COMP220: Graphics & Simulation

3: Mathematics for graphics

Learning outcomes

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

- **Explain** the role of vectors and matrices in computer graphics
- **Calculate** basic transformation matrices using the GLM library
- **Explain** the constituents of the model-view-projection matrix

FALMOUTH
UNIVERSITY

Transformations

Transformations and matrices

Transformations and matrices

- A **transformation** is a **mathematical function** that **changes points in space**

Transformations and matrices

- A **transformation** is a **mathematical function** that **changes points in space**
- E.g. shifts them, rotates them, scales them, ...

Transformations and matrices

- A **transformation** is a **mathematical function** that **changes points in space**
- E.g. shifts them, rotates them, scales them, ...
- Many useful transformations can be **represented** by matrices

Transformations and matrices

- A **transformation** is a **mathematical function** that **changes points in space**
- E.g. shifts them, rotates them, scales them, ...
- Many useful transformations can be **represented** by matrices
- Multiplying these matrices together **combines** the transformations

Transformations and matrices

- A **transformation** is a **mathematical function** that **changes points in space**
- E.g. shifts them, rotates them, scales them, ...
- Many useful transformations can be **represented** by matrices
- Multiplying these matrices together **combines** the transformations
- Multiplying a vector by the matrix **applies** the transformation

GLM

GLM

- We will use the **GLM** library to do matrix calculations for us

GLM

- We will use the **GLM** library to do matrix calculations for us
- `http://glm.g-truc.net/`

GLM

- We will use the **GLM** library to do matrix calculations for us
- `http://glm.g-truc.net/`
- GLM aims to mirror GLSL data types (`vec4`, `mat4` etc) in C++

GLM

- We will use the **GLM** library to do matrix calculations for us
- `http://glm.g-truc.net/`
- GLM aims to mirror GLSL data types (`vec4`, `mat4` etc) in C++
- Lets us perform calculations with vectors and matrices in C++

GLM

- We will use the **GLM** library to do matrix calculations for us
- `http://glm.g-truc.net/`
- GLM aims to mirror GLSL data types (`vec4`, `mat4` etc) in C++
- Lets us perform calculations with vectors and matrices in C++
- GLM types can be passed into shaders as uniforms, e.g.

```
// transformLocation points to a uniform of type mat4
glm::mat4 transform = ...;
glUniformMatrix4fv(transformLocation, 1, GL_FALSE, glm::value_ptr(transform));
```

Identity

Identity

The identity transformation does not change anything

Identity

The identity transformation does not change anything

```
// Default constructor for glm::mat4 creates an identity matrix
glm::mat4 transform;
```

Translation

Translation

Translation shifts all points by the same vector offset

Translation

Translation shifts all points by the same vector offset

```
transform = glm::translate(transform, glm::vec3(0.3f,
    0.5f, 0.0f));
```

Scaling

Scaling

Scaling moves all points closer or further from the origin by the same factor

Scaling

Scaling moves all points closer or further from the origin by the same factor

```
transform = glm::scale(transform, glm::vec3(1.2f, 0.5f ↩
    , 1.0f));
```

Rotation

Rotation

- How do we represent a rotation in 3 dimensions?

Rotation

- How do we represent a rotation in 3 dimensions?
- One way is by specifying the **axis** (as a vector) and the **angle** (in radians)

Rotation

- How do we represent a rotation in 3 dimensions?
- One way is by specifying the **axis** (as a vector) and the **angle** (in radians)
- Axis always runs through the origin

Rotation

- How do we represent a rotation in 3 dimensions?
- One way is by specifying the **axis** (as a vector) and the **angle** (in radians)
- Axis always runs through the origin

```
float angle = glm::pi<float>() * 0.5f;
glm::vec3 axis(0, 0, 1);
transform = glm::rotate(transform, angle, axis);
```

Combining transformations

Combining transformations

```
transform = glm::translate(transform, glm::vec3(0.5f,
    0.5f, 0.0f));
transform = glm::rotate(transform, angle, axis);
```

Combining transformations

```
transform = glm::translate(transform, glm::vec3(0.5f,
    0.5f, 0.0f));
transform = glm::rotate(transform, angle, axis);
```

- Transformations **do not commute** in general — changing the order will change the result

Combining transformations

```
transform = glm::translate(transform, glm::vec3(0.5f,
    0.5f, 0.0f));
transform = glm::rotate(transform, angle, axis);
```

