graphics hardware

http://tinyurl.com/lrsmfj6

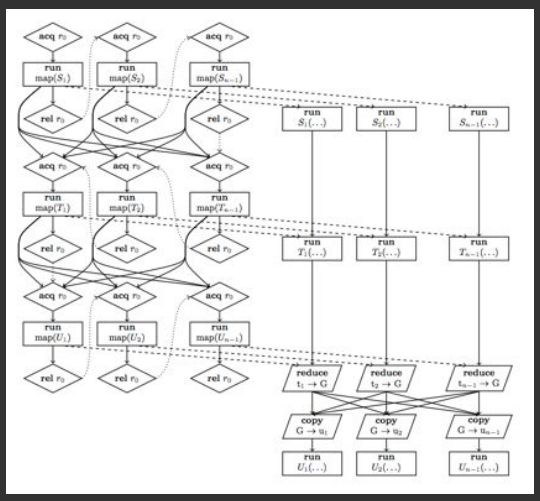
CS148 / 24 july 2014

today

GPU tutorial w/ special guest!

programmable shaders (GLSL)

mike bauer
parallel computing commodore



http://web.stanford.edu/~mebauer/publications.html

GPU Architecture
CS 148 Summer 2014

Michael Bauer

Thanks to Kayvon Fatahalian (CMU)

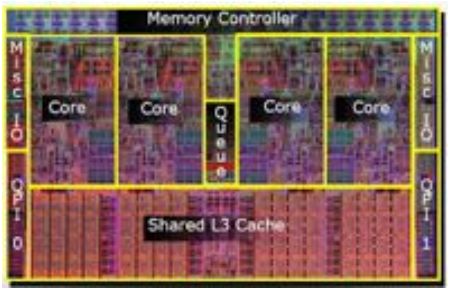
Latency vs. Throughput

- Why do GPUs exist?
- Common engineering problem:
  - Optimize for throughput or latency?
  - Can't do both

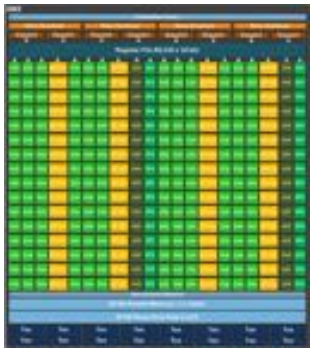
Simple Analogy: Railroads

- Optimize for Latency:
  - Europe
  - Fast passenger trains
  - 10s trains per day
  - 250+ mph
  - Hours between cities
- Optimize for Throughput
  - USA
  - Slow freight trains
  - 100s trains per day
  - 40-50 mph max speed
  - Days between cities
  - Massive throughput

Comparing CPUs and GPUs



- Intel Core i7 (Latency)
  - Out-of-order execution
  - Branch prediction
  - Superscalar
  - Multi-level caches
  - 25 stage pipeline

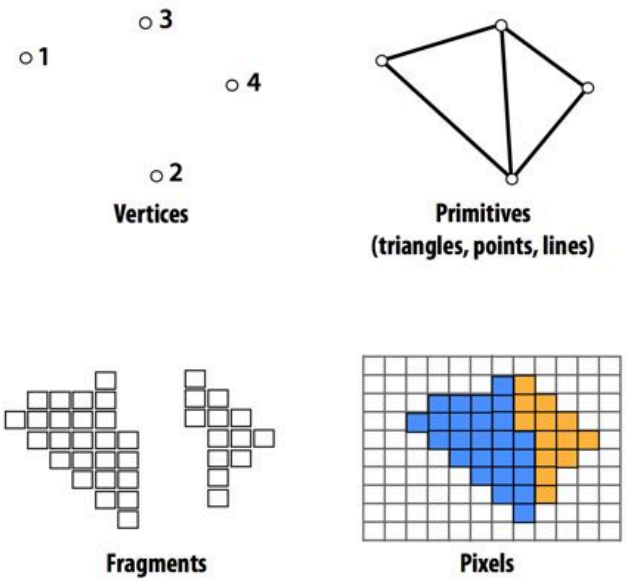


- NVIDIA K20 (Throughput)
  - In-order execution
  - 32-wide SIMD units
  - Lots of ALUs
  - High-bandwidth memory

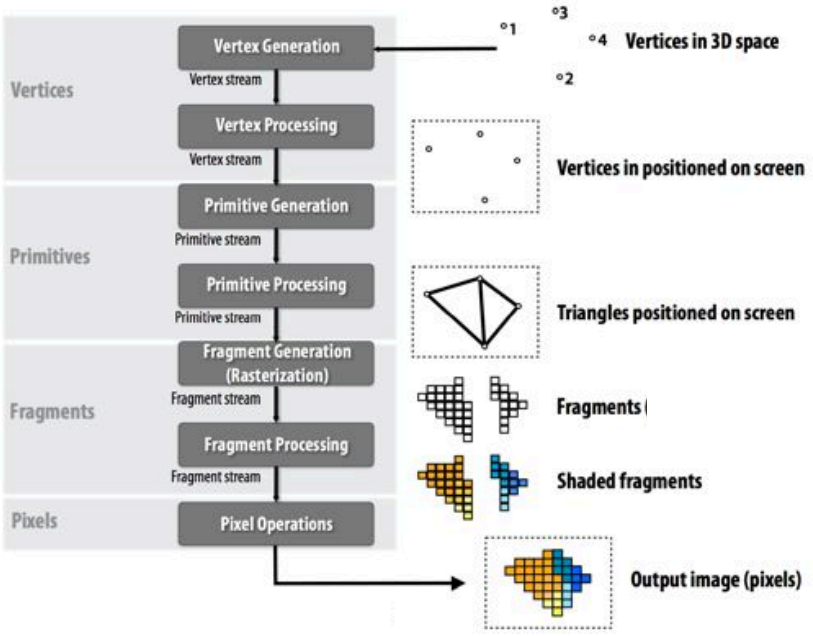
Graphics == Throughput

- Humans are "slow"
  - Perception ~30 ms
- Do we care about how long it takes to render 1 pixel?
  - Not really
- What do we care about: how long to render 1 frame?

