CSE 328 (Spring 2022)
Tutorial Recitation

Xi Han
Department of Computer Science, Stony Brook University
xihan1@cs.stonybrook.edu

Notice

- A copy of this slide is available at the bottom of the TA Help Page.
- https://www3.cs.stonybrook.edu/~xihan1/courses/cse328/ta_help_page.html
- All pages marked "**AFT**" on top-right corner will be covered in detail during the lecture time. Those are for reference after class!

Contents

- Your TA & Assignment Info
- Introduction to OpenGL
- OpenGL Environment Setup: CMake + GLFW + GLAD on ubuntu
- A Quick Start on OpenGL Programming with C/C++
- Some Tips

Contents

- **Your TA & Assignment Info**
- Introduction to OpenGL
- OpenGL Environment Setup: CMake + GLFW + GLAD on ubuntu
- A Quick Start on OpenGL Programming with C/C++
- Some Tips

Your TA

- Your TA:
  - Xi Han
  - Please reach me by email xihan1@cs.stonybrook.edu
- Office Hours:
  - TBD;
  - If you are coming, please **email** your TA for **at least 12 hours in advance**!
- Shall there be any discrepancy between this PPT (including this page and all following pages) and TA Help Page, course website, or Blackboard, the **latter** shall prevail.

Assignments

- Please compress your assignment in a **.zip** file **`<your_sbuid>_pa1.zip`** and submit via Blackboard.
- Please include:
  - A README file, format requirements to be detailed in programming assignment manuals.
  - Your source code
- Please do NOT include:
  - Temporary files, IDE-specific configuration files, etc.
- All assignments should be done using C/C++ and OpenGL. The build system of your program should be **`CMake`**.

Assignments

- Structure of your submission (**after unzipping**):

```
<your_subid>_pa<x>
├── CMakeLists.txt
├── README.md
├── include
│   ├── bar.h
│   └── foo.h
└── src
    ├── bar.cpp
    ├── foo.cpp
    └── main.cpp
```

(*) After unzipping, all your files should lie in one root directory **<your_subid>_pa<x>**. They should **not** be bloated into the parent directory containing the **zip** package!

Contents

- Your TA & Assignment Info
- **Introduction to OpenGL**
- OpenGL Environment Setup: CMake + GLFW + GLAD on ubuntu
- A Quick Start on OpenGL Programming with C/C++
- Some Tips

Introduction to OpenGL

- **Open G**raphics **L**ibrary (**OpenGL**) is mainly considered a 3D rendering API that we can use to manipulate graphics and images.
- However, OpenGL **by itself** is **NOT** an API, but merely a **standard specification** developed and maintained by www.khronos.org.
  - In terms of OOP, OpenGL is merely an **interface** (Java) or an **abstract class** (C++)

Introduction to OpenGL

- An actual OpenGL **library** depends on hardware and is very platform-specific. The people **implementing** actual OpenGL libraries are usually graphics card manufacturers. Each graphics card that you buy supports specific versions of OpenGL.
  - **NVIDIA** distribute OpenGL libraries via its GPU drivers
  - Apple system: OpenGL library is maintained by **Apple** themselves
  - Virtual machines: Provided by VM vendors, e.g., **VMWare**

Introduction to OpenGL

- OpenGL standard is specified in C functions.
- OpenGL standard is completely **platform-independent**, which means, any platform-specific functionality (i.e., creating a window) is **NOT** offered by OpenGL.
  - These functionalities are offered by many **third-party libraries**!
  - If you see an OpenGL-related library and you don't know what it is, please refer to https://www.khronos.org/opengl/wiki/Related_toolkits_and_APIs

OpenGL only takes as input primitives to render, and outputs a raster image, i.e., OpenGL only involves the rendering pipeline.
Frontend GUI, how we determine what primitives to render, etc., are **not** OpenGL's functionality. These are your own program logic!

Introduction to OpenGL

- OpenGL works as a **state machine**. Some functions will modify a global context, and other functions work as guided by the context.
- E.g., `glClearColor` will set a global context called `GL_COLOR_CLEAR_VALUE`, and `glClear` will paint the whole viewport with the color specified by `GL_COLOR_CLEAR_VALUE`.
- Due to different OpenGL contexts, same call signatures of OpenGL functions may have totally different results!

Modern OpenGL

- Starting from OpenGL 3.3, OpenGL switches from ~~**immediate mode**~~ (legacy OpenGL) to **core-profile** mode (modern OpenGL).
- The ~~**immediate mode**~~ abstracted from and hard-coded many rendering operations (e.g., rendering algorithms, double buffers, coordinate transformation, etc.)
- The new **core-profile** mode is more complicated, yet **more efficient** and provides users with **higher dimension of freedom**.
- ~~**Immediate mode**~~ is officially announced deprecated.
- I.e., you should **NOT** refer to anything containing deprecated ~~**immediate-mode**~~ functions like ~~`glBegin`~~, ~~`glEnd`~~, ~~`glVertex3f`~~, ~~`glColor3f`~~, ~~`glDrawPixels`~~, etc.

Using OpenGL

Three things to do before you can ask OpenGL to render anything in your program:

- **Pick your language**.
  - This course asks you to pick **C/C++**.
  - There are other bindings including C#, Java, Python and Lua. Please remember all other language bindings are ultimately based on the C/C++ bindings: they are just wrappers.
- **Create a window with OpenGL Context**.
  - OpenGL draw on a window as guided by **OpenGL Context**.
  - As mentioned before, creating a window is very platform-specific and is done by **window toolkit**.
  - This course asks you to pick **GLFW**.
- **Load API functions**.
  - Is also very platform-specific and complicated. For details, please refer to https://stackoverflow.com/questions/34662134/why-is-opengl-designed-in-a-way-that-the-actual-functions-have-to-be-loaded-manu.
  - Done by **OpenGL Loading Library**.
  - This course asks you to pick **GLAD**.

