GPU Programming Intro

Lecture 22/01/2015
Why Use Programmable GPU?
Computer Graphics

Geometry

Transforms + Viewing

Projection + Clipping + Screen-mapping

Colour
FF Shading

Fixed function OpenGL pipeline provides facility for two types of shading:

```plaintext
glShadeModel(GL_FLAT)  
glShadeModel(GL_SMOOTH)
```
**Flat Shading:**

For every polygon
- Calculate normal \( n \), light \( l \)
- Rasterize polygon
  - For all pixels \( p \)
  - Scale color \( c \) by \( n \cdot l \) to get \( c_L \)

**Gouraud Shading:**

For every polygon
- Calculate face_normal \( n \);
- For each vertex, \( i \)
  - Calculate normal (average normals of polygons at this vertex) \( n[i] \)
  - Calculate light \( l[i] \)
  - Scale color \( c \) by \( n[i] \cdot l[i] \) to get \( c_L[i] \)
- Rasterize polygons
  - For all pixels
  - Interpolate between \( c_L[i] \)'s

vertex normal in Gouraud can be calculated by averaging face_normals of adjacent faces
Phong Shading

For every polygon
  Calculate normal \( n \),
  For each vertex, \( i \)
    Calculate normal (average normals of polygons at this vertex) \( n[i] \)
Rasterize polygon
  For all pixels, \( p \)
    Interpolate between \( n[i] \) to get \( n[p] \), calculate light \( l[p] \)
    Scale color \( c \) by \( n[p].l[p] \) to get \( c_L[p] \)

N.B. Above examples showing Phong specular effects (code not shown here)
Per-vertex v.s. Per Pixel Lighting

Advanced Illumination Example

Final Fantasy: The Spirits Within, 2001
(Off-line Rendered)
© Square Pictures and Columbia

Final Fantasy XIII, 2009
(In-game Screenshot)
© Square Enix
Advanced Illumination Example

Advanced Illumination Example

- Subsurface scattering
  - Most (non-metal) partially translucent
  - Some light is diffused under the surface
  - In interactive apps: apply a multi-pass technique to account for scattered light
Advanced Illumination Example

Stylistic Rendering Example

- Geometry stages of pipeline are computed as usual
- Approximations of $n.l$ during lighting
- Thresholded and clamped to achieve cel-shading (toon-shaded) effects
- Combine with other effects such as silhouettes, hatching
Other Effects

Fixed Function Limitations (OpenGL)

- Only supports basic Illumination Model
  - Defined as part of the open GL specification
- Lighting calculations are done per vertex
  - OK for Gouraud Shading
  - Insufficient for Phong Shading
  - No capability for dealing with things like refraction, bump-mapping

- Despite this FF still offers many gains to be made through hardware acceleration.
  - However Graphics Hardware has changed dramatically
  - Fixed function calls are automatically converted to shaders by the drivers.
  - Fixed function only is no longer sufficient for state-of-the-art effect
Fixed Function Overview

Input Vertices

T&L fixed computations

Co-ordinate Transformation to Viewport System

Geometry Stage (per-vertex level)

T&L fixed computations

Co-ordinate Transformation to Viewport System

Geometry Stage (per-vertex level)

Rasterization

Fixed Texture Stages

Final per-fragment computations: Fog, Alpha Test, Depth test, Stencil test

Raster Stage (per-pixel level)

Output to framebuffer

From Typhoon Labs GLSL tutorial http://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/
Programmable Pipeline Overview

Input Vertices

Programmable Vertex Processors
T&L custom computations; Per-pixel lighting, displacement mapping, particle systems etc.

