CSE 167 (FA21) Computer Graphics: OpenGL

Albert Chern
Graphics API

- Our goals are
  1. Some programming interface to command the graphics card.
  2. Some programming interface to create window and interact with mouse & keyboard etc.

GLUT (OpenGL Utility Toolkit)
GLFW
OS specific language Win32, X11, Cocoa
OpenGL

- OpenGL is a graphics Application Programming Interface (API)
- OpenGL consists of a list of functions that can command the graphics card to do certain things.
- There are many other graphics card APIs.
OpenGL

- OpenGL consists of a list of functions that can command the graphics card to do certain things.

- These functions enable a **rasterization** pipeline.

- OpenGL is just a specification of what functions should exist.

- The actual implementation of OpenGL is done by the graphics card manufacturer.
  - OpenGL is usually not open-source.
  - OpenGL binaries come with the graphics card driver.
OpenGL Utility Toolkit (GLUT)

- GLUT has a list of functions that control window and monitor mouse & keyboard events cross platform.
- GLUT was written by Mark J. Kilgard in the 90’s with the goal of making learning OpenGL programming easier.
  - Only a few lines of code is required to open a window.
  - Does not require knowledge of OS’s windowing system.
- GLUT was no-longer maintained. Open-source replica freeglut is available.

Kilgard 1996
Where is OpenGL & GLUT?

Windows

GL and its extensions come with WindowsSDK

- Need the GL extension wrangler library (GLEW) to properly include modern OpenGL cross-platform and cross-hardware.
- Need to install GLUT or FreeGLUT

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

Linux

GL and its extensions come with Mesa

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

Mac

GL and GLUT come with CommandLineTools

- Apple implements OpenGL, not the graphics card party. GLEW is not needed.

```
#include <OpenGL/gl3.h>
#include <GLUT/glut.h>
```
GLUT HelloWindow
Video game logic

Declare global variables

```
init()
{
    // initialize global variables
}
```

```
main()
{
    Call init();
    While ( True )
        Check keyboard & mouse events
        Draw graphics if needed
    EndWhile
}
```

- Define the state
- Initialize the state
- If certain key is pressed, modify the state.
  Turn on “Redisplay” flag.
HelloGL.cpp

```cpp
#include <OpenGL/gl3.h>
#include <GLUT/glut.h>

static const int width = 400;
static const int height = 300;

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glutSwapBuffers();
    glFlush();
}

void keyboard(unsigned char key, int x, int y)
{
    switch(key){
        case 27: // Escape to quit
            exit(0);
            break;
    }
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitContextVersion(3,1);
    glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
    glutInitWindowSize(width, height);
    glutCreateWindow("Hello Window");

    glClearColor (0.0, 0.2, 0.5, 1.0);
    glViewport(0,0,width,height);

    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);

    glutMainLoop();
    return 0;
}
```
OpenGL Pipeline
The Draw command

• There is one single OpenGL command

\[ \text{glDrawElements(...);} \]

That will

- Take the geometric data that is activated;
- Produce pixel colors stored in a frame buffer.
The Draw command

- There is one single OpenGL command
  
  ```
  glDrawElements( ... );
  ```

  That will
  
  - Take the geometric data that is activated;
  - Produce pixel colors stored in a frame buffer.
Open GL is a state machine

- Open GL is a state machine
  - The pipeline already knows how to process an in-coming data and draw something on the screen.
  - What determines the final image boils down to the state given by which **buffers** and which **shaders** are selected.

A **buffer** is a part of memory for storing data for later access.

