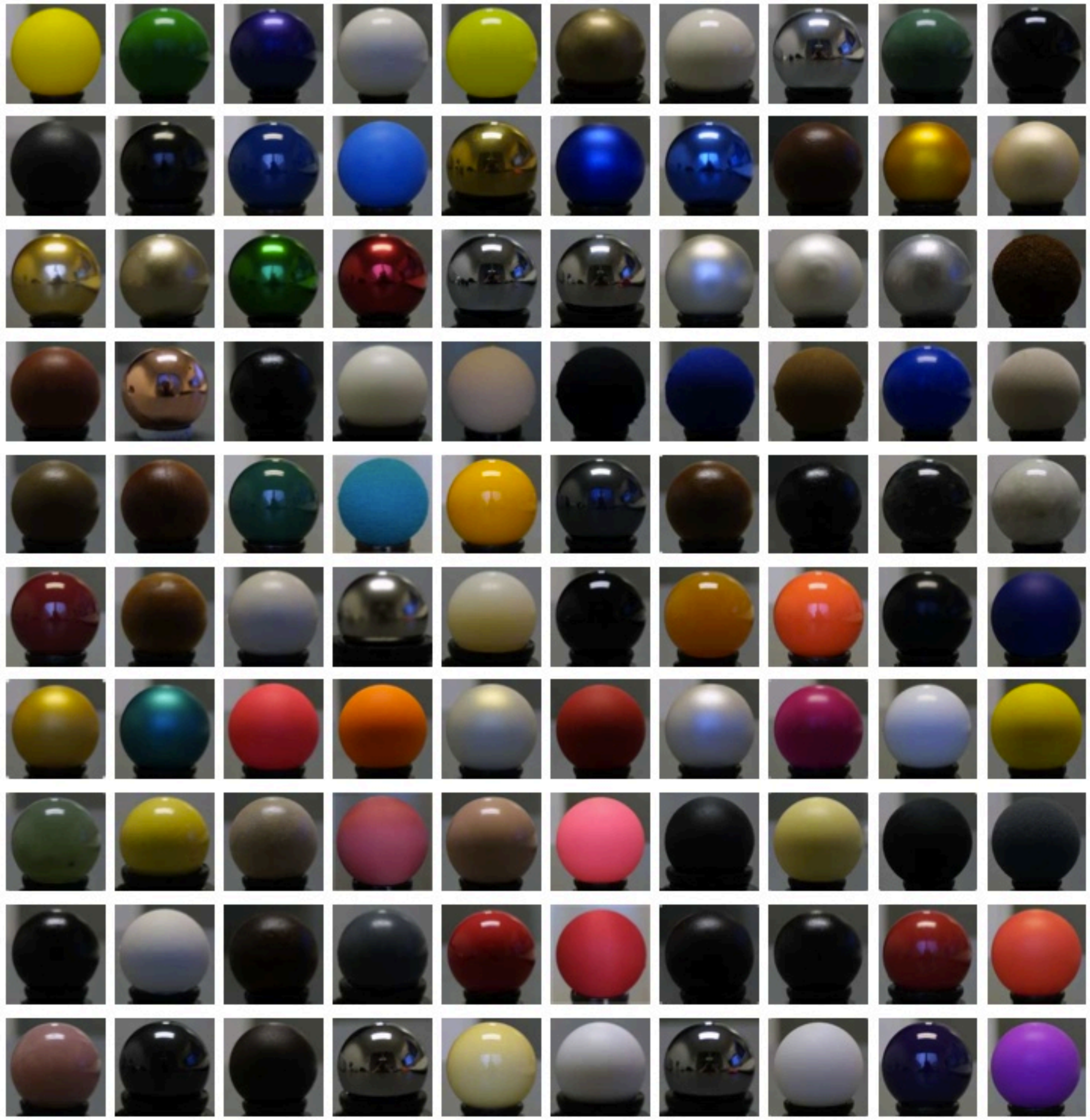
BSSRDF





- 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



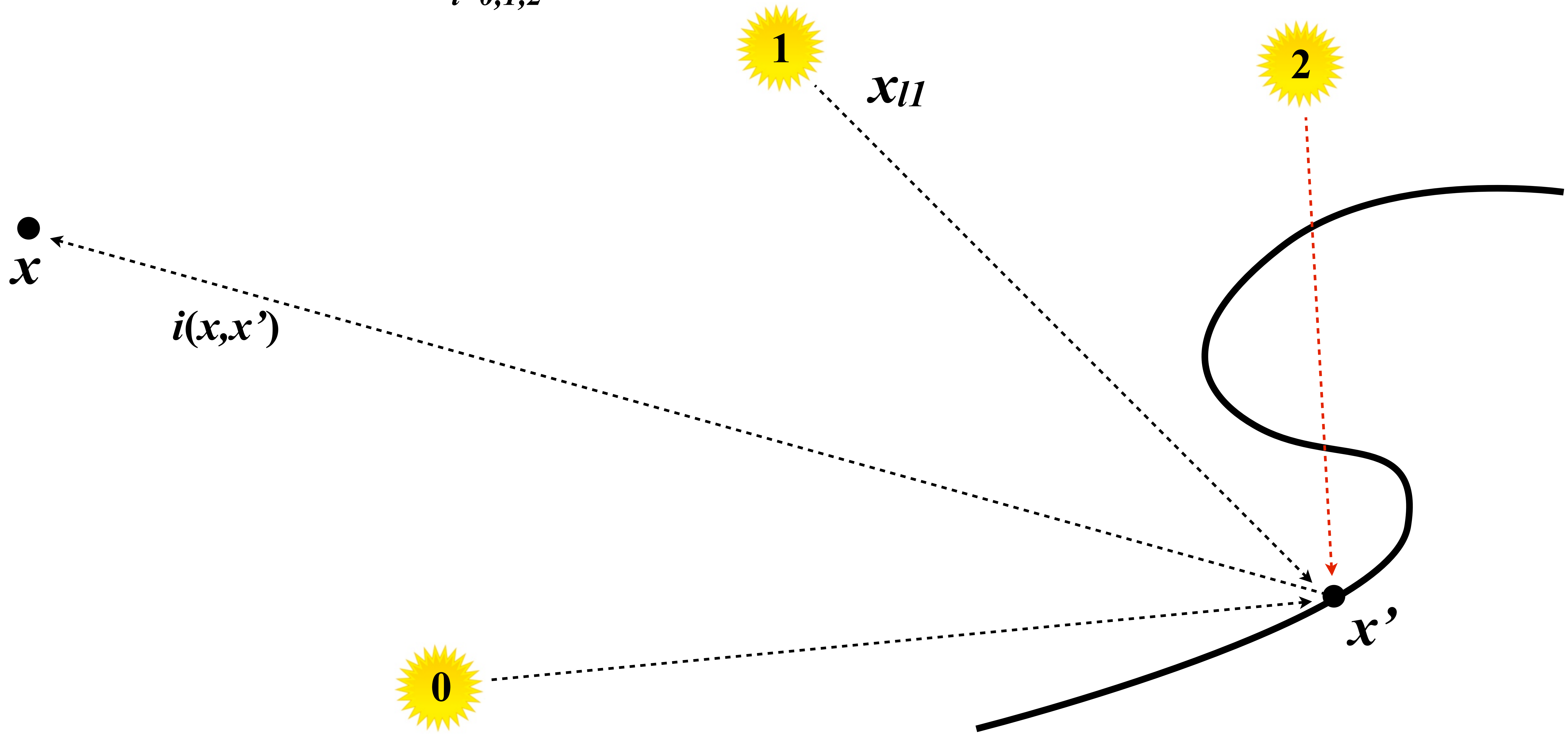
Tabulated BRDFs

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

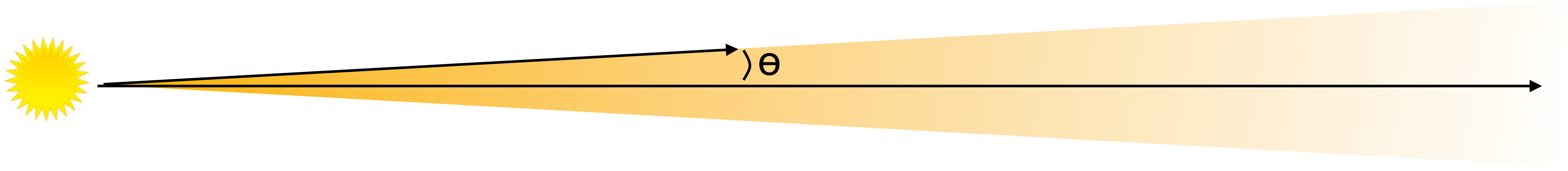
$$i(x, x') = \sum_{i=0,1,2} L_i v(x', x_{l_i}) r(x, x', x_{l_i})$$



