Programmable GPUs

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

- 290 GB/s vs 60 GB/s
so far:
- rendering pipeline = set of stages with **fixed functionality**

Programmable Pipeline

- now: programmable rendering pipeline!

vertex shader

fragment shader
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

![Image of teapot and L-shaped object](courtesy NVIDIA)

**Fragment Shader**

- Runs for all “initialized” fragments:
  - “initialized” → rendered to after rasterization

- Common Tasks:
  - texture mapping
  - Shading

- Synonymous with Pixel Shader
Fragment Shader

- input (interpolated over primitives by rasterizer):
  - 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

Many languages exist to write shaders:

- GLSL – GL Shading Language (OpenGL)
- HLSL – High Level Shading Language (Direct3D)
- CG (Nvidia mid-level language for both)
We are using 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
  - (might be worthwhile looking into for Assignment 4)

Look at these links for more info:
- http://www.opengl.org/wiki/Data_Type_%28GLSL%29
- http://www.opengl.org/wiki/Sampler_%28GLSL%29#Sampler_types

GLSL – Built in Variables

- GLSL has some variables built in
  - These variables are always there and accessible in the corresponding shader

- Vertex Shader
  - In: gl_Vertex (position), gl_Normal, gl_Color
  - Out: gl_Position

- Fragment Shader
  - In: glFragCoord (fragment location), gl_Color
  - Out: gl_FragColor, gl_FragDepth
GLSL – Built in Variables

- Accessible in all shaders:
  - gl_ModelViewMatrix
  - gl_ModelViewProjectionMatrix
  - gl_ProjectionMatrix

- Here is a quick reference guide:

GLSL Example – Vertex Shader

- Vertex Shader: scale vertices

```glsl
#version 200

void main( )
{
    // scale passed in vertex
    vec4 a = gl_Vertex;
    a.x = a.x * 1.5;
    a.y = a.y * 1.5;

    // transform vertex
    gl_Position = gl_ModelViewProjectionMatrix * a;
}
```
GLSL Example – Fragment Shader

- Fragment Shader: color green

```glsl
#version 200

void main( )
{
    // color rendered fragments green
    gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

GLSL – Uniform Variables

- Used to access data from the CPU on the GPU
- Need to be given a value from the openGL side
GLSL Example – Uniform Variables

GLint loc1, loc2, loc3, loc4;
float specIntensity = 0.98;
float sc[4] = {0.8, 0.8, 0.8, 1.0};
float threshold[2] = {0.5, 0.25};
float colors[12] = {0.4, 0.4, 0.8, 1.0,
                    0.2, 0.2, 0.4, 1.0,
                    0.1, 0.1, 0.1, 1.0};

loc1 = glGetUniformLocation(p, “specIntensity”);
glUniformf(loc1, specIntensity);

loc2 = glGetUniformLocation(p, “specColor”);
glUniform4fv(loc2, 1, sc);

loc3 = glGetUniformLocation(p, “t”);
glUniform1fv(loc3, 2, threshold);

loc4 = glGetUniformLocation(p, “colors”);
glUniform4fv(loc4, 3, colors);

http://www.lighthouse3d.com/tutorials/glsl-tutorial/uniform-variables/

- Within shader:

```glsl
#version 200

uniform float specIntensity;
uniform vec4 specColor;
uniform float t;
uniform vec4 colors;

void main()
{
    // do something
}
```
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, such as the image being processed in A4

OpenGL Error Checking

- When Things go Wrong:
  - OpenGL wont tell you
  - To ask, call glGetError()
    - Tells you the gl state (ok, error, etc)
  - For A4, this is all done for you, but you will need to break before the end of the program to read the output (in the black terminal)
### 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></td>
<td>Tessellation Evaluation/Hull Shader</td>
</tr>
<tr>
<td></td>
<td>Geometry Shader</td>
</tr>
<tr>
<td></td>
<td>Fragment Shader</td>
</tr>
<tr>
<td></td>
<td>... Compute Shader</td>
</tr>
</tbody>
</table>

### OpenGL updated graphics pipeline
OpenGL 3.0+ changes

- Removed many of the GLSL built in variables
- Removed GLSL/OpenGL built in matrices
- Removed glVertex(), glColor, glTexCoord, glMaterial(), ...

Why?

- Efficiency
  - in most cases you don't need everything
  - lots of computation wasted checking what applies
- Control
  - with less dictation, shaders can be used to do more
OpenGL 3.0+ Advanced Pipeline

- Tesselation 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

OpenGL 3.0+ Advanced Pipeline

- Geometry Shaders
  - Perform operations on groups of vertices

- Compute Shaders
  - Use the GPU to do math for you (no rendering)
  - This executes after the geometry shader, replacing the rest of the pipeline
## GPGPU Applications

<table>
<thead>
<tr>
<th>Scalable Molecular Dynamics: NAMD</th>
<th>Flame Fractals</th>
<th>Fast Blood Flow Visualization of High-resolution Laser Speckle Imaging Data</th>
<th>Computational Fluid Dynamics (CFD) using GPUs</th>
<th>Accelerating Lattice Boltzmann Fluid Flow Simulations using GPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="image1">Image</a></td>
<td><a href="image2">Image</a></td>
<td><a href="image3">Image</a></td>
<td><a href="image4">Image</a></td>
<td><a href="image5">Image</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Law-Viscosity Flow Simulations for CFD</th>
<th>Accelerated Image Registration with CUDA</th>
<th>Ray Tracing with CUDA (CUDAFT-3D)</th>
<th>U73 PhytoMol Pack Using CUDA</th>
<th>GPU Acceleration Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="image6">Image</a></td>
<td><a href="image7">Image</a></td>
<td><a href="image8">Image</a></td>
<td><a href="image9">Image</a></td>
<td><a href="image10">Image</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GpuCV: GPU Accelerated Computer Vision Library</th>
<th>Fast Computed Tomography</th>
<th>OM Pro-Advanced 3D Seismic Analysis</th>
<th>KL immunity: Multires MDS on the GPU</th>
<th>GPU Particle Tracking and Multi-Fluid Simulations with Orslev Enhanced</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="image11">Image</a></td>
<td><a href="image12">Image</a></td>
<td><a href="image13">Image</a></td>
<td><a href="image14">Image</a></td>
<td><a href="image15">Image</a></td>
</tr>
</tbody>
</table>

[Courtesy NVIDIA]

---

## References and Resources

- [http://www.davidcornette.com/glsl/glsl.html](http://www.davidcornette.com/glsl/glsl.html)
- [http://nehe.gamedev.net/article/glsl%20an%20introduction/25007/](http://nehe.gamedev.net/article/glsl%20an%20introduction/25007/)
- [http://www.opengl.org/wiki/Data_Type_%28GLSL%29](http://www.opengl.org/wiki/Data_Type_%28GLSL%29)
- [http://www.opengl.org/wiki/Sampler_%28GLSL%29#Sampler_types](http://www.opengl.org/wiki/Sampler_%28GLSL%29#Sampler_types)
- [http://zach.in.tu-clausthal.de/teaching/cg_literatur/glsl_tutorial/](http://zach.in.tu-clausthal.de/teaching/cg_literatur/glsl_tutorial/)