Shaders and GLSL

Prof. George Wolberg
Dept. of Computer Science
City College of New York

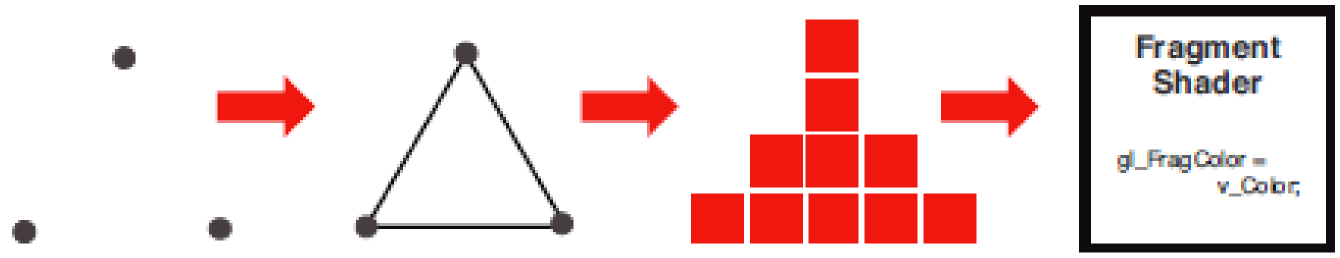
Objectives

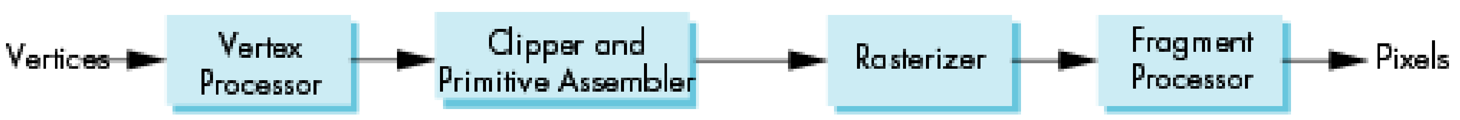
- Introduce shaders
    - Vertex shaders
    - Fragment shaders

  - Introduce a standard program structure
- Initialization steps and program structure
- Review sample shaders

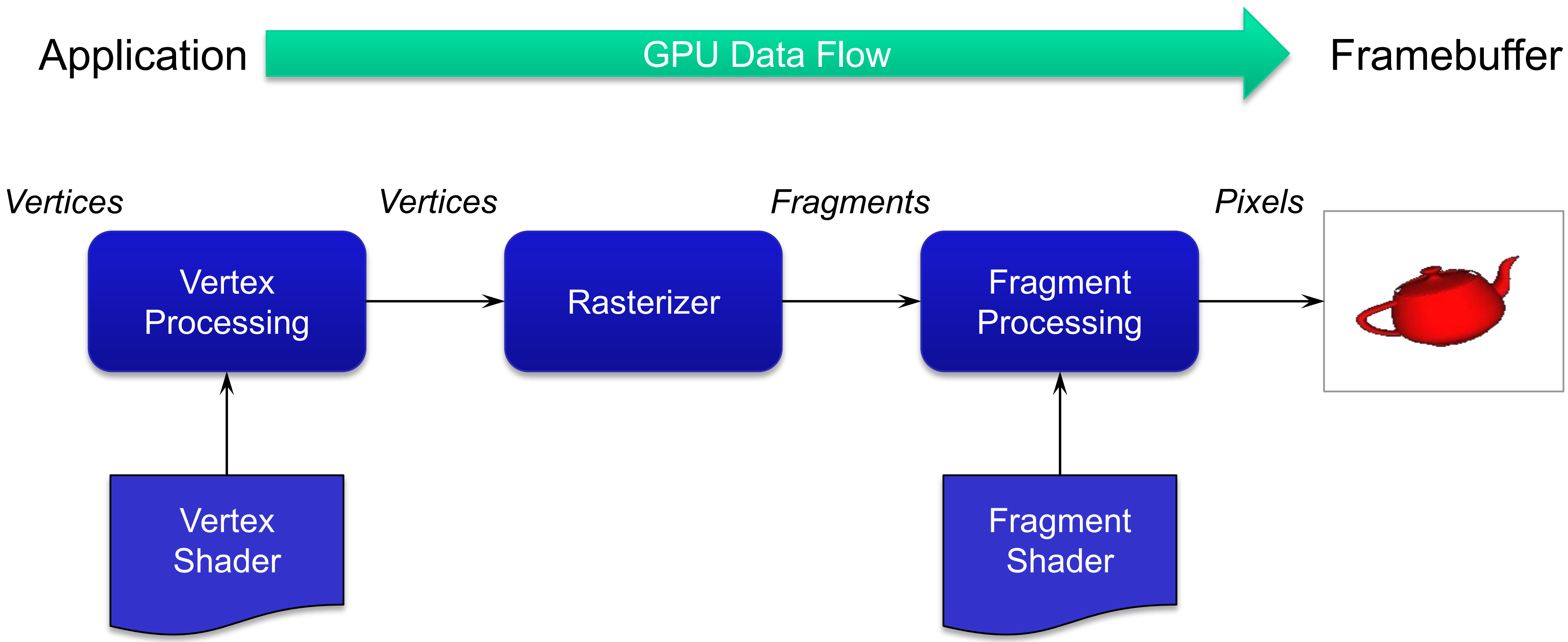
Graphics Pipeline

- Vertices stream into vertex processor and are transformed into new vertices
- These vertices are collected to form primitives
- Primitives are rasterized to form fragments
- Fragments are colored by fragment processor

