Shaders and OpenGL Shading Language

Professor Raymond Zavodnik

Tbilisi State University April 9, 2016

Why Shaders and OpenGL Shading Language(GLSL)?

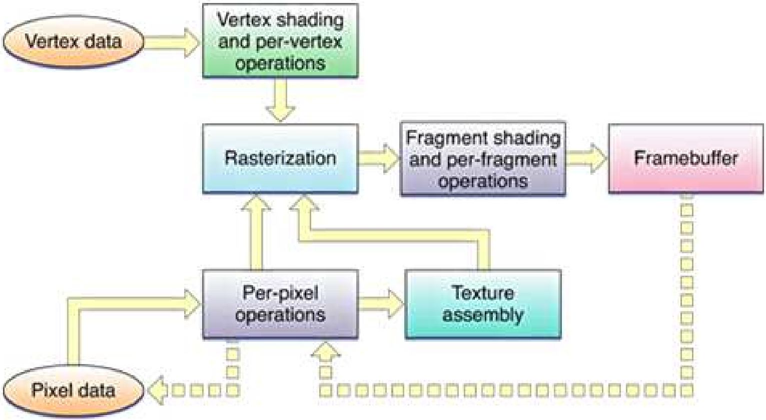
1. OpenGL is by design a static API that offers graphics applications a fixed functionality
2. Until recently the only way to modify the API was to extend it
    - Extensions usually are there to support new graphics hardware
    - Today almost all graphics hardware is programmable
    - Vendors today must offer programmable graphics architectures
3. Over the years more than 400 extensions have been implemented by the ARB!

Why Shaders and OpenGL Shading Language(GLSL)?

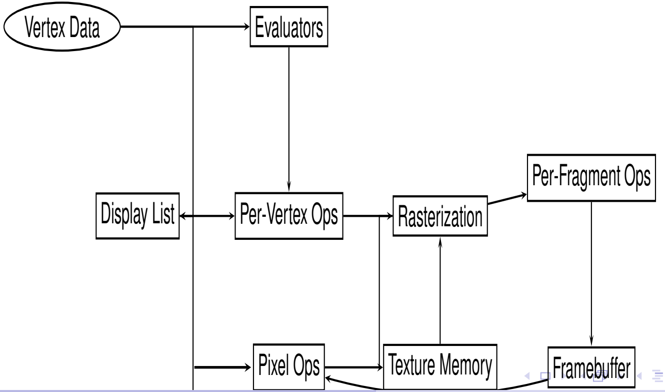
1. OpenGL is by design a static API that offers graphics applications a fixed functionality
2. Until recently the only way to modify the API was to extend it
    - Extensions usually are there to support new graphics hardware
    - Today almost all graphics hardware is programmable
    - Vendors today must offer programmable graphics architectures
3. Over the years more than 400 extensions have been implemented by the ARB!

Why Shaders and OpenGL Shading Language(GLSL)?

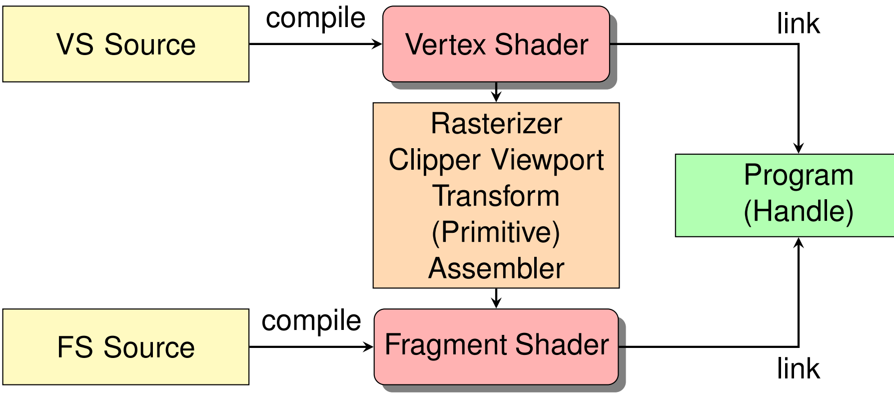
1. OpenGL is by design a static API that offers graphics applications a fixed functionality
2. Until recently the only way to modify the API was to extend it
    - Extensions usually are there to support new graphics hardware
    - Today almost all graphics hardware is programmable
    - Vendors today must offer programmable graphics architectures
3. Over the years more than 400 extensions have been implemented by the ARB!


