Shader Overview

CPU

GPU

slow; synchronization issues

textures

uniforms

samplers

attributes

v_TexCoord = a_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_TexCoord = a_TexCoord;
// u,v texture coordinates
gl_Position = proj_pos; // final assigned vertex position (in CCS) }

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;

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_ViewNormal = u_Normal; // normal vector
v_TexCoord = a_TexCoord;
// u,v texture coordinates
gl_Position = proj_pos; // final assigned vertex position (in CCS)
Example Fragment Shader

```cpp
#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;
}
```

Computer Graphics:

Hardware Architecture
Software Architecture
Shaders
GPUs vs CPUs

- 4500 GFLOPS vs ~500 GFLOPS

Theoretical GFLOPs

Graph showing performance comparison between GPUs and CPUs.
GPUs vs CPUs

- 290 GB/s vs 60 GB/s

Programmable Pipeline

**vertex shader**
- Geometry Database
- Model/View Transform
- per-vertex Lighting
- Perspective Transform
- Clipping

**fragment shader**
- Scan Conversion
- per-pixel Lighting, Texturing
- Depth Test
- Blending
- Frame-buffer
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."


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/ecs/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

**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., \textit{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
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

Vertex & Fragment Shader

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 won’t 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

<table>
<thead>
<tr>
<th>GL 1.2 – 2.1</th>
<th>GL 3.0 – 4.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Shader</td>
<td>Vertex Shader</td>
</tr>
<tr>
<td>Pixel Shader</td>
<td>Tessellation (Control) Shader</td>
</tr>
<tr>
<td>Tessellation Evaluation/Hull Shader</td>
<td>Geometry Shader</td>
</tr>
<tr>
<td>Geometry Shader</td>
<td></td>
</tr>
<tr>
<td>Fragment Shader</td>
<td></td>
</tr>
<tr>
<td>... Compute Shader</td>
<td></td>
</tr>
</tbody>
</table>

### OpenGL updated graphics pipeline

[Diagram of the updated graphics pipeline]

- **Vertex Shader**
- **Tessellation Shaders** (Control, Evaluation, Analysis)
- **Geometry Shader**
- **Fragment Shader**
- **Compute Shader**
- **Chipping**
- **Rasterization**
- **Fragment Shader**
- **Fragment Texturing/Blending**
- **Write to Framebuffer**
- **Write to Memory**
- **Transform Feedback**
Tessellaton Control shader
  - Synonymous with Tessellation shader (d3d)
  - Subdivide geometry based on vertices

Tessellation Evaluation
  - Synonymous with Hull shader (d3d)
  - Rearrange new vertices from tesselation control

GPGPU Applications

[courtesy NVIDIA]
References and Resources

- http://www.opengl.org/wiki/Rendering_Pipeline_Overview
- http://nehe.gamedev.net/article/gls%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/