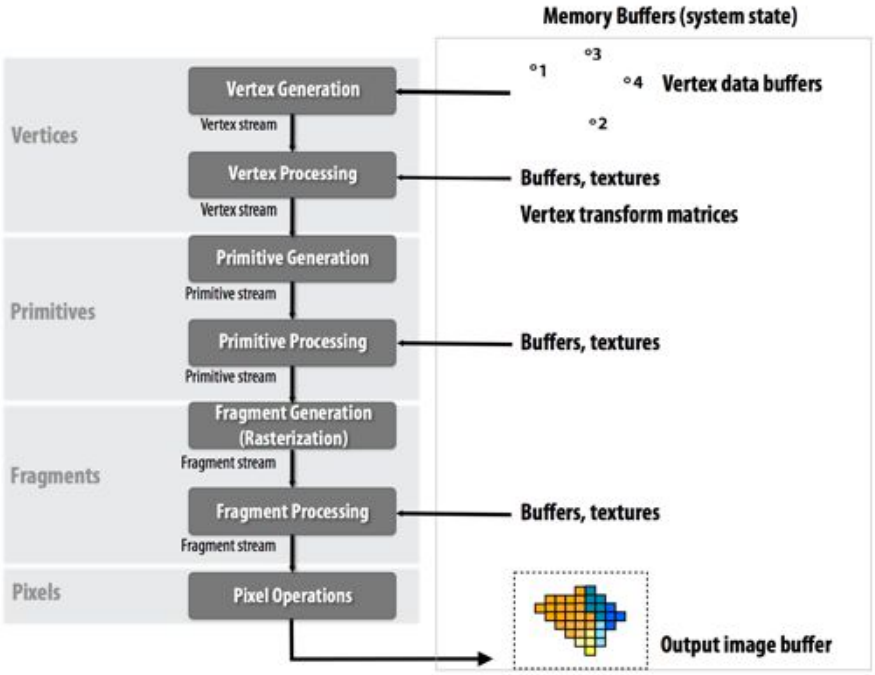
OpenGL Pipeline Primitives



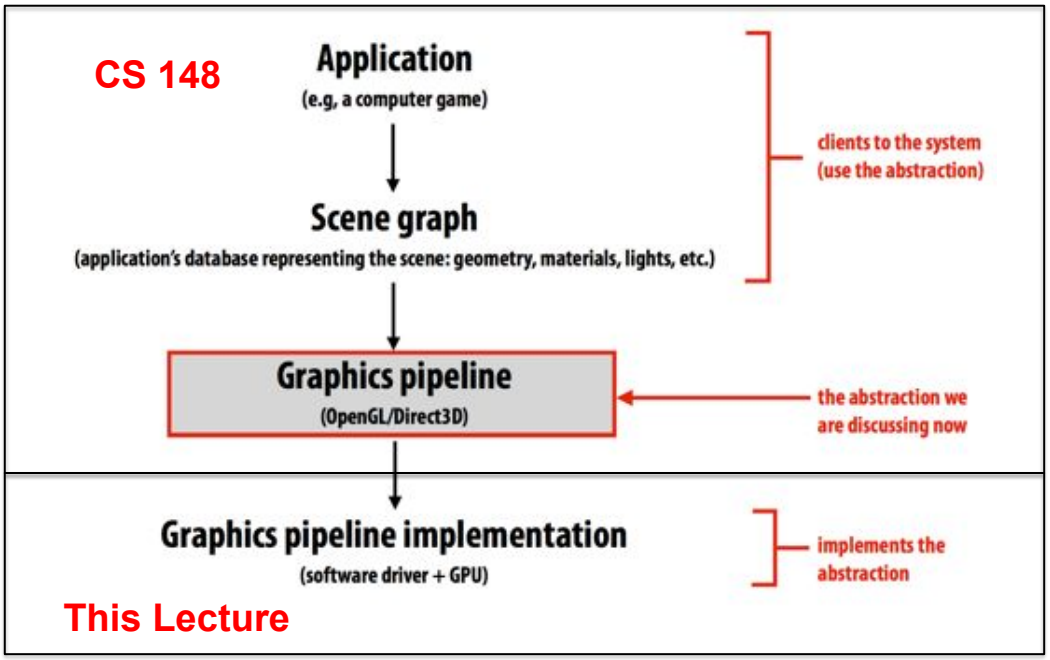
OpenGL Pipeline Operations



OpenGL Pipeline State

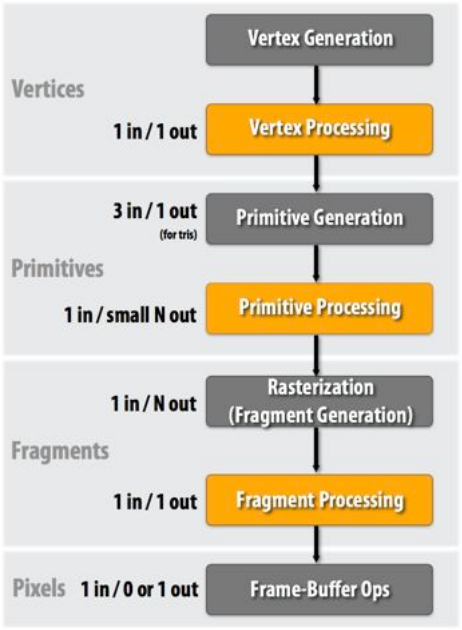


3D Graphics System Stack



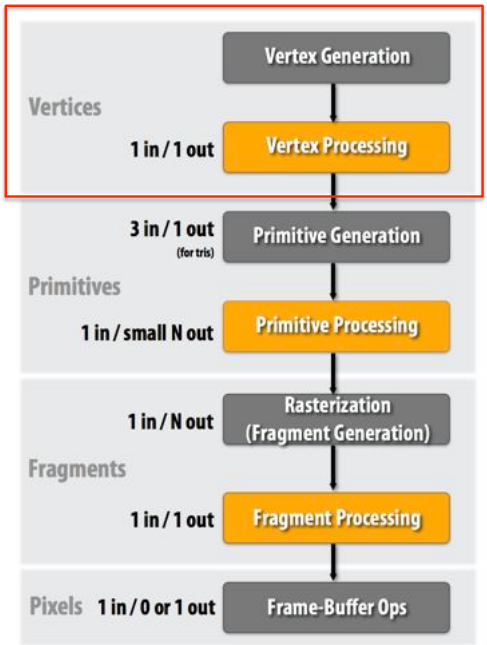
Fixed Function vs. Programmable

- Natural tension in the graphics pipeline
- Which stages should be programmable and which are fixed function?
- Programmable
  - Adaptable, but slower
- Fixed Function
  - Fast, but rigid



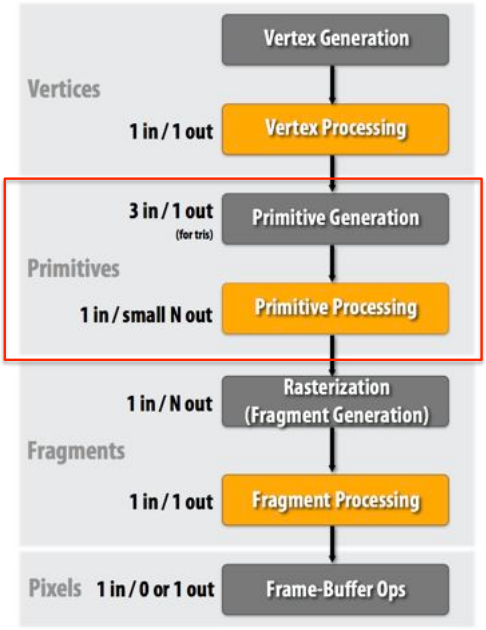
Vertex Processing

- Fixed function
  - Specify vertecies
  - Attach attributes (up to 16 128b attributes each)
- Programmable: vertex shaders
  - Bare minimum: apply transform matricies
  - Optional: compute values based on physics, lighting, etc



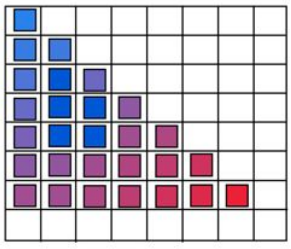
Primitive Processing

- Fixed function
  - Generate triangles from verticies, 3 in 1 out
- Programmable:
  - Can refine triangles into sub-triangles
  - Limited few output
  - Advanced: tessellation shaders

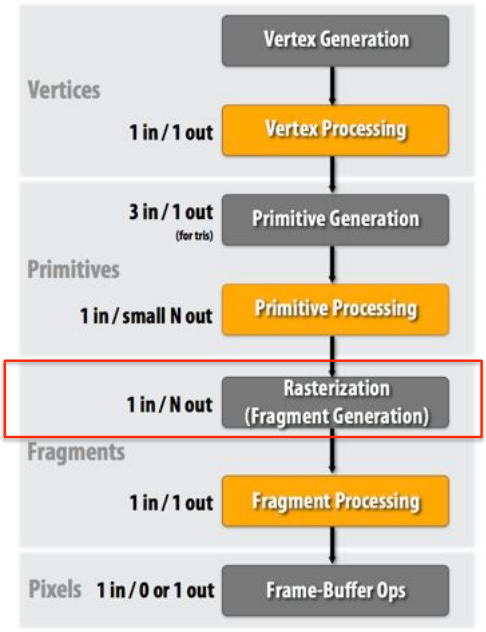


Rasterization

- Fixed function:
  - Convert primitives into one fragment per pixel
  - Currently done entirely in hardware

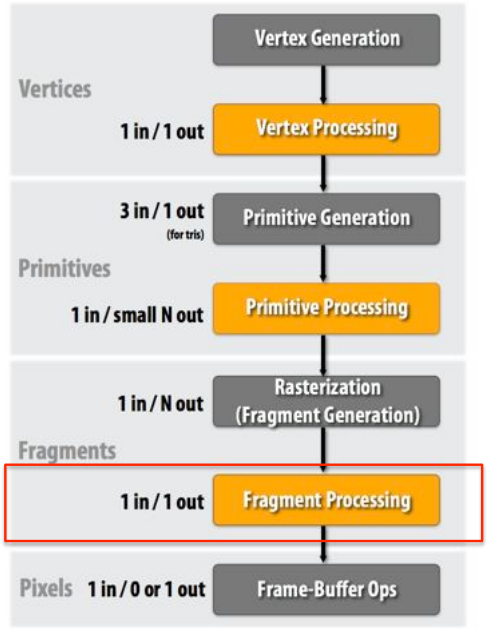


- Programmable:
  - Control sampling granularity



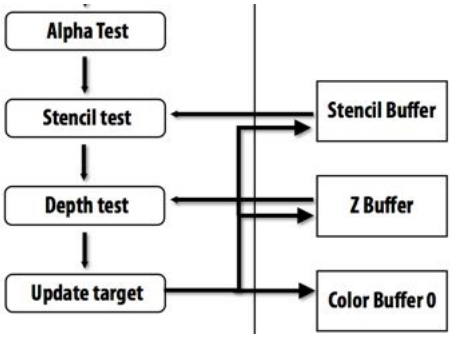
Fragment Processing

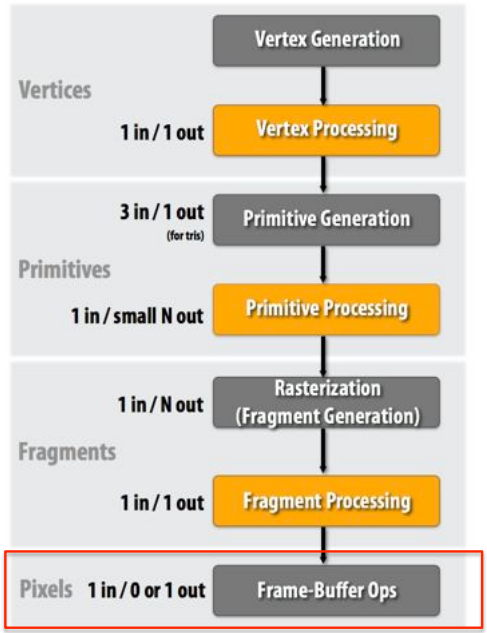
- Programmable:
  - Compute color output for each fragment
  - Based on textures, lighting, physics, vertex data, whatever...
- Written as fragment shaders
  - More on this later…



Frame Buffer

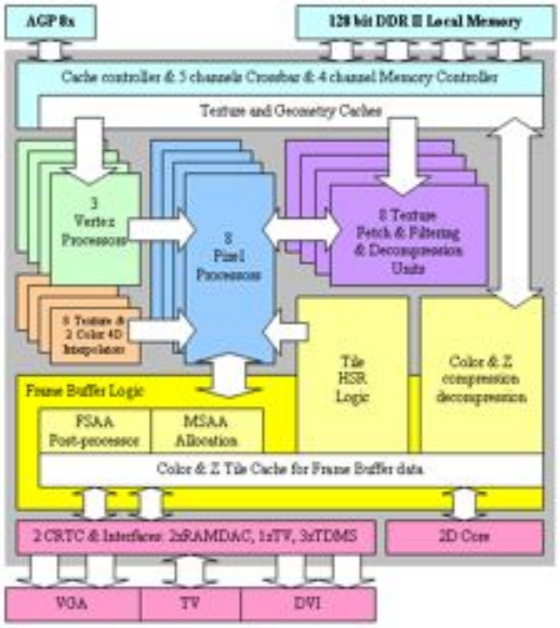
- Fixed function:
  - Order fragments rendered on the same pixel
  - Sample, apply alpha blending, z-depth test





The First GPUs (the dawn of time)

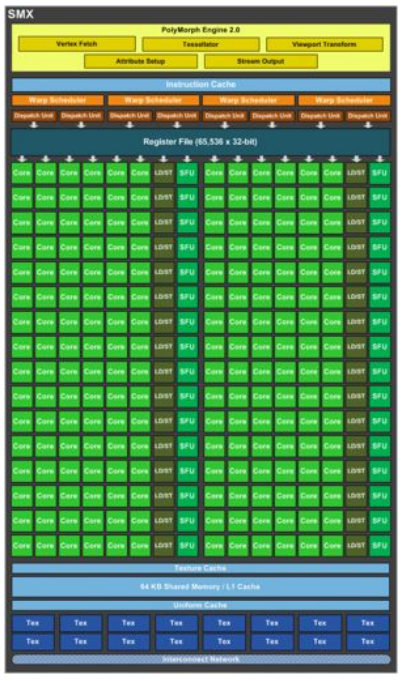
- NV30
  - November 18, 2002
- Separate hardware units for each pipeline stage
  - Vertex processors
  - Pixel processors
  - Frame buffer logic
- Mapping OpenGL onto hardware is easy
  - It's baked in



