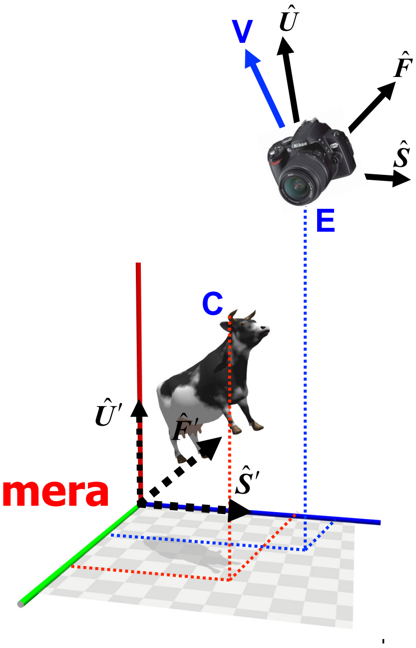
MANCHESTER 1824

COMP27112

Computer Graphics and Image Processing

5: Viewing 1: The camera

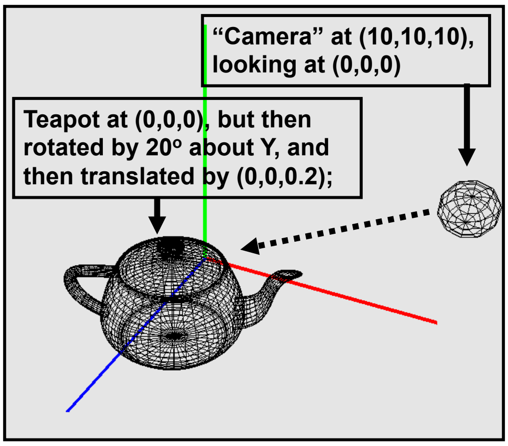


Toby.Howard@manchester.ac.uk

MANCHESTER 1824

Introduction

- In part 1 of our study of Viewing, we'll look at
  - Viewing in 2D
  - Clipping
  - Viewing in 3D
  - Viewing in OpenGL

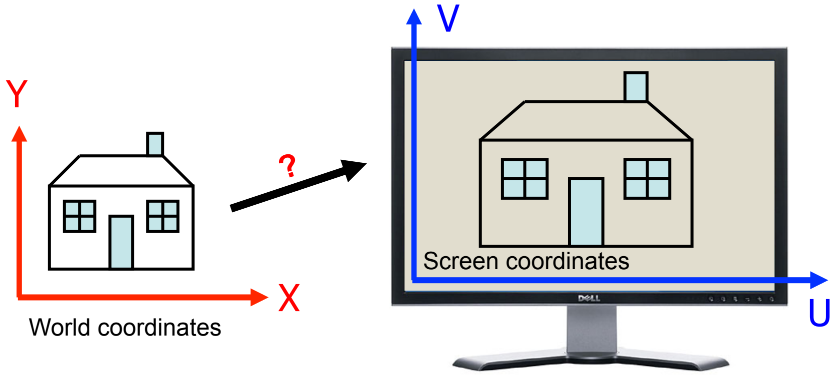


2

MANCHESTER
1824

Viewing in 2D

- We'll start with looking at viewing in 2D



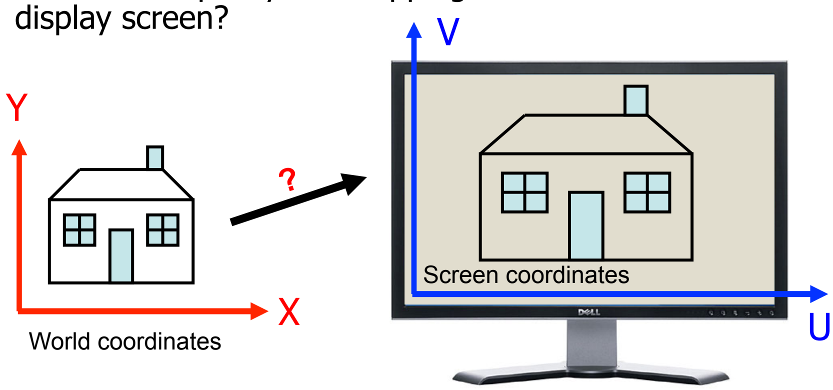
- We need to specify **what** we want to see and **where** we want to see it

3

MANCHESTER
1824

Viewing in 2D

- How do we specify the mapping from our scene to the display screen?



