SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

50.017 Graphics and Visualization

Programming with C++ and OpenGL

Peng Song, ISTD

Week 1 – Cohort Class

May – August Term, 2021

Outline

1. C++ Programming
2. Modern OpenGL
3. Window API

Outline

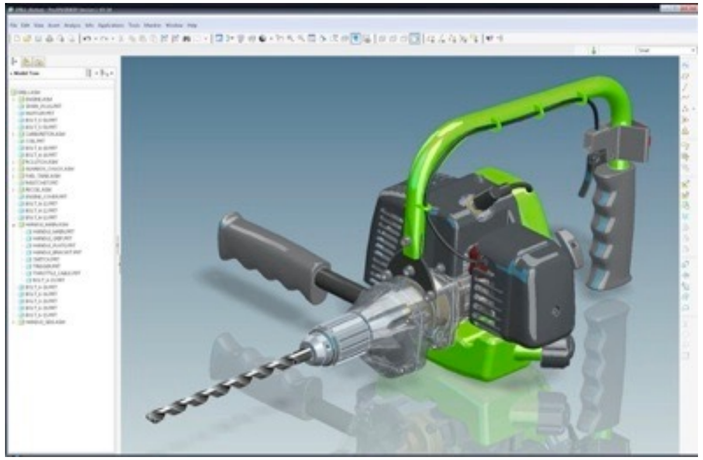
1. **C++ Programming**
2. Modern OpenGL
3. Window API

C++

Widely used in the CG industry

- large application
- interactivity

game



CAD

visualization

C++ vs Python

	C++	**Python**
Implementation	Compiled code	Interpreted code
Runtime speed	Fast	A lot slower than C++
Memory management	Does not support garbage collection	Support garbage collection
Development	IDE is necessary to develop in C++	IDE is not necessary
Maintenance	C++ code is hard to understand	Code is easy to understand
Usage	Large applications, embedded systems	University studies, proof of concept, small apps

Variable Declarations

- Variables must be declared with their types before first use!

```
int i = 1;                    // signed integer
float f = 1.0f;               // single-precision (32-bit) floating point
double d = 1.0;               // double-precision (64-bit) floating point
```

- Variables' types never change
  - Their *values* can be converted to other types:

```
int  truncated = f; // truncation occurred without warning;
double promoted = f;
```

Variable Scope

- Variables are in scope from after declaration until the end of the enclosing curly braces (exception: class members):

```
if (true) {
    // i, j undeclared here
    int i = 0;
    // i visible here
    {
        int j = 0;
        // i, j visible here
    }
    // j out of scope here
}
// i, j out of scope here
```

- Variables are destroyed when they go out of scope
- Extra curly braces can be useful to limit scope/control cleanup

Variable Scope

```
#include <iostream>
int main(int argc, const char *argv[]) {
    int i = 1;

    {
        int i = 2;
        std::cout << i << '\n';
    }

    std::cout << i << '\n';
    return 0;
}
```

- Does this compile?
  - Yes, without warnings! What does it print?
- Potential source of bugs, e.g., with loop variables hiding surrounding variables!

Functions in C++

- Function declarations list return value and argument types
- Functions must be declared before they are called

```
// f invisible here; calling it would be an error

void f(int a) { }


// f now visible
void g(int a) {
    f(a); // OK; f declared (and defined) above
}
```

Declaration vs Definition

- Function can be declared without definition (implementation)
- This is called a “function prototype”:

```
void f(int a);

void g(int a);
// f and g visible here (but aren't defined yet!)
```

- Why have prototypes?
  - Provide the callable interface for a code library (without letting customers see implementation)
  - Modularize your code into separate `.cpp` files for faster compilation

Program Entry Point

- The main function:

```
#include <iostream>
int main() {
    std::cout << "Hello World!";
    return 0;  // exit status (0 for success)
}
```

```
Hello World!
```

Resource for Learning C++

- W3 schools C++ tutorial
  - https://www.w3schools.com/CPP/default.asp

- GeeksforGeeks
  - https://www.geeksforgeeks.org/c-plus-plus/

Outline

1. C++ Programming
2. **Modern OpenGL**
3. Window API

Computer Graphic APIs

- **Low-level rendering** (rendering commands)
  - **OpenGL** (**Open Graphics Library**)
  - Direct3D
  - Vulkan

