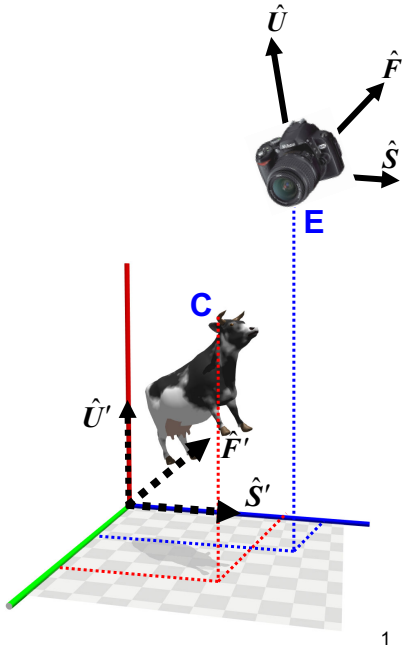
MANCHESTER 1824

COMP27112

Computer Graphics and Image Processing

5: Viewing 1



Toby.Howard@manchester.ac.uk

MANCHESTER 1824

Introduction

- In part 1 of our study of viewing, we'll look at

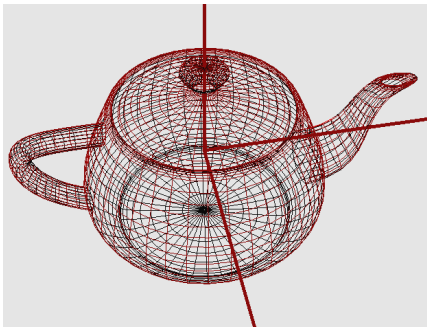
1. Viewing in 3D
2. Using a camera
3. There is no CG camera!
4. The duality of modelling and viewing
5. Simulating a camera using transformations

6. Reference material: implementing viewing in OpenGL

MANCHESTER 1824

Viewing 3D in 2D

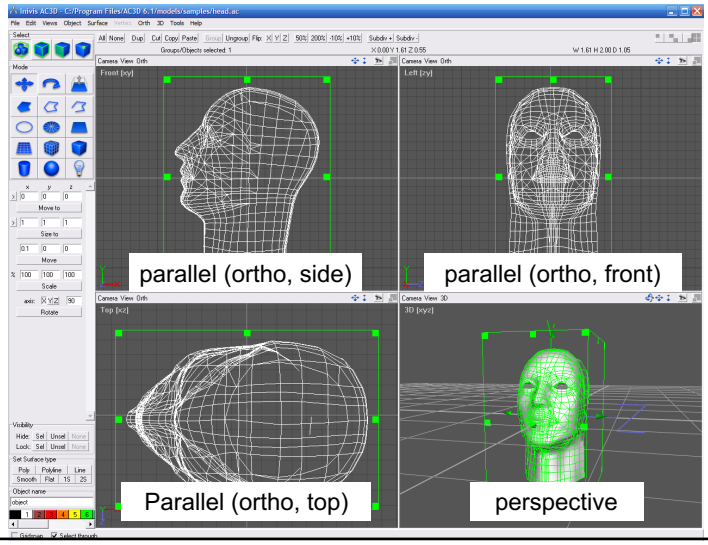
- We usually only have 2D displays – flat screens
- So a 3D object has to be projected from 3D to 2D



- In daily life we make 3D to 2D views all the time – using a camera
- (Our eyes reconstruct 3D from two 2D projections on our retinas)

MANCHESTER 1824

- Example of 3D viewing. AC3D (Coursework 2) uses 3 **parallel orthogonal** views and one **perspective** view. We look at these projections in Lecture 6.



