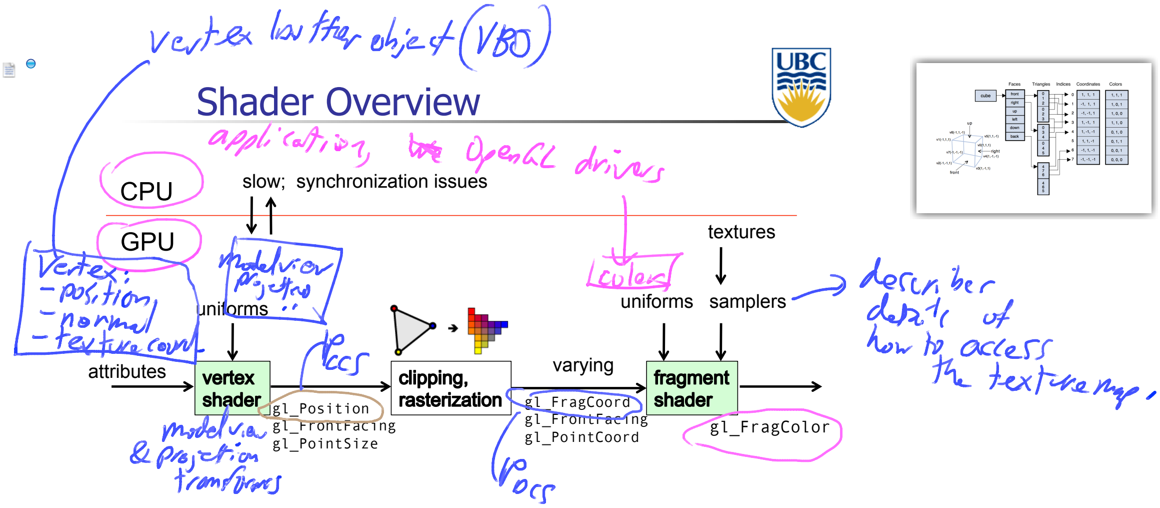
Shader Overview



key hw architecture consideration:
400-600 cycle wait for a memory stall!
Switch tasks while waiting.

complex architecture:
performance tuning can be tricky

Example Vertex Shader

```
attribute vec4 a_Position;
attribute vec4 a_Normal;
attribute vec2 a_TexCoord;

uniform mat4 u_ModelViewMatrix;
uniform mat4 u_ProjectionMatrix;
uniform float u_DistortionTime;
uniform float u_DistortionAmp;

varying vec4 v_ViewPosition;
varying vec4 v_ViewNormal;
varying vec2 v_TexCoord;

void main() {
  vec4 view_pos = u_ModelViewMatrix * a_Position;
  vec4 proj_pos = u_ProjectionMatrix * view_pos;
  // variable attributes, interpolated across triangle, used by fragment shader
  v_ViewPosition = vec4(view_pos.xyz, 1);    // vertex location in VCS
  v_TexCoord = a_TexCoord;                   // u,v texture coordinates
  gl_Position = proj_pos;    // final assigned vertex position (in CCS)
}
```

stored per vertex in VBO

these are interpolated across the triangles

Example Fragment Shader

```
#ifdef GL_ES
    precision mediump float;
#endif

uniform vec4 u_FragColor;
uniform sampler2D u_AlbedoTex;

varying vec4 v_ViewPosition;
varying vec4 v_ViewNormal;
varying vec2 v_TexCoord;

void main() {
    vec2 iResolution = vec2(800,400);
    vec2 pN = gl_FragCoord.xy / iResolution.xy;    // compute fragment coords, in [0,1]
    vec2 pNDCS = pN * 2.0 - 1.0;                   // compute NDCS coords

    vec4 texColour = texture2D(u_AlbedoTex, v_TexCoord);
    gl_FragColor = texColour;
}
```

specify a default color (not used here)

