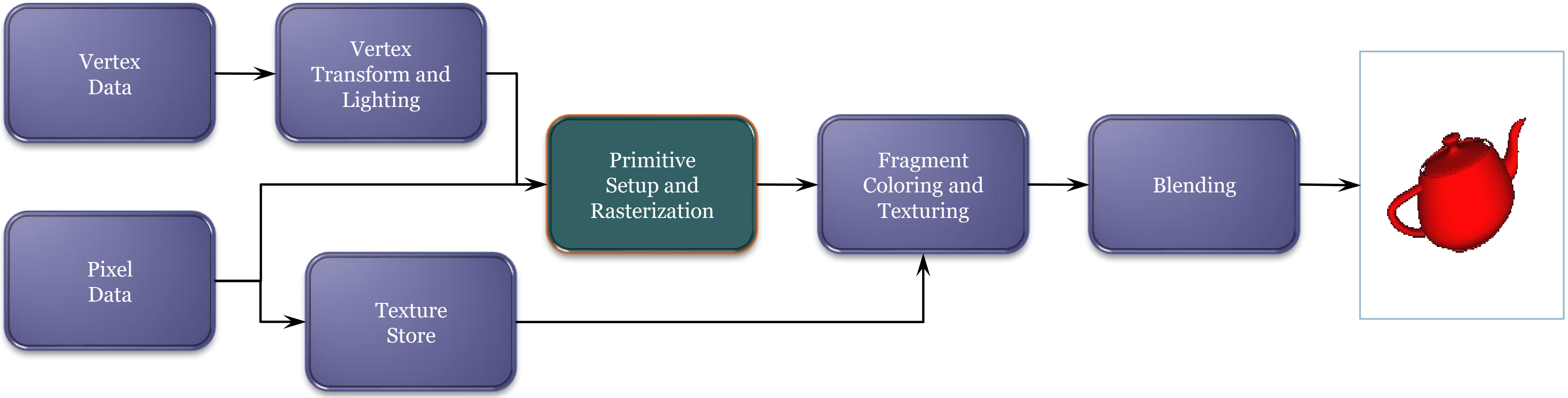
OpenGL
II –Shader based Pipeline

Stefan BORNHOFEN

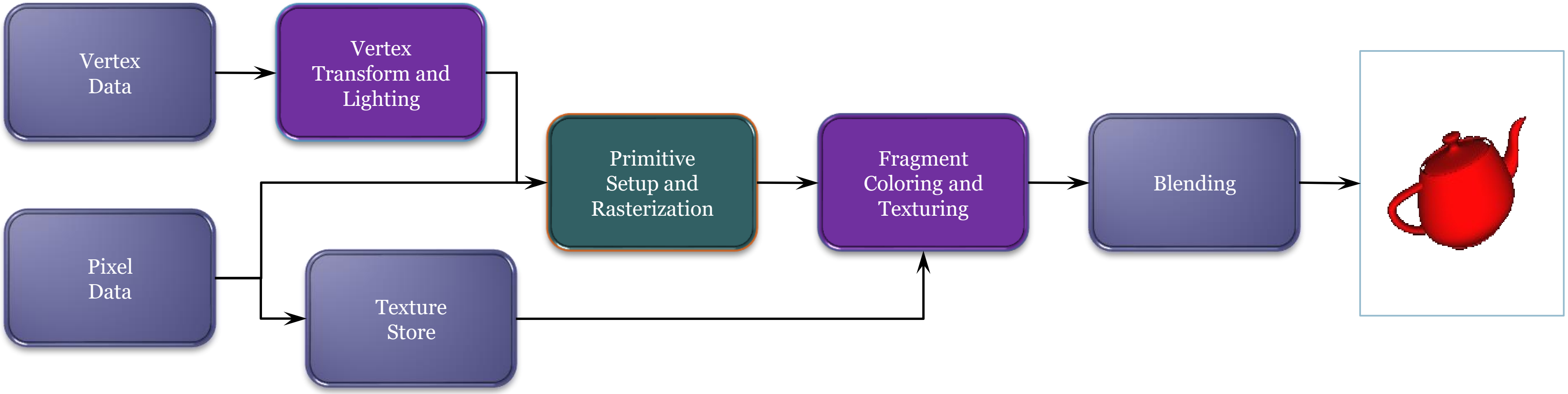
Evolution of OpenGL

- OpenGL 1.x (1994) has a *fixed function pipeline*
- An application can only change a set of input values (colors, positions, etc.).