Vertex data
Vertex shading and per-vertex operations
Rasterization
Fragment shading and per-fragment operations
Framebuffer
Per-pixel operations
Texture assembly
Pixel data

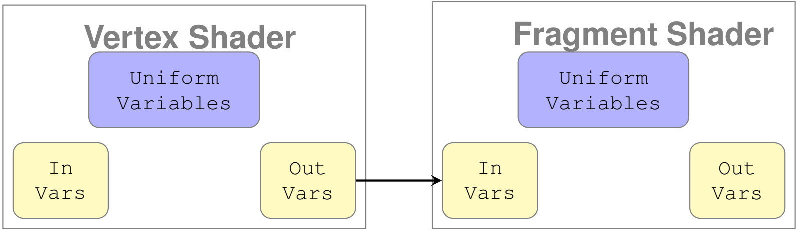
OpenGL Pipeline (Simplified)



Basic Shader Pipeline



Shader Components



Remarks

- Input for VS In Variables: Vertex Attributes from GPU
- Output for FS Output Variables: Framebuffer (for rendering)
- Role of Uniform Variables for all shaders: Share Data with main OpenGL program
- Output Vars of VS must match Input Vars of FS

Graphics Processing Pipeline

- Graphics operations in OpenGL happen in a (more or less) fixed order
- This functionality has remained fixed
  - **Per Vertex Operations**: Single vertices and their associated attributesare processed independently of one another
  - **Primitive Assembly**: Vertices are collected into graphic primitives
  - **Primitive Processing**: Primitives are clipped and perspective-projected
  - **Rasterization**: Primitives are decomposed into rasterizable units, i.e. pixels in the framebuffer. These units are called *Fragments*

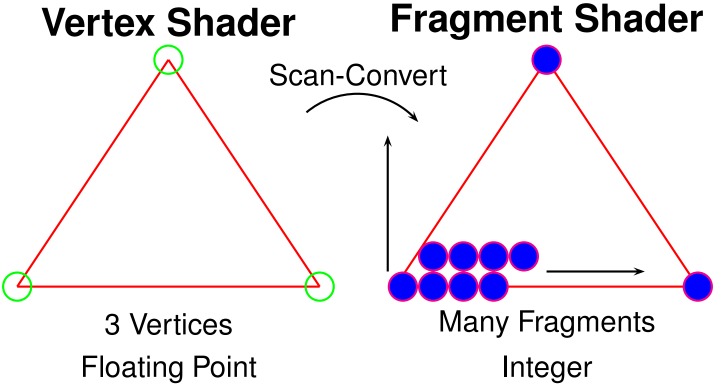
Graphics Processing Pipeline

- **Fixed Functionality Continued**
  - **Fragment Processing**: Textures, etc are combined with fragments
  - **Per-Fragment Operations**: Determine pixel visibiity, execute alpha, stencil, scissor tests
  - **Framebuffer Operations**: Determine buffers of pixel data and output them
- **The Bad News**: All of this fixed functionality has been deprecated!
- **The Good News**: The Graphics Pipeline is retained, the individual components being replaced by programmable units called *shaders*

Going From Vertices to Fragments

- 3 Floating Point Vertices are mapped to 3 Fragments
- These three Fragments are interpolated to the interior of the triangle
- Horizontally affinely interpolate from the edges, starting with the lowest vertex or side
- Vertically bump the incremented y-value by affinely interpolating between lower and higher vertex
- At any point the input variable of the Fragment Shader contains the currently scan-converted fragment (or simply, pixel)

Going from Vertices to Fragments



Compatibility Issues

- **Problem**: What about applications that use the fixed functionality?
- **Constraint**: They must run as before
- **Solution**: Although deprecated, the old features become **extensions** that are part of a complicated hierarchy of libraries that can be called apart from the new core functionality
- OpenGL3.0 introduced the *deprecation model* with features to be removed by OpenGL 4.0. Writing for a specific release the programmer would use the *core profile* for that release.
- **The Bottom Line**: Practically *all* of the old functionality has been deprecated, being replaced by the **OpenGL**