MANCHESTER 1824

The camera analogy

Step 1: Arrange the models into the desired composition

5

MANCHESTER 1824

The camera analogy

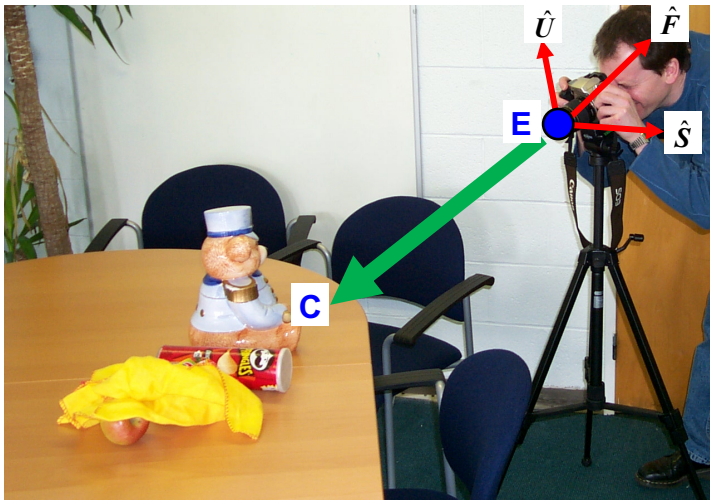
Step 2: Position the camera and point it at the scene