"sampler": pointer to a texture map

interpolated quantities

better: ~~pass~~ use to uniforms instead

texture lookup

DCS

NDCS

sampler i.e. texture

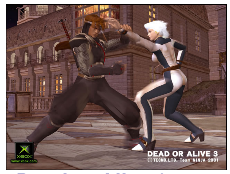
texture coords (u,v)

Computer Graphics:

Hardware Architecture
Software Architecture
Shaders

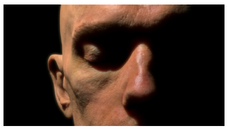
Real Time Graphics

Virtua Fighter 1995
(SEGA Corporation) **NV1**



Dead or Alive 3 2001
(Tecmo Corporation)
Xbox (NV2A)

Nalu 2004
(NVIDIA Corporation)
GeForce 6



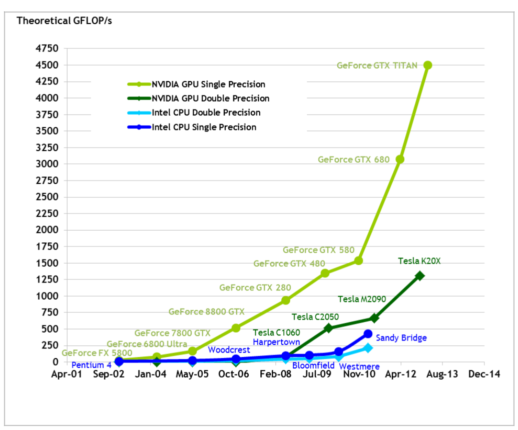
Human Head 2006
(NVIDIA Corporation)
GeForce 7

Medusa 2008
(NVIDIA Corporation)
GeForce GTX 200

Real-Time Dynamic Fracture 2013
(NVIDIA Corporation)
GeForce GTX 700

GPUs vs CPUs

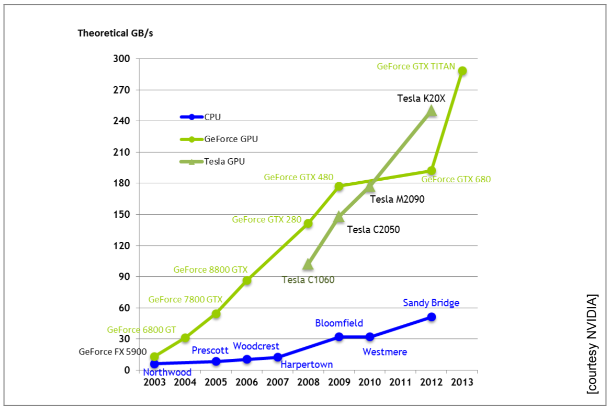
- 4500 GFLOPS vs ~500 GFLOPS



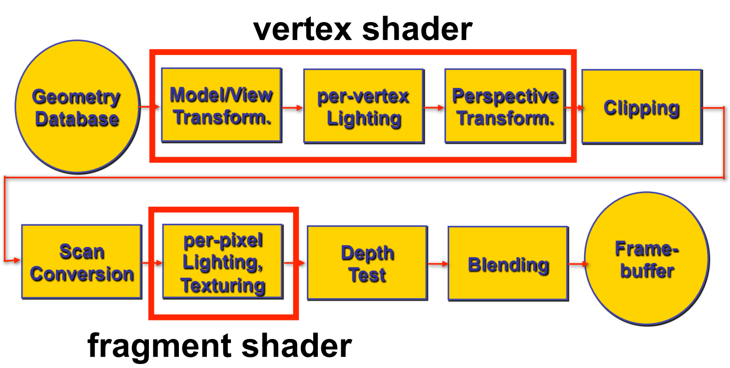
[courtesy NVIDIA]

GPUs vs CPUs

- 290 GB/s vs 60 GB/s



Programmable Pipeline



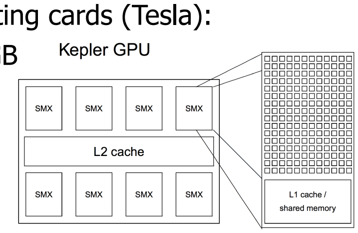
Unified Shader Model

- uses a consistent instruction set for all shader types: geometry, vertex, fragment shaders
- "Unified Shader Architecture allows more flexible use of the graphics rendering hardware. For example, in a situation with a heavy geometry workload the system could allocate most computing units to run vertex and geometry shaders. In cases with less vertex workload and heavy pixel load, more computing units could be allocated to run pixel shaders."

