CS3388B: Lecture 8

February 7, 2023

8 User Interaction

A large part of what makes graphical computer programs so pervasive is the instantaneous **interaction** and **feedback** they give to the users.

Up to now, we have been rendering static scenes or scenes which are following a pre-defined animation. In this section we will examine the basic building blocks of receiving user input, processing it, and providing feedback to the user.

8.1 Interaction Paradigms

The existence of a user's interaction with a program brings about the existence of **event-driven programming**. This programming paradigm is one where the flow of execution of a program is directly manipulated by the user's interactions.

This most programs you use. Indeed, since you are "using" the program, you must be interacting with it in some way. A counter-example would be a scientific simulation. After the user "sets up" the experiment, the rules of the simulation dictate how the program evolves.

There are two main ways which a program can receive and process user interaction.

1. Polling
2. "Event Handlers": Callbacks

In polling, the program consistently asks (the windowing system, graphics API, etc.) "did this event A happen?" "Did event B happen?" It's not so simple that it asks "did any event happen?" Rather, it only asks if certain events of interest happen.

For example, "Has the user pressed the *Enter* key?"

Using callbacks, the program registers **callbacks** or **event handlers** with the API. These callbacks say the following: "if event A happens, call the function `callbackA()`."

Both are useful in their own right and depends on the needs of the program.

8.2 Mouse Interaction

The mouse (or touch pad, or touch screen) is a critical **peripheral device** which allows users to interact with precise locations in a graphical program. It, of course, controls the position of the **cursor** of the windowing system. It also allows the program to receive the events based on left, middle, and right mouse button clicks, presses, and holds.

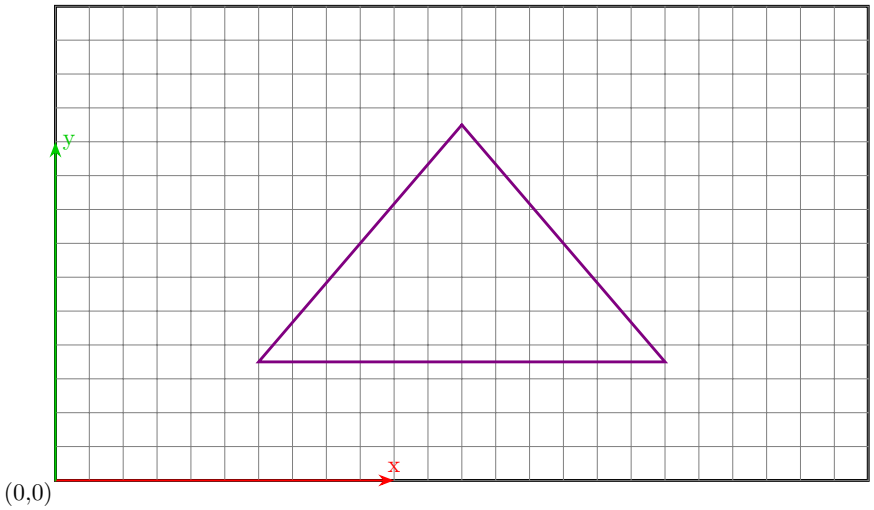

Figure 1: Viewport coordinates

We have seen that the viewport matrix in OpenGL defines a system which maps NDC to a *canvas* or *viewport* with bottom-left origin as a subset of the overall window.

In most windowing systems, their coordinate system has a top-left origin, with positive x extending right and positive y extending down. This contrasts with the **window coordinates** in two ways. First, the viewport's origin may not be the window's origin. The viewport's origin may not even be at *any* of the window's corners or edges. Second, the viewport has the positive y axis extending up. The window (typically) has positive y extending down.