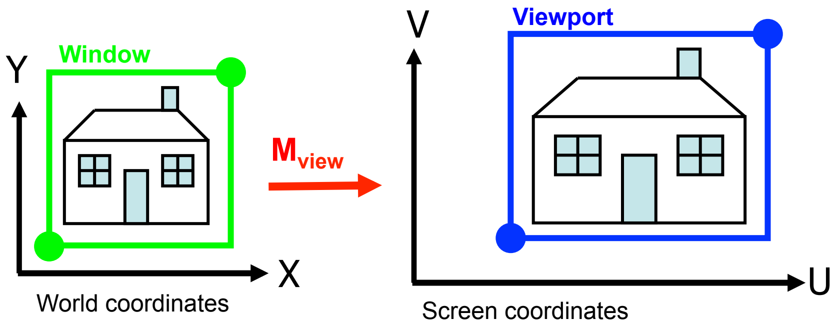
- We use the **analogy** of photographing the scene with a camera
- In this case the camera is confined to the 2D plane

4

MANCHESTER 1824

Viewing in 2D

- We specify a "window" in world coordinates, and a "viewport" in screen coordinates



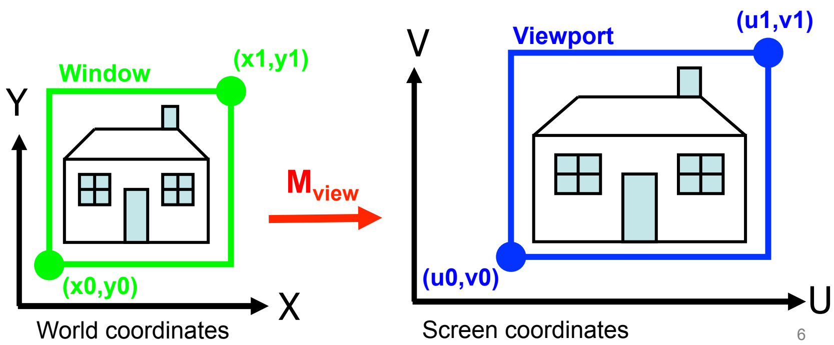
- We find the matrix $\mathbf{M}_{view}$ which transforms the window to the viewport.
- $\mathbf{M}_{view}$ is called the **viewing transformation**

5

MANCHESTER 1824

Window to viewport mapping

- We can find $\mathbf{M}_{view}$ easily, in 3 steps
  - **M1** : Translate by (-x0, -y0) to place the window at the origin
  - **M2** : Scale the window to be the same shape as the viewport
  - **M3** : Shift to the viewport position

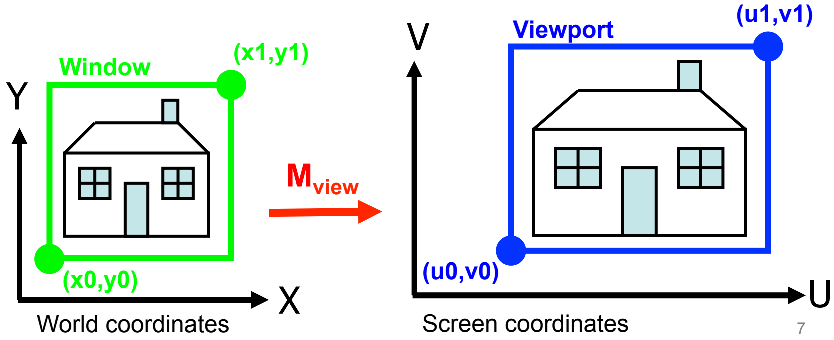


6

MANCHESTER 1824

Window to viewport mapping

- **M1** : Translate (-x0, -y0)
- **M2** : Scale (u1-u0/x1-x0, v1-v0/y1-y0)
- **M3** : Translate (u0, v0)
- $\mathbf{M_{view}} = \mathbf{M3} * \mathbf{M2} * \mathbf{M1}$
- $\mathbf{P_{screen}} = \mathbf{M_{view}} * \mathbf{P_{world}}$