6

MANCHESTER 1824

The camera analogy

Step 2: Position the camera and point it at the scene



MANCHESTER 1824

The camera analogy

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

MANCHESTER 1824

The camera analogy

Step 4: Decide the size of the final photograph

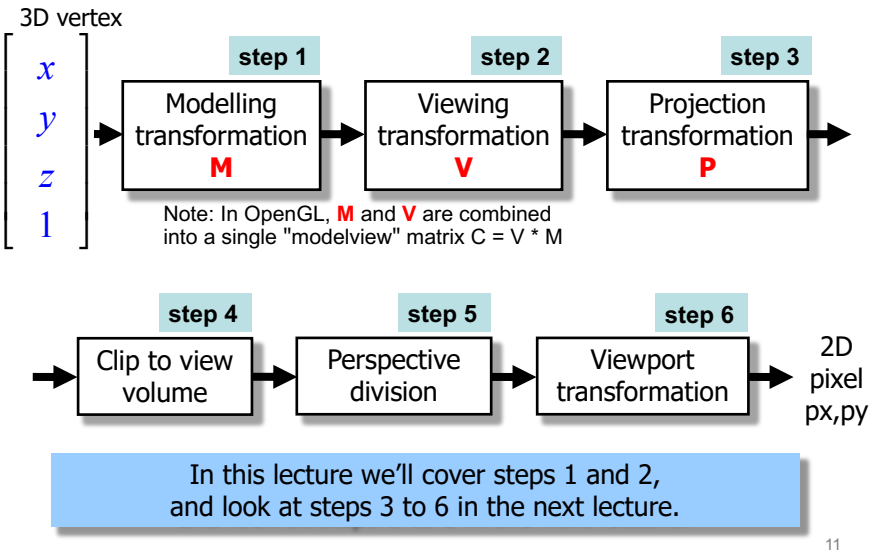
MANCHESTER 1824

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

MANCHESTER 1824

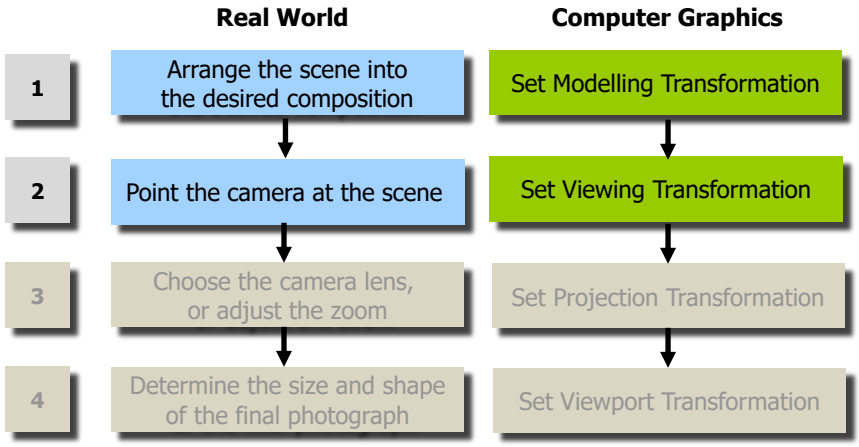
The 3D Viewing Pipeline



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

MANCHESTER 1824

3D Viewing: the camera analogy



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 C = V * M

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

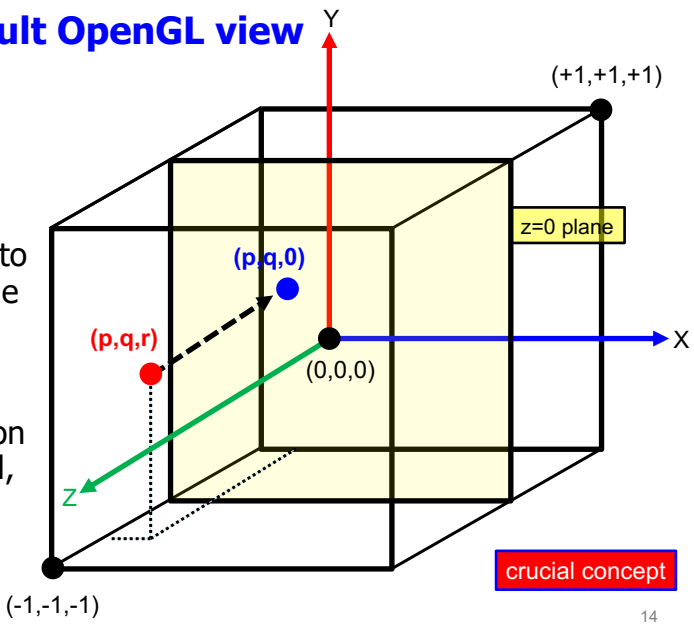
In this lecture we'll cover steps 1 and 2, and look at steps 3 to 6 in the next lecture.

MANCHESTER 1824

The default OpenGL view

An (x,y,z) point drawn by the user (red) is projected onto the z=0 plane (blue).

The projection is orthogonal, with projectors parallel to the z axis



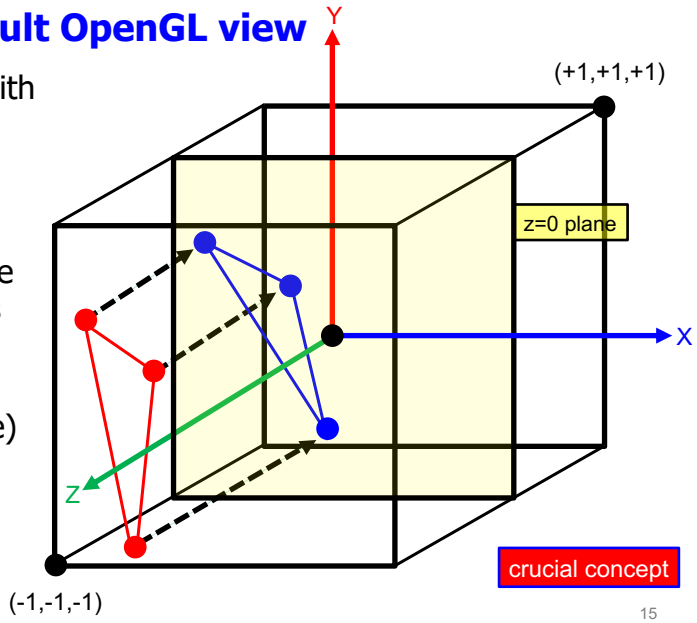
crucial concept

MANCHESTER 1824

The default OpenGL view

Example with lines.