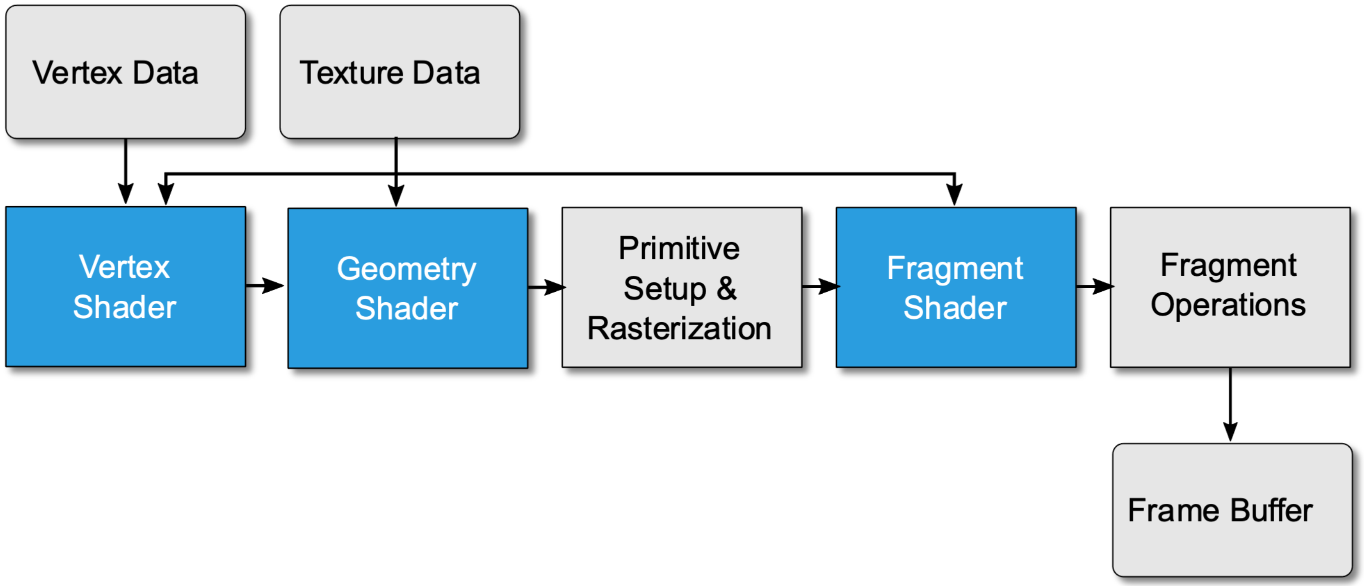
- High level scene graphs (scene description)
  - OpenInventor
  - OpenSG, InstantReality
  - OpenSceneGraph
  - Java3D
  - …

What is OpenGL?

- Computer Graphics Rendering API
  - render images from geometry and image data
  - process simple primitives (points, lines, triangles)
  - create interactive 3D graphics applications!
  - see http://www.opengl.org
- OpenGL is
  - operating system independent (Linux, Mac, Win)
  - window system independent (X-Windows, …)
  - hardware accelerated (NVIDIA, ATI/AMD, Intel, …)

Modern OpenGL

- Modern OpenGL refers to OpenGL 3.0 and higher
- We will focus on OpenGL 3.2 (and higher).