More sophisticated lights

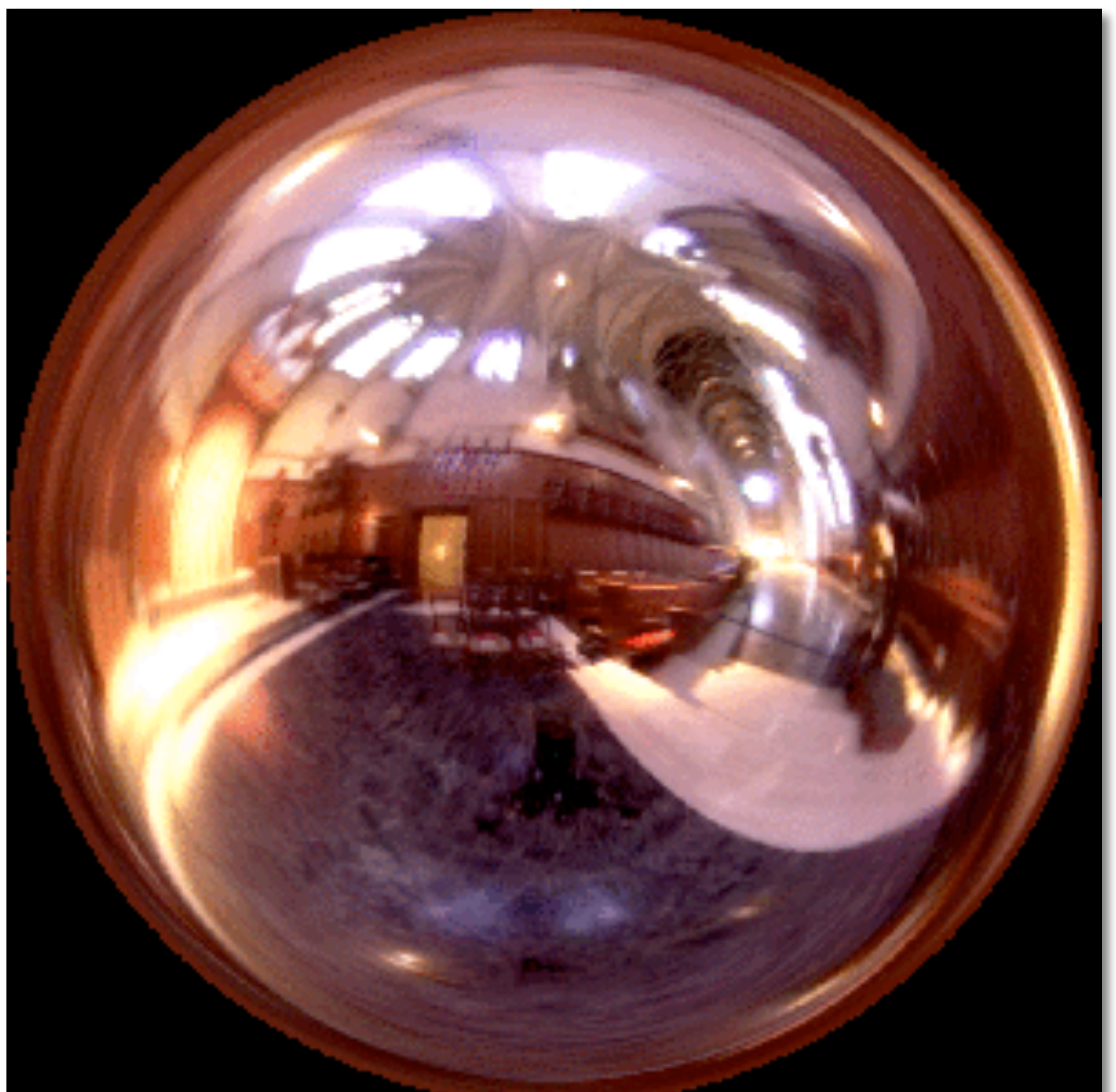
- **Attenuated omnidirectional point light**
  (emits equally in all directions, intensity falls off with distance: $1/R^2$ falloff)