- Transformations **do not commute** in general — changing the order will change the result
- The order they are applied is the **reverse** of what you might think — i.e. the above rotates **then** translates

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

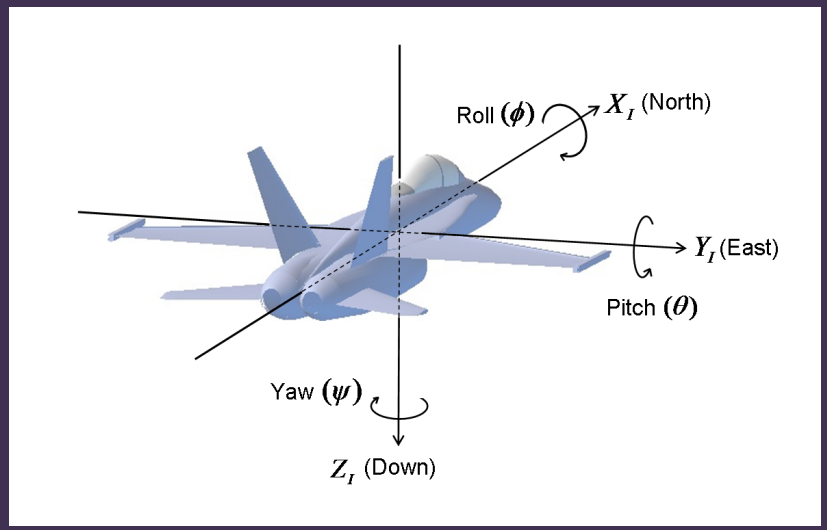
- Any orientation of an object in 3D space can be described by **three** rotations around:
  - The x-axis $(1, 0, 0)$
  - The y-axis $(0, 1, 0)$

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**



Gimbal lock

https://youtu.be/rrUCBOlJdt4?t=1m55s

FALMOUTH
UNIVERSITY

Model, View, Projection

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**:

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

(remember, multiplication goes in reverse order)

The model matrix

The model matrix

Exactly what we've been doing so far today...

The view matrix

The view matrix

Need to translate and rotate the scene so that the "camera" is at $(0, 0, 0)$ and looking in the negative z direction

The view matrix

Need to translate and rotate the scene so that the "camera" is at $(0, 0, 0)$ and looking in the negative z direction

```
glm::mat4 view = glm::lookAt(
  glm::vec3(2, 0, 2),     // eye
  glm::vec3(0, 0, 0),     // centre
  glm::vec3 up(0, 1, 0)   // up
);
```

The view matrix

Need to translate and rotate the scene so that the "camera" is at $(0, 0, 0)$ and looking in the negative z direction

```
glm::mat4 view = glm::lookAt(
  glm::vec3(2, 0, 2),     // eye
  glm::vec3(0, 0, 0),     // centre
  glm::vec3 up(0, 1, 0)   // up
);
```

- `eye` is the position of the camera

The view matrix

Need to translate and rotate the scene so that the "camera" is at $(0, 0, 0)$ and looking in the negative z direction

```
glm::mat4 view = glm::lookAt(
  glm::vec3(2, 0, 2),     // eye
  glm::vec3(0, 0, 0),     // centre
  glm::vec3 up(0, 1, 0)   // up
);
```

- `eye` is the position of the camera
- `centre` is a point for the camera to look at

The view matrix

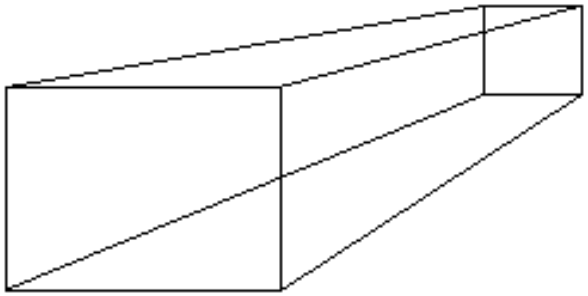
Need to translate and rotate the scene so that the "camera" is at $(0, 0, 0)$ and looking in the negative z direction

```
glm::mat4 view = glm::lookAt(
  glm::vec3(2, 0, 2),     // eye
  glm::vec3(0, 0, 0),     // centre
  glm::vec3 up(0, 1, 0)   // up
);
```