Accessing Core Functionality

- **Main problems**
  1. Obtaining the correct libraries for a specific release
  2. There are no more matrices
- Use GLEW (GL **E**xtension **W**rangler) to obtain the correct extensions automatically: just use the include
  ```
  #include <GL/glew.h>
  ```
  at the very beginning of your program
- Use the header-only **GLM** math library for vectors and matrices:
  ```
  #include <glm/glm.hpp>
  #include <glm/gtc/matrix_transform.hpp>
  #include <glm/gtx/transform2.hpp>
  ```
  in your program using the glm namespace

Accessing Core Functionality

- **Main problems**
  1. Obtaining the correct libraries for a specific release
  2. There are no more matrices
- Use GLEW (GL **E**xtension **W**rangler) to obtain the correct extensions automatically: just use the include

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

  at the very beginning of your program
- Use the header-only **GLM** math library for vectors and matrices:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtx/transform2.hpp>
```

  in your program using the glm namespace

Shaders and the OpenGL Pipeline

- Per-Vertex Operations are carried out by the **Vertex Shader**. This shader is mandatory
- Primitive Assembly is carried out by the **Tesselation Shaders** *Tesselation Control Shader* and *Tesselation Evaluation Shader*. These shaders are optional. They are concerned with *patches* (later).
- Primitive Processing is done in the **Geometry Shader.** Input are all the vertices it needs to produce a primitive, e.g. a Quad. This shader is likewise optional
- Rasterization is internal producing fragments]item Fragment Processing is carried out by the **Fragment Shader**. This shader is also always required.

The Vertex Shader

Input:

- Things that change with every new vertex:
  - A single vertex
  - Its *attributes*, such as normal vector, color, texture coordinate
  - These global variables are called `IN VARIABLES` or `ATTRIBUTES GLSL 1.2`

Output: Per Vertex calculations, e.g. light intensity

The Vertex Shader

What It Does:

- Transforms vertices
- Transforms and normalize normal vectors
- Generates texture
- Calculates lighting
- Applies colors and materials

The Vertex Shader

How It Works:

- GLSL 1.2 and later: Vertices can be input via the dedicated variable `gl_Vertex`
- Normals via `gl_Normal`, Colors via `gl_Color`, etc.
- Better: Use dedicated channels associated with Vertex Attribute Objects and provide your own variables names for the `IN` or `ATTRIBUTE` variables (later).
- Use the function `glBindAttributeLocation()` to assign the channel (later)
- Use buffers on the server to store your vertex data and its attributes
- Using this data, implement the shader algorithm, e.g. calculate lighting

The Vertex Shader

How It Works (Continued):

- Export data to be used in the fragment shader using `OUT` resp. `VARYING` variables, e.g. light intensity.
- GLSL 1.2 also uses `gl_Position`, `gl_FrontColor`, etc.

A Simple Vertex Shader

```
#vertex 400   //the GLSL version
in vec3 VertexPosition;
in vec3 VertexColor;
out vec3 Color;
void main()
{
   Color = VertexColor;
   gl_Position = vec4(VertexPosition, 1.0);
}
```

Explanation

- This shader just passes through its vertex data
- Note the built-in output variable `gl_position`
- Note the two input attributes
- Values for this type must be provided via **generic vertex attributes**
- The output variable `Color` is the input for the fragment shader, which is just the contents of `VertexColor`

The Fragment Shader

Input: The `out` variablers of the vertex shader

Output: The specification of the associated fragments

- Things to be passed to the Fragment Shader:
  - The same as for fixed-function processing
  - The homogeneous coordinates of the vertex in clipping space
  - These global variables are called `OUT VARIABLES`

A Simple Fragment Shader