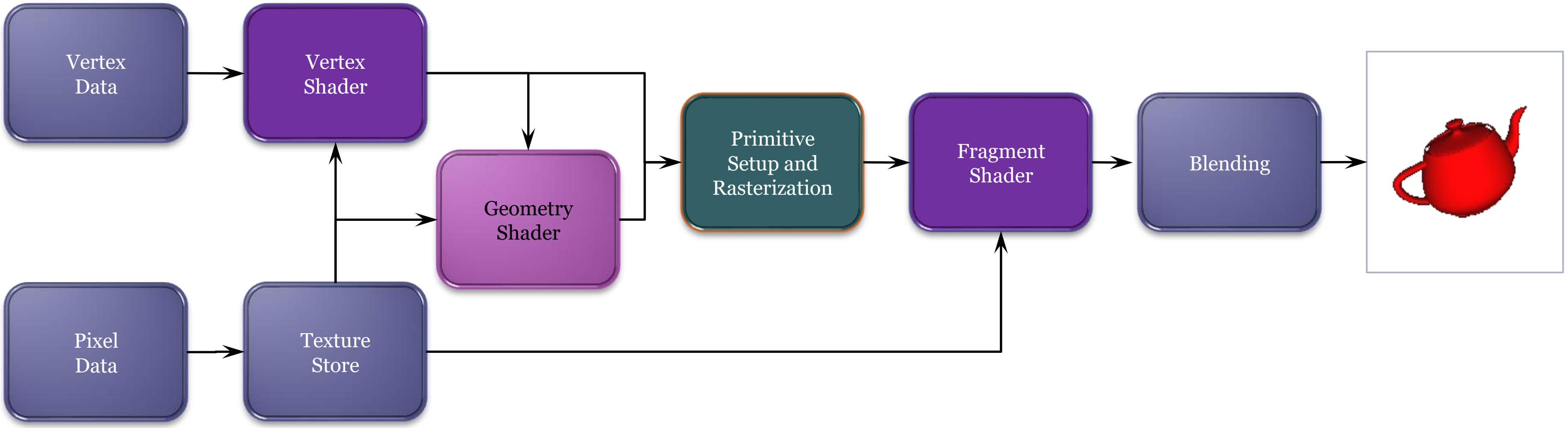
Evolution of OpenGL

- OpenGL 2.x (2004) added programmable shaders
  - *vertex shaders* for the transform and lighting stage
  - *fragment shaders* for the fragment coloring stage
- OpenGL Shaders are written in GLSL
- The fixed-function pipeline was still available