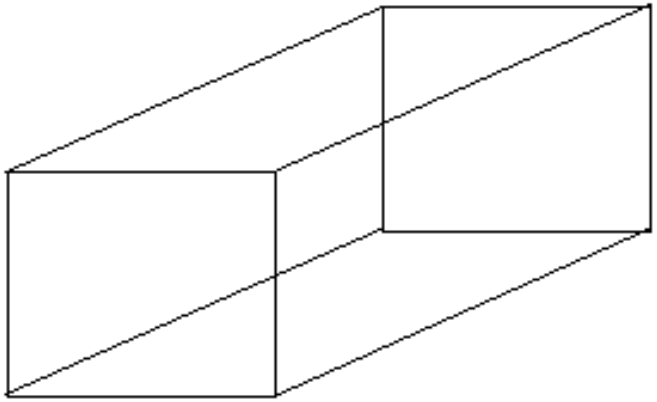
- `eye` is the position of the camera
- `centre` is a point for the camera to look at
- `up` is which direction is "up" for the camera (usually the positive y-axis)

Types of projection

Types of projection

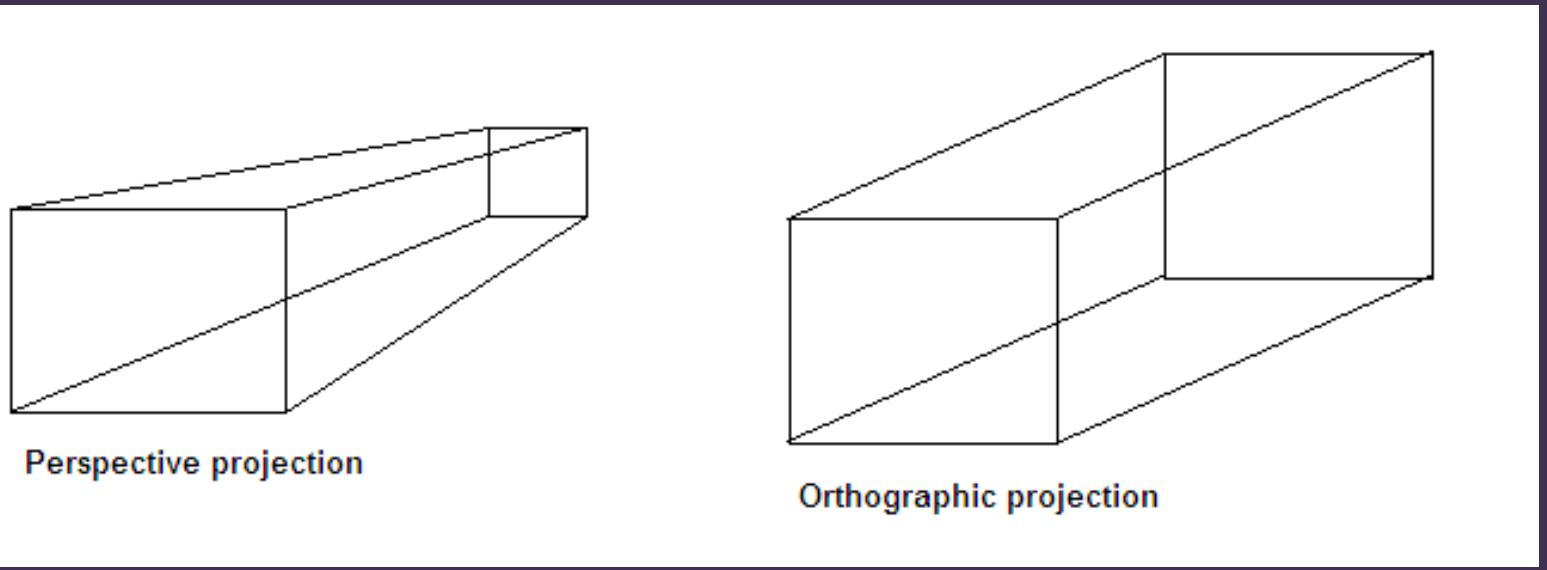


Perspective projection



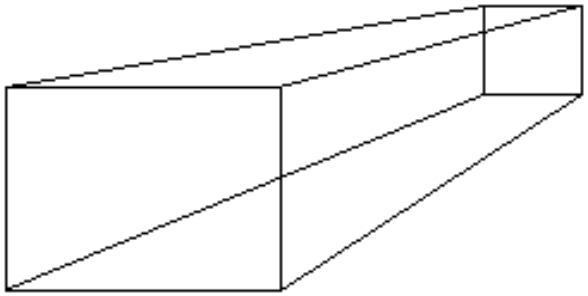
Orthographic projection

Types of projection

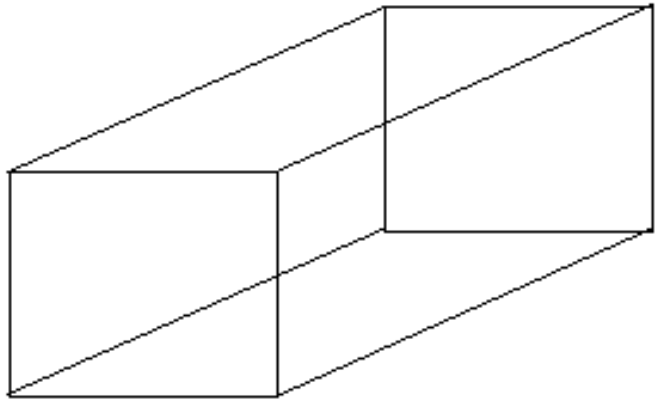


- Generally use **perspective** for 3D graphics

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 perspective)

The projection matrix

The projection matrix

```
glm::mat4 projection = glm::perspective(
    glm::radians(45.0f), // field of view
    4.0f / 3.0f,         // aspect ratio
    0.1f,                // near clip plane
    100.0f               // far clip plane
);
```

The projection matrix

```
glm::mat4 projection = glm::perspective(
    glm::radians(45.0f), // field of view
    4.0f / 3.0f,         // aspect ratio
    0.1f,                // near clip plane
    100.0f               // far clip plane
);
```

- **Field of view (FOV)**: how "wide" or "narrow" the view is

The projection matrix

```
glm::mat4 projection = glm::perspective(
    glm::radians(45.0f), // field of view
    4.0f / 3.0f,         // aspect ratio
    0.1f,                // near clip plane
    100.0f               // far clip plane
);
```

- **Field of view (FOV)**: how "wide" or "narrow" the view is
- **Aspect ratio**: should be `screenWidth / screenHeight`

The projection matrix

```
glm::mat4 projection = glm::perspective(
    glm::radians(45.0f), // field of view
    4.0f / 3.0f,         // aspect ratio
    0.1f,                // near clip plane
    100.0f               // far clip plane
);
```

- **Field of view (FOV)**: how "wide" or "narrow" the view is
- **Aspect ratio**: should be `screenWidth / screenHeight`
- **Near and far clip planes**: fragments that fall outside this range of distances from the camera are not drawn