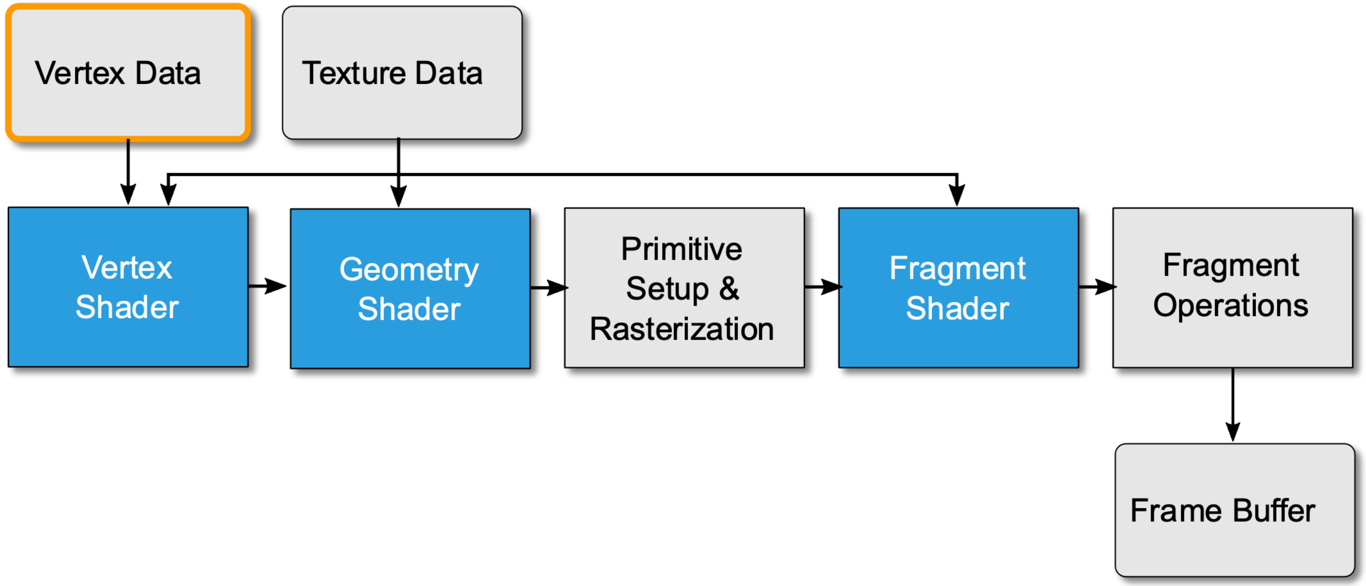


Why not old OpenGL?

GPU, OS	old OpenGL immediate mode	new OpenGL vertex buffer objects
NVIDIA GeForce 9400M, Mac OS X	12	56
NVIDIA GeForce GT 120, Mac OS X	27	216
NVIDIA GeForce GTX 285, Windows 7	10	315
ATI Radeon HD 5680, Mac OS X	20	592
NVIDIA GeForce GTX 580, Linux	7	600

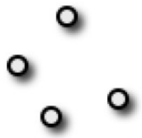
Rendering performance in million triangles per second, measured on a model with 400k triangles

OpenGL 3.2 Pipeline

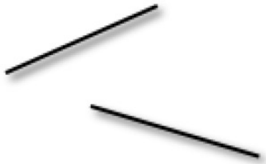


- How to store & send geometry data?

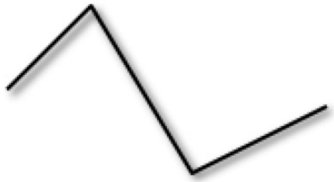
Primitives in Core Profile



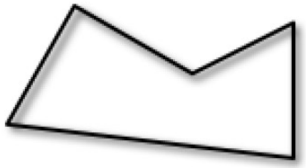
`GL_POINTS`



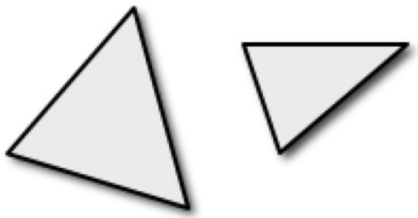
`GL_LINES`



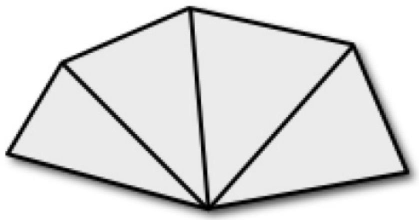
`GL_LINE_STRIP`



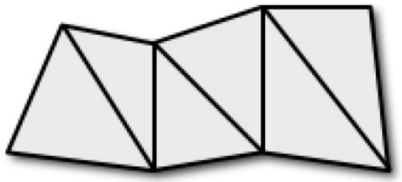
`GL_LINE_LOOP`



`GL_TRIANGLES`



`GL_TRIANGLE_FAN`



`GL_TRIANGLE_STRIP`

Geometric Objects

- Vertex data must be stored in *vertex arrays* (VAs)
  – No more (inefficient) immediate mode rendering!
- VAs must be stored in *vertex buffer objects* (VBOs)
  – VBOs are vertex arrays that are stored on the GPU
- VBOs must be stored in *vertex array objects* (VAOs)
  – VAOs manage all VBOs and VertexAribPointers
  – Rendering: Simply enable VAO per geometric object

