Programmable shaders

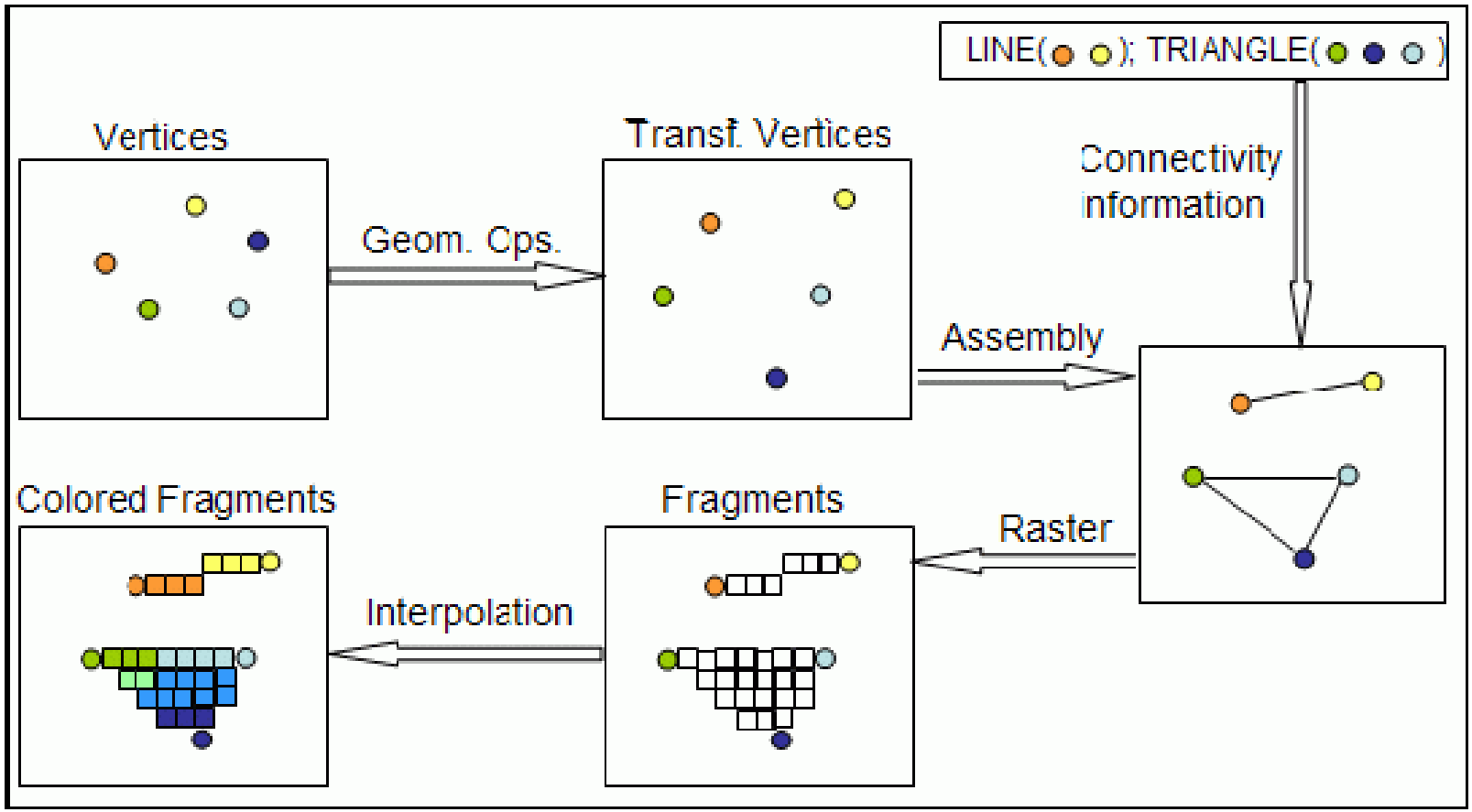
Charalambos Poullis

Department of Computer Science & Software Engineering
Faculty of Engineering & Computer Science

February 7, 2019

Lecture Overview

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**.