Co-ordinate Transformation to Viewport System

GeometryStage
(per-vertex level)

Rasterization

Programmable Fragment Processors
Custom texture applications, Custom pixel data combinations, bump-parallax mapping, NPR

Input Textures

Final per-fragment computations: Fog, Alpha Test, Depth test, Stencil test

Raster Stage
(per-pixel level)

Output to framebuffer

From Typhoon Labs GLSL tutorial http://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/
Shaders

- Objects are typical affected by unique illumination and shading
- Popular Shading Languages:
  - RenderMan – Pixar (software based in toy story)
  - Cg – nVidia, 1st Commercial SL
  - HLSL – Microsoft/nVidia, Cg&Xbox Project
  - GLSL – SGI, ARB/3DLabs

SLIM, a shader tool for renderman

3DS Max material editor
A Brief History of GPU’s

- Hardware acceleration has tended to start at the end:
  - Earliest hardware provided fast rasterizing (hw frame buffers)
  - Later hardware added fast matrix maths for transformations and projections
  - Then we had T&L (Transform and Lighting) hardware

From nVidia. Available at http://www.nvidia.com/
OpenGL

- Open Graphics Library: Platform independent open industry-standard API for hardware accelerated graphical rendering
A note on APIs

- DirectX controlled by Microsoft
  - Dictates the platform that hardware manufacturers must support (in reality MS negotiates with hardware manufacturers)
  - Each version is essentially locked = DirectX is updated relatively frequently

- OpenGL controlled by the OpenGL Architecture
  - Review Board (ARB), part of Khronos Group
  - [http://www.opengl.org/](http://www.opengl.org/)
  - IBM, Intel, NVIDIA, AMD, etc.
  - Updates can be proposed / implemented by vendors via extensions
OpenGL Ecosystem

- Different platforms need -> API overlap not convergence
- Cooperation at Khronos -> consistency and synergies

OpenGL ES 2.0 on desktop as subset of OpenGL 4.1 for content mobility

High-end functionality developed first on desktop

Only OpenGL provides graphics functionality on all platforms

WebGL drives need for security and pervasive OpenGL ES 2.0

http://www.khronos.org
OpenGL Generations

3.1 uniform blocks, transform feedback, 3.0 texture arrays, in, out, centroid, flat, smooth, invariant, perspective

2.1 pixel buffer objects

2.0 GLSL multi render targets

1.5 VBO, occlusions

1.4 mipmap gen

1.3 multi-tex, compressed tex

1.2 3d textures

1.1 Vertex arrays and texture objects

http://www.khronos.org/
OpenGL Operation

Image from the OpenGL Shading Language (Orange Book) 3rd Edition
Programmable Graphics Pipeline

Image from the OpenGL Shading Language (Orange Book) 3rd Edition
Programmable Graphics Pipeline

- Programmable Functionality
  - Exposed via small programs – compiled to IL (assembly)
  - Now accessible through high level languages similar to c/c++
  - Hardware support variable across vendors

- Main Programmability:
  - Vertex Shaders
    - Input: Application geometry & per vertex attributes
    - Transform input in a meaningful way
  - Fragment Shaders
    - Input: Perspective Correct Attributes (interpolated)
    - Transform input into color or discard
  - (Recently) Geometry Shaders
OpenGL Shaders

- Initially defined as an OpenGL extension

Aside: OpenGL Extensions. Unlike DirectX, where changes occur only with new versions, OpenGL extensions are used to enable the core to change slowly. However we need to query the OpenGL implementation in app for available extensions

```c
#ifdef GL_EXT_bgra
    glDrawPixels(width, height, GL_BGRA_EXT, GL_UNSIGNED_BYTE, pixels);
#endif

... // Query run-time loaded extensions to be sure...
extensions = glGetString(GL_EXTENSIONS);
```

- Made Part of the core language with the release of OpenGL2
  - First programmable shaders available with the GeForce3 (shaders applied limited functionality to vertices)

- Modern cards have
  - Vertex Shaders
  - Fragment Shaders
  - with GeForce8: Geometry Shaders

- Shader functionality is variable between vendors
GLSL

- **OpenGL Shading Language aka GLslang**
  - A high level shading language
  - Based on the C programming language.
  - Created by the OpenGL ARB
  - More direct control of the graphics pipeline without having to use assembly or hardware-specific languages.

- **Advantages:**
  - Cross platform compatibility
  - Shaders that can be used on any hardware vendor’s graphics card that supports the OpenGL Shading Language.
  - Hardware vendor includes the GLSL compiler in their driver, allowing code optimized for their architecture.
Some GLSL Tools

- RenderMonkey – ATI (Win): create, compile and debug GLSL shaders [2008]

- Lumina (Platform independent) - a GLSL development tool [2008]
  - http://lumina.sourceforge.net/

- Shader Designer - extensive, easy to use GLSL IDE has ceased production by TyphoonLabs. However, it can still be downloaded and used for free. Example shaders and beginner tutorial documentation. [2004]
  - http://www.opengl.org/sdk/tools/ShaderDesigner/

- Demoniak3D – rapid coding and testing 3D solutions. Uses a mixture of XML, LUA scripting and GLSL to build a real time 3d scenes.