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

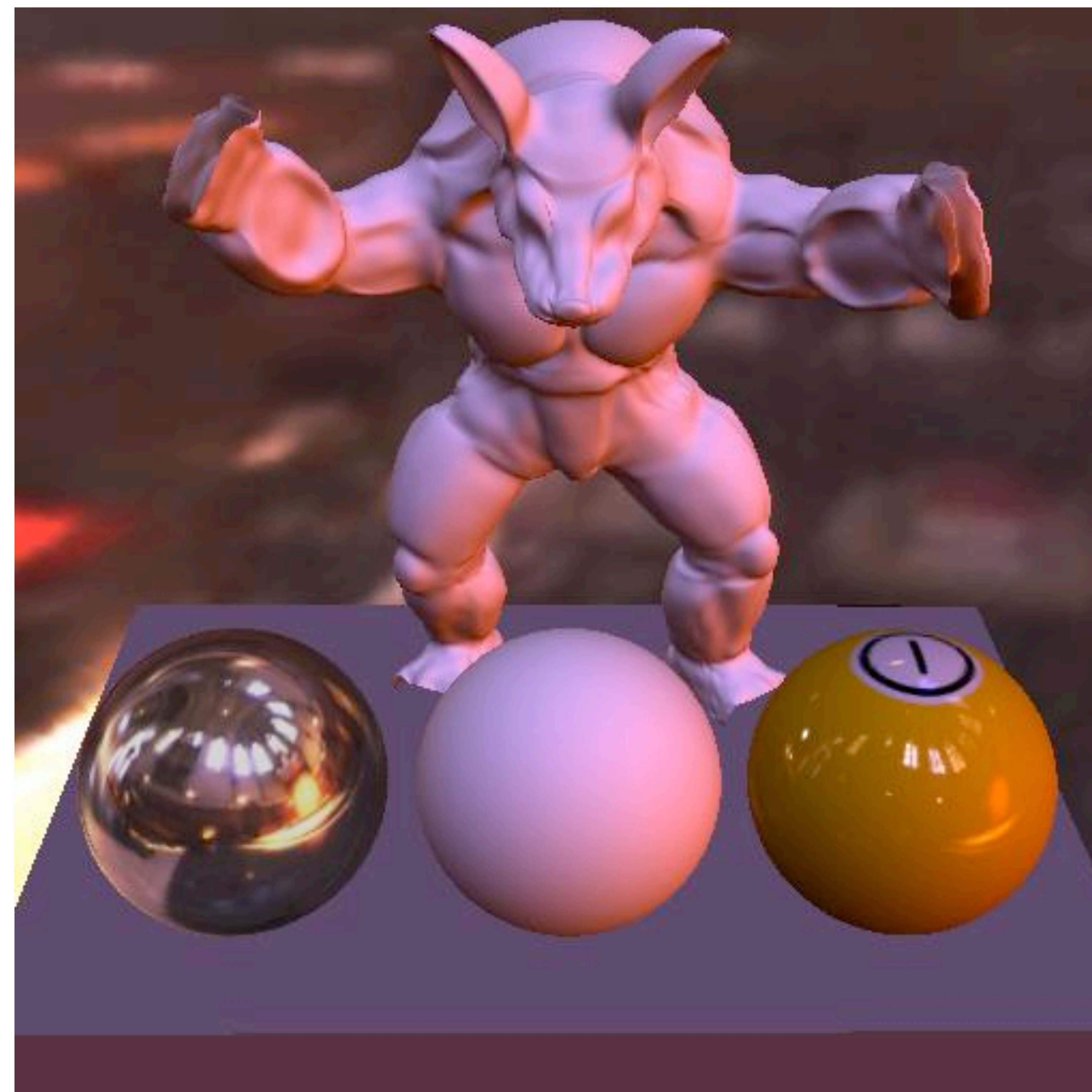


More sophisticated lights

- **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 OpenGL

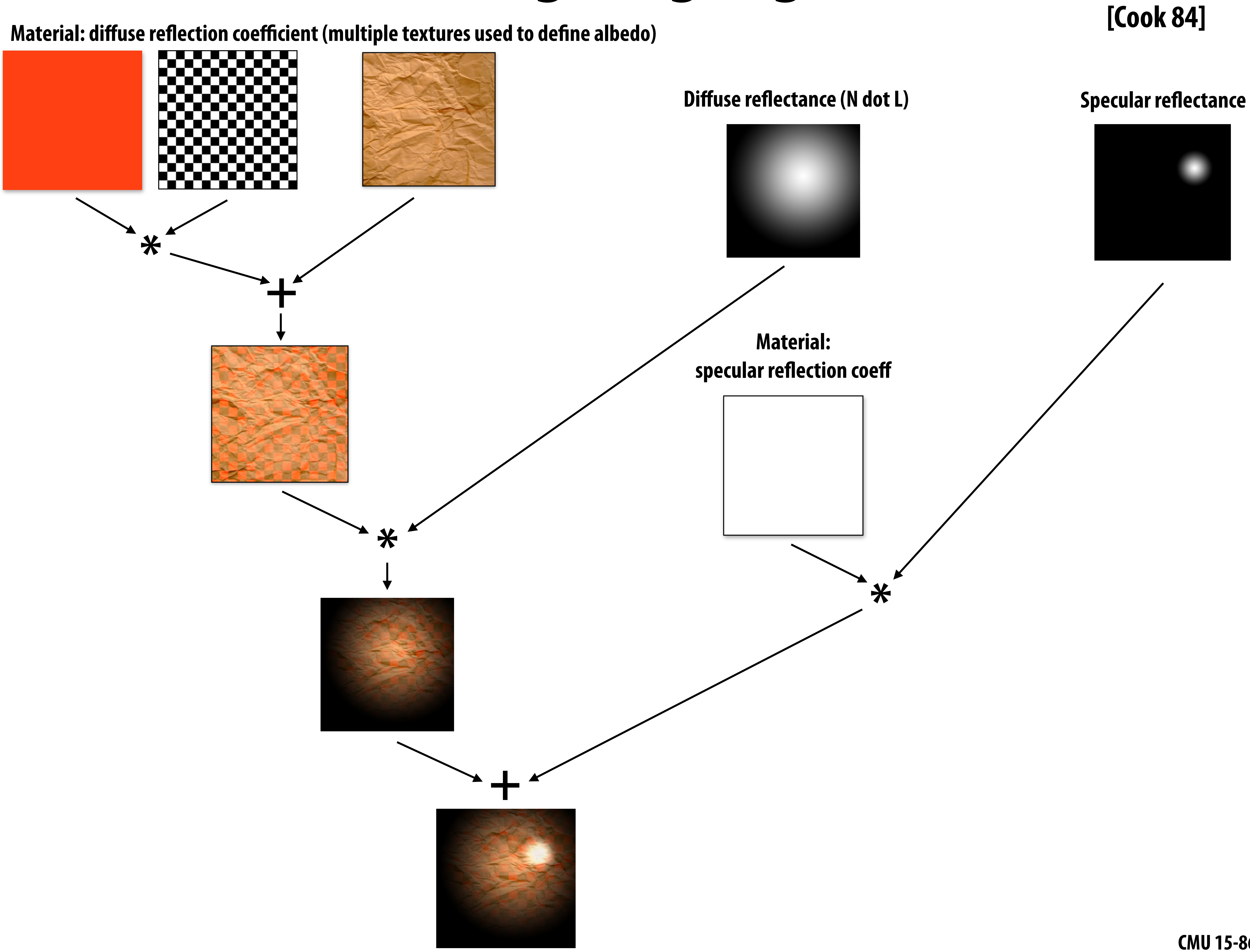
(prior to programmable shading)

- `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

Precursor to shading languages: shade trees



Shading languages

- **Goal: support wide diversity in materials and lighting conditions**

- **Idea: allow application to <u>extend</u> graphics pipeline by providing a programmatic definition of the shading function**

Tension: flexibility vs. performance

- Graphics pipeline provides high-performance implementation of visibility tasks
  - Examples: clipping, culling, rasterization, z-buffering
  - Highly optimized implementations on canonical data structures (triangles, fragments, and pixels)
  - Recall how 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
  - Allow applications to define these behaviors programmatically
  - Shading language forms interface between application-defined surface, lighting, material reflectance functions and the graphics pipeline

Shading language design questions

- **Issue: programmer convenience vs. application scope**
  - Adopt high-level (graphics-specific) or low-level (more flexibile) abstractions?
  - e.g., Should graphics concepts such as materials and lights be in the programming model?

- **Issue: preserve high performance**
  - Permit wide data-parallel implementation of fragment shader stage
  - Permit use of fixed function hardware for key operations (e.g., texture filtering)

Renderman shading language (RSL) [Hanrahan and Lawson 90]

- High-level, domain-specific language
  - Domain: describing propogation of light through scene