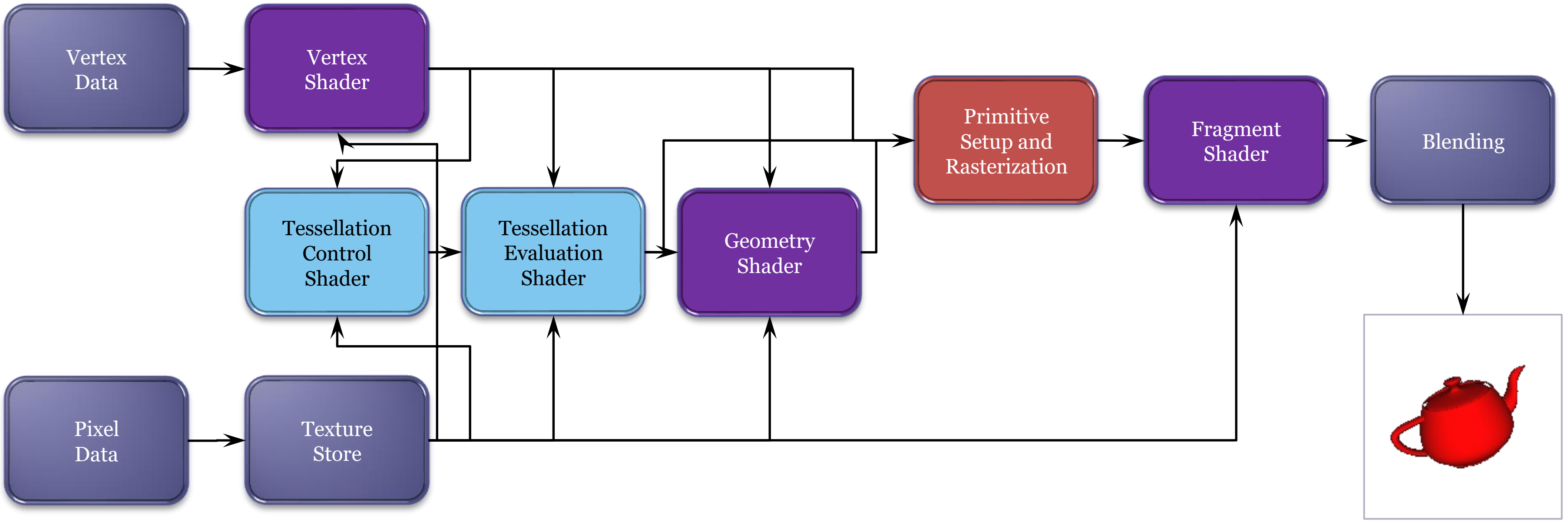
Evolution of OpenGL

- OpenGL 3.x (2009) removed the fixed-function pipeline
- All programs were required to use shaders
- Almost all data is GPU-resident (using buffer objects)
- An additional shading stage was introduced: geometry shaders to modify geometric primitives within the graphics pipeline



Evolution of OpenGL

- OpenGL 4.1 (2010) included two additional shading stages: tessellation-control and tessellation-evaluation shaders
- Latest version is OpenGL 4.6 (2017)



Evolution of OpenGL

- OpenGL ES (2003)
  - A somewhat subsetted version of OpenGL Designed for embedded and hand-held devices such as cell phones
- WebGL (2013)
  - JavaScript implementation of ES 2.0
  - Runs on most browsers
  - Only vertex and fragment shaders

Recap

- Fixed-function graphics operations, like vertex lighting and transformations are today **deprecated**
- All modern OpenGL applications use **shaders** for their graphics processing

More effort for the programmer,

but also more control.

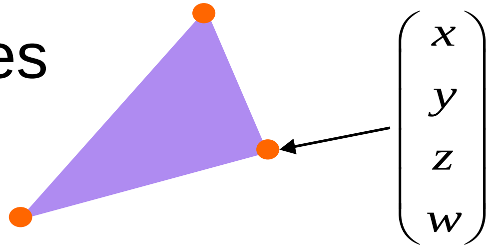
Modern OpenGL in a Nutshell

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

Geometric objects

- Geometric objects are represented using *vertices*

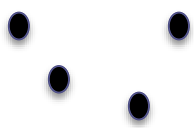
- A vertex is a collection of generic attributes
  - positional coordinates (x,y,z,w)
  - colors (r,g,b,a)
  - texture coordinates (u,v or s,t)
  - any other data associated with that point in space



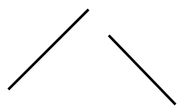
- Vertex data is stored in the GPU memory as **vertex buffer objects** (VBOs)
- VBOs are stored in the GPU memory as **vertex array objects** (VAOs)

Geometric Primitives

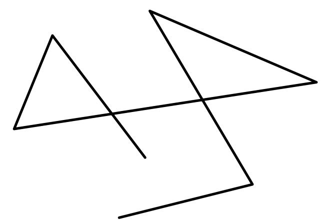
- All primitives are specified by vertices
- OpenGL only knows how to draw points, lines, and triangles



