UNIVERSITÉ Concordia UNIVERSITY
FACULTY OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science and Software Engineering

COMP371
COMPUTER GRAPHICS

LECTURE 11
PROGRAMMABLE SHADERS

Lecture Overview

- Review of last class
- GPU programming

OpenGL in Application Setup

- Create a Framebuffer, Depth Buffer, all the buffers you need, with the resolution you need
  - Customize with glfwWindowHint
  - Create with GLFW: glfwCreateWindow
- Set OpenGL Rendering States
  - Example

```
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
```

Create Vertex Array/Buffer Objects

- Create a Vertex Array Object Which will manage your Vertex Buffer Objects

  glGenVertexArrays(1, &vao);

- Vertex Buffer Objects contains Geometry information in Model Space. Draw calls will later use VBO data.
  - Create an Array containing all of your Geometry Data (Position, Normals, Colors, UV Coordinates, etc)

  float vertexBuffer[] = {1.0f, 2.0f, …. };
  - Create a Vertex Buffer Object

  glGenBuffers(1, &vbo);

  glBindBuffer(GL_ARRAY_BUFFER, vbo);

  glBufferData(GL_ARRAY_BUFFER, sizeof(vertexBuffer), vertexBuffer, GL_STATIC_DRAW);

Load Shaders

- You can have multiple shader pairs (Vertex/Fragment) for different type of geometry
- To create Shader Objects, you must load, compile, and link your shaders (normally in the program initialization)

```
glCreateShader
glCompileShader
glAttachShader
glLinkProgram
```

Frame Initialization

- When the OpenGL Setup is done, you are ready to draw!
- Before Drawing, make sure your Scene is updated for the next frame. The means, the scene goes forward by a time step (typically 1/60 second)
  - Animations, Physics, Camera, etc…
- Clear the rendering Context to start drawing a new frame from Scratch (Frame buffer, Depth buffer, etc)

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Drawing

- Bind Shader

  glUseProgram(shaderID);

- Set Shader Constants (eg: VAR matrix in Shader)

  GLuint loc = glGetUniformLocation(shaderID, “VAR");

  glUniformMatrix4fv(loc, 1, GL_FALSE, &varMatrix);

- Bind Vertex Array Object

  glBindVertexArray(vao);

Drawing cont'd

- Set Shader Input Data (VBO)

```
glEnableVertexAttribArray(0); // 1st input in vertex shader
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glVertexAttribPointer(   0,// attribute
                             3,// size
                               GL_FLOAT,// type
                               GL_FALSE,// normalized?
                               sizeof(Vertex), // stride
                               (void*)0 );   // offset
```