- Developed for Pixar's Renderman renderer

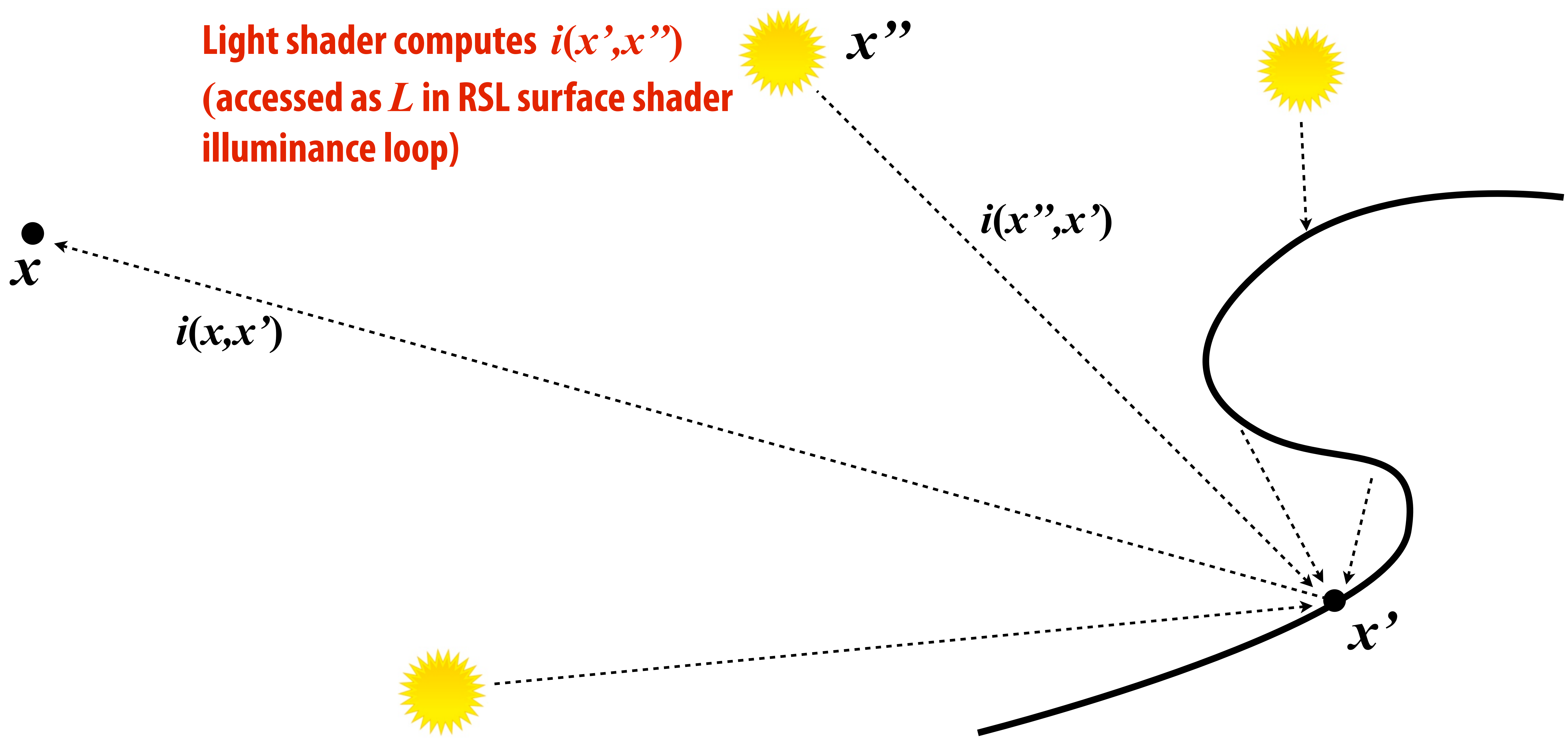
Key RSL abstractions

- Shaders: separate surface shaders and light shaders
  - Surface shaders:
    - Define surface reflection function (BRDF)
    - Integrate contribution of light from all light sources.
  - Light shaders: define directional distribution of energy emitted from light
  - Shader instances: shader code + environment state = closure
- Color, point data types
- Texture sampling functions
- Computation rates: uniform: independent of surface position (per surface), varying: change with position (per shading sample)
- Light shader's `illuminate` construct
- Surface shader's `illuminance` loop (integrate light)

Recall: rendering equation

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

Surface shader integrates contribution to reflection from all lights



Shading objects in RSL

compiled code
(e.g., plastic material)

current transforms

bound parameters
kd = 0.5
ks = 0.3

Surface shader object

compiled code
(spotlight)

current transforms

bound parameters
intensity =0.75
color = (1.0, 1.0, 0.5)
position = (5,5,10)
axis = (1.0, 1.0, 0)
angle = 35

compiled code
(point light)

current transforms

bound parameters
position = (5,5,5)
intensity =0.75
color = (1.0, 1.0, 0.5)

compiled code
(point light)

current transforms

bound parameters
position = (20,20,100)
intensity =0.5
color = (0.0, 0.0, 1.0)

Light shader objects
(bound to scene surface)

RSL light shaders

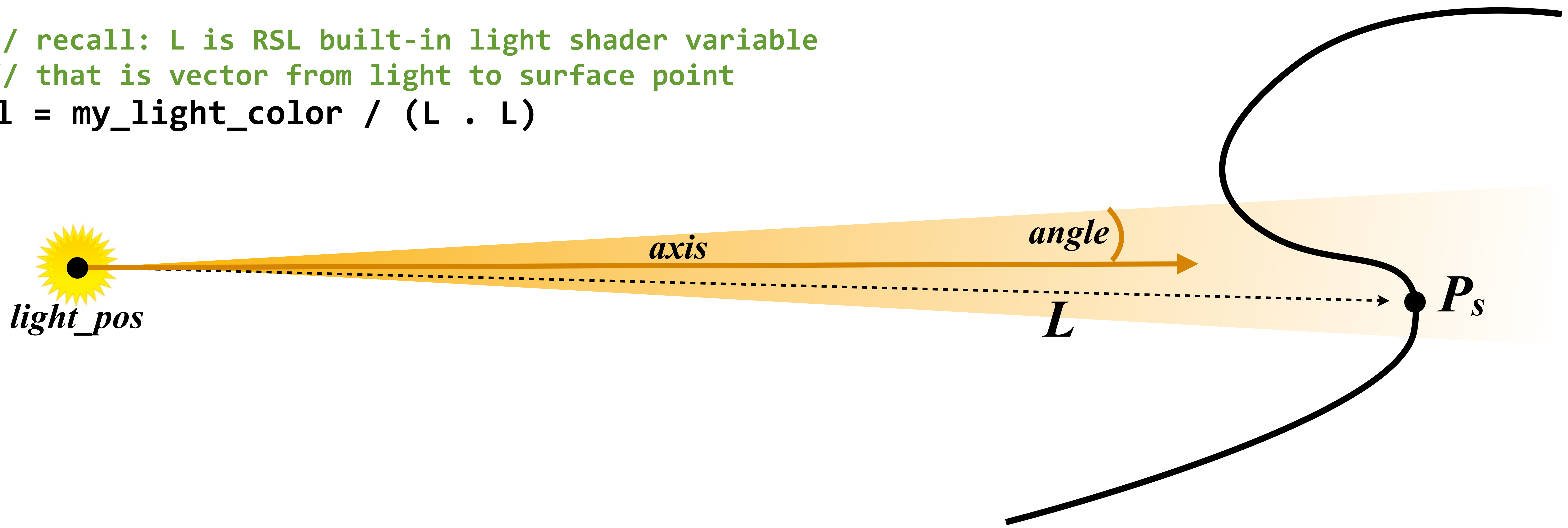
Key abstraction: illuminate block

```
illuminate (light_pos, axis, angle)
{

}
```

Example: attenuating spot-light (no area fall off)

```
illuminate (light_pos, axis, angle)
{
   // recall: L is RSL built-in light shader variable
   // that is vector from light to surface point
  Cl = my_light_color / (L . L)
}
```



RSL surface shaders

Key abstraction: illuminance loop

```
illuminance (position, axis, angle)
{

}
```

Example: computing diffuse reflectance

```
surface diffuseMaterial(color Kd)
{
   Ci = 0;

   // integrate light from all lights (over hemisphere of directions)
   illuminance (P, Nn, PI/2)
   {
      Ci += Kd * Cl * (Nn . normalize(L));
   }
}
```

`Cl` = Value computed by light shader

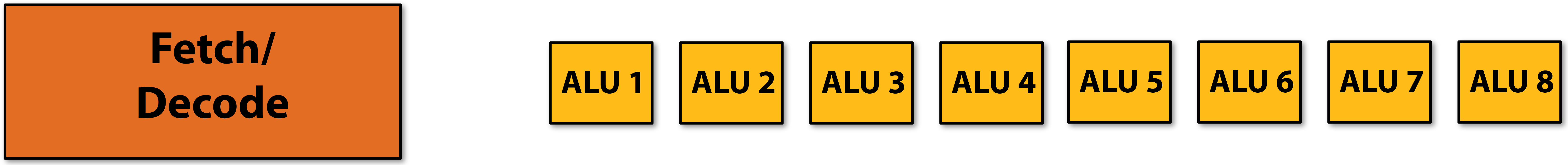
`L` = Vector from light position (recall `light_pos` argument to light shader's `illuminate`) to surface position being shaded (see P argument to `illuminance`)

