Computer Graphics

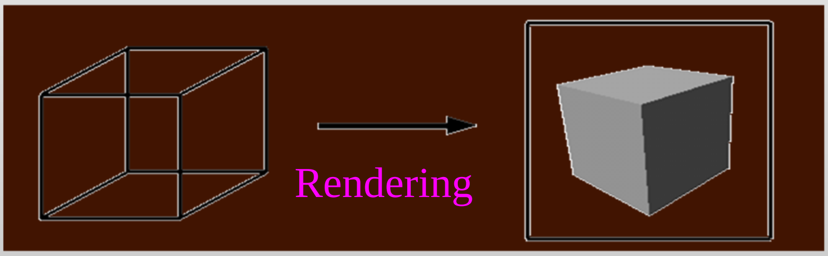
Lecture 5
Geometric primitives

Instructor: Dr. MAO Aihua

ahmao@scut.edu.cn

Rendering

Generate an image from geometric primitives



Geometric Primitives

Raster Image

Rendering geometric primitives

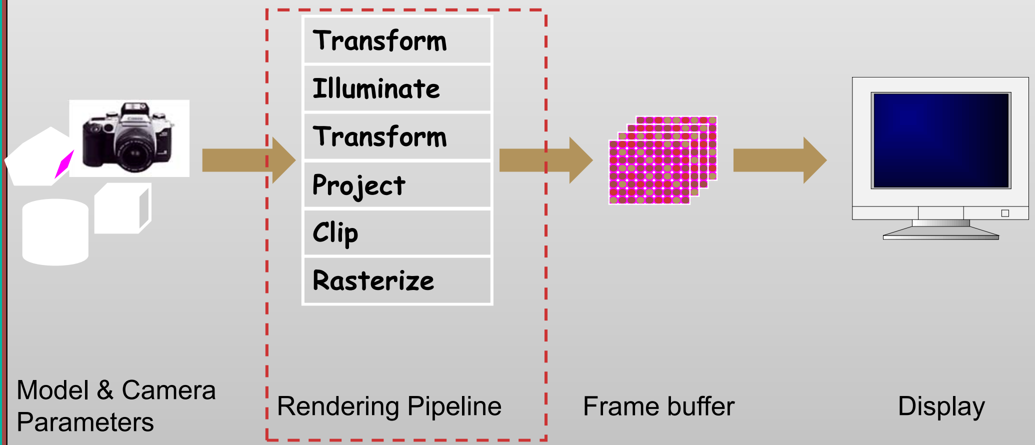
Describe objects with points, lines, and surfaces

- Compact mathematical notation
- Operators to apply to those representations

Render the objects

- The rendering pipeline

Rendering 3D Scenes



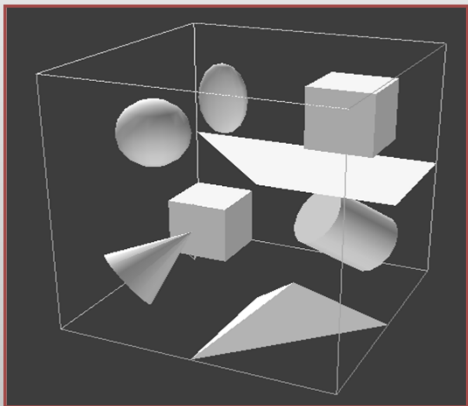
Pipeline (for direct illumination)

3D Geometric Primitives

Modeling Transformation	Transform into 3D world coordinate system
Lighting	Illuminate according to lighting and reflectance
Viewing Transformation	Transform into 3D camera coordinate system
Projection Transformation	Transform into 2D screen coordinate system
Clipping	Clip primitives outside camera's view
Scan Conversion	Draw pixels (includes texturing, hidden surface, ...)

Image

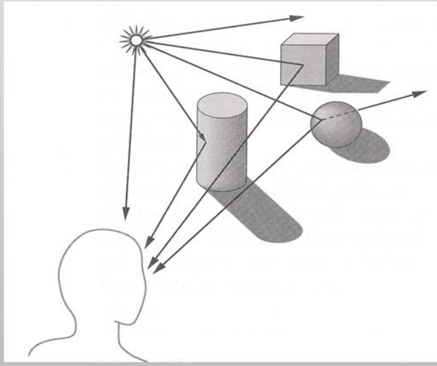
3D Rendering Example



What issues must be addressed by a 3D rendering system?