OpenGL projects the user's lines (red) onto the z=0 plane (blue)

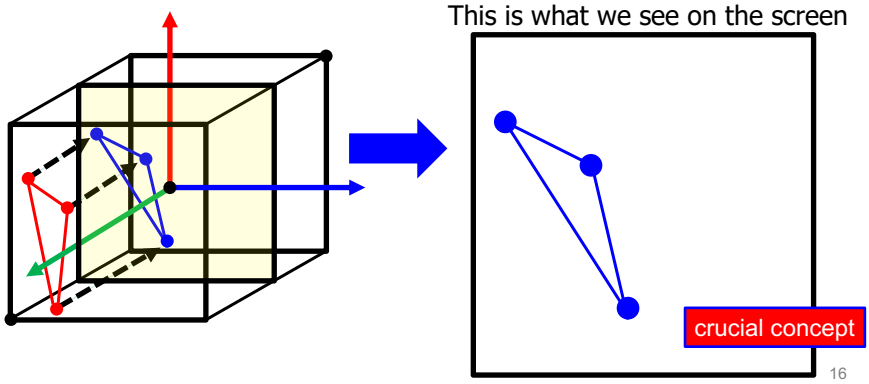


crucial concept

MANCHESTER 1824

The default OpenGL view

The z=0 plane then gets mapped to the display screen, and whatever geometry is there gets scan-converted (aka rasterised) and the z-buffer applied.



crucial concept

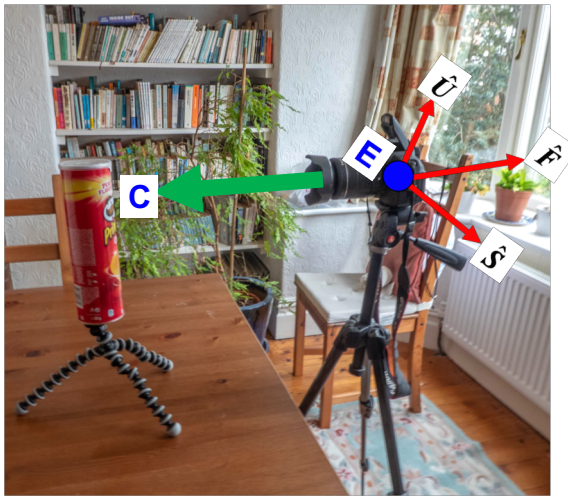
MANCHESTER 1824

The duality of modelling and viewing

17

MANCHESTER 1824

The duality of modelling and viewing



18

MANCHESTER 1824

The duality of modelling and viewing

- image A: camera at position P, looking straight at Pringles.
- image B: Pringles stay fixed. Camera rotated around X-axis by +60 degrees.
- image C: Camera reset to P. Pringles rotated around X-axis by -60 degrees.

crucial concept

MANCHESTER 1824

The duality of modelling and viewing

- We obtain the same view from a camera at a certain location and orientation, by **instead** transforming the object.
- In CG we have no camera. Only transformations.
- But because the idea of a camera is so familiar, we **imagine** we have a camera.
- The OpenGL API allows us to use a "camera", but in fact we are using modelling transformations.
- From now on, we will just say camera, not "camera".

MANCHESTER 1824

in CG, there is no camera.

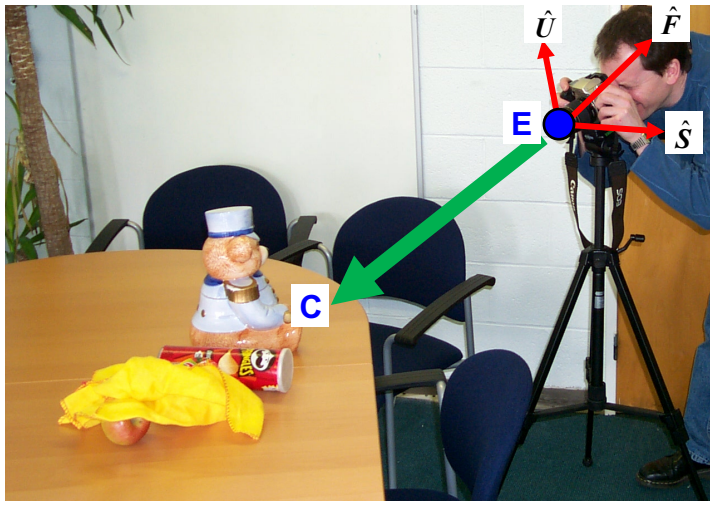
we simulate a camera by transforming the model.