```
#vertex 400   //the GLSL version
in vec3 Color;
out vec4 FragColor;
void main()
{
   FragColor = vec4(Color,1.0);
}
```

Explanation

- There is just one input variable, which is identical to the output variable of the vertex shader in name and type
- The value is an interpolated value, calculated from the output vertices
- The result is the (interpolated) fragment color, given by the `out` variable
- Writing the two above shaders results in a shader program that hands the input vertices and their associated colors to the graphics pipeline, producing a shaded object

Remarks: The Vertex Shader

Remarks:

- Each vertex is processed independently
- Cannot mix vertex ops and fixed-functionality ops
- Items required by the Vertex Shader that do not vary with each vertex, such as matrices, are passed in as `UNIFORM VARIABLES`
- The standard out variable `gl_position` has been deprecated(!)

Remarks:The Fragment Shader

Input:

- User-defined in variables as results of rasterization:
  - The window coordinates for output
  - Its relevant properties such as whether it is front-facing, calculated refraction index and any other interpolated values such as color
  - These global variables are called its `IN VARIABLES`
  - These must match up in number and type with the Vertex Shader's out variables

Remarks:The Fragment Shader

What It Does:

- Realizes the shading algorithm applied to its input
- Processes each fragment independently of all others
- Does not replace the fixed-functionality per-fragment and framebuffer operations
- Automatically accepts the results of rasterization
- Deals freely with textures

Remarks:The Fragment Shader

Output:

- The results of the applied algorithm on the fragment
  - The fragment's depth, color, etc.t
  - If it is to be discarded, i.e. not written
  - These global variables are called `OUT VARIABLES`

Remarks: The Fragment Shader

- The fragment shader does not have access to the framebuffer
- The fixed functionality does pixel ownership testing, scissoring, etc
- Quantities that do not change on a per fragment basis, such as light position, density, etc. are called again `UNIFORM`
- No one-to-one correspondence with vertices: 3 vertices can produce a huge number of fragments by the fixed functionality pipeline part
- Incoming values are most often the result of interpolation, e.g. color

Shader Compilation

- The GLSL compilation system is part of OpenGL
  - There is no system compilation command
  - Shader text is submitted to the compiler as a string in a character array
  - The GLSL compiler is an OpenGL call!!
  - Error handling and debugging are problematic
- Each shader is compiled separately
- Compilation produces a "Handle" for each shader
- Compile with the OpenGL call `glCompileShader()`

Example: Compile a Vertex Shader

```
GLuint verShader;
GLint verCompiled;
const char *vShader, *fShader;
verShader = glCreateShader(GL_VERTEX_SHADER);
vShader = readShader("Verts.vert",
                     EVertexShader);
glShaderSource(verShader, 1, &vShader, NULL);
glCompileShader(verShader);
glGetShaderiv(verShader, GL_COMPILE_STATUS,
              &verCompiled);
if (verCompiled == GL_FALSE){...}
```

Remarks

- Assume the shader source has been written to the file `Verts.vert`
- Notice the distinct steps:
  - Create a vertex shader object with `glCreateShader(GL_VERTEX_SHADER)`
  - Load the shader source into a string: `readShader()`, which is written by the programmer
  - Give it a handle with `glShaderSource()`
  - Compile with `glCompileShader()`
  - Get compilation status: `glGetShaderiv()`
  - Fragment shaders are compiled analogously

How to Link Several Shaders into a Program

- All compiled shaders must be linked together into a program
- The linker must coordinate output variables from one shader to input variables of the next
- Linker must also link OpenGL variables with shaders (uniform variables)
- The linked program must be installed into the OpenGL pipeline

Link Shaders into a Program

```
GLuint programHandle, linked;
programHandle = glCreateProgram();
glAttachShader(programHandle, verShader);
glAttachShader(programHandle fragShader);
glLinkProgram(programHandle);
glDeleteShader(verShader);
glDeleteShader(fragShader);
glGetProgramiv(programHandle, GL_LINK_STATUS,
              &linked);
if (linked == GL_FALSE){ ...}
glUseProgram(programHandle);
```

Remarks about Linking

- Use `glCreateProgram()` to create the program object
- Attach the shaders to the program object with `glAttachShader()`
- Link with `glLinkProgram()`
- Delete the compiled shaders, because they have been linked
- Check the link status and react accordingly
- Install the program into the OpenGL pipeline with `glUseProgram()`
- If needed, program is available by its handle `programHandle`

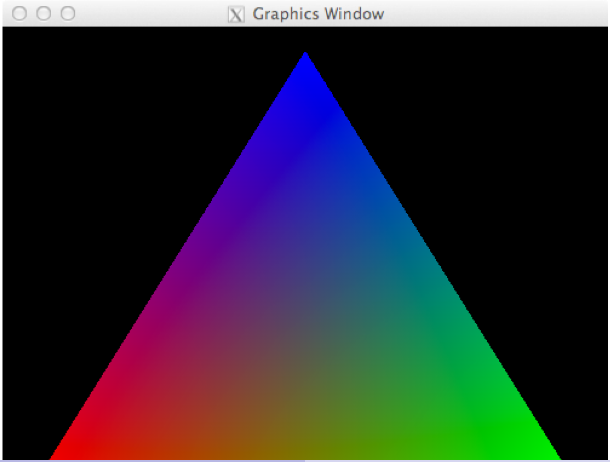
Example: Draw of Gouraud triangle

