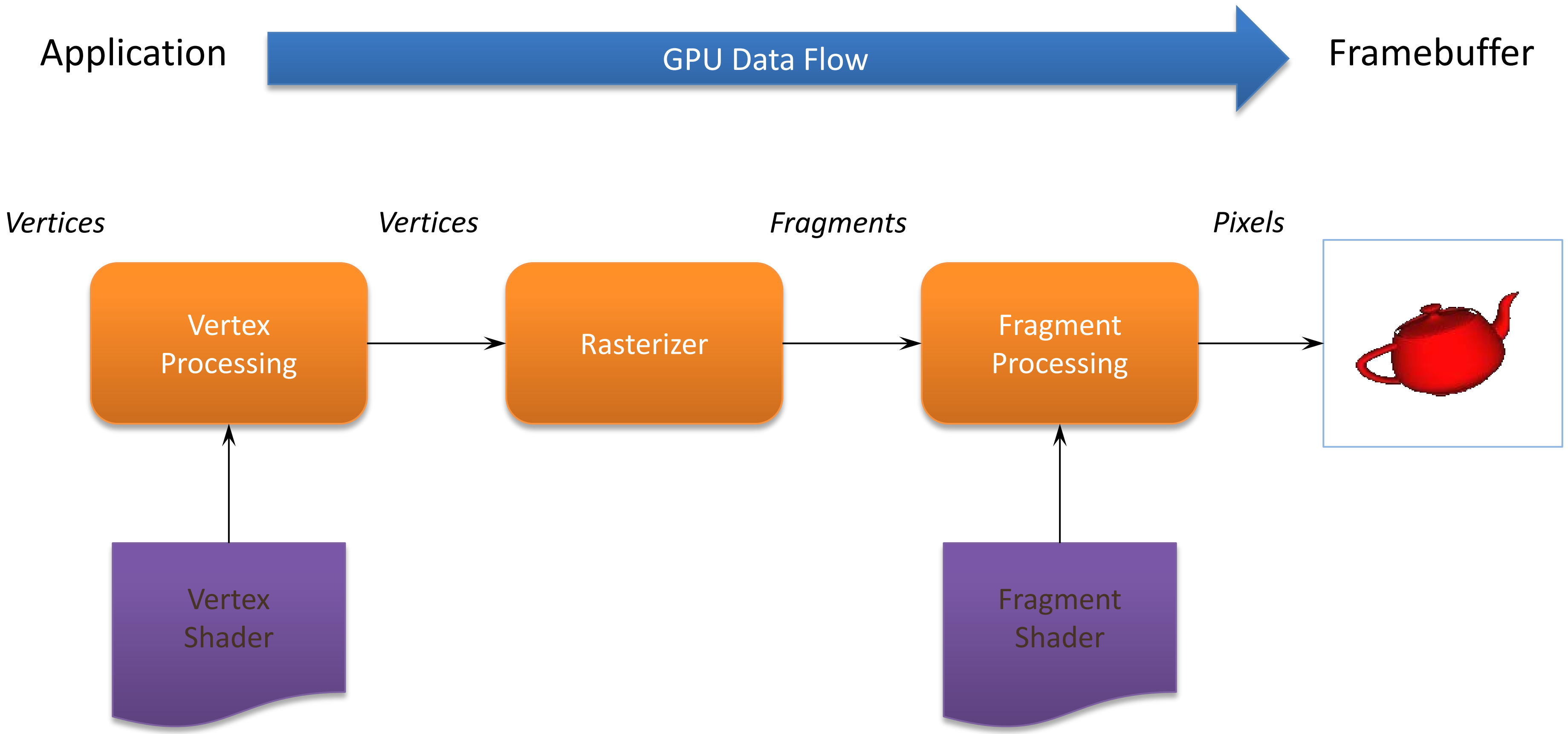
Introduction
A Simplified Pipeline Model

A Simplified Pipeline Model



A Simplified Pipeline Model

- Application will provide *vertices*, which are collections of data that are composed to form geometric objects, to the OpenGL pipeline.
- The *vertex processing* stage uses a vertex shader to process each vertex, doing any computations necessary to determine where in the frame buffer each piece of geometry should go.
- The other shading stages we mentioned – tessellation and geometry shading – are also used for vertex processing, but we're trying to keep this simple.
- After all the vertices for a piece of geometry are processed, the *rasterizer* determines which pixels in the frame buffer are affected by the geometry.
- and for each pixel, the *fragment processing* stage is employed, where the *fragment shader* runs to determine the final color of the pixel.