7

MANCHESTER 1824

Window to viewport mapping

- $\mathbf{M_{view}} = \mathbf{M3} * \mathbf{M2} * \mathbf{M1}$

$$\underset{\mathbf{M_3}}{\begin{bmatrix} 1 & 0 & u_0 \\ 0 & 1 & v_0 \\ 0 & 0 & 1 \end{bmatrix}} * \underset{\mathbf{M_2}}{\begin{bmatrix} \frac{u_1-u_0}{x_1-x_0} & 0 & 0 \\ 0 & \frac{v_1-v_0}{y_1-y_0} & 0 \\ 0 & 0 & 1 \end{bmatrix}} * \underset{\mathbf{M_1}}{\begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix}}$$

- So we can now map a world space point $\mathbf{P_{world}}$ into screen coordinates $\mathbf{P_{screen}}$ as follows. This is our "view".

$$\mathbf{P_{screen}} = \underset{\mathbf{M_{view}}}{\begin{bmatrix} \frac{u_1-u_0}{x_1-x_0} & 0 & -x_0 * \frac{u_1-u_0}{x_1-x_0} + u_0 \\ 0 & \frac{v_1-v_0}{y_1-y_0} & -y_0 * \frac{v_1-v_0}{y_1-y_0} + v_0 \\ 0 & 0 & 1 \end{bmatrix}} * \mathbf{P_{world}}$$

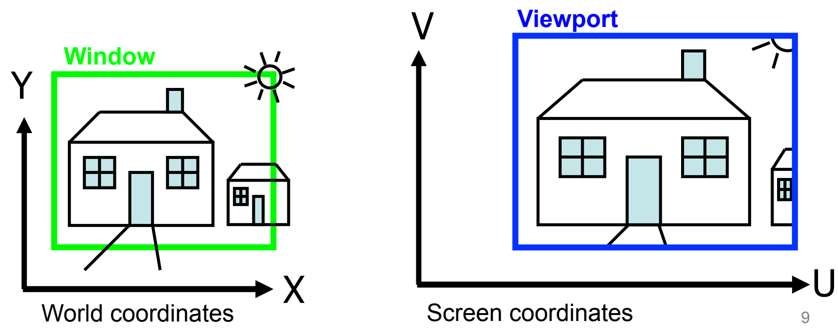
Note: We rarely multiply matrices by hand like this. The graphics system will multiply all matrices together for us.

8

MANCHESTER 1824

Clipping

- Normally we will want to **CLIP** against the viewport
  - …to remove those parts of primitives whose coordinates are outside the window
  - There are standard algorithms for clipping lines and polygons