Window Toolkits

These toolkits are designed specifically around creating and managing OpenGL windows. They also manage input, but little beyond that.	Several "multimedia libraries" can create OpenGL windows, in addition to input, sound and other tasks useful for game-like applications.	Many widget toolkits have the ability to create OpenGL windows, but their primary focus is on being widget toolkits.
freeglut A crossplatform windowing and keyboard/mouse handler. Its API is a superset of the GLUT API, and it is more stable and up to date than GLUT. It supports creating a core OpenGL context.	**Allegro version 5** A cross-platform multimedia library with a C API focused on game development. Supports core OpenGL context creation.	**FLTK** A fast and light-weight cross-platform C++-based widget library that is tightly integrated with OpenGL. It includes a graphical UI editor (fluid) that makes development easy and efficient. It is used in Bjarne Stroustrup's book "Programming - Principles and Practice Using C++".
GLFW A crossplatform windowing and keyboard/mouse /joystick handler. Is more aimed for creating games. Supports Windows, Mac OS X and *nix systems. Supports creating a core OpenGL context.	**SDL** A cross-platform multimedia library with a C API. Supports creating a core OpenGL context.	**Qt** A C++ toolkit which abstracts the Linux, MacOS X and Windows away. It provides a number of OpenGL helper objects, which even abstract away the difference between desktop GL and OpenGL ES.
GLUT Very old, **do not use**.	**SFML** A cross-platform multimedia library with a C++ API. Supports creating a core OpenGL context.	**wxWidgets** A C++ cross-platform widget toolkit.
	Ecere SDK Provides rendering APIs for OpenGL, OpenGL ES and DirectX, 3D math and concepts like cameras, windowing and GUI widget library, wrapped in a compiler and IDE for its own streamlined "eC" language that cross-compiles to desktop, mobile and web platforms.	**Game GUI** An OpenGL based C++ widget toolkit with skin support and integrated window manager for games.

https://www.khronos.org/opengl/wiki/Related_toolkits_and_APIs#Context.2FWindow_Toolkits

OpenGL Loading Libraries

1 GLEW (OpenGL Extension Wrangler)
    1.1 Initialization of GLEW 1.13.0 and earlier
2 GL3W
3 glLoadGen (OpenGL Loader Generator)
4 Galogen
5 glad (Multi-Language GL/GLES/EGL/GLX/WGL Loader-Generator)
6 Glatter
7 glsdk (Unofficial OpenGL SDK)
8 glbinding (C++)
9 libepoxy
10 GLee

https://www.khronos.org/opengl/wiki/OpenGL_Loading_Library

Useful OpenGL Websites

- These sites will solve most of your questions regarding OpenGL:
  - Official OpenGL wiki that explains how OpenGL works: https://www.khronos.org/opengl/wiki/Getting_Started#Writing_an_OpenGL_Application
  - Official OpenGL Reference Page: https://www.khronos.org/registry/OpenGL-Refpages/gl4/
  - A great OpenGL tutorial website: https://learnopengl.com

Contents

OpenGL Environment Setup

AFT

- Setup on 64bit ubuntu 20.04 VM (VMWare)
  - Build system: CMake
  - OpenGL context library: GLFW
  - OpenGL loading library: GLAD
  - Utilities: GLM (OpenGL matrices), OpenCV (images)
  - IDE: Clion
- A read-to-use (C/C++/CMake/OpenGL-related configuration done) VMWare VM hard disk will be available on Blackboard under "Course Documents"!

OpenGL Environment Setup

AFT

- Your TA personally recommends that you setup OpenGL environment on ubuntu VM
  - Easier to setup on ubuntu than on Windows/macOS
  - Does not pollute your own local OS
  - In terms of performance, a VM is enough for programming assignments (while, may not be enough for projects, if you select to do projects.)
  - The testing platform is also ubuntu VM
- VMWare:
  - https://www.vmware.com/products.html
  - Windows users: download & install **VMWare Workstation**
  - Mac users: download & install **VMWare Fusion**
- 64bit ubuntu 20.04.1 image:
  - https://ubuntu.com/download/desktop/thank-you?version=20.04.1&architecture=amd64

OpenGL Environment Setup

AFT

- Enable OpenGL 4.1 on VMWare guest OS:
  - **`sudo add-apt-repository ppa:kisak/kisak-mesa`**
  - **`sudo apt update`**
  - **`sudo apt-get dist-upgrade`**
  - **`sudo reboot`**
- CLion (An excellent C/C++ IDE that is free to students!):
  - https://www.jetbrains.com/clion/download/#section=linux
- All other libraries can be installed & managed by **`apt`**:
  - **`sudo apt install cmake libopencv-dev libglm-dev libglew-dev libglfw3-dev mesa-utils libx11-dev libxi-dev libxrandr-dev`**

OpenGL Environment Setup

AFT

- How to check your OpenGL vendor and versions?
  - Command **`glxinfo`** (provided by package **`mesa-utils`**)
  - **`sudo glxinfo | grep "OpenGL"`**

```
ax@ubuntu:~$ sudo glxinfo | grep "OpenGL"
[sudo] password for ax:
OpenGL vendor string: VMware, Inc.
OpenGL renderer string: SVGA3D; build: RELEASE;  LLVM;
OpenGL core profile version string: 3.3 (Core Profile) Mesa 20.0.8
OpenGL core profile shading language version string: 3.30
OpenGL core profile context flags: (none)
OpenGL core profile profile mask: core profile
OpenGL core profile extensions:
OpenGL version string: 3.3 (Compatibility Profile) Mesa 20.0.8
OpenGL shading language version string: 3.30
OpenGL context flags: (none)
OpenGL profile mask: compatibility profile
OpenGL extensions:
OpenGL ES profile version string: OpenGL ES 2.0 Mesa 20.0.8
OpenGL ES profile shading language version string: OpenGL ES GLSL ES 1.0.16
OpenGL ES profile extensions:
```

VMWare VM

```
xihan1@alien130:~$ sudo glxinfo | grep "OpenGL"
[sudo] password for xihan1:
OpenGL vendor string: NVIDIA Corporation
OpenGL renderer string: GeForce RTX 2080 Ti/PCIe/SSE2
OpenGL core profile version string: 4.6.0 NVIDIA 450.80.02
OpenGL core profile shading language version string: 4.60 NVIDIA
OpenGL core profile context flags: (none)
OpenGL core profile profile mask: core profile
OpenGL core profile extensions:
OpenGL version string: 4.6.0 NVIDIA 450.80.02
OpenGL shading language version string: 4.60 NVIDIA
OpenGL context flags: (none)
OpenGL profile mask: (none)
OpenGL extensions:
OpenGL ES profile version string: OpenGL ES 3.2 NVIDIA 450.80.02
OpenGL ES profile shading language version string: OpenGL ES GLSL ES 3.20
OpenGL ES profile extensions:
```

PC with NVIDIA Geforce RTX 2080 Ti

- GLAD:
  - Obtain from webservice https://glad.dav1d.de/
  - Select options according to **`glxinfo`**
  - Check option "Generate a loader"
  - Download 2 headers and a C source file, add to project

Contents

- Your TA & Assignment Info
- Introduction to OpenGL
- OpenGL Environment Setup: CMake + GLFW + GLAD on ubuntu
- **A Quick Start on OpenGL Programming with C/C++**
- Some Tips