A **shader** is a program (a piece of code) that runs on GPU.
OpenGL Legacy v.s. Modern

- The biggest difference between old and modern OpenGL
  - Modern OpenGL gives more control to GPU.
  - These controls are programmable shaders.
  - A shader is a program (a piece of code) that runs on GPU, which is not necessarily related to shading lights and shadows.
Vertex attributes are stored in an array in a vertex buffer. Vertex shader takes this array and apply coordinate transformations (and other things). With IndexBuffer they are assembled into primitives. (There is a Geometry Shader before rasterization that runs over primitives.) Clipping and culling away geometries outside the viewing box. Rasterization turns 3D geometries into fragments, which will be assigned to pixels. Fragment shader computes the pixel color. Pixel colors are placed into some frame buffer (or discarded according to depth test).
CPU v.s. GPU

• OpenGL API allows us to use this pipeline to access GPU

• The rest of our C++ program is performed by CPU

• Type of tasks:
  ▶ Single instruction single data (SISD). CPU is good at SISD.
  ▶ Single instruction multiple data (SIMD). GPU is good at SIMD.

• Examples
  ▶ Host CPU C++ program: manage usr input, physics, ... output set of vertices
  ▶ GPU: coordinate transformation (world, camera, perspective, device coordinate), vertex computation, shading, clipping & culling. Limitation the pipeline is still fixed, vertex computation then linearly interpolated.
Manipulating Data and Program

- Manipulating buffers and shaders are like operating a factory.
OpenGL Graphics Factory
Geometry spreadsheet studio

Graphisch processing unit (GPU)

Vending center

Shader studio

Storage and delivery warehouse
• Double buffering
• There are several other frame buffers (depth, RGB, transparency)
There are 3 main stages:
- Pre-rasterization (per-vertex computation)
- Rasterization
- Post-rasterization (per-fragment computation)

In each stage, the workers execute the same arithmetics; but they are handed in with different input data.

Uniform variables are data that are constant for all workers.
Objects to deal with

- **Geometry spreadsheet** (a.k.a. **Vertex array object**)
  - A table of vertex attributes (data per vertex), and a table of vertex indices per triangle.

- **Buffer object**
  - The geo spreadsheet is actually pointers to this actual storage device
Objects to deal with

- **Shader object**
  - Where we type the shader source code.

- **Program object**
  - A binder that collects all shader objects.
Machines where we can write numbers into a buffer object.

The geometry spreadsheet remembers which buffer corresponds to which attribute.

When the “draw” command is called, the source data are read from these machines.
• Shaders needed to be typed in, compiled, and linked into a program.
Draw a Square
(Code available in HelloSquare.cpp)
Step 0: Declare global variables

`static GLuint program;`  
`static GLuint square_vao;`  
`static GLuint buffers[3];`

- **shader program**
- **geometry spreadsheet**
- **actual storage for geometric data**
- **global variable only within this C++ file.**
1. Geometry Spreadsheet
GLfloat positions[] = {
  -0.5f,-0.5f, // pt0
  0.5f,-0.5f, // pt1
  0.5f, 0.5f, // pt2
  -0.5f, 0.5f // pt3
};
GLfloat colors[] = {
  1.0f, 0.0f, 0.0f, // pt0: red
  1.0f, 1.0f, 0.0f, // pt1: yellow
  0.0f, 1.0f, 0.0f, // pt2: green
  0.0f, 0.0f, 1.0f // pt3: blue
};
GLuint inds[] = {
  0, 1, 3, // first triangle
  2, 3, 1 // second triangle
};
Step 1: Define geometry

```c
glGenVertexArrays(1, &(square_vao));
glGenBuffers(3, buffers);

// activate geometry spreadsheet
glBindVertexArray(square_vao);
```
glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(positions), positions, GL_STATIC_DRAW);

glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, (void*)0);

glBindBuffer(GL_ARRAY_BUFFER, buffers[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof(colors), colors, GL_STATIC_DRAW);

glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);

Step 1: Define geometry
Step 1: Define geometry

```c
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, (void*)0);

glBindBuffer(GL_ARRAY_BUFFER, buffers[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof(colors), colors, GL_STATIC_DRAW);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
```

```c
(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[2]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(inds), inds, GL_STATIC_DRAW);
```
Step 1: Define geometry

```c
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[2]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(inds), inds, GL_STATIC_DRAW);
```

```c
glfwSwapBuffers(window);
```
Step 1: Define geometry

```c
// Define geometry

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[2]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(inds), inds, GL_STATIC_DRAW);

// Unbind the vao to avoid further modification to the geometry spreadsheet.

glBindVertexArray(0);

// END of setting up the square geometry.
```
Step 1: Define geometry

```c
// unbind the vao to avoid further modification to the geometry spreadsheet.
glBindVertexArray(0);

// END of setting up the square geometry.
```

