Computer Graphics (cse167)
OpenGL
Albert Chern
OpenGL

• OpenGL is a graphics Application Programming Interface (API)
  ▶ Many different graphics API, like DirectX, Vulkan, WebGL
  ▶ List of standardized functions, specification maintained by Khronos group, implemented by your hardware manufacturer.
  ▶ GLEW is a library providing headers that finds the binary of the graphics card driver containing the implementation of the functions
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.
OpenGL is a state machine

- OpenGL 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.
OpenGL Pipeline

- 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 primitive.
- Clipping and culling away geometries outside the frustum (outside the clipping box).
- Rasterization turns 3D geometries into data (fragment) required for coloring pixels.
- Fragment shader computes the pixel color.
- Pixel colors are placed into some frame buffer.
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.
The OpenGL Factory
Frame buffers

- Store final pixel colors
- Usually use double buffers
Empty buffers, spreadsheets, shaders, programs

- These are the raw ingredients for all geometry, material etc.
The GL_ARRAY_BUFFER machine

- We can insert one cassette tape (buffer)
- We can write values into it
- We can print value from the tape
- We can format the printout as a spreadsheet
The shader workshop

- There is a typewriter
- A compiler
- A linker
The GPU workers

- Listen to the command in the program
- Take data from the spreadsheet
- Look at the same uniform variables
The Draw command

- There are several cookbooks
- We will use GL_TRIANGLES
1. GLUT
2. Draw command
3. Geometry data
4. Geometry spreadsheet (VAO)
5. Element/index array
6. Shaders
7. Uniform variables

OpenGL Code:
1. GLUT
OpenGL Program

• Core OpenGL functions and constants (gl…, GL…)
  ▶ Commands that control the drawing pipeline, e.g. glBindBuffer(…).
  ▶ OpenGL does not provide any mechanism to manage windows, user inputs etc.

• Utility libraries for managing windows, I/O.
  ▶ GLUT: native event loops & callbacks. Not open-source and is no longer maintained.
  ▶ FreeGLUT: open source replacement of GLUT.
  ▶ GLFW: light weight, programmer has more control over the event loops.
  ▶ SDL: popular in game development, portable not only to desktops but also Android & iOS.
GLUT setup: Window and frame buffer

```c
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    // Requests the type of buffers.
    // Think about what buffers you would need...
    // OSX systems require another flag
    #ifdef __APPLE__
        glutInitDisplayMode (GLUT_3_2_CORE_PROFILE | GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    #else
        glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    #endif
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow ("Simple Demo with Shaders");

    #ifndef __APPLE__ // Do not use GLew on OSX systems!
    GLenum err = glewInit();
    if (GLEW_OK != err) {
        std::cerr << "Error: " << glewGetString(err) << std::endl;
    } #endif

    init();
    ...
```
GLUT setup: Register callbacks

```c
int main(int argc, char** argv)
{
    ...
    init();

    // Now, we define callbacks and functions for various tasks.
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMouseFunc(mouse);
    glutMotionFunc(mousedrag);
    glutIdleFunc(animate); // or glutIdleFunc(NULL);

    glutMainLoop(); // Start the main code

    deleteBuffers(); // helper to delete buffers generated in init()

    return 0; /* ANSI C requires main to return int. */
}
```

- **init()**: create buffers, parse & compile shaders, initialize all parameters
- **display()**: plug in the desired buffers, draw command, will be called whenever glut thinks redisplay is needed.
// Defines a Mouse callback to zoom in and out
// This is done by modifying gluLookAt
// The actual motion is in mousedrag
// mouse simply sets state for mousedrag
void mouse(int button, int state, int x, int y)
{
    if (button == GLUT_LEFT_BUTTON) {
        if (state == GLUT_UP) {
            // Do Nothing ;
        }
    }
    else if (state == GLUT_DOWN) {
        mouseoldx = x ; mouseoldy = y ; // so we can move wrt x , y
    }
    else if (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
    {
        // Reset gluLookAt
        eyeloc = 2.0 ;
        modelview = glm::lookAt(glm::vec3(0, -eyeloc, eyeloc), glm::vec3(0, 0, 0), glm::vec3(0, 1, 1));
        // Send the updated matrix to the shader
        glUniformMatrix4fv(modelviewPos, 1, GL_FALSE, &(modelview)[0][0]);
        glutPostRedisplay();
    }
}
OpenGL Code:

2. Where is the Draw Command

1. GLUT
2. Draw command
3. Geometry data
4. Geometry spreadsheet (VAO)
5. Element/index array
6. Shaders
7. Uniform variables
// Draws the teapot.
void drawteapot() {
    glBindVertexArray(teapotVAO);
    glDrawElements(GL_TRIANGLES, teapotIndices.size(), GL_UNSIGNED_INT, 0);
    glBindVertexArray(0);
}

void display(void)
{
    // clear all pixels
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    ... 
    drawteapot();
    ... 
    glutSwapBuffers();
}
• How do we setup the buffers and shaders to generate the following 2D colored square?
OpenGl Code:

3. Geometry Data

1. GLUT
2. Draw command
3. Geometry data
4. Geometry spreadsheet (VAO)
5. Element/index array
6. Shaders
7. Uniform variables
Vertex buffer

- At initialization stage (called in init())
- Each vertex has a position attribute and a color attribute

```c
GLfloat vtx_attribs[] = {
    -0.5, -0.5, 1.0, 0.0, 0.0, // pt: red
    0.5, -0.5, 1.0, 1.0, 0.0, // pt1: yellow
    -0.5, 0.5, 0.0, 0.0, 1.0, // pt3: blue
    0.5, 0.5, 0.0, 1.0, 0.0, // pt2: green
    -0.5, 0.5, 0.0, 0.0, 1.0, // pt3: blue
    0.5, -0.5, 1.0, 1.0, 0.0 // pt1: yellow
};
```
Vertex buffer

```c
GLfloat vtx_attribs[] = {
    -0.5, -0.5, 1.0, 0.0, 0.0, // pt0: red
    0.5, -0.5, 1.0, 1.0, 0.0, // pt1: yellow
    -0.5,  0.5, 0.0, 0.0, 1.0, // pt3: blue
    0.5,  0.5, 0.0, 1.0, 0.0, // pt2: green
    -0.5,  0.5, 0.0, 0.0, 1.0, // pt3: blue
    0.5, -0.5, 1.0, 1.0, 0.0 // pt1: yellow
};

‣ Create a buffer object to store the data.
‣ Use the GL_ARRAY_BUFFER machine to write the data.

GLuint VBO;
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, // which buffer machine
             sizeof(vtx_attribs), // number of bytes
             vtx_attribs, // actual data
             GL_STATIC_DRAW); // mode
```
Vertex buffer

GLfloat vtx_attribs[] = {
-0.5, -0.5, 1.0, 0.0, 0.0, // pt0: red
0.5, -0.5, 1.0, 1.0, 0.0, // pt1: yellow
-0.5, 0.5, 0.0, 0.0, 1.0, // pt3: blue
0.5, 0.5, 0.0, 1.0, 0.0, // pt2: green
-0.5, 0.5, 0.0, 0.0, 1.0, // pt3: blue
0.5, -0.5, 1.0, 1.0, 0.0 // pt1: yellow
};

GLuint VBO;
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER,
    sizeof(vtx_attribs), // number of bytes
    vtx_attribs, // actual data
    GL_STATIC_DRAW); // mode

▶ If we are done with the data, we can eject the buffer object from the GL_ARRAY_BUFFER machine

glBindBuffer(GL_ARRAY_BUFFER, 0);
Later in the main display() callback, when we want to draw this geometry, we insert VBO back to GL_ARRAY_BUFFER.

```cpp
glBindBuffer(GL_ARRAY_BUFFER, VBO);

glUseProgram(shaderProgram);  // we will talk about it later
glDrawArrays(GL_TRIANGLES,  // which cookbook
    0,  // starting index in the array
    6   // count of vertices
);

GLfloat vtx_attribs[] = {
    -0.5, -0.5, 1.0, 0.0, 0.0, // pt0: red
    0.5, -0.5, 1.0, 1.0, 0.0,  // pt1: yellow
    -0.5, 0.5, 0.0, 0.0, 1.0,  // pt3: blue
    0.5, 0.5, 0.0, 1.0, 0.0,  // pt2: green
    -0.5, 0.5, 0.0, 0.0, 1.0,  // pt3: blue
    0.5, -0.5, 1.0, 1.0, 0.0,  // pt1: yellow
};
```
glBindBuffer(GL_ARRAY_BUFFER, VBO);

- Need to format the printout of GL_ARRAY_BUFFER so that the program can interpret.
  