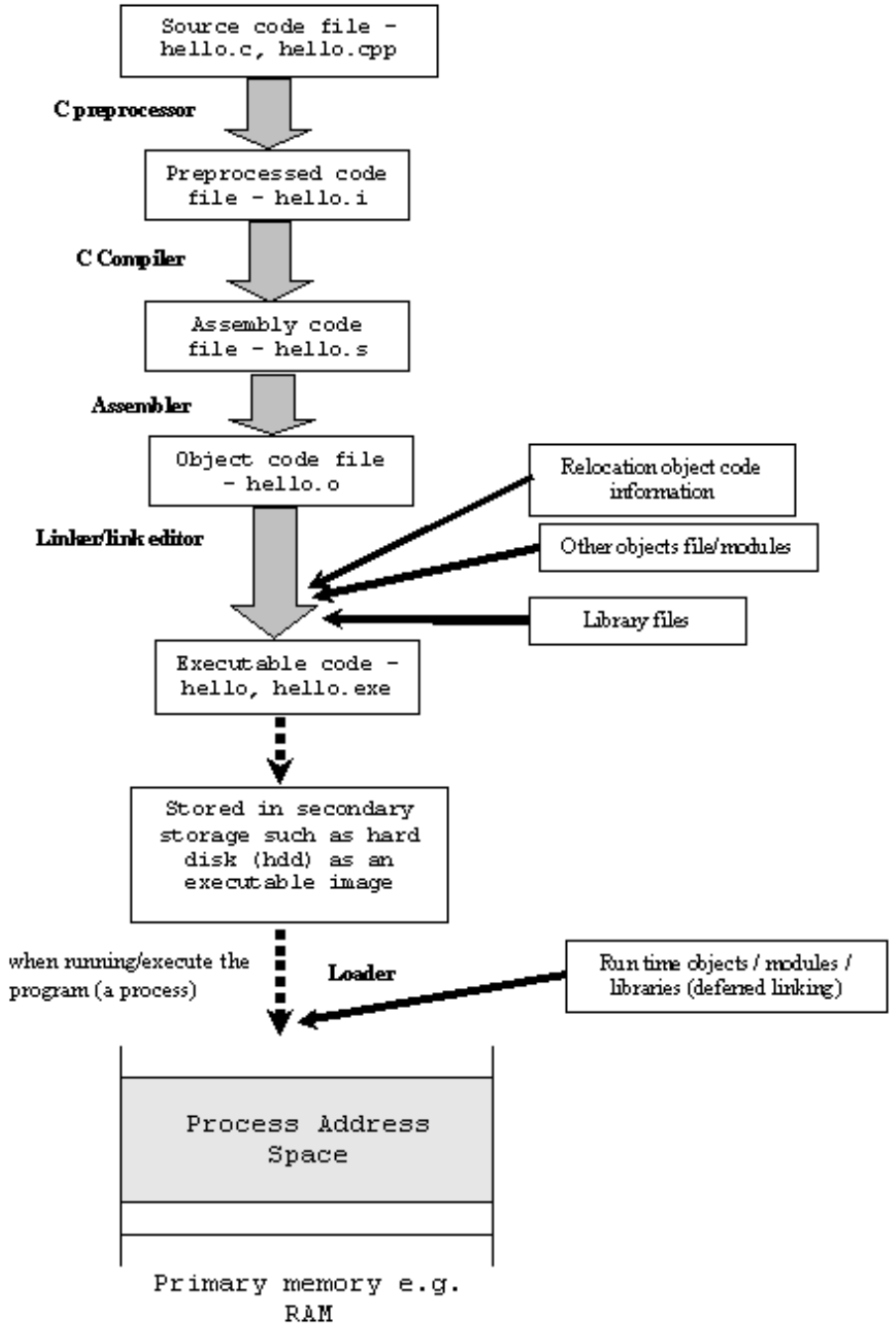
A Quick Start on OpenGL Programming with C/C++

- One prerequisite of CSE 328 is C/C++ programming skills.
- The following tutorial is **not** guaranteed to be 100% correct.
- This is just for a quick (but not so precise) start.
- **Many details/exceptions/edge cases are omitted**.
- You should always refer to official manuals and documents.

Translation

- A **C/C++ executable** is **translated** (i.e., created) from
  - **Source Code** (text file, typically headers (**.h**) and sources (**.cpp**));
  - **Objects** (binary file):
    - Dynamic-link libraries: Shared object (**.so**) / Application Extension (**.dll**);
    - Static libraries: Archive (**.a**) / (**.lib**).
- **Translation**
  1. Source code need to be **preprocess**ed, **compile**d and then **assemble**d into binary objects;
  2. Binary objects are **link**ed together to form a final executable/
- **Translation** is done by **compilers**
  - We are using **G**NU **C C**ompiler (**GCC**)
  - **GCC** has complicated command line argument rules (Details omitted!)
- Instructing **GCC** to build your program is a complex procedure
  - **make**: Build a file as guided by a script called **`Makefile`**, that contains all rules to generate the file, platform-dependent.
  - **CMake**: A C/C++ abstract from **make**, easier to use, cross-platform. Also guided by a script called **`CMakeLists.txt`**.

Translation Pipeline



AFK

CMake

- **CMake** also works as guided by a script called `CMakeLists.txt`
- A `CMakeLists.txt` will be provided for all programming assignment templates, so you don't have to master all **CMake** details when doing Programming Assignment 1.
- But you are highly likely to find yourself want to learn more about **CMake** later in this semester.

CMake Grammar

- Variables
  - Usually in upper case (just a convention, don't have to)
  - Can be set using **`set(VAR_NAME value)`**
    - **`value`** can be another variable or a literal
  - When being referred, variables need to be quoted by **`${VAR_NAME}`**
    - The only exception: inside **`set()`** command when being set
- Literals
  - Most-frequently used: string (quote with "" when containing space)
  - Lists: multiple strings separated by space
- Targets
  - Likely to be an executable, a library or some other stuff

CMake Commands

- **set**
- **project**
  - `project(pa1)` will call `set(PROJECT_NAME pa1)`
- **add_executable**
  - E.g., `add_executable(pa1 main.cpp foo.cpp bar.cpp)`
  - This command generates an executable CMake target object that may be the target of the following targeted commands
- **target_compile_definitions**
  - E.g., `target_compile_definitions(pa1 PUBLIC -DDEBUG)`
- **target_compile_options**
  - E.g., `target_compile_options(pa1 PUBLIC –Wall -Wextra)`

CMake Commands

- **target_include_directories**
  - Add extra directories to search for headers when you call **`#include`** command in your source code
  - Do not need to care about **`PUBLIC/PRIVATE`** option for now. Just provide it.
  - E.g. **`target_include_directories(pa1 PUBLIC ${ALL_INCLUDE_DIRS})`**
- **target_link_libraries**
  - Link precompiles, binary shared object or archive to your executable
  - E.g. **`target_link_libraries(pa1 ${ALL_LIBRARIES})`**

