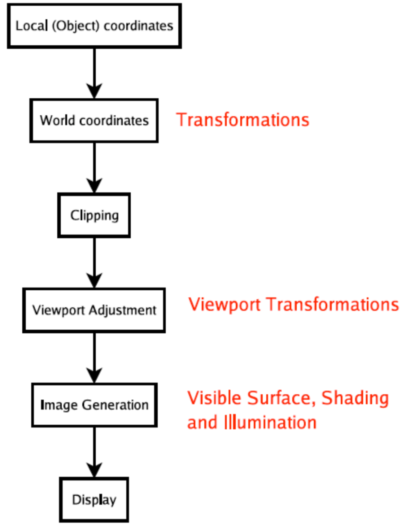
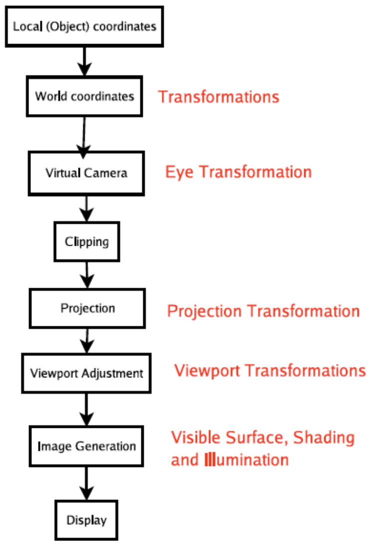
Visualisation Pipeline : The Virtual Camera

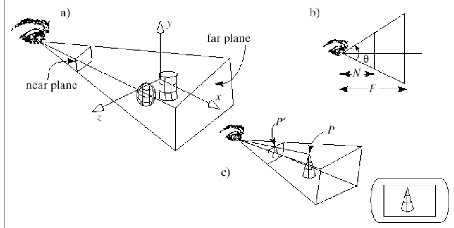
The Graphics Pipeline



3D Pipeline



The Virtual Camera

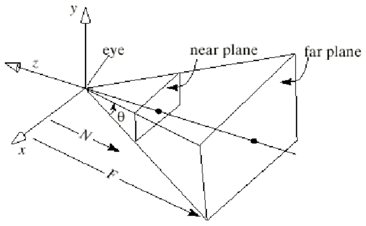


- The Camera is defined by using a parallelepiped as a view volume with two of the walls used as the near and far view planes
- Most applications also allow for a perspective view to be created, this is done by changing the shape of the view volume.

Perspective Camera

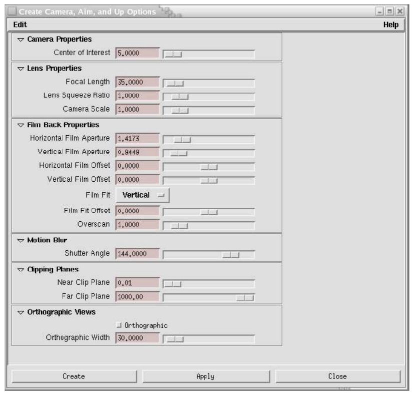
- The Camera has an eye positioned at some point in space.
- Its view volume is a portion of a rectangular pyramid, whose apex is at the eye.
- The opening of the pyramid is set by the viewangle θ (part b of figure)
- Two planes are defined perpendicular to the axis of the pyramid : the near and the far plane.
- Where these planes intersect the pyramid they form rectangular windows which have an adjustable aspect ratio.
- The application clips points which lie outside the view volume. Points lying inside the the view volume are projected onto the viewplane to a corresponding point P'
- With a perspective projection the point P' is determined by finding where a line from the eye to P intersects the viewplane.

Setting the View Volume



- Most applications provides a simple way to set the view volume in a program by setting the projection Matrix

Maya Camera setup



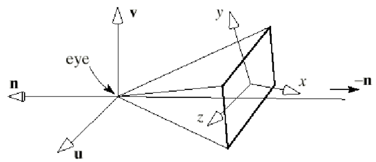
Modelling and Viewing

- Typically Modelling and Viewing are a combination of two matrices
- The Modelling Matrix contains the local (world) transformations used to create the model
- The Viewing Matrix contains the Camera transformations to position the model relative to the Camera.
- Usually these two matrices are combined by multiplication to set one matrix know as the MODELVIEW matrix
- All vertices are then passed through this to give the final matrix

Positioning and Pointing the Camera

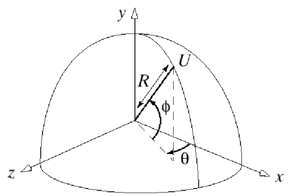
- In order to obtain the desired view of a scene, we move the camera from its default position as shown in the previous slide
- To point it in a particular direction we need to perform rotations and translations which result in changes to the modelview matrix