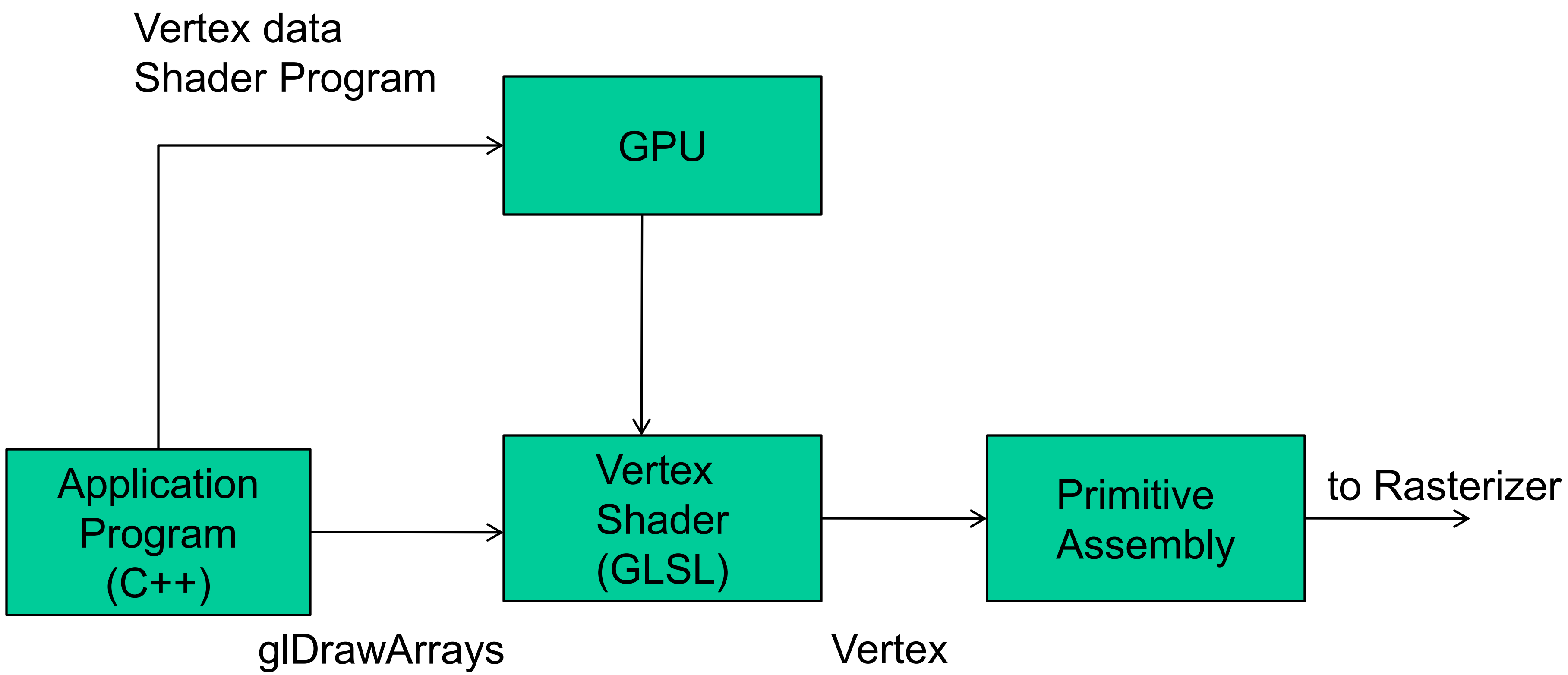




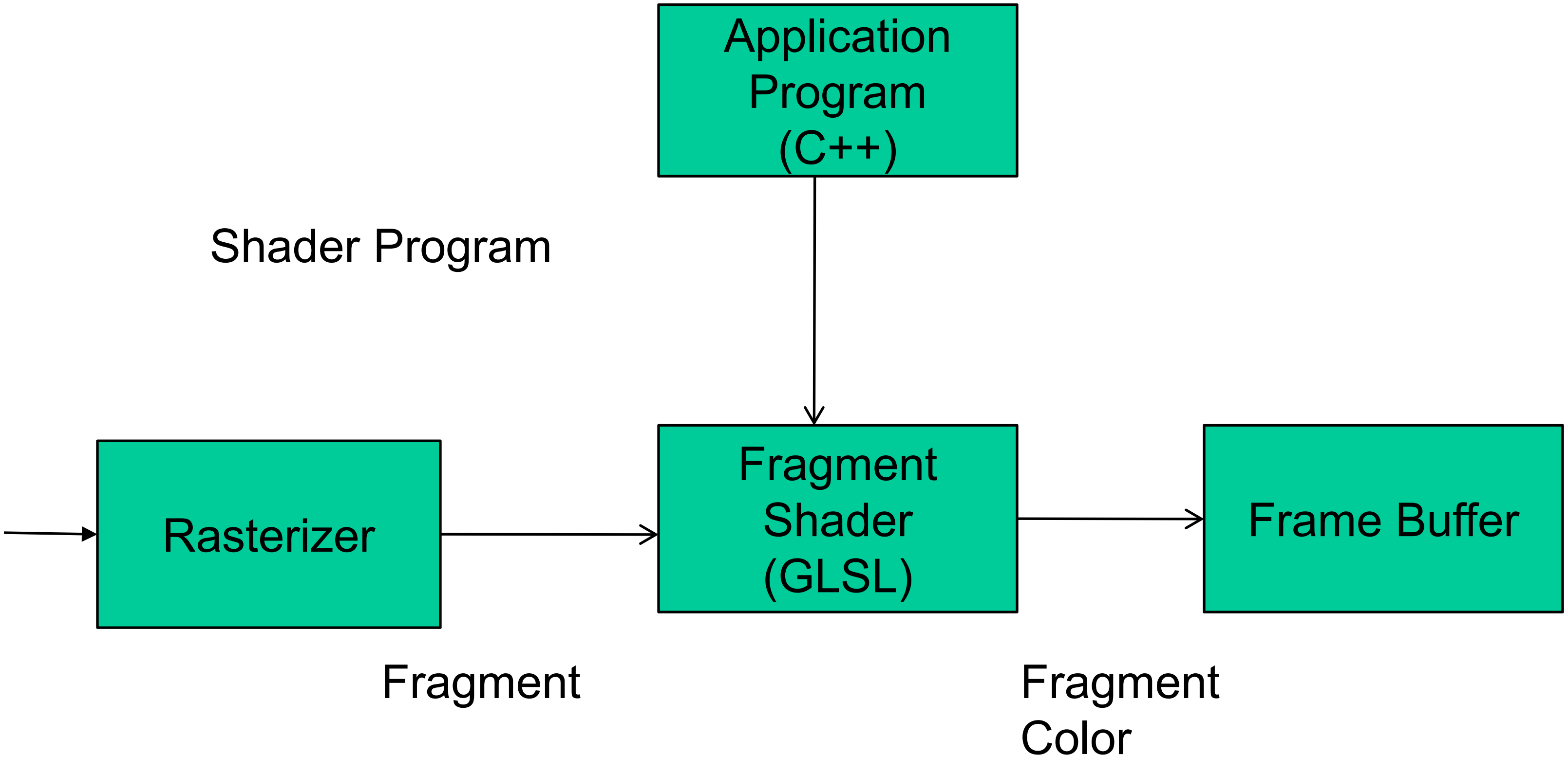
Simplified Pipeline Model



Execution Model



Execution Model



Writing Shaders

- As of OpenGL 3.1, application programs must provide shaders
  - Application programs reside on CPU
  - Shader programs reside on GPU
- OpenGL extensions added for vertex and fragment shaders
- Shaders are written with the OpenGL Shading Language (GLSL)

GLSL: OpenGL Shading Language

- Part of OpenGL 2.0 and up
- High level C-like language
- New data types
  - Matrices (**`mat2, mat3, mat4`**)
  - Vectors (**`vec2, vec3, vec4,`** …)
  - Samplers (**`sampler1D, sampler2D, …`**)
- New qualifiers: **`in, out, uniform`**
- Similar to Microsoft HLSL
- New OpenGL functions to compile, link, and get information to shaders

Differences between GLSL and C

- Matrix and vector types are built into GLSL
  - they can be passed into and output from GLSL functions, e.g. mat3 func(mat3 a)
- GLSL is designed to be run on massively parallel implementations
  - Recursion is not allowed in GLSL
  - No pointers in GLSL
  - Precision requirements for floats are not as strict as IEEE standards that govern C implementations

GLSL Data Types

- Scalar types: `float, int, bool`
- Vector types: `vec2, vec3, vec4`
  `ivec2, ivec3, ivec4`
  `bvec2, bvec3, bvec4`
- Matrix types: `mat2, mat3, mat4`
- Texture sampling: `sampler1D, sampler2D, sampler3D, samplerCube`
- C++ Style Constructors

```
vec3 a = vec3(1.0, 2.0, 3.0);
```

Qualifiers (1)

- GLSL has many of the same qualifiers as C/C++
- Need others due to the nature of the execution model
- Variables can change
  - Once per primitive
  - Once per vertex
  - Once per fragment
  - At any time in the application
- Vertex attributes are interpolated by the rasterizer into fragment attributes

Qualifiers (2)

- `in`, `out`
  - Copy vertex attributes and other variable into and out of shaders

```
in  vec2 texCoord;
out vec4 color;
```

- `uniform`
  - shader-constant variable from application

```