Modern GPUs

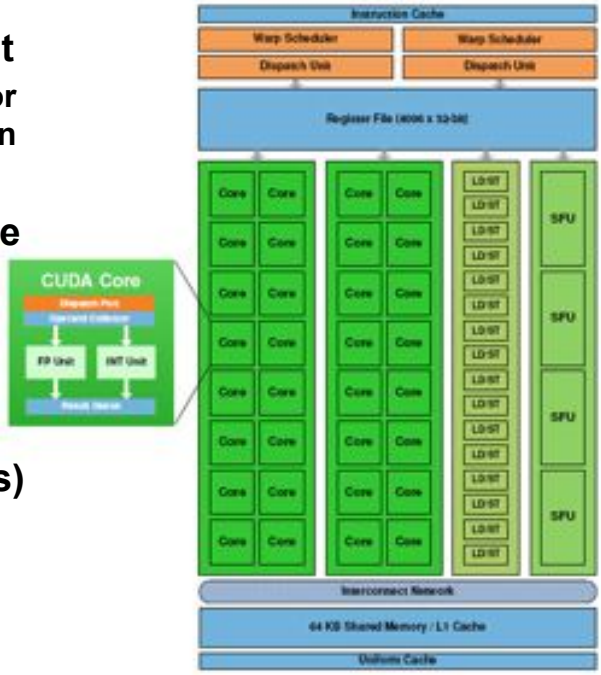
SM = Streaming Multiprocessor
4 – 16 per GPU

- General purpose computation cores (green)
- Simplify implementation, design, testing
- Support more programmable graphics
- Money in general purpose (GPGPU) computing
- Still fixed function units
  - Rasterizer, texture caches



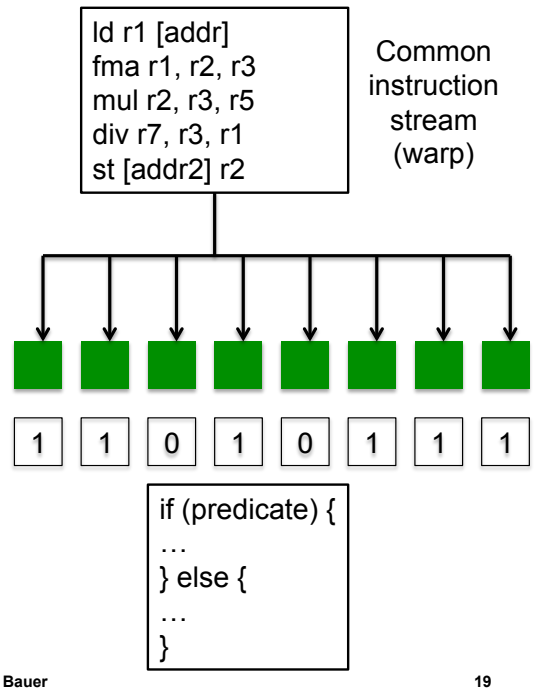
Compute Cores

- Optimize for throughput
  - Use most transistors for performing computation
- Shader core/ CUDA core are really just ALUs
- All cores share a common instruction stream (32-wide vectors)
  - Single Instruction Multiple Data (SIMD)
  - Like SSE/AVX



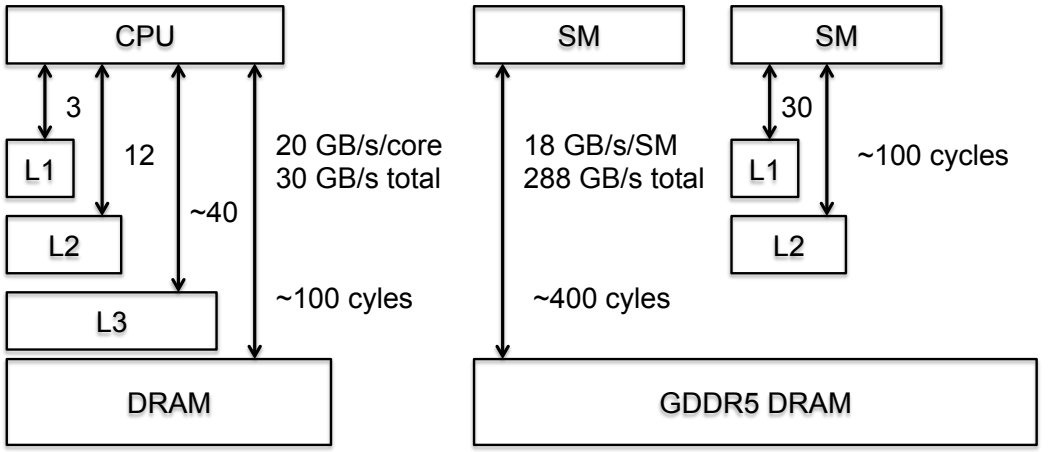
Running Code on Compute Cores

- SIMD continued…
  - Shared I-cache, dispatch unit, register file
  - Amortize overhead
  - More transistors for math
- Each core is called lane
- Special hardware support for branches
  - Mask off unused lanes
  - Can serialize execution



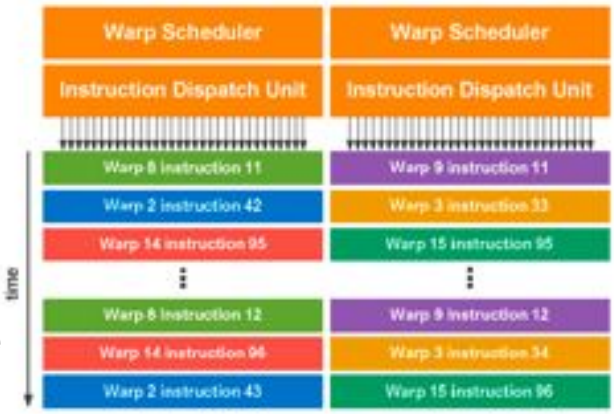
Memory Hierarchy

- GPU memory systems are optimized for throughput
- High bandwidth, high latency
- No cache coherence (more transistors for math)



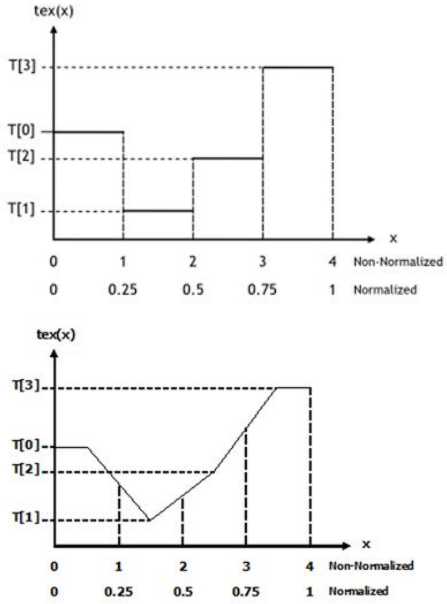
Hiding Memory Latency

- Instruction streams usually miss on loads and stores
  - Need to hide memory latency
- Multiplex different instruction streams (warps) onto hardware
- Hardware only idle if there is nothing to do



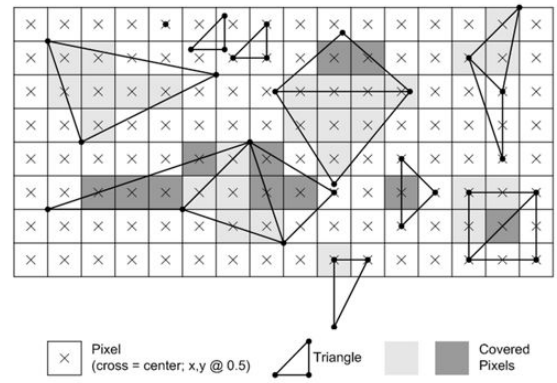
Texture/Constant Caches

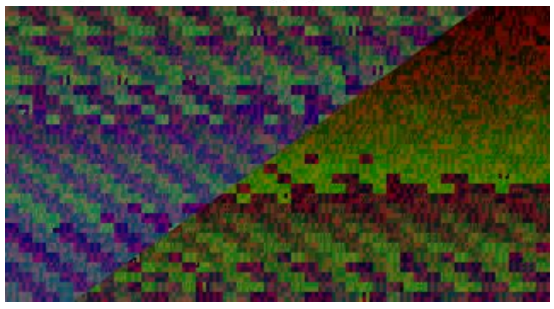
- Very small (~8-12KB) read-only caches near the compute core
  - Non-coherent
- Constant caches
  - Uniform variables
  - Transform matricies
- Texture caches
  - Specialized interpolation operations
  - Automatic normalization
  - Optimized for 2D locality



Fixed Function Redux: the Rasterizer

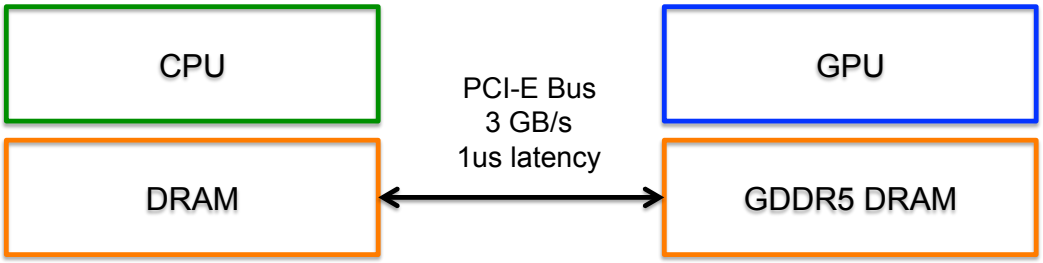
- Still fixed function
  - More on this later
- Many special cases
- Support for anti-aliasing
  - Interpolation based on adjacent fragments
- Need efficient ways of traversing screen space





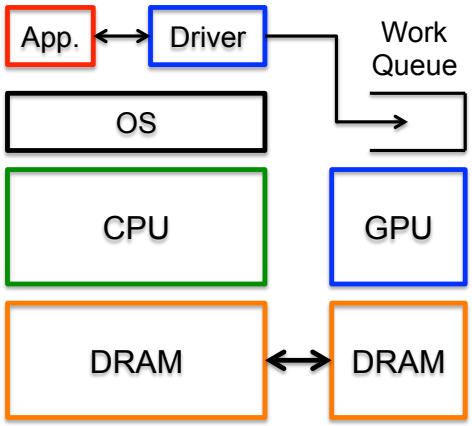
System Architecture

- Many (especially high-end) GPUs are discrete
- Separate components on the motherboard
- PCI-Express Bus for communication
  - Long-latency and low-bandwidth