- Blender - contains GLSL support in its game engine, as of version 2.41.
OpenGL Versions

- Latest graphics cards required for OpenGL 4.1 (released June 2010)
  
- OpenGL 3.0+ deprecates certain old (FF) functionality (still available as legacy extensions)
  - `glTranslate...`, `glRotate`, `glVertex`

- Each comes with newer versions of shader language too

- For now see:
  - [http://www.opengl.org/wiki/Getting_started](http://www.opengl.org/wiki/Getting_started)
  - [http://www.opengl.org/wiki/OpenGL_3.0_and_beyond,_creating_a_context](http://www.opengl.org/wiki/OpenGL_3.0_and_beyond,_creating_a_context)
Vertices and Fragments

- A *vertex* is a location in space: 3D (or 4D) points that describe what we are modelling

- A *fragment*, generated at rasterization stage, carries with it intermediate information including its colour and location and depth information, that is used to update the corresponding pixel in the frame buffer

Vertex Shaders

- Allow us to define operations to be performed on vertices
  - Basically a short program in (usually) a C-like language
  - Executed on each vertex as it passes down the pipeline, while the shader is loaded.
  - have access to the OpenGL State
- If defined, completely replaces the fixed function pipelines' processing
  - This means that for each vertex the vertex shader must output all the information that the rasterizer needs to do its job

- Inputs: vertex data,
  - position,
  - color,
  - normals,
  - etc (as required by graphical program)

- Typical Tasks:
  - Vertex position transformation using the modelview and projection matrices
  - Normal transformation, and if required its normalization
  - Texture coordinate generation and transformation
  - Lighting per vertex or computing values required for lighting per pixel
  - Color computation
**Vertex Processor**

- **Input from:**
  - User-defined variables
  - Uniform variables (built-in or user defined)
  - Texture maps

- **Can also access**
  - Built-in constants

- **Output:**
  - Built-in special variables
  - User defined *varying* variables
  - Special vertex shader output variables
Vertex Processor

- **Input from:**
  - User-defined variables
  - Uniform variables (built-in or user defined)
  - Texture maps

- **Can also access**
  - Built-in constants

- **Output:**
  - Built-in special variables
  - User defined *varying* variables
  - Special vertex shader output variables
Vertex Shader Architecture

- Every time glVertex3f(...) or some variant is called the shader executes
A Simple GLSL Vertex Shader

```c
void main (void) {
    gl_position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
}
```

- GLSL includes its own data types such as `mat4` which represents a 4x4 matrix.
- The above code reproduces some functionality of the fixed pipeline.
  - positions the vertex based on the modelview matrix and performs a projection based on the Projection Matrix.
  - it does not set colour or any other attributes (colour could later be set by the fragment vertex shade).
  - N.B. the variables prefixed with gl are part of the OpenGL state and need not be declared.
Vertex Buffer Objects

- Shaders operate on vertices and the resulting data generated when vertices are passed down the pipeline.
- `glVertex` (deprecated in OpenGL3.0) used to specify a vertex
- Display Lists
  - Cannot be edited and must be rebuilt
  - Intended to be a macro, piles up commands that are then processed in a block
  - Deprecate in 3.0+
- Vertex Arrays
  - Used to reduce the number of function calls
  - Can have dynamic objects
  - But are still CPU based and stored in main memory - faster to store the data on the graphics hardware
- Vertex Buffer Objects
  - Similar to vertex arrays
  - Data stored directly on Graphics Hardware
  - In modern versions of OpenGL (3.0, ES) these are the only correct way to pass vertex data to graphics hardware
VBO Syntax – quick overview

- Create the Buffer
  ```
  glGenBuffers(1, &vertexVBOId);
  ```

- Enable and bind the Array Functionality
  ```
  glEnableClientState(GL_VERTEX_ARRAY);
  glBindBuffer(GL_ARRAY_BUFFER, vertexVBOId);
  ```