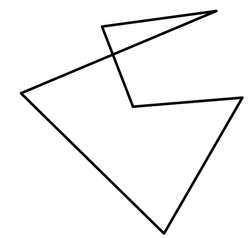
GL_POINTS



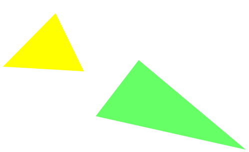
GL_LINES



GL_LINE_STRIP

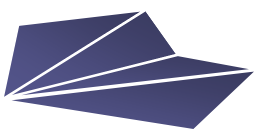


GL_LINE_LOOP



GL_TRIANGLES

GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN

Vertex and Fragment shaders

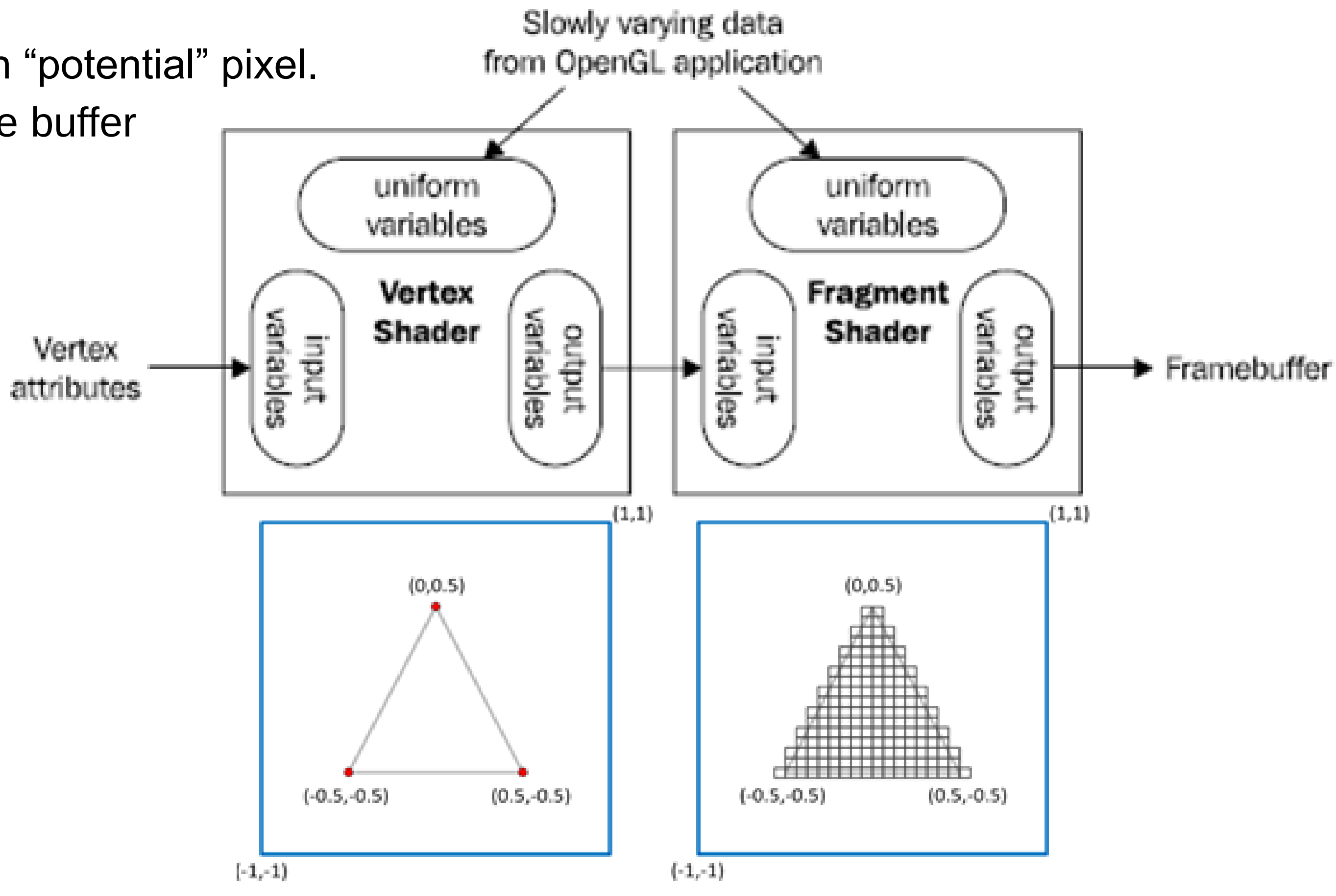
Shaders are written in GLSL (OpenGL Shading Language)