Overview

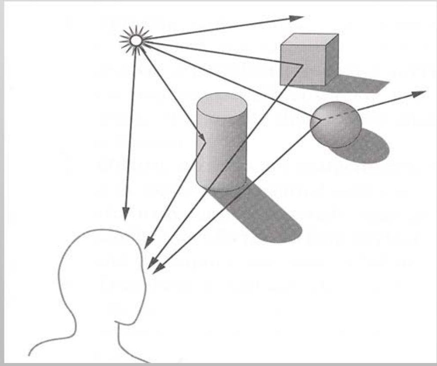
- 3D scene representation
- 3D viewer representation
- Visible surface determination
- Lighting simulation



Overview

- 3D scene representation
- 3D viewer representation
- Visible surface determination
- Lighting simulation

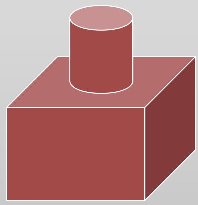
How is the 3D scene described in a computer?



3D Scene Representation

Scene is usually approximated by 3D primitives

- Point
- Line segment
- Polygon
- Polyhedron
- Curved surface
- Solid object
- etc.



3D Point

Specifies a location

3D Point

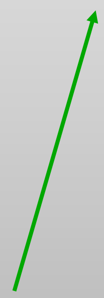
Specifies a location

- Represented by three coordinates
- Infinitely small

(x,y,z)

3D Vector

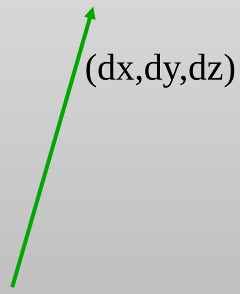
Specifies a direction and a magnitude



3D Vector

Specifies a direction and a magnitude

- Represented by three coordinates
- Magnitude ||V|| = sqrt(dx dx + dy dy + dz dz)
- Has no location



Vector Addition/Subtraction

- operation **u + v**, with:
  - *Identity* $\mathbf{0}$ *:* $\mathbf{v} + \mathbf{0} = \mathbf{v}$
  - *Inverse* **-** *:* $\mathbf{v} + (-\mathbf{v}) = \mathbf{0}$
- Addition uses the “parallelogram rule”:

Vector Space

Vectors define a vector space

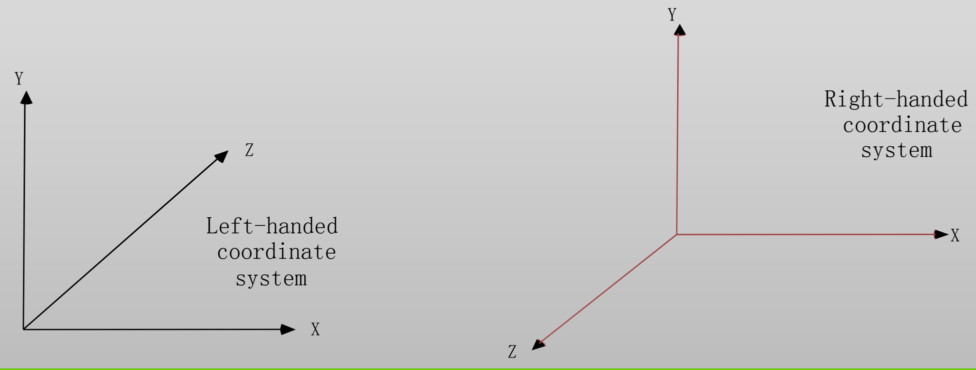
- They support vector addition
  - *Commutative*: $\mathbf{X}+\mathbf{Y}=\mathbf{Y}+\mathbf{X}.$ *Associative*: $(\mathbf{X}+\mathbf{Y})+\mathbf{Z}=\mathbf{X}+(\mathbf{Y}+\mathbf{Z}).$
  - *Identity*: $\mathbf{0}+\mathbf{X}=\mathbf{X}+\mathbf{0}=\mathbf{X}.$ *Inverse* $\mathbf{X}+(-\mathbf{X})=\mathbf{0}.$
- They support scalar multiplication
  - *Associative* $r(s\,\mathbf{X})=(r\,s)\,\mathbf{X}.$ *Distributive* $r(\mathbf{X}+\mathbf{Y})=r\,\mathbf{X}+r\,\mathbf{Y}.$
  - *Possess identity* $1\,\mathbf{X}=\mathbf{X}.$