A Simplified Pipeline Model

- In your OpenGL applications, you’ll usually need to do the following tasks:
  - specify the vertices for your geometry
  - load vertex and fragment shaders (and other shaders, if you’re using them as well)
  - issue your geometry to engage the OpenGL pipeline for processing

OpenGL Programming

- 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 Framework Requirements

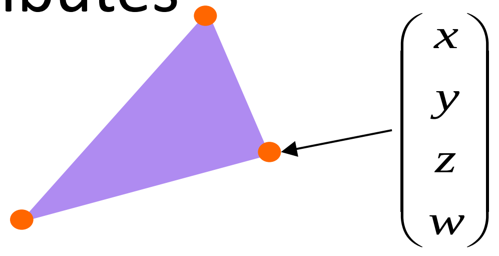
- OpenGL applications need a place to render into
  - usually an on-screen window
- Need to communicate with native windowing system
- Each windowing system interface is different
- We use GLUT (more specifically, freeglut)
  - simple, open-source library that works everywhere
  - handles all windowing operations:
    - opening windows
    - input processing

Simplifying Working with OpenGL

- Operating systems deal with library functions differently
  - compiler linkage and runtime libraries may expose different functions
- Additionally, OpenGL has many versions and profiles which expose different sets of functions
  - managing function access is cumbersome, and window-system dependent
- We use another open-source library, GLEW(the OpenGL Extension Wrangler library), to hide those details
  - (It removes all the complexity of accessing OpenGL functions, and working with OpenGL extensions).

Representing Geometric Objects

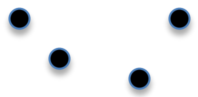
- Geometric objects are represented using *vertices*
- A vertex is a collection of generic attributes
  - positional coordinates
  - colors
  - texture coordinates
  - any other data associated with that point in space

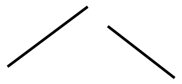


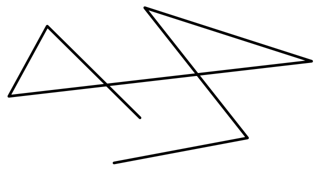
- Position stored in 4 dimensional homogeneous coordinates
- Vertex data must be stored in vertex buffer objects (VBOs)
- VBOs must be stored in vertex array objects (VAOs)

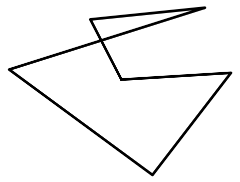
OpenGL's Geometric Primitives

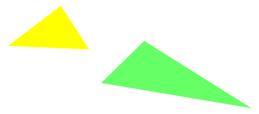
- All primitives are specified by vertices

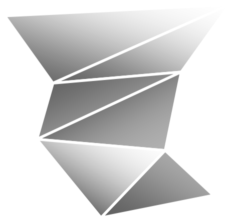

GL_POINTS

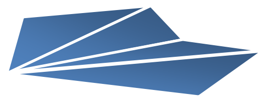

GL_LINES


GL_LINE_STRIP


GL_LINE_LOOP


GL_TRIANGLES


GL_TRIANGLE_STRIP


GL_TRIANGLE_FAN

A First Program: Cube

- 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

- 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
  - we'll also typedef it to add logical meaning

```
typedef  vec4  point4;
typedef  vec4  color4;
```

Initializing the Cube's Data (cont'd)

- A **Vertex Buffer Object** (VBO) is an OpenGL feature that provides methods for uploading vertex data (position, normal vector, color, etc.) to the video device for non-immediate-mode rendering.
- 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

```
point4  vPositions[NumVertices];
color4  vColors[NumVertices];
```

