

CS770/870 Spring 2017

Open GL Shader Language

GLSL

Based on material from
Angel and Shreiner, *Interactive Computer Graphics, 6th Edition*, Addison-Wesley, 2011
Bailey and Cunningham, *Graphics Shaders 2nd Edition*, CRC Press, 2012
web source too numerous to remember

Preview

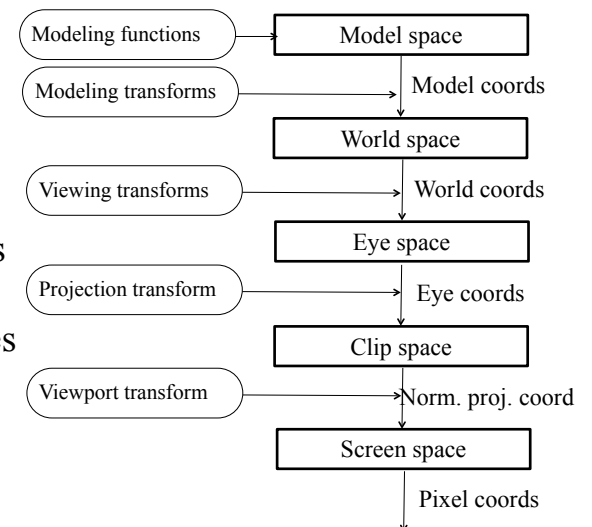
- Review traditional graphics pipeline
- CPU/GPU mixed pipeline issues
- Shaders
- GLSL graphics pipeline

Traditional Graphics Pipeline

- Original OpenGL based on traditional graphics pipeline
- Programmer specifies
 - Geometry, viewing, and projection
 - Object properties: vertices, normals
 - Appearance properties: colors, textures, shading, light
- Traditional OpenGL processing was predefined and straightforward with 2 major connected operations:
 - Vertex processing
 - Create and transform polygons into screen coords, rasterize polys
 - Pixel processing (now called *fragment* processing)
 - Color pixels

Traditional Vertex Pipeline

- View of graphics pipeline based on processing of vertex information
- OpenGL provides a fixed set of functions to control each step
- Programmer specifies parameters to the functions

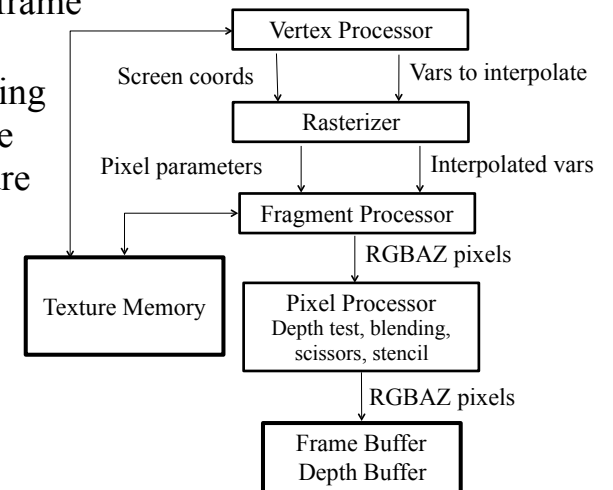


Shaders

- 1984: Rob Cook (Pixar) invented *shade trees*
 - Allow programmers to modify standard graphics pipeline with small shader scripts inserted in pipeline
 - Greatly expanded flexibility of (software) pipelines
 - Lots of others expanded nature and power of shaders
- Silicon Graphics' GL was hardware-oriented and used fixed function pipeline
- OpenGL 1.0 (1993) generalized GL and made it opensource – still fixed function
- OpenGL 2.0 (2004) added shader language, GLSL and support for GPU programming; cpu-driven, gpu adjunct
- OpenGL 3.0 (2008) far more emphasis on gnu
- OpenGL 4.0 (2012) all “old” functionality deprecated