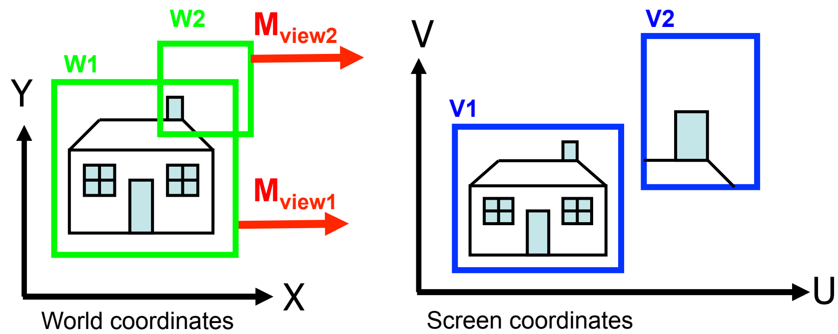


9

MANCHESTER 1824

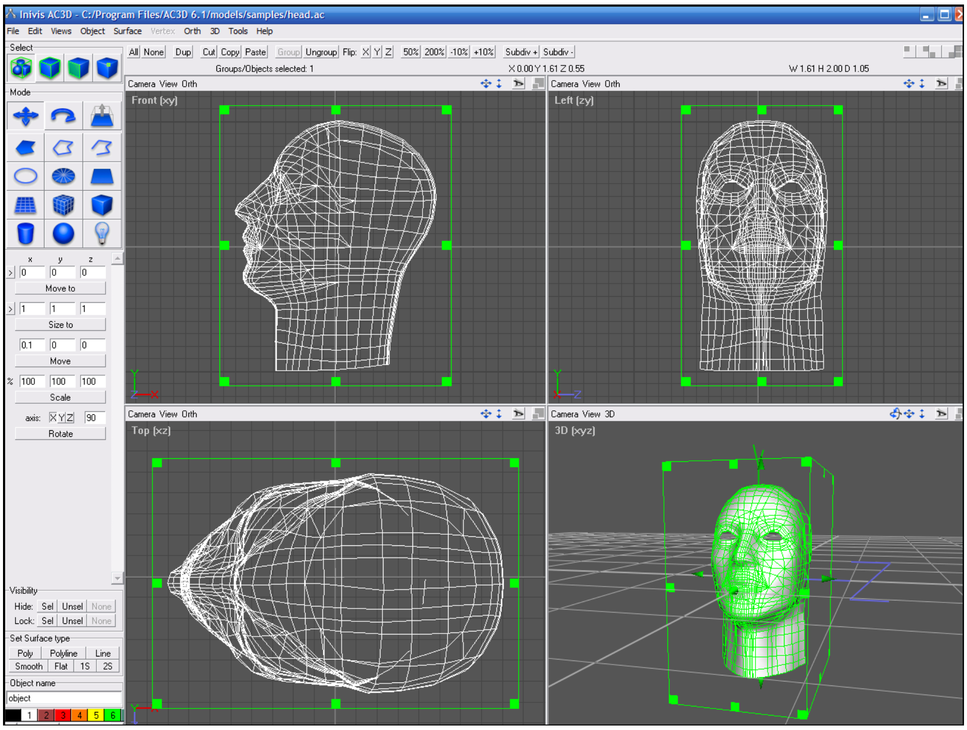
Multiple windows and viewports

- Sometimes it's useful to use multiple windows and viewports, to help arrange items on the screen



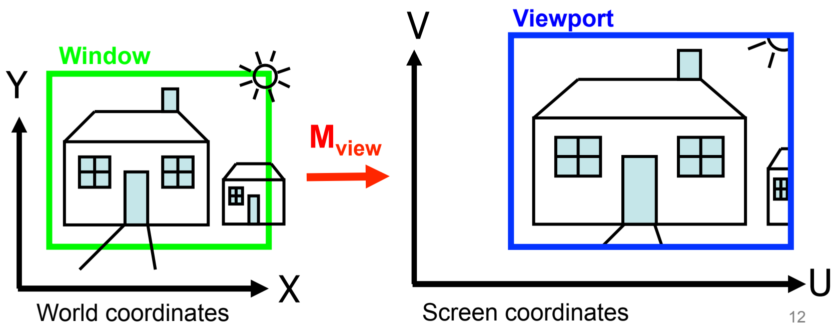
- Real example: the 4 different views in AC3D

10



Viewing in 2D: summary

- We use the analogy of photographing our scene with a 2D "camera" which can slide in the XY plane
- We compute a **viewing transformation** $\mathbf{M}_{\mathbf{view}}$
- $\mathbf{P}_{\mathbf{screen}} = \mathbf{M}_{\mathbf{view}} * \mathbf{P}_{\mathbf{world}}$



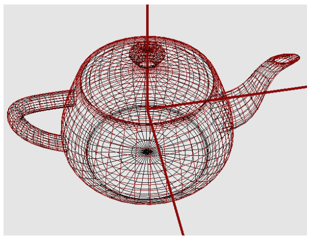
12

MANCHESTER 1824

Viewing in 3D

- In 2D graphics, we "view" our world by mapping from 2D world coordinates to 2D screen coordinates: easy and obvious
- In 3D graphics, in order to "view" our 3D world, we have to somehow **reduce** our 3D information to 2D information, so that it can be displayed on the 2D display: **not** so easy and not obvious

- Here we see a 2D view of an object defined in 3D. It's been projected from 3D to 2D.
- To specify how this view is created, we again use the analogy of "taking a picture using a camera", but this time our "camera" is like a real-world camera: It has a position and orientation in 3D space, and a particular type of lens.