the ultimate crucial computer graphics concept

21

MANCHESTER 1824

Specifying the camera

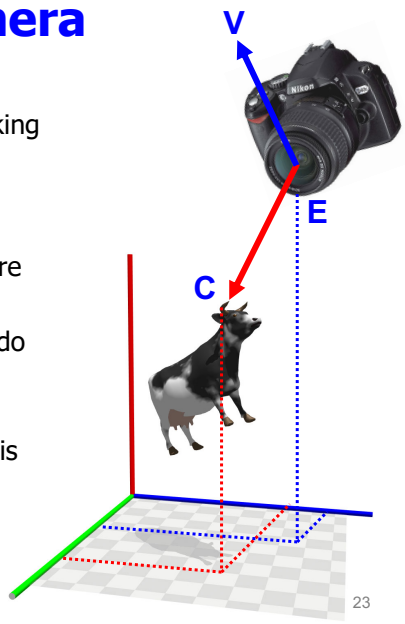


22

MANCHESTER 1824

Specifying the camera

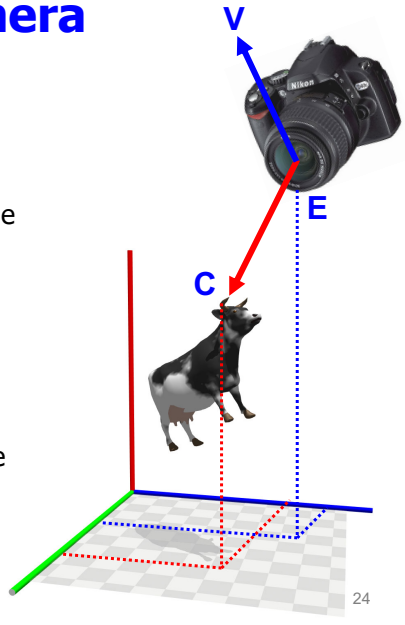
- In the default OpenGL view (slides 14-16) the camera is a (0,0,0) looking down the z-axis
- We move the default camera to desired point **E**, with desired orientation **V**, and pointing at centre of interest **C**
- Call the transformation needed to do this move from (0,0,0), $\mathbf{T_C}$
- We then compute the **inverse** transformation, $\mathbf{T_C^{-1}}$, and apply this to our models
- We'll now look at details of this.



MANCHESTER 1824

Specifying the camera

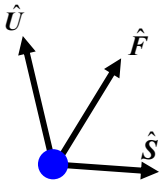
- Where is our camera in 3D space? the **eye point, E**
- Where is our camera looking at? the **centre of interest C**
- What is the **up** direction of our camera? the "view up vector" **V**
- From **E**,**C**, and **V** we derive $\mathbf{T_C}$, the transformation which would move the camera here from (0,0,0)

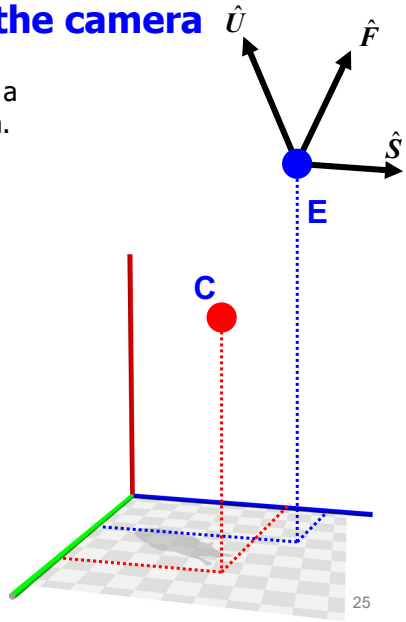


MANCHESTER 1824

A coordinate system for the camera

- Things are easiest if we establish a coordinate system for the camera.
- We use **E**, **C** and **V** to derive this.
- We'll call the axes of the camera coordinate system $\hat{S}\,\hat{U}\,\hat{F}$