  // Use layout location 0 for position
  glEnableVertexAttribArray(0);
glVertexAttribPointer(0, // attribute
  2, // num elems per vtx, here (x,y)
  GL_FLOAT, // type of each element
  GL_FALSE, // converting int to float?
  5 * sizeof(GLfloat), // this attrib appears every 5 floats
  0 // offset
);

// Use layout location 1 for color
  glEnableVertexAttribArray(1);
glVertexAttribPointer(1, // attribute
  3, // num elems per vtx, here (r,g,b)
  GL_FLOAT, // type of each element
  GL_FALSE, // converting int to float?
  5 * sizeof(GLfloat), // this attrib appears every 5 floats
  2 * sizeof(GLfloat) // offset
);

glUseProgram(shaderProgram); // we will talk about it later
glDrawArrays(GL_TRIANGLES, // which cookbook
  0 // starting index in the array
);
1. GLUT
2. Draw command
3. Geometry data
4. Geometry spreadsheet (VAO)
5. Element/index array
6. Shaders
7. Uniform variables

OpenGL Code:

4. Geometry Spreadsheet (VertexArrayObject)
The concept of **geometry spreadsheet**, a.k.a. **vertex array** is a stream of data together with the attribute layout.

The next stage of the pipeline, the vertex shader, is looking for a geometry spreadsheet.

We saw that, to obtain the geometry spreadsheet, we had to insert the buffer to GL_ARRAY_BUFFER machine, and format it every time we want to execute the draw command.

A **Vertex Array Object** (VAO) is a geometry spreadsheet.

Formatting the printout of GL_ARRAY_BUFFER can all be done at the init() stage. The resulting geo spreadsheet is remembered as a VAO.

At run time (display()) we just need to tell GL which VAO to use.
Instead of putting all data in one buffer, we have a more organized initialization.

```c
GLfloat positions[] = {
    -0.5, -0.5, // pt0
    0.5, -0.5, // pt1
    -0.5, 0.5, // pt3
    0.5, 0.5, // pt2
    -0.5, 0.5, // pt3
    0.5, -0.5, // pt1
};
GLfloat colors[] = {
    1.0, 0.0, 0.0, // pt0: red
    1.0, 1.0, 0.0, // pt1: yellow
    0.0, 0.0, 1.0, // pt3: blue
    0.0, 1.0, 0.0, // pt2: green
    0.0, 0.0, 1.0, // pt3: blue
    1.0, 1.0, 0.0 // pt1: yellow
};
```

Create two buffers for both attributes.

Create one vertex array object as the spreadsheet.

```c
GLuint VAO;
GLuint buffers[2];
glGenVertexArrays(1, &VAO);
glGenBuffers(2, buffers);
```
Building geometry spreadsheet

GLint VAO;
GLuint buffers[2];
glGenVertexArrays(1, &VAO);
glGenBuffers(2, buffers);

- Write data and assign location in the spreadsheet.

glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(positions), positions, GL_STATIC_DRAW);
glEnableVertexAttribArray(0); // layout location 0 for position
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(GLfloat), 0);

glBindBuffer(GL_ARRAY_BUFFER, buffers[1]); // this would eject buffers[0]
glBufferData(GL_ARRAY_BUFFER, sizeof(colors), colors, GL_STATIC_DRAW);
glEnableVertexAttribArray(1); // layout location 1 for color
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), 0);

glBindVertexArray(0); // unbind to prevent further modification
To draw the object at the display stage, we just need activate the geometry spreadsheet

```c
glBindVertexArray(VAO);
glUseProgram(shaderProgram);
glDrawArrays(GL_TRIANGLES, 0, 6);
```
1. GLUT
2. Draw command
3. Geometry data
4. Geometry spreadsheet (VAO)
5. Element/index array
6. Shaders
7. Uniform variables

OpenGL Code:
5. Element/Index Array
In this example, pt1 and pt3 are repeated (shared by 2 triangles).

```c
GLfloat positions[] = {
    -0.5, -0.5, // pt0
    0.5, -0.5, // pt1
    -0.5, 0.5, // pt3
    0.5, 0.5, // pt2
    -0.5, 0.5, // pt3
    0.5, -0.5 // pt1
};

GLfloat colors[] = {
    1.0, 0.0, 0.0, // pt0: red
    1.0, 1.0, 0.0, // pt1: yellow
    0.0, 0.0, 1.0, // pt3: blue
    0.0, 1.0, 0.0, // pt2: green
    0.0, 0.0, 1.0, // pt3: blue
    1.0, 1.0, 0.0 // pt1: yellow
};
```
Building geometry spreadsheet

- 4 vertices and 2 triangles

```c
GLfloat positions[] = {
    -0.5, -0.5, // pt0
    0.5, -0.5, // pt1
    0.5, 0.5, // pt2
    -0.5, 0.5 // pt3
};
GLfloat colors[] = {
    1.0, 0.0, 0.0, // pt0: red
    1.0, 1.0, 0.0, // pt1: yellow
    0.0, 1.0, 0.0, // pt2: green
    0.0, 0.0, 1.0 // pt3: blue
};
GLubyte inds[] = {
    0, 1, 3, // first triangle
    2, 3, 1 // second triangle
};
```
4 vertices and 2 triangles

```cpp
GLfloat positions[] = {
    -0.5, -0.5, // pt0
    0.5, -0.5, // pt1
    0.5, 0.5,  // pt2
    -0.5, 0.5  // pt3
};
GLfloat colors[] = {
    1.0, 0.0, 0.0, // pt0: red
    1.0, 1.0, 0.0, // pt1: yellow
    0.0, 1.0, 0.0, // pt2: green
    0.0, 0.0, 1.0  // pt3: blue
};
GLubyte inds[] = {
    0, 1, 3,  // first triangle
    2, 3, 1   // second triangle
};
```
Building geometry spreadsheet