Vertex shader

- Executed for each vertex
- Outputs a position in device space, as well as useful data for the fragment shader

Fragment shader

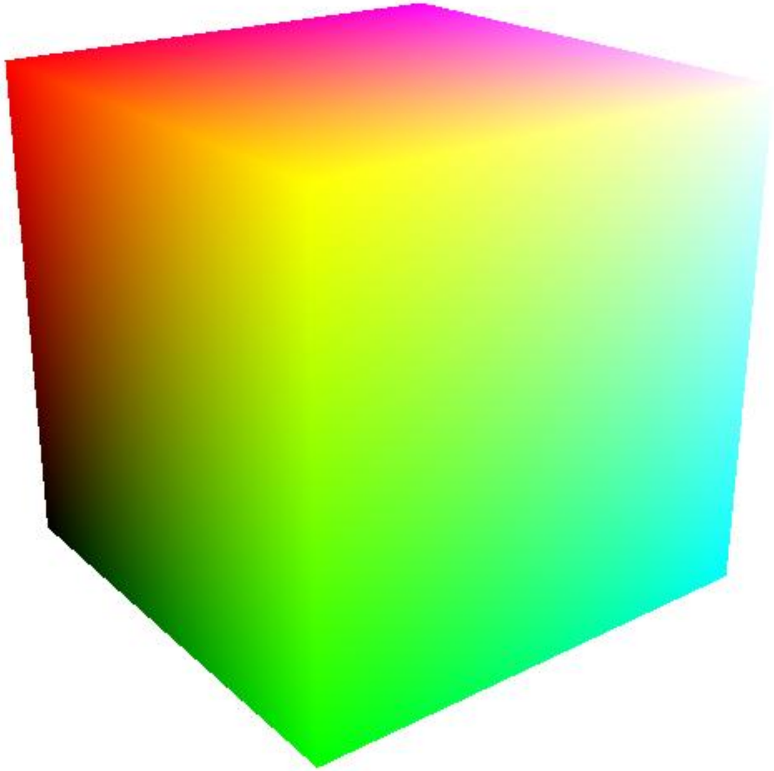
- Executed for each “potential” pixel.
- Updates the frame buffer



Hello OpenGL

Objective:

A rotating cube with different colors at each vertex.



GLEW

- Any functionality beyond OpenGL1.1 must be accessed through the OpenGL extension mechanism
- GLEW is an open-source library to access these extensions.
- At the same time, it hides a lot of OS-specific problems.

Exercise

Create a new project using GLUT
Install GLEW
Follow the GLEW part of the document
« Setting up OpenGL, GLUT & GLEW for Visual C++ in Windows »

and try to initialize it:

```
#include <GL/glew.h>
#include <GL/glut.h>

...

void main(int argc, char** argv)
{
         ... init GLUT ...

         GLenum err = glewInit();
         if (GLEW_OK != err)
         {
           /* glewInit failed*/
           fprintf(stderr, "Error: %s\n", glewGetErrorString(err));
           exit(EXIT_FAILURE);
         }
         fprintf(stdout, "Using GLEW %s\n", glewGetString(GLEW_VERSION));

         glutMainLoop();
}
```

GLM

- “OpenGL Mathematics“
- A C++ mathematics library for graphics software based on the GLSL specification
- Provides classes and functions designed and implemented with the same naming conventions and functionalities than GLSL

Exercise

- Install glm: http://glm.g-truc.net/0.9.9

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
...
glm::vec4 example_vertex;
glm::mat4 example_matrix;
```

Define Vertices

A unit cube centered at origin aligned with axes.

```
glm::vec4 vertices[8] = {
  {-0.5,-0.5, 0.5, 1.0 },
  {-0.5, 0.5, 0.5, 1.0 },
  { 0.5, 0.5, 0.5, 1.0 },
  { 0.5,-0.5, 0.5, 1.0 },
  {-0.5,-0.5,-0.5, 1.0 },
  {-0.5, 0.5,-0.5, 1.0 },
  { 0.5, 0.5,-0.5, 1.0 },
  { 0.5,-0.5,-0.5, 1.0 }
};
```

Define Colors

An array of RGBA colors.

```
glm::vec4 colors[8] = {
  { 1.0, 0.0, 0.0, 1.0 },  // red
  { 1.0, 1.0, 0.0, 1.0 },  // yellow
  { 0.0, 1.0, 0.0, 1.0 },  // green
  { 0.0, 0.0, 1.0, 1.0 },  // blue
  { 1.0, 0.0, 1.0, 1.0 },  // magenta
  { 0.0, 1.0, 1.0, 1.0 },  // cyan
  { 0.0, 0.0, 0.0, 1.0 },  // black
  { 1.0, 1.0, 1.0, 1.0 }   // white
};
```

Define the Cube

```
vector<glm::vec4> vPositions;
vector<glm::vec4> vColors;

void colorCube()
{
    // define 12 triangles for the cube
    // = 36 positions and 36 colors
    vPositions.push_back(...);
    vColors.push_back(...);
    ...
}
```

Define Vertex Array object (VAO)

- VAOs store the data of a geometric object on the GPU
- Allows a single function call to make all the data of an object current.

```
GLuint vaoCube;
...

init section
glGenVertexArrays(1, &vaoCube);
glBindVertexArray(vaoCube);

...
cleanup section
glDeleteVertexArrays(1, &vaoCube);
```

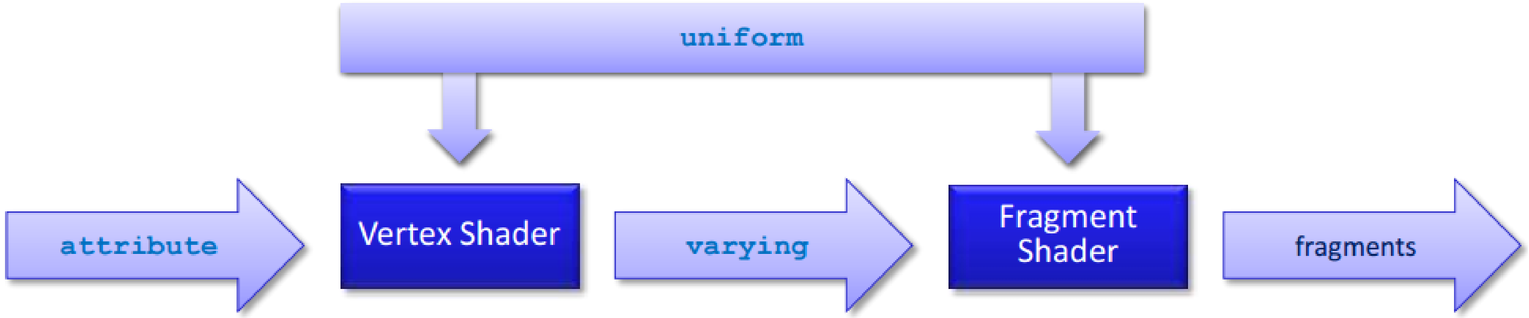
Define Vertex Buffer object (VBO)

Contains vertex data (position, normal vector, color, etc.) for non-immediate-mode rendering.

```
GLuint vboCube;
...

init section
glGenBuffers(1, &vboCube);
glBindBuffer(GL_ARRAY_BUFFER, vboCube);

int sp = vPositions.size()*sizeof(glm::vec4);
int sc = vColors.size()*sizeof(glm::vec4);

glBufferData(GL_ARRAY_BUFFER, sp + sc, NULL, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER,0,sp, &vPositions[0]); //load
glBufferSubData(GL_ARRAY_BUFFER, sp, sc, &vColors[0]); // load

...
cleanup section
glDeleteBuffers(1, &vboCube);
```

GLSL Variable Qualifiers

- **Attributes** (“`in`”) are the inputs into vertex shaders
- **Varyings** (“`out`”) outputs of vertex shaders
  and inputs into fragment shaders
- **Uniform** (‘`uniform`”) are constants available to any shader stage



Built-in Variables

- `gl_Position`
  - Output: position computed by the vertex shader
- `gl_FragCoord`
  - Input: window-relative coordinates of the fragment
- `gl_FragDepth`
  - Input: depth value in fragment shader

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);
```