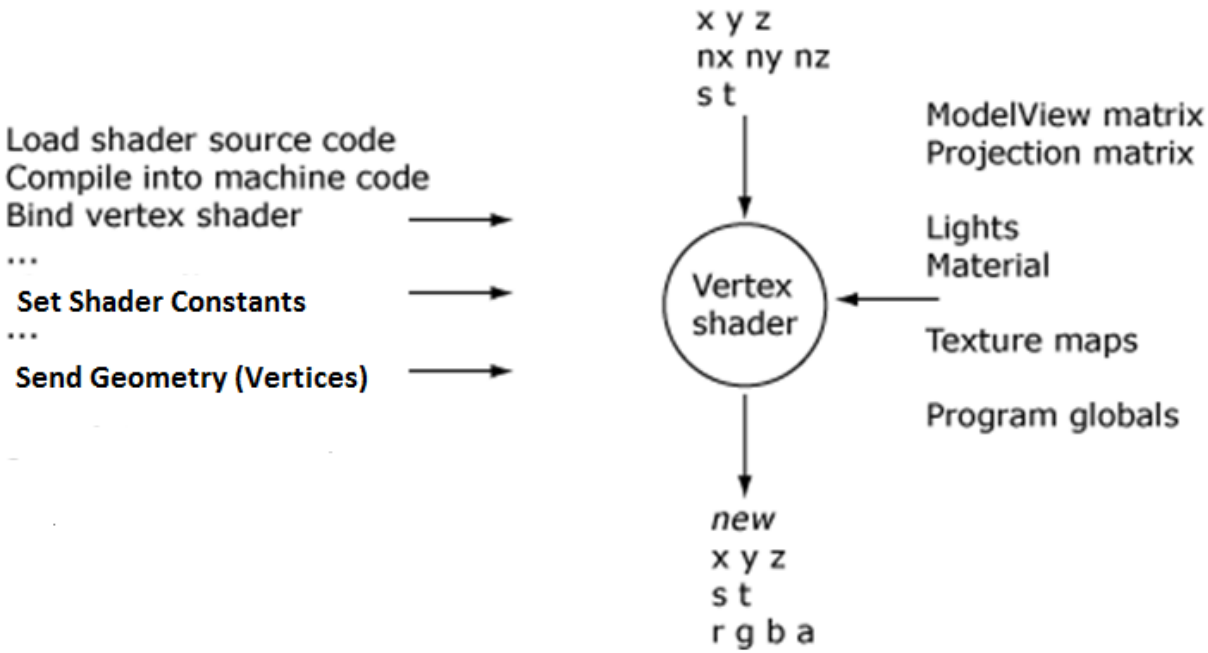
Vertex processor

Responsible for the following operations:

- vertex transformations
- normal transformation and normalization
- texture coordinates generation and transformation
- lighting (per vertex)
- color and 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.
No information sharing among vertices $\rightarrow$ vertex shaders operate in parallel on all vertices

Vertex shader



Fragment processor

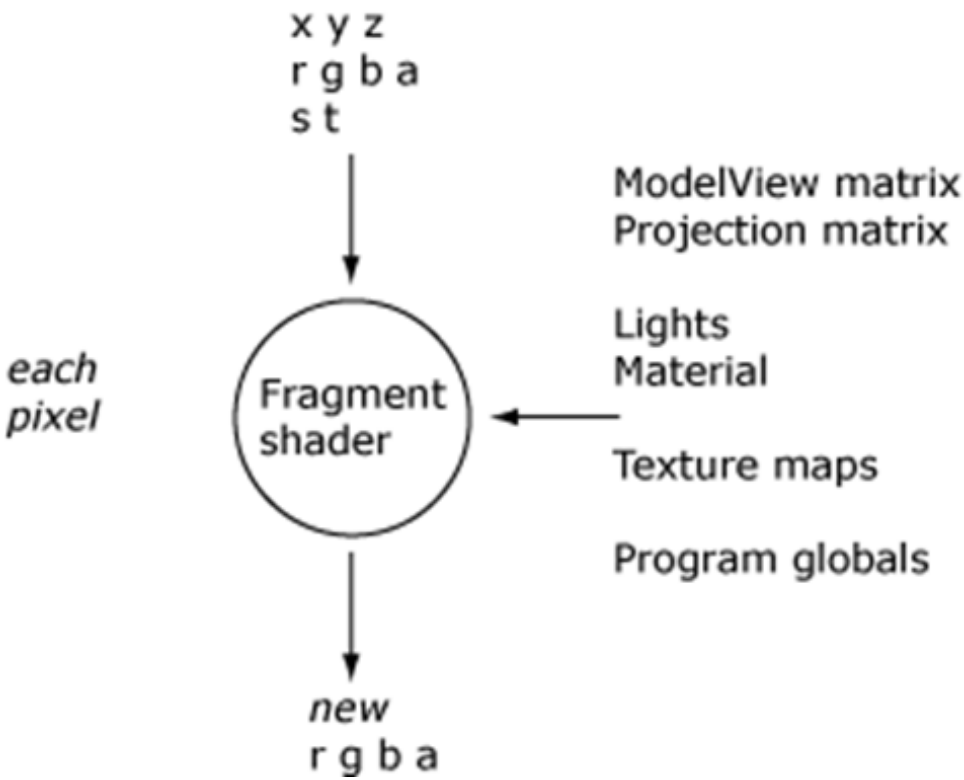
Operates on fragments **after** the interpolation and rasterization of vertex data. Allows the following programming:

- operation on interpolated vertex values e.g. Phong lighting
- texture access and application
- fog
- color sum, etc.