  [http://en.wikipedia.org/wiki/Unified_shader_model]

Nvidia Kepler generation (GeForce 700)

- Consumer graphics cards (GeForce):
  - GTX 770: 1536 cores, 2/4GB
  - GTX Titan: 2688 cores, 6GB
- High Performance Computing cards (Tesla):
- K10: 2×1536 cores, 2×4GB
- K20: 2496 cores, 5GB
- K40: 2880 cores, 12GB



- 8-64 SMX building blocks:

  192 cores, 64k registers, 8k constants, 48k texture cache, up to 2k threads

[https://people.maths.ox.ac.uk/gilesm/cuda/lecs/lec1.pdf]

Vertex Shader

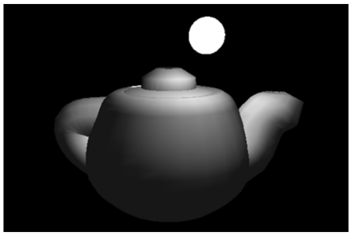
- Run once for every vertex in your scene:
  - Common Functionality:
    - Performs viewing transforms (MVP)
    - Transforms texture coordinates
    - Calculates per-vertex lighting
  - A "vertex" is a malleable definition, you can pass in, and perform pretty much any operation you want

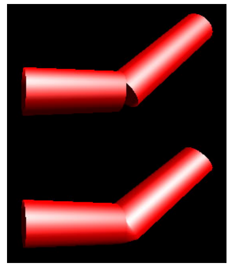
Vertex Shader

- Common Inputs:
  - vertex position
  - Normal texture coordinate(s)
  - Modelview and projection matrix
  - Vertex Material or color
  - Light sources – color, position, direction etc.
- Common Outputs:
  - Clip-space vertex position (mandatory)
  - transformed texture coordinates
  - vertex color

Vertex Shader - Applications

- deformable surfaces – on the fly vertex position computation
  - e.g. skinning





[courtesy NVIDIA]

Fragment Shader

- Runs for all "initialized" fragments:
  - "initialized" → rendered to after rasterization
  - may never appear, i.e., following depth check
  - early depth checks
- Common Tasks:
  - texture mapping
  - per-pixel lighting and shading
- Synonymous with Pixel Shader

Fragment Shader

- input (interpolated over primitives by rasterizer, i.e., `varying`):
  - Fragment coordinates (mandatory)
  - texture coordinates
  - color
- output:
  - fragment color (mandatory)
  - fragment depth

Fragment Shader - Applications

Not really shaders, but very similar to NPR!
A Scanner Darkly, Warner Independent Pictures

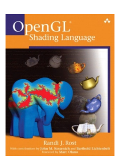
GPU raytracing, NVIDIA

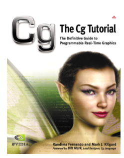
Vertex & Fragment Shader

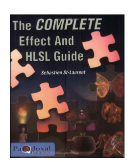
- massively parallel computing by parallelization
- same shader is applied to all data (vertices or fragments) – SIMD (single instruction multiple data)
- parallel programming issues:
  - main advantage: high performance
  - main disadvantage: no access to neighboring vertices/fragments

Shader Languages

- Many languages exist to write shaders:
- GLSL – GL Shading Language (Opengl)
- HLSL – High Level Shading Language (Direct3D)
- CG (Nvidia mid-level language for both)