The General Camera with Arbitrary Orientation and Position



- It is useful to attach an explicit co-ordinate system to the camera as shown in the figure above.
- This co-ordinate system has its origin at the eye and has three axes, usually called the u-,v-, and n-axis which define the orientation of the camera
- The axes are pointed in directions given by the vectors **u**,**v**, and **n**
- Because, by default, the camera looks down the negative z-axis, we say in general that the camera looks down the negative n-axis in the direction -**n**
- The direction **u** points off "to the right of" the camera and the direction **v** points "upward"

Spherical Geometry



- The above figure shows how a point U is defined in spherical coordinates.
- R is the radial distance of U from the origin, and ϕ is the angle that U makes with the xz-plane, know as the **latitude** of the point U.
- θ is the **azimuth** of U, the angle between the xy-plane and the plane through U and the y-axis.
- ϕ lies in the interval $-\frac{\pi}{2} \leq \phi < \frac{\pi}{2}$ and θ lies in the range $0 \leq \theta < 2\pi$

Spherical Co-ordinates

- Using trigonometry we can work out a relationship between spherical co-ordinates and Cartesian coordinates (u_x, u_y, u_z) for U the equations are

$u_x = Rcos(\phi)cos(\theta)$, $u_y = Rsin(\phi)$, and $u_z = Rcos(\phi)sin(\theta)$

- We can also invert the relations to express (R, ϕ, θ) in terms of (u_x, u_y, u_z)

$R = \sqrt{u_x^2 + u_y^2 + u_z^2}$, $\phi = sin^{-1}\left(\frac{u_y}{R}\right)$, $\theta = arctan(u_z, u_x)$

- The function arctan(,) is the two argument form of the arctangent, defined as (atan2 in C)

$$arctan(y,x) = \begin{cases} tan^{-1}(y/x) & if\, x > 0 \\ \pi + tan^{-1}(y/x) & if\, x < 0 \\ \pi/2 & if\, x = 0\, and\, y > 0 \\ -\pi/2 & if\, x = 0\, and\, y < 0 \end{cases}$$

Example

Suppose that Point U is at a distance 2 from the origin, is 60^o up from the xz-plane, and is along the negative x-axis.

Hence U is in the xy-plane. Then U is expressed in spherical coordinates as $(2, 60^o, 180^o)$

To convert this to Cartesian coordinates we use the equations :

$u_x = Rcos(\phi)cos(\theta)$,
$u_y = Rsin(\phi)$,
$u_z = Rcos(\phi)sin(\theta)$
Where
$u_x = 2cos(60^o)cos(180^o)$ =-1
$u_y = 2sin(60^o) = 1.732$
$u_z = 2cos(60^o)sin(180^o) \approx 0$

C Example

```c
#include <stdio.h>
#include <math.h>


int main(void)
{
//cos and sin need angle in radians
//to convert we mult by PI/180
float scale=M_PI/180;
float phi=60*scale;
float theta=180*scale;
float radius=2.0;

printf("x=%f\n",radius*cos(phi)*cos(theta));
printf("y=%f\n",radius*sin(phi));
printf("z=%f\n",radius*cos(phi)*sin(theta));

return 1;
}
```

Direction Cosines

- The direction of point U in the previous example is given in terms of two angles : the **azimuth** and the **latitude**.
- Directions are often specified in an alternative useful way through direction cosines.
- The direction cosines of a line through the origin are the cosines of the three angles it makes with the $x-, y-$ and $z-$axes respectively.
- Recall that the cosine of the angle between two unit vectors is given by their dot product.
- Using the given point U we form the position vector (u_x, u_y, u_z)
- We also know that the length of the vector is R so we normalise it to unit length thus $\mathbf{m} = (u_x/R, u_y/R, u_z/R)$

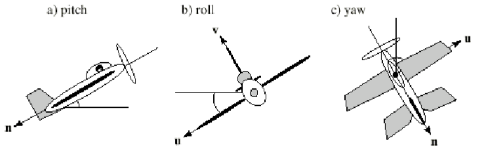
Direction Cosines II

- Then the cosine of the angle it makes with the $x-$axis is given by the dot product $\mathbf{m} \bullet \mathbf{i} = u_x/R$ which is the first component of $\mathbf{m}$
- Similarly the second and third components of $\mathbf{m}$ are the second and third direction cosines respectively
- Calling the angles made with the $x-, y-$ and $z-$axis α, β and γ respectively, we have, for the three direction cosines for the line from 0 to U

$cos(\alpha) = \frac{u_x}{R}$,
$cos(\beta) = \frac{u_y}{R}$,
and
$cos(\gamma) = \frac{u_z}{R}$