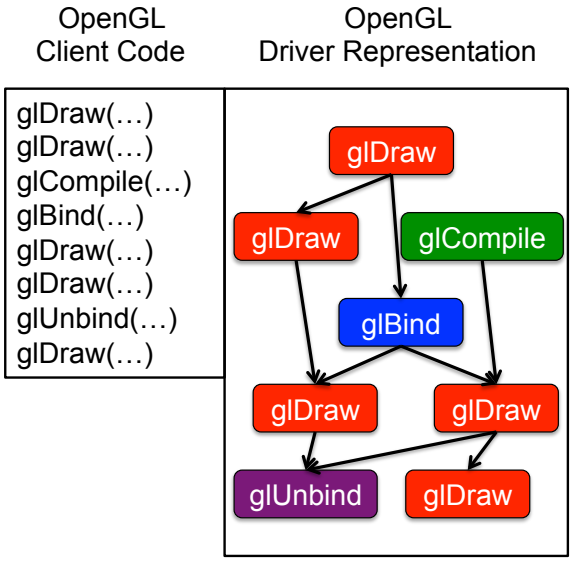
OpenGL/GPU Drivers

- Take commands from OpenGL client code
  - Convert to device commands
  - Schedule onto GPU
- Several responsibilities
  - Hide latency of communication to GPU
  - Optimize GPU compute
  - Manage limited memory in GPU framebuffer



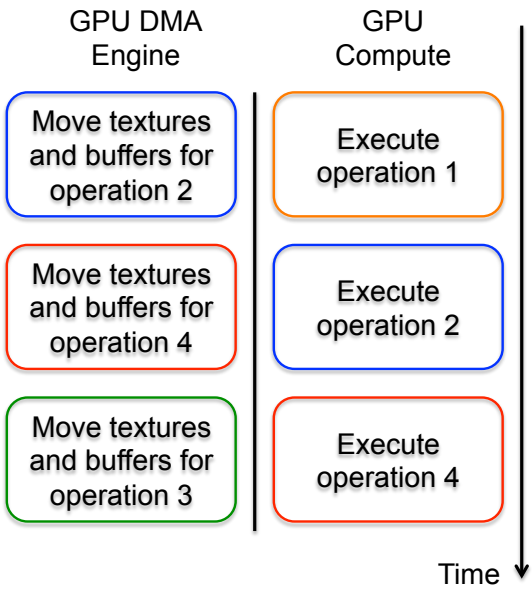
OpenGL Deferred Execution Model

- All OpenGL calls are asynchronous
  - Operations are deferred
- Client code runs ahead of actual GPU rendering
- OpenGL driver constructs operation DAG with dependences
  - Dependences based on buffers and textures



Hiding Communication Latency

- Deferred execution allows OpenGL driver to overlap computation and data movement
- Move data for next operation while previous operation in flight
- Also save buffer overhead in limited framebuffer DRAM

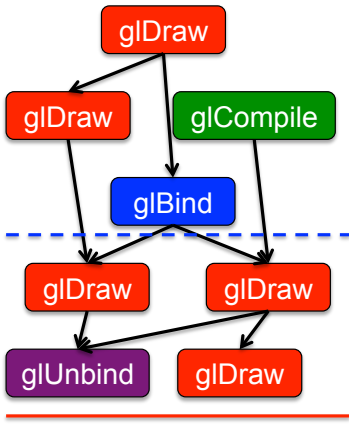


glFlush and glFinish

- Two most misused commands in OpenGL
- glFlush – execute all prior operations in finite time (non-blocking)
  - Optimization hint
- glFinish – wait for all prior operations to terminate execution
  - Should rarely be used

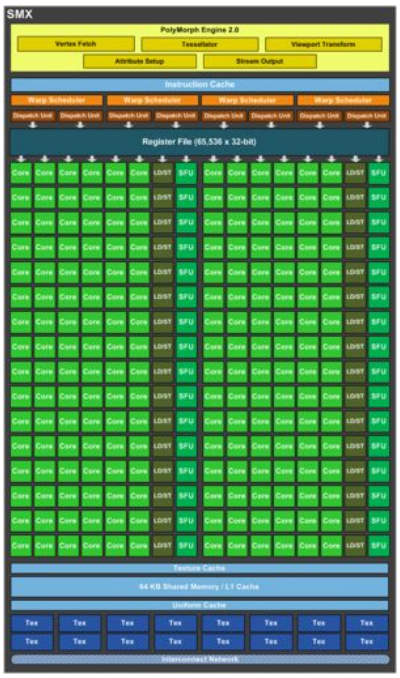
OpenGL Client Code

```
glDraw(…)
glDraw(…)
glCompile(…)
glBind(…)
glFlush();
glDraw(…)
glDraw(…)
glUnbind(…)
glDraw(…)
glFinish();
```



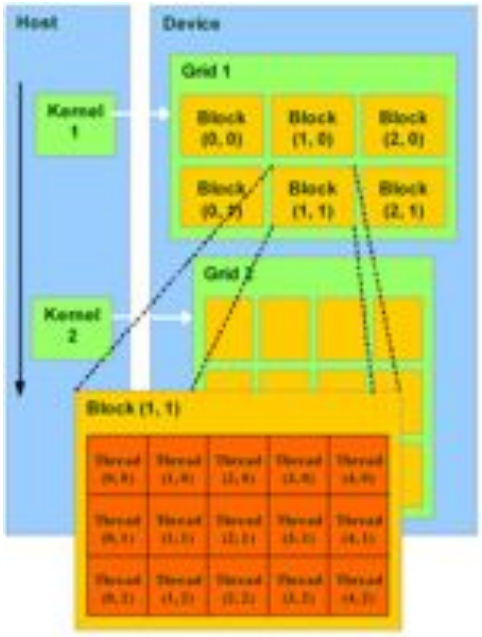
General Purpose (GP) GPU Programming

- GPUs have tons of floating point units
- Who needs to do lots of floating point math?
  - Supercomputing
  - Machine Learning
  - Financial Industry
  - Oil Companies
- Brook for GPUs
  - Stanford 2006
  - General purpose GPU language built on OpenGL



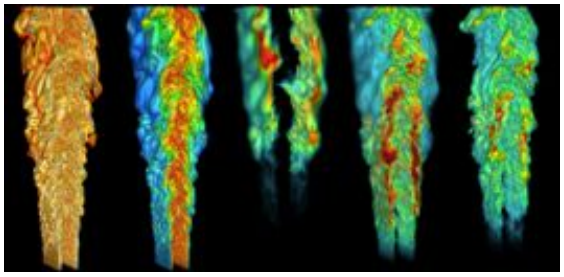
CUDA/OpenCL

- Compute Unified Device Architecture (CUDA)
- Open Compute Language (OpenCL)
- Languages for writing general purpose shaders
- Expose underlying hardware to programmer
- Interoperates with OpenGL

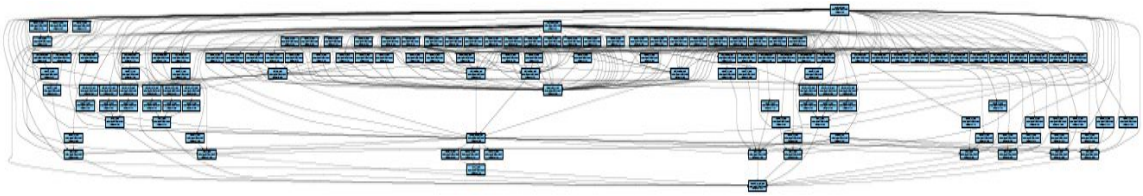


Example: Combustion in S3D

- S3D models combustion
  - Physics of turbulence
  - Reaction chemistry
  - Navier Stokes fluid flow
- Very computationally intense workload

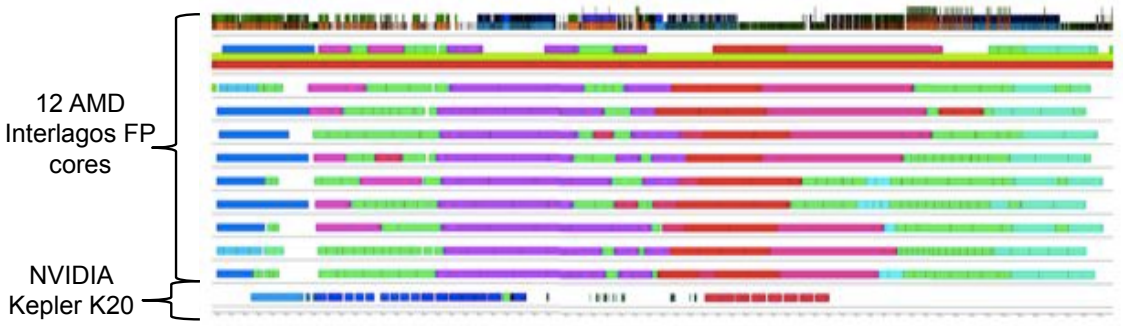


- Dataflow graph for S3
  - Difficult to schedule



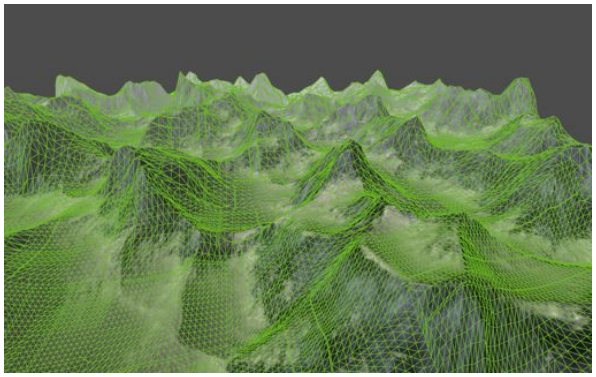
Legion: Programming Heterogeneous Supercomputers

- Use deferred execution
  - Overlap data movement and computation both within a node and between nodes
  - Hide latency, optimize for throughput



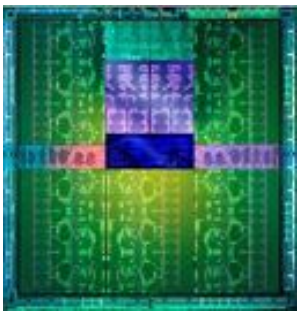
The Future of GPU Architecture

- Real-time ray tracing vs. rasterization
  - Need to trace about 100 billion rays/s, currently ~1 billion rays/s on GPU
  - Rasterization for the immediate future
- Fixed function vs. programmable
  - Tessellation shaders
  - Specialized physics (hair, cloth, fluids)
  - Can it run Crysis?