```
void redraw()
{
  glClear(GL_COLOR_BUFFER_BIT);
  glShadeModel(GL_SMOOTH);
  glBegin(GL_POLYGON);
    glColor3f(1.0,0.0,0.0);
    glVertex2f(-0.9,-0.9);
    glColor3f(0.0,1.0,0.0);
    glVertex2f(0.9,-0.9);
    glColor3f(0.0,0.0,1.0);
    glVertex2f(0.0,0.9);
  glEnd();
  if (doubleBuffer) glXSwapBuffers(display,main_wind
```

Remarks

- The OpenGL program normal with drawing taking place in `redraw()`
- There is no initialization
- Use the Vertex and Fragment Shaders above
- Write a `shader()` program that compiles. links and installs the shaders
- Run it (in X windows)
- Result: Gouraud shaded triangle

Gouraud Triangle Using Shaders



Remarks About the Gouraud Triangle

- The shaders involved use standard variable to pass the attribute variables
- The has limited value
- GLSL uses **Vertex Attribute Objects** to transfer the vertices and their attributes into the Vertex Shader
- These,in turn, are built on Buffer Objects, defined on the server
- Pointers to the attribute arrays are defined using `glVertexAttributePointer()` in the compilation process
- Consequence: You can use your own `in` resp. `attribute` variables using this preferred method

Vertex Attribute Data

- Vertex Attributes are the input variables to the Vertex Shader
- Problem: How do we pass values from our OpenGL program to these input variables?
- Early versions of GLSL used dedicated variables, e.g. `gl_position` to enable the communication
- Now this mechanism is under the control of the programmer with **Vertex Array Objects**
- Use **Vertex Buffer Objects** to store the data serverside using `GL_ARRAY_BUFFER` binding (for now)
- Tie the buffer objects together in one data structure using VAOs
- This is not specific to Vertex Attributes

Example of Vertex Array Objects: Defining

```
GLuint vaoHandle;
GLuint vertexBufferHandle = bufferID[0]; //VBO
GLuint colorBufferHandle = bufferID[0]; //VBO
glGenVertexArrays(1, &vaoHandle);
glBindVertexArray(vaoHandle);
glBindBuffer(GL_ARRAY_BUFFER, vertexBufferHandle);
glEnableClientState(GL_VERTEX_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, normalBufferHandle);
glEnableClientState(GL_NORMAL_ARRAY);
glBindVertexArray(0); //temporarily disable
```

Example of Vertex Array Objects: Using

```
void redraw()
{
  glBindVertexArray(vaoHandle);
  glDrawArrays(...);
  glBindVertexArray(0);
}
```

Using VAOs with Vertex Attributes

- Using `glBindVertexArray` makes `vaoHandle` the current VAO
- The first time used binds all of the default state of the current context
- All future calls will use that entire state
- To set the state for rendering use the important `glVertexAttribPointer()` to associate a channel for the desired data as well as a pointer to that data on the GPU
- This stores the VBO in the VAO
- Binding a new VAO does not change those buffer bindings, i.e. the current bound buffers are not stored in the VAO

Description of `glVertexAttribPointer`

- Parameter 1: The actual channel to be used for the associated data (generic attribute index)
- Parameter 2: The number of components per vertex attribute (1,2,3 or 4)
- Parameter 3: The type of each component (they are all the same)
- Parameter 4: Used for normalized data
- Parameter 5: The stride, used for tightly packed data
- Parameter 6: A pointer to the start of the data on the GPU (actually a byte offset from the beginning of the buffer)

Example: VAO that defines Vertex Attributes