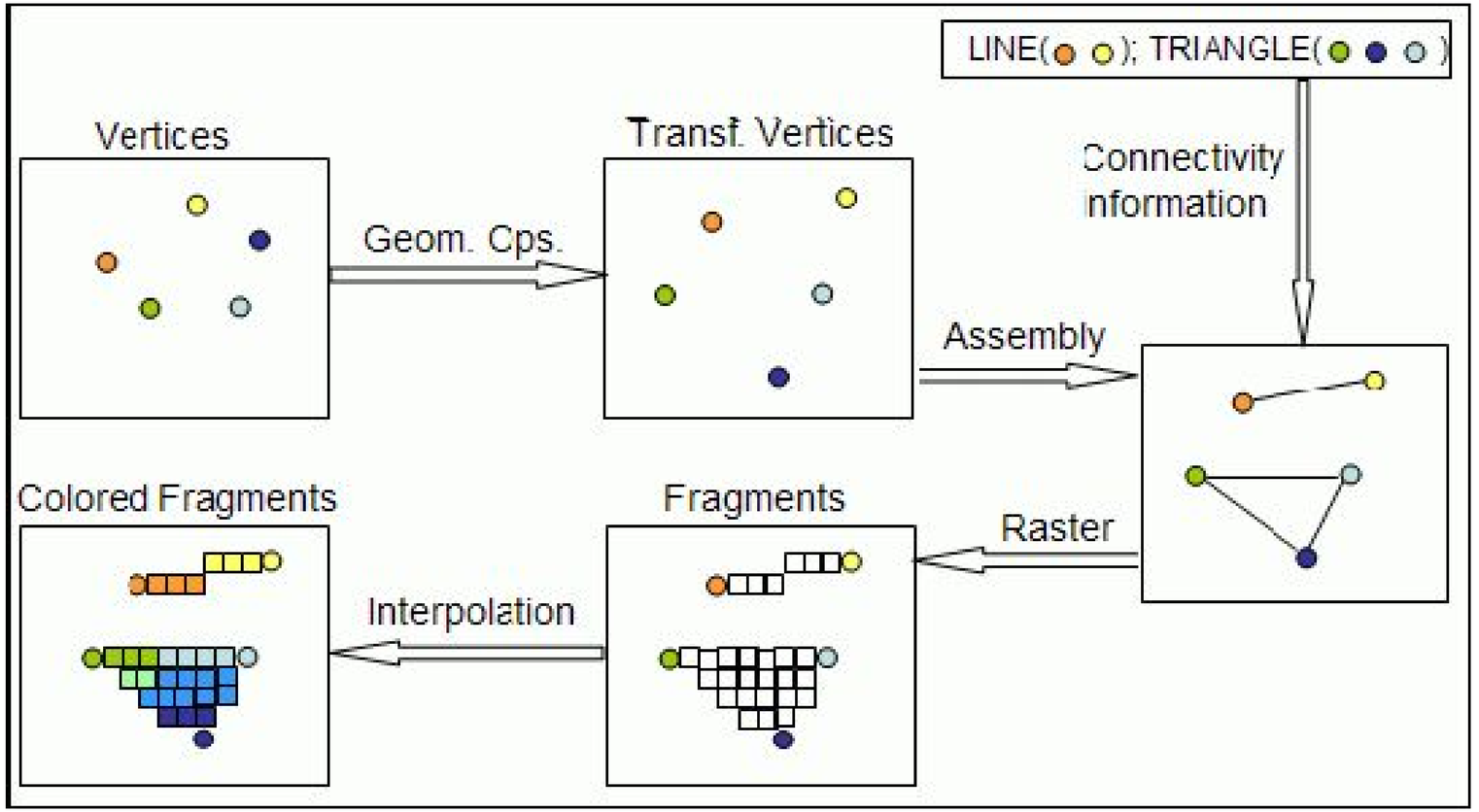
- Draw Call `glDrawArrays(GL_TRIANGLE_STRIP, 0, numVertices);`
- Cleanup `glDisableVertexAttribArray(0);`

Finishing a Frame Swap Buffers

- Modern Renderer typically use Double Buffering
  - While Front Buffer is sent to Video Controller
  - Draw in the Back Buffer
  - When done drawing, Swap the buffers