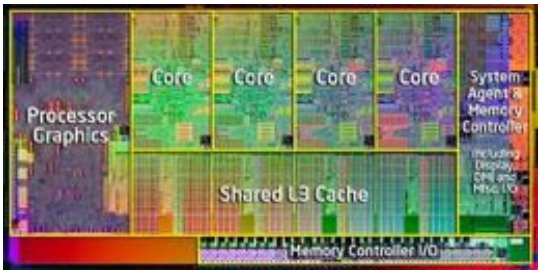


The Future of GPU Architecture

- **Compute vs. Graphics**
  - **Currently share the same silicon die**
  - **More transistors for graphics (AMD/ATI)**
  - **More transistors for compute (NVIDIA, Intel)**



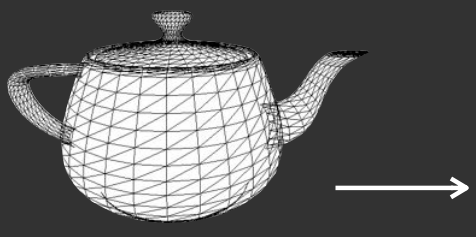
- **Heterogeneity**
  - **Integrate CPUs and GPUs on the same die**
  - **Programming model?**
  - **Performance?**



3D rendering problem:

given: models
description of surface properties
lighting conditions

compute: how each triangle contributes to each image pixel

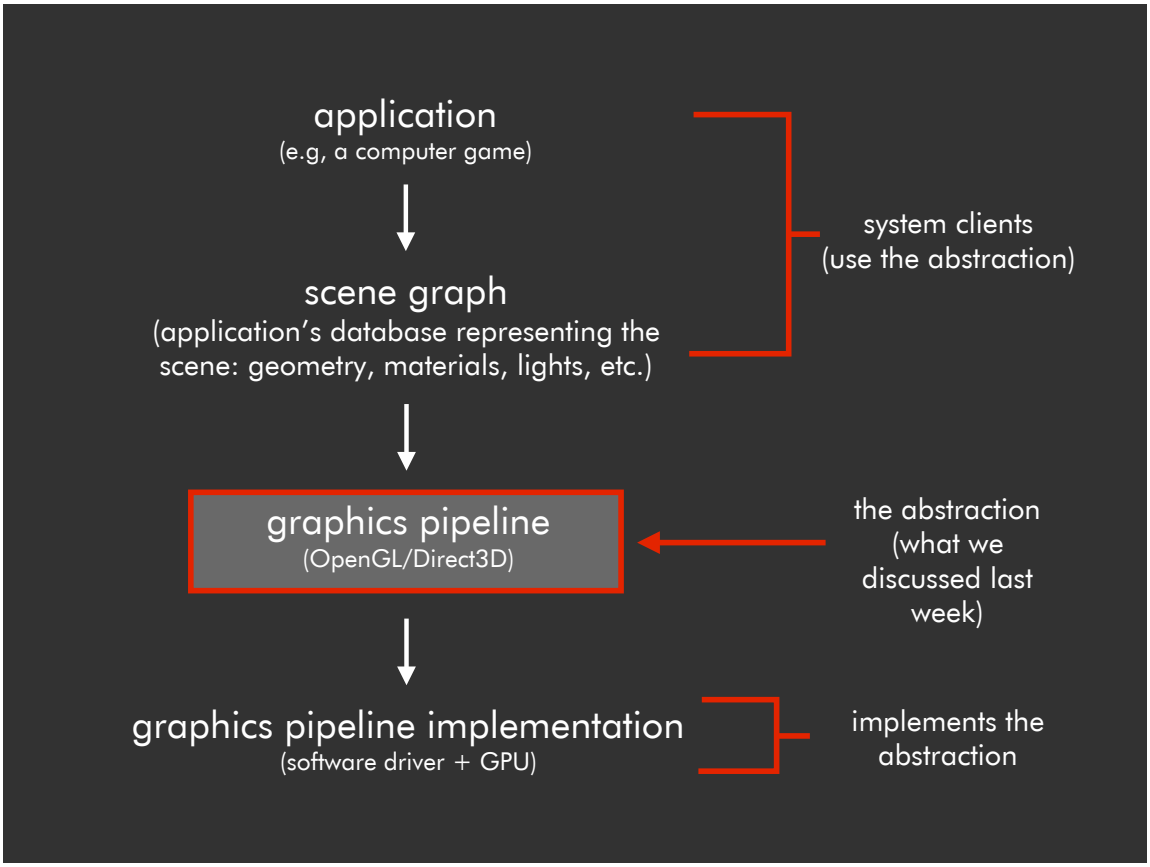


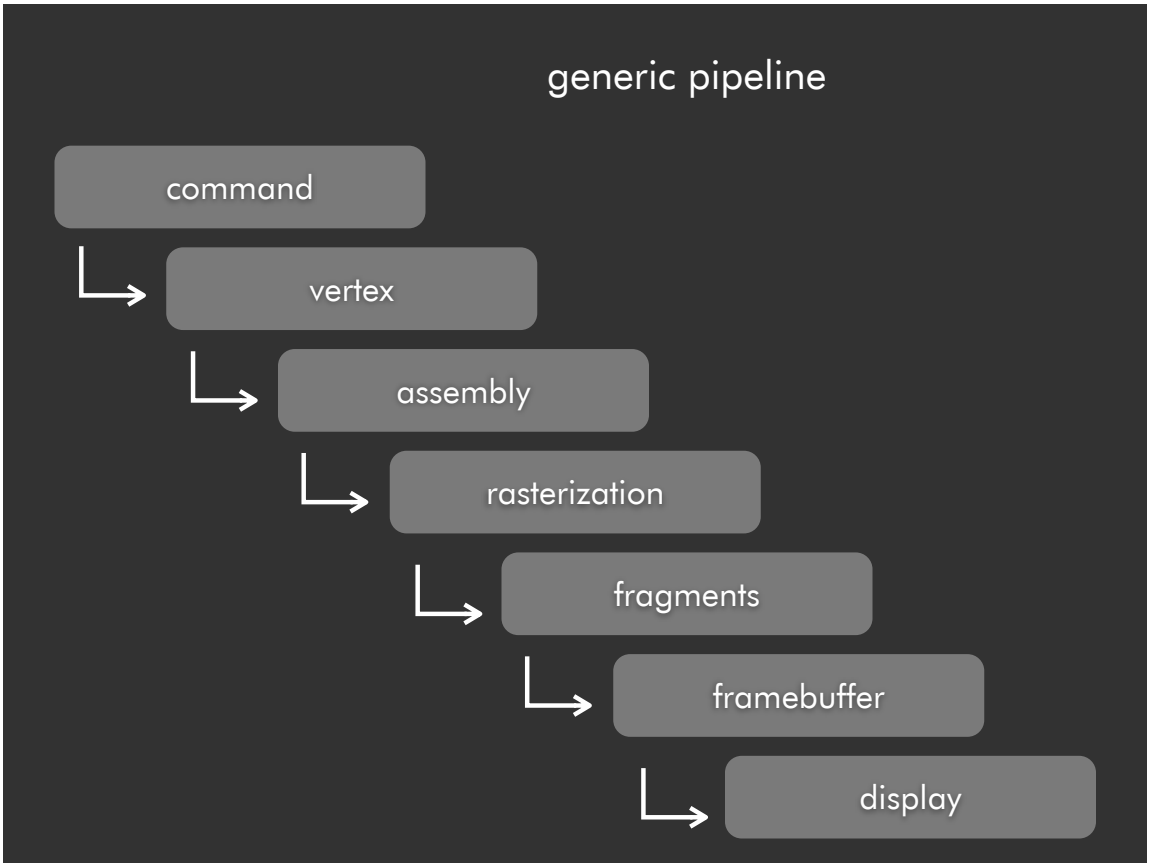
3D surface geometry
(e.g., triangle mesh)
surface materials, lights, camera…

