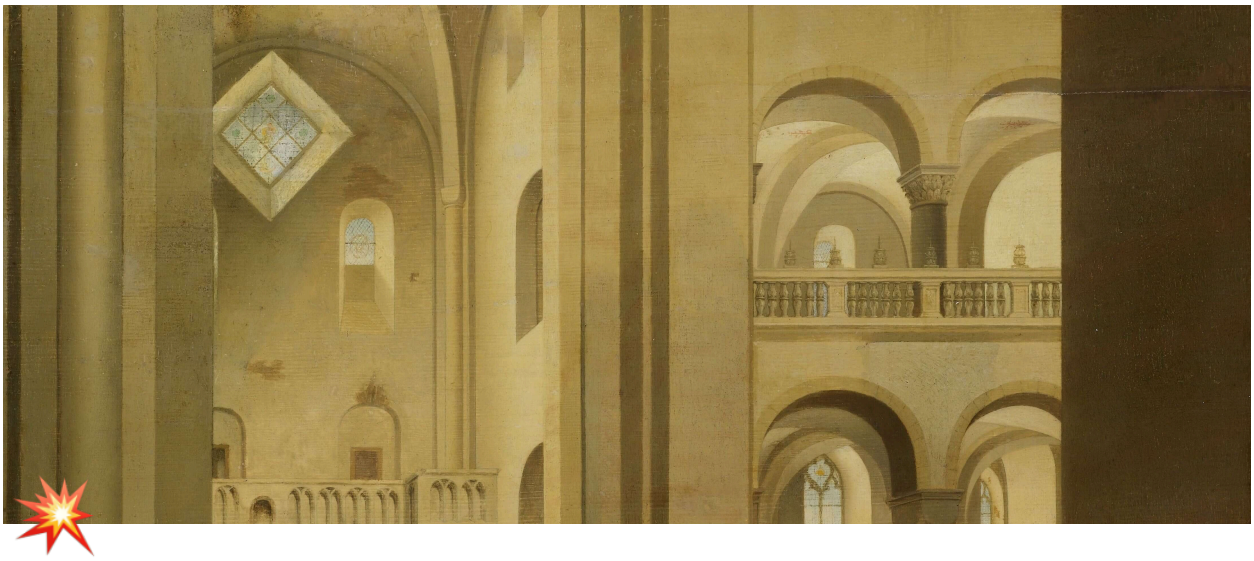

Section 6: Create a polyline using the mouse

The Main concept of call back function

```
// Callback function for mouse button events
void mouseButtonCallback(GLFWwindow* window, int button, int action, int mods) {
    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS) {
        double xpos, ypos;
        glfwGetCursorPos(window, &xpos, &ypos);

        // Convert screen coordinates to OpenGL coordinates (-1 to 1)
        int width, height;
        glfwGetWindowSize(window, &width, &height);
        float x = (2.0f * xpos) / width - 1.0f;
        float y = 1.0f - (2.0f * ypos) / height;

        points.push_back(x);
        points.push_back(y);
    }
    // Clear points if right mouse button is pressed
    else if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_PRESS) {
        points.clear();
    }
}
```

The `mouseButtonCallback` function is designed to handle mouse button events in an application using the GLFW library. It processes mouse clicks and updates a collection of points based on the user's interactions. Here's a step-by-step breakdown of what this function does:

Function Signature

```
void mouseButtonCallback(GLFWwindow* window, int button, int action, int mods)
```

- **Parameters**:
  - `GLFWwindow* window`: A pointer to the window that received the mouse event.
  - `int button`: The mouse button that was pressed or released (e.g., left, right).
  - `int action`: The action that occurred (e.g., press or release).
  - `int mods`: Any modifier keys that were held down during the event (e.g., Shift, Control).

Steps Taken by the Function

1. **Check for Left Mouse Button Press**:

   ```
   if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS) {
   ```

   - The function first checks if the left mouse button was pressed. If so, it proceeds to the next steps.

2. **Get Cursor Position**:

   ```
   double xpos, ypos;
   glfwGetCursorPos(window, &xpos, &ypos);
   ```

   - It retrieves the current position of the mouse cursor in screen coordinates (xpos and ypos) using `glfwGetCursorPos`.
   - The function then obtains the dimensions of the window (width and height) using `glfwGetWindowSize`. This information is necessary for converting screen coordinates to OpenGL coordinates.

3. **Get Window Size**:

   ```
   int width, height;
   glfwGetWindowSize(window, &width, &height);
   ```

4. **Convert Screen Coordinates to OpenGL Coordinates**:

   ```
   float x = (2.0f * xpos) / width - 1.0f;
   float y = 1.0f - (2.0f * ypos) / height;
   ```

- The screen coordinates are converted to normalized device coordinates (NDC), which range from -1 to 1 in both x and y directions:
  - The x-coordinate is scaled from [0, width] to [-1, 1].
  - The y-coordinate is also scaled from [0, height] to [-1, 1], but with a flip because OpenGL's origin is at the bottom-left corner while screen coordinates start from the top-left.