CMake Commands

AFK

- **find_package**
  - Has a complex rule
    - At this stage, you may just refer to https://cmake.org/cmake/help/latest/manual/cmake-modules.7.html#find-modules for find modules provided by CMake installation
    - Some packages (e.g., glfw3) will also offer ready-to-use find modules
  - This command will set some variables or introduce other targets for further use, E.g.
    - **`find_package(glfw3)`** will introduce a target called **`glfw`**
    - **`find_package(OpenCV)`** will set 2 variables, namely **`${OPENCV_INCLUDE_DIR}`** and **`${OpenCV_LIBS}`**
    - You may just provide these inside **`target_include_directories`** command and **`target_link_libraries`** command

C/C++ IDE: CLion

- Tutorials on:
  - How to create a C++ project
  - How to configure CMakeLists.txt & frequently-used CMake commands
  - How to switch between debug/release build
  - How to debug
  - How to set working directory
- CLion Tutorial
  - https://www.jetbrains.com/help/clion/clion-quick-start-guide.html
- CMake Tutorials
  - https://www.jetbrains.com/help/clion/quick-cmake-tutorial.html
  - https://cmake.org/cmake/help/latest/index.html (Official Document)

C++ for Java/Python Users

- In recent years:
- C++ syntax is similar to Java!
  - C++ and Java are copying their syntax back and forth in the past 30 years
  - Java syntax was initially designed to mimic C++
  - C++ copied from Java: Range-based loop ("for each"), `volatile`, `final`, `override`, etc.
- C++ syntax is becoming more and more Pythonic!
  - Adopting Pythonic easy-to-use grammars

Java-like

- Java `ArrayList` alternative: `std::vector`
- Similar import syntax, program entry point, range-based loop, …

```cpp
#include <iostream>
#include <vector>

int main(int argc, char * argv[])
{
    std::vector<int> vec;
    vec.emplace_back(0);
    vec.emplace_back(1);
    vec.emplace_back(2);

    for (int i : vec)
    {
        std::cout << i << "\n";
    }

    return 0;
}
```

C++

```java
import java.io.*;
import java.util.ArrayList;

public class Main
{
    public static void main(String[] args)
    {
        ArrayList<Integer> lst = new ArrayList<>();
        lst.add(0);
        lst.add(1);
        lst.add(2);

        for (Integer i : lst)
        {
            System.out.print(i + "\n");
        }
    }
}
```

Java

AFT

Java-like

C++

```cpp
std::vector<int> vec {0, 1, 2};

for (auto it : iterator<...> = vec.begin(), e : iterator<...> = vec.end(); it != e;)
{
    std::cout << *(it++) << '\n';
}
```

- P.S. C++/Java range-based loops are implemented on **iterator**s.
- C++ **uniform initialization** (**Braced initializer list** "`{1, 2, …}`")

Java

```java
for (Iterator<Integer> it = lst.iterator(); it.hasNext();)
{
    System.out.println(it.next());
}
```

Pythonic

AFT

C++

```cpp
std::tuple<int, int, int> tup {1, 2, 3};
auto [a :int , b :int , c :int ] = tup;
std::cout << a << ' ' << b << ' ' << c << '\n';

std::map<int, std::string> dic {{ x: 1,  y: "one"}, { x: 2,  y: "two"}};

for (auto [k :const int , v :string ] : dic)
{
    std::cout << k << ' ' << v << '\n';
}
```

- Python **`Tuple`** alternative: **`std::tuple`**
- Pythonic tuple unpacking
- C++ **`auto`** type specifier: Let compiler deduce variable type

Python

```python
tup: typing.Tuple[int, int, int] = (1, 2, 3)
a, b, c = tup
print(a, b, c)

dic: typing.Dict[int, str] = {1: "one", 2: "two"}

for k, v in dic.items():
    print(k, v)
```

AFT

Pythonic

- Python **`with`** statement alternative: **`if`**
- **`fin`** will be closed automatically after **`if`** ends (scope of **`fin`** ended, destructed automatically, destructor calls close method)

```cpp
if (std::ifstream fin {s: "1.txt"})
{
    std::string line;

    while (std::getline(&: fin, &: line))
    {
        std::cout << line << '\n';
    }
}
```

C++ line-by-line file input

```python
with open('1.txt') as fin:
    for line in fin:
        print(line)
```

Python line-by-line file input

"Isn't it the case that C/C++ have no garbage collection mechanics?"
Yes, but the term "garbage collection" itself is for heap memory.
For this **`fin`**, it is on stack memory instead.

C++ for Java/Python Users

- What we want to stress: C/C++ is **not** terrifying!
- Modern C++ has absorbed many good aspects from other popular languages, including Java, Python, etc.
- Basic C++ syntax is almost identical to Java, so you could just code C++ in a Java-like manner…
  - Except several special cases (to be detailed in the following pages)

C++ V.S. Java

AFT

- Basic data type: Almost same as Java
  - `char`, `short`, `int`, `long`, `long long`, `float`, `double`
  - `unsigned` prefix for `char` and integral types
  - Java `boolean` → C++ `bool`
  - Java object `final` → C++ object `const`
- Branch & loop: Same as Java
  - `if`, `switch`, `?:`, `for`, `while`, `do while`
- Function: Almost same as Java
  - Declaration, definition, call, etc.
- Program Entry Point: `main` function
  - Java: `MainClass.main(String[] args)`
  - C++: `int main(int argc, char * argv[])`

C++ V.S. Java

AFT

- Memory
  - Objects can be allocated on heap or stack
  - **No** `new` needed when creating objects on stack
- Header & source
  - **Declaration**: In headers (`.h`, `.hpp`)
  - **Definition**: In source files (`.cpp`)
  - All **declaration** statements are also **definition**s **except** several special cases: `class` declaration, function declaration, `extern` object, etc.

Pointer And Reference

Knowing memory layout is critical to OpenGL!