- With GLFW
  - glfwSwapBuffers

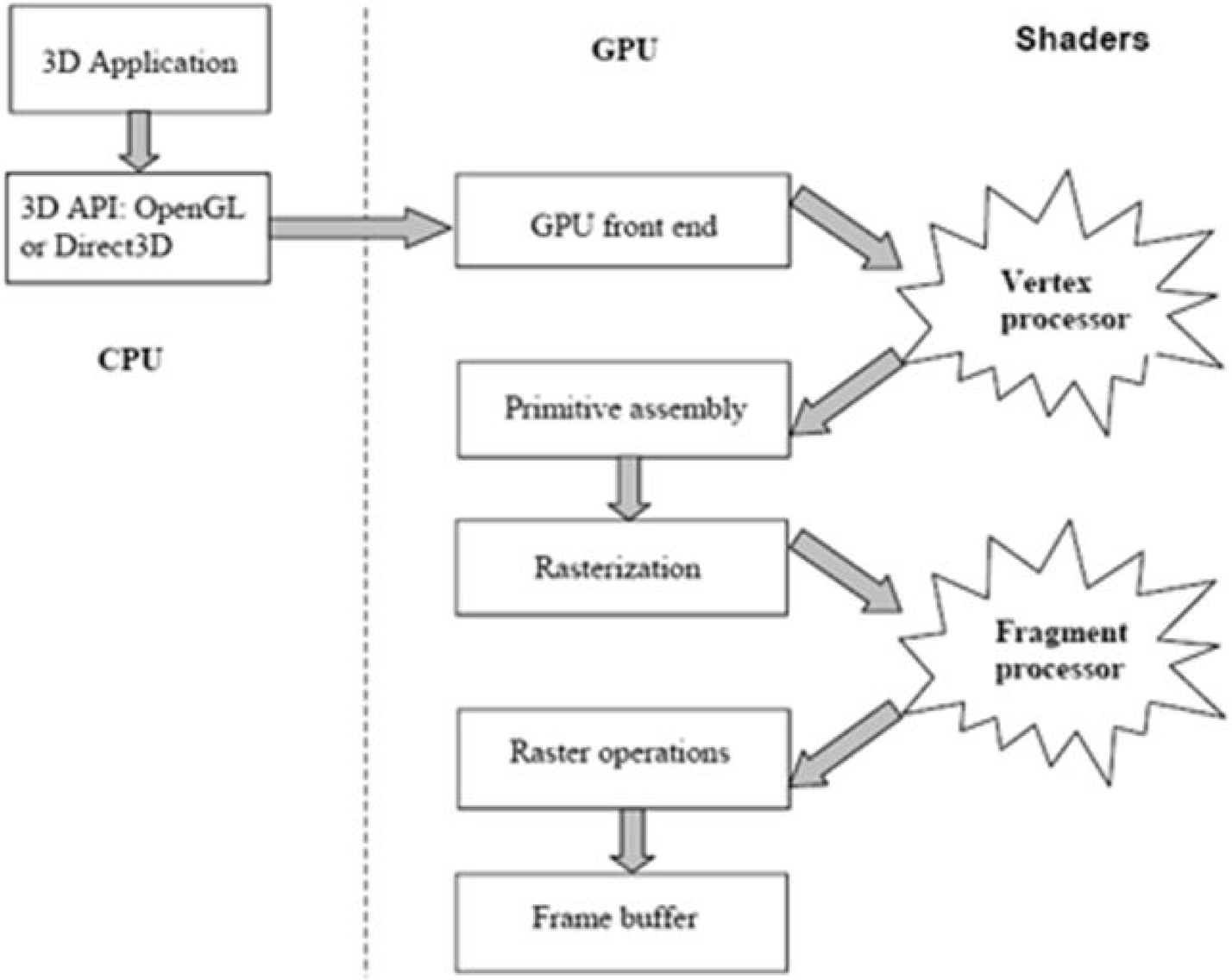
GPU Pipeline



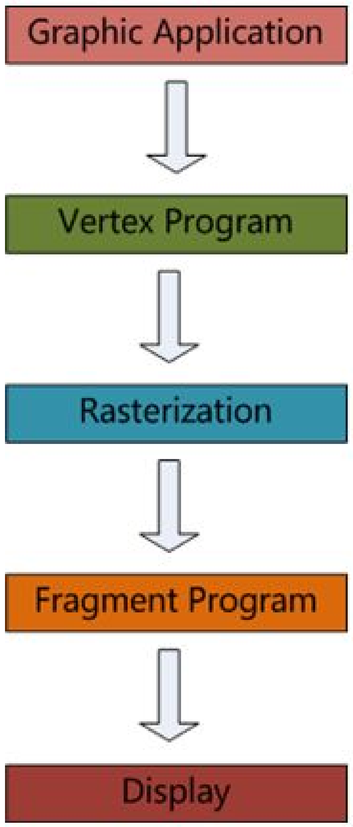
Programmable Hardware

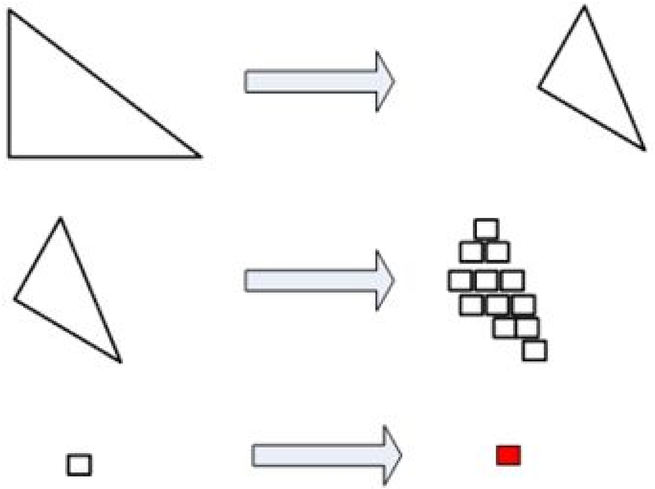
- Most modern graphics cards allow parts of the GPU pipeline to be modified by a developer
  - Increased functionality with hardware support
  - Complex graphics algorithms with fast implementation.
  - General Computing
- Until recently, the programmable parts were
  - Vertex Processor
  - Fragment (Pixel) Processor
- OpenGL 4 also allows the programming of *Geometric processors* and *Compute Processors* for General Purpose GPU programming
- All these programs are called **Shaders.**