Affine Spaces

- Since vector spaces lack position and distance
  - *They have magnitude and direction but no location*
- Combine the point and vector primitives
  - *Permits describing vectors relative to a common location*
- A point and three vectors define a 3-D coordinate system
- Point-point subtraction yields a vector

Coordinate Systems

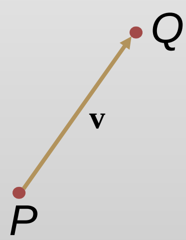
- Grasp z-axis with hand
- Thumb points in direction of z-axis
- Roll fingers from positive x-axis towards positive y-axis



Points + Vectors

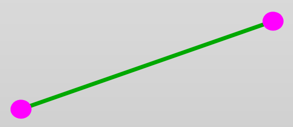
Points support these operations

- Point-point subtraction: $Q - P = \mathbf{v}$
  - *Result is a vector pointing from* P *to* Q
- Vector-point addition: $P + \mathbf{v} = Q$
  - *Result is a new point*
- Note that the addition of two points is not defined



3D Line Segment

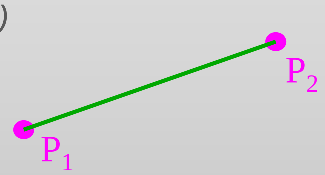
Linear path between two points



3D Line Segment

Use a linear combination of two points

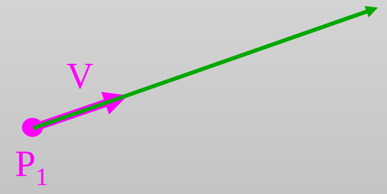
- Parametric representation:
  - $P = P_1 + t\,(P_2 - P_1), \quad (0 \le t \le 1)$



3D Ray

Line segment with one endpoint at infinity

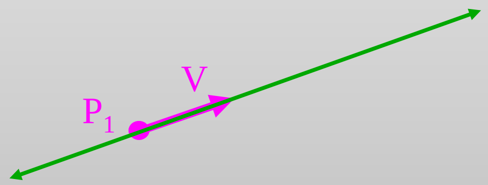
- Parametric representation:
  - $P = P_1 + t\,V, \quad (0 <= t < \infty)$



3D Line

Line segment with both endpoints at infinity

- Parametric representation:
  - $P = P_1 + t\,V, \quad (-\infty < t < \infty)$



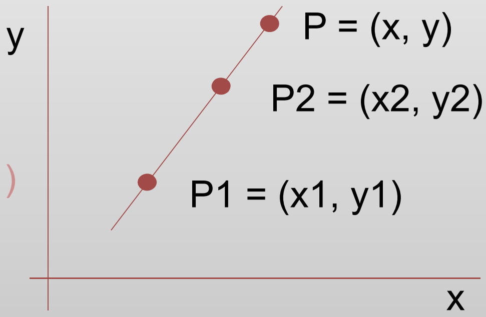
3D Line – Slope Intercept

Slope =m

= rise / run

Slope = (y - y1) / (x - x1)

= (y2 - y1) / (x2 - x1)



Solve for y:

y = [(y2 - y1)/(x2 - x1)]x + [-(y2-y1)/(x2 - x1)]x1 + y1

or: y = mx + b

Euclidean Spaces

Named by Ancient Greek mathematician Euclid:

express relationships in terms of distance and angle.

Q: What is the distance function between points and vectors in affine space?

A: Dot product

- Euclidean affine space = affine space + dot product
- Permits the computation of distance and angles

Dot Product

- The dot product or, more generally, inner product of two vectors is a scalar:

$$\mathbf{v}_1 \bullet \mathbf{v}_2 = \mathbf{x}_1\mathbf{x}_2 + \mathbf{y}_1\mathbf{y}_2 + \mathbf{z}_1\mathbf{z}_2 \qquad \text{(in 3D)}$$

$$\vec{a} = x_1\vec{i} + y_1\vec{j} + z_1\vec{k}$$

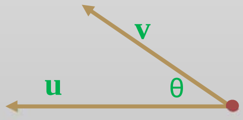
$$\vec{b} = x_2\vec{i} + y_2\vec{j} + z_2\vec{k}$$

$$\vec{a} \bullet \vec{b} = \left(x_1\vec{i} + y_1\vec{j} + z_1\vec{k}\right) \bullet \left(x_2\vec{i} + y_2\vec{j} + z_2\vec{k}\right)$$

$$= x_1x_2\vec{i}\bullet\vec{i} + x_1y_2\vec{i}\bullet\vec{j} + x_1z_2\vec{i}\bullet\vec{k} + y_1x_2\vec{j}\bullet\vec{i} + y_1y_2\vec{j}\bullet\vec{j} + y_1z_2\vec{j}\bullet\vec{k} + z_1x_2\vec{k}\bullet\vec{i} + z_1y_2\vec{k}\bullet\vec{j} + z_1z_2\vec{k}\bullet\vec{k}$$

$$\vec{i}\bullet\vec{i} = \vec{j}\bullet\vec{j} = \vec{k}\bullet\vec{k} = 1 \qquad \vec{i}\bullet\vec{j} = \vec{i}\bullet\vec{k} = \vec{j}\bullet\vec{k} = 0$$

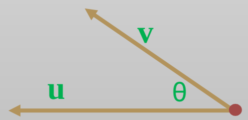
$$\vec{a} \bullet \vec{b} = x_1x_2 + y_1y_2 + z_1z_2$$



Dot Product

Useful for many purposes

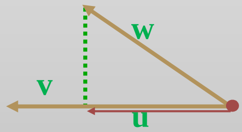
- Computing the length (Euclidean Norm) of a vector:
  - length(*v*) = ||**v**|| = *sqrt*(*v* • *v*)
- *Normalizing* a vector, making it unit-length: **v = v / ||v||**
- Computing the angle between two vectors:
  - $\boldsymbol{u} \bullet \boldsymbol{v} = |\boldsymbol{u}|\ |\boldsymbol{v}|\ cos(\theta)$
- Checking two vectors for orthogonality
  - $\boldsymbol{u} \bullet \boldsymbol{v} = 0.0$



Dot Product

Projecting one vector onto another

- If **v** is a unit vector and we have another vector, **w**
- We can project **w** perpendicularly onto **v**



$$\|u\| = \|w\|\cos(\theta) = \|w\|\left(\frac{v \bullet w}{\|v\|\|w\|}\right) = v \bullet w$$

- And the result, **u**, has length **w • v**

Dot Product

Is commutative

- u • v = v • u

Is distributive with respect to addition

- u • (v + w) = u • v + u • w

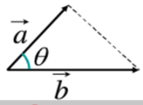
Cross Product

The cross product or vector product of two vectors is a vector:

$$\mathbf{v}_1 \times \mathbf{v}_2 = (y_1 z_2 - y_2 z_1)u_x + (x_2 z_1 - x_1 z_2)u_y + (x_1 y_2 - x_2 y_1)u_z$$

$$= \begin{vmatrix} u_x & u_y & u_z \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \end{vmatrix}_{\text{determinant}}$$

$S = \frac{1}{2}|\vec{a} \times \vec{b}|$



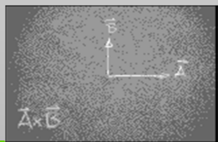
$\|v_1 \times v_2\| = 2*$ Area of triangle

The cross product of two vectors is orthogonal to both

Right-hand rule dictates direction of cross product

Cross Product Right Hand Rule

- See: http://www.phy.syr.edu/courses/video/RightHandRule/index2.html
- Orient your right hand such that your palm is at the beginning of A and your fingers point in the direction of A
- Twist your hand about the A-axis such that B extends perpendicularly from your palm
- As you curl your fingers to make a fist, your thumb will point in the direction of the cross product



