The GPU Pipeline

Programmable Hardware
The GPU

- Graphics Processing Unit
- Designed to produce real-time graphics and effects
- GPUs today are the strongest parallel processor

![Graph showing performance of GPUs over time]
NVIDIA 580GTX

- State-of-the-art GPU
- Characteristics:
  - Processor cores: 512(!)
  - Computation speed: 1581 GFLOPS
    - 3.33 GHz Intel 6-Core – 100 GFLOPS peak
  - Texture Fill Rate: 50 Billion Pixel/Sec
- OpenGL Version: 4.1
- DX11
GPU Pipeline
Programmable Hardware

- Originally GPU had only fixed operations
  - E.g.: Support OpenGL 1.2 API functionality
- Modern GPUs allow modifications
  - Flexibility (programmable vs. fixed)
  - Fast implementation of complex algorithms
  - GP-GPU (GPU general computing)
    - OpenCL / CUDA
- Programmable parts can run programs (Shaders):
  - Vertex
  - Fragment (Pixel)
  - Geometry (introduced few years ago)
Programmable Pipeline

\[
\begin{align*}
(x, y, z, w) & \quad (x', y', z', w') \\
(nx, ny, nz) & \quad (nx', ny', nz') \\
(s, t, r, q) & \quad (s', t', r', q') \\
(r, g, b, a) & \quad (r', g', b', a')
\end{align*}
\]
The Vertex Processor

- Responsible for the following operations:
  - Vertex transformation
  - Normal transformation and normalization
  - Texture Coordinates Generation and transformation
  - Lighting (per vertex)
  - Color & Material (per vertex)

- Shader replaces ALL functionality of the fixed processor
  - When shader is active the fixed processing is disabled
  - Shader must support all the above in its code

- Vertex Shaders runs in parallel (almost) on all vertices
  - No vertex information can affect another vertex.
The Fragment Processor

- Operates on the fragments AFTER the interpolation and rasterization of vertex data.

- Allows the following programming:
  - Operation on interpolated vertex values (phong, anyone?)
  - Texture access & application
  - Fog
  - Color sum etc.

- All fragment shaders work in parallel
  - Again, no access to neighboring fragments.
  - Does not replace back-end operations
    - alpha blending, pixel ownership tests, masking and such.
Shading Languages

- **Low level languages - a GPU assembler**
  - ARB (OpenGL), PTX (Cuda)

- **High level languages – C-like**
  - Cg (NVidia)
  - GLSL *(OpenGL)* – *we will use this one*
  - HLSL *(DirectX)*

- **OpenGL and DirectX support API for**
  - Loading shaders
  - Feeding them with user-defined data
  - Deleting them.

Open GL - Center for Graphics and Geometric Computing, Technion
GLSL

- A high-level procedural languages (similar to C++).
- As of OpenGL 2.0, it is a standard, with a simple glXYZ API from OpenGL applications
  - Former implementations of GLSL used OpenGL extensions
- The same language used for both types of shaders
  - There are some small differences
- Native to 3D graphics usage
  - Supports types for position, color, matrix, etc…
Initial Example – RGB Cube

- **Vertex Shader:**

```cpp
varying float xpos;
varying float ypos;
varying float zpos;
void main(void)
{
    xpos = clamp(gl_Vertex.x, 0.0, 1.0);
    ypos = clamp(gl_Vertex.y, 0.0, 1.0);
    zpos = clamp(gl_Vertex.z, 0.0, 1.0);
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

From http://www.clockworkcoders.com/oglsl
RGB Cube – Cont’d

- Fragment Shader:

```c
varying float xpos;
varying float ypos;
varying float zpos;
void main (void) {
    gl_FragColor = vec4 (xpos, ypos, zpos, 1.0);
}
```
GLSL Definitions

- **Data Types:**
  - Scalar – float, int, bool
  - Vectors – (i|b|nothing)vec2|3|4
  - Matrices – (i|b|nothing)mat2|3|4

- **Texture (sample) types:**
  - Regular texture – Sampler1/2/3D
  - Texture with depth – Sampler1|2|3DShadow

- **Structures, main() function, arrays** - as in C
  - NO implicit type casting, however!
Qualifiers and Attributes

- Serve as I/O for a shader from the application and between the shaders
  - **Attribute** – For frequently changing information, from the application to a vertex shader
  - **Uniform** - For infrequently changing information, from the application to either shader.
  - **Varying** – For interpolated information passed on from the vertex shader to the fragment shader.
  - **Const** – Compile-time constants (as in C).
Inside a Vertex Shader

- **Shader input**
  - Supplied by the user through the glVertex() or similar
    - gl_Vertex – vertex position
    - gl_Normal – vertex normal
    - gl_TexCoord – vertex texture mapping coordinates
    - ...
  - gl_ModelViewMatrix - current model view matrix
  - gl_ProjectionMatrix - current projection matrix
  - gl_LightSource[2].spotDirection
  - ...

- **Must output gl_Position (the position after transformation, in camera space coordinates).**
Writing A Fragment Shader

- Given data from the application and the Vertex shader:
  - gl_Color – interpolated color
  - gl_TexCoord

- Output the following variables
  - gl_FragColor
  - gl_FragDepth – can be overridden. Default is the fragment’s depth.
  - Gl_FragData – can be used for other buffers, to be accessed by the program with glGet() functions.
OpenGL shader API

- In order for a shader to be programmed unto the hardware, the following steps must be taken:
  - Create a shader object with `glCreateShader()`
  - Provide source code of shader (a text, in sets of strings) with `glShaderSource()`
  - Compile (just like in C!) with `glCompileShader()`
  - Create program object with `glCreateProgram()`
  - Attach shader object to program with `glAttachShader()`
  - Link (again, like in C) program with `glLinkProgram()`
  - Install program to OpenGL with `glUseProgram()`
  - Passing Attributes through `glUniform()`, `glVertexAttrib()`
GLSL Links and Tools

- **GLSL Tutorials**

- **GLSL shader tools**
  - AMD rendermonkey (end of life, but does the work)
  - ShaderDesigner – quite old shader designer (WinXP only!)
    - See link in HW assignment section
  - GLSLDevil - GLSL debugger (very unstable)
    - [http://www.vis.uni-stuttgart.de/glsldevil](http://www.vis.uni-stuttgart.de/glsldevil)
Examples

Toon Teapot

Glass Klein-Bottle
More Examples

http://www.bencloward.com/images/shaders_offset_complete.jpg
Yet More Examples

Ray Tracing, Courtesy: Martin Christen University of Basel

Deformations

Geometric Textures

OpenGL - Center for Graphics and Geometric Computing, Technion