uniform float time;
uniform vec4 rotation;
```

Simple Vertex Shader

input from application; may use **attribute** (older style) instead of **in**

```
in vec4 vPosition;
void main(void)
{
    gl_Position = vPosition;
}
```

must link to variable in application

built-in variable

Simple Fragment Program

```
void main(void)
{
  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Attribute Qualifier

- Attribute-qualified variables can change at most once per vertex
- There are a few built in variables such as gl_Position but most have been deprecated
- User defined (in application program)
  - Use **in** or **attribute** qualifier to get to shader
  - `in float temperature`
  - `attribute vec3 velocity`

Varying Qualified

- Variables that are passed from vertex shader to fragment shader
- Automatically interpolated by the rasterizer
- Old style used the varying qualifier

```
varying vec4 color;
```

- Now use **out** in vertex shader and **in** in the fragment shader

```
out vec4 color;
```

Attribute and Varying Qualifiers

- Starting with GLSL 1.5 attribute and varying qualifiers have been replaced by in and out qualifiers
- No changes needed in application
- Vertex shader example:

```
#version 1.4
attribute vec3 vPosition;
varying vec3 color;
```

```
#version 1.5
in vec3 vPosition;
out vec3 color;
```

Uniform Qualified

- Variables that are constant for an entire primitive
- Can be changed in application and sent to shaders
- Cannot be changed in shader
- Used to pass information to shader such as the bounding box of a primitive

Built-in Variables

- **`gl_Position`**
  - (required) output position of current vertex
- **`gl_PointSize`**
  - pixel width/height of the point being rasterized
- **`gl_FragCoord`**
  - input fragment position
- **`gl_FragDepth`**
  - input depth value in fragment shader

Simple Vertex Shader

```
#version 450

in  vec4 a_Position;
in  vec4 a_Color;
out vec4 color;

void main()
{
   color = a_Color;
   gl_Position = a_Position;
}
```

Simple Fragment Shader