Programmable GPU Pipeline



Programmable GPU Pipeline cont'd

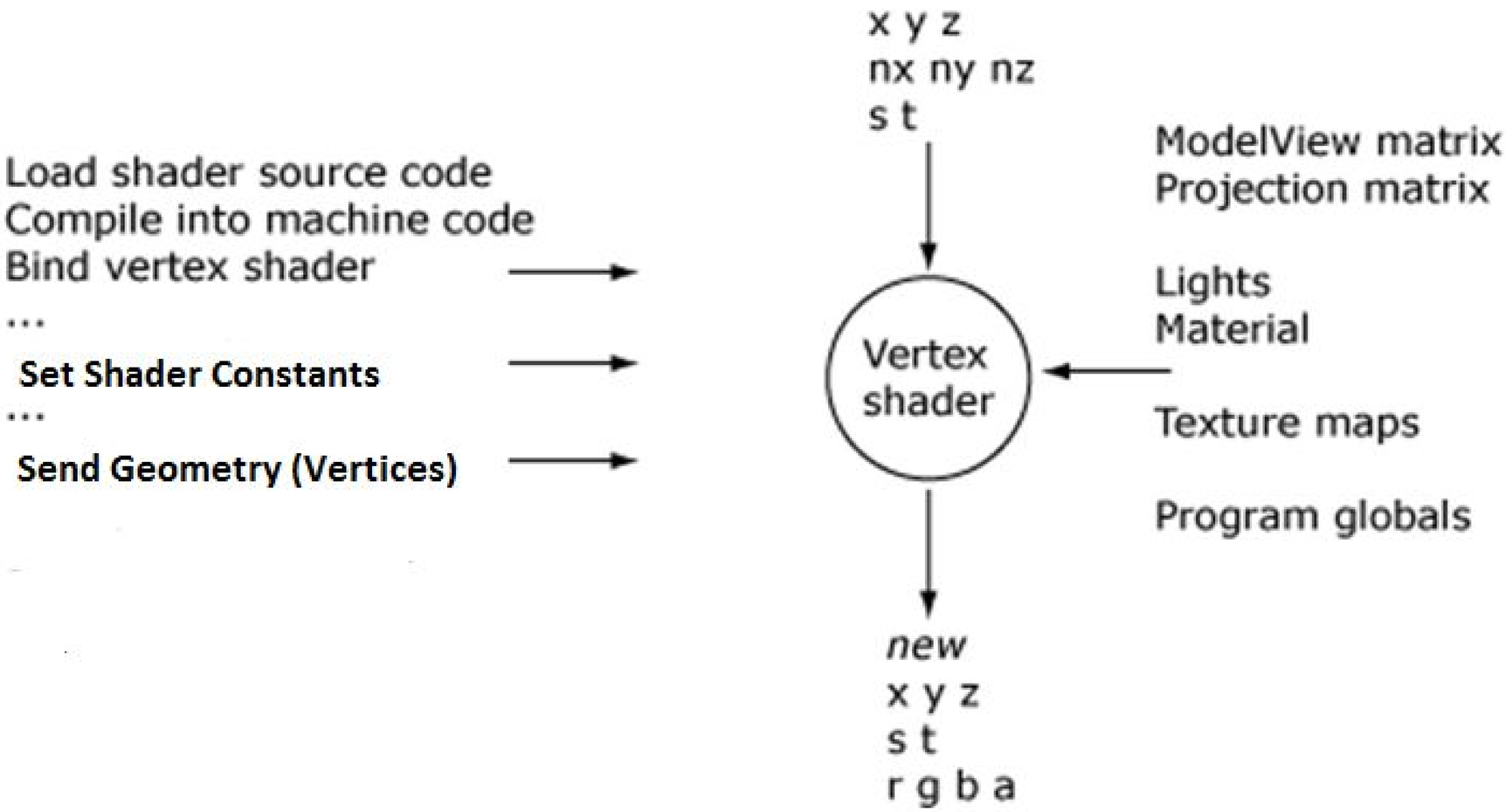




Vertex Processor

- Responsible for the following operations:
  - Vertex transformation
  - Normal transformation and normalization
  - Texture Coordinates Generation and transformation
  - Lighting (per vertex)
  - Color & Material (per vertex)
- A Vertex shader replaces the entire functionality of the fixed vertex processor!
  - When writing a shader, one must replace the entire functionality.
- Vertex Shaders operate simultaneously on all vertices (in parallel)
  - No vertex information can affect another vertex.