Again, no access to neighbouring fragments $\rightarrow$ all fragment shaders work in parallel

Does not replace back-end operations (alpha blending, and such).

Fragment shader



Vertex Array/Buffer Objects

Create a Vertex Array Object(VAO) which will manage your Vertex Buffer Objects (VBOs):

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

Assuming an array *vertexBuffer* contains all of your data e.g. position, normals, colors, UV coordinates, etc

```
float vertexBuffer[] = {1.0f, 2.0f, ...};
```

Vertex Array/Buffer Objects

Vertex Buffer Objects contain information in **model space**. Draw calls will later use the VBO data. Set Shader Input Data (VBO)

```
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertexBuffer),
vertexBuffer, GL_STATIC_DRAW);
glEnableVertexAttribArray(0); // 1st input in vertex shader
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glVertexAttribPointer(0,// attribute (layout)
3,// size
GL_FLOAT,// type
GL_FALSE,// normalized?
sizeof(float), // stride
(void*)0 ); // offset
```

Loading the shaders

You can have multiple shader pairs (vertex-fragment) for different types of geometry

Create shader objects: load, compile, and link [vertex|fragment] 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. This means, the scene goes forward by a time step (typically $\frac{1}{60}$ second) e.g. 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 (e.g. model_matrix in vertex shader)

```
glm::mat model_matrix = glm::mat4;
GLuint model_matrix_id = glGetUniformLocation(shaderID,
"model_matrix");
glUniformMatrix4fv(model_matrix_id, 1, GL_FALSE,
glm::value_ptr(model_matrix));
```

Bind the Vertex Array Object

```
glBindVertexArray(vao);
```

Drawing

Draw call

glDrawArrays(GL_TRIANGLE_STRIP, 0, number_of_vertices);

Cleanup

glDeleteProgram, glDeleteShader, glDeleteShader, glDeleteBuffers, glDeleteVertexArrays

Example - initialization


```
bool initialize()   {

    ///Initialize the GLFW window
    if (!glfwInit()) return false;

    ///Create the window
    window_handle = glfwCreateWindow(VIEWER_WIDTH, VIEWER_HEIGHT, "COMP371 - Interaction",0x00,0x00);
    if (!window_handle)    {
        std::cout << "ERROR: Failed to create a window." << std::endl;
        glfwTerminate();
        return false;
    }
    glfwMakeContextCurrent(window_handle);

    ///Initialize GLEW extension handler
    glewExperimental = GL_TRUE; ///Needed to get the latest version of OpenGL
    glewInit();

    ///Test if all is good
    const GLubyte  *renderer = glGetString(GL_RENDERER);
    std::cout << "Renderer: " << renderer << std::endl;
    const GLubyte  *version = glGetString(GL_VERSION);
    std::cout << "OpenGL version: " << version << std::endl;

    ///Enable the depth test
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);


    return true;
}
```

Example - setup


```
///load the shaders
shader_program_id = loadShaders("../basic.vs", "../basic.fs");

///define geometry
float points[] =    {0.0f, 0.5f, 0.0f,
                     0.5f, -0.5f, 0.0f,
                     -0.5f,-0.5f,0.0f};

///setup the containers
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, 9*sizeof(float), points, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(0);

glBindVertexArray(0);

///get the location of the matrix
ctm_id = glGetUniformLocation(shader_program_id, "ctm");
```

Example - rendering


```
std::cout << "Rendering..." << std::endl;
while (!glfwWindowShouldClose(window_handle))  {
    ///erase the contents every time
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    ///set the viewport to have the same size as the window
    glViewport(0,0,VIEWER_WIDTH,VIEWER_HEIGHT);

    ///start using the shader program
    glUseProgram(shader_program_id);

    ///compute the ctm
    ctm = persp_matrix * view_matrix * model_matrix;
    ///send the ctm
    glUniformMatrix4fv(ctm_id,1,GL_FALSE,glm::value_ptr(ctm));


    ///draw points 0-3 from the currently bound VAO with current shader program
    glBindVertexArray(vao);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glBindVertexArray(NULL);
    ///update other events like input handling
    glfwPollEvents();
    ///swap the buffers
    glfwSwapBuffers(window_handle);
}

///cleanup
cleanup();
```

