CSCI 420 Computer Graphics
Lecture 7

Shaders

Shading Languages
GLSL
Vertex Array Objects
Vertex Shader
Fragment Shader
[Angel Ch. 1, 2, A]

Jernej Barbic
University of Southern California

1

Introduction

- The major advance in real time graphics has been the *programmable* pipeline:
  - First introduced by NVIDIA GeForce 3 (in 2001)
  - Supported by all modern high-end commodity cards
    - NVIDIA, AMD, Intel
  - Software Support
    - Direct3D
    - OpenGL
- This lecture: programmable pipeline and shaders

2

OpenGL Extensions

- Initial OpenGL version was 1.0
- Current OpenGL version is 4.5
- As graphics hardware improved, new capabilities were added to OpenGL
  - multitexturing
  - multisampling
  - non-power-of-two textures
  - shaders
  - and many more

3

OpenGL Grows via Extensions

- Phase 1: vendor-specific: GL_NV_multisample
- Phase 2: multi-vendor: GL_EXT_multisample
- Phase 3: approved by OpenGL's review board GL_ARB_multisample
- Phase 4: incorporated into OpenGL (v1.3)

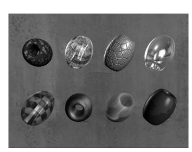
4

OpenGL 2.0 Added Shaders

- Shaders are customized programs that replace a part of the OpenGL pipeline
- They enable many effects not possible by the fixed OpenGL pipeline
- Motivated by Pixar's Renderman (offline shader)

5

Shaders Enable Many New Effects


Complex materials

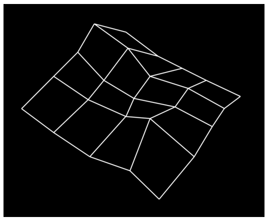

Shadowing


Lighting environments


Advanced mapping

6

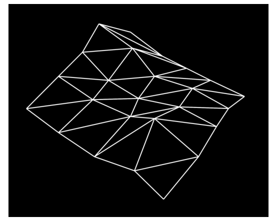
Vertex Shader

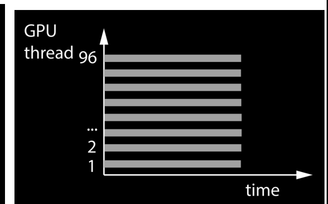


5x5 terrain (as in hw1)
5x5 = 25 vertices
4x4 = 16 quads

7

Vertex Shader



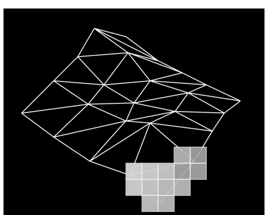


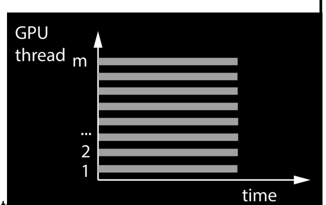
96 vertex shaders
execute in parallel

User must tessellate into
triangles (in the VBO)
4 x 4 x 2 = 32 triangles
32 x 3 = 96 vertices (assuming GL_TRIANGLES)

8

Fragment Shader





Rasterization

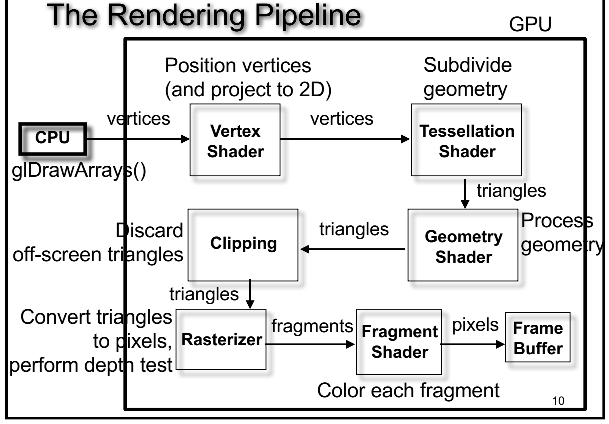
rasterizes into
13 pixels

m fragment shaders
execute in parallel

Triangles (now in 2D) cover m pixels
Some pixels may repeat in multiple triangles

9

The Rendering Pipeline



10

Shaders

- Vertex shader (= vertex program)
- Tessellation control and evaluation shader (OpenGL 4.0; subdivide the geometry)
- Geometry shader
  (OpenGL 3.2; process, generate, replace or delete geometry)
- Fragment shader (= fragment program)
- Compute shader (OpenGL 4.3; general purpose)

11

Shaders

- Compatibility profile: Default shaders are provided by OpenGL *(fixed-function pipeline)*
- Core profile: no default vertex or fragment shader; must be provided by the programmer
- Tessellation shaders, geometry shaders and compute shaders are *optional*