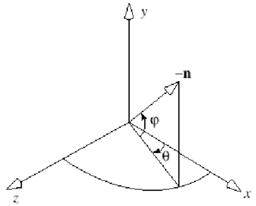
- Note that the three direction cosines are related, since the sum of their squares is always unity

Describing Orientation



- Position is easy to describe, but orientation is difficult. It helps to specify orientation using the aviation terms **pitch,heading,yaw** and **roll** as shown in the figure above
- The pitch of an air plane is the angle that its longitudinal axis (running from tail to nose having direction -**n**) makes with the plane.
- A roll is a rotation about the longitudinal axis; the roll is the amount of rotation relative to the horizontal.
- An airplane's heading is the direction it is headed (This is also called azimuth and bearing)

Describing Orientation



- To find the heading and the pitch, given $\mathbf{n}$ simply express $-\mathbf{n}$ in spherical coordinates, as shown in the figure
- The vector $-\mathbf{n}$ has longitude and latitude given by the angles θ and ϕ respectively.
- The heading of the plane is given by the longitude of $-\mathbf{n}$ and the pitch is given by the latitude of $-\mathbf{n}$

Finding roll, pitch, and heading given vectors u, v, and n.

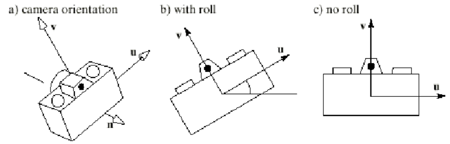
Assuming the camera is based on a coordinate system with axes in the directions $\mathbf{u}, \mathbf{v}$, and $\mathbf{n}$, all unit vectors. The heading and pitch of the camera is denoted by θ and ϕ. The relationship of Cartesian and spherical coordinates system is given by:

$$n_x = cos(\phi)cos(\theta)$$
$$n_y = sin(\phi)$$
$$n_x = sin(\phi)cos(\theta) \Rightarrow \begin{cases} \theta = tan^{-1}(-n_z/-n_x) \\ \phi = sin^{-1}(-n_y) \end{cases}$$

The roll of the camera is the angle its u-axis makes with the horizontal, To find it construct a vector

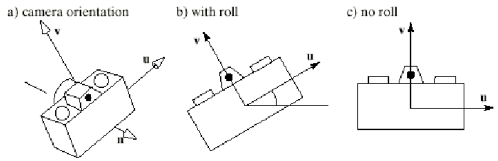
$\mathbf{b} = \mathbf{j} \times \mathbf{n} = (n_z, 0, -n_x)$
$\mathbf{b} \bullet \mathbf{j} = (\mathbf{j} \times \mathbf{n}) \bullet \mathbf{j} = 0 \Rightarrow \mathbf{b}$ is horizontal
$\mathbf{b} \bullet \mathbf{n} = (\mathbf{j} \times \mathbf{n}) \bullet \mathbf{n} = 0 \Rightarrow \mathbf{b}$ is perpendicular to n and therefore lies in the uv-plane
Since
$|\mathbf{b}| = \sqrt{n_x^2 + n_z^2}$
$\mathbf{b} \bullet \mathbf{u} = u_x n_z - u_z n_x$
The angle between $\mathbf{b}$ and $\mathbf{u}$ is given by
$cos(roll) = \frac{\mathbf{b} \bullet \mathbf{u}}{|\mathbf{b}|} = \left(\frac{u_x n_z - u_z n_x}{\sqrt{n_x^2 + n_z^2}} \right)$ so the roll of the camera can be expressed as
$roll = cos^{-1}\left(\frac{u_x n_z - u_z n_x}{n_x^2 + n_z^2} \right)$
$heading = arctan(-n_z, -n_x)$ and $pitch = sin^{-1}(-n_y)$

Camera



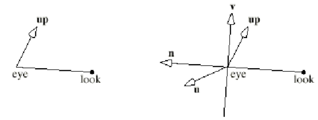
- The above figure shows a camera with the same coordinate system attached to it
- It has **u-,v-** and **n-**axes and an origin at position eye
- b) shows the camera with a roll applied to it
- c) shows the camera with zero roll or “no-roll” camera
- The u-axis of a no-roll camera is horizontal, that is perpendicular to the y-axis of the “world”
- Note that a no-roll camera can still have an arbitrary **n** direction, so it can have any pitch or heading.

Camera



- The above figure shows a camera with the same coordinate system attached to it
- It has u-,v- and n-axes and an origin at position eye
- b) shows the camera with a roll applied to it
- c) shows the camera with zero roll or "no-roll" camera
- The u-axis of a no-roll camera is horizontal, that is perpendicular to the y-axis of the "world"
- Note that a no-roll camera can still have an arbitrary **n** direction, so it can have any pitch or heading.