5. **Store Points**:

```
points.push_back(x);
points.push_back(y);
```

- After conversion, the new point (x, y) is added to a collection called `points` using `push_back`. This typically represents a list of points where the user has clicked.

6. **Check for Right Mouse Button Press**:

```
else if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_PRESS) {
    points.clear();
}
```

- If the left mouse button was not pressed, it checks if the right mouse button was pressed. If true, it clears all points stored in the `points` collection using `points.clear()`. This effectively resets any previously recorded click positions.

Summary

In summary, this function captures mouse click events in a GLFW window. When the left button is pressed, it records the cursor's position as an OpenGL coordinate and stores it in a vector called `points`. When the right button is pressed, it clears all recorded points. This functionality is useful for applications such as drawing or selecting points in a graphical interface.

---

now that we defined our main function that will take our mouse inputs and create a normalized vertices from them we need a place to save those vertices we will use `Vectors` for this

What are Vectors

In C++, a **vector** is a dynamic array provided by the Standard Template Library (STL) that can resize itself automatically when elements are added or removed. Vectors are sequence containers that store elements of the same type in a linear arrangement, allowing for efficient random access to any element.

Key Features of C++ Vectors

- **Dynamic Sizing**: Unlike static arrays, vectors can grow and shrink in size as needed, making them suitable for situations where the number of elements is not known at compile time 14.
- **Template Class**: Vectors are implemented as a class template, which allows them to store any data type, such as `int`, `double`, or user-defined types .

- **Memory Management**: Vectors handle memory allocation automatically. They manage their own storage and can reallocate memory as needed when the size changes .
- **Random Access**: Vectors support fast random access to elements using an index, similar to arrays. This means you can access any element in constant time *O*(1)

Basic Syntax

To declare a vector in C++, you include the vector header and use the following syntax:

Implementation

Step 1: import libraries and our source shaders

```
#include <glad/glad.h>// Include the glad library
#include <GLFW/glfw3.h> // Include the GLFW library
#include <vector> // Include the vector library
#include <iostream>// Include the iostream library

// Vertex shader source code
const char* vertexShaderSource = R"(
    #version 330 core
    layout (location = 0) in vec2 aPos;
    void main() {
        gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
    }
)";

// Fragment shader source code
const char* fragmentShaderSource = R"(
    #version 330 core
    out vec4 FragColor;
    void main() {
        FragColor = vec4(1.0, 1.0, 1.0, 1.0); // White color
    }
)";
```

Step 2: create our vector that will store our data and the callback function

```
std::vector<float> points; // Store points for the polyline

// Callback function for mouse button events
void mouseButtonCallback(GLFWwindow* window, int button, int action, int mods) {
    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS) {
        double xpos, ypos;
        glfwGetCursorPos(window, &xpos, &ypos);

        // Convert screen coordinates to OpenGL coordinates (-1 to 1)
        int width, height;
        glfwGetWindowSize(window, &width, &height);
        float x = (2.0f * xpos) / width - 1.0f;
        float y = 1.0f - (2.0f * ypos) / height;

        points.push_back(x);
        points.push_back(y);
    }
    // Clear points if right mouse button is pressed
    else if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_PRESS) {
        points.clear();
    }
}
```

Step 3: Create our main function and window , Viewport ,and set the mouse button callback function to the current window

```cpp
int main() {
    // Initialize GLFW
    glfwInit();

    // Tell GLFW what version of OpenGL we are using
    // In this case we are using OpenGL 3.3
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    // Tell GLFW we are using the CORE profile
    // So that means we only have the modern functions
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    // Create a GLFWwindow object of 800 by 800 pixels, naming it "Polyine Using Mouse!!"
    GLFWwindow* window = glfwCreateWindow(640, 480, "Polyine Using Mouse!!", NULL, NULL);
    // Error check if the window fails to create
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    // Introduce the window into the current context
    glfwMakeContextCurrent(window);

    //Load GLAD so it configures OpenGL
    gladLoadGL();
    // Specify the viewport of OpenGL in the Window
    // In this case the viewport goes from x = 0, y = 0, to x = 640, y = 480
    glViewport(0, 0, 640, 480);
    //*******************************************************
    // Set mouse button callback
        glfwSetMouseButtonCallback(window, mouseButtonCallback); // The new part in the code
        //*******************************************************
```