The projection matrix

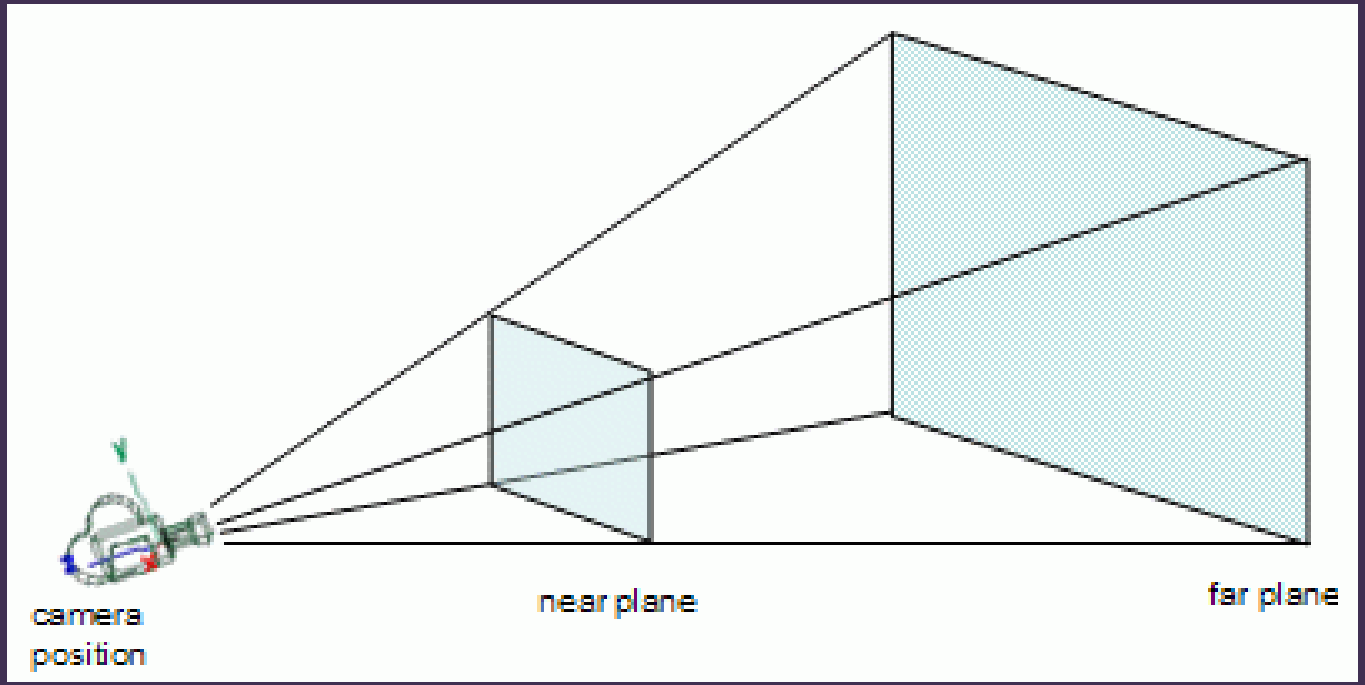
```
glm::mat4 projection = glm::perspective(
    glm::radians(45.0f), // field of view
    4.0f / 3.0f,         // aspect ratio
    0.1f,                // near clip plane
    100.0f               // far clip plane
);
```

- **Field of view (FOV)**: how "wide" or "narrow" the view is
- **Aspect ratio**: should be `screenWidth / screenHeight`
- **Near and far clip planes**: fragments that fall outside this range of distances from the camera are not drawn

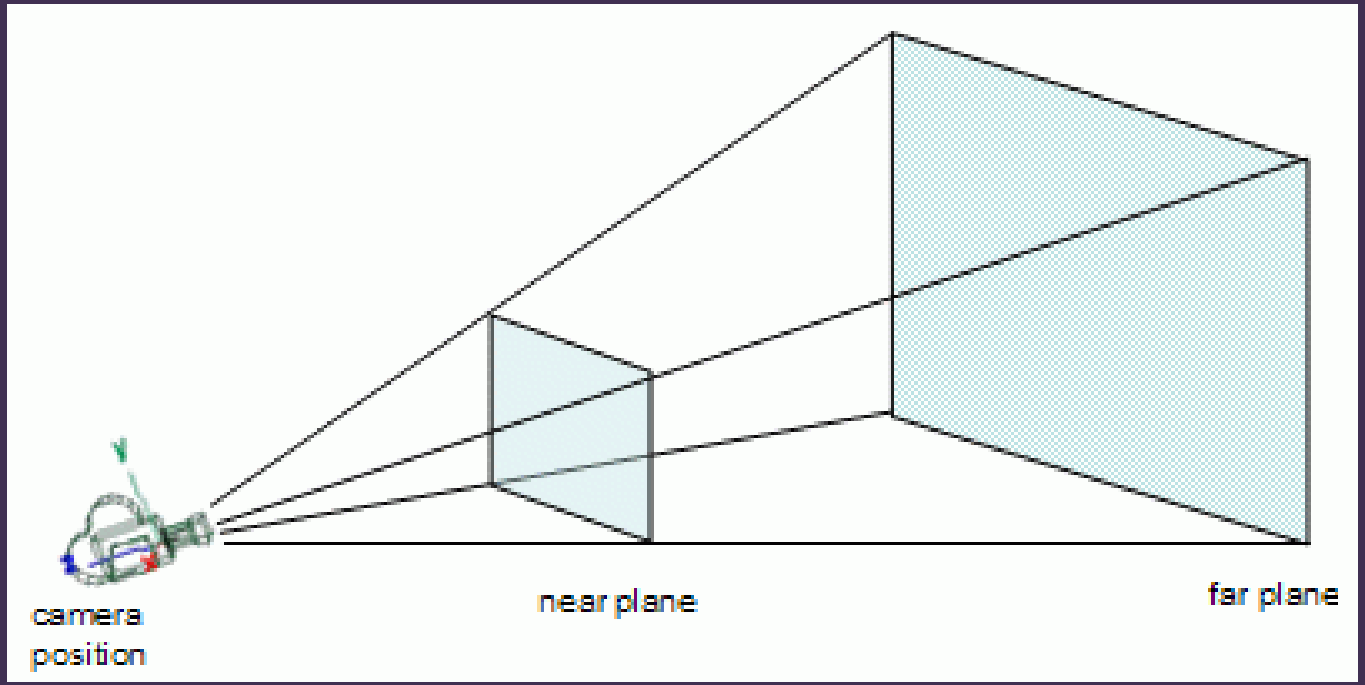
Also available: `glm::ortho` for orthographic projection