12

Shader Variables Classification

- Attribute
  - Information specific to each vertex/pixel passed to vertex/fragment shader

Example:
Vertex Color

- Uniform
  - Constant information passed to vertex/fragment shader
  - Cannot be written to in a shader

Example:
Light Position
Eye Position

- Out/in
  - Info passed from vertex shader to fragment shader
  - Interpolated from vertices to pixels
  - Write in vertex shader, but only read in fragment shader

Example:
Vertex Color
Texture Coords

- Const
  - To declare non-writable, constant variables

Example:
pi, e, 0.480

13

Shaders Are Written in *Shading Languages*

- Early shaders: assembly language
- Since ~2004: high-level shading languages
  - OpenGL Shading Language (GLSL)
    - highly integrated with OpenGL
  - Cg (NVIDIA and Microsoft), very similar to GLSL
  - HLSL (Microsoft), the shading language of Direct3D
  - All of these are simplified versions of C/C++

14

GLSL

- The shading language of OpenGL
- Managed by OpenGL Architecture Review Board
- Introduced in OpenGL 2.0
- We use shader version 1.50:
  #version 150
  (a good version supporting the core profile features)
- Current shader version: 4.50

15

Vertex Shader

- Input: vertices, in object coordinates, and per-vertex attributes:
  - color
  - normal
  - texture coordinates
  - many more
- Output:
  - vertex location in clip coordinates
  - vertex color
  - vertex normal
  - many more are possible

16

Basic Vertex Shader in GLSL

```
#version 150
in vec3 position; // input position, in object coordinates
in vec4 color; // input color
out vec4 col; // output color

uniform mat4 modelViewMatrix; // uniform variable to store the modelview mtx
uniform mat4 projectionMatrix; // uniform variable to store the projection mtx

void main()
{
  // compute the transformed and projected vertex position (into gl_Position)
  gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0f);
  // compute the vertex color (into col)
  col = color;
}
```

17

Fragment Shader

- Input: fragments (tentative pixels), and per-pixel attributes:
  - color
  - normal
  - texture coordinates
  - many more are possible
- Inputs are outputs from the vertex shader, interpolated (by the GPU) to the pixel location !
- Output:
  - pixel color
  - depth value
  - can discard the fragment using the **discard** keyword

18

Basic Fragment Shader

```
#version 150

in vec4 col; // input color (computed by the interpolator)
out vec4 c; // output color (the final fragment color)

void main()
{
  // compute the final fragment color
  c = col;
}
```

19

Another Fragment Shader

```
#version 150

in vec4 col; // input color (computed by the interpolator)
out vec4 c; // output color (the final fragment color)

void main()
{
  // compute the final fragment color
  c = vec4(1.0, 0.0, 0.0, 1.0);
}
```

20

Pipeline program

- Container for all the shaders
- Vertex, fragment, geometry, tessellation, compute
- Can have several pipeline programs
  (for example, one for each rendering style)
- Must have at least one (core profile)
- At any moment of time, exactly one pipeline program is bound (active)

21

Installing Pipeline Programs

- Step 1: Create Shaders
  – Create handles to shaders
- Step 2: Specify Shaders
  – load strings that contain shader source
- Step 3: Compiling Shaders
  – Actually compile source (check for errors)
- Step 4: Creating Program Objects
  – Program object controls the shaders
- Step 5: Attach Shaders to Programs
  – Attach shaders to program objects via handle
- Step 6: Link Shaders to Programs
  – Another step similar to attach
- Step 7: Enable Shaders
  – Finally, let OpenGL and GPU know that shaders are ready

22

Our helper library: PipelineProgram

```
// load shaders from a file
int BuildShadersFromFiles(const char * filenameBasePath,
        const char * vertexShaderFilename,
        const char * fragmentShaderFilename,
        const char * geometryShaderFilename = NULL,
        const char * tessellationControlShaderFilename = NULL,
        const char * tessellationEvaluationShaderFilename =
  NULL);
```

23

Our helper library: PipelineProgram

```
// load shaders from a C text string
int BuildShadersFromStrings(const char * vertexShaderCode,
        const char * fragmentShaderCode,
        const char * geometryShaderCode = NULL,
        const char * tessellationControlShaderCode = NULL,
        const char * tessellationEvaluationShaderCode = NULL);
```

24

Our helper library: PipelineProgram

```
// global variable
BasicPipelineProgram * pipelineProgram;

// during initialization:
pipelineProgram = new BasicPipelineProgram();
pipelineProgram->Init("../openGLHelper-starterCode");

// before rendering, bind (activate) the pipeline program:
pipelineProgram->Bind();
```

