

COMP220: Graphics & Simulation

## **4: Meshes and movement**

# Agenda

- ▶ Portfolio task check-in (sprint planning)
- ▶ Complex meshes (goodbye triangle, hello cube!)
- ▶ First-person camera control

# Next week

- ▶ Timetable change: COMP220 lecture is on **Wednesday morning** in Seminar D
- ▶ **Compulsory** catch-up tutorials on Tuesday 18th, in the afternoon
  - ▶ Book now – if you haven't booked by Friday 5pm then I will book for you!
  - ▶ If you can't make it on Tuesday afternoon, let me know ASAP so that we can arrange another time

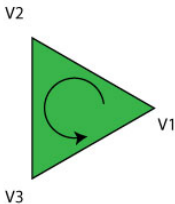
# Portfolio task check-in

- ▶ Please show me your **Trello board** now
- ▶ You are now in your **first sprint**
- ▶ Your first sprint review is **in 2 weeks**

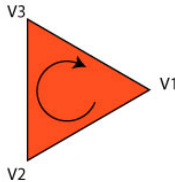
**More complex meshes**

# Winding order

- ▶ It is sometimes important to know which side of a triangle is the “front” and which is the “back”
- ▶ OpenGL determines this by **winding order**



If the vertices go **anticlockwise**, you are looking at the **front**



If the vertices go **clockwise**, you are looking at the **back**

# Backface culling

```
glEnable(GL_CULL_FACE);
```

- ▶ This will cause only the front faces of triangles to be drawn
- ▶ Triangles whose front face is not visible will be **culled**
- ▶ Culled faces are not passed through the rasteriser or fragment shader
- ▶ Saves time, and should make no difference to appearance — as long as all meshes are closed and have correct winding

# When backface culling goes bad?





# Passing coordinates

- ▶ Like many C functions, `glBufferData` takes an array as a **pointer** and a **size**

```
GLfloat vertexPositions[] = {  
    -0.5f, -0.5f, 0.0f,  
    0.5f, -0.5f, 0.0f,  
    0.0f, 0.5f, 0.0f,  
};  
  
glBufferData(GL_ARRAY_BUFFER,  
    sizeof(vertexPositions), // the size  
    vertexPositions, // the pointer  
    GL_STATIC_DRAW);
```

# Passing vertices

- ▶ `glm::vec3` has the same layout in memory as three `GLfloat`s, so we could also do this:

```
glm::vec3 vertexPositions[] = {  
    glm::vec3(-0.5f, -0.5f, 0.0f),  
    glm::vec3(0.5f, -0.5f, 0.0f),  
    glm::vec3(0.0f, 0.5f, 0.0f),  
};  
  
glBufferData(GL_ARRAY_BUFFER,  
    sizeof(vertexPositions), // the size  
    vertexPositions,          // the pointer  
    GL_STATIC_DRAW);
```

- ▶ The third argument to `glBufferData` is a `void*`, which can accept any pointer type

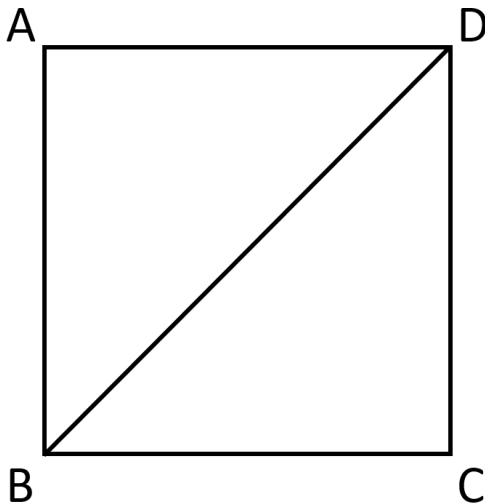
# Vectors of vectors

- ▶ We can use a `std::vector` instead of a C-style array:

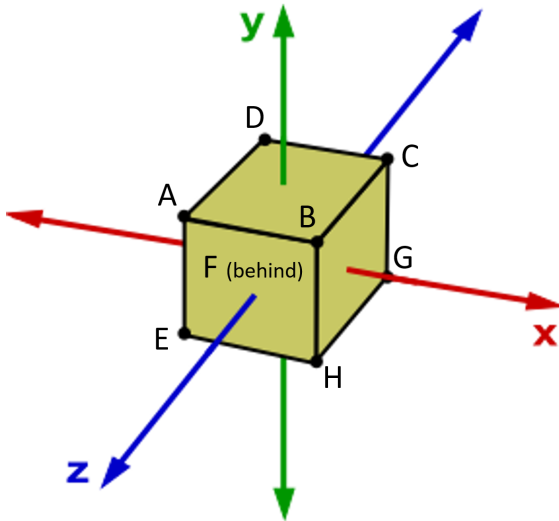
```
std::vector<glm::vec3> vertexPositions;  
vertexPositions.push_back(...);  
  
glBufferData(GL_ARRAY_BUFFER,  
             vertexPositions.size() * sizeof(glm::vec3),  
             vertexPositions.data(),  
             GL_STATIC_DRAW);
```

- ▶ `data()` returns a pointer to the data inside a `std::vector`
- ▶ `size()` returns the number of elements, so multiplying by `sizeof(glm::vec3)` gives the size in bytes

Let's draw a square!



Let's draw a cube!



**First person camera control**

# The plan

- ▶ Represent the player's **position** by a 3D vector
- ▶ Represent the player's **orientation** by Euler angles
- ▶ Mouse events change these angles
- ▶ View matrix is calculated using position and orientation
- ▶ To move forwards, use the Euler angles to find the "forward" vector, and offset the position by this vector

# Keyboard and mouse in SDL

Use **relative mouse mode**

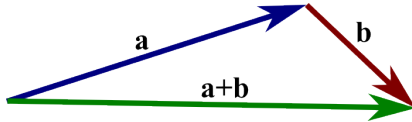
- ▶ Hides the mouse pointer
- ▶ Prevents the mouse pointer from hitting the edge of the screen
- ▶ Gives us the distance the mouse has moved since last frame, rather than its current position

Use `SDL_GetKeyboardState` instead of handling individual keyboard events

- ▶ Allows us to check on every frame whether the key is held down
- ▶ Otherwise, the player will move jerkily according to the key repeat rate



# Vector addition

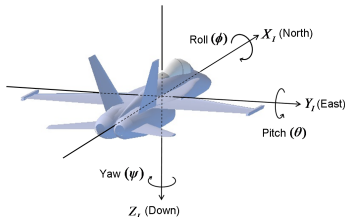


- ▶ E.g. if  $a$  is current position, and
- ▶  $b$  is the distance and direction we want to move, then
- ▶  $a + b$  is the new position

# Unit vectors

- ▶ A **unit vector** is a vector of length 1
- ▶ I.e.  $\sqrt{x^2 + y^2 + z^2} = 1$  (Pythagoras)
- ▶ Useful to represent **direction**
- ▶ Multiplying a vector of length  $a$  by a number  $b$  gives a vector of length  $a \times b$ , parallel to the original vector
- ▶ So multiplying a unit vector by  $b$  gives a vector of length  $b$ , parallel to the unit vector

# Representing look direction



- ▶ Euler angles
- ▶ Don't need roll, just pitch and yaw
- ▶ (Not using roll eliminates the gimbal lock problem)
- ▶ Forward vector and look vector can be obtained by appropriate rotation of a unit vector