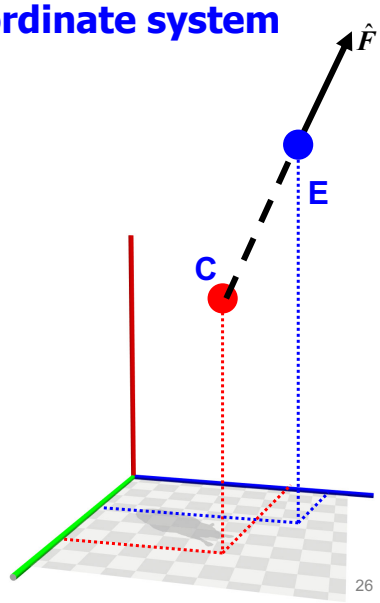




MANCHESTER 1824

Defining the camera coordinate system

- Let's start with $\hat{F}$. This is straightforward to define, in two steps:
- Step 1. $F = E - C$
- Step 2. The length of **F** could be anything, depending on **C** and **E**, and since we are using **F** to specify a direction only, we normalize it be of length 1, and then call it $\hat{F}$



MANCHESTER 1824

Defining the camera coordinate system

- Now we consider the "up" direction of the camera. The user defines this by specifying a vector **V**, called the "view-up" vector.
- Here are some possible values of **V**:

V = (0,1,0)

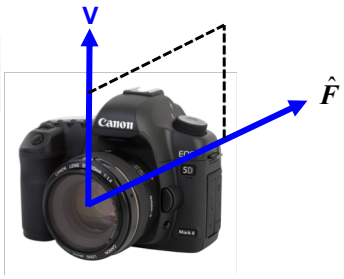
V = (-1,1,0)

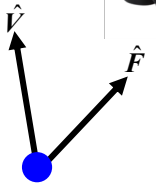
V = (-1,0,0)

MANCHESTER 1824

Defining the camera coordinate system

- Let's continue defining the $\hat{S}\,\hat{U}\,\hat{F}$ coordinate system of the camera. So far we have one axis, $\hat{F}$, so let's try and define another axis.
- If **V** is the 'up' direction of the camera, you would think that V would be orthogonal to F
- OK, let's assume it is, then let's normalise it, to give $\hat{V}$, and we now have defined two of the three axes of the camera.
- We will come back to the red box above, shortly...

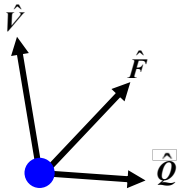




MANCHESTER 1824

Defining the camera coordinate system

- Now all that remains is to define the third axis, which needs to orthogonal to both $\hat{V}$ and $\hat{F}$
- This is straightforward, we just take the cross product of $\hat{V}$ and $\hat{F}$. Let's call this $\hat{Q}$
- $\hat{Q}$ = normalise($\hat{V}$ x $\hat{F}$)



- So now we have established the camera's coordinate system.
- But there's a problem...

MANCHESTER 1824

Defining the camera coordinate system

- This system is based on this assumption:

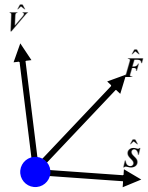
> If **V** is the 'up' direction of the camera, you would think that V would be orthogonal to F

- In other words, we are **assuming** that the user has made sure that **V** is actually orthogonal to **F**. To do this, they would first have to work out **F**, and then define **V** accordingly.
- Can we rely on all users to always do that? No!
- If they don't, and specify a **V** that is not orthogonal to **F**, then the whole derivation of the camera coordinate system will not work, and the axes will point in all kinds of strange directions.
- So what's the solution?