The view frustum

The view frustum

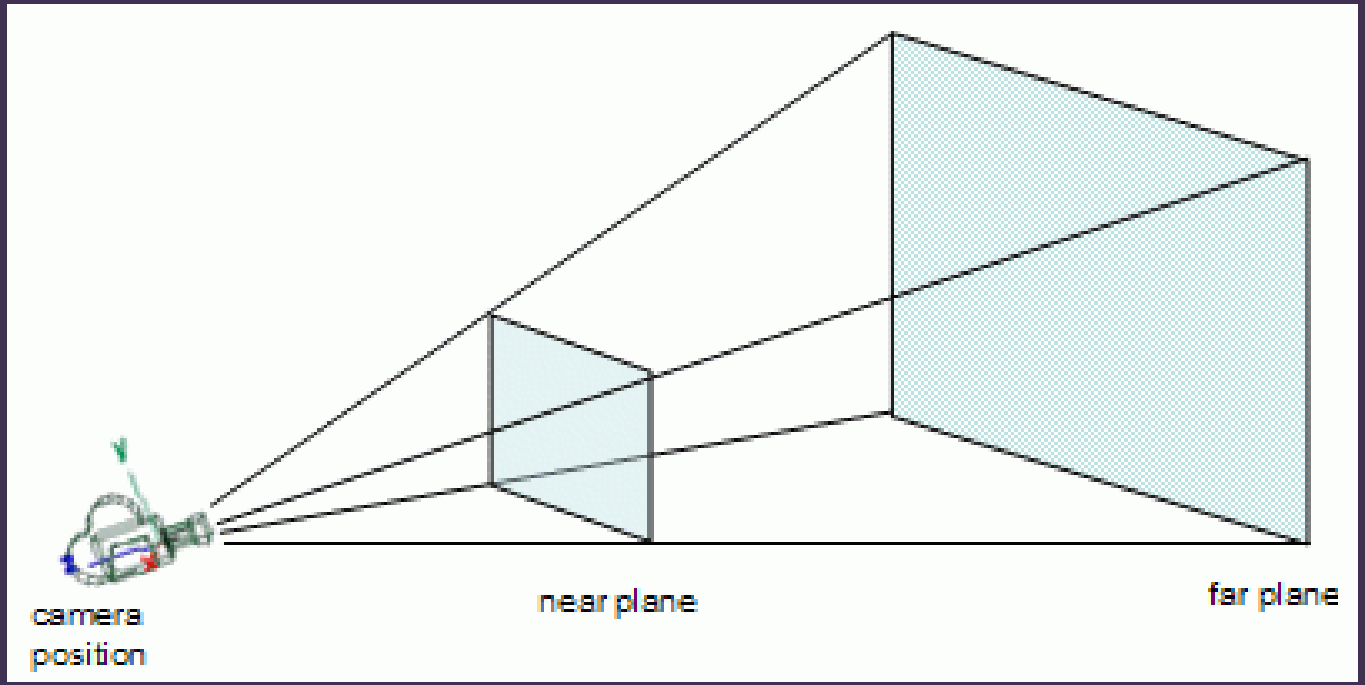


The view frustum



- 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

Live Coding - Transformations

Exercise

- Using the following link, first implement model transformations and then view and projection
  - `http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/`
- Play around with the order of the Model Transformation, what happens when you do this?
- Use the keyboard to move, rotate and scale the triangle
- Use the Keyboard to move the camera
- **Stretch Goal:** Create an FPS camera using the mouse and keyboard