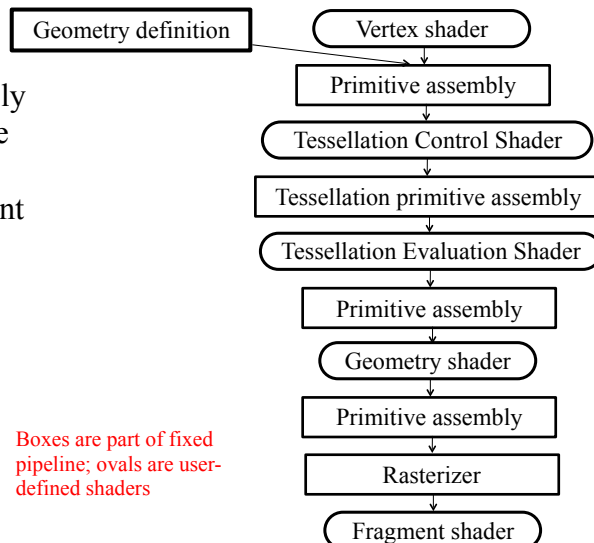
Fragment Processing Pipeline

- *pixel*: rgbaz for a frame buffer location
- *fragment*: everything needed to compute pixel: rgbaz, texture coords, and more



Expanded Graphics Pipeline

- Shaders can actually be inserted in more places in pipeline
- Vertex and fragment shaders are most important



Rise of GPUs

- GPU processing has evolved incredibly rapidly
 - Early goal was pixel per processor (data parallelism)
 - Hardware low level calculations
- Massively parallel computation meshes well with notion of user-written computations (shaders) for each pixel (*fragment shaders*)
- Architecture is also good for user-written computations for each vertex (*vertex shaders*) and for some geometric processing as well (*tessellation shaders*)

CPU/GPU Characteristics

- CPU strengths
 - General computational power
 - Access to large secondary storage
 - Well-known powerful software tools
 - Large commodity primary memory
- GPU strengths
 - Much faster for image/graphics computations
 - Massive parallelism
 - Large specialized memory

CPU/GPU Pipeline Distribution

- CPU best for
 - Application data access
 - Modeling
 - User interface
- GPU best for
 - Fragment processing
- Either can do vertex and tessellation processing
 - How decide?
 - Can depend on power of GPU
 - Can depend on nature of application processing
 - Let application programmer decide

CPU/GPU Pipeline Issues

- CPU/GPU normally have separate memories
 - GPU memory must already be dual-ported: GPU/display
- CPU/GPU data transfer is major bottleneck
- Need to distribute work so data transfer is minimized
- Standard OpenGL uses *direct* drawing
 - Every redraw event (30-60 time/sec during animation) must recreate entire image:
 - `glBegin; glVertex; glVertex ... glEnd;`
 - A lot of data transfer if vertices generated/stored on CPU.
- How can work distribution be defined by programmer?

Data Transfer Model

- Array based
- Buffers in GPU
- CPU code creates data
 - Asks GPU for buffer space
 - Uploads the data via arrays into buffer using buffer id
- Arrays can combine vertex, color and other info
- Can also define *uniform* variables: essentially global variables accessible to the shaders and to CPU code

Programming Framework

- CPU code
 - creates models
 - Specifies shader code to run on gpu
 - Compiles and uploads shader code
 - Can read and set *uniform* variables
 - Can read and get array information
- GPU pipeline code
 - Manages GPU pipeline and environment variables
 - Includes variables allowing shaders to communicate
 - Invokes user-written shader code
 - For each vertex, the vertex shader
 - For each fragment, the fragment shader
 - For each patch, the tessellation shaders

GLSL Vertex Shaders

- GLSL provides access in vertex shaders to
 - One vertex
 - *uniform* variables (user/shader defined)
 - *in* variables (position, normal, texture coords, color) passed to the shader from previous shader in pipeline
 - *out* variables (position, etc): passed to next shader
- Vertex shaders are responsible for
 - Vertex transformations
 - Normal transformations
 - Normal normalization (making unit length)
 - Handling per-vertex lighting
 - Handling texture coordinates at vertex

GLSL Fragment Shader

- For each fragment, maps fragment information (and *uniform* variables) to pixel information
- Rasterizer interpolates colors, normals, depths, texture coordinates, user variables defined for vertex: all are fragment shader variables
- Fragment shaders are responsible for
 - Color computation
 - Texturing
 - Per-pixel lighting
 - Fog
 - Discarding pixel fragments