GLSL Operators & Functions

- Standard C/C++ arithmetic and logic operators
- Overloaded operators for matrix and vector operations

```
mat4 m;
vec4 a, b, c;
b = a*m;
c = m*a;
```

- Built in functions
  - Arithmetic: `sqrt, power, abs, ...`
  - Trigonometric: `sin, asin, ...`
  - Vector/Matrix: `length, reflect, ...`
- User defined functions

GLSL Components and Swizzling

- Access vector components
  - [ ] (c-style array indexing)
  - `xyzw`, `rgba` or `strq` (named components)

```
vec3 v;
v[1], v.y, v.g, v.t
```
: all refer to the same element

- Swizzling

```
vec3 a, b;
a.xy = b.yx;
vec4 c = a.xyxx;
```

A Simple Vertex Shader

```
// File: “passthrough.vert"
// Caution: Use UNIX EOL-format

#version 430
in vec4 vPosition;
in vec4 vColor;
out vec4 color;

void main()
{
    color = vColor;
    gl_Position = vPosition;
}
```

A Simple Fragment Shader

```
// File: "passthrough.frag"
// Caution: Use UNIX EOL-format

#version 430

in vec4 color;
out vec4 fColor; // final color

void main()
{
    fColor = color;
}
```

Note: if the colors across the geometric primitive are not the same, the rasterizer will interpolate those colors across the primitive, passing each iterated value into the `color` variable.

How to make the shaders work

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

Create Program	`glCreateProgram()`
Create Shader	`glCreateShader()`
Load Shader Source	`glShaderSource()`
Compile Shader	`glCompileShader()`
Attach Shader to Program	`glAttachShader()`
Link Program	`glLinkProgram()`
Use Program	`glUseProgram()`

These steps need to be repeated for each type of shader in the shader program

```
GLchar* readShaderSource(const char * shaderFile)
{
      FILE* fp;
      fopen_s(&fp, shaderFile, "r");
      GLchar* buf;
      long size;

      if (fp == NULL) return NULL;
      fseek(fp, 0L, SEEK_END);//go to end
      size = ftell(fp);       //get size
      fseek(fp, 0L, SEEK_SET);//go to beginning

      buf = (GLchar*)malloc((size+1)*sizeof(GLchar));
      fread(buf, 1, size, fp);
      buf[size] = 0;
      fclose(fp);
      return buf;
}
```

```
// Create a GLSL program object from vertex and fragment shader files
GLuint initShaders(const char* vShaderFile, const char* fShaderFile)
{
	struct Shader {
	const char*  filename;
	GLenum       type;
	GLchar*      source;
	}  shaders[2] = {
	{ vShaderFile, GL_VERTEX_SHADER, NULL },
	{ fShaderFile, GL_FRAGMENT_SHADER, NULL }
	};

	GLuint program = glCreateProgram();

	for (int i = 0; i < 2; ++i) {
	  Shader& s = shaders[i];
	  s.source = readShaderSource(s.filename);
	  if (shaders[i].source == NULL) {
	    printf("Failed to read %s\n", s.filename);
	    exit(EXIT_FAILURE);
	  }

	  GLuint shader = glCreateShader(s.type);
	  glShaderSource(shader, 1, (const GLchar**)&s.source, NULL);
	  glCompileShader(shader);
	  GLint  compiled;
	  glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);
	  if (!compiled) {
	    printf("%s failed to compile:\n", s.filename);
	    GLint  logSize;
	    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &logSize);
	    char* logMsg = new char[logSize];
	    glGetShaderInfoLog(shader, logSize, NULL, logMsg);
	    printf("%s\n", logMsg);
	    delete[] logMsg;
	    exit(EXIT_FAILURE);
	  }
	  delete[] s.source;

	  glAttachShader(program, shader);
	}

	/* link  and error check */
	glLinkProgram(program);

	GLint  linked;
	glGetProgramiv(program, GL_LINK_STATUS, &linked);
	if (!linked) {
	  printf("Shader program failed to link:\n");
	  GLint  logSize;
	  glGetProgramiv(program, GL_INFO_LOG_LENGTH, &logSize);
	  char* logMsg = new char[logSize];
	  glGetProgramInfoLog(program, logSize, NULL, logMsg);
	  printf("%s\n", logMsg);
	  delete[] logMsg;
	  exit(EXIT_FAILURE);
	}

	/* use program object */
	glUseProgram(program);

	return program;
}
```

```
GLuint passthrough;
passthrough =
initShaders("passthrough.vert",
"passthrough.frag");
```