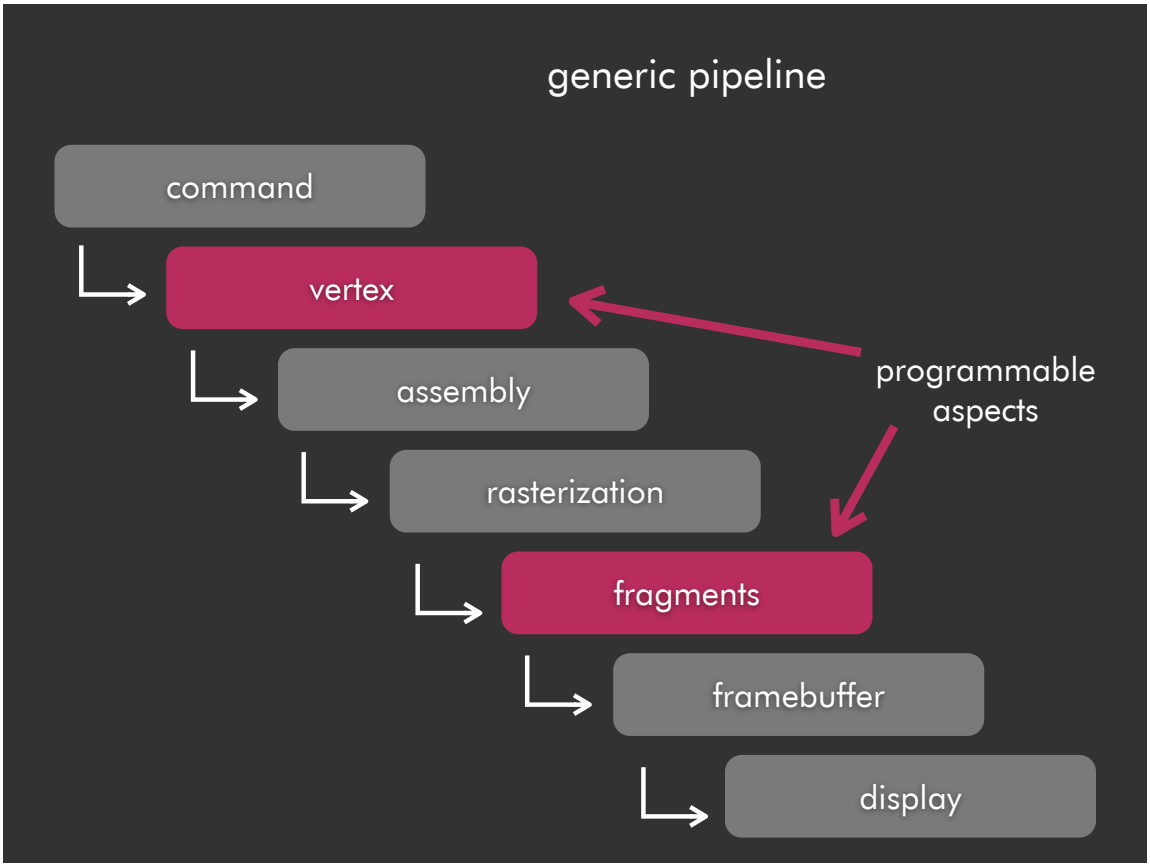
input: model of a scene

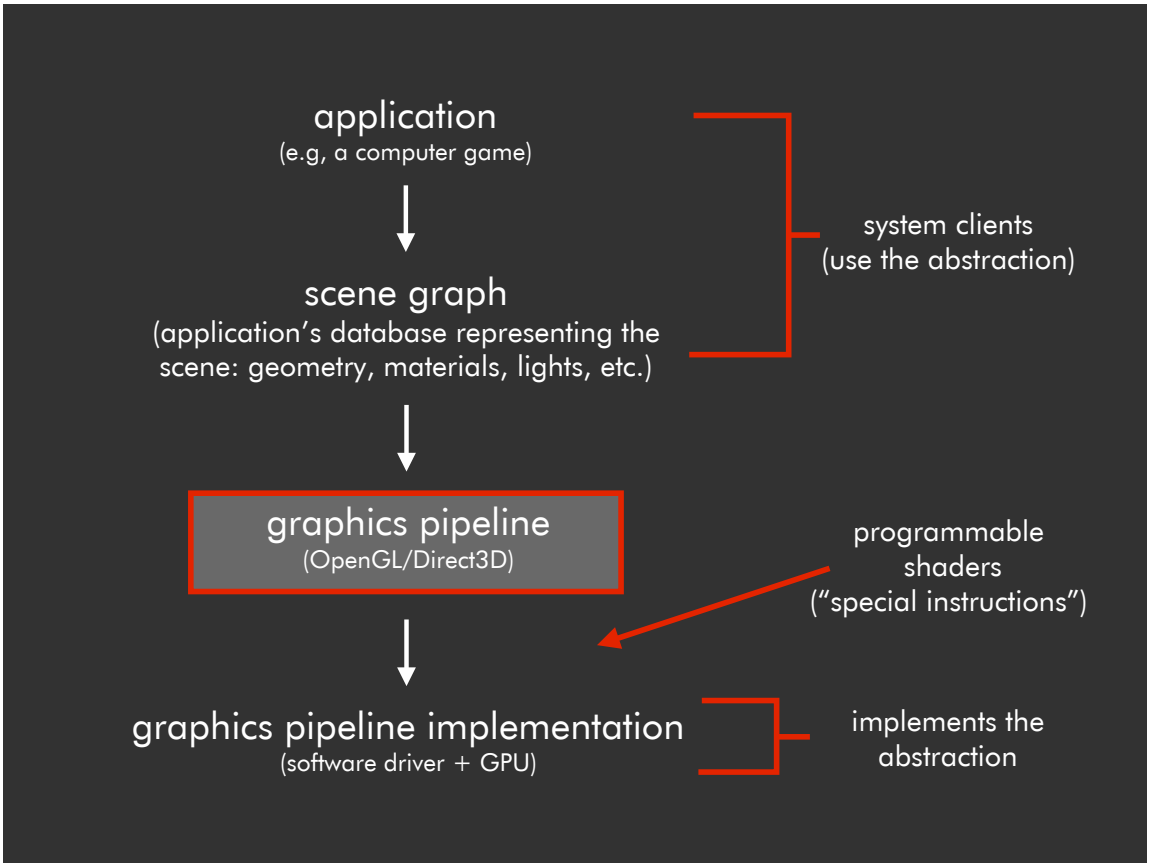
image: henrik wann jensen

output: image


application
(e.g, a computer game)
scene graph
(application's database representing the scene: geometry, materials, lights, etc.)
system clients
(use the abstraction)
graphics pipeline
(OpenGL/Direct3D)
the abstraction
(what we discussed last week)
graphics pipeline implementation
(software driver + GPU)
implements the abstraction


generic pipeline
command
vertex
assembly
rasterization
fragments
framebuffer
display


generic pipeline
command
vertex
assembly
rasterization
fragments
framebuffer
display
programmable aspects

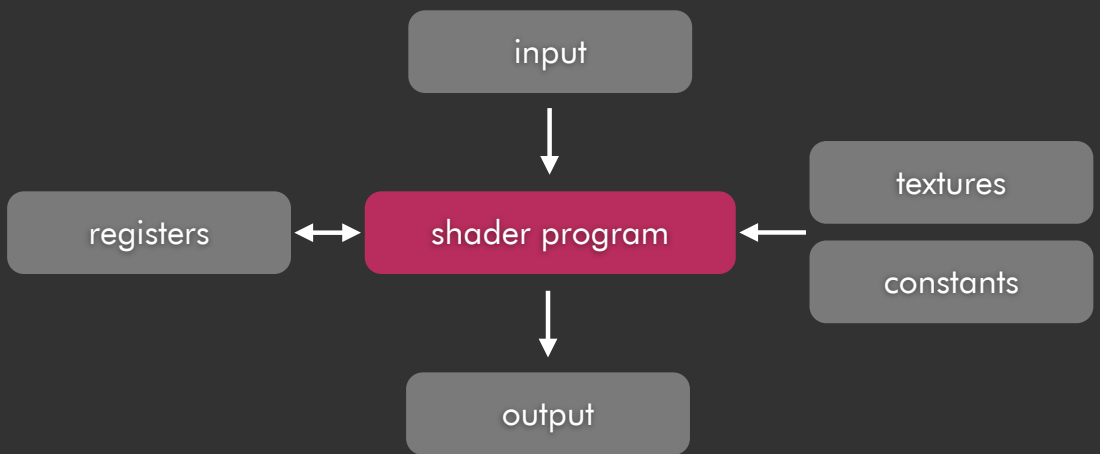

application
(e.g, a computer game)
scene graph
(application's database representing the scene: geometry, materials, lights, etc.)
system clients
(use the abstraction)
graphics pipeline
(OpenGL/Direct3D)
programmable shaders
("special instructions")
graphics pipeline implementation
(software driver + GPU)
implements the abstraction

shaders

short program that customizes
a part of the graphics pipeline

(they can do a lot more than “shade”!)

shaders



shaders

two different types*

command

vertex

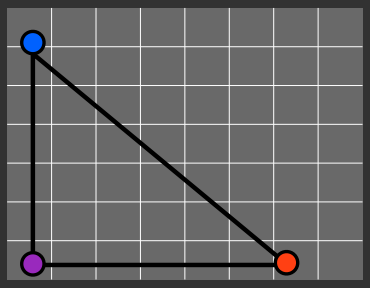
assembly

rasterization

fragments

framebuffer

display



* that we will talk about here

customizes:

color and lighting
texture coordinates
position, transformations
normals

shaders

two different types*

command

vertex

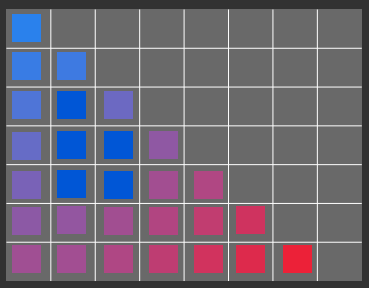
assembly

rasterization

fragments

framebuffer

display

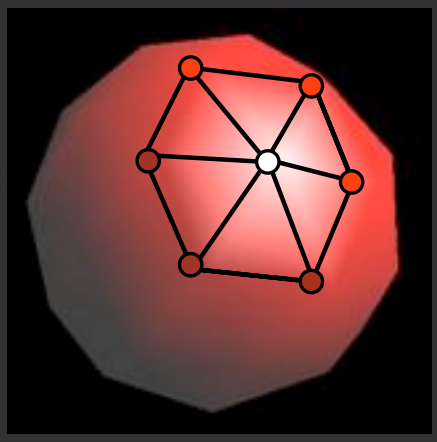


* that we will talk about here

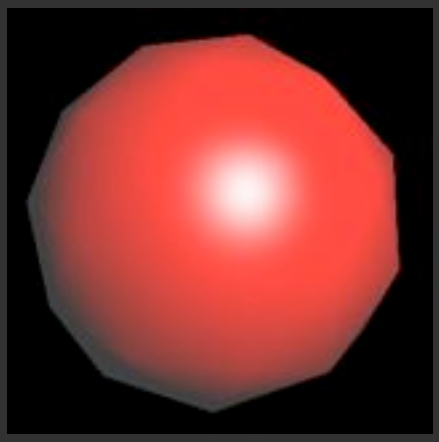
customizes:

visibility and transparency
texturing, shading
composition
color

vertex processing example: lighting



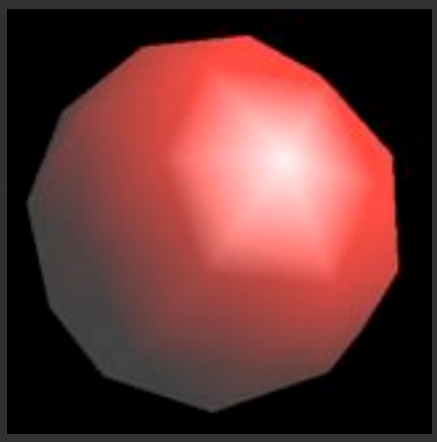
per-vertex lighting computation:
lighting calculated at vertices,
interpolated over fragments



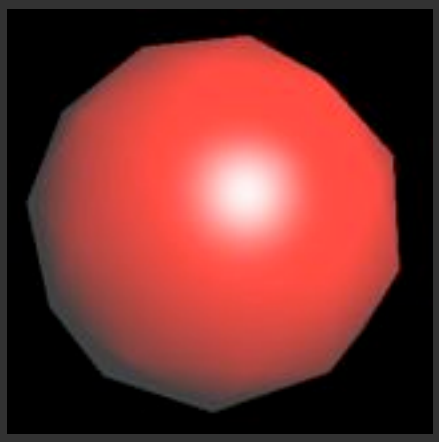
Per-vertex normal computation:
normals interpolated over fragments
used to calculate lighting per fragment

uniform data: light direction, light color
per-vertex data: surface normal, surface color

vertex processing example: lighting



per-vertex lighting computation:
lighting calculated at vertices,
interpolated over fragments



Per-vertex normal computation:
normals interpolated over fragments
used to calculate lighting per fragment

uniform data: light direction, light color
per-vertex data: surface normal, surface color

language options

assembly
- issue commands directly to the GPU

GLSL
- OpenGL shading language
- build right into your existing OpenGL programs!