- `square_vao` remembers how the 3 buffers are mapped to the desired attributes.
- Later, when we want to draw the square, we bind `square_vao` and call the draw command.
2. Shading Language (GLSL)
Step 2: Write shaders

**vertex shader**

```glsl
# version 330 core

layout (location = 0) in vec2 pos;
layout (location = 1) in vec3 color;

out vec3 vertexcolor;
out vec2 vertexpos;

void main(){
  gl_Position = vec4(pos.x, pos.y, 0.0f, 1.0f);
  vertexcolor = color;
  vertexpos = pos;
}
```

**fragment shader**

```glsl`
#version 330 core

in vec3 vertexcolor;
in vec2 vertexpos;

out vec4 fragColor;

void main (void){
  fragColor = vec4(vertexcolor, 1.0f);
}
```

- First line: version
- Variable declaration: in, out, uniform.
- Additional keyword: layout( location = … )
- The input of vtx shader comes from vertex array. The input of frag shader comes from the output of vtx shader.
- vtx shader needs to set gl_Position.
const char *vertShaderSrc =
R"("# version 330 core
layout (location = 0) in vec2 pos;
layout (location = 1) in vec3 color;
out vec3 vertexcolor;
out vec2 vertexpos;

void main(){
    gl_Position = vec4(pos.x, pos.y, 0.0f, 1.0f);
    vertexcolor = color;
    vertexpos = pos;
}
")";

const char *fragShaderSrc =
R"("#version 330 core
in vec3 vertexcolor;
)";
Step 2: Write shaders

```cpp
vertexcolor = color;
vertexpos = pos;
}
)
const char *fragShaderSrc =
R"(
#version 330 core

in vec3 vertexcolor;
in vec2 vertexpos;

out vec4 fragColor;

void main (void){
    fragColor = vec4(vertexcolor, 1.0f);
}
)";
```
Step 2: Write shaders

GLuint vs = glCreateShader(GL_VERTEX_SHADER);
GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
Step 2: Write shaders

```c
GLuint vs = glCreateShader(GL_VERTEX_SHADER);  
GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);  

glShaderSource(vs, 1, &vertShaderSrc, NULL);  
glShaderSource(fs, 1, &fragShaderSrc, NULL);  
```
Step 2: Write shaders

```c
GLuint vs = glCreateShader(GL_VERTEX_SHADER);
GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(vs, 1, &vertShaderSrc, NULL);
glShaderSource(fs, 1, &fragShaderSrc, NULL);
glCompileShader(vs);
glCompileShader(fs);
```

```c
program = glCreateProgram();
glAttachShader(program, vs);
glAttachShader(program, fs);
glLinkProgram(program);
glDeleteShader(vs);
glDeleteShader(fs);
```
Step 2: Write shaders

```cpp
GLuint vs = glCreateShader(GL_VERTEX_SHADER);
GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);

glShaderSource(vs, 1, &vertShaderSrc, NULL);
glShaderSource(fs, 1, &fragShaderSrc, NULL);

glCompileShader(vs);
glCompileShader(fs);

program = glCreateProgram();
glAttachShader(program, vs);
glAttachShader(program, fs);
glLinkProgram(program);
glDeleteShader(vs);
glDeleteShader(fs);
// END compiling shaders
```
3. Draw Command
void display(void){
    glClear(GL_COLOR_BUFFER_BIT);

    // BEGIN draw square
    glBindVertexArray(square_vao);
    glUseProgram(program);
    glDrawElements(GL_TRIANGLES, // cookbook
                   6, // length of the array inds[]
                   GL_UNSIGNED_INT,
                   0
                   );
    // END draw square

    glutSwapBuffers();
    glFlush();
}

Step 3: Draw
void display(void){
    glClear(GL_COLOR_BUFFER_BIT);