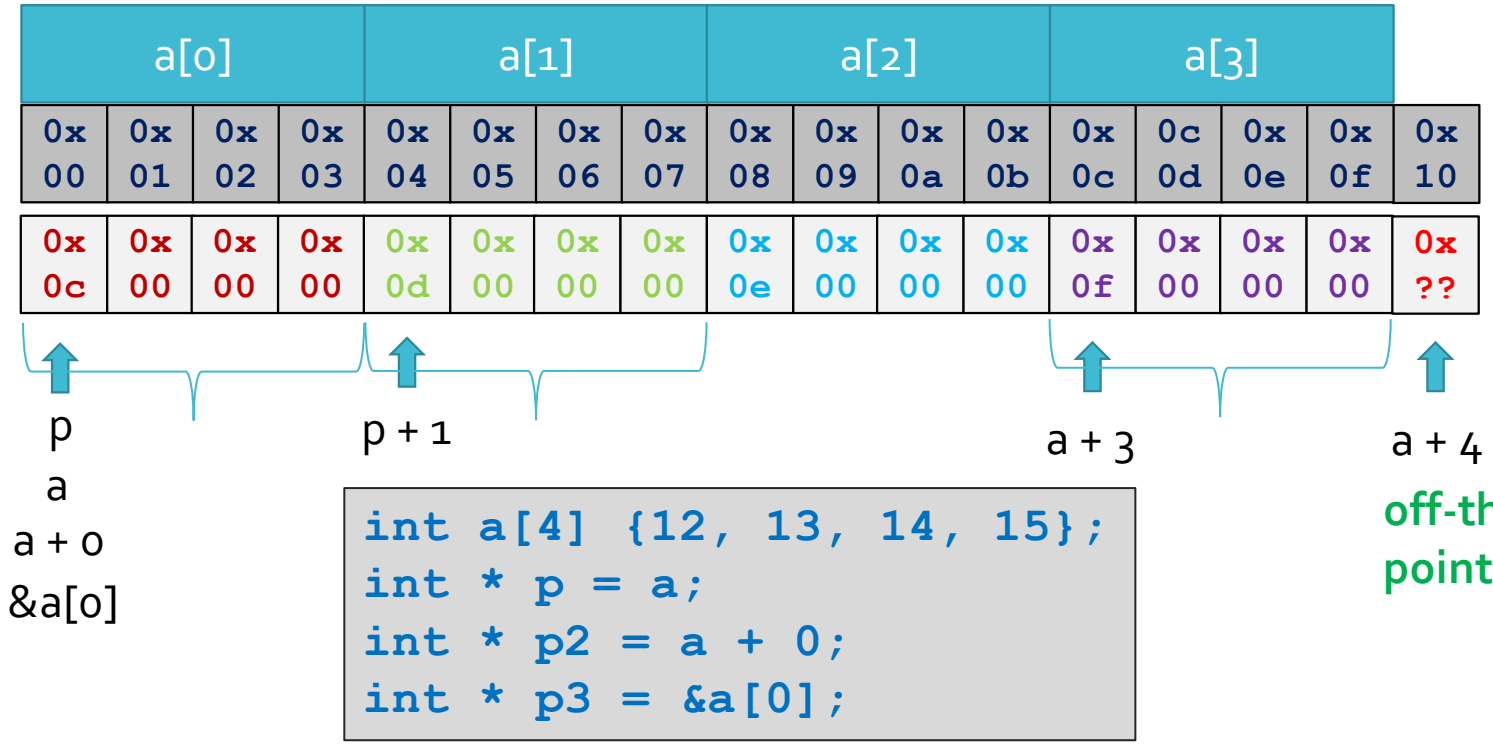
- Recall the concept of **reference** in Java
  - **When changing a reference, the original value also changes**
  - Built-in types **pass-by-value**, classes **pass-by-reference**
- In C++, all types (both built-in and classes) **pass-by-value**, **except explicitly asked by the programmer** to **pass-by-reference**
- C/C++ **pointer** is a type that stores the **address** (i.e., one specific block of memory) of an object of the specified type
  - To declare pointer to type **`T`**, use statement **`T * p`**;
  - We can use the **address-of operator** "**`&`**" to **get the address** of an object
  - We can use **de-reference operator** "**`*`**" to **read the value** stored in the address
- C++ **reference is just a shortcut of de-referenced pointer**

```
int a = 1;

// pointer to int
int * p = &a;
*p = 2;  // *p == a == 2

// reference to int
int & r = a;
r = 3;   // r == *p == a == 3
```

```
// pass-by-value, WON'T work
void increment(int a)   { ++a;  }

// pass-by-value,
// but modifications on de-referenced pointers
// also changes original value, so it works
void increment(int * a) { ++*a; }

// pass-by-reference, works
void increment(int & a) { ++a;  }
```

Pointer Logic

`sizeof(int) == 4`




```
int a[4] {12, 13, 14, 15};
int * p = a;
int * p2 = a + 0;
int * p3 = &a[0];
```

(1) Pointer stores address as an integral value. If an object occupies multiple Bytes, its address its first Byte.

(2) Array head `T a[]` could be implicitly casted into `T * p = &a[0]`

(3) De-referencing a pointer `T * p` is to decode `sizeof(T)` Bytes of binary data (the Byte it points to plus `sizeof(T) -1` succeeding Bytes) into a `T`-type value.

(4) Adding `T * p` by `1` is actually adding `p` by `sizeof(T)`

(5) When changing a reference, the original value also changes, because C++ reference is just a shortcut of de-referenced pointer.

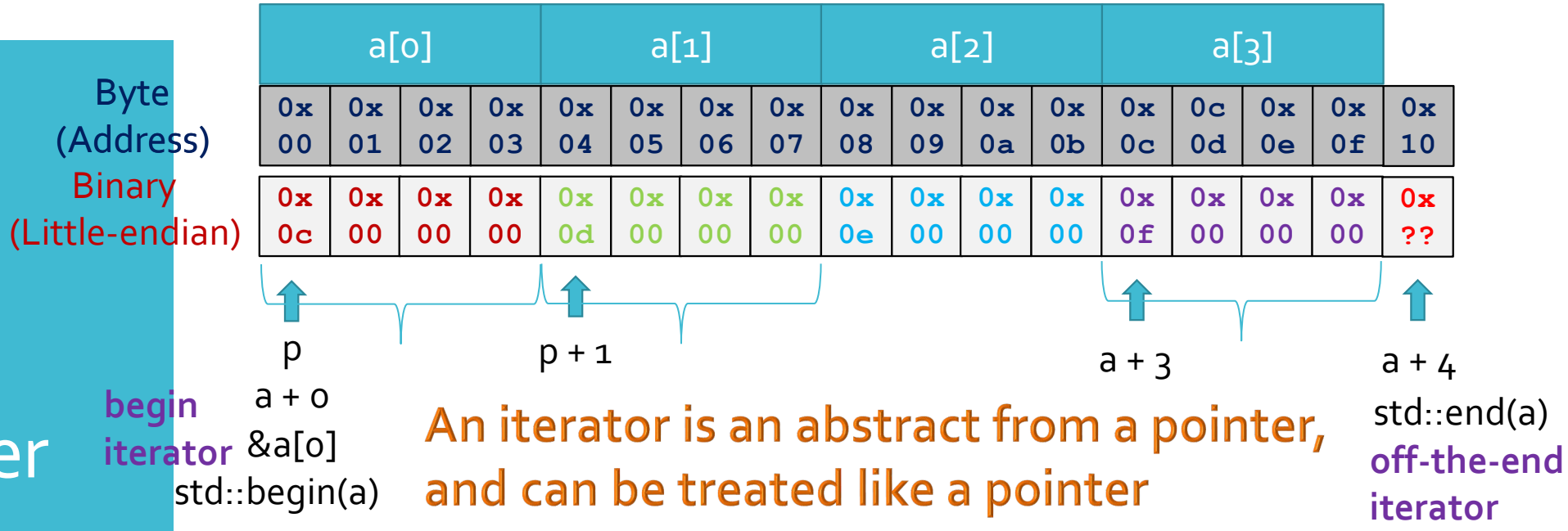
From Pointer to Reference

Byte (Address)	a[0]				a[1]				a[2]				a[3]			
	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
Binary (Little-endian)	0x0c	0x00	0x00	0x00	0x0d	0x00	0x00	0x00	0x0e	0x00	0x00	0x00	0x0f	0x00	0x00	0x00

```
int a[4] {[0]: 12, [1]: 13, [2]: 14, [3]: 15};

(1) for (std::size_t i = 0; i != 4; ++i)
        ++a[i];

(2) for (int n : a)  // won't work, but why the previous works?
        ++n;

(3) for (std::size_t i = 0; i != 4; ++i)  // x[y] <==> *(x + y)
        ++(*(a + i));

(4) for (int & n : a)
        ++n;
```

When changing a reference, the original value also changes, because C++ reference is just a shortcut of de-referenced pointer.