Example - rendering


```
bool cleanup()  {
    //Release memory e.g. vao, vbo, etc
    glDeleteProgram(shader_program_id);
    glDeleteShader(vertex_shader_id);
    glDeleteShader(fragment_shader_id);

    glDeleteBuffers(1, &vbo);
    glDeleteVertexArrays(1, &vao);

    if (window_handle)      glfwDestroyWindow(window_handle);
    window_handle = 0x00;

    glfwTerminate();

    return true;
}
```

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 e.g. Cg (NVidia), GLSL (openGL), HLSL (DirectX)

GLSL

A high-level procedural language (similar to C++).

As of OpenGL 2.0, it is a standard, with a simple gl_XYZ 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, math functions, etc).

Language syntax

Similar to C/C++

Data types: vec, mat, sampler*
Control flow: same as C e.g. for, if, while

Language syntax

Built-in functions for

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

Link between application data and shader variables

OpenGL $\rightarrow$ shaders: use uniform variables

Application $\rightarrow$ vertex shader $\rightarrow$ fragment shader: use input/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 write) in both vertex and fragment shaders

Example - in shader


```
#version 130

out vec4 frag_color;

uniform int flag;
uniform vec3 random_color;

void main() {
    if (flag == 1)   {
        frag_color = vec4(0.8, 0.8,0.6,1.0f);
    }
    else    {
        frag_color = vec4(random_color,1.0f);
    }
    return;
}
```

Example - in application


```
///get the location of the matrix
ctm_id = glGetUniformLocation(shader_program_id, "ctm");

///get the location of the random color variable in the fragment shader
GLint random_color_id = glGetUniformLocation(shader_program_id, "random_color");

///get the location of the flag variable
GLint flag_id = glGetUniformLocation(shader_program_id, "flag");

std::cout << "Rendering..." << std::endl;
glm::vec3 random_color;
while (!glfwWindowShouldClose(window_handle))  {
    ///erase the contents every time
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    ///set the viewport to have the same size as the window
    glViewport(0,0,VIEWER_WIDTH,VIEWER_HEIGHT);

    ///start using the shader program
    glUseProgram(shader_program_id);

    ///compute the ctm
    ctm = persp_matrix * view_matrix * model_matrix;
    ///send the ctm
    glUniformMatrix4fv(ctm_id,1,GL_FALSE,glm::value_ptr(ctm));

    ///compute the random color
    random_color = glm::vec3(drand48(),drand48(),drand48());
    ///send the random color
    glUniform3fv(random_color_id, 1, glm::value_ptr(random_color));

    ///send the flag
    glUniform1i(flag_id, flag);

    ///draw points 0-3 from the currently bound VAO with current shader program
    glBindVertexArray(vao);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glBindVertexArray(NULL);
    ///update other events like input handling
    glfwPollEvents();
    ///swap the buffers
    glfwSwapBuffers(window_handle);
}
```

Typical uniform variables

Some Matrices

- uniform mat4 ModelViewMatrix;
- uniform mat3 NormalMatrix;
- uniform mat4 TextureMatrix[n];

User-defined data structures

- uniform MaterialParameters FrontMaterial;
- uniform LightSourceParameters LightSource[MaxLights];

Input variables

in datatype dataname - read only variables

In vertex shader - typically Vertex Data (position, normal, uv, color)

In fragment shader - output variables from Vertex Shader become input to Fragment Shader

The variable names must match

Example - in shader


```
#version 330

layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec2 tex_coord;

uniform mat4 ctm;

void main() {
    gl_Position = ctm * vec4(vertex, 1.0);
}
```

Example - in application


```
///load the shaders
shader_program_id = loadShaders("../basic.vs", "../basic.fs");

///define geometry
float points[] =    {0.0f, 0.5f, 0.0f,
                     0.5f, -0.5f, 0.0f,
                     -0.5f,-0.5f,0.0f};

///setup the containers
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, 9*sizeof(float), points, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(0);

glBindVertexArray(0);

///get the location of the matrix
ctm_id = glGetUniformLocation(shader_program_id, "ctm");
```

Live demo

- uniform, input variables