13

MANCHESTER 1824

The camera analogy

- The process of transforming a synthetic 3D model into a 2D view is analogous to using a camera in the real world to take 2D pictures of a 3D scene

14

MANCHESTER
1824

The camera analogy

Step 1: Arrange the scene into the desired composition

15

MANCHESTER
1824

The camera analogy

Step 2: Position and point the camera at the scene

16

MANCHESTER
1824

The camera analogy

Step 3: Choose a camera lens (wide-angle? zoom?)

17

MANCHESTER
1824

The camera analogy

Step 4: Decide the size of the final photograph

18

MANCHESTER
1824

3D Viewing: the camera analogy

	Real World	Computer Graphics
1	Arrange the scene into the desired composition	Set Modelling Transformation
2	Point the camera at the scene	Set Viewing Transformation
3	Choose the camera lens, or adjust the zoom	Set Projection Transformation
4	Determine the size and shape of the final photograph	Set Viewport Transformation

19

MANCHESTER
1824

3D Viewing: the camera analogy

	Real World	Computer Graphics
1	Arrange the scene into the desired composition	Set Modelling Transformation
2	Point the camera at the scene	Set Viewing Transformation
3	Choose the camera lens, or adjust the zoom	Set Projection Transformation
4	Determine the size and shape of the final photograph	Set Viewport Transformation

20

MANCHESTER 1824

The 3D Viewing Pipeline

3D vertex

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

→ **1** Modelling transformation **M** → **2** Viewing transformation **V** → **3** Projection transformation **P** →

Note: In OpenGL, **M** and **V** are combined into a single "modelview" matrix

→ **4** Clip to view volume → **5** Perspective division → **6** Viewport transformation → 2D pixel px,py

In this lecture we'll cover steps 1 and 2, and look at steps 3 to 6 in the next lecture.

21

MANCHESTER 1824

The duality of Modelling and Viewing

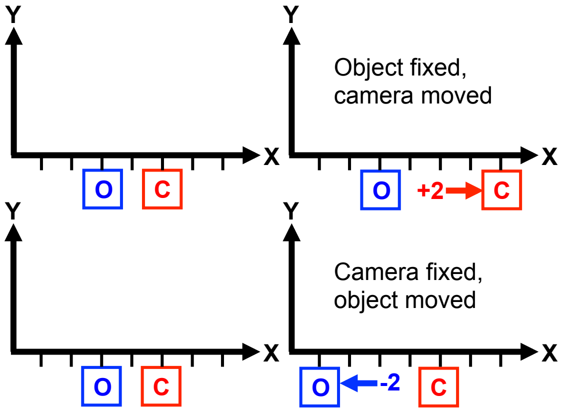
- Question: to obtain these two images of the model, was the camera moved **or** was the model moved?
- Answer: it is impossible to tell, just by looking at the images. It could have been either method.
- Example: moving the model by (x,y,z) is equivalent to moving the camera by (-x,-y,-z). This also applies to rotations.

22

MANCHESTER 1824

The duality of Modelling and Viewing

- Here, in 2D, we have an object **O** and a camera **C**, both sitting on the X axis
- If we keep the object fixed and move the camera by **+2, O** and **C** are now **4** units apart.
- If we keep the camera fixed and move the object by **-2, O** and **C** are now **4** units apart.