```
glGenVertexArrays(1, &vaoHandle);
glBindVertexArray(vaoHandle);
glEnableVertexAttribArray(0);   //Vertices
glBindBuffer(GL_ARRAY_BUFFER, bufferID[0]);
glVertexAttribPointer((GLuint)0, 3, GL_FLOAT,
        GL_FALSE, 0, ((GLubyte *)NULL + (0)) );
glEnableVertexAttribArray(1);   //Colors
glBindBuffer(GL_ARRAY_BUFFER, bufferID[1]);
glVertexAttribPointer((GLuint)1, 3, GL_FLOAT,
        GL_FALSE, 0, ((GLubyte *)NULL + (0)) );
glBindVertexArray(0);
```

UniformVariables: Sharing Other OpenGL Data

- Importing vertices and associates attributes is only one kind of application data that is needed for use in a shader
- For example, light intensity or a ModelView transformation are also needed
- Solution: Uniform Variables–data that does change on a per-vertex basis
- Uniform variables are read-nly in a shader, i.e. their values are set by the application
- Consequence: Uniform Variables must be initialized by the application
- The GLSL linker must associate these locations with `glGetUniformLocation()` mapping shader names to actual variable locations

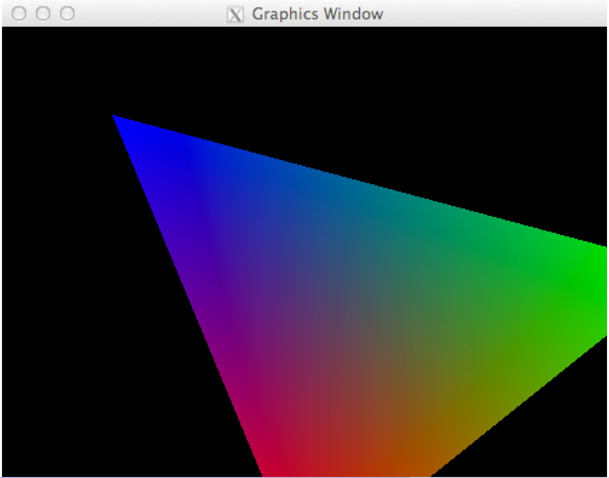
Rotate a Gouraud Triangle:Vertex Shader

```
#version 120

attribute vec3 VertexPosition;
attribute vec3 VertexColor;

uniform mat4 RotationMatrix;

void main(void)
{
  gl_Position = RotationMatrix *
                vec4(VertexPosition, 1.0);
  gl_FrontColor = VertexColor;
}
```

Roate a Gouraud Triangle: App Code

```
int Angle = 20;
glm::mat4 rotationMatrix =
    glm::rotate(mat4(1.0f), Angle,
            vec3(0.0f, 0.0f, 1.0f));

GLuint location =
     glGetUniformLocation(prog, "RotationMatrix");
 if (location >= 0 )
  {
    glUniformMatrix4fv(location, 1, GL_FALSE,
                        &rotationMatrix[0][0]);
  }
```

Rotating a Gouraud Triangle



OpenGL Shading Language (GLSL)

- Basically the C Programming Language, with some extensions and omissions
- Easily learned by those with knowledge of C
- Omissions
    - No pointers
    - No memory management
    - No standard input and output

OpenGL Shading Language (GLSL)

- Extensions
  - VectorTypes: **vecn, ivecn, uvecn, bvecn** (n = 2,3,4)
  - Matrix Types: **matn** (n = 2,3,4) with overloaded multiplication and composition with vectors "*"
  - Shader Interfaces: **in, out, uniform, const**
  - Buffer Interfaces: **layout**
  - Texture Interfaces: **Samplers**
  - Many, many built-in functions
  - See Orange Book, pp. 65-99

References

- Schreiner, Dave *et al.*, "OpenGL Programming Guide," *Addison-Wesley*, Eighth Ed., Boston 2013.
- Kilgard, Mark, "OpenGL Programming for the X-Window System", Addison-Wesley, Boston 1996
- Rost, Randi, Licea-Kane, Bill, "OpenGL Shading Language," *Addison-Wesley*, Third Ed., Boston 2010
- Martz, Paul, "OpenGL Distilled," Addison-Wesley, Boston 2006
- Wolff, David, "OpenGL 4.0 Shading Language Cookbook", *Pakt Publishing*, Birmingham 2011