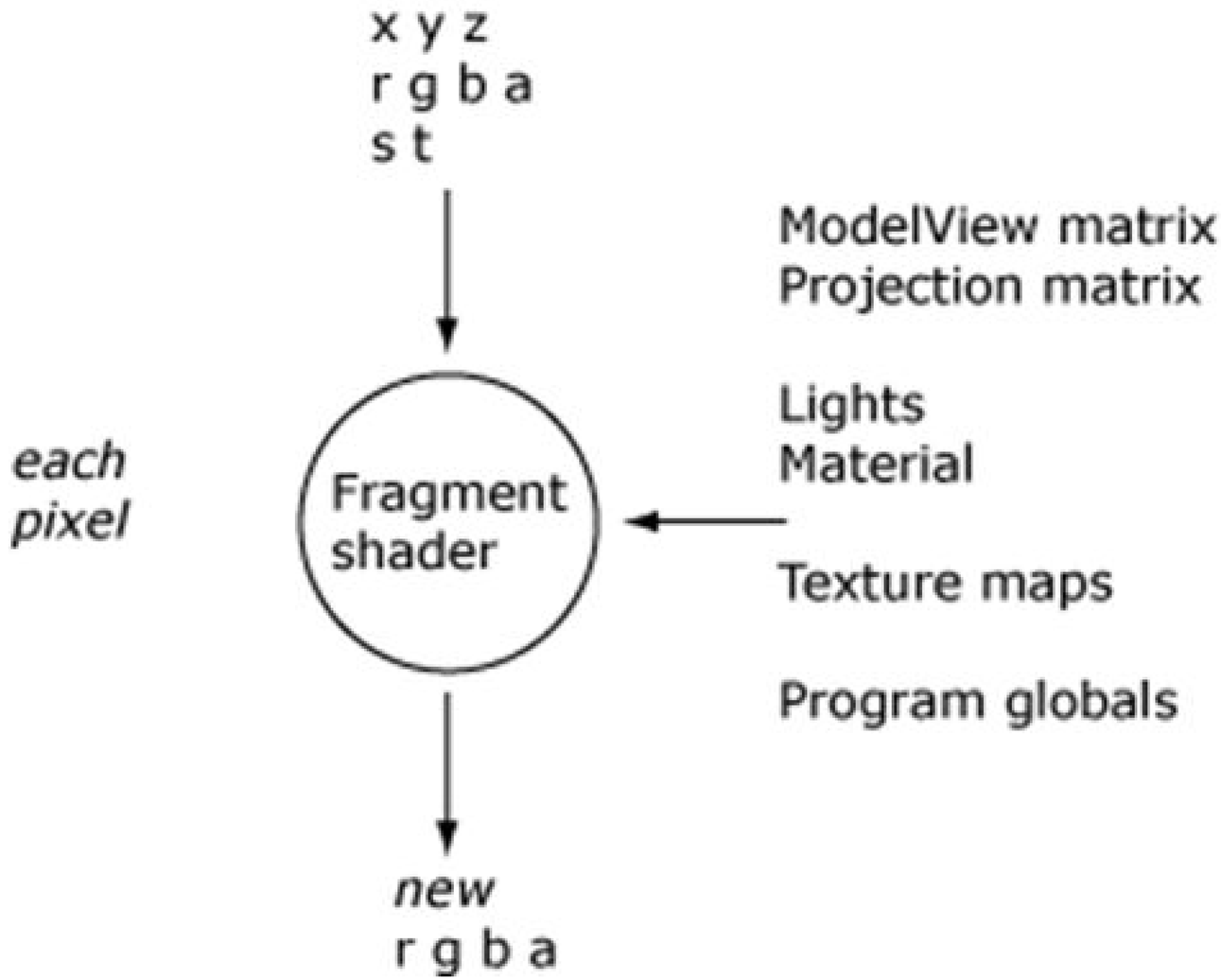
Vertex Shader



Fragment Processor

- Operates on the fragments AFTER the interpolation and rasterization of vertex data.
- Allows the following programming:
  - Operation on interpolated vertex values (Phong, anyone?)
  - Texture access & application
  - Fog
  - Color sum, etc.
- All fragment shaders work in parallel
  - Again, no access to neighbouring fragments.
  - Does not replace back-end operations (alpha blending, and such).