Cg
- OpenGL/Direct3D

HLSL
- Direct3D

the purpose of a shader is to...

vertex shaders:
- set `gl_Position`

fragment shaders:
- set `gl_FragColor`

... and everything else just aids in calculating these two things.

hello, red!
(example)

GLSL tips

types:

- `uniform` vs. `varying`
- `vec2`, `vec3`, `vec4`, …

many helpful built-in functions:

- `reflect`, `normalize`, … `<>.xyzw` / `<>.rgba` accessors
- `clamp`, `pow`, `sqrt`, `max`, `min`

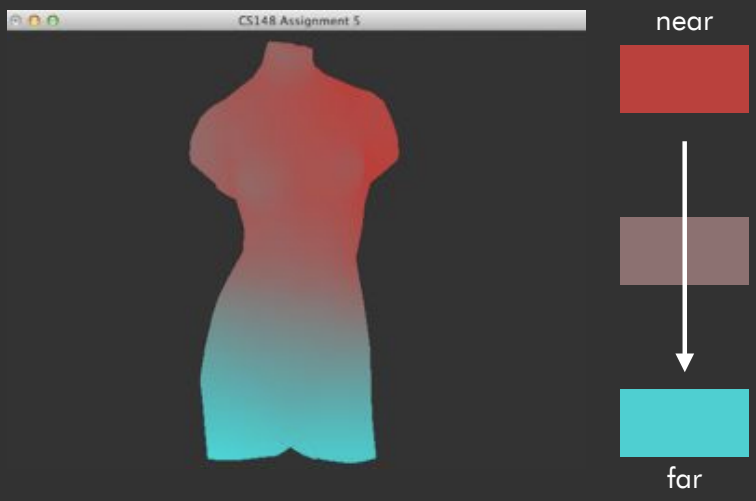
access values from OpenGL:

- via `gl_ThatThingYouWant`
- e.g.: `gl_NormalMatrix`, `gl_LightSource[0].position`, …

similarly, you'll need to set values that OpenGL would normally:

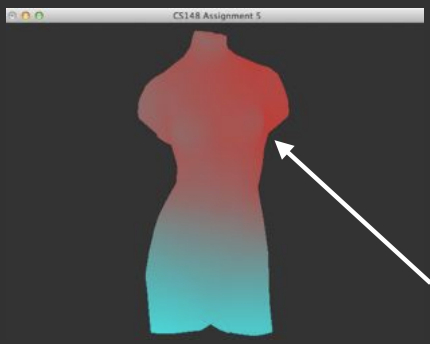
- e.g.: `gl_Position`, `gl_FragColor`

slightly more interesting example



goal:
color fragments according to distance from a point

slightly more interesting example

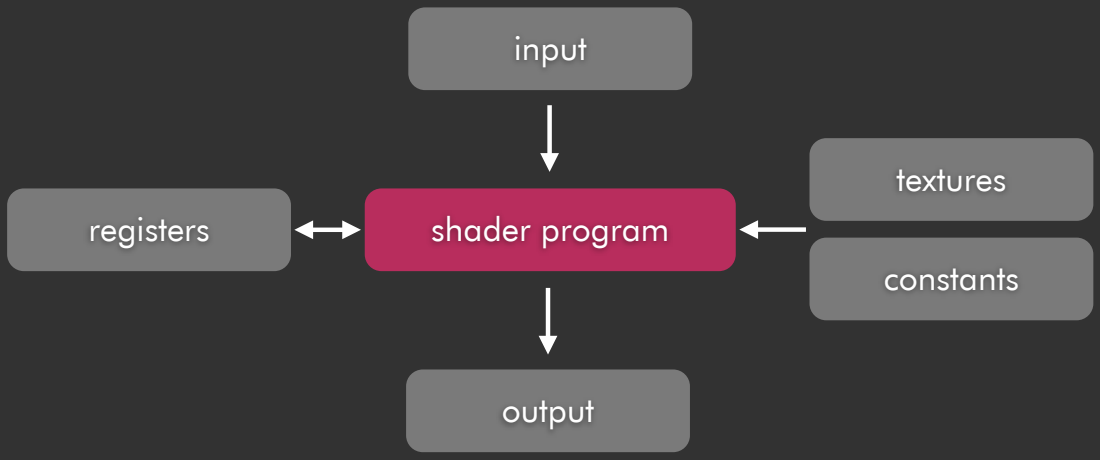


set source vertex (a uniform value):

using `SimpleShaderProgram` (included w/ HW5):

```
SimpleShaderProgram shader;
... // setup shader
shader->Bind();
shader->SetUniform("sourceVertex", x, y, z);
```

shaders



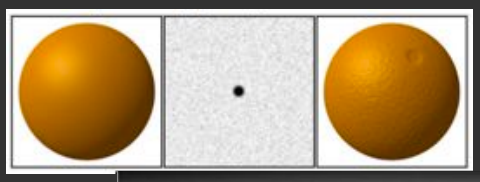
access to pipeline

many uses for textures:
e.g.: provide surface color, reflectance

slide: pat hanrahan

access to pipeline

many uses for textures:
more complex-looking surfaces

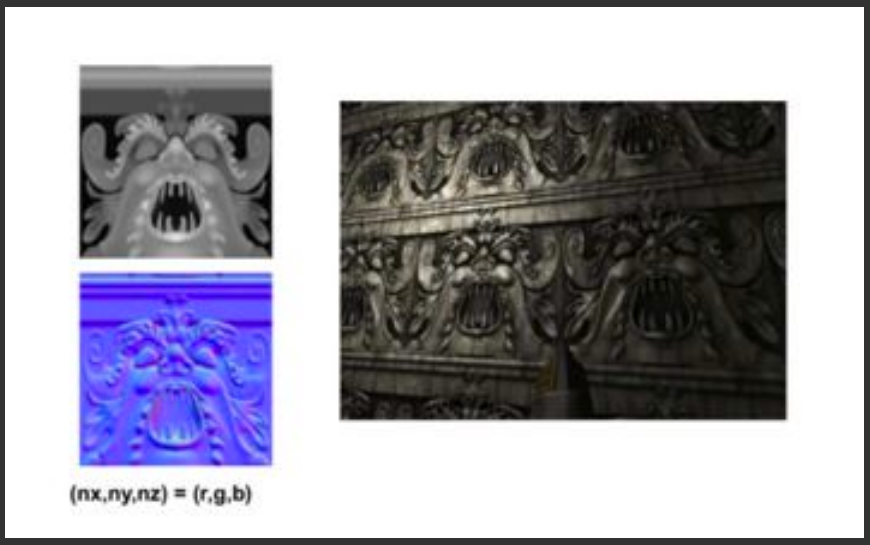


bump mapping:
displace surface in direction
of normal before doing
lighting calculations

slide: kayvon fatahalian, images: wikipedia

access to pipeline

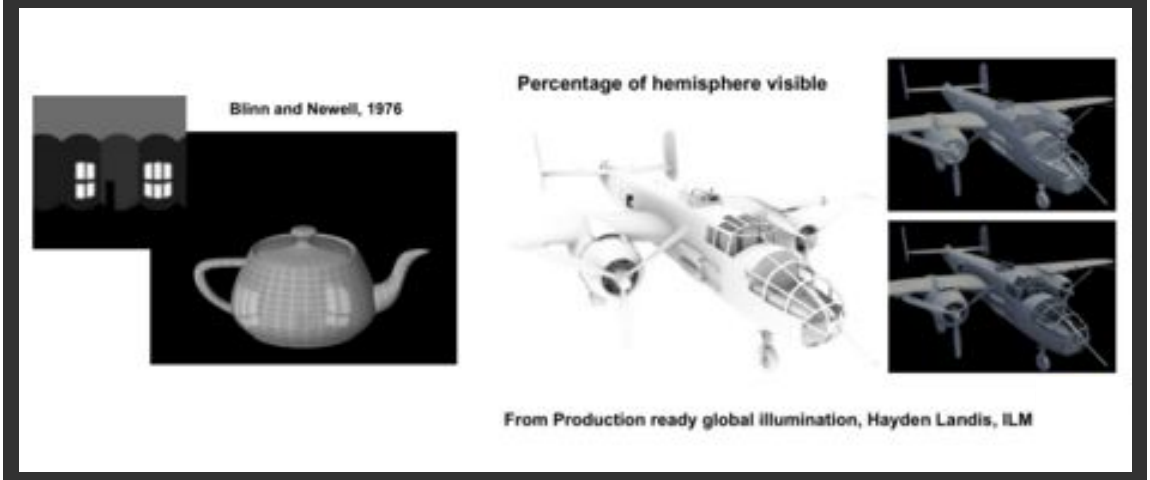
many uses for textures:
normal mapping (modulate the interpolated normal)