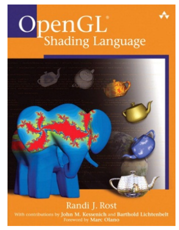






GLSL

- WebGL works with GLSL:
  - C-like programming language for GPUs
  - Highly Parallel (SIMD)
  - Differs greatly between versions



GLSL - Types

- Has all the basic C types
- Has "vector" types: vec2, vec3, vec4
- Has "matrix" types: mat2, mat3, mat4
- Has "sampler" types
  - Used for reading data from textures and framebuffers

GLSL – Samplers

- A type of uniform used to read from a texture within shaders
- There are different samplers for the different types of textures
- 2D textures store square textures
- Rectangle textures store non-square textures

OpenGL Error Checking

- When Things go Wrong:
  - Opengl wont tell you
  - To ask, call glGetError()
    - Tells you the gl state (ok, error, etc)
- WebGL: some information on Console

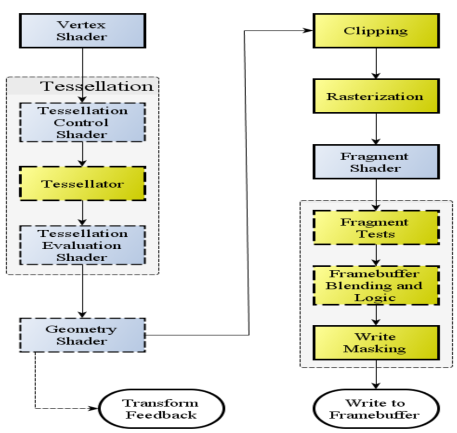
OpenGL the old and the new

GL 1.2 – 2.1	GL 3.0 – 4.4
Vertex Shader	Vertex Shader
Pixel Shader	Tessellation (Control) Shader
	Tessellation Evaluation/Hull Shader
	Geometry Shader
	Fragment Shader
	... Compute Shader

same

new

OpenGL updated graphics pipeline

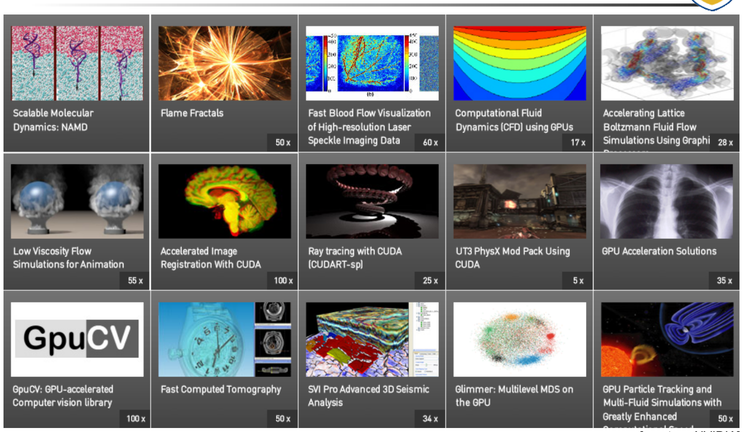


(skip)

OpenGL 3.0+ Advanced Pipeline

- Tesselaton Control shader
  - Synonymous with Tesselation shader (d3d
  - Subdivide geometry based on vertices
- Tesselation Evaluation
  - Synonymous with Hull shader (d3d)
  - Rearrange new vertices from tesselation control

GPGPU Applications



[courtesy NVIDIA]

References and Resources

- http://www.opengl.org/wiki/Uniform_%28GLSL%29
- http://www.lighthouse3d.com/tutorials/glsl-tutorial/uniform-variables/
- http://www.opengl.org/wiki/Rendering_Pipeline_Overview
- http://www.davidcornette.com/glsl/glsl.html
- http://nehe.gamedev.net/article/glsl%20an%20introduction/25007/
- http://www.opengl.org/wiki/Data_Type_%28GLSL%29
- http://www.opengl.org/wiki/Sampler_%28GLSL%29#Sampler_types
- http://zach.in.tu-clausthal.de/teaching/cg_literatur/glsl_tutorial/