Fragment Shader



Shading Languages

- The first approaches used an assembler language of the cards (with significantly different instruction sets on both shaders).
  - Both OpenGL and DirectX also provide a different API
    - load shaders
    - pass user-defined data
    - delete
- High level languages
  - Cg (NVidia), ***GLSL (openGL),*** HLSL (DirectX)

GLSL

- A high-level procedural languages (similar to C++).
- As of OpenGL 2.0, it is a standard, with a simple glXYZ API from OpenGL applications
  - Former implementations of GLSL API to OpenGL used OpenGL extensions
- The same language, with small differences, is used for all types of shaders.
- Natively supports “graphical needs” (such as containing types like vectors and matrices).

Can I use GLSL?

- Almost every current high-end GPU supports it
- How to check
  - Run *glewinfo.exe*
  - It will tell you which features you can use

Language Syntax

- Similar to C/C++
- Data Types
  - vector, matrix, texture

- Control Flow
  - same as C.
    - e.g.: for, if, while

Language Syntax cont'd

- Built-in functions for
  - math
  - interpolation (mix, step, smoothstep)
  - geometric
  - matrix
  - vector relational
  - texture
  - shadow
  - vertex, fragment
  - noise

Communication

- OpenGL → Shaders
  - Uniform variables

- Application → Vertex Shader → Fragment shader
  - Input variables
  - Output variables

Uniform Variables

- **uniform datatype dataname**
- suitable for values that remain constant along a primitive, frame, or even the whole scene
- can be read (but not written) in both vertex and fragment shaders

Example

```
uniform float specIntensity;  // in shader

// in application
...
GLint loc1;
float specIntensity = 0.98;

// gets the location of the variable.
loc1 = glGetUniformLocation(p,"specIntensity");

// sets the value.
glUniform1f (loc1,specIntensity);
...
```

Typical Uniform Variables

- Some Matrices
  - **uniform mat4 ModelViewMatrix;**
  - **uniform mat3 NormalMatrix;**
  - **uniform mat4 TextureMatrix[n];**
- Lighting, depth, material
  - defined as some structures
  - **uniform MaterialParameters FrontMaterial;**
  - **uniform LightSourceParameters LightSource[MaxLights];**
- And many variables
  - it refers OpenGL state variables

Input Variables

- **in datatype dataname**
- **Read only variables**

- In Vertex Shader
  - Typically Vertex Data (position, normal, uv, color)
- In Fragment Shader
  - Outputs from Vertex Shader become Input in Fragment Shader
  - The variable names must match

Input Variable Example for Vertex Shader

```
in vec3 position; // in shader
```

```
// in application
...
//gets the location of the variable.
positionID = glGetAttribLocation (p,"position");
...

glEnableVertexAttribArray(positionID);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glVertexAttribPointer( … );


...
```

or:
glBindAttribLocation(p, 0, "position");
….
glEnableVertexAttribArray(0);
….

Output Variables

- **out datatype dataname**

- In Vertex Shader
  - Output variables are inputs in Fragment Shader
  - Values will be interpolated in Fragment Shader
- Declared in both vertex and fragment shader
  - type and name should be matched

Vertex Shader Program

```
#version 330

uniform mat4 view_matrix, model_matrix, proj_matrix;

in  vec3 in_Position;
in  vec3 in_Color;
out vec3 out_Color;

void main () {
    mat4 MVP = proj_matrix * view_matrix * model_matrix;
    gl_Position = MVP * vec4 (in_Position, 1.0);

    out_Color = in_Color;
};
```

Fragment Shader Program

```
#version 330

in vec3 out_Color;
out vec4 frag_colour;

void main () {

    frag_colour = vec4 (out_Color, 1.0);

};
```

After loading the shaders:

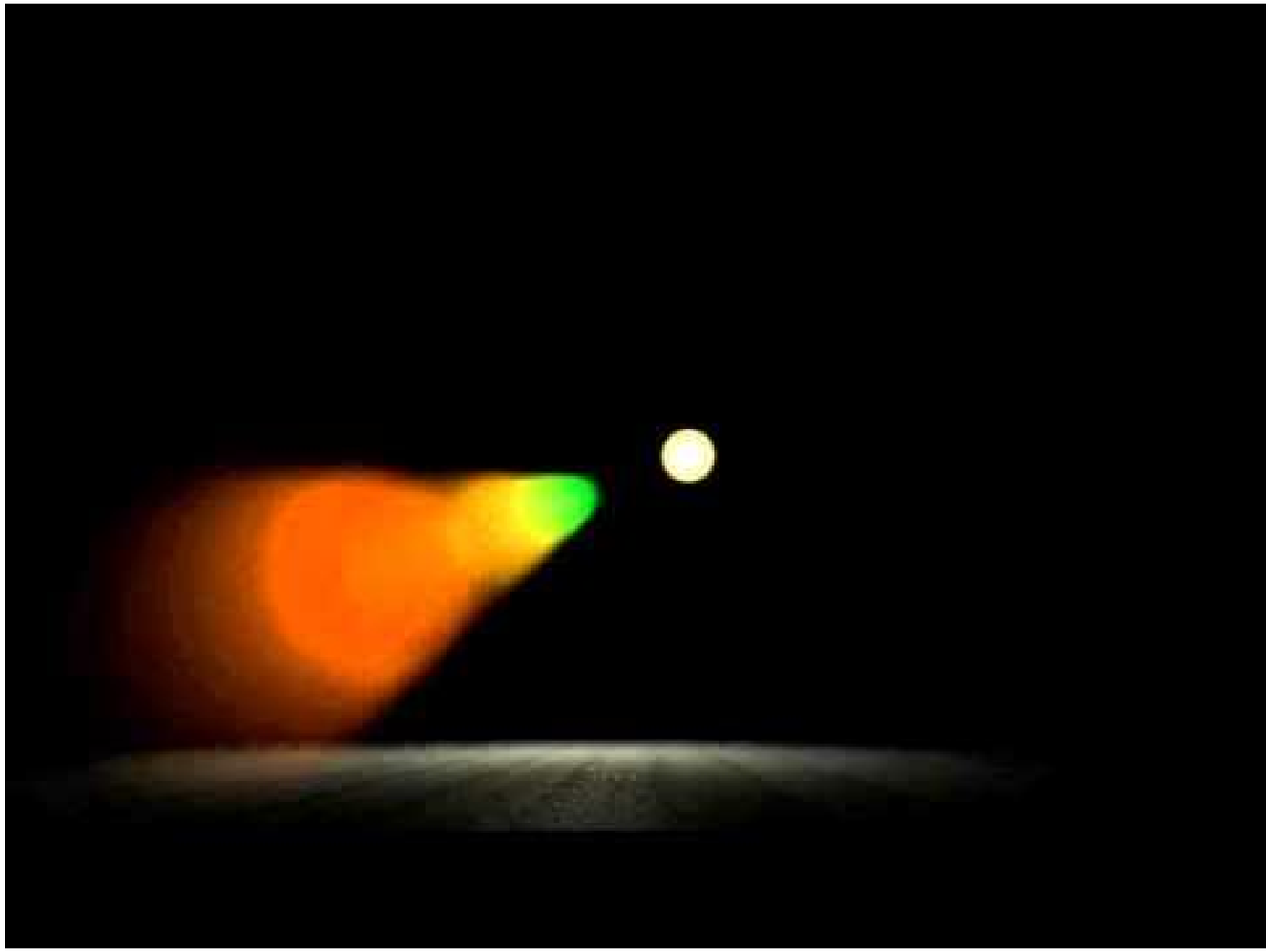
```
view_matrix_id = glGetUniformLocation(ProgramID, "view_matrix");
model_matrix_id = glGetUniformLocation(ProgramID, "model_matrix");
proj_matrix_id = glGetUniformLocation(ProgramID, "proj_matrix");
```

```
glBindAttribLocation(ProgramID, 0, "in_Position");
glBindAttribLocation(ProgramID, 1, "in_Color");
```

At every drawing iteration:

```
glUseProgram(shader_program);

glUniformMatrix4fv(proj_matrix_id, 1, GL_FALSE, glm::value_ptr(proj_matrix));
glUniformMatrix4fv(view_matrix_id, 1, GL_FALSE, glm::value_ptr(view_matrix));
glUniformMatrix4fv(model_matrix_id, 1, GL_FALSE, glm::value_ptr(model_matrix));
```



Review

- GPU programming

Next Lecture

- OpenGL shading

CONCORDIA.CA

Studying Support Slides