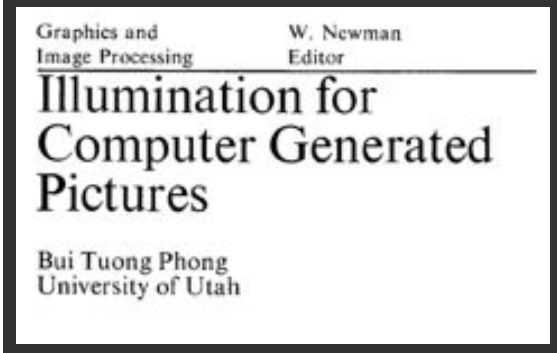
slide: pat hanrahan

access to pipeline

many uses for textures:
store precomputed lighting



slide: pat hanrahan

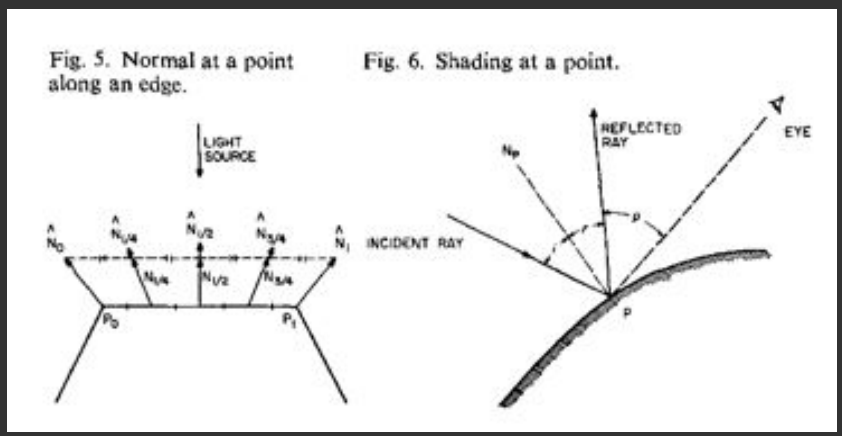

Graphics and Image Processing
W. Newman Editor

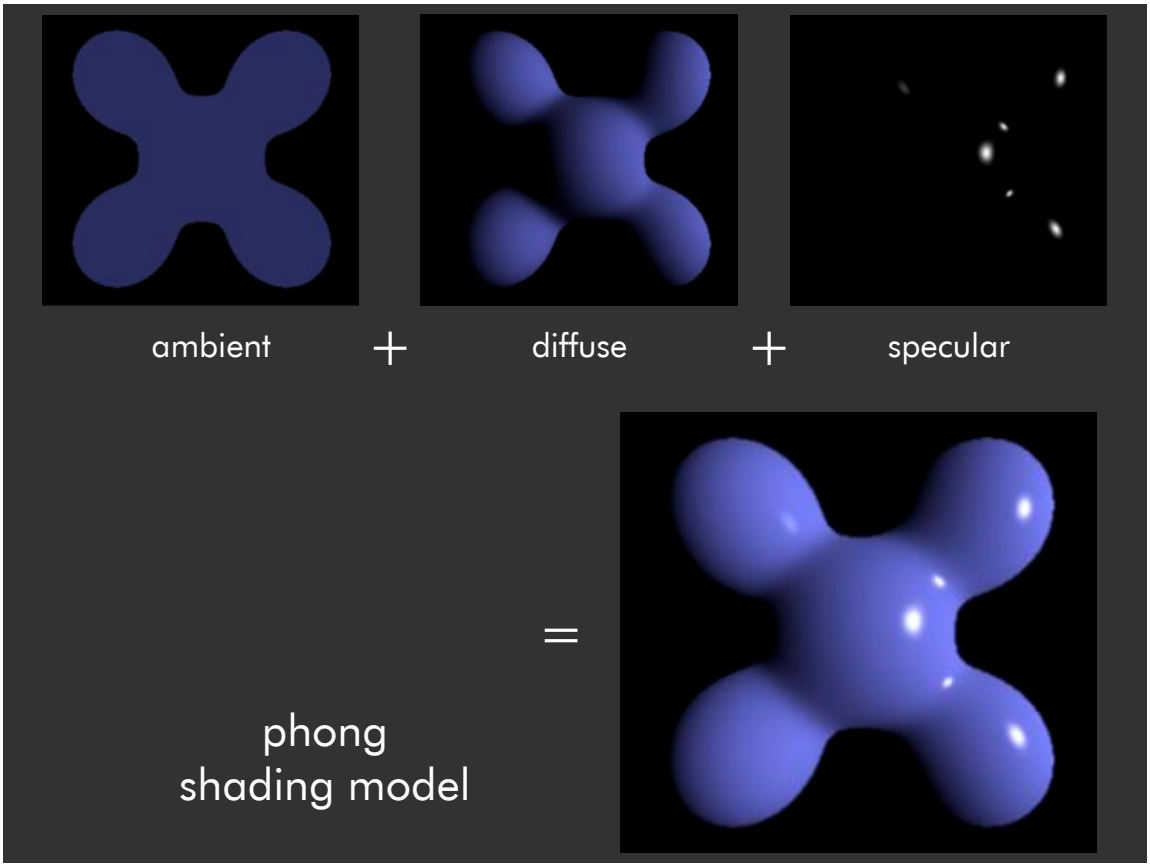
Illumination for Computer Generated Pictures

Bui Tuong Phong
University of Utah

Phong shading model, 1975

specular reflections
normal interpolation




ambient
+
diffuse
+
specular
=
phong
shading model

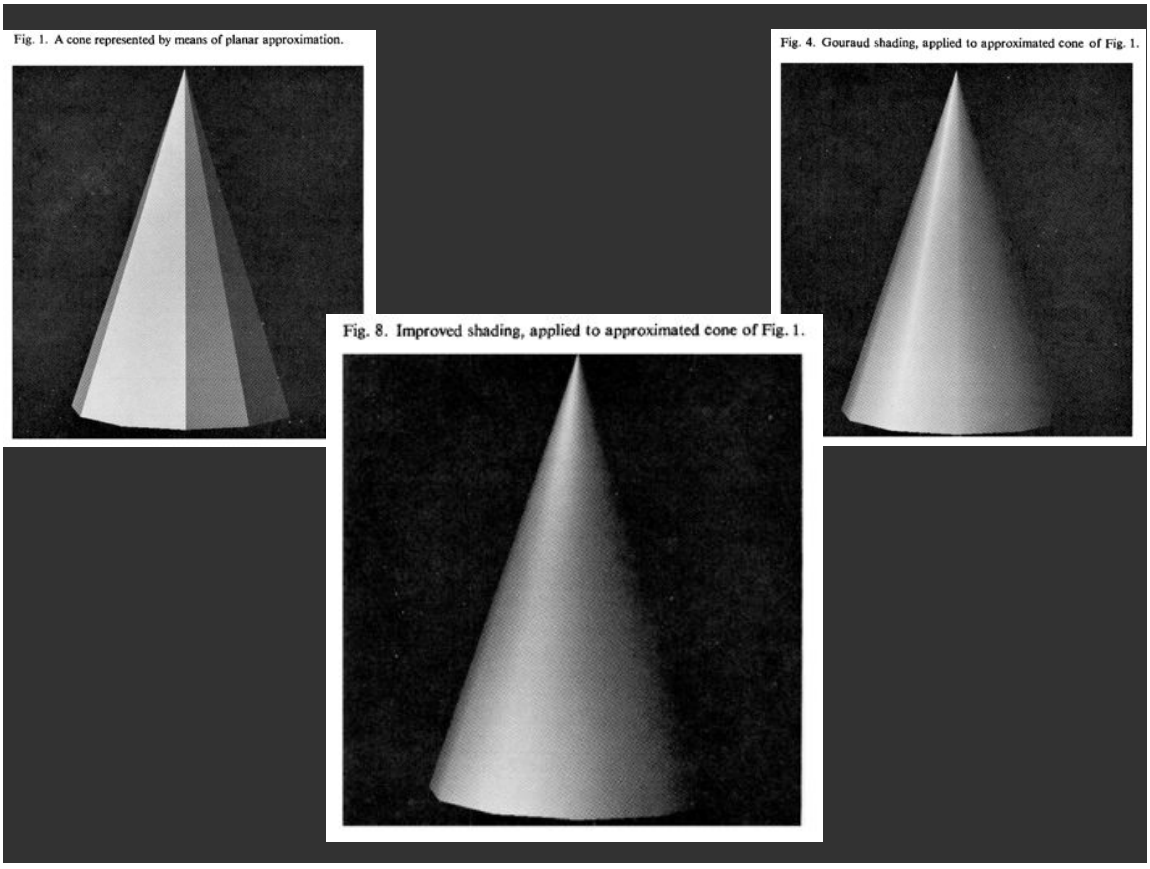
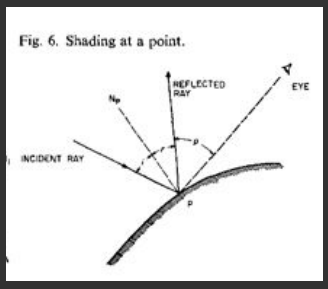


Fig. 1. A cone represented by means of planar approximation.
Fig. 4. Gouraud shading, applied to approximated cone of Fig. 1.
Fig. 8. Improved shading, applied to approximated cone of Fig. 1.

brainstorm!

ambient diffuse specular



what do we need in order to compute phong shading?

brainstorm!

will need to calculate:

vertex position (screen space)
vertex position (world space)
vertex normal
light particulars
material particulars
...

life is hard

single precision arithmetic

branching, loops are expensive

no access to neighboring fragments/vertices!

cannot bind output buffer as input

limited stack/instruction count

timeouts are a possibility

driver support, debugging are inconsistent at best

…be careful out there

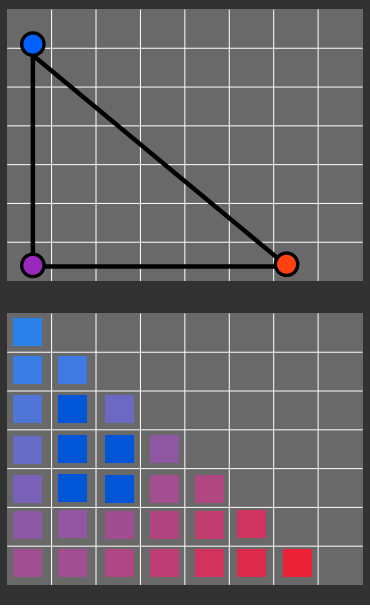
things to remember

all vertices are treated exactly the same

- vertex programs don't know about primitive types!

all primitives are converted into fragments for per-pixel shading

- fragment programs are (mostly) oblivious to the vertex program



things to remember
(systems perspective)

the graphics pipeline is imperative, not declarative

- "draw these triangles, using this fragment shader, with depth testing on"

programmable stages provide flexibility

- e.g., to implement a wide range of materials and lighting techniques

the pipeline itself is configurable, but not programmable

- turn stages on and off, create feedback loops, ...

goldilocks level of abstraction:

- low enough to allow apps to implement many techniques;
- high enough to abstract over radically different GPU implementations

that's all

additional slide credits: justin solomon!