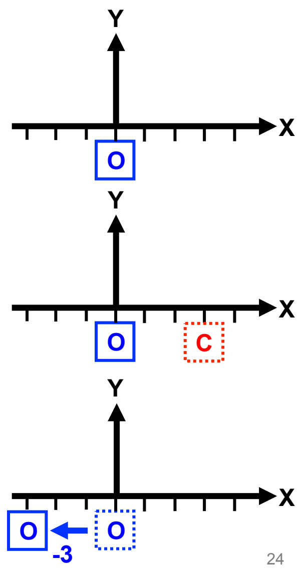
- Whether we move **O** or **C**, their **relative positions** will be the same, so the view from the camera will be the same

23

MANCHESTER 1824

The duality of Modelling and Viewing

- Now for the **KEY IDEA**, which we'll present in 2D
- Imagine we have an object **O** sitting at the origin
- We want to view **O** with a camera **C** located at position **X=3**
- But we don't actually have a camera!
- But we can **SIMULATE** the effect, by instead moving the object by **-3 in X**



24

MANCHESTER 1824

Achieving viewing by modelling

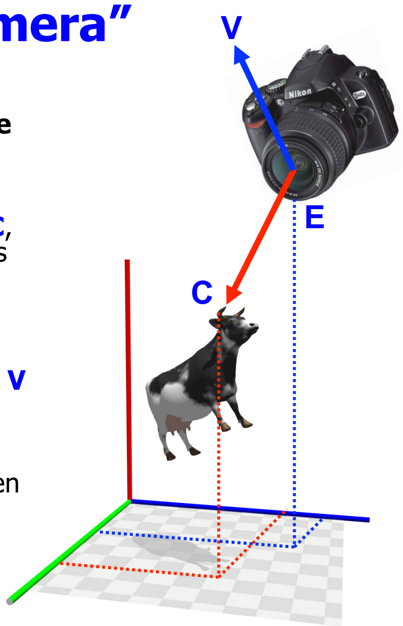
- We've just seen that we can create the same view from a camera at a certain location and orientation, by **instead** transforming the object
- This is exactly what we do in computer graphics
- However, the idea of having a camera is very natural to us, so we **pretend** we really have a camera…
- …and we express the view we want in terms of "camera location and orientation", but to implement this we actually compute a suitable viewing transformation which we **apply to the object**

25

MANCHESTER 1824

Specifying the "camera"

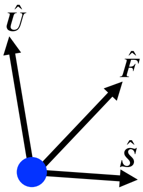
- We specify where the "camera" is located in 3D space. This is the **eye point, E**
- We specify a **centre of interest C**, a 3D point at which the "camera" is looking
- We specify the **up** direction of the "camera", using a "view up vector" **V**
- We can then use **E**,**C**, and **V** to derive a transformation which, when applied to the **model**, would give the same view as if we really had this "camera".

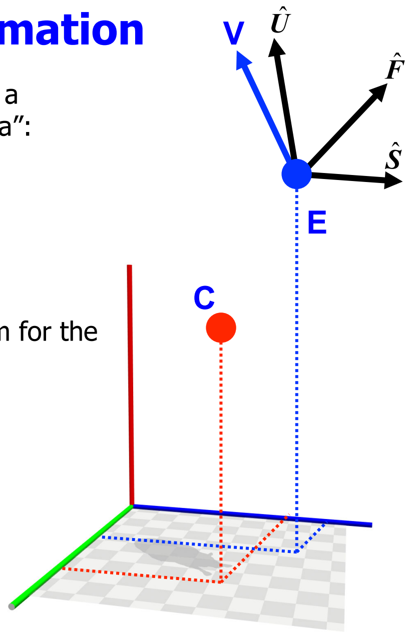


MANCHESTER 1824

The viewing transformation

- First we use **E**, **C** and **V** to derive a coordinate system for the "camera":
  1. $F = C - E$, then normalise F
  2. Normalise V (up vector)
  3. $\hat{S} = \hat{F} \times \hat{V}$ (cross product)
  4. $\hat{U} = \hat{S} \times \hat{F}$ (cross product)
- We now have a coordinate system for the "camera": with axes $\hat{S}\,\hat{U}\,\hat{F}$