Step 4 :continue normally and create shaders, shader program and generate VAO &VBO

```cpp
// Create and compile vertex shader
Gluint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);

// Create and compile fragment shader
Gluint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);

// Create shader program
Gluint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);

// Clean up shaders
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);

// Create VBO and VAO
Gluint VBO, VAO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
```

💡 Note that we did not fully define our VAO and VBO because the data that we will store in them is dynamic so we have to update them during runtime (inside the while loop)
and this is the next step in our code

Step 5: create our while loop

```
// Render loop
while (!glfwWindowShouldClose(window)) {
    // Clear the screen
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);


    // Draw polyline if we have points
    if (points.size() >= 4) { // At least 2 points (4 coordinates)
        glUseProgram(shaderProgram);
            // Update VBO with new points
        glBindVertexArray(VAO);
        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferData(GL_ARRAY_BUFFER, points.size() * sizeof(float), points.data(), GL_DYNAMIC_DRAW);
            // Set vertex attribute pointers
        glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*)0);
        glEnableVertexAttribArray(0);
            // Draw polyline
        glDrawArrays(GL_LINE_STRIP, 0, points.size() / 2);
    }

    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

in this part during the while loop while the window is not closed the loop uses `glfwPollEvents` to check for the user inputs and when the user inputs 2 vertices (4 points)

the if statement condition becomes true and these points are then inserted inside the VBO and VAO to be rendered inside the GPU

Full code

```
#include <glad/glad.h>// Include the glad library
#include <GLFW/glfw3.h> // Include the GLFW library
#include <vector> // Include the vector library
#include <iostream>// Include the iostream library

// Vertex shader source code
const char* vertexShaderSource = R"(
    #version 330 core
    layout (location = 0) in vec2 aPos;
    void main() {
        gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
    }
)";

// Fragment shader source code
const char* fragmentShaderSource = R"(
    #version 330 core
    out vec4 FragColor;
    void main() {
        FragColor = vec4(1.0, 1.0, 1.0, 1.0); // White color
    }
)";

std::vector<float> points; // Store points for the polyline

// Callback function for mouse button events
void mouseButtonCallback(GLFWwindow* window, int button, int action, int mods) {
    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS) {
        double xpos, ypos;
        glfwGetCursorPos(window, &xpos, &ypos);

        // Convert screen coordinates to OpenGL coordinates (-1 to 1)
        int width, height;
        glfwGetWindowSize(window, &width, &height);
        float x = (2.0f * xpos) / width - 1.0f;
        float y = 1.0f - (2.0f * ypos) / height;

        points.push_back(x);
        points.push_back(y);
    }
    // Clear points if right mouse button is pressed
    else if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_PRESS) {
        points.clear();
    }
}

int main() {
    // Initialize GLFW
    glfwInit();

    // Tell GLFW what version of OpenGL we are using
    // In this case we are using OpenGL 3.3
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    // Tell GLFW we are using the CORE profile
    // So that means we only have the modern functions
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    // Create a GLFWwindow object of 800 by 800 pixels, naming it "Polyine Using Mouse!!"
    GLFWwindow* window = glfwCreateWindow(640, 480, "Polyine Using Mouse!!", NULL, NULL);
    // Error check if the window fails to create
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    // Introduce the window into the current context
    glfwMakeContextCurrent(window);

    //Load GLAD so it configures OpenGL
    gladLoadGL();
    // Specify the viewport of OpenGL in the Window
    // In this case the viewport goes from x = 0, y = 0, to x = 640, y = 480
    glViewport(0, 0, 640, 480);

    // Set mouse button callback
    glfwSetMouseButtonCallback(window, mouseButtonCallback);

    // Create and compile vertex shader
    unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
    glCompileShader(vertexShader);

    // Create and compile fragment shader
    unsigned int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
    glCompileShader(fragmentShader);

    // Create shader program
    unsigned int shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    glLinkProgram(shaderProgram);

    // Clean up shaders
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);

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

    // Render loop
    while (!glfwWindowShouldClose(window)) {
        // Clear the screen
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);

        // Draw polyline if we have points
        if (points.size() >= 4) { // At least 2 points (4 coordinates)
            glUseProgram(shaderProgram);
            // Update VBO with new points
            glBindVertexArray(VAO);
            glBindBuffer(GL_ARRAY_BUFFER, VBO);
            glBufferData(GL_ARRAY_BUFFER, points.size() * sizeof(float), points.data(),
GL_DYNAMIC_DRAW);
            // Set vertex attribute pointers
            glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*)0);
            glEnableVertexAttribArray(0);
            // Draw polyline
            glDrawArrays(GL_LINE_STRIP, 0, points.size() / 2);
        }

        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    // Cleanup
    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);
    glDeleteProgram(shaderProgram);

    glfwTerminate();
    return 0;
}
```

Output: