COMP220: Graphics & Simulation

3: Vertices, Transforms & Projections

Learning outcomes

By the end of this week, you should be able to:

- **Recall** alternative ways to represent mesh vertices in memory.
- **Apply** basic transforms using the GLM library.
- **Explain** the constituents of the model-view-projection matrix and how it can be used to create a first-person camera controller.

Agenda

Agenda

- Lecture (async):
  - **Compare** different ways to store vertex data in memory.
  - **Review** the transforms required to display 3D objects on a 2D screen.

Agenda

- Lecture (async):
  - **Compare** different ways to store vertex data in memory.
  - **Review** the transforms required to display 3D objects on a 2D screen.
- Workshop (sync):
  - **Adapt** our basic triangle implementation to draw meshes with multiple triangles efficiently.
  - **Experiment** with creating transforms using GLM and using them to move objects and the camera.

FALMOUTH
UNIVERSITY

Vertices and Elements

Vertex Data

Vertex Data

- Vertices always have a **position**, but may also have other data associated, e.g. colour.

Vertex Data

- Vertices always have a **position**, but may also have other data associated, e.g. colour.
- One way we could store this data is to use a separate Vertex Buffer for each **attribute**:

Vertex Data

- Vertices always have a **position**, but may also have other data associated, e.g. colour.
- One way we could store this data is to use a separate Vertex Buffer for each **attribute**:

```
struct Vertex { float x,y,z; };
Vertex v[]={{-0.5f,-0.5f,0.0f},
            {0.5f,-0.5f,0.0f},
            {0.0f,0.5f,0.0f}};

struct Colour { float r,g,b,a; };
Colour c[]={{1.0f,0.0f,0.0f,1.0f},
            {0.0f,1.0f,0.0f,1.0f},
            {0.0f,0.0f,1.0f,1.0f}};
```

Using Separate Vertex Buffers

```
GLuint vertexbuffers[2];
glGenBuffers(2, vertexbuffers);

// Set up vertex position buffer
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffers[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(v), v, GL_STATIC_DRAW);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
        3 * sizeof(GL_FLOAT), (void*)0);

// Set up vertex colour buffer
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffers[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof(c), c, GL_STATIC_DRAW);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE,
        4 * sizeof(GL_FLOAT), (void*)0);
```

Interleaved Vertices

Interleaved Vertices

- Alternatively, we can create a **C structure** which represents a Vertex, which has member variables which represent positions, colours, normals etc.

Interleaved Vertices

- Alternatively, we can create a **C structure** which represents a Vertex, which has member variables which represent positions, colours, normals etc.
- This is known as **Interleaved Vertices** and in **MOST** cases is more efficient.

Interleaved Vertices

- Alternatively, we can create a **C structure** which represents a Vertex, which has member variables which represent positions, colours, normals etc.
- This is known as **Interleaved Vertices** and in **MOST** cases is more efficient.

```
struct Vertex
{
    float x,y,z;
    float r,g,b,a;
};

Vertex v[]={{-0.5f,-0.5f,0.0f, 1.0f,0.0f,0.0f,1.0f},
            {0.5f,-0.5f,0.0f,  0.0f,1.0f,0.0f,1.0f},
            {0.0f,0.5f,0.0f,   0.0f,0.0f,1.0f,1.0f}};
```

Using a Single Buffer

Using a Single Buffer

- We need to update the Vertex Attribute Array to interpret the data correctly:

Using a Single Buffer

- We need to update the Vertex Attribute Array to interpret the data correctly:
  - Increase the **stride** for all attributes

Using a Single Buffer

- We need to update the Vertex Attribute Array to interpret the data correctly:
  - Increase the **stride** for all attributes
  - Include the **offset** for everything but the position

Using a Single Buffer

- We need to update the Vertex Attribute Array to interpret the data correctly:
  - Increase the **stride** for all attributes
  - Include the **offset** for everything but the position

```
GLuint vertexbuffer;
glGenBuffers(1, &vertexbuffer);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(v), v, GL_STATIC_DRAW);

glEnableVertexAttribArray(0);    // Position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
        sizeof(Vertex), (void*)0);

glEnableVertexAttribArray(1);    // Colour attribute
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE,
        sizeof(Vertex), (void*)(3 * sizeof(GL_FLOAT)));
```

Vertices in Memory

	X	Y	Z	R	G	B	A
V_0	-0.5	-0.5	0.0	1.0	0.0	0.0	1.0
V_1	0.5	-0.5	0.0	0.0	1.1	0.0	1.0
V_2	0.0	0.2	0.0	0.0	0.0	1.0	1.0

Vertices in Memory: Stride

V_0	-0.5	-0.5	0.0	1.0	0.0	0.0	1.0
V_1	0.5	-0.5	0.0	0.0	1.1	0.0	1.0
V_2	0.0	0.2	0.0	0.0	0.0	1.0	1.0

Vertices in Memory: Offset

V_0	-0.5	-0.5	0.0	1.0	0.0	0.0	1.0
V_1	0.5	-0.5	0.0	0.0	1.1	0.0	1.0
V_2	0.0	0.2	0.0	0.0	0.0	1.0	1.0

Multiple Triangles

Multiple Triangles

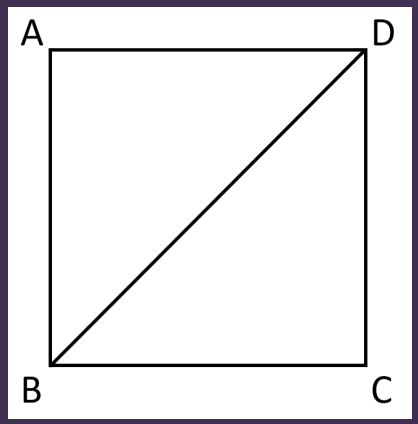
- For the triangle, we listed each vertex's data in the buffer.

Multiple Triangles

- For the triangle, we listed each vertex's data in the buffer.
- What if we have shapes made of multiple triangles (known as **meshes**)?

Multiple Triangles

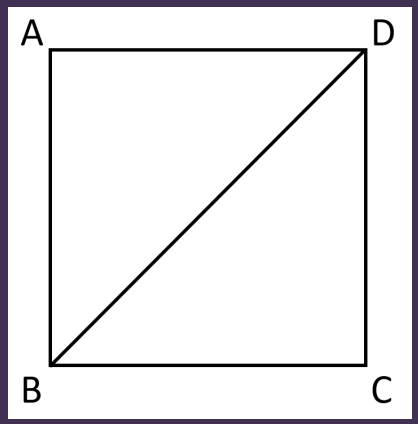
- For the triangle, we listed each vertex's data in the buffer.
- What if we have shapes made of multiple triangles (known as **meshes**)?



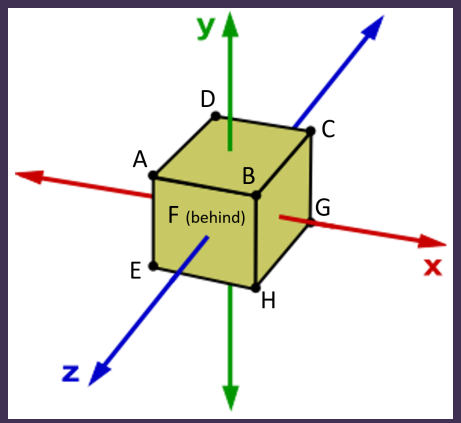
Square: 4 vertices
2 triangles: 6 vertices

Multiple Triangles

- For the triangle, we listed each vertex's data in the buffer.
- What if we have shapes made of multiple triangles (known as **meshes**)?



Square: 4 vertices
2 triangles: 6 vertices



Cube: 8 vertices
12 triangles: 36 vertices
(6 square faces)

Element Buffer

Element Buffer

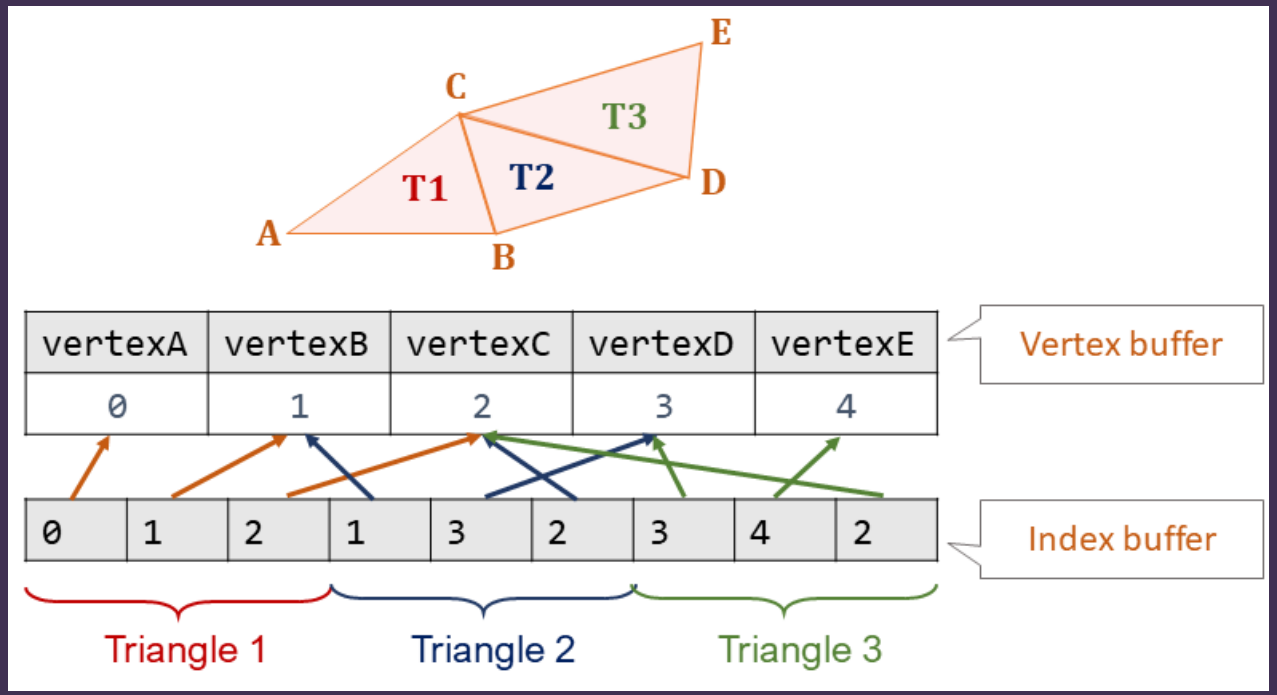
- We can use an **Element Buffer** to optimise our drawing and avoid storing duplicate vertices.

Element Buffer

- We can use an **Element Buffer** to optimise our drawing and avoid storing duplicate vertices.
- An Element Buffer holds an integer which is an **offset** (or index) into a Vertex Buffer.

Element Buffer

- We can use an **Element Buffer** to optimise our drawing and avoid storing duplicate vertices.
- An Element Buffer holds an integer which is an **offset** (or index) into a Vertex Buffer.



Vertex Order

Vertex Order

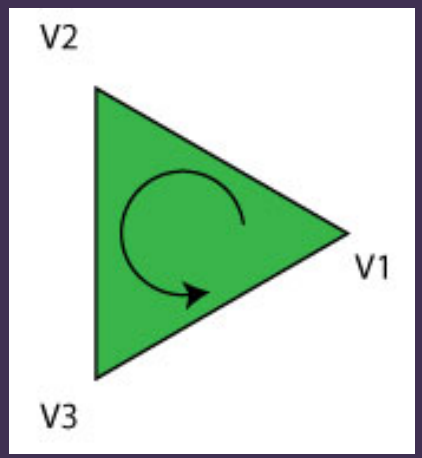
- It is sometimes important to know which side of a triangle is the “front” and which is the “back”

Vertex 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**

Vertex Order

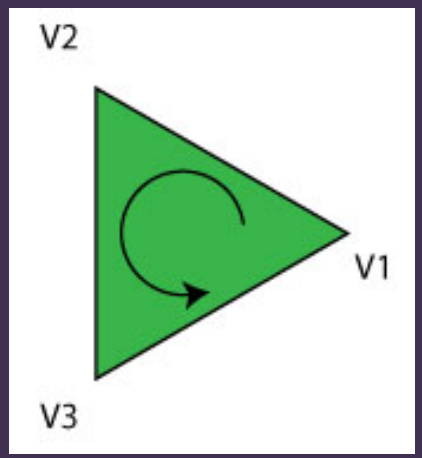
- It is sometimes important to know which side of a triangle is the "front" and which is the "back"
- OpenGL determines this by **winding order**



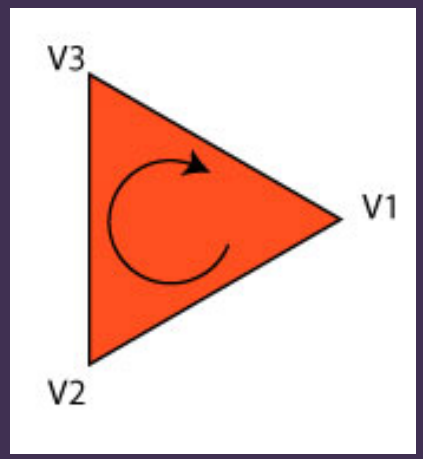
If the vertices go **anticlockwise**, you are looking at the **front**

Vertex 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

Backface culling

```
glEnable(GL_CULL_FACE);
```

Backface culling

```
glEnable(GL_CULL_FACE);
```

- This will cause only the front faces of triangles to be drawn

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**

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

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?

FALMOUTH
UNIVERSITY

Transforms & Projections

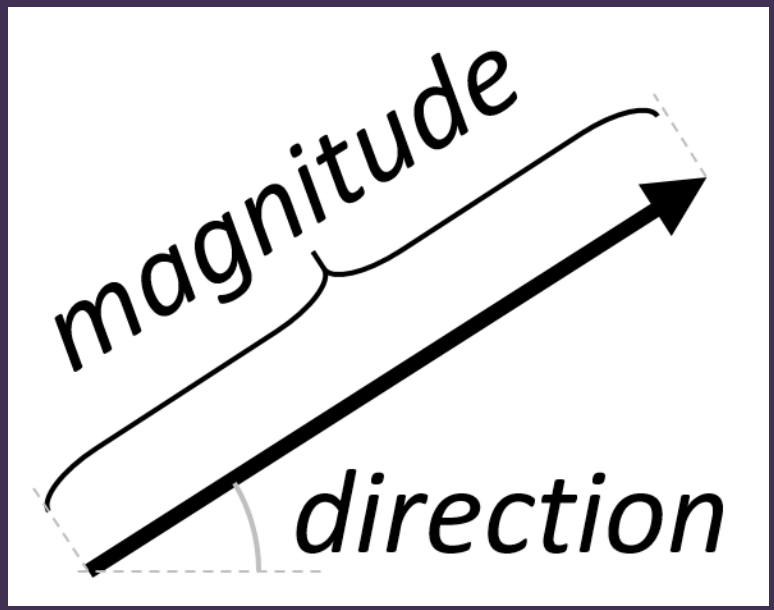
Recap: Vectors

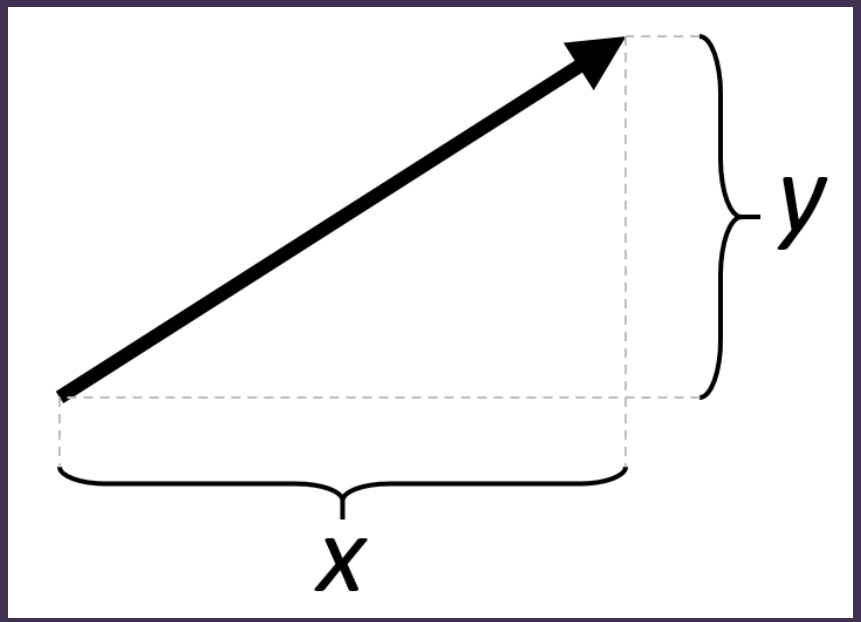
Recap: Vectors

- A **vector** has a **direction** and a **magnitude**, represented by its **components**:

Recap: Vectors

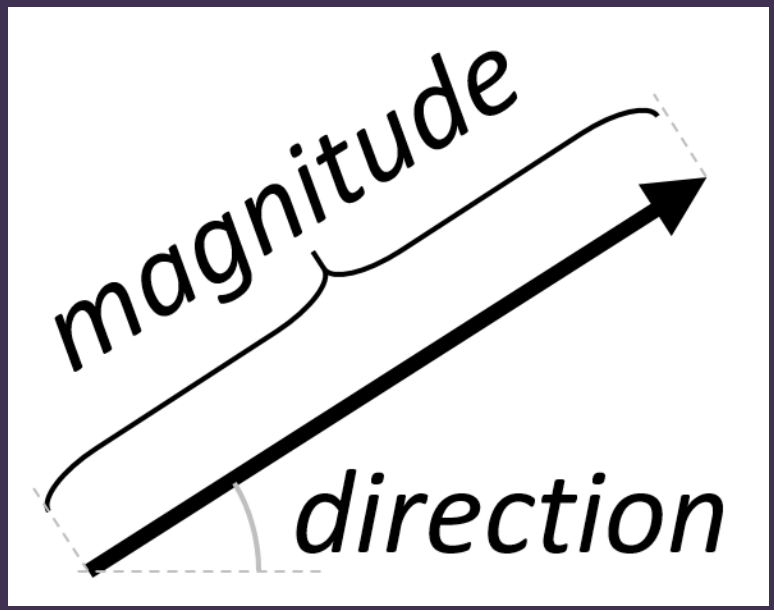
- A **vector** has a **direction** and a **magnitude**, represented by its **components**:

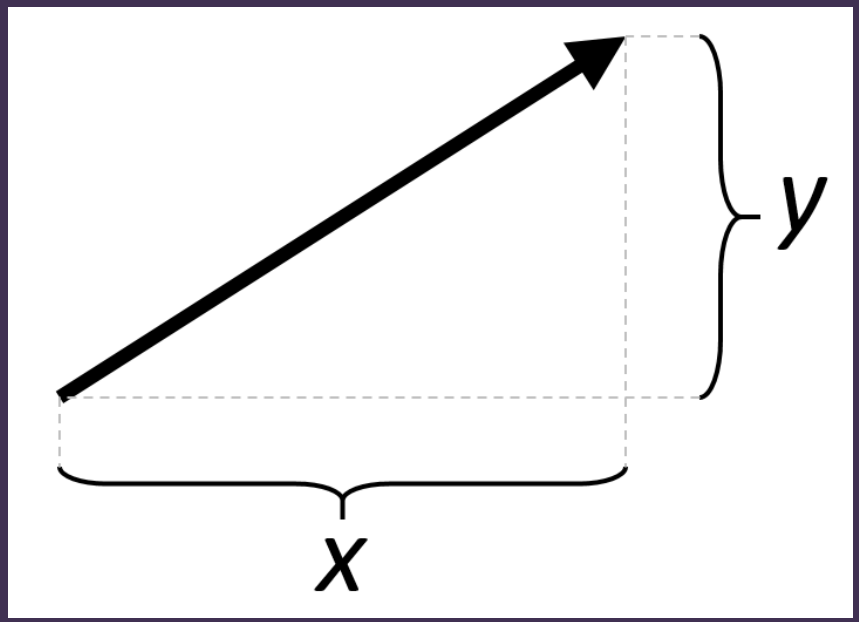




Recap: Vectors

- A **vector** has a **direction** and a **magnitude**, represented by its **components**:





- A vector can represent a position in space as an offset from the origin.

Recap: Matrices and Transforms

Recap: Matrices and Transforms

- A vector can be transformed (moved in space) by multiplying with a **matrix**:

Recap: Matrices and Transforms

- A vector can be transformed (moved in space) by multiplying with a **matrix**:
  - **Translation** applies a linear offset to each component (equivalent to vector addition).

Recap: Matrices and Transforms

- A vector can be transformed (moved in space) by multiplying with a **matrix**:
  - **Translation** applies a linear offset to each component (equivalent to vector addition).
  - **Rotation** applies an angular displacement around the origin.

Recap: Matrices and Transforms

- A vector can be transformed (moved in space) by multiplying with a **matrix**:
  - **Translation** applies a linear offset to each component (equivalent to vector addition).
  - **Rotation** applies an angular displacement around the origin.
  - **Scale** multiplies each component by a constant (equivalent to scalar multiplication for uniform scale).

Recap: Matrices and Transforms

- A vector can be transformed (moved in space) by multiplying with a **matrix**:
  - **Translation** applies a linear offset to each component (equivalent to vector addition).
  - **Rotation** applies an angular displacement around the origin.
  - **Scale** multiplies each component by a constant (equivalent to scalar multiplication for uniform scale).
- Transforms can be combined by **multiplying** the matrices together.