Surface shader computes `Ci`

Discussion:
Design differences between RSL and Cg

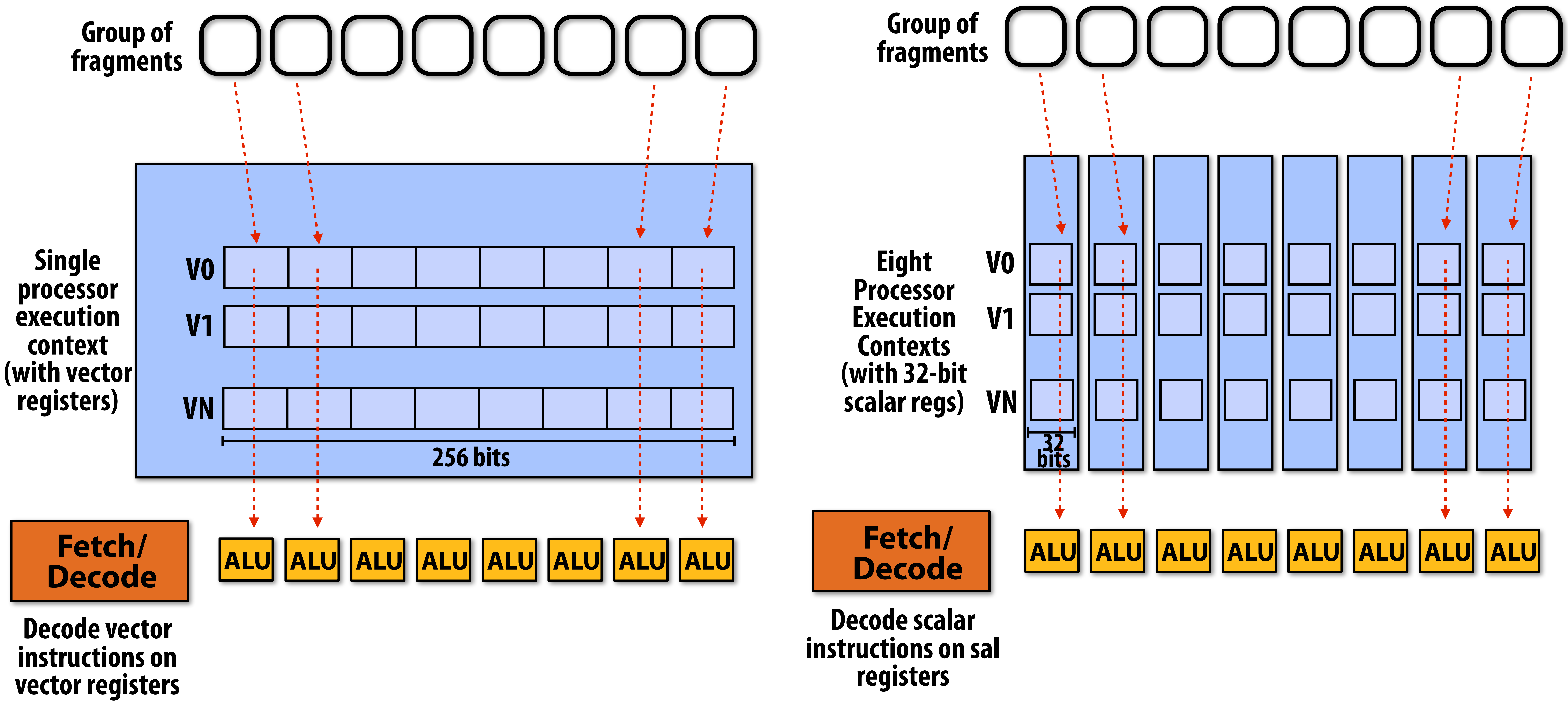
Implementing programmable shading

Review: fictitious throughput processor



- **Processor decodes one instruction per clock**
- **Instruction broadcast to 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 without one hardware execution context
- **Implicit SIMD (SIMT):**
  - Eight unique execution contexts operate in "lockstep"
  - Scalar instructions manipulate contents of 32-bit registers

Mapping fragments to "vector lanes" (explicit parallelism) or groups of SIMT threads (parallelization by hardware)



Review: SIMD execution

- **SIMD: single instruction multiple data**

- **How is SIMD execution expressed by a program?**
  - On modern CPUs: explicit short vector instructions: SSE (4-wide), AVX (8-wide)
  - Intel Xeon Phi: 16-wide short vector instructions
  - On older Cray "vector" processors: (instruction, array pointer, N)

- **A nice description of Intel's Xeon Phi vector instruction set:**
  – M. Abrash, A First Look at the Larrabee New Instructions (LRBni). Dr. Dobbs Portal, 2009

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**

- **Let's implement a SPMD program on a SIMD (whiteboard)**

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
  {
     tmp = backColor;
  }
  else
  {
     tmp = frontColor;
     tmp *= myTex.sample(mySamp, st);
  }
  return tmp;
}
```

Shader 2: nested conditional

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

float4 fragmentShader(float3 norm, float2 st, float4 frontColor, float4 backColor)
{
  float4 tmp;
  if (norm[2] < 0)  // sidedness check
  {
     tmp = backColor;
  }
  else
  {
     tmp = frontColor;
     if (fontColor == float4(1.0, 0.0, 0.0, 1.0))
        tmp *= myTex.sample(mySamp, st);
     else
        tmp *= 0.5;
  }
  return tmp;
}
```

Shader 3: while loop (homework)

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

float4 fragmentShader(float3 norm, float2 st, float4 frontColor, float4 backColor)
{
  float4 tmp;
  if (norm[2] < 0)  // sidedness check
  {
     tmp = backColor;
  }
  else
  {
     tmp = frontColor;
     while (tmp[0] < tmp[1])
     {
         tmp[0] += 0.1;
     }
  }
  return tmp;
}
```

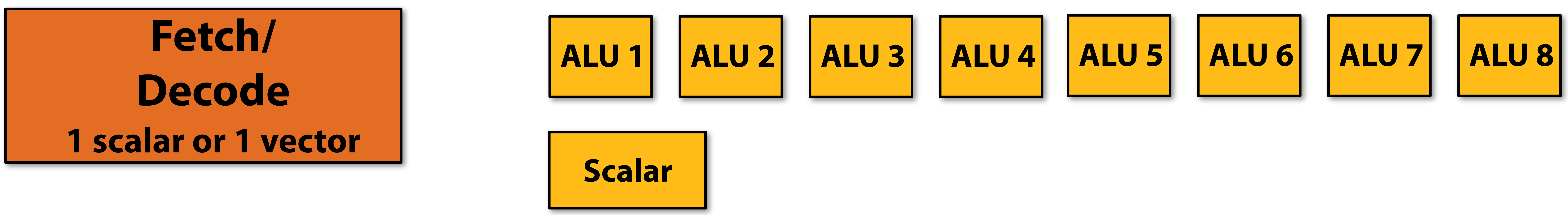
Shader 4: predicate is uniform expression

```
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

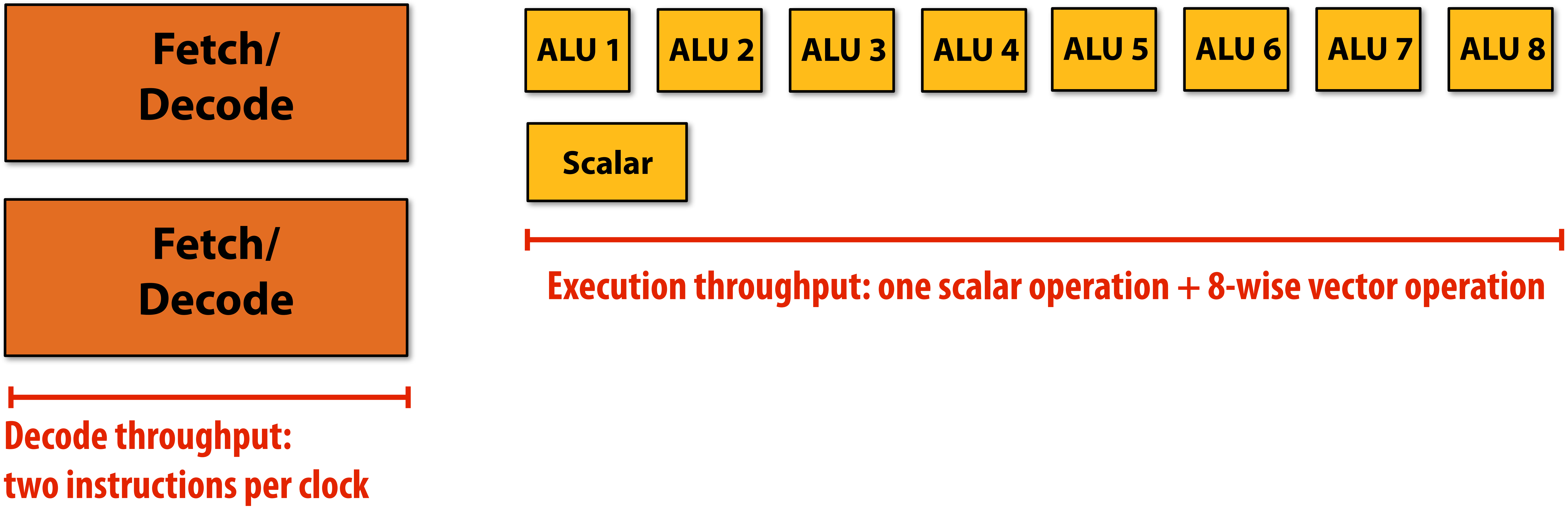
Improved efficiency: processor executes uniform instructions with scalar execution units



- **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)**

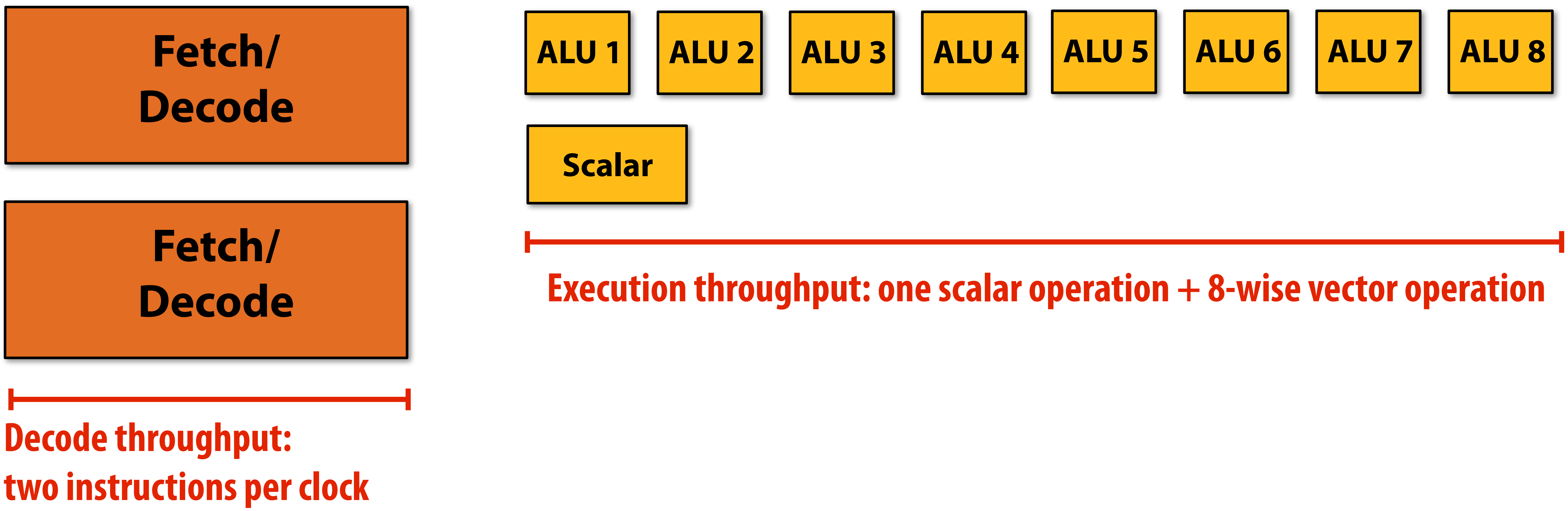
```
float3 lightDir[MAX_NUM_LIGHTS];
int numLights;
float4 multiLightFragShader(float3 norm, float4 surfaceColor)
{
   float4 outputColor;
   for (int i=0; i<num_lights; i++) {
     outputColor += surfaceColor * clamp(0.0, 1.0, dot(norm, lightDir[i]));
   }
}
```

Improving the fictitious throughput processor



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

Three possible organizations

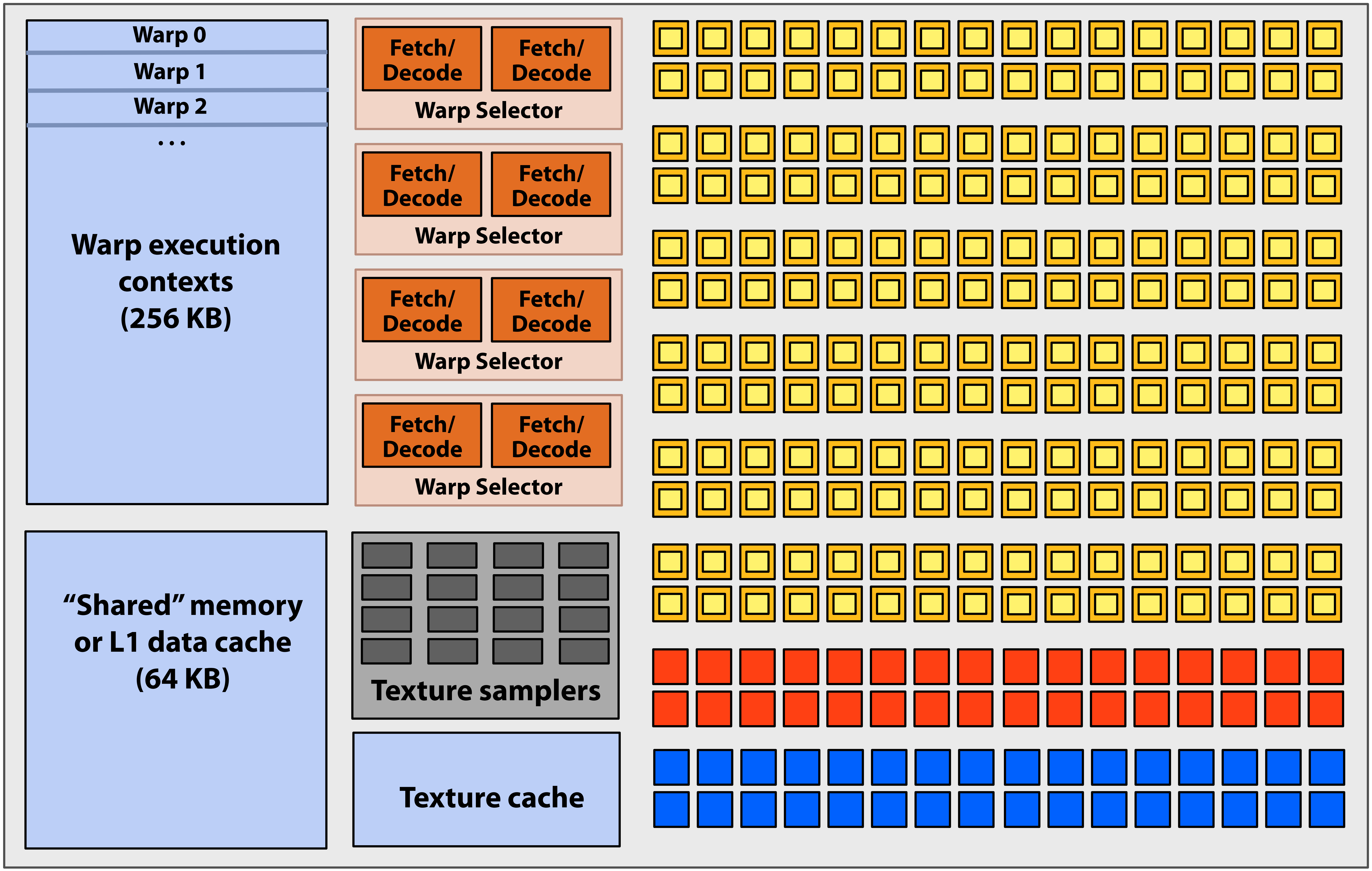


- Execute two instructions (one scalar, one vector) within same execution context
  - Only requires one execution context, but requires instruction-level-parallelism (ILP) in instruction stream
- Execute unique instructions in two different execution contexts
  - Processor needs two runnable execution contexts (twice as much parallel work must be available)
  - But no ILP in any instruction stream required to run machine at full rate
- 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)



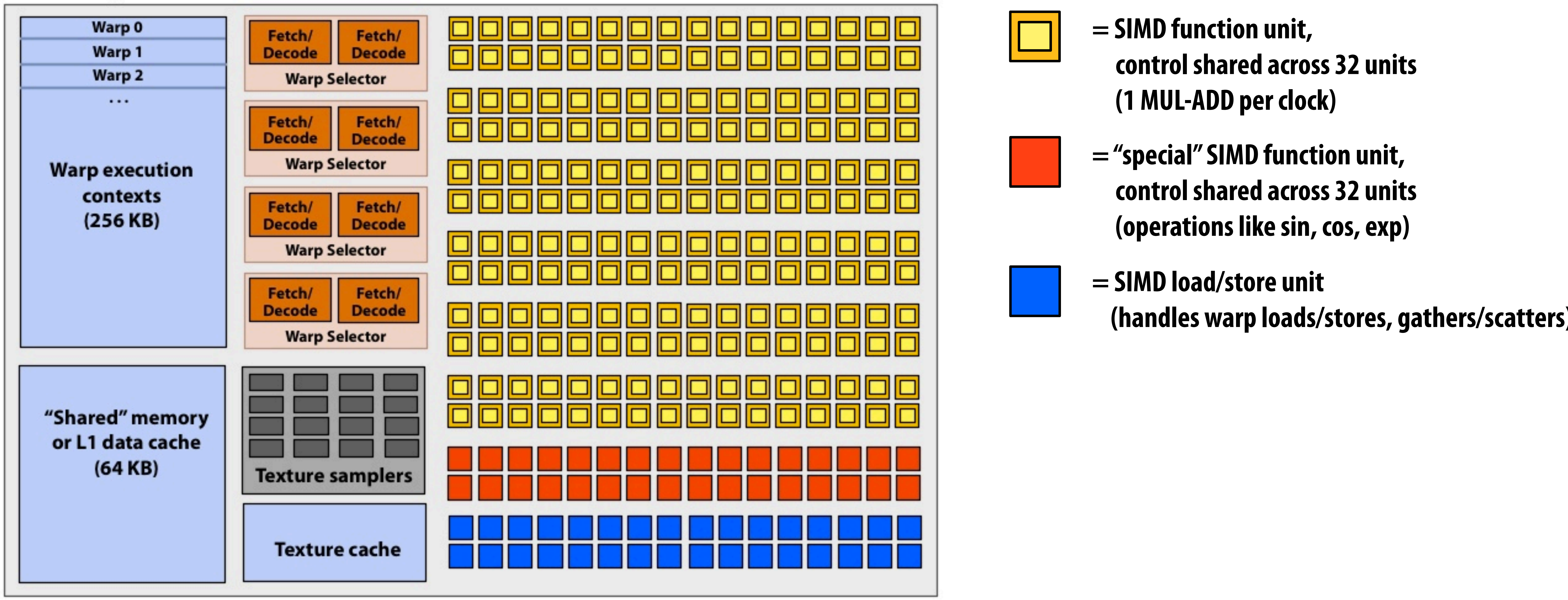
= SIMD function unit, control shared across 32 units (1 MUL-ADD per clock)

= "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")

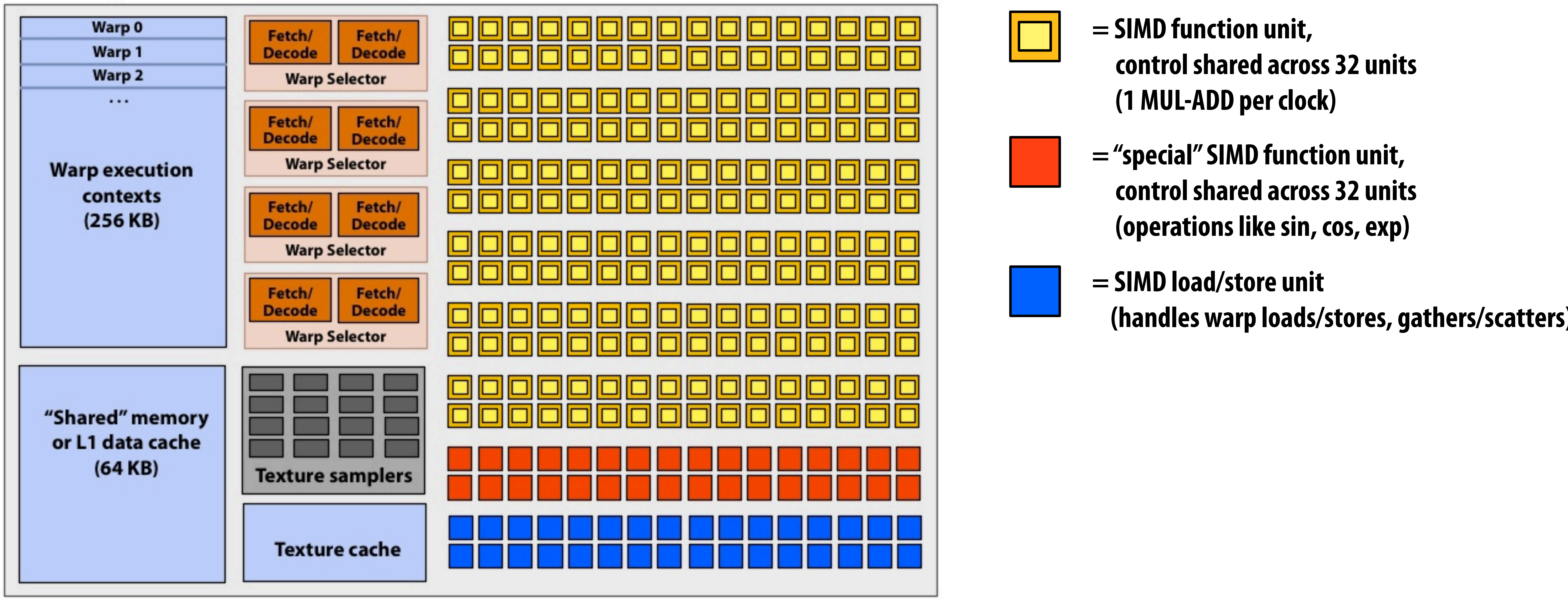


- **SMX core resource limits:**
  - Maximum warp execution contexts: 64 (2,048 total CUDA threads)

- **Why storage for 64 warp execution contexts if only four can execute at once?**
  - Multi-threading to hide memory access latency (texture latency in graphics pipeline)

NVIDIA GTX 680 (2012)