GLuint VAO;
GLuint buffers[3];
glGenVertexArrays(1, &VAO);
glGenBuffers(3, buffers);

glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(positions), positions, GL_STATIC_DRAW);
glEnableVertexAttribArray(0); // layout location 0 for position
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2*sizeof(GLfloat), 0);

glBindBuffer(GL_ARRAY_BUFFER, buffers[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof(colors), colors, GL_STATIC_DRAW);
glEnableVertexAttribArray(1); // layout location 1 for color
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3*sizeof(GLfloat), 0);

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

glBindVertexArray(0); // unbind to prevent further modification
Later in `display()`, use `glDrawElements` (instead of `glDrawArrays`)

```c
glBindVertexArray(VAO);
glUseProgram(shaderProgram);
glDrawElements(GL_TRIANGLES, // primitive type
               6, // number of indices in the element array
               GL_UNSIGNED_BYTE, // type of index array
               0); // offset in byte (casted to pointer)
```
1. GLUT
2. Draw command
3. Geometry data
4. Geometry spreadsheet (VAO)
5. Element/index array
6. Shaders
7. Uniform variables

OpenGL Code:
6. Shaders
• Shaders are programs running on GPU.

• The two main shaders that will be executed are the vertex shader and the fragment shader.

• OpenGL shaders are written in a C-like language called GLSL.
**vertex shader**

```
#version 330 core

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

// Extra outputs, if any
out vec3 color;

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

**fragment shader**

```
#version 330 core

// Inputs to the fragment shader are outputs of the same name of the vertex shader
in vec3 color;

// Output the frag color
out vec4 fragColor;

void main (void) {
    fragColor = vec4(color, 1.0f);
}
```
In our C++ code init() stage

```cpp
const char *vertShaderSrc = "#version 330 core\n" "layout (location = 0) in vec2 pos;\n" "layout (location = 1) in vec3 color;\n" "out vec3 color;\n"
"void main(){\n" "   gl_Position = vec4(pos.x, pos.y, 0.0f, 1.0f);\n"
"};
const char *fragShaderSrc = "#version 330 core\n" "in vec3 color;\n" "out vec4 fragColor;\n"
"void main (void){\n" "   fragColor = vec4(color,1.0f);\n"
"};
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);
GLuint program = glCreateProgram();
glAttachShader(program, vs);
glAttachShader(program, fs);
glLinkProgram(program);
glDeleteShader(vs);
glDeleteShader(fs);
```

- This process is usually written in helper functions that read the external source code and do the compilation routine.
Use Program

Later in display():

```c
glBindVertexArray(VAO);
glUseProgram(program);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, 0);
```
1. GLUT
2. Draw command
3. Geometry data
4. Geometry spreadsheet (VAO)
5. Element/index array
6. Shaders
7. Uniform variables

OpenGL Code:
7. Uniform Variables
Uniform variables

- vertex shader

```glsl
#version 330 core

// Inputs
layout (location = 0) in vec2 position;
layout (location = 1) in vec3 color;

// Uniforms
uniform mat4 projection;
uniform mat4 modelview;

// Extra outputs, if any
out vec3 color;

void main() {
    gl_Position = projection * modelview * vec4(position.x, position.y, 0.0f, 1.0f);
}
```

Data that vary over different vertices are **vertex attributes**, coming from the geometry spreadsheet.

Other parameters that are independent of vertices, passed over from our C++ program as **uniforms**.
void init(void){
    ... // compile shader, link program etc
    GLuint projectionPos, modelviewPos; // Locations of uniform variables
    projectionPos = glGetUniformLocation(program,"projection");
    modelviewPos = glGetUniformLocation(program,"modelview");
    ...
}

void display(void){
    ... // compute projection P, modelview matrix M
    glUseProgram(program);
    glUniformMatrix4fv(projectionPos, 1, GL_FALSE, &(P)[0][0]);
    glUniformMatrix4fv(modelviewPos, 1, GL_FALSE, &(M)[0][0]);
    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, 0);
    ...
}
Next time

• Lighting
• Matrix stacks