    // BEGIN draw square
    glBindVertexArray(square_vao);
    glUseProgram(program);
    glDrawElements(GL_TRIANGLES, // cookbook
                   6, // length of the array inds[]
                   GL_UNSIGNED_INT,
                   0);

    // END draw square

    glutSwapBuffers();
    glFlush();
}
Uniform Variables
Goal

(Code available in HelloSquare.cpp)
fragment shader

#version 330 core
in vec3 vertexcolor;
in vec2 vertexpos;

out vec4 fragColor;

void main (void){
    fragColor = vec4(vertexcolor, 1.0f);
}
fragment shader

#version 330 core
in vec3 vertexcolor;
in vec2 vertexpos;

uniform float circleradius;
uniform vec3 circlecolor;

out vec4 fragColor;

void main (void){
    fragColor = vec4(vertexcolor,1.0f);
}
#version 330 core
in vec3 vertexcolor;
in vec2 vertexpos;

uniform float circleradius;
uniform vec3 circlecolor;

out vec4 fragColor;

void main (void){
    fragColor = vec4(vertexcolor, 1.0f);
    if (length(vertexpos)<circleradius){
        fragColor = vec4(circlecolor, 1.0f);
    }
}

Pass uniform vars from C++ to shader

Global variables

```c++
static GLfloat myradius = 0.2;
static GLuint circleradius_loc;
static glm::vec3 mycolor = glm::vec3(1.0,1.0,1.0);
static GLuint circlecolor_loc;
```

After compile program (in init)

```c++
circleradius_loc = glGetUniformLocation( program, "circleradius" );
circlecolor_loc = glGetUniformLocation( program, "circlecolor" );
```

Set uniform values (in display)

```c++
glUseProgram(program);
glUniform1f(circleradius_loc, myradius);
glUniform3f(circlecolor_loc, mycolor[0], mycolor[1], mycolor[2]);
```

last line is equivalent to `glUniform3fv(circlecolor_loc, 1, &mycolor[0]);`
What we have learned so far
What we have learned so far

- We have learned almost everything for OpenGL programming!
  - Read the code from HW0.
  - At low level, everything in graphics is to set up buffers and shaders, and call “draw” at the right time.

- The remaining topics are abstraction and mathematics:
  - When there are more objects to draw, the code will be very messy. We need to group the variables and functionalities in a semantically meaningful way for a cleaner OpenGL code.
  - How do we set the gl_Position for each vertex so that the image looks 3D?
Object oriented OpenGL code
static GLuint program;
static GLuint square_vao;
static GLuint buffers[3];
static GLfloat myradius = 0.2;
static GLuint circleradius_loc;
static glm::vec3 mycolor =
    glm::vec3(1.0, 1.0, 1.0);
static GLuint circlecolor_loc;
Lots of global variables

```c
static GLuint program;
static GLuint square_vao;
static GLuint buffers[3];
static GLfloat myradius = 0.2;
static GLuint circleradius_loc;
static glm::vec3 mycolor = glm::vec3(1.0,1.0,1.0);
static GLuint circlecolor_loc;
```

owned by the “square” geometry

a special shader with a particular set of parameter interface
#include <vector>
#ifndef __GEOMETRY_H__
#define __GEOMETRY_H__
class Geometry {
public:
    GLenum mode = GL_TRIANGLES; // the cookboook for glDrawElements
    int count; // number of elements to draw
    GLenum type = GL_UNSIGNED_INT; // type of the index array
    GLuint vao; // vertex array object a.k.a. geometry spreadsheet
    std::vector<GLuint> buffers; // data storage

    // meaningful implementations only in inherited classes
    virtual void init(){};

    void draw(void){
        glBindVertexArray(vao);
        glDrawElements(mode,count,type,0);
    }
};
#endif
```cpp
#include "Geometry.h"
#ifndef __SQUARE_H__
#define __SQUARE_H__

class Square : public Geometry {
public:
    void init() {
        ...
    }
};
#endif
```
#include "Square.h"

static Square square;

Initialization

// Initialize square
square.init();