```
#version 450

in  vec4 color;
out vec4 fColor; // fragment's final color

void main()
{
    fColor = color;
    // OR: gl_FragColor = color;
}
```

Operators and Functions

- Standard C functions
  - Arithmetic: `sqrt`, `power`, `abs`
  - Trigonometric: `sin`, `asin`
  - Graphical: `length`, `reflect`
- Overloading of vector and matrix types

```
mat4 a;
vec4 b, c, d;
c = b*a; // a row vector stored as a 1D array
d = a*b; // a column vector stored as a 1D array
```

Swizzling and Selection

- Can refer to array elements by element using [ ] or selection (.) operator with
  - x, y, z, w
  - r, g, b, a
  - s, t, p, q
  - **`a[2], a.b, a.z, a.p`** are the same
- **Swizzling** operator lets us manipulate components

```
vec4 a;
a.yz = vec2(1.0, 2.0);
```

Programming with OpenGL: More GLSL

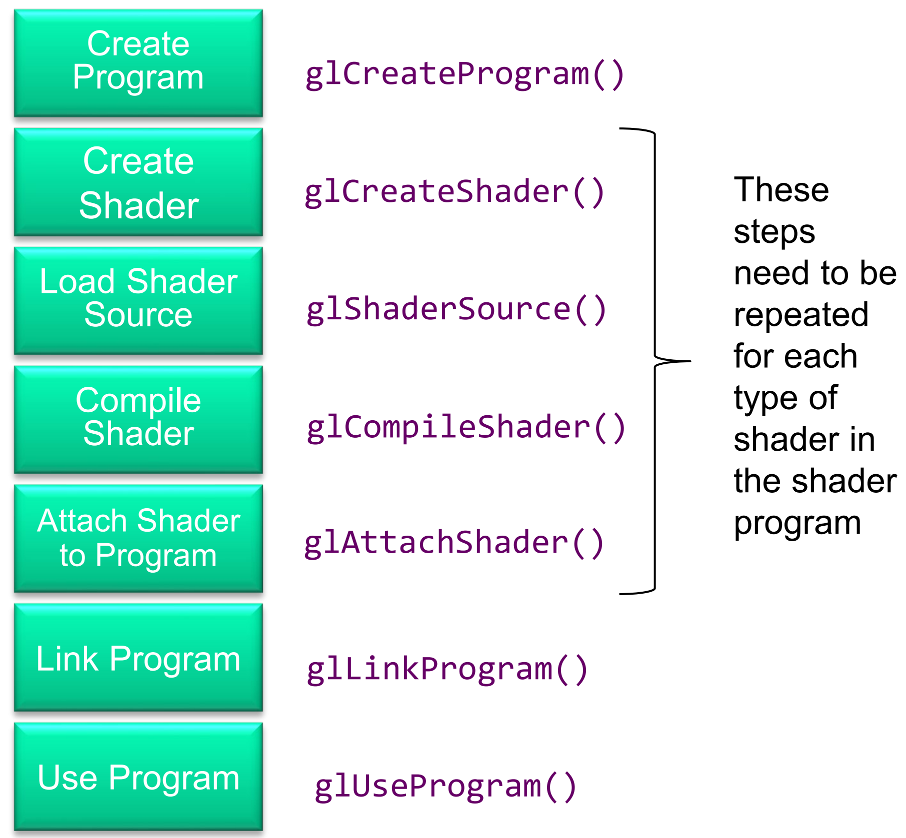
Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

- Coupling shaders to applications
  - Reading
  - Compiling
  - Linking
- Vertex Attributes
- Setting up uniform variables
- Example applications

Getting Your Shaders into OpenGL

- Shaders need to be compiled and linked to form an executable shader program
- OpenGL provides the compiler and linker
- A program must contain
  - vertex and fragment shaders
  - other shaders are optional



Linking Shaders with Application

- Read shaders
- Compile shaders
- Create a program object
- Link everything together
- Link variables in application with variables in shaders
  - Vertex attributes
  - Uniform variables

Program Object

- Container for shaders
  - Can contain multiple shaders
  - Other GLSL functions

```
Gluint program = glCreateProgram();

// define shader objects here
glUseProgram (program);
glLinkProgram(program);
```

Reading a Shader

- Shaders are added to the program object and compiled
- Usual method of passing a shader is as a null-terminated string using the function **`glShaderSource`**
- If the shader is in a file, we can write a reader to convert the file to a string

Adding a Vertex Shader (1)

```
GLuint LoadShaders(const char * vertex_file_path,const char * fragment_file_path){

  // Create the shaders
  GLuint VertexShaderID   = glCreateShader(GL_VERTEX_SHADER);
  GLuint FragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);

  // Read the Vertex Shader code from the file
  std::string VertexShaderCode;
  std::ifstream VertexShaderStream(vertex_file_path, std::ios::in);
  if(VertexShaderStream.is_open())
  {
    std::string Line = "";
    while(getline(VertexShaderStream, Line))
      VertexShaderCode += "\n" + Line;
    VertexShaderStream.close();
  }

  // Read the Fragment Shader code from the file
  std::string FragmentShaderCode;
  std::ifstream FragmentShaderStream(fragment_file_path, std::ios::in);
  if(FragmentShaderStream.is_open()){
    std::string Line = "";
    while(getline(FragmentShaderStream, Line))
      FragmentShaderCode += "\n" + Line;
    FragmentShaderStream.close();
  }
```

Adding a Vertex Shader (2)

```
GLint Result = GL_FALSE;
int InfoLogLength;

// Compile Vertex Shader
printf("Compiling shader : %s\n", vertex_file_path);
char const * VertexSourcePointer = VertexShaderCode.c_str();
glShaderSource (VertexShaderID, 1, &VertexSourcePointer , NULL);
glCompileShader(VertexShaderID);

// Check Vertex Shader
glGetShaderiv(VertexShaderID, GL_COMPILE_STATUS, &Result);
glGetShaderiv(VertexShaderID, GL_INFO_LOG_LENGTH, &InfoLogLength);
std::vector<char> VertexShaderErrorMessage(InfoLogLength);
glGetShaderInfoLog(VertexShaderID, InfoLogLength, NULL, &VertexShaderErrorMessage[0]);
fprintf(stdout, "%s\n", &VertexShaderErrorMessage[0]);

// Compile Fragment Shader
printf("Compiling shader : %s\n", fragment_file_path);
char const * FragmentSourcePointer = FragmentShaderCode.c_str();
glShaderSource (FragmentShaderID, 1, &FragmentSourcePointer , NULL);
glCompileShader(FragmentShaderID);

// Check Fragment Shader
glGetShaderiv(FragmentShaderID, GL_COMPILE_STATUS, &Result);
glGetShaderiv(FragmentShaderID, GL_INFO_LOG_LENGTH, &InfoLogLength);
std::vector<char> FragmentShaderErrorMessage(InfoLogLength);
glGetShaderInfoLog(FragmentShaderID, InfoLogLength, NULL, &FragmentShaderErrorMessage[0]);
fprintf(stdout, "%s\n", &FragmentShaderErrorMessage[0]);
```

Adding a Vertex Shader (3)

```
  // Link the program
  fprintf(stdout, "Linking program\n");
  GLuint ProgramID = glCreateProgram();
  glAttachShader(ProgramID, VertexShaderID);
  glAttachShader(ProgramID, FragmentShaderID);
  glLinkProgram(ProgramID);

  // Check the program
  glGetProgramiv(ProgramID, GL_LINK_STATUS, &Result);
  glGetProgramiv(ProgramID, GL_INFO_LOG_LENGTH, &InfoLogLength);
  std::vector<char> ProgramErrorMessage( max(InfoLogLength, int(1)) );
  glGetProgramInfoLog(ProgramID, InfoLogLength, NULL, &ProgramErrorMessage[0]);
  fprintf(stdout, "%s\n", &ProgramErrorMessage[0]);

  glDeleteShader(VertexShaderID);
  glDeleteShader(FragmentShaderID);

  return ProgramID;
}
```

A Simpler Way

- Qt created a routine to make it easy to load shaders

```
#include <QGLShaderProgram>
QGLShaderProgram program;
program.addShaderFromSourceFile(QGLShader::Vertex,   “:/vshader.glsl”);
program.addShaderFromSourceFile(QGLShader::Fragment, “:/fshader.glsl”);
```

- Fails if shaders don’t compile, or program doesn’t link
- Add shader programs in qrc file:

```
<RCC>
     <qresource prefix=“/”>
         <file>vshader.glsl</file>
         <file>fshader.glsl</file>
     </qresource>
</RCC>
```

Associating Shader Variables and Data

- Vertex attributes are named in the shaders
- Linker forms a table
- Application can get index from table and tie it to an application variable
- Similar process for uniform variables

Vertex Attribute Example

```
GLuint positionLoc = glGetAttribLocation( program, "a_Position" );
glEnableVertexAttribArray( positionLoc );
glVertexAttribPointer(positionLoc, // attribute at location positionLoc
                      2,             // size
                      GL_FLOAT,      // type
                      GL_FALSE,      // normalized?
                       0,            // stride
                      (void *) 0     // array buffer offset
                      );
```

Uniform Variable Example

```
GLint angleLoc; // location of angle defined in shader
angleLoc = glGetUniformLocation(program, "angle");

// my_angle set in application
GLfloat my_angle = 5.0 // or some other value

glUniform1f(angleLoc, my_angle);
```

Adding Color

- If we set a color in the application, we can send it to the shaders as a vertex attribute or as a uniform variable depending on how often it changes
- Let's associate a color with each vertex
- Set up an array of same size as positions
- Send to GPU as a vertex buffer object

Setting Colors

```
vec3  base_colors[4] = {vec3(1.0, 0.0. 0.0), ….
vec3  colors[NumVertices];
vec3  points[NumVertices];

// in loop setting positions

colors[i] = base_colors[color_index]
points[i] = …….
```

Setting Up Buffer Object

```
// this will identify our buffer
Gluint vertexbuffer

// generate one buffer, put the resulting identifier in vertexbuffer
glGenBuffers(1, &vertexbuffer);

// the following commands will talk about our "vertexbuffer" buffer
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);

// pass vertices to OpenGL; NULL ptr: data will be loaded later
glBufferData(GL_ARRAY_BUFFER, sizeof(points) + sizeof(colors),
       NULL, GL_STATIC_DRAW);

// load data separately
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(points), points);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(points), sizeof(colors),
          colors);
```

Link Buffer with Vertex Attributes

```
// vPosition and vColor identifiers in vertex shader

loc1 = glGetAttribLocation(program, “vPosition”);
glEnableVertexAttribArray(loc1);
glVertexAttribPointer(loc1, 3, GL_FLOAT, GL_FALSE, 0,
                      (void *) 0);

loc2 = glGetAttribLocation(program, “vColor”);
glEnableVertexAttribArray(loc2);
glVertexAttribPointer(loc2, 3, GL_FLOAT, GL_FALSE, 0,
                      (void *) sizeof(points));