From Pointer to Iterator




```
int a[4] {[0]:12, [1]:13, [2]:14, [3]:15};

// pointer
for (int * p = a, * e = a + 4; p != e; ++p)   (1)
    ++(*p);

// An abstract from pointer: Iterator
// Can increase/decrease, can de-reference
for (auto it:int* = std::begin(&:a), e:int* = std::end(&:a); it != e; ++it)   (2)
    ++(*it);
```

AFT

Range-based Loop

- Range-based loop is just a shortcut of iterator-based loop!

```
for (T e : sequence)
{
    ++e;
}

for (auto it : iterator<...> = std::begin( &: sequence), end : iterator<...> = std::end( &: sequence); it != end; ++it)
{
    T e = *it;
    ++e;
}
```

```
for (T & e : sequence)
{
    ++e;
}

for (auto it : iterator<...> = std::begin( &: sequence), end : iterator<...> = std::end( &: sequence); it != end; ++it)
{
    T & e = *it;
    ++e;
}
```

Class

- Class
  - C++ **`class`**: Just like classes in Java (members **`private`** by default)
  - **`this`**: Just like Java, except that **`this`** is a **pointer**, **not** a **reference**
  - **Constructor**: Also appears in Java/Python
  - **Destructor**: What we should do when an object is destructed (when it goes out of scope or deleted manually)

C++

```cpp
class Entry
{
public:
    Entry(int key, std::string value)
    {
        this->mKey = key;
        this->mValue = value;
    }

    ~Entry() {}

private:
    int mKey;
    std::string mValue;
};
```

Java

```java
class Entry
{
    public Entry(Integer key, String value)
    {
        this.mKey = key;
        this.mValue = value;
    }

    private Integer mKey;
    private String mValue;
}
```

Class

- C++ **`struct`**: Just like classes, except that members **`public`** by default
- Friend class and friend function: Declare a class or function friend inside another class to access that class's private members.

Polymorphism

AFT

- Polymorphism
  - Java: When virtual function called on Base **reference** to Derived
  - C++: When virtual function called on Base **pointer or reference** to Derived
- Java interface alternative
  - **Abstract base class** (A class with **pure virtual function**)
  - Syntax: `virtual void fun() = 0;`
  - A pure virtual function needs no definition (i.e., function body)
  - An abstract base class can **not** be instantiated (same as a Java interface!)
- Java-like extension + implementation alternative
  - Multiple inheritance
  - Inherit a normal Base class as well as an abstract base class
- One key difference:
  - Java: Non-static member functions are virtual by default
  - C++: **Non-virtual** by default. Use `virtual` keyword

Polymorphism

```cpp
class AbstractBase
{
public:
    virtual ~AbstractBase() {}
    virtual void api() = 0;
};

class Base
{
public:
    virtual ~Base()    {}
    virtual void foo() {}
    virtual void bar() {}
};

class Derived: public AbstractBase, public Base
{
public:
    ~Derived() override {}

    void api() override
    { std::cout << "Derived::api()\n"; }

    void foo() override
    { std::cout << "Derived::foo()\n"; }

    void bar() final
    { std::cout << "Derived::bar()\n"; }
};
```

C++

AFT

```java
interface Interface
{
    void api();
}

class Base
{
    public void foo() {}
    public void bar() {}
}

class Derived extends Base implements Interface
{
    public void api()
    { System.out.println("Derived.api()"); }

    @Override
    public void foo()
    { System.out.println("Derived.foo()"); }

    @Override
    public final void bar()
    { System.out.println("Derived.bar()"); }
}
```

Java

Polymorphism

AFT

```
Base base();
Derived derived();
Base wrong = derived;  // clipped into Base
Base * p = &derived;
Base & r = derived;

// calls Derived::foo
p->foo();
r.foo();

// calls Base::foo
wrong.foo();
```

Namespace

- Namespace
  - Used to limit the scope of entities to prevent naming conflicts
  - Just like packaging in Java
  - All stuff from **C++ standard library** (**S**tandard **T**emplate **L**ibrary, **STL**) are under namespace **`std`**
  - To access an entity inside a namespace, use **scope operator** "**`::`**"
    - E.g., **`std::cout`**, **`std::vector`**

C++

```cpp
namespace xihan1
{

class Entry { /* ... */ };

}  // namespace xihan1
```

```cpp
xihan1::Entry e;
```

```java
package xihan1;

public class Entry { /* ... */ }
```

```java
xihan1.Entry e = new Entry();
```

Java

- One special case: C++ **class attributes** are also accessed via **scope operator** "**`::`**" (**not member operator** "**`.`**")

Namespace

- Namespaces are used to avoid naming conflicts.
- Namespaces also exist in Python!
- Another example: Recall how you use NumPy in Python.

```cpp
// All vector stuff will be available
// under namespace std with this statement
#include <vector>

std::vector<int> a {1, 2, 3};
```

```cpp
// All vector stuff will be available
// under namespace std with this statement
#include <vector>

// NOT RECOMMENDED! Just for comparasion.
// Further alias "std" with "ABC".
namespace ABC = std;

ABC::vector<int> a {1, 2, 3};
```

C++

```python
# All Numpy stuff will be available
# under namespace numpy with this statement
import numpy

a: numpy.ndarray = numpy.array([1, 2, 3])
```

```python
# "import numpy":
# All Numpy stuff will be available
# under namespace numpy.
# "as np":
# You actually further rename "numpy" with "np".
import numpy as np

a: np.ndarray = np.array([1, 2, 3])
```

Python

Scope Operator In Class

```
struct S
{
    static void foo();

    static int a;

    S();

    void bar();

    int b {2};
};

void S::foo() {}

int S::a = 1;

S::S() = default;

void S::bar() {}
```

```
int main(int argc, char * argv[])
{
    S::a = 2;
    S::foo();

    S s;
    s.b = 3;
    s.bar();
```