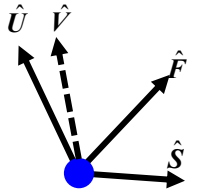
MANCHESTER 1824

Defining the camera coordinate system

- The solution is to decouple $\hat{V}$ and $\hat{F}$, by **not making any assumptions** about their relationship
- We first derive $\hat{F}$ as before: $\hat{F}$ = normalize(C - E)
- Then we use $\hat{F}$ and $\hat{V}$ to create $\hat{S}$ orthogonal to them both: $\hat{S}$ = normalise($\hat{F}$ x $\hat{V}$)



- Then we use $\hat{S}$ and $\hat{F}$ to create $\hat{U}$: $\hat{U}$=normalise($\hat{S}$ x $\hat{F}$) and we **forget all about** $\hat{V}$

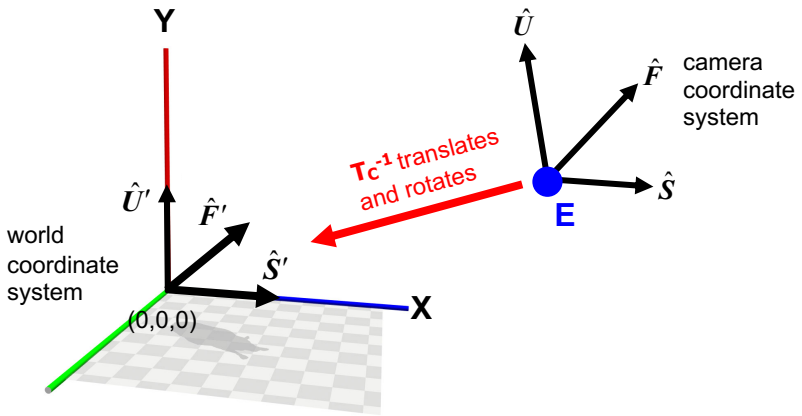


- We've constructed a camera coordinate system $\hat{S}\,\hat{U}\,\hat{F}$ that has involved **making no assumptions**, and is guaranteed to be a correct orthogonal coordinate system.

MANCHESTER 1824

Deriving the viewing transformation

- Now we have the camera axes, we can derive T_C^{-1} which translates & rotates the $\hat{S}\,\hat{U}\,\hat{F}$ camera system to (0,0,0):



MANCHESTER 1824

Deriving T_C^{-1} in three steps

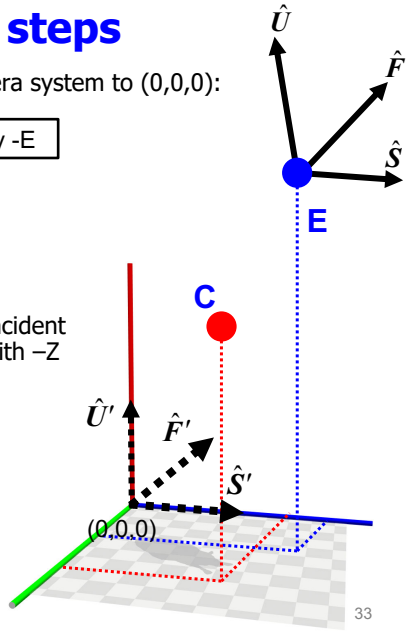
1. translate the origin of the $\hat{S}\hat{U}\hat{F}$ camera system to (0,0,0):

T1 = $\begin{bmatrix} 1 & 0 & 0 & -E_x \\ 0 & 1 & 0 & -E_y \\ 0 & 0 & 1 & -E_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$ just a translate by -E

2. now rotate the camera axes to be coincident with the world axes, with $\hat{F}$ aligned with –Z

T2 = $\begin{bmatrix} \hat{S}_x & \hat{S}_y & \hat{S}_z & 0 \\ \hat{U}_x & \hat{U}_y & \hat{U}_z & 0 \\ -\hat{F}_x & -\hat{F}_y & -\hat{F}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ we omit the derivation of this.