// draw the triangles
glDrawArrays(GL_TRIANGLES, 0, NumVertices);
```

Vertex Shader Examples

- A vertex shader is initiated by each vertex output by `glDrawArrays()`
- A vertex shader must output a position in clip coordinates to the rasterizer
- Basic uses of vertex shaders
  - Transformations
  - Lighting
  - Moving vertex positions

Wave Motion Vertex Shader

```
in vec4 vPosition;
uniform float xs, zs, // frequencies
uniform float h;      // height scale
void main()
{
  vec4 t = vPosition;
  t.y = vPosition.y
      + h*sin(time + xs*vPosition.x)
      + h*sin(time + zs*vPosition.z);
  gl_Position = t;
}
```

Particle System

```
in vec3 vPosition;
uniform mat4 ModelViewProjectionMatrix;
uniform vec3 init_vel;
uniform float g, m, t;

void main(){
   vec3 object_pos;
   object_pos.x = vPosition.x + vel.x*t;
   object_pos.y = vPosition.y + vel.y*t + g/(2.0*m)*t*t;
   object_pos.z = vPosition.z + vel.z*t;
   gl_Position  = ModelViewProjectionMatrix*vec4(object_pos,1);
}
```

Pass Through Fragment Shader

```
// pass-through fragment shader
in vec4 color;
void main(void)
{
      gl_FragColor = color;
}
```

Fragment Shader Applications (1)

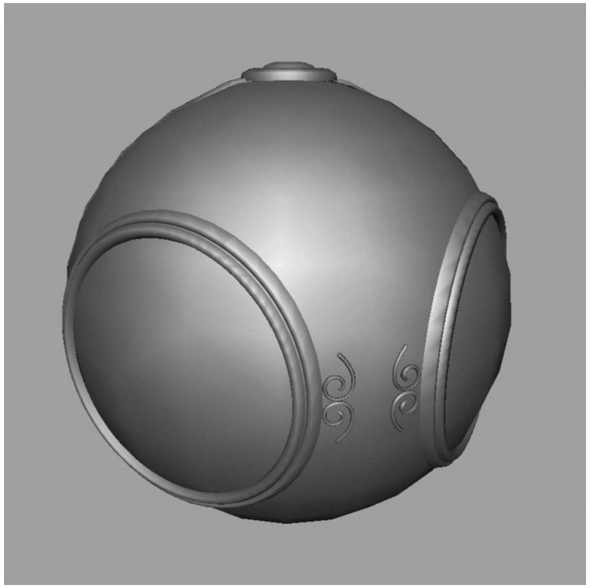
Per fragment lighting calculations

per vertex lighting

per fragment lighting

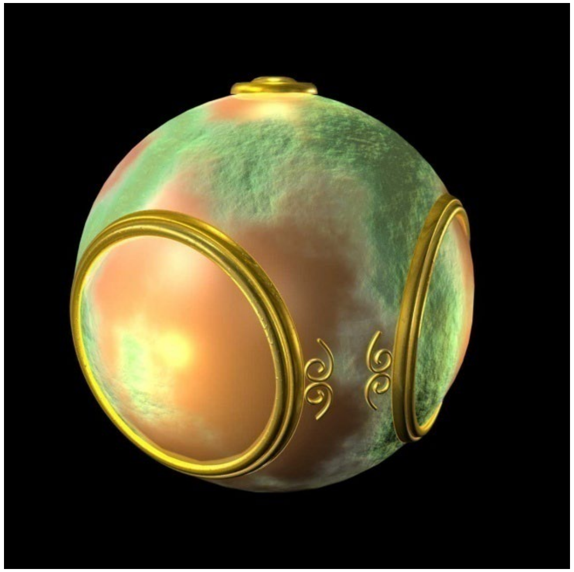
Fragment Shader Applications (2)

Texture mapping



smooth shading

environment mapping



bump mapping

Programming with OpenGL: A Complete Program with Shaders

Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

- Build a complete shader-based program
  - Application program (C++)
  - Shaders (GLSL)
    - Vertex shaders
    - Fragment shaders
  - Introduce a standard program structure
- Simple viewing
  - Two-dimensional viewing as a special case of three-dimensional viewing
- Initialization steps and program structure

OpenGL Programming in a Nutshell

- Modern OpenGL programs essentially do the following steps:
  - Create shader programs
  - Create buffer objects and load data into them
  - “Connect” data locations with shader variables
  - Render

Application Program Structure

- Most OpenGL programs have a similar structure that consists of the following functions
  - **`main()`**:
    - Opens main window with control panel and OpenGL canvas
    - Enters event loop (last executable statement)
  - **`initializeGL()`**: sets the state variables
    - Viewing
    - Attributes
  - **`resizeGL()`**: handles window resizing event
    - Sets viewport
    - Sets viewing coordinates for orthographic or perspective projection
  - **`paintGL ()`**: render scene
    - Clear framebuffer
    - Call glDrawArrays() to pump vertices to vertex shader

paintGL()

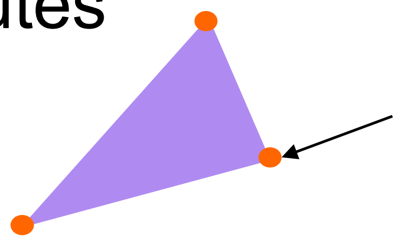
- Key issue is that we must form a data array to send to GPU and then render it
- Once we get data to GPU, we can initiate the rendering with a call to paintGL()

```
void paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

- Arrays are buffer objects that contain vertex arrays

Representing Geometric Objects

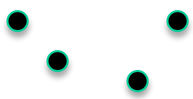
- Geometric objects are represented using *vertices*
- A vertex is a collection of generic attributes
  - positional coordinates
  - colors
  - texture coordinates
  - any other data associated with that point in space
- Position stored in 4D homogeneous coordinates
- Vertex data must be stored in vertex buffer objects (VBOs)
- VBOs may be stored in vertex array objects (VAOs)