If you want to change the pipeline program, call "Bind" on the new pipeline program

Setting up the Uniform Variables

Uploading the modelview matrix transformation to the GPU (in the display function)

```
// get a handle to the program
GLuint program = pipelineProgram->GetProgramHandle();
// get a handle to the modelViewMatrix shader variable
GLint h_modelViewMatrix =
  glGetUniformLocation(program, "modelViewMatrix");

float m[16]; // column-major
// here, must fill m (missing code; use OpenGLMatrix class)

// upload m to the GPU
pipelineProgram->Bind(); // must do (once) before glUniformMatrix4fv
GLboolean isRowMajor = GL_FALSE;
glUniformMatrix4fv(h_modelViewMatrix, 1, isRowMajor, m);
```


Setting up the Uniform Variables

Repeat the same process also for the projection matrix:

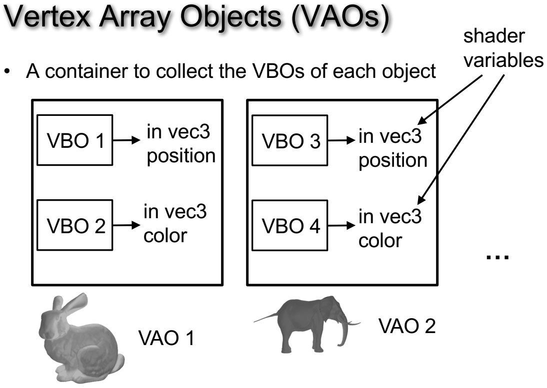
```
// get a handle to the program
GLuint program = pipelineProgram->GetProgramHandle();
// get a handle to the projectionMatrix shader variable
GLint h_projectionMatrix =
  glGetUniformLocation(program, "projectionMatrix");

float p[16]; // column-major
// here, must fill p… (use our OpenGLMatrix class)
// (missing code to fill p)

// upload p to the GPU
GLboolean isRowMajor = GL_FALSE;
glUniformMatrix4fv(h_projectionMatrix, 1, isRowMajor, p);
```


Vertex Array Objects (VAOs)

- A container to collect the VBOs of each object

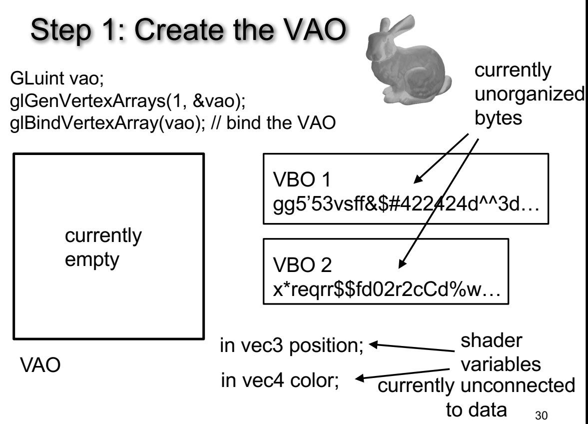


Vertex Array Objects (VAOs)

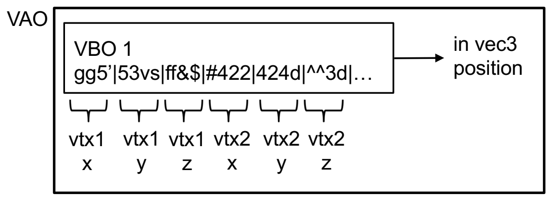
- A container to collect the VBOs of each object
- Usage is mandatory (by the OpenGL standard)
- During initialization: create VBOs (one or more per object), create VAOs (one per object), and place the VBOs into the proper VAOs
- At render time: bind the VAO, then call glDrawArrays(), then unbind

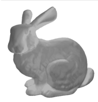
Step 1: Create the VAO

```
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao); // bind the VAO
```



Step 2: Connect VBO to VAO and the shader variable, and interpret VBO





```
GLuint loc = glGetAttribLocation(program, "position");
glVertexAttribPointer(loc, 3, GL_FLOAT,
    GL_FALSE, stride, offset);
```


VAO code ("position" shader variable)

During initialization:

```
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao); // bind the VAO

// bind the VBO "buffer" (must be previously created)
glBindBuffer(GL_ARRAY_BUFFER, buffer);
// get location index of the "position" shader variable
GLuint loc = glGetAttribLocation(program, "position");
glEnableVertexAttribArray(loc); // enable the "position" attribute
const void * offset = (const void*) 0;  GLsizei stride = 0;
GLboolean normalized = GL_FALSE;
// set the layout of the "position" attribute data
glVertexAttribPointer(loc, 3, GL_FLOAT, normalized, stride, offset);
```