Recap: Matrices and Transforms

- A vector can be transformed (moved in space) by multiplying with a **matrix**:
  - **Translation** applies a linear offset to each component (equivalent to vector addition).
  - **Rotation** applies an angular displacement around the origin.
  - **Scale** multiplies each component by a constant (equivalent to scalar multiplication for uniform scale).
- Transforms can be combined by **multiplying** the matrices together.
  - Matrix multiplication is **NOT commutative**: the order matters.

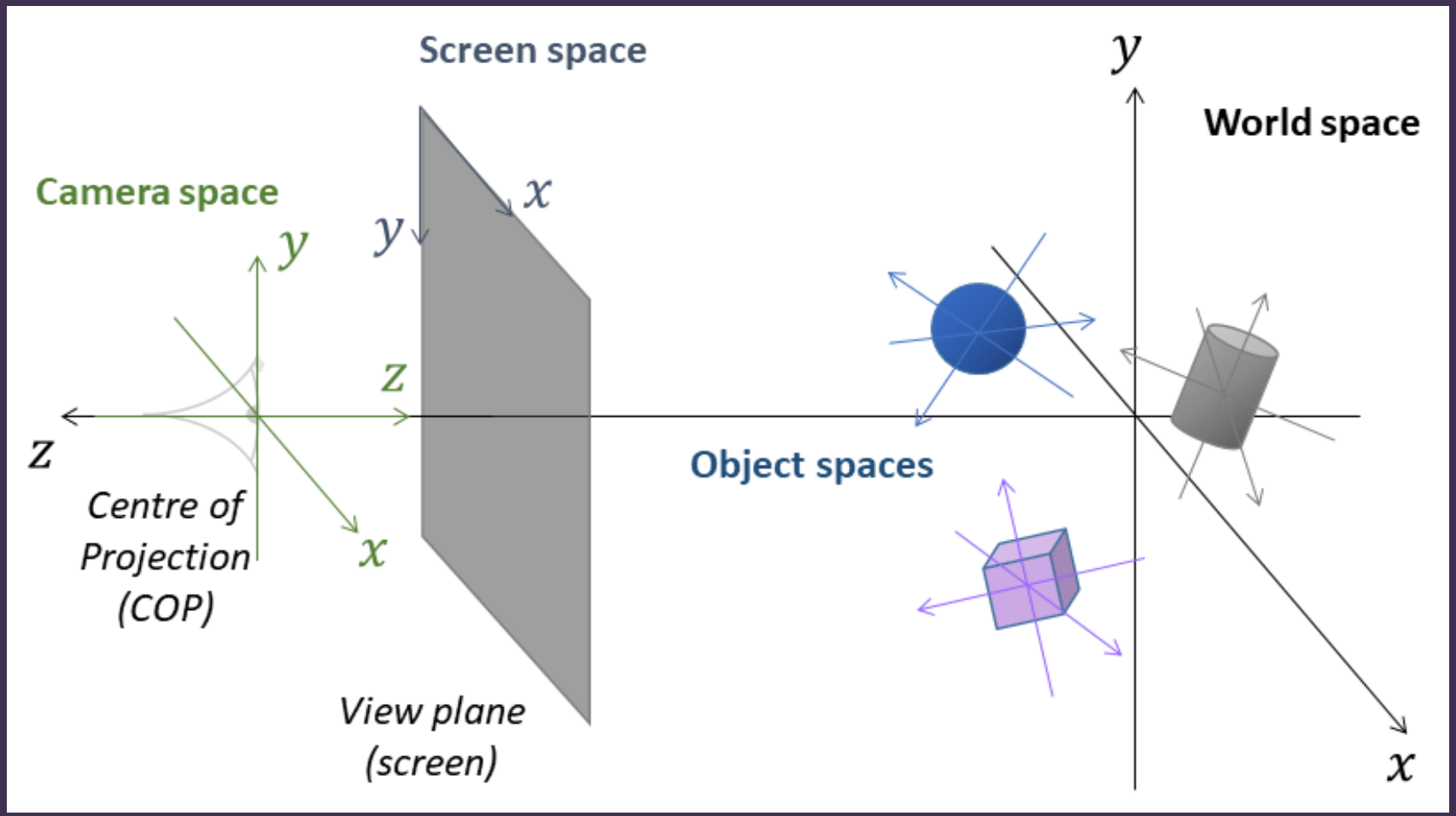
Recap: Matrices and Transforms

- A vector can be transformed (moved in space) by multiplying with a **matrix**:
  - **Translation** applies a linear offset to each component (equivalent to vector addition).
  - **Rotation** applies an angular displacement around the origin.
  - **Scale** multiplies each component by a constant (equivalent to scalar multiplication for uniform scale).
- Transforms can be combined by **multiplying** the matrices together.
  - Matrix multiplication is **NOT commutative**: the order matters.
  - Standard order is **right to left**.

Recap: Matrices and Transforms

- A vector can be transformed (moved in space) by multiplying with a **matrix**:
  - **Translation** applies a linear offset to each component (equivalent to vector addition).
  - **Rotation** applies an angular displacement around the origin.
  - **Scale** multiplies each component by a constant (equivalent to scalar multiplication for uniform scale).
- Transforms can be combined by **multiplying** the matrices together.
  - Matrix multiplication is **NOT commutative**: the order matters.
  - Standard order is **right to left**.
  - It's generally best to apply scale first, then rotation, then finally translation - otherwise results can be unexpected...

Recap: Coordinate Spaces



Model, View, Projection

Model, View, Projection

Drawing a 3D object on screen generally involves **three** transformations:

Model, View, Projection

Drawing a 3D object on screen generally involves **three** transformations:

- **Model**: translate, rotate and scale the object into its place in the scene

Model, View, Projection

Drawing a 3D object on screen generally involves **three** transformations:

- **Model**: translate, rotate and scale the object into its place in the scene
- **View**: translate and rotate the scene to put the observer at the origin

Model, View, Projection

Drawing a 3D object on screen generally involves **three** transformations:

- **Model**: translate, rotate and scale the object into its place in the scene
- **View**: translate and rotate the scene to put the observer at the origin
- **Projection**: convert points in 3D space to points on the 2D screen

Model, View, Projection

Drawing a 3D object on screen generally involves **three** transformations:

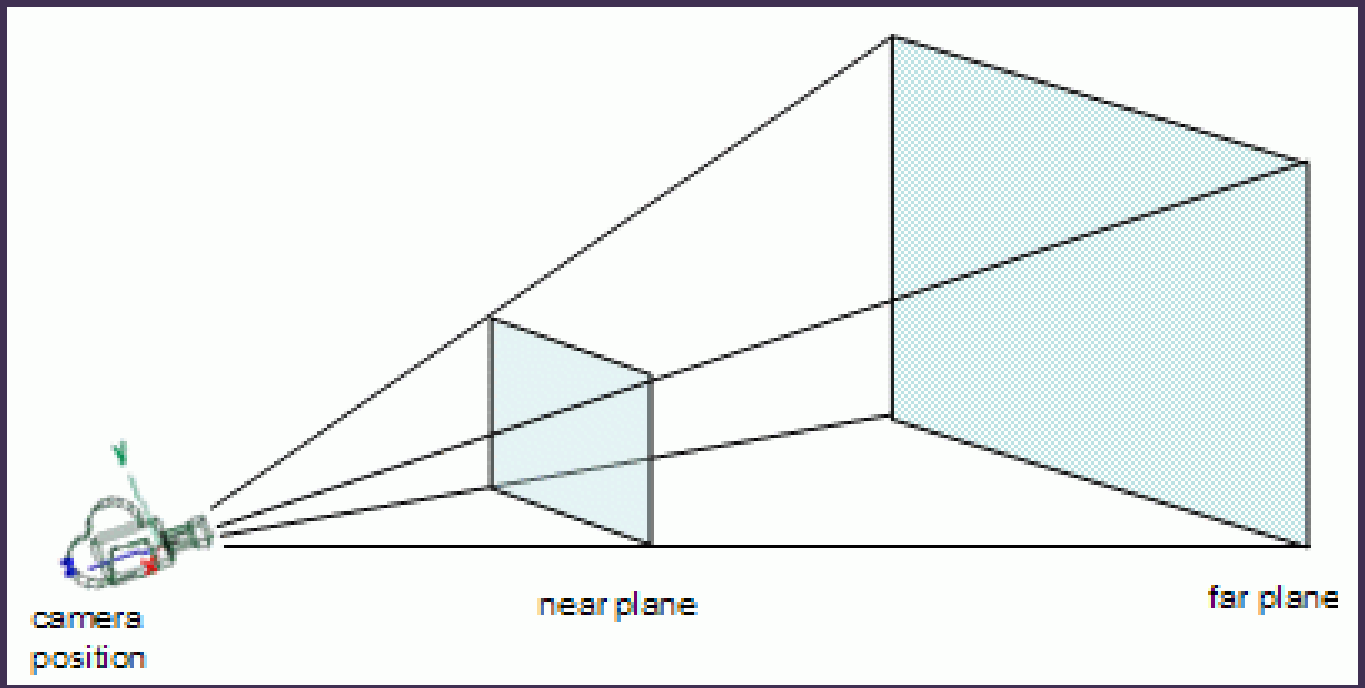
- **Model**: translate, rotate and scale the object into its place in the scene
- **View**: translate and rotate the scene to put the observer at the origin
- **Projection**: convert points in 3D space to points on the 2D screen

The **model-view-projection (MVP) matrix**:

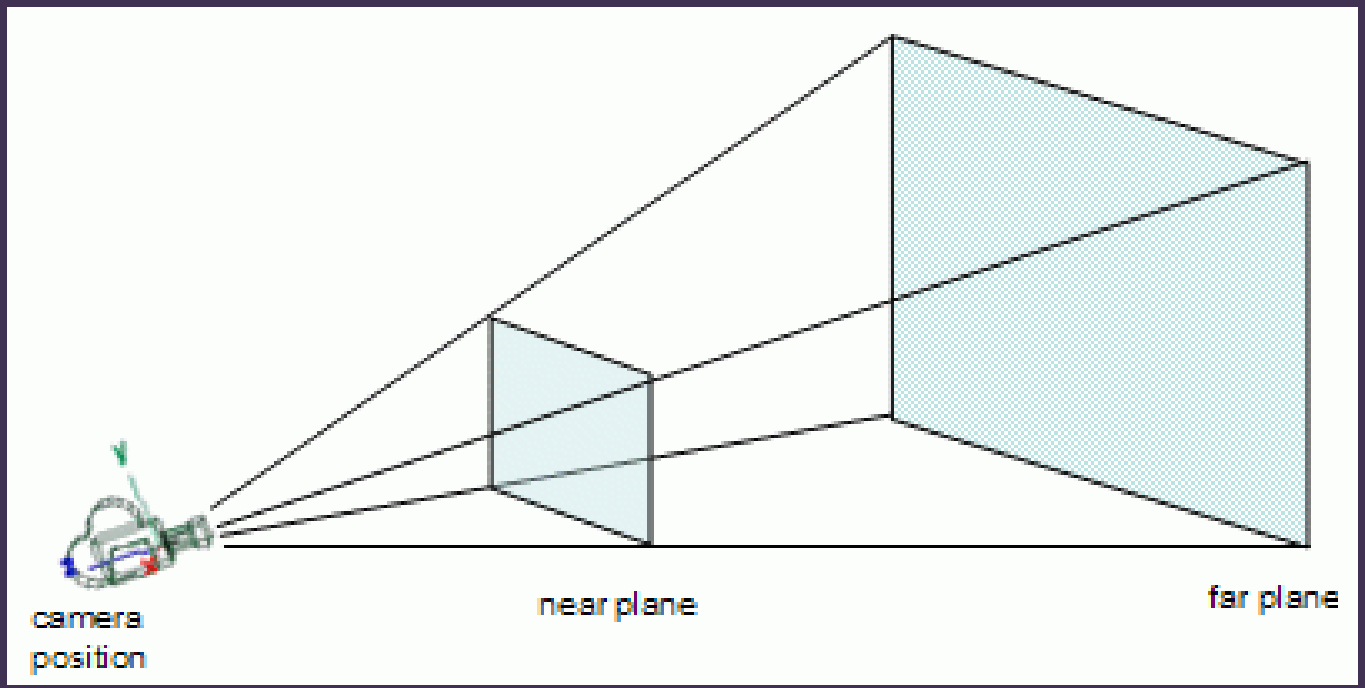
$$M_{MVP} = M_{\text{projection}} \times M_{\text{view}} \times M_{\text{model}}$$

The view frustum

The view frustum

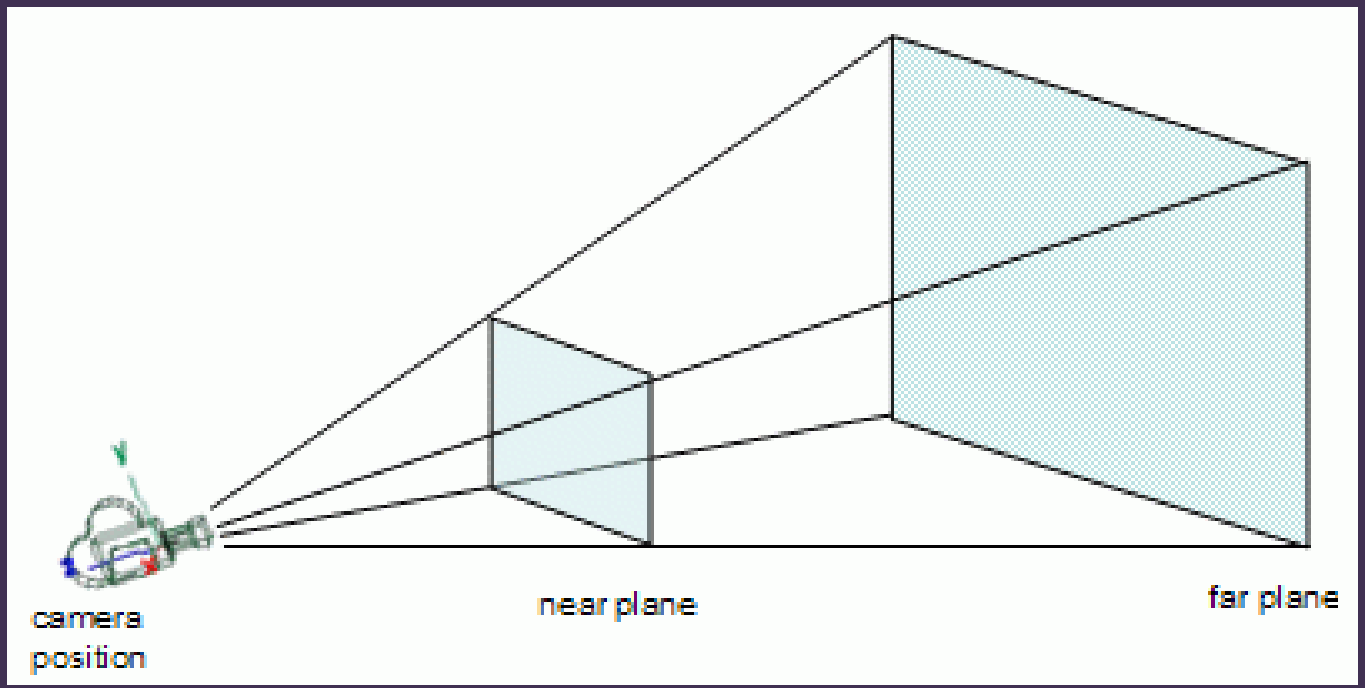


The view frustum



- Defined by the **near and far clipping planes** and the **edges of the screen**

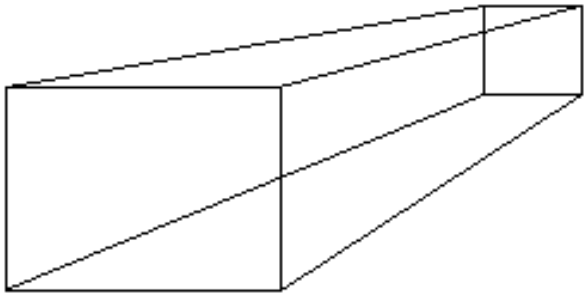
The view frustum



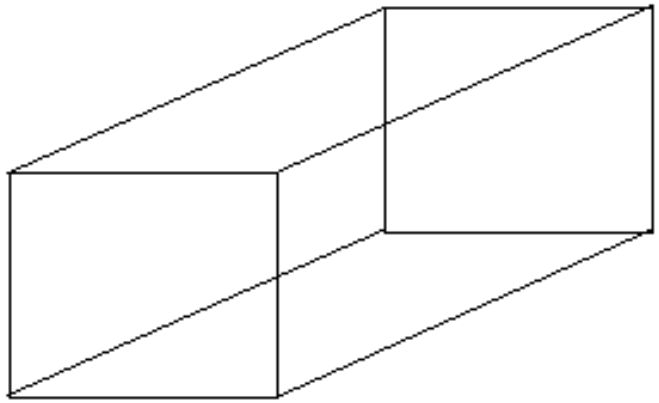
- Defined by the **near and far clipping planes** and the **edges of the screen**
- **Nothing outside** the view frustum is visible

Types of projection

Types of projection

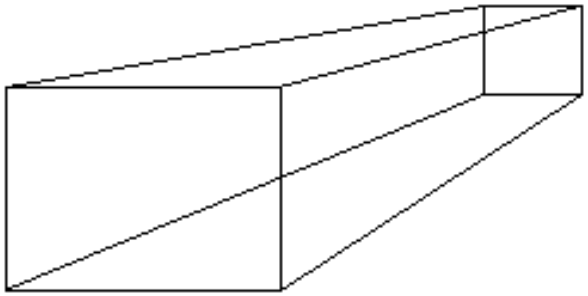


Perspective projection

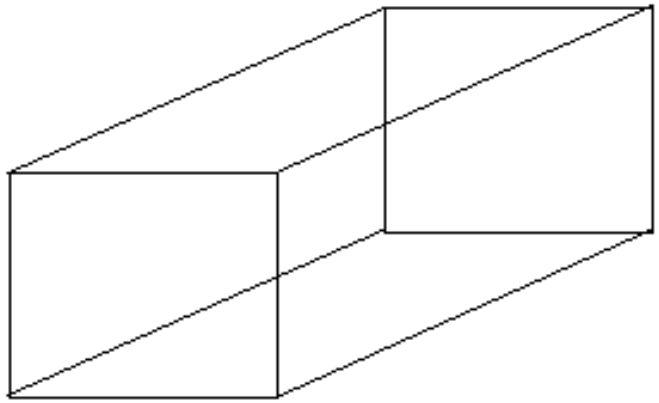


Orthographic projection

Types of projection



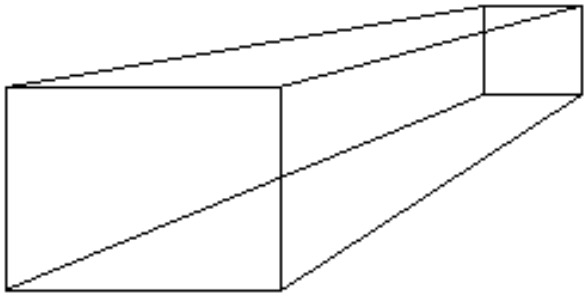
Perspective projection



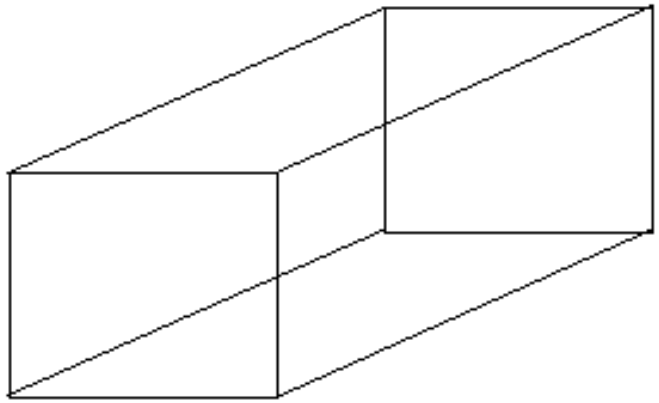
Orthographic projection

- Generally use **perspective** for 3D graphics

Types of projection



Perspective projection



Orthographic projection

- Generally use **perspective** for 3D graphics
- **Orthographic** is useful for 2D or pseudo-2D graphics (e.g. isometric projection for engineering or technical drawings)

FALMOUTH
UNIVERSITY

First person camera control

The plan

The plan

- Represent the player's **position** by a 3D vector

The plan

- Represent the player's **position** by a 3D vector
- Represent the player's **orientation** by Euler angles

The plan

- Represent the player's **position** by a 3D vector
- Represent the player's **orientation** by Euler angles
- Mouse events change these angles

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

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

Euler angles

Euler angles

- Any orientation of an object in 3D space can be described by **three** rotations around:

Euler angles

- Any orientation of an object in 3D space can be described by **three** rotations around:
  - The x-axis $(1, 0, 0)$

Euler angles

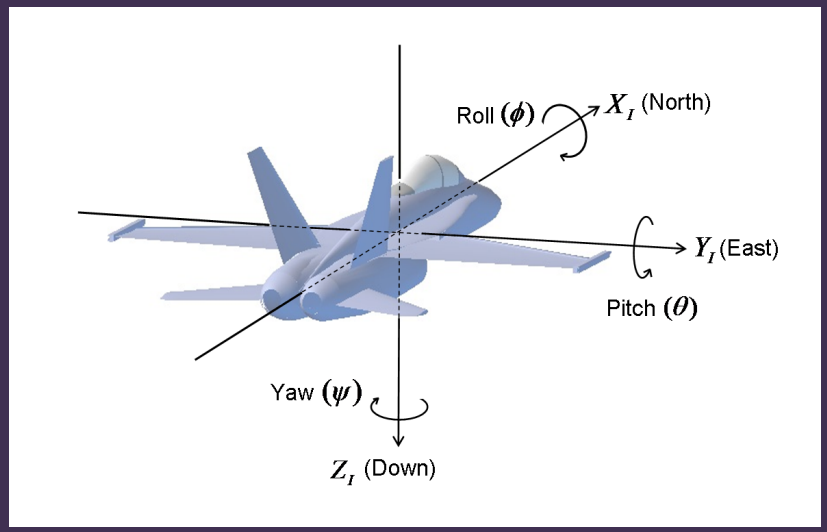
- Any orientation of an object in 3D space can be described by **three** rotations around:
  - The x-axis $(1, 0, 0)$
  - The y-axis $(0, 1, 0)$

Euler angles

- Any orientation of an object in 3D space can be described by **three** rotations around:
  - The x-axis $(1, 0, 0)$
  - The y-axis $(0, 1, 0)$
  - The z-axis $(0, 0, 1)$

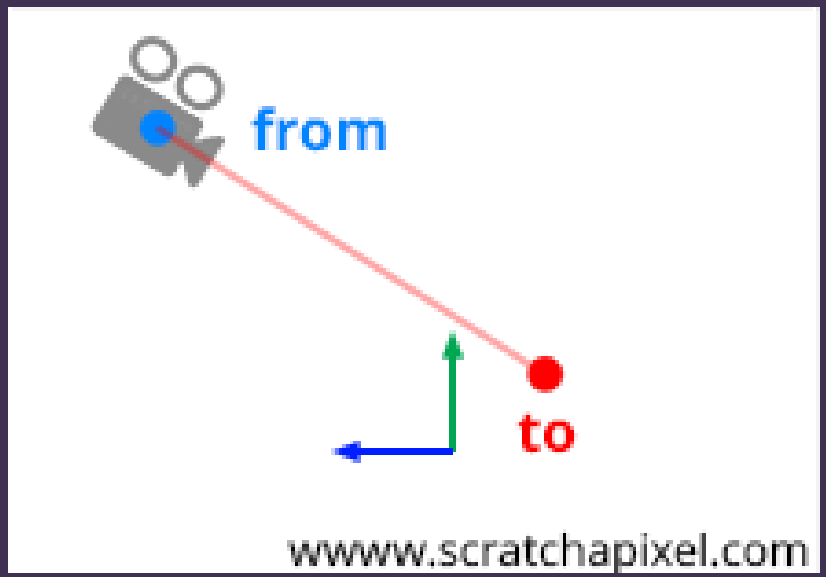
Euler angles

- Any orientation of an object in 3D space can be described by **three** rotations around:
  - The x-axis $(1, 0, 0)$
  - The y-axis $(0, 1, 0)$
  - The z-axis $(0, 0, 1)$
- These angles are sometimes called **roll**, **pitch** and **yaw**

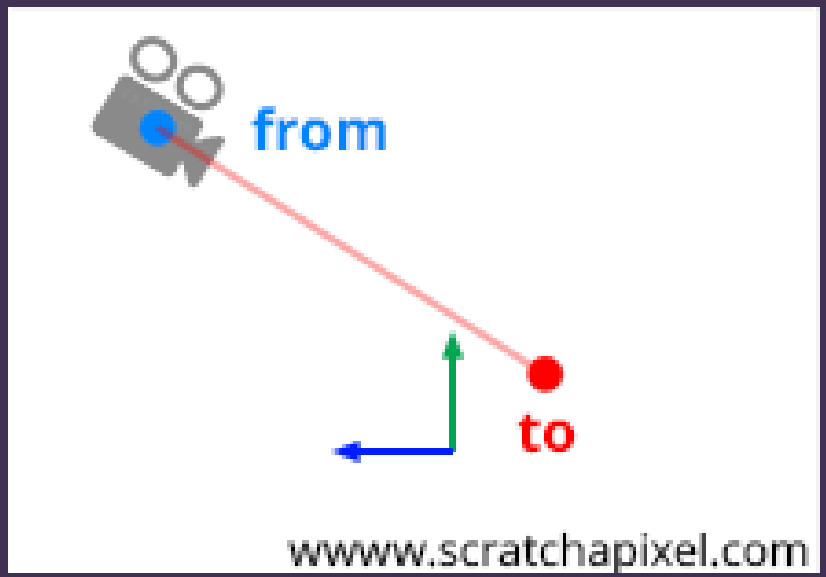


Representing look direction

Representing look direction

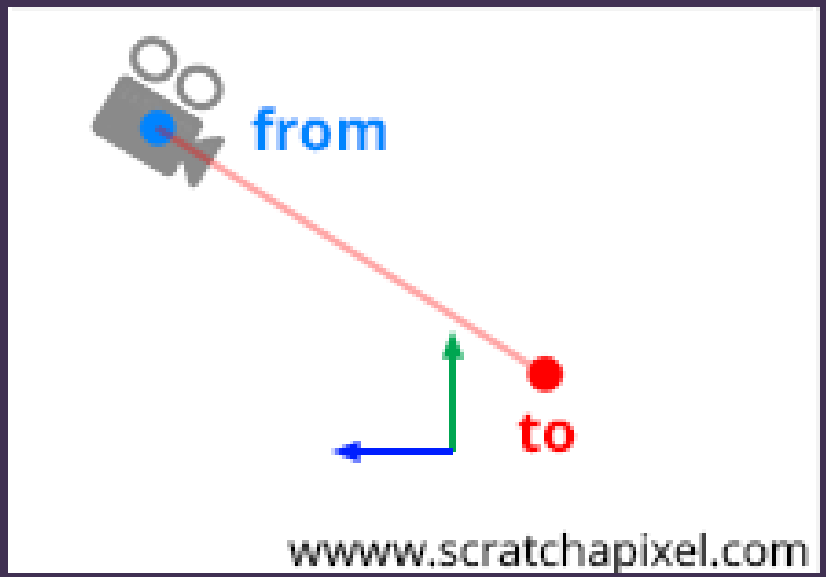


Representing look direction



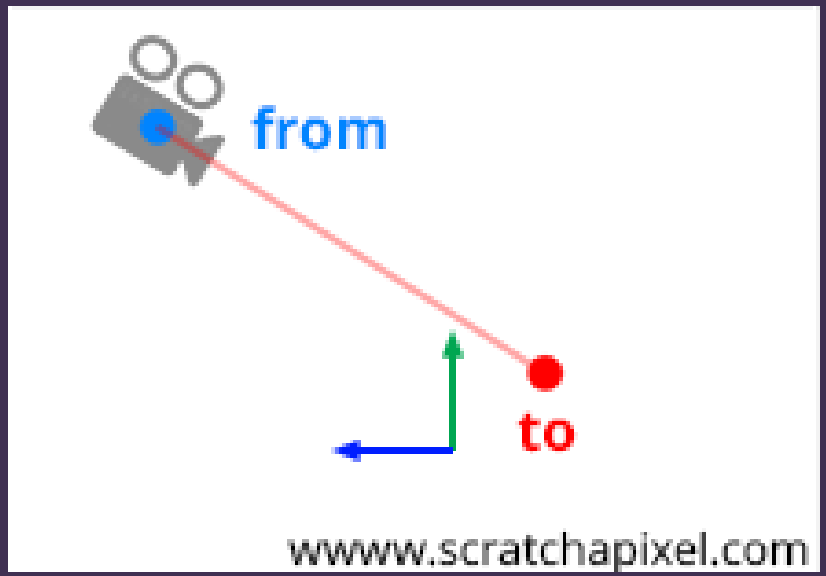
- Don't need roll, just pitch and yaw to look up/down and left/right.