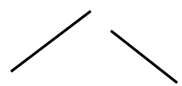
$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

OpenGL's Geometric Primitives

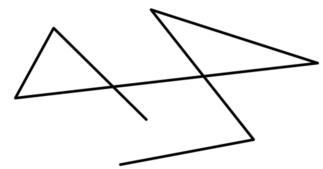
- All primitives are specified by vertices



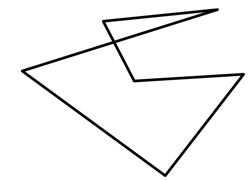
GL_POINTS



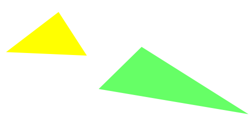
GL_LINES



GL_LINE_STRIP



GL_LINE_LOOP

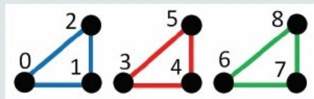
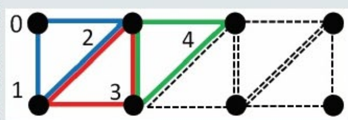
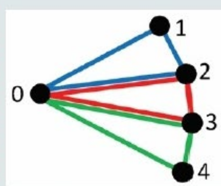
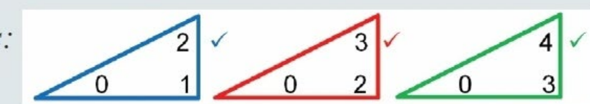


GL_TRIANGLES

GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN

OpenGL's Geometric Primitives

GL_TRIANGLES 	The most common primitive type in this book. Vertices that pass through the pipeline form distinct triangles: *vertices:* [0 1 2] [3 4 5] [6 7 8] etc. *triangles:* ✓ ✓ ✓
GL_TRIANGLE_STRIP 	Each vertex that passes through the pipeline efficiently forms a triangle with the previous two vertices: *vertices:* 0 1 2 3 4 etc. *triangles:* ✓ ✓ ✓
GL_TRIANGLE_FAN 	Each pair of vertices that passes through the pipe-line forms a triangle with the very first vertex: *vertices:* 0 1 2 3 4 etc. *triangles:* 
GL_LINES 	Vertices that pass through the pipeline form distinct lines: *vertices:* [0 1] [2 3] [4 5] etc. *lines:* ✓ ✓ ✓
GL_LINE_STRIP 	Each vertex that passes through the pipeline efficiently forms a line with the previous vertex: *vertices:* 0 1 2 3 etc. *lines:* ✓ ✓ ✓

Vertex Arrays

- Vertices can have many attributes
  - Position
  - Color
  - Texture Coordinates
  - Application data
- A vertex array holds these data
- Using QVector2D types

```
typedef QVector2D vec2;
vec2 points[3] = {
       vec2(0.0, 0.0), vec2(0.0, 1.0), vec2(1.0, 1.0)
};
```

Vertex Array Object

- Bundles all vertex data (positions, colors, ..,)
- Get name for buffer then bind

```
Glunit abuffer;
glGenVertexArrays(1, &abuffer);
glBindVertexArray(abuffer);
```

- At this point we have a current vertex array but no contents
- Use of glBindVertexArray lets us switch between VBOs
- At least one VAO must be created whenever shaders are used, even if no VBOs are used in application.

Buffer Object

- Buffer objects allow us to transfer large amounts of data to the GPU
- Need to create, bind and identify data

```
Gluint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(points), points);
```

- Data in current vertex array is sent to GPU

Our First Program

- We'll render a cube with colors at each vertex
- Our example demonstrates:
  - initializing vertex data
  - organizing data for rendering
  - simple object modeling
    - building up 3D objects from geometric primitives
    - building geometric primitives from vertices

Initializing the Cube's Data (1)

- We'll build each cube face from individual triangles
- Need to determine how much storage is required
  - (6 faces)(2 triangles/face)(3 vertices/triangle)

```
const int NumVertices = 36;
```

- To simplify communicating with GLSL, we'll use a `vec4` class (implemented in C++) similar to GLSL's `vec4` type

Initializing the Cube's Data (2)

- Before we can initialize our VBO, we need to stage the data
- Our cube has two attributes per vertex
  - position
  - color
- We create two arrays to hold the VBO data

```
vec4  positions[NumVertices];
vec4  colors   [NumVertices];
```

Cube Data (1)

- Vertices of a unit cube centered at origin
  - sides aligned with axes

```
vec4 xyz[8] = {
    vec4( -0.5, -0.5,  0.5, 1.0 ),
    vec4( -0.5,  0.5,  0.5, 1.0 ),
    vec4(  0.5,  0.5,  0.5, 1.0 ),
    vec4(  0.5, -0.5,  0.5, 1.0 ),
    vec4( -0.5, -0.5, -0.5, 1.0 ),
    vec4( -0.5,  0.5, -0.5, 1.0 ),
    vec4(  0.5,  0.5, -0.5, 1.0 ),
    vec4(  0.5, -0.5, -0.5, 1.0 )
};
```

Cube Data (2)