MANCHESTER 1824

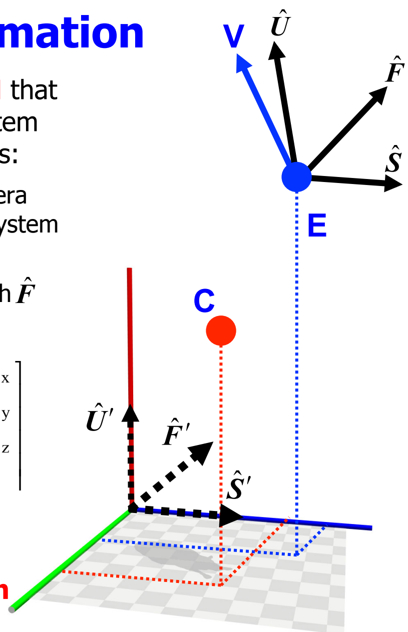
The viewing transformation

- Next we derive a transformation **M** that maps the "camera" coordinate system into world coordinates, in two steps:
  1. We translate the origin of the camera system to the origin of the world system
  2. We rotate the camera axes to be coincident with the world axes, with $\hat{F}$ aligned with **-Z**

$$\mathbf{M} = \underbrace{\begin{bmatrix} \hat{S}_x & \hat{S}_y & \hat{S}_z & 0 \\ \hat{U}_x & \hat{U}_y & \hat{U}_z & 0 \\ -\hat{F}_x & -\hat{F}_y & -\hat{F}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{rotation}} * \underbrace{\begin{bmatrix} 1 & 0 & 0 & -E_x \\ 0 & 1 & 0 & -E_y \\ 0 & 0 & 1 & -E_z \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{translation}}$$

(we omit the derivation)

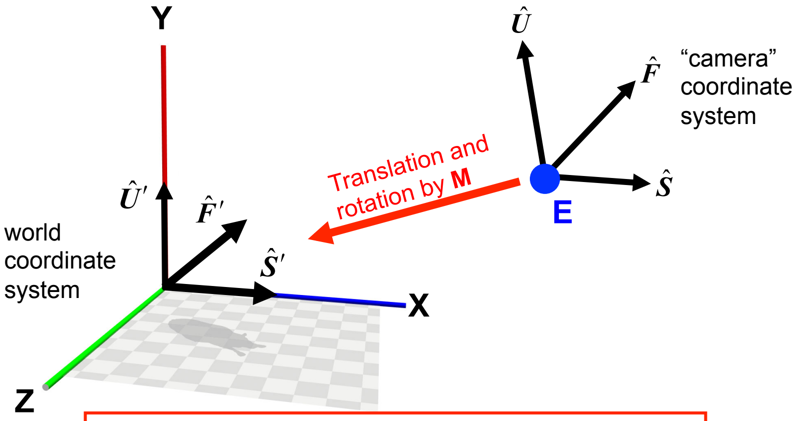
- **M** is the **viewing transformation**



MANCHESTER 1824

The viewing transformation: summary

- **The viewing transformation M** maps the $\hat{S}\,\hat{U}\,\hat{F}$ "camera" axes system to the world coordinates axes system:



- **KEY IDEA:** If we apply **M** to **objects**, we will get the **same view** as if we had a real camera

29

MANCHESTER 1824

The viewing transformation in OpenGL

- This OpenGL function takes (**E**,**C**,**V**) and computes the viewing transformation we have just seen

```
void gluLookAt (GLdouble eyex,                OpenGL
               GLdouble eyey,
               GLdouble eyez,
               GLdouble centrex,
               GLdouble centrey,
               GLdouble centrez,
               GLdouble upx,
               Gldouble upy,
               Gldouble upz);
```

- As we have seen with other OpenGL transformation functions, **`gluLookAt()`** creates a temporary matrix **T**, and then multiplies the modelview matrix by **T**: $\mathbf{M}_{\mathbf{modelview}} = \mathbf{M}_{\mathbf{modelview}} * \mathbf{T}$

30

MANCHESTER 1824

The viewing transformation in OpenGL

- The viewing transformation specifies the location and orientation of the "camera" (by in fact transforming the model)
- We incorporate this transformation into the modelview matrix as follows:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); // M= identity matrix (I)
gluLookAt(…stuff…) // M is now I * VIEW
```

- Because we want the viewing transformation to take place **AFTER** any true modelling transformations, we need to "pre-load" the modelview matrix with the viewing transformation...
- ...And then all subsequent modelling transformations will get multiplied into the modelview matrix

31

MANCHESTER 1824

Modelling and viewing together