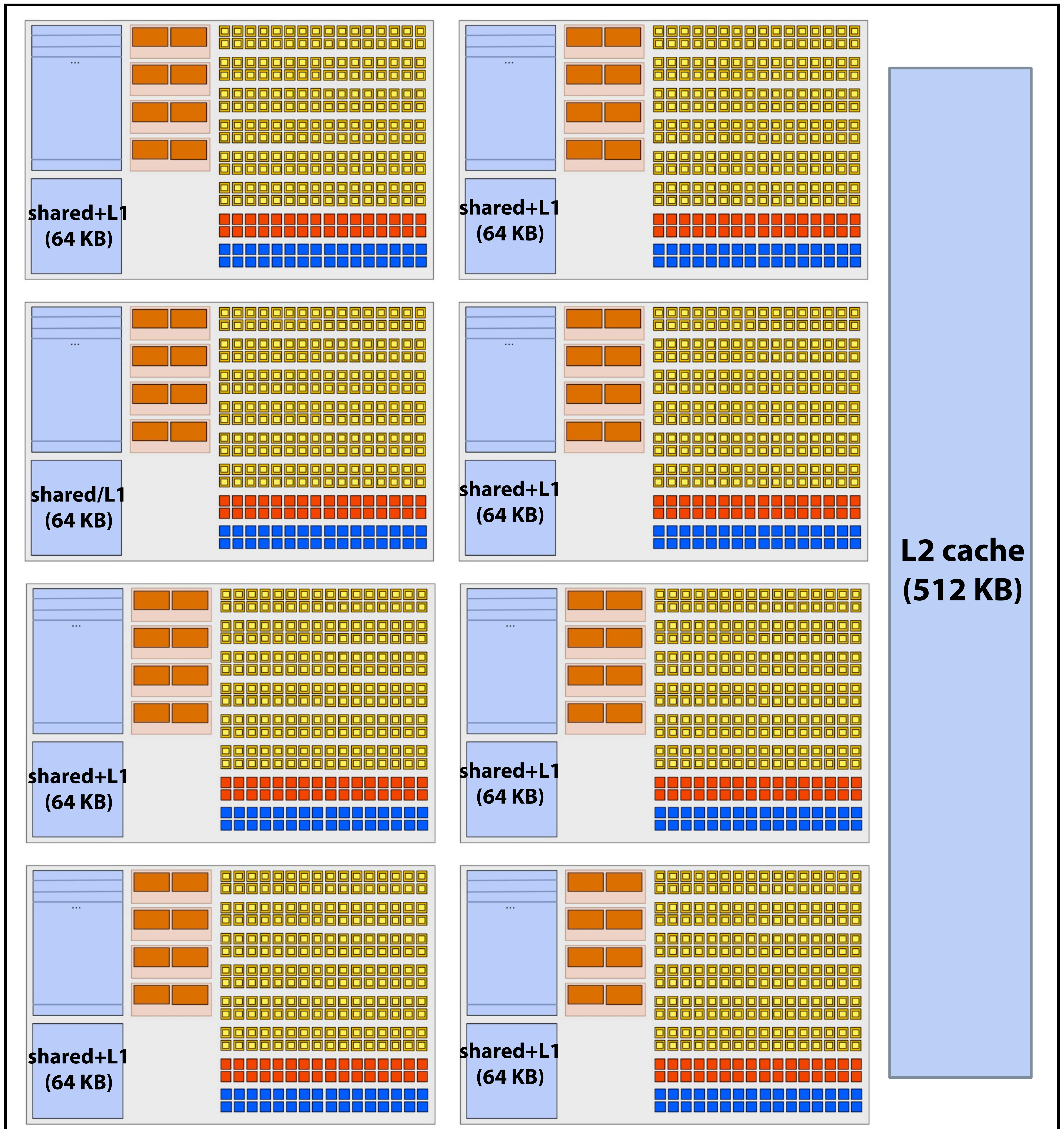
NVIDIA Kepler GK104 architecture SMX unit (one "core")



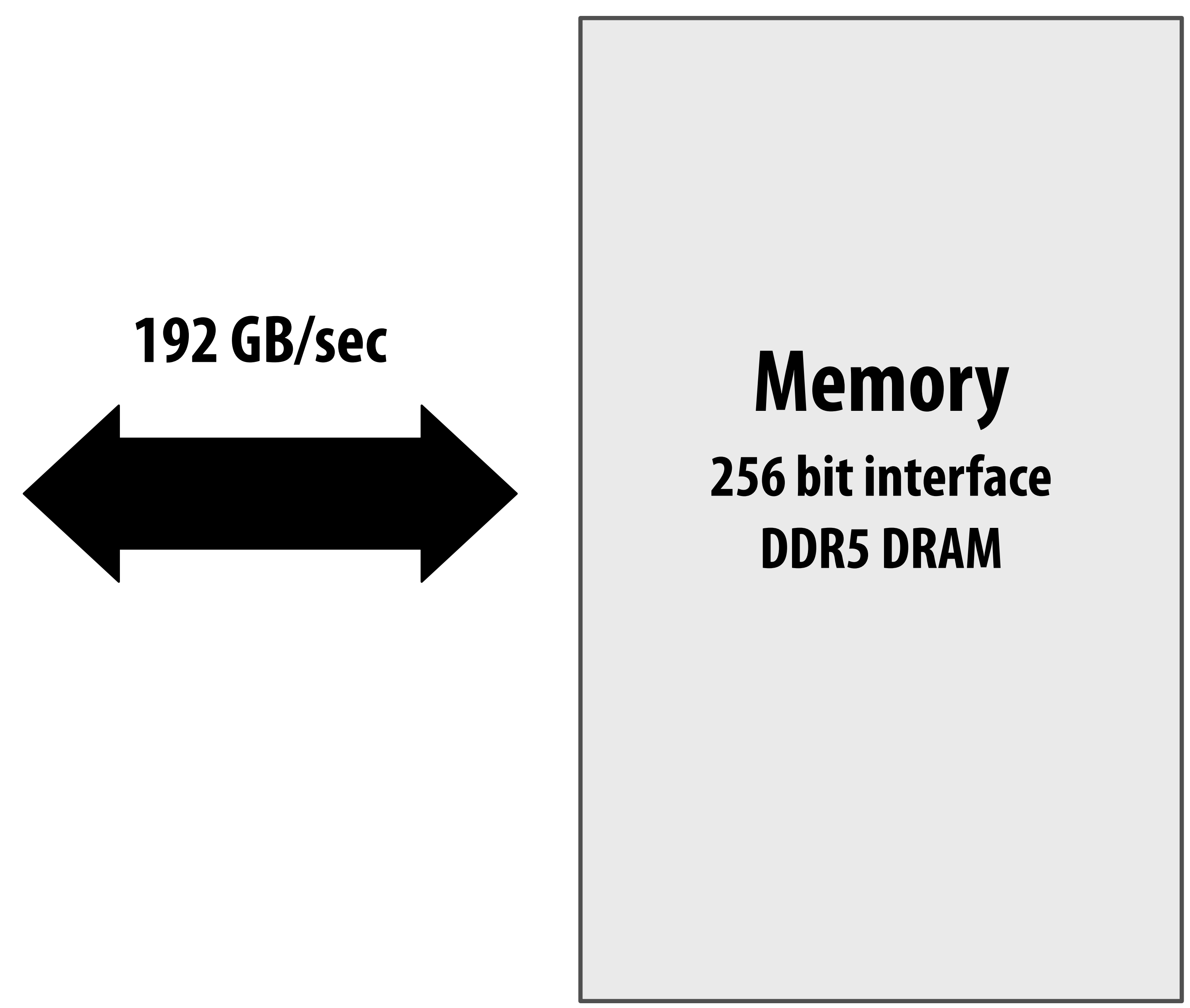
- **SMX programmable core operation each clock:**
  - Select up to four runnable warps from up to 64 resident on core (thread-level parallelism)
  - Select up to two runnable instructions per warp (instruction-level parallelism)
  - Execute instructions on available groups of SIMD ALUs, special-function 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



Arithmetic throughput to bandwidth ratio

- **Multiply add:**
  - $y = ax + b$
  - Three 32-bit inputs, one 32-bit output (16 bytes of data in total)

- **GTX 680 example: 1,536 SIMD multiply-add ALUs at 1 GHz**
  - Requires 22 TB/sec of bandwidth
  - About 100 times more than GPU memory system can provide

- **Programs must exhibit high arithmetic intensity (in addition to significant parallelism) to run efficiently on GPU cores**
  - So most instructions sourced by values in register file (or nearby caches)
  - Arithmetic intensity: ratio of math instructions to memory accesses

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)

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

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 elements

- Application-programmable logic forms the inner loops of the rendering computation, not the outer loops!

- Ongoing research question: can we flip this design around?
  - Maintain efficiency of heterogeneous hardware implementation, but give programmers control of how hardware is used and managed.

Readings

- **T. Foley et al.** ***Spark: Modular, Composable Shaders for Graphics Hardware.*** **SIGGRAPH 2011**
- **J. Nickolls et al.** ***Scalable Parallel Programming with CUDA*****. ACM Queue 2008**