Draw

void display(void){
    glClear(GL_COLOR_BUFFER_BIT);
    // BEGIN draw square
    square.draw();
    // END draw square
    // BEGIN display loop
    glutSwapBuffers();
    glFlush();
}
class Shader {
public:
    GLuint program; // the shader program
    std::string vertexshader_source; // source code
    std::string fragmentshader_source; // source code

    void read_source(const char * vertexshader_filename,
                     const char * fragmentshader_filename);
    void compile();
    GLint getVertexShaderCompileStatus(){return compiled_vs;}
    GLint getFragmentShaderCompileStatus(){return compiled_fs;}
    GLint getLinkStatus(){return linked;}

private:
    ...
};
class Shader {
public:
    GLuint program; // the shader program
    std::string vertexshader_source; // source code
    std::string fragmentshader_source; // source code

    void read_source(const char * vertexshader_filename,
                     const char * fragmentshader_filename);
    void compile();
    GLint getVertexShaderCompileStatus(){return compiled_vs;}
    GLint getFragmentShaderCompileStatus(){return compiled_fs;}
    GLint getLinkStatus(){return linked;}

private:
    ...
};
class Shader {
public:
    GLuint program;  // the shader program
    std::string vertexshader_source;   // source code
    std::string fragmentshader_source;  // source code

    void read_source(const char * vertexshader_filename,
                     const char * fragmentshader_filename);
    void compile();
    GLint getVertexShaderCompileStatus(){return compiled_vs;}
    GLint getFragmentShaderCompileStatus(){return compiled_fs;}
    GLint getLinkStatus(){return linked;}

private:
    ...
};
Inherited classes of Shader

- Specialized shaders are inherited classes of the Shader class, they have the customized uniform variables.

```cpp
class CircleShader : public Shader {
public:
  GLfloat radius = 0.2; GLuint radius_loc;
  glm::vec3 color = glm::vec3(1.0, 1.0, 1.0); GLuint color_loc;
  void initUniforms() {
    radius_loc = glGetUniformLocation(program, "circleradius");
    color_loc = glGetUniformLocation(program, "circlecolor");
  }
  void setUniforms() {
    glUniform1f(radius_loc, radius);
    glUniform3f(color_loc, color[0], color[1], color[2]);
  }
};
```
Using Square and CircleShader classes

Global variable

```java
static Square square;
static CircleShader shader;
```

Initialization

```java
// Initialize square
square.init();

// Initialize shader
shader.read_source("shaders/hello.vert", "shaders/hello.frag");
shader.compile();
glUseProgram(shader.program);
shader.initUniforms();
```

Draw

```java
void display(void){
    glClear(GL_COLOR_BUFFER_BIT);

    // BEGIN draw square
    shader.setUniforms();
    square.draw();
    // END draw square

    glutSwapBuffers();
    glFlush();
}
```
HW1: Mandelbrot Shader
Global variable

```cpp
static Square canvas;
static MandelbrotShader shader;
```

Initialization

```cpp
// Initialize square
square.init();
// Initialize shader
shader.read_source("shaders/simple.vert",
    "shaders/Mandelbrot.frag");
shader.compile();
glUseProgram(shader.program);
shader.initUniforms();
```

Draw

```cpp
display(void){
    glClear(GL_COLOR_BUFFER_BIT);
    // BEGIN draw square
    shader.setUniforms();
square.draw();
    // END draw square
    glutSwapBuffers();
    glutFlush();
}
```
In the fragment shader

- For each pixel of the “canvas” evaluate a given mathematical algorithm.

- The resulting value will vary across different pixels, forming a fractal on the canvas.
Mandelbrot iteration

**Input:** \( c \in \mathbb{C}; \)

1: \( z_0 = 0 \in \mathbb{C}; \)
2: for \( k = 0, 1, 2, \ldots \) do
3: \( z_{k+1} = z_k^2 + c; \)
4: end for

- Each pixel corresponds to a complex value \( c \).
- The color shows the *number of iteration* it takes for the iteration to produce \( |z_k| \geq 2 \).
Mandelbrot fractal