```
// First set the viewing transformation
glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); // M= identity matrix (I)
gluLookAt(…stuff…) // M is now I * VIEW

// Now draw a transformed teapot
glTranslatef(tx, ty, tz);
// OpenGL computes temp translation matrix T,
// then sets M= M x T, so now M is (VIEW x T)
glRotatef(theta, 0.0, 1.0, 0.0);
// OpenGL computes temp rotation matrix R,
// then sets M= M x R, so M is now (VIEW x T x R)
glutWireTeapot(1.0);
```

- So all points P will be transformed first by **R**, then **T**, then **VIEW**

32

MANCHESTER
1824

Example: Setting a view in OpenGL

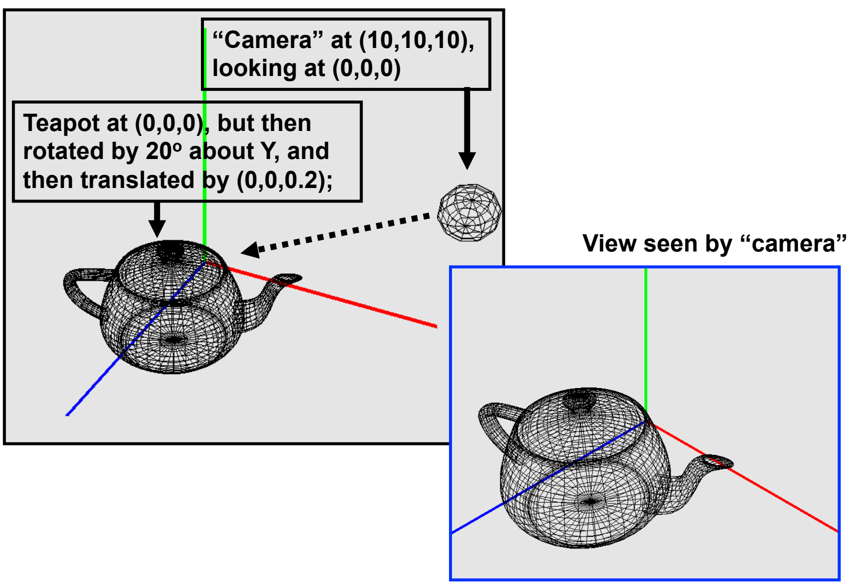
- Here's a real fragment showing the use of a view transformation and a modelling transformation together
- Note that we also need to set the **projection**, but we'll cover that in the next lecture, so ignore it for now.

```
glMatrixMode(GL_MODELVIEW); // select modelview matrix

glLoadIdentity(); // initialise it

// set the projection (see next lecture)
gluPerspective(…stuff…);
// set the view transformation
gluLookAt(10,10,10, 0,0,0, 0,1,0);

// move/rotate the model however we want
glTranslatef(0.0, 0.0, 0.2);
glRotatef(20.0, 0.0, 1.0, 0.0);

glutWireTeapot(3.0); // draw it
```

See the next slide for a visualisation of this.

33

MANCHESTER
1824

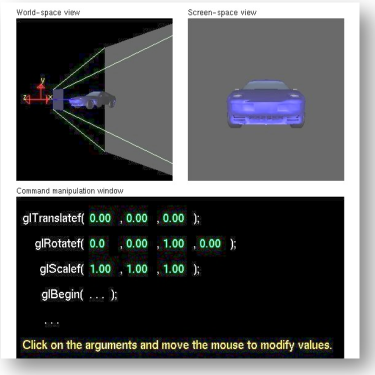


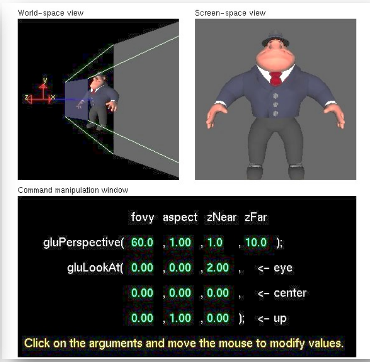
34

MANCHESTER
1824

Demonstrations

- Nate Robbins' demonstrations will help you to visualise viewing.
- /opt/info/courses/OpenGL/tutor (Linux)





35