Scope Operator In Namespace

```
namespace xihan1
{

int a = 1;

void foo() {}

}  // namespace xihan1

int a = 2;

void foo() {}
```

```
int main(int argc, char * argv[])
{
    xihan1::a = 4;
    xihan1::foo();

    a = 3;
    foo();
```

Template

C++

```cpp
std::vector<float> vec {0, 1, 2};
std::cout << vec[0] << "\n";
```

- Templates
  - Used for generic type coding (Recall how you use Java **`ArrayList`**)
  - Template functions and template classes
  - Template parameters are specified in "**`<>`**"
  - E.g., **`std::vector<int>`** is a vector that stores floating point numbers

Java

```java
ArrayList<Float> lst = new ArrayList<>();
lst.add(0.f);
lst.add(1.f);
lst.add(2.f);
System.out.print(lst.get(0) + "\n");
```

STL Container

AFT

- STL Container
  - **`std::vector`** – Most frequently used container type. A size-adaptable array, just like **`ArrayList`** in Java or **`List`** in Python
  - Unlike Java, all elements are fixed to one single type
  - Refpage: https://en.cppreference.com/w/cpp/container/vector
- Commonly-used functions to access data:
  - **`emplace_back()`**: Append data to end of vector
  - **`pop_back()`**: Pop the last element out of the vector
  - **`front()`**, **`back()`**: Return reference to the first/last element
  - **`begin()`**, **`end()`**: Return begin and off-the-end iterators
  - **`vec[i]`**: Array-like indexing (e.g., `vec[2] = 3;`)
  - **`size()`**: Return number of elements in the vector
  - **`clear()`**: Remove all elements in the vector
  - **`erase(it)`**: Remove the element pointed to by iterator **`it`**
  - **`data()`**: Return pointer to the underlying C array of the vector (used for interaction with C APIs such as **`glBufferData()`**)

STL Container

AFT

```
std::vector<glm::vec2> vec;

glm::vec2 v1 { x: 12,  y: 12};
vec.emplace_back(v1);
vec.emplace_back(glm::vec2 { x: 11,  y: 11});
vec.emplace_back( x: 10,  y: 10);

// 12 12, 11 11, 10 10,
for (glm::vec2 * p = vec.data(), * e = p + vec.size(); p != e; ++p)
    std::cout << *p << ", ";
std::cout << '\n';

// 12 12, 11 11, 10 10,
for (auto it : iterator<...> = vec.begin(), e : iterator<...> = vec.end(); it != e; ++it)
    std::cout << *it << ", ";
std::cout << '\n';

// 12 12, 11 11, 10 10,
for (const auto & p : const vec<...> & : vec)
    std::cout << p << ", ";
std::cout << '\n';
```

STL Container

AFT

```
// vec: 12 12, 11 11, 10 10

std::cout << vec.front() << ' '       // 12 12
          << vec[0] << ' '            // 12 12
          << *(vec.begin()) << ' '    // 12 12
          << *(vec.data()) << '\n';   // 12 12

std::cout << vec.back() << ' '            // 10 10
          << vec[2] << ' '                // 10 10
          << *(vec.end() - 1) << ' '      // 10 10
          << *(vec.begin() + 2) << ' '    // 10 10
          << *(vec.data() + 2) << '\n';   // 10 10

vec.erase( position: vec.begin() + 1);  // 12 12, 10 10
vec.pop_back();                         // 12 12
vec.clear();                            // empty
std::cout << vec.empty() << '\n';       // 1
std::cout << vec.size() << '\n';        // 0
```

std::vector

AFT

- Concatenating twos **`std::vector`**s

```cpp
std::vector<int> a {0, 1, 2};
std::vector<int> b {3, 4, 5};

// Concatenate a and b, a will be {0, 1, 2, 3, 4, 5}
a.insert(a.end(), b.cbegin(), b.cend());

// another way to concatenate a and b
std::copy(b.cbegin(), b.cend(), std::back_inserter(a));
```

- Sorting **`std::vector`**

```cpp
std::vector<int> a {1, 9, 6, 8, 7, 3, 4, 5, 2};
std::sort(a.begin(), a.end(), [](const auto & x, const auto & y)
{
    return x > y;
});
// a is now {9, 8, 7, 6, 5, 4, 3, 2, 1}
std::sort(a.begin(), a.end());
// a is now {1, 2, 3, 4, 5, 6, 7, 8, 9}
std::sort(a.begin(), a.end(), std::greater<>());
// a is now {9, 8, 7, 6, 5, 4, 3, 2, 1}
```

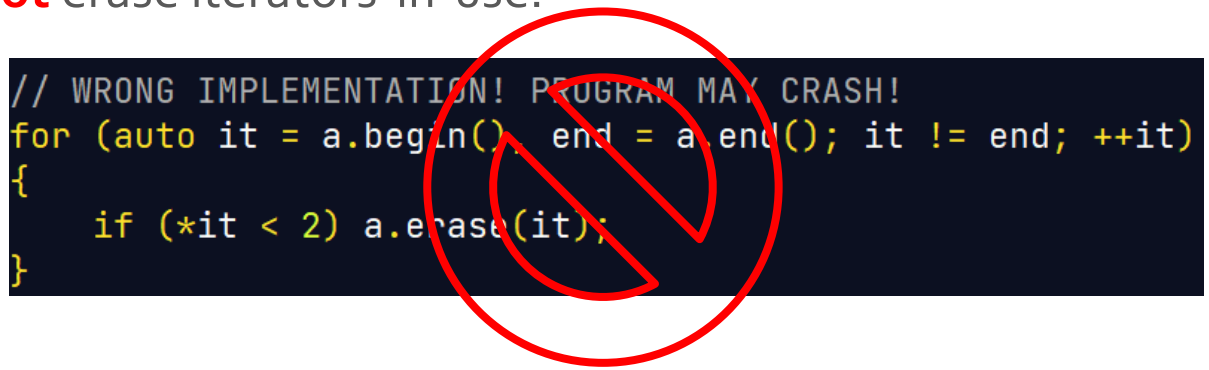
std::vector

AFT

- Removing multiple elements from **`std::vector`**

```
std::vector<int> a {1, 1, 3, 1, 3, 3, 1, 3};

// std::remove_if is actually a "sorting" (re-ordering) function
auto it = std::remove_if(a.begin(). a.end(), [](const auto & e)
{
    return e < 2;
});

//                      it          a.end()
//                      |           |
// a is now {3, 3, 3, 3, 1, 1, 1, 1}

a.erase(it, a.end());

// a is now {3, 3, 3, 3}
```

- Do **not** erase iterators-in-use!



C++ & OpenGL

- Again, the TA strongly recommend that you refer to this online tutorial: https://learnopengl.com/
  - Its chapters "Introduction" and "Getting Started" covers almost everything about OpenGL you need throughout this semester!
- OpenGL
  - How to setup OpenGL context in your program (GLFW & GLAD)
  - How to handle input events (GLFW)
  - **`shader`** → **`VAO`** → **`VBO`** pipeline

OpenGL Context Setup

- The process is kind of fixed
  - Just call a bunch of GLFW/GLAD functions before you call OpenGL core functions
- You can **just rely on the same bunch of code for context setup** for all programming assignments this semester.