Cube data

- Vertices of a unit cube centered at origin
  - sides aligned with axes
- In our example we'll copy the coordinates of our cube model into a VBO for OpenGL to use. Here we set up an array of eight coordinates for the corners of a unit cube centered at the origin. We use a *homogenous coordinate*

```
point4 positions[8] = {
    point4( -0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5, -0.5, -0.5, 1.0 ),
    point4( -0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5, -0.5, -0.5, 1.0 )
};
```

Cube Data (cont'd)

- We'll also set up an array of RGBA colors
- we'll set up a matching set of colors for each of the model's vertices, which we'll later copy into our VBO. Here we set up eight RGBA colors. In OpenGL, colors are processed in the pipeline as floating-point values in the range [0.0, 1.0]. *normalizing* values.

```
color4 colors[8] = {
    color4( 0.0, 0.0, 0.0, 1.0 ),  // black
    color4( 1.0, 0.0, 0.0, 1.0 ),  // red
    color4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
    color4( 0.0, 1.0, 0.0, 1.0 ),  // green
    color4( 0.0, 0.0, 1.0, 1.0 ),  // blue
    color4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
    color4( 1.0, 1.0, 1.0, 1.0 ),  // white
    color4( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
```

Generating a Cube Face from Vertices

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

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

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

Generating the Cube from Faces

- Generate 12 triangles for the cube
  - 36 vertices with 36 colors
  - generation of our cube's VBO data by specifying the six faces using index values into our original `positions` and `colors` arrays. It's worth noting that the order that we choose our vertex indices is important, as it will affect something called *backface culling* later.

```
void colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```

Vertex Array Objects (VAOs)

- VAOs store the data of an geometric object
- Steps in using a VAO
  - generate VAO names by calling `glGenVertexArrays()`
  - bind a specific VAO for initialization by calling `glBindVertexArray()`
  - update VBOs associated with this VAO
  - bind VAO for use in rendering
- This approach allows a single function call to specify all the data for an objects
  - previously, you might have needed to make many calls to make all the data current
- Code: Create a vertex array object

```
GLuint vao;
glGenVertexArrays( 1, &vao );
glBindVertexArray( vao );
```

Storing Vertex Attributes

- Vertex data must be stored in a VBO, and associated with a VAO
- The code-flow is similar to configuring a VAO
  - 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, … )
    ```

  - bind VAO for use in rendering `glBindVertexArray()`

VBOs in Code

- Create and initialize a buffer object

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

Connecting Vertex Shaders with Geometric Data

- The final step in preparing you data for processing by OpenGL (i.e., sending it down for rendering) is to specify which vertex attributes you'd like issued to the graphics pipeline. While this might seem superfluous, it allows you to specify multiple collections of data, and choose which ones you'd like to use at any given time.
- Each of the attributes that we enable must be associated with an "in" variable of the currently bound vertex shader. You retrieve vertex attribute locations was retrieved from the compiled shader by calling `glGetAttribLocation().`
- 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 vPosition =
   glGetAttribLocation( program, “vPosition" );
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT,
   GL_FALSE, 0,BUFFER_OFFSET(0) );

GLuint vColor =
   glGetAttribLocation( program,"vColor" );
glEnableVertexAttribArray( vColor );
glVertexAttribPointer( vColor, 4, GL_FLOAT,
   GL_FALSE, 0, BUFFER_OFFSET(sizeof(vPositions))
);
```

Drawing Geometric Primitives

- In order to initiate the rendering of primitives, you need to issue a drawing routine. The simplest routine is `glDrawArrays()`, to which you specify what type of graphics primitive you want to draw (e.g., here we're rending a triangle strip), which vertex in the enabled vertex attribute arrays to start with, and how many vertices to send.

- This is the simplest way of rendering geometry in OpenGL Version 3.1. You merely need to store you vertex data in sequence, and then `glDrawArrays()` takes care of the rest. However, in some cases, this won't be the most memory efficient method of doing things.
- For contiguous groups of vertices

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

- Usually invoked in display callback
- Initiates vertex shader