3. T_C^{-1} = **T2 * T1**



MANCHESTER 1824

Summary of viewing

- The duality of modelling and viewing says we get the same view by transforming the camera by **T**, or the object by $\mathbf{T^{-1}}$
- The default camera (i.e. OpenGL default view) is at (0,0,0) looking down the z-axis
- If the transformation that moves the default camera to the desired viewpoint is T_C then we transform an object by T_C^{-1}

crucial concept

MANCHESTER 1824

Using T_C^{-1} in OpenGL

- The good news is that the programmer doesn't need to know anything about these transformations!
- They are worked out for us by OpenGL, and automatically applied to the current modelview matrix
- `gluLookAt()` takes (**E**,**C**,**V**) and computes T_C^{-1}
- So we think about a camera, not about transformations

MANCHESTER 1824

Reference material on OpenGL viewing

- The following slides describe in more detail the programming steps for using the OpenGL camera
- Refer to these for your lab exercises.

MANCHESTER 1824

Using T_C^{-1} in OpenGL

- `gluLookAt()` takes (**E**,**C**,**V**) and computes T_C^{-1}

OpenGL

```
void gluLookAt (GLdouble eyex,
                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**: $M_{modelview} = M_{modelview} * T$

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

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**

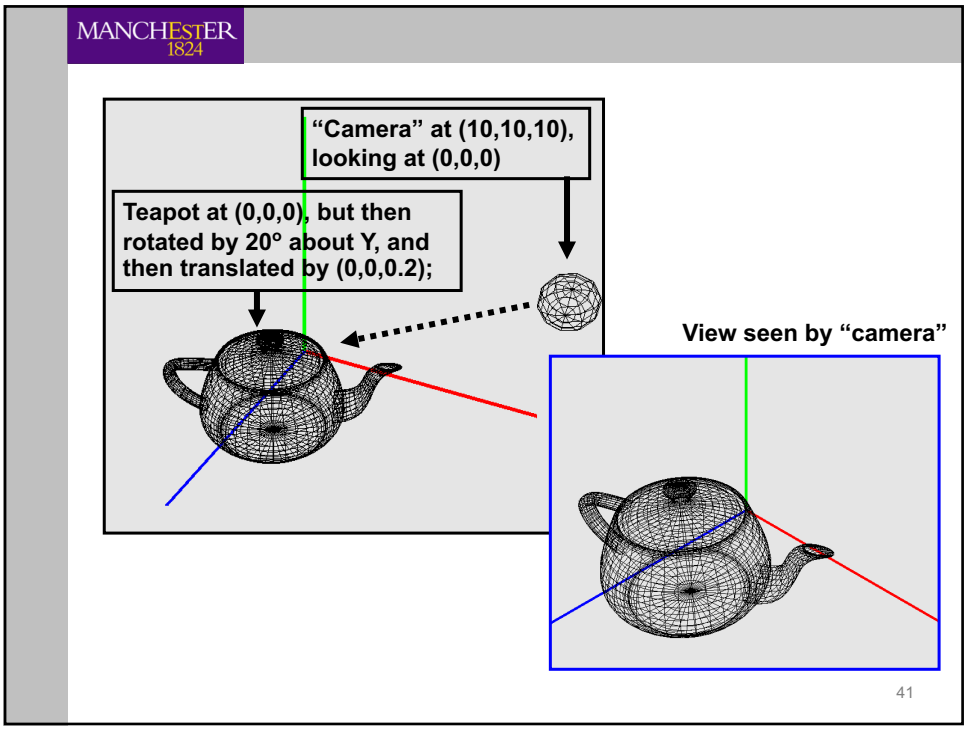
MANCHESTER 1824

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.


MANCHESTER
1824
"Camera" at (10,10,10), looking at (0,0,0)
Teapot at (0,0,0), but then rotated by 20° about Y, and then translated by (0,0,0.2);
View seen by "camera"
41