VAO code ("color" shader variable)

```
// get the location index of the "color" shader variable
loc = glGetAttribLocation(program, "color");
glEnableVertexAttribArray(loc); // enable the "color" attribute
offset = (const void*) sizeof(positions);
GLsizei stride = 0;
Glboolean normalized = GL_FALSE;
// set the layout of the "color" attribute data
glVertexAttribPointer(loc, 4, GL_FLOAT, normalized, stride, offset);

glBindVertexArray(0); // unbind the VAO
```


Use the VAO

In the display function:

```
pipelineProgram->Bind(); // bind the pipeline program
glBindVertexArray(vao); // bind the VAO

GLint first = 0;
GLsizei count = numVertices;
glDrawArrays(GL_TRIANGLES, first, count);

glBindVertexArray(0); // unbind the VAO
```


GLSL: Data Types

- Scalar Types
  - float - 32 bit, very nearly IEEE-754 compatible
  - int - at least 16 bit
  - bool - like in C++
- Vector Types
  - vec[2 | 3 | 4] - floating-point vector
  - ivec[2 | 3 | 4] - integer vector
  - bvec[2 | 3 | 4] - boolean vector
- Matrix Types
  - mat[2 | 3 | 4] - for 2x2, 3x3, and 4x4 floating-point matrices
- Sampler Types
  - sampler[1 | 2 | 3]D - to access texture images

GLSL: Operations

- Operators behave like in C++
- Component-wise for vector & matrix
- Multiplication on vectors and matrices

- Examples:
  - Vec3 t = u * v;
  - float f = v[2];
  - v.x = u.x + f;

GLSL: Swizzling

- Swizzling is a convenient way to access individual vector components

```
vec4 myVector;
myVector.rgba; // is the same as myVector
myVector.xy; // is a vec2
myVector.b; // is a float
myVector[2]; // is the same as myVector.b
myVector.xb; // illegal
myVector.xxx; // is a vec3
```

37

GLSL: Flow Control

- Loops
  - C++ style if-else
  - C++ style for, while, and do
- Functions
  - Much like C++
  - Entry point into a shader is void main()
  - No support for recursion
  - Call by value-return calling convention
- Parameter Qualifiers
  - in - copy in, but don't copy out
  - out - only copy out
  - inout - copy in and copy out

Example function:

```
void ComputeTangent(
  in vec3 N,
  out vec3 T,
  inout vec3 coord)
{
  if((dot(N, coord)>0)
    T = vec3(1,0,0);
  else
    T = vec3(0,0,0);
  coord = 2 * T;
}
```

38

GLSL: Built-in Functions

- Wide Assortment
  - Trigonometry (cos, sin, tan, etc.)
  - Exponential (pow, log, sqrt, etc.)
  - Common (abs, floor, min, clamp, etc.)
  - Geometry (length, dot, normalize, reflect, etc.)
  - Relational (lessThan, equal, etc.)
- Need to watch out for common reserved keywords
- Always use built-in functions, do not implement your own
- Some functions are not implemented on some cards

39

GLSL: Built-in Variables

- Always prefaced with gl_
- Accessible to both vertex and fragment shaders
- Examples:
  - (input) gl_VertexID: index of currently processed vertex
  - (input) gl_FrontFacing: whether pixel is front facing or not
  - (input) gl_FragCoord : x,y: coordinate of pixel, z: depth
  - (output) gl_FragDepth: pixel depth

40

GLSL: Accessing OpenGL State (Compatibility Profile Only)

- Vertex shader: Have access to several vertex attributes: gl_Color, gl_Normal, gl_Vertex, etc.
- Fragment shader: Write to special output variable: gl_FragColor
- Uniform Variables
  - Matrices (ModelViewMatrix, ProjectionMatrix, inverses, transposes)
  - Materials (in MaterialParameters struct, ambient, diffuse, etc.)
  - Lights (in LightSourceParameters struct, specular, position, etc.)
- Varying Variables
  - FrontColor for colors
  - TexCoord[] for texture coordinates

These do not work in the core profile!

41

Debugging Shaders

- More difficult than debugging C programs
- Common show-stoppers:
  - Typos in shader source
  - Assuming implicit type conversion (cannot convert vec4 to vec3)
  - Attempting to connect VAOs to non-existent (say, due to a typo) shader variables
- Very important to check error codes; use status functions like:
  - glGetShaderiv(GLuint shader, GLenum pname, GLint * params)

42

Summary

- Shading Languages
- Program Pipeline
- Vertex Array Objects
- GLSL
- Vertex Shader
- Fragment Shader

43