How a typical camera works inside a modelling application



- What are the directions of **u**,**v** and **n** when we place our camera with given values for eye, look and up
- If given the locations of eye, look and up, we immediately know than **n** must be parallel to the vector **eye-look**, as shown above. so we can set **n=eye-look**
- We now need to find a **u** and a **v** that are perpendicular to **n** and to each other.
- The **u** direction points "off to the side" of a camera, so it is natural to make it perpendicular to **up** which the user has said is the "upward" direction.

How a camera works II

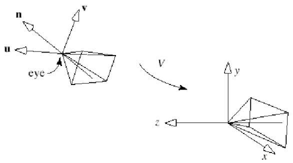
- An easy way to build a vector that is perpendicular to two given vector is to form their cross product, so we set **u=up** X **n**
- The user should not choose an **up** direction that is parallel to **n**, because **u** then would have zero length.
- We choose **u**=**up** X **n** rather than **n**X**up** so that **u** will point "to the right" as we look along -**n**
- With **u** and **n** formed it is easy to determine **v** as it must be perpendicular to both and is thus the cross product of **u** and **n** thus **v**=**n** X **u**
- Notice that **v** will usually not be aligned with **up** as **v** must be aimed perpendicular to **n** whereas the user provides **up** as a suggestion of "upwardness" and the only property of **up** that is used is its cross product with **n**

Camera III

- To summarise, given eye look and up, we form

$\mathbf{n} = eye - look$
$\mathbf{u} = \mathbf{up} \times \mathbf{n}$
and
$\mathbf{v} = \mathbf{n} \times \mathbf{u}$
and then normalise the vectors to unit length.

The Camera and the modelview Matrix



- The modelview is the product of two matrices the matrix **V** that accounts for the transformation of the world point into camera coordinates.
- and **M** that embodies all of the modelling transformations applied to the points.
- Typically we build the **V** matrix and post-multiplies the current matrix by it.
- Because the job of the **V** matrix is to convert world coordinates to camera coordinates it must transform the camera's coordinate system into the generic position for the camera as shown in the figure.

MODEL / VIEW Matrix

- This means that V must transform eye into the origin, $\mathbf{u}$ into the vector $\mathbf{i}$, $\mathbf{v}$ into $\mathbf{j}$, and $\mathbf{n}$ $\mathbf{k}$
- The easiest way to define V is to use the following matrix

$$V = \begin{pmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Where $(d_x, d_y, d_z) = (-eye \bullet \mathbf{u}, -eye \bullet \mathbf{v}, -eye \bullet \mathbf{n})$

The matrix V is created by *gluLookAt* and post-multiplies the current matrix.

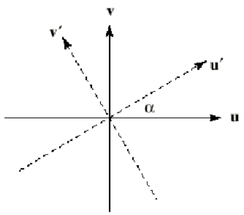
The Camera Revisited

- In order to have full control over the viewing of a scene we need to create our own Camera
- After each change to the camera's state the camera will "tell" the applicaion where the current View of the scene is to be modifying the MODELVIEW matrix
- When we use the Virtual Camera we create a camera by setting the EYE, LOOK as points and UP as a vector.
- We then call the functions **roll**, **pitch**, **yaw** and **slide** to move the camera's position (or equivalent).

Sliding the Camera

- Sliding the camera means moving it along one of it's own axis, that is in the **u**,**v** or **n** axis without rotating it.
- Since the camera is looking along the negative **n-**axis, movement along **n** is "forward" or "back"
- Similarly, movement along **u** is "left" or "right" and along **v** is "up" or "down"
- It is simple to move the camera along one of its axes. To move it a distance D along its u-axis, set eye to eye+Du
- This can be done for each of the axes and combined into a single function

Rotating the Camera



- To roll the camera is to rotate it about its own n-axis. This means that both the directions u and v must be rotated as shown below
- We form two new axes u' and v' that lie in the same plane as u and u, yet have been rotated through the angle α radians.
- To do this we need to form u' as the appropriate linear combination of u and v and similarly for v'

rolling the Camera

$$\boxed{\begin{array}{l}\mathbf{u}' = cos(\alpha)\mathbf{u} + sin(\alpha)\mathbf{v} \\ \mathbf{v}' = -sin(\alpha)\mathbf{u} + cos(\alpha)\mathbf{v}\end{array}}$$

- To roll the camera we need to replace the current u and v axes with the new u' and v' axes
- The Function to do this is shown next

References

- Computer Graphics With OpenGL, F.S. Hill jr, Prentice Hall (most images from the instructors pack of this book)
- Basic Algebra and Geometry. Ann Hirst and David Singerman. Prentice Hall 2001
- "Essential Mathematics for Computer Graphics fast" John VinceSpringer-Verlag London
- "Geometry for Computer Graphics: Formulae, Examples and Proofs" John Vince Springer-Verlag London 2004