Determinant

Second order determinant

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}$$

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$$

Thrid order determinant

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} - a_{13}a_{22}a_{31}$$

Determinant and matrix

Determinant: is a function, and can computed a result

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}$$

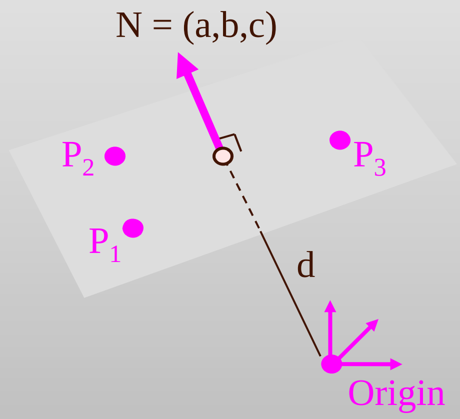
Matrix: is a list of coefficients

$$\mathbf{AB} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

3D Plane

A linear combination of three points

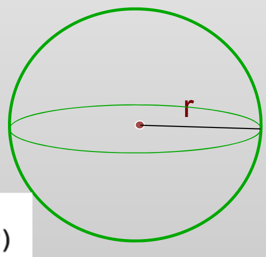
- Implicit representation:
  - $a(x-x_0)+b(y-y_0)+c(z-z_0)=0$
  - $ax + by + cz + d = 0$, or
  - $P \cdot N + d = 0$
- N is the plane “normal”
  - *Unit-length vector*
  - *Perpendicular to plane*

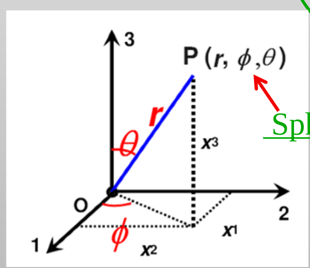


3D Sphere

All points at distance “r” from point “(c_x, c_y, c_z)”

- Implicit representation:
  - $(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2$
- Parametric representation:
  - $x = r \cos(\phi) \cos(\Theta)$
  - $y = r \cos(\phi) \sin(\Theta)$
  - $z = r \sin(\phi)$



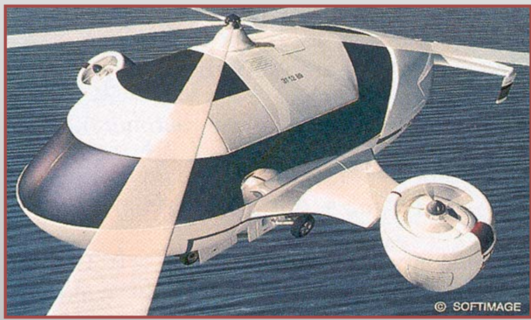


Spherical Coordinates

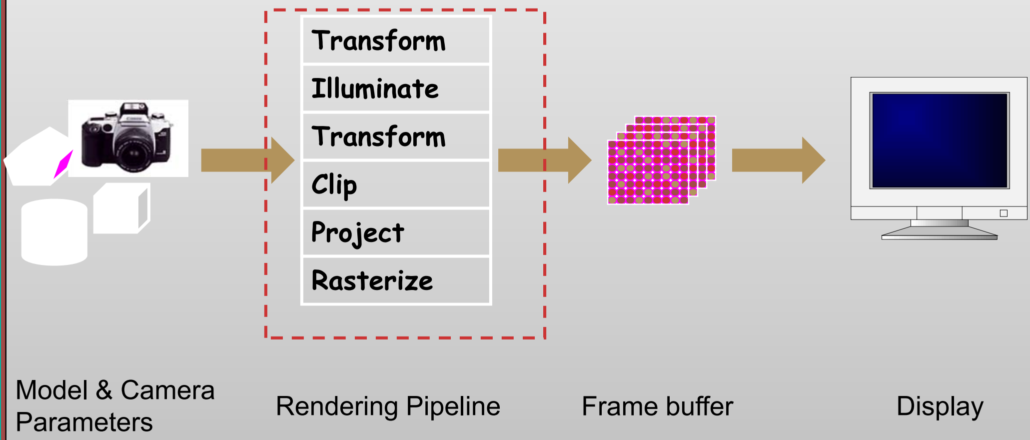
3D Geometric Primitives

More detail on 3D modeling later in course

- Point
- Line segment
- Polygon
- Polyhedron
- Curved surface
- Solid object
- etc.