- Store the data
  ```
  glBufferData(GL_ARRAY_BUFFER, data_size_bytes, pointer_to_data, GL_DYNAMIC_DRAW);
  ```

- Optionally Index the data to specify a draw order of vertices and allow the same vertices to be reused
  ```
  glGenBuffers(1, &vertexIndexVBOId);
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vertexIndexVBOId);
  glBufferData(GL_ELEMENT_ARRAY_BUFFER, size_of_index_data, pointer_to_index_data, GL_DYNAMIC_DRAW);
  ```
Drawing the VBO

```c
glBindBuffer(GL_ARRAY_BUFFER, vertexVBOId);
glEnableClientState(GL_VERTEX_ARRAY);
```

- Specify where the data can be found for drawing
  ```c
  glVertexPointer(3, GL_FLOAT, data_structure_size), NULL);
  ```
  - `3` = the number of elements per data item

- Normal, texture Coordinate , and colour data (as well as GLSL attributes ) can be passed in
  ```c
  glEnableClientState(GL_NORMAL_ARRAY); ... glNormalPointer(...
  glEnableClientState(GL_COLOR_ARRAY);.. glColorPointer(...
  ```
  etc..

- But more detail on this later
Fragment Shaders

- In GLSL they have the same syntax as vertex shader programs

```cpp
void main()
{
    gl_FragColour = gl_FrontColor
}
```

- `gl_FrontColor` is not a value generated by the vertex shader, but is produced by interpolating the colours between vertices

- Typical role:
  - Operations on interpolated values
  - Texture access
  - Texture application
  - Fog
  - Color sum
  - Alpha Test
Fragment Processor

- **Special input variables**
  - `gl_FragCoord`
  - `gl_FrontFacing`

- **User-defined in variables**
  - Normal
  - ModelCoord
  - RefractionIndex
  - Density
  - etc.

- **Built-in uniform variables**
  - `gl_DepthRange.near`, etc.

- **User-defined uniform variables**
  - `ModelScaleFactor`, `EyePos`, `Epsilon`, `LightPosition`, `WeightingFactor1`, etc.

- **Texture maps**

- **Special output variables**
  - `gl_FragDepth`

- **User-defined output**
  - `FragColor`, etc.

Legend:
- Provided directly by application
- Provided indirectly by application
- Produced by rasterization
- Produced by the fragment processor
Fragment Shader Architecture

- OpenGL State
- Application Program
- Vertices
  - gl_Position
  - gl_FrontColor (Per Vertex)
- Rasterizer
- Fragments
  - gl_FrontColor (Interpolated)
- Fragment Shader
  - Pixels
    - gl_FragmentColor
- Frame Buffer
Using Shader Programs

- The Shader Program needs to be loaded and linked from the OpenGL Application.
  - Create (empty) shader objects by calling `glCreateShader`
  - Provide source code for these shaders by calling `glShaderSource`
  - Compile each of the shaders by calling `glCompileShader`
  - Create program object by calling `glCreateProgram`
  - Attach all the shader objects to the program by calling `glAttachShader`
  - Link the Program by calling `glLinkProgram`
  - Install the executable program as part of OpenGL’s current state by using `glUseProgram`
Using Shader Programs

- Minimal sample code:

```c
GLuint myProgObj
myProgObj = glCreateProgram();
GLchar vertexProg[] =
    "my_vertex_shader_filename";
GLuint myVertexObj;
myVertexObj = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(myVertexObj,1, vertexProg, NULL);
glCompileShade(myVertexObj);
glAttachObject(myProgObj, myVertexObj);
...
glLinkProgram(myProgObj);
glUseProgram(myProgObj);
```

- Once the above code has been run subsequent graphics primitives will be drawn with the shaders you provide rather than with OpenGL’s defined fixed functionality pipeline
Resources

- **GLSL Tutorials**

- **GLEW**
  - Automatically sets up whichever OpenGL extensions are available (i.e. Enables calls to shader hardware if it exists)

- **nVidia OpenGL SDK**
  - Contains Examples (which also use the GLEW library)