Shader Plumbing

```
#define BUFFER_OFFSET(offset) ((GLvoid*)(offset))

// Connect shader variables to VBO
// Do this after the shaders are loaded.

GLuint vPosition =
    glGetAttribLocation(passthrough, "vPosition");
glEnableVertexAttribArray(vPosition);
glVertexAttribPointer(vPosition, 4, GL_FLOAT,
    GL_FALSE, 0,BUFFER_OFFSET(0));

int sp = vPositions.size()*sizeof(glm::vec4);
GLuint vColor =
    glGetAttribLocation(passthrough, "vColor");
glEnableVertexAttribArray(vColor);
glVertexAttribPointer(vColor, 4, GL_FLOAT,
    GL_FALSE, 0, BUFFER_OFFSET(sp));
```

Finishing the Cube Program

```
int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
    glutInitWindowSize(SCREEN_X, SCREEN_Y);
    glutCreateWindow(TITLE);
    glutSetOption(GLUT_ACTION_ON_WINDOW_CLOSE,
           GLUT_ACTION_GLUTMAINLOOP_RETURNS);

    init(); // containing all preparations seen above

    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    cleanup();

    return 0;
}
```

GLUT Callbacks

```
void display()
{
      glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
      glUseProgram(passthrough); // make shader current
      glBindVertexArray(vaoCube); // make vao current
      // initiate rendering:
      // the current shader program draws the current VAO
      glDrawArrays(GL_TRIANGLES, 0, vPositions.size());
      glutSwapBuffers();
}

void keyboard(unsigned char key, int x, int y)
{
    switch(key) {
        case 27: case 'q': case 'Q':
            exit(EXIT_SUCCESS);
            break;
    }
}
```

Here is the cube!

Communication OpenGL => Shaders

One way communication, three “channels”.

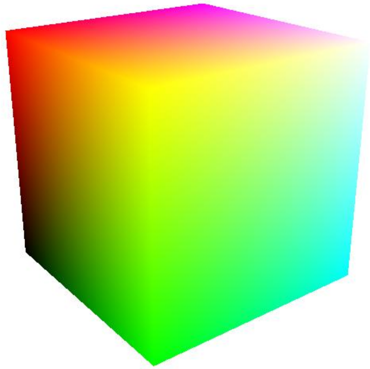
1) The shaders have access to part of the OpenGL state (e.g. light color). When an application alters this subset of the OpenGL state, it is, in a way, communicating with the shaders.
2) User defined variables in GLSL code:
   - Per vertex attributes (“in”)
   - Constant values (”uniform”)
3) Textures. A texture does not have to represent an image, it can be interpreted as an array of data.

Vertex Shader using MVP

```
#version 430
in vec4 vPosition;
in vec4 vColor;
uniform mat4 MVP; // Projection*View*Model
out vec4 color;

void main()
{
    color = vColor;
    gl_Position = MVP*vPosition;
}
```

Sending MVP from Application

Update MVP in the motion callback.



```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

void updateMVP() {
      glm::mat4 Projection = glm::perspective(...);
      glm::mat4 View = glm::lookAt(...);
      glm::mat4 Model = ... // translate, rotate, scale
      glm::mat4 MVP = Projection * View * Model;
      glUniformMatrix4fv(
            glGetUniformLocation(passthrough, "MVP"),
            1, GL_FALSE, &MVP[0][0]);
}
```

Exercise: Rotate the cube by mouse drag.

Rendering multiple objects

Modify your GLUT display method.

```
for each object
{
      bind shader
      bind VAO
      compute appropriate MVP matrix
      update uniform MVP shader variable
      call OpenGL draw method
}
```

Exercise

(Shader plumbing according to different data structures)

- Create a second cube with uniformly colored faces. Use a single VBO, but this time vertex positions and colors are interlaced.
- Create a B&W octahedron. This time use two separate VBO, one for the positions, one for the colors.