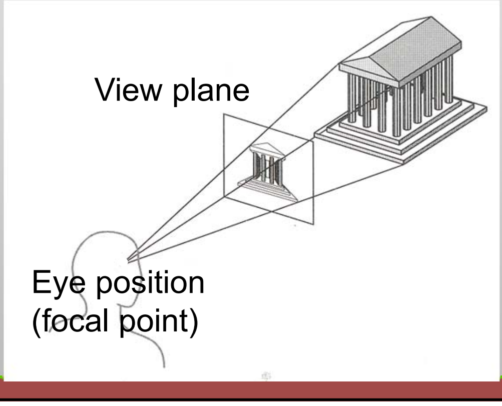
Rendering 3D Scenes



Camera Models

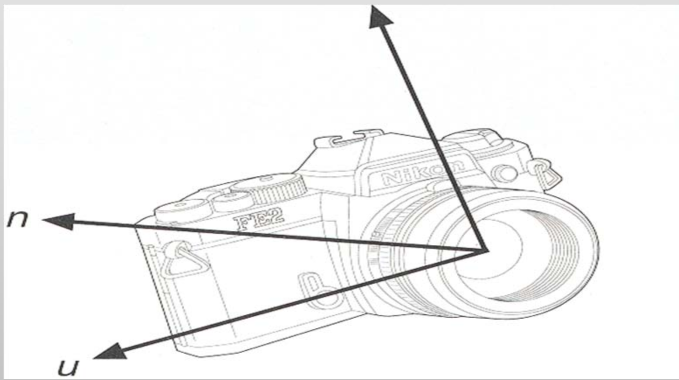
The most common model is pin-hole camera

- All captured light rays arrive along paths toward focal point without lens distortion (everything is in focus)
- Sensor response proportional to radiance



Camera Parameters

What are the parameters of a camera?



Camera Parameters

Position

- Eye position (px, py, pz)

Orientation

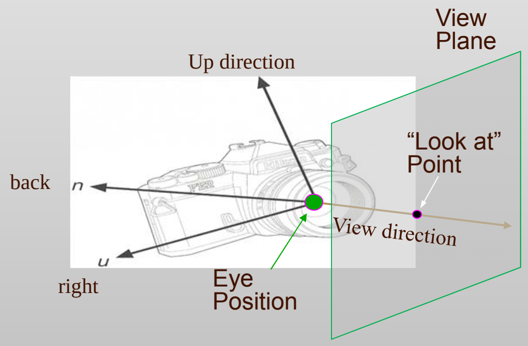
- View direction (dx, dy, dz)
- Up direction (ux, uy, uz)

Aperture

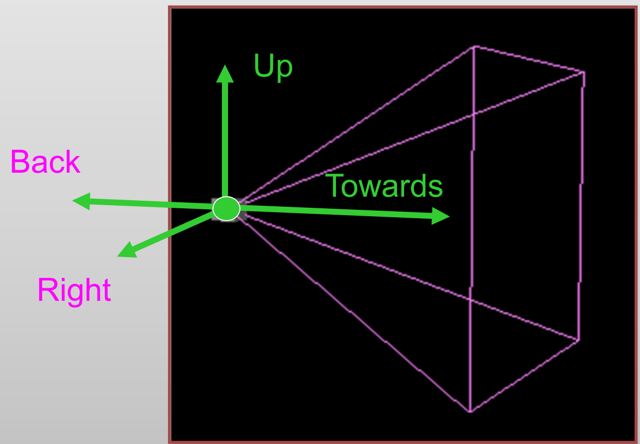
- Field of view (xfov, yfov)

Film plane

- "Look at" point
- View plane normal

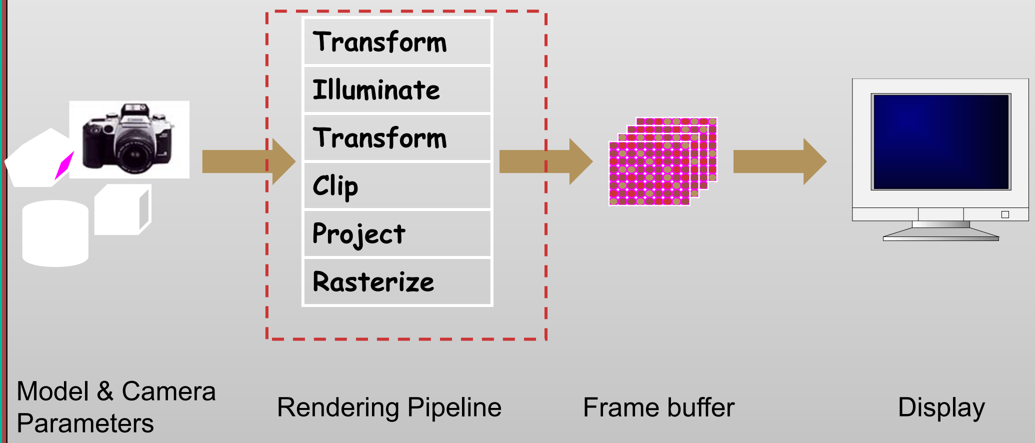


Moving the camera



View Frustum

Rendering 3D Scenes



Rendering: Transformations

We've learned about transformations

But they are used in three ways:

- *Modeling transforms (decide the object)*
- *Viewing transforms (Move the camera)*
- *Projection transforms (Change the type of camera)*

The Rendering Pipeline: 3-D

Scene graph
Object geometry

Modeling Transforms

Lighting Calculations

Viewing Transform

Clipping

Projection Transform

The Rendering Pipeline: 3-D

Scene graph
Object geometry

Modeling Transforms

Lighting Calculations

Viewing Transform

Clipping

Projection Transform

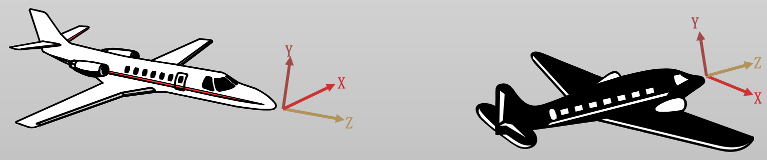
Result:

- All vertices of scene in shared 3-D "world" coordinate system

Rendering: Transformations

Modeling transforms

- Size, place, scale, and rotate objects and parts of the model with regard to each other
- Object coordinates -> world coordinates



The Rendering Pipeline: 3-D

Scene graph
Object geometry

Modeling Transforms

Lighting Calculations

Viewing Transform

Clipping

Projection Transform

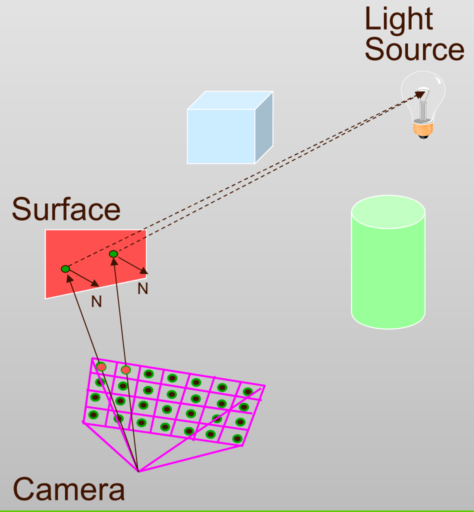
Result:

•All geometric primitives are illuminated

Lighting Simulation

Lighting parameters

- Light source emission
- Surface reflectance
- Atmospheric attenuation
- Camera response



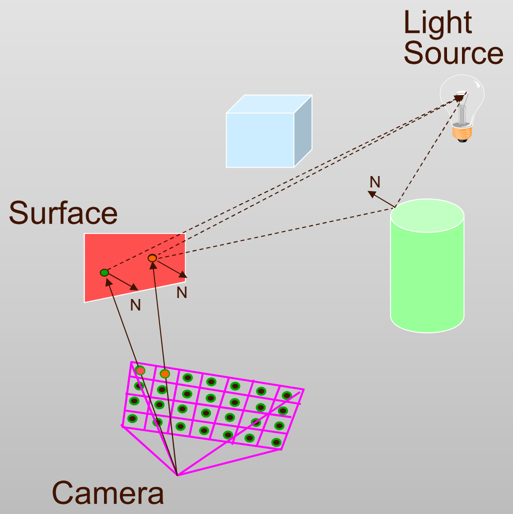
Lighting Simulation

Local illumination

- Ray casting
- Polygon shading

Global illumination

- Ray tracing
- Monte Carlo methods
- Radiosity methods



The Rendering Pipeline: 3-D

Scene graph
Object geometry

Modeling Transforms

Lighting Calculations

Viewing Transform

Clipping

Projection Transform

Result:

- Scene vertices in 3-D “view” or “camera” coordinate system

Rendering: Transformations

Viewing transform

- Rotate & translate the world to lie directly in front of the camera
  - *Typically place camera at origin*
  - *Typically looking down -Z axis*
- **World coordinates -> view coordinates**

Rendering: Transformations

Scene graph
Object geometry

Modeling Transforms

Lighting Calculations

Viewing Transform

Clipping

Projection Transform

Result:

• Remove geometry that is out of view

Rendering: Transformations

Scene graph
Object geometry

Modeling Transforms

Lighting Calculations

Viewing Transform

Clipping

Projection Transform

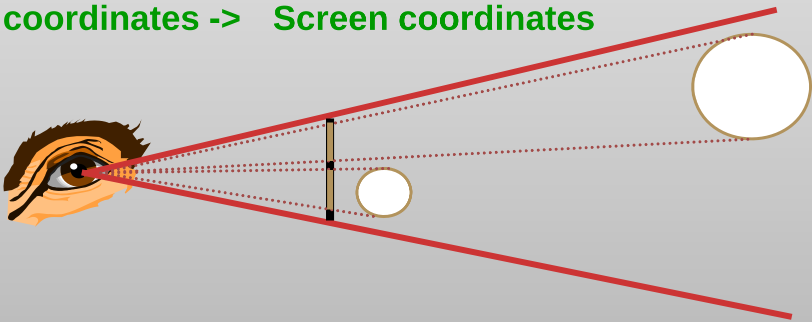
Result:

• 2-D screen coordinates of clipped vertices

Rendering: Transformations

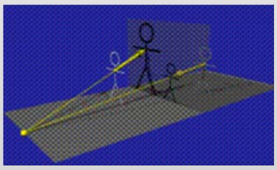
Projection transform

- Apply perspective foreshortening
  - *Distant = small: the* pinhole camera *model*
- **View coordinates -> Screen coordinates**

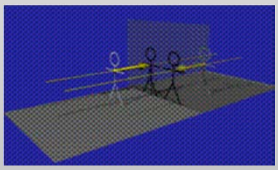


Rendering: Transformations

Perspective Camera



Orthographic Camera



Projection Matrix

Before, We talked about geometric transforms, focusing on modeling transforms

- Ex: translation, rotation, scale
- These are encapsulated in the OpenGL ***modelview matrix***

Projection is also represented as a matrix

Next few slides: representing orthographic and perspective projection with the projection matrix

Projection Matrix

In OpenGL, two matrix stack to operate the transformation

glMatrixMode (GL_PROJECTION | GL_MODELVIEW)

Use glPushMatrix(); glPopMatrix() to separate a individual transformation, like /begin, /end

- glPushMatrix(): copy the top matrix and push into the stack
- glPopMatrix(): pop up the top matrix, recover to the previous one

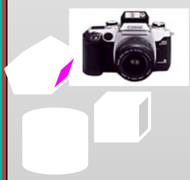
Projection Matrix

```
void display(void)
{
  glClear (GL_COLOR_BUFFER_BIT);
  glPushMatrix();
  glTranslatef (-1.0, 0.0, 0.0);
  glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
  glTranslatef (1.0, 0.0, 0.0);
  glPushMatrix();
  glScalef (2.0, 0.4, 1.0);
  glutWireCube (1.0);
  glPopMatrix();
```

Projection Matrix

```
void display(void)
{
 glTranslatef (1.0, 0.0, 0.0);
  glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
  glTranslatef (1.0, 0.0, 0.0);
  glPushMatrix();
  glScalef (2.0, 0.4, 1.0);
  glutWireCube (1.0);
  glPopMatrix();
  glPopMatrix();
  glutSwapBuffers();
```

Rendering 3D Scenes



Transform

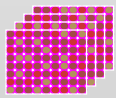
Illuminate

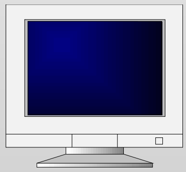
Transform

Clip

Project

Rasterize





Model & Camera Parameters

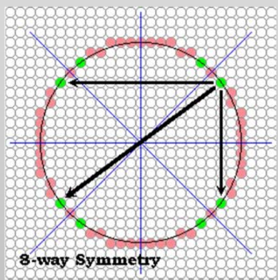
Rendering Pipeline

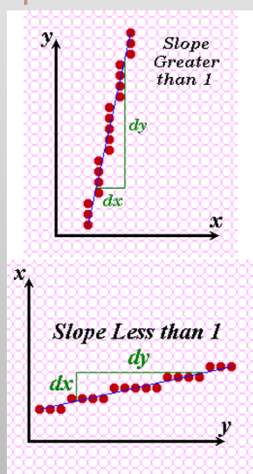
Frame buffer

Display

Rasterize

Convert screen coordinates to pixel colors





Summary

Geometric primitives

- Points, vectors

Operators on these primitives

- Dot product, cross product, norm

The rendering pipeline

- Move models, illuminate, move camera, clip, project to display, rasterize