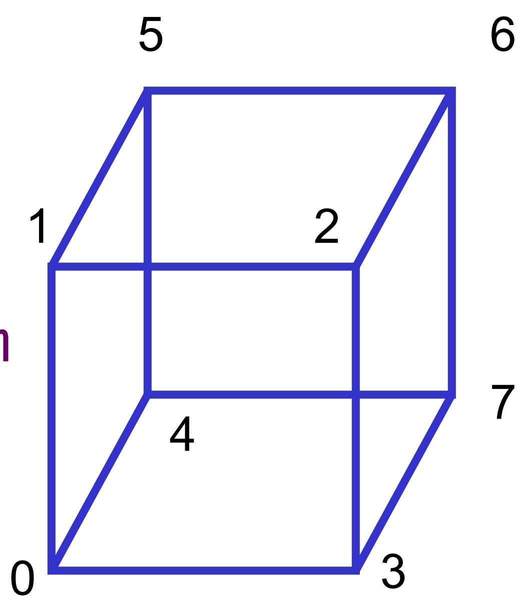
- We’ll also set up an array of RGBA colors

```
vec4 rgba[8] = {
    vec4( 0.0, 0.0, 0.0, 1.0 ),  // black
    vec4( 1.0, 0.0, 0.0, 1.0 ),  // red
    vec4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
    vec4( 0.0, 1.0, 0.0, 1.0 ),  // green
    vec4( 0.0, 0.0, 1.0, 1.0 ),  // blue
    vec4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
    vec4( 1.0, 1.0, 1.0, 1.0 ),  // white
    vec4( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
```

Generating the Cube from Faces

- Generate 12 triangles for the cube
  - 36 vertices with 36 colors

```
void colorcube()
{
    quad( 1, 0, 3, 2 );  // front
    quad( 2, 3, 7, 6 );  // right
    quad( 3, 0, 4, 7 );  // bottom
    quad( 6, 5, 1, 2 );  // top
    quad( 4, 5, 6, 7 );  // rear
    quad( 5, 4, 0, 1 );  // left
}
```



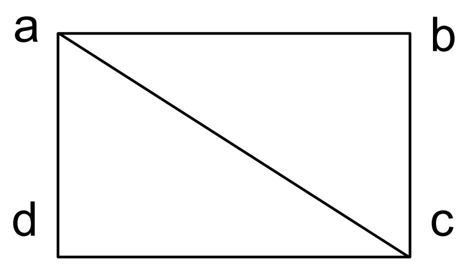
Vertices are ordered to obtain correct outward facing normals

Generating a Cube Face from Vertices

- To simplify generating the geometry, we use a convenience function `quad()`
  - create two triangles for each face and assign colors to the vertices

```
int Index = 0;  // global variable indexing into VBO arrays

void quad( int a, int b, int c, int d )
{
    colors[Index] = rgba[a]; positions[Index] = xyz[a]; Index++;
    colors[Index] = rgba[b]; positions[Index] = xyz[b]; Index++;
    colors[Index] = rgba[c]; positions[Index] = xyz[c]; Index++;
    colors[Index] = rgba[a]; positions[Index] = xyz[a]; Index++;
    colors[Index] = rgba[c]; positions[Index] = xyz[c]; Index++;
    colors[Index] = rgba[d]; positions[Index] = xyz[d]; Index++;
}
```



Storing Vertex Attributes

- Vertex data must be stored in a VBO
- Generate VBO names by calling `glGenBuffers()`
- Bind a specific VBO for initialization by calling

```
glBindBuffer( GL_ARRAY_BUFFER, … )
```

- load data into VBO using

```
glBufferData( GL_ARRAY_BUFFER, … )
```

VBOs in Code

- Create and initialize a buffer object

```
GLuint buffer;
glGenBuffers( 1, &buffer );
glBindBuffer( GL_ARRAY_BUFFER, buffer );
glBufferData( GL_ARRAY_BUFFER,
              sizeof(positions) + sizeof(colors),
              NULL, GL_STATIC_DRAW );
glBufferSubData( GL_ARRAY_BUFFER, 0,
                 sizeof(positions), positions );
glBufferSubData( GL_ARRAY_BUFFER, sizeof(vPositions),
                 sizeof(colors), colors );
```

Connecting Vertex Shaders with Geometric Data

- Application vertex data enters the OpenGL pipeline through the vertex shader
- Need to connect vertex data to shader variables
  - requires knowing the attribute location
- Attribute location can either be queried by calling `glGetVertexAttribLocation()`

Vertex Array Code

- Associate shader variables with vertex arrays
  - do this after shaders are loaded

```
GLuint a_Position =
    glGetAttribLocation( program, “a_Position" );
glEnableVertexAttribArray( a_Position );
glVertexAttribPointer( a_Position, 4, GL_FLOAT,
    GL_FALSE, 0, (void *) 0);

GLuint a_Color =
    glGetAttribLocation( program,“a_Color" );
glEnableVertexAttribArray( a_Color );
glVertexAttribPointer( a_Color, 4, GL_FLOAT,
    GL_FALSE, 0, (void *) sizeof(a_Positions));
```

A Better Approach

- Associate shader variables with attribute variables
  - do this after shaders are loaded

```
enum { ATTRIB_VERTEX, ATTRIB_COLOR, ATTRIB_TEXTURE_POSITION };
glBindAttribLocation(program, ATTRIB_VERTEX, “a_Position”);
glBindAttribLocation(program, ATTRIB_COLOR,  “a_Color”);

glEnableVertexAttribArray(ATTRIB_VERTEX);
glVertexAttribPointer(ATTRIB_VERTEX, 4, GL_FLOAT, false, 0, NULL);

glEnableVertexAttribArray(ATTRIB_COLOR);
glVertexAttribPointer(ATTRIB_COLOR, 4, GL_FLOAT,
       GL_FALSE, 0, (void *) sizeof(a_Positions));
```

Drawing Geometric Primitives

- For contiguous groups of vertices

```
glDrawArrays( GL_TRIANGLES, 0, NumVertices );
```

- Usually invoked in display callback
- Initiates vertex shader