Representing look direction



- Don't need roll, just pitch and yaw to look up/down and left/right.
- Eliminates the gimbal lock problem! :D

Representing look direction



- Don't need roll, just pitch and yaw to look up/down and left/right.
- Eliminates the gimbal lock problem! :D
- **Forward vector** / **look vector** can be obtained by appropriate rotation of a **unit vector**.

Unit vectors

Unit vectors

- A **unit vector** is a vector of length 1

Unit vectors

- A **unit vector** is a vector of length 1
- I.e. $\sqrt{x^2 + y^2 + z^2} = 1$ (Pythagoras)

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**

Unit vectors

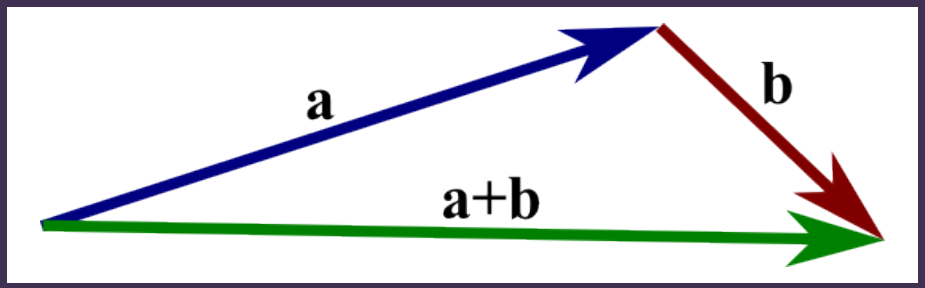
- A **unit vector** is a vector of length 1
- I.e. $\sqrt{x^2 + y^2 + z^2} = 1$ (Pythagoras)
- Useful to represent **direction**
- Multiplying a vector of length a by a number b gives a vector of length $a \times b$, parallel to the original vector

Unit vectors

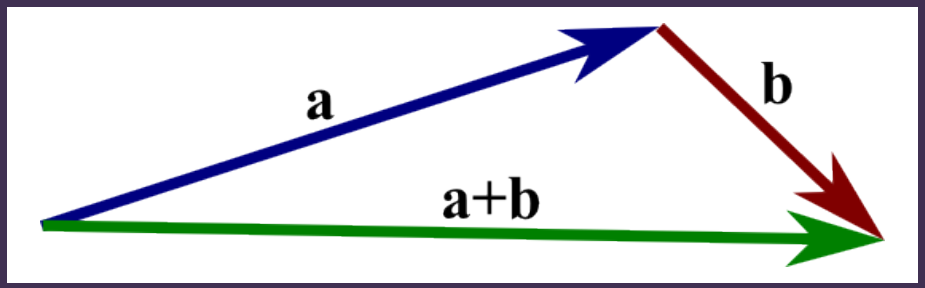
- A **unit vector** is a vector of length 1
- I.e. $\sqrt{x^2 + y^2 + z^2} = 1$ (Pythagoras)
- Useful to represent **direction**
- Multiplying a vector of length a by a number b gives a vector of length $a \times b$, parallel to the original vector
- So multiplying a unit vector by b gives a vector of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

Moving forwards

Moving forwards

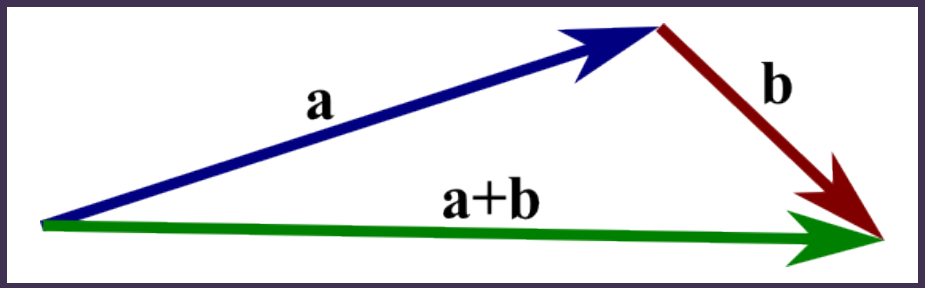


Moving forwards



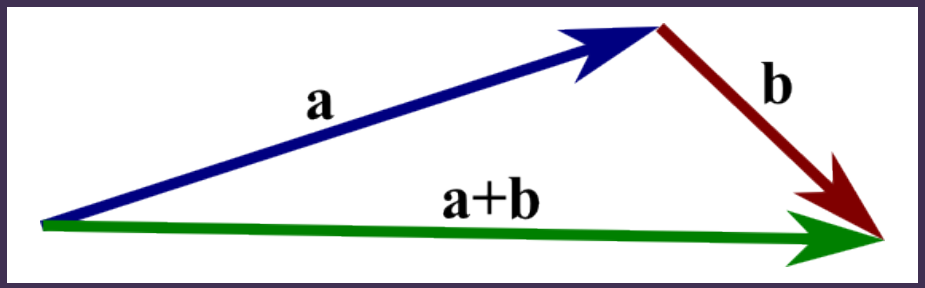
- E.g. if *a* is current position, and

Moving forwards



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

Moving forwards



- 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

Addition in homogeneous coordinates

Addition in homogeneous coordinates

- Recall: homogeneous coordinates have an extra w component, i.e. (x, y, z, w)

Addition in homogeneous coordinates

- Recall: homogeneous coordinates have an extra w component, i.e. (x, y, z, w)
- $w = 1$ for positions, $w = 0$ for offsets

Addition in homogeneous coordinates

- Recall: homogeneous coordinates have an extra w component, i.e. (x, y, z, w)
- $w = 1$ for positions, $w = 0$ for offsets

offset + offset = offset $\quad w = 0 + 0 = 0$

Addition in homogeneous coordinates

- Recall: homogeneous coordinates have an extra w component, i.e. (x, y, z, w)
- $w = 1$ for positions, $w = 0$ for offsets

offset	+	offset	=	offset	$w = 0 + 0 = 0$
position	+	offset	=	position	$w = 1 + 0 = 1$

Addition in homogeneous coordinates

- Recall: homogeneous coordinates have an extra w component, i.e. (x, y, z, w)
- $w = 1$ for positions, $w = 0$ for offsets

offset	+	offset	=	offset	$w = 0 + 0 = 0$
position	+	offset	=	position	$w = 1 + 0 = 1$
position	+	position	=	???	$w = 1 + 1 = 2$

Next steps

- **Review** the additional asynchronous material for more background on representing mesh vertices and calculating the model, view and projection matrices.
- **Attend** the workshop to see how to implement an Element Buffer in OpenGL and use GLM to create and apply transforms.