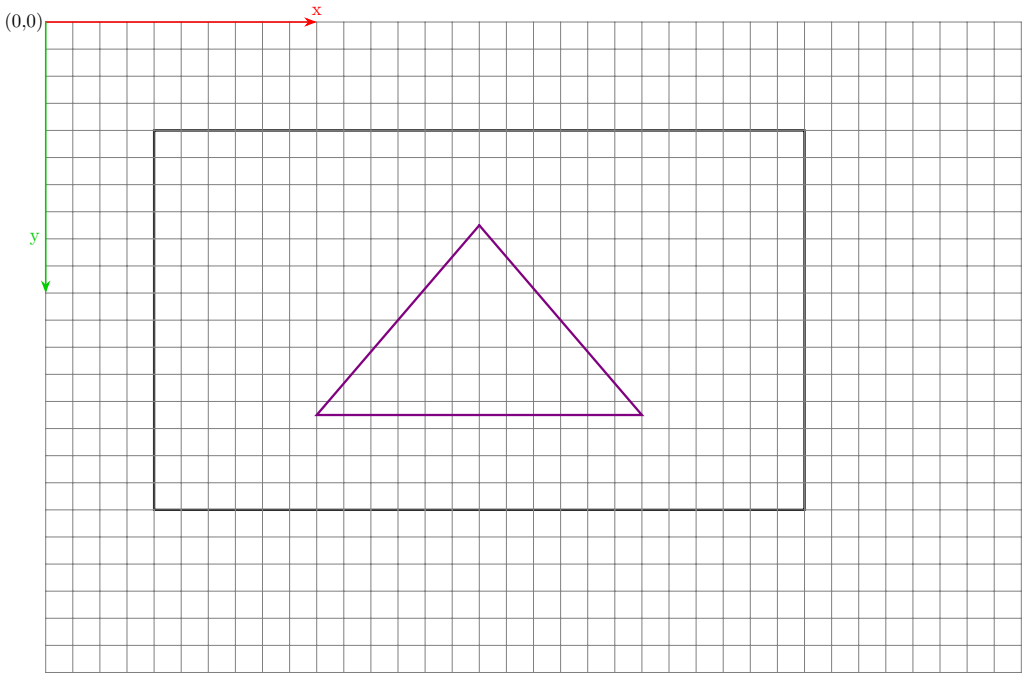

Figure 2: Window coordinates

In GLFW, and likewise in most windowing and GUI APIs, getting the cursor position is easy:

```
void glfwGetCursorPos(GLFWwindow* window, double* xpos, double* ypos);

GLFWwindow window* = glfwCreateWindow(...);

double mouseX, mouseY;
glfwGetCursorPos(window, &mouseX, &mouseY);
```

If you want a certain function to be called every time the cursor position changes, there's a callback for that too!

```
void myMousePosCallback(GLFWwindow* window, double xpos, double ypos) {
    std::cout << "Mouse moved to: " << xpos << "," << ypos << std::endl;
};

GLFWwindow window* = glfwCreateWindow(...);
glfwSetCursorPosCallback(window, myMousePosCallback);
```

When we receive the cursor's position (by polling or a callback), its coordinates are in the window coordinate system. The *units* of the window coordinate system are measured in *pixels*. The window's top-left corner has coordinates (0,0).

We must manually convert those window coordinates to viewport coordinates if we want to determine what part of the scene the cursor is over. How? See problem set 5.

8.3 Button/Key Interaction

Buttons typically refer to the button on a mouse. Keys typically refer to the keys on a keyboard. They have typically have separate functions in an API to get their state. But, they both share the same kinds of state.

In the *most basic* setup, a key/button has exactly two states:

1. **Released**: The key/button is not currently pressed/activated.
2. **Pressed**: The key/button is currently pressed/activated.

Keys on the keyboard also have a special state: **repeat**. After holding down a key long enough, an event is triggered which causes that key to be "repeated" as if it was pressed again.

For example: **AHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH**

And that's really the basics. A key is pressed or it is not pressed. It's a binary switch.

You can, of course, register a callback to say "call this function when the state of key A changes"

In GLFW, a key (not mouse button) callback receives five parameters:

1. The window pointer

2. The key whose state changed, as an enumeration (e.g. GLFW_KEY_A)
3. An OS-specific "scancode".
4. The action/event that occurred. Press, Release, or Repeat.
5. An integer which represents what combination of Ctrl/Alt/Shift were pressed.

```
void pressedA(GLFWwindow* window, int key, int scancode, int action, int
    mods) {
     if (key == GLFW_KEY_A) {
          if (action == GLFW_PRESS) {
               std::cout << "A is pressed\n";
          } else {
               std::cout << "A is released\n";
          }
     }
}


GLFWwindow window* = glfwCreateWindow(...);
glfwSetKeyCallback( window, keyPressedCallback);
```

You can also poll the state of a particular key, rather than a single callback for all keys.

```
if (glfwGetKey(window,GLFW_KEY_E) == GLFW_PRESS) {
     std::cerr << "E is pressed\n";
}
if (glfwGetKey(window,GLFW_KEY_E) == GLFW_RELEASE) {
     std::cerr << "E is not pressed\n";
}
```

For mouse buttons it is very much the same.

A mouse button callback has four parameters: the window, the button whose state is modified, the action that occurred, and any modifiers that were also pressed (Ctrl/Alt/Shift).

For a mouse, it has three (main) buttons: left, right, middle. The keyboard has many more options. To get the state of the left mouse button, for example, we just call

`glfwGetMouseButton(window, MOUSE_BUTTON,`

where `MOUSE_BUTTON` comes from a pre-defined constant:

1. `GLFW_MOUSE_BUTTON_LEFT`
2. `GLFW_MOUSE_BUTTON_RIGHT`
3. `GLFW_MOUSE_BUTTON_MIDDLE`

8.4 More Sophisticated Events

Sometimes a graphics API gives you more information than just yes/no, pressed/released. Those of interest are:

1. **Mouse clicked**: a mouse button we pressed and released in quick succession
2. **Mouse double clicked**
3. **Mouse/Key down**: a single event that is triggered the *first frame* a button/key is pressed
4. **Mouse/Key up**: a single event that is triggered the *first frame* a button/key is no longer pressed
5. **Mouse/Key held**: a event that is triggered *each frame after the first* that a button/key is pressed.

These are sometimes called **virtual events** since such events can be defined in terms of the basic pressed/release binary events. How? Simple Booleans and polling.

```
bool leftButtonPressed = false;

while (!glfwWindowShouldClose(window))
{
    /* Poll for and process events */
    glfwPollEvents();

    int leftState = glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT);
    if (leftState == GLFW_PRESS && leftButtonPressed) {
        //left mouse button is being held

    } else if (leftState == GLFW_PRESS && !leftButtonPressed) {
        //This is the first frame that mouse mouse button is pressed.
        //"Button down" event
        leftButtonPressed = true;

    } else if (leftState == GLFW_RELEASE && leftButtonPressed) {
        //This is the first frame the mouse button is released.
        //"Button up" event

    } else {
        //Current state is GLFW_RELEASE and leftButtonPressed is false
        //Don't do anything
    }
}
```