OpenGL Context Setup

```
// 1. Initialize OpenGL content by GLFW

glfwInit();
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 1);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);

GLFWwindow * window = glfwCreateWindow(context.WINDOW_WIDTH,
                                       context.WINDOW_HEIGHT,
                                       "PA1 - Line Segment Mode",
                                       nullptr,
                                       nullptr);

if (!window)
{
    glfwTerminate();
    throw std::runtime_error("failed to create GLFW window");
}
```

OpenGL Context Setup

```
// Register all GUI callbacks here
glfwMakeContextCurrent(window);
glfwSetCursorPosCallback(window, cursorPosCallback);
glfwSetKeyCallback(window, keyCallback);
glfwSetMouseButtonCallback(window, mouseButtonCallback);

// 2. Load OpenGL functions pointers by GLAD

if (!gladLoadGLLoader(reinterpret_cast<GLADloadproc>(glfwGetProcAddress)))
{
    glfwTerminate();
    throw std::runtime_error("failed to initialize GLAD");
}
```

Input Handling with GLFW

- GLFW is our window toolkit, it is an OpenGL GUI library that is capable of a variety types of input
- For one specific type of input, we need to **register callback functions** that deals with the input event
- E.g., keyboard callback

If you wish to be notified when a physical key is pressed or released or when it repeats, set a key callback.

```
glfwSetKeyCallback(window, key_callback);
```

The callback function receives the **keyboard key**, platform-specific scancode, key action and **modifier bits**.

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    if (key == GLFW_KEY_E && action == GLFW_PRESS)
        activate_airship();
}
```

Input Handling with GLFW

- GLFW is our window toolkit, and it is capable of a variety types of input
- For one specific type of input, we need to **register callback functions** that deals with the input event
- GLFW calls **registered callback functions** when the function **`glfwPollEvents`** is called
  - We usually call it manually inside the **render loop**
- Refpage: https://www.glfw.org/docs/3.3/input_guide.html

```
while (!glfwWindowShouldClose(window))
{
    // send render commands to OpenGL server
    context.render();

    // Check and call events and swap the buffers
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

Sample render loop

Raw Input

- GLFW callback is called **once for each action**
  - E.g., **`GLFW_PRESS`**, **`GLFW_RELEASE`**
  - If you press a key and hold it there, GLFW will call the callback only once!
  - Not the desired behavior under many circumstances (e.g., camera movement via W, S, A, D. You want the camera keep moving rather than move only one step!)
- GLFW provides raw input process interface (i.e., rather than registered callbacks, deal with some specific types of input by ourselves).

```
while (!glfwWindowShouldClose(window))
{
    // ...
    processKeyInput(window);
    // ...
}
```

```
void processKeyInput(GLFWwindow * window)
{
    if (glfwGetKey(window, key: GLFW_KEY_W) == GLFW_PRESS)
    {
        forwardCamera();
    }
}
```

Screen Space Coordinates to NDC

- All coordinates passed to OpenGL pipeline should be in **Normalized Device Coordinate (NDC)**, that is, map the visible region into a unit cube.
- Any vertex outside of the valid range of NDC (i.e., x and y not in the range [-1, 1]) will **not** be rendered.
- We need to map screen space coordinates into NDC before sending them to OpenGL pipeline.

```
// Screen space coordinate to NDC
// x: [0, screenWidth - 1] -> [-1, 1]
// y: [0, screenHeight - 1] -> [-1, 1]
xNDC = 2.0 * x / screenWidth - 1.0,
yNDC = 2.0 * y / screenHeight - 1.0;
```

- The helper function **`Shape::getVertexBufferData`** will do this automatically for you.

More OpenGL Examples

- When the TA was new to OpenGL, he followed the tutorial website: https://learnopengl.com/
- He wrote his own programs as guided by this online tutorial (Chapter "Getting Started", this chapter is sufficient for you to succeed in this course!)
- Source code available at bottom of TA Help Page (also available in your VMWare VM under directory **`~/workspace/`**. You can also go over the online tutorial while playing with the sample code!

Contents

How to Draw Points in PA1

- In PA1, we only require you to understand the following OpenGL stuff (More details left to PA2 recitation):
  - OpenGL environment setup;
  - OpenGL frontend logic (how to use GLFW);
  - And:
- Know **what to do in PA1 template to draw a point**.
- Append points to **`context.vertexBufferData`**
  - **`std::vector<glm::vec2> vertexBufferData;`**
  - Each **`glm::vec2`** (2D vector) represents a 2D point to draw;
  - All coordinates are in **NDC**;
  - PA1 template will render all points stored in this vector for you automatically.

Utils

- **`glm::vec2`** can **not** be printed out with **`std::cout`** by default
- Two ways to print out **`glm::vec2`** for debugging:
  1. Manually print out its x, y components;
  2. **`#include "global/Utils.h"`**

```cpp
#include "global/Utils.h"

glm::vec2 p {x: 0.2, y: 0.5};
std::cout << p.x << ' ' << p.y << '\n';  // 1
std::cout << p << '\n';                  // 2
```

PA1 Template

- PA1 template has taken care of most frontend logic and OpenGL stuff
  - **Except** poly-lines and polygons, which could be done by stacking the raster pixels of its components (line segments);
  - This is done by updating `vertexBufferData` in corresponding TODOs.
  - All you need to do in PA1 is to setup `vertexBufferData` properly. All post-processing on it all and OpenGL-related stuff is already done in PA1 template.
- You need to complete:
  - Midpoint algorithm for all geometric primitives;
  - You simply call `appendToPath` method with **screen-space coordinates** when you want to set a pixel;
  - **The program template will deal with all other stuff for you**, including coordinate transformation, buffering VBO, calling rendering commands, etc.
  - Again, **except** poly-lines and polygons, as they can be implemented only by frontend logic without their own version of midpoint algorithm!

Most Pertinent Stuffs

- What you should master from this tutorial to complete PA1:
  - Know the common sense in OpenGL (as covered today) ;
  - Know the basic stuff in C/C++ programming (as covered today);
  - Know how to configure OpenGL context in your program;
  - Know how to deal with input events with GLFW;
  - Know how to render points with PA1 template.
- What we will leave to future recitations:
  - How to setup VAO and VBO;
  - How VAO manage OpenGL generic attribute arrays and VBOs;
  - And more OpenGL stuff not mentioned today.

General Tips

- **Start early**, and go to TA Office Hour early;
- Consider the structure of your code before you start;
- Code and test bit by bit;
- Write good comments;
- Google official documents rather than copy-and-paste from random untrusted sources!

Thank you!

- Q & A