Store Triangle Data in VBO/VAO

```
float vertices[] = {
    // positions         // colors
     0.5f, -0.5f, 0.0f,  0.4f, 1.0f, 0.4f,  // bottom right
    -0.5f, -0.5f, 0.0f,  0.4f, 1.0f, 0.4f,  // bottom left
     0.0f,  0.5f, 0.0f,  0.4f, 1.0f, 0.4f   // top
};

unsigned int VBO, VAO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);

// bind the Vertex Array Object first, then bind and set vertex buffer(s), and then configure
// vertex attributes(s).
glBindVertexArray(VAO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);
```

Geometric Objects

- Render using vertex arrays

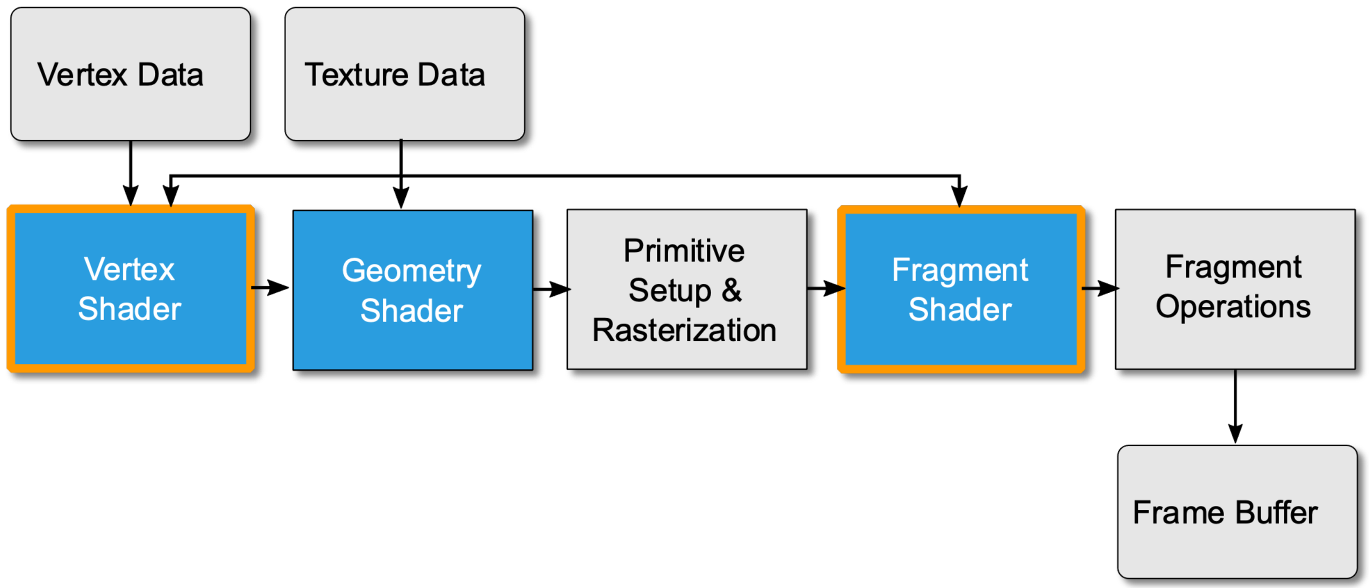
```
// activate VAO (this activates all VBOs and attrib pointers)
glBindVertexArray(VAO);
// draw triangles
glDrawArrays(GL_TRIANGLES, 0, 3);
```