GLSL Language Overview

- Simplified C++ (or extended C) with some Java-like features
 - “classes” are based on *struct*, so no inheritance
 - Predefined vector and matrix objects
 - No pointers
 - Arrays are first class objects
 - No *new*; dynamic allocation via function entry/exit
 - Function parameters passed by *value-return*
 - Variable qualifiers: *in*, *out*, *uniform* and more
 - Predefined variable names for each shader
 - Shared namespace between shaders

Vectors and Matrices

- GLSL has builtin vector and matrix objects:
 - vec2, vec3, vec4
 - mat2, mat3, mat4
- Predefined arithmetic operators:
 - vec op vec : +, -, *, / corresponding elements
 - mat * vec : matrix * vector yields vector
 - mat * mat : matrix * matrix yields matrix
- Multiple valid *vec* “field” names (*name sets*)
 - vec.x, vec.y, vec.z, vec.w for point variables
 - vec.r, vec.g, vec.b, vec.a for color variables
 - vec.s, vec.t, vec.p, vec.q for texture variables

Shared Namespace

- Each shader occurs in a pipeline
 - compute variables for shaders later in the pipeline
- How do shaders in a shader program communicate?
 - shared *namespace* for variables created in gpu memory
- *attribute* namespace
 - attributes are values associated with a vertex
 - defined in application program using a call such as
`glVertexAttribPointer( index, size, ... , GLvoid* )`
where *index* is an identifier
 - same array can contain position, normal, color and more
- *uniform* variables created by application as *read-only*
- *shader-defined* variables

General Storage Qualifiers

- Storage qualifiers describe where storage comes from
- global variable qualifiers:
 - `<none>` : scope is shader stage, so 2 vertex shaders with same named global variable, reference the same variable.
 - *const*: compile time constant; value can never be changed
 - *in*: input variable set by previous stages; cannot be changed
 - *out*: output variables that are passed to next stage, must be set
 - *in/out* variables cannot be structs; can be arrays
 - *uniform*: parameters passed to shader from user; do not change value during a single rendering call
- local variable qualifiers
 - `<none>` for local variable that can be changed by the shader
 - *const*: compile time constant; value can never be changed

Uniform Variables

Shader code

```
uniform vec4 ambientProduct;  
uniform vec4 diffuseProduct;  
uniform mat4 ModelView;  
uniform vec4 LightPosition;  
uniform float Shininess;
```

- CPU application

```
vec4 ambProduct = lightAmbient * materialAmbient;  
GLuint ambLoc // identifier for gpu location  
    = glGetUniformLocation(program, "ambientProduct");  
glUniform4fv( ambLoc, 1, ambProduct ); // set value  
vec4 diffProduct = lightDiffuse * materialDiffuse;  
GLuint diffLoc = . . .  
. . .
```

Interpolation Qualifiers

- A fragment shader *in* parameter can also have an interpolation qualifier:
 - *flat*: no interpolation is done; desired value (such as for color) is taken from first vertex (by default, but changeable)
 - *noperspective*: linear interpolation is done in window space (seldom desirable)
 - *smooth*: interpolation is correct for perspective distortion
- Other variable qualifiers can affect the interpolation
- *centroid* and *sample* have effect in pixel *multisampling* mode

Interface Blocks

- *uniform*, *in*, and *out* storage specification can use interface blocks to group variables
- It's a *struct*-like notation, but they are not structs.

```
uniform Transformations
{
    mat4 modelView;
    mat4 projection;
} name;
```

can refer to components as
name.modelView, etc.
name is optional
name can use array notation, such
as many[3]

GLSL Status

- GLSL has evolved rapidly and unevenly
 - Windows, Linux, Mac OS X have been at significantly different stages until very recently; the gaps are closing
 - Hard to write portable code
 - Debugging tools are pretty primitive

Key Features

- Data buffers
 - See basic GLSL demo:
 - glGenVertexArrays, glGenBuffers, glBufferData
- Projection/viewing
 - GLSL: no predefined matrix stacks nor viewing functions
 - Implement your own viewing/projection in shader program