- Delete buffers and vertex array in the end

```
// delete vertex buffer objects
glDeleteBuffers(1, &VBO);
// delete vertex array object
glDeleteVertexArrays(1, &VAO);
```

OpenGL 3.2 Pipeline

- Vertex & Fragment Shaders



Vertex Shader

- Input
  - vertex attributes (position, normal, color, ...)
  - constant (uniform) parameters
- Output
  - required: vertex’ clip space coordinates
    (after model, view, and projection transformation)
  - optional: vertex colors, texture coordinates, point size, ...
  - outputs are interpolated and sent to fragment shader
- One vertex in, one vertex out
  - No connectivity information

Vertex Shader

- Vertex shader in Assignment 0

```
const char *vertexShaderSource ="#version 330 core\n"
    "layout (location = 0) in vec3 aPos;\n"
    "layout (location = 1) in vec3 aColor;\n"
    "out vec3 ourColor;\n"
    "void main()\n"
    "{\n"
    "   gl_Position = vec4(aPos, 1.0);\n"
    "   ourColor = aColor;\n"
    "}\0";
```

Vertex Shader

- What we have to do ourselves
  - modelview and projection transform
  - normal transformation
  - per-vertex lighting (for Gouraud shading)
- What we cannot do ourselves
  - clipping
  - perspective division
  - viewport transformation

Fragment Shader

- Input
  - fragment’s window position
  - primary / secondary colors
  - texture coordinates
  - for per-pixel lighting: data for Phong model
  - all input is interpolated from vertex shader output
- Output
  - pixel’s color
  - Optional: pixel’s depth value

Fragment Shader

- Fragment shader in Assignment 0

```
const char *fragmentShaderSource = "#version 330 core\n"
    "out vec4 FragColor;\n"
    "in vec3 ourColor;\n"
    "void main()\n"
    "{\n"
    "   FragColor = vec4(ourColor, 1.0f);\n"
    "}\n\0";
```

Fragment Shader

- What we have to do ourselves
  - texture fetch
  - texture application
  - per-pixel lighting (for Phong shading)
- What we cannot do ourselves
  - per-fragment operations
  - alpha blending

Outline

1. C++ Programming
2. Modern OpenGL
3. **Window API**

Window APIs

- GLX, CGL, AGL
  - Glue between OpenGL and windowing system
- GLUT, GLFW, Qt, …
  - Portable windowing (GUI) APIs
  - Handle windows, key/mouse events, …
  - Provide OpenGL widgets

Which Window API?

- GLUT
  - Quite old, but support Windows/Mac/Linux
  - https://user.xmission.com/~nate/glut.html
- freeglut
  - OK for Linux/Windows, not ideal for Mac
  - http://freeglut.sourceforge.net/
- GLFW
  - Successor of GLUT, very minimalistic
  - http://www.glfw.org
- Qt
  - Nice (but complex!) GUI API for Linux, Windows, Mac
  - http://www.qt.io

Used for assignments

Useful for large projects

What is GLFW?

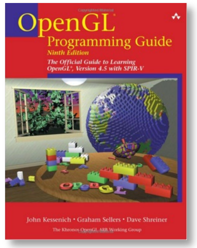
- Graphics Library Framework (GLFW) is a lightweight utility library for use with OpenGL
  - free, open source, multi-platform library
  - NOT a part of OpenGL
- GLFW provides the following functionalities.
  - create and manage windows and OpenGL contexts
  - read input
  - handle keyboard, mouse, and joystick input

How to Use GLFW

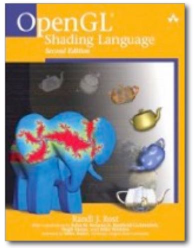
- To learn how to use GLFW, please follow the documentation: https://www.glfw.org/docs/latest/
- You don’t have to be familiar with every single function of GLFW.
- Assignment_0 code provides a good example about how to creating windows and to handle events with GLFW.

Literature

- Shreiner, Seller, Kessenich, Licea-Kane: *OpenGL Programming Guide*, 9th edition, 2016.



- Rost, et al.: *OpenGL Shading Language*, 3rd edition, 2009.



- Seller, Wright, Haemel: *OpenGL SuperBible*, 7th edition, 2015