Three Main Themes of Computer Graphics

Modeling

- How do we represent (or model) 3-D objects?
- How do we construct models for specific objects?

Animation

- How do we represent the motion of objects?
- How do we give animators control of this motion?

Rendering

- How do we simulate the formation of images?
- How do we simulate the real-world behavior of light?

Modeling

How do we represent objects/environments?

- shape — the geometry of the object
- appearance — emission, reflection, and transmission of light

How do we construct these models?

- manual description (e.g., write down a formula)
- interactive manipulation
- procedurally — write a generating program (e.g., fractals)
- scan a real object
  - laser scanners,
  - computer vision, …

Animation

How do we represent the motion of objects?

- positions, view angles, etc. as functions of time

How do we control/specify this motion?

- generate poses by hand
- behavioral simulation
- physical simulation
- motion capture

Rendering

How do we simulate the formation of images?

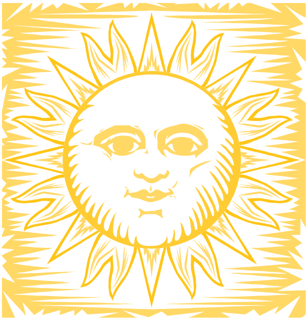
- incoming light is focused by a lens
- light energy “exposes” a light-sensitive “film”
- represent images as discrete 2-D arrays of pixels $I(x,y)$
- need suitable representation of a camera

How do we simulate the behavior of light?

- consider light as photons (light particles)
- trace straight-line motion of photons
- must model interactions when light hits surfaces
  - refraction, reflection, etc.

Image Formation at a Glance

Exposure



Reflection

Illumination

This is light transport.

Illumination is generated at light sources, propagates thru world.

Interacts with objects in scene.

Absorption

Image Formation

Elements of image formation:

- Illumination sources
- Objects
- Viewer (e.g., camera and eye)
- Attributes of materials

How can we design graphics hardware and software to mimic the image formation process?

Image Formation

Modeling the flow of light

- Light has a dual nature
- Interaction with surface
- Composition of colors

Human perception

- Cone and rods

Simple camera

- Pinhole camera
- Camera with refractive lenses

Raster image representation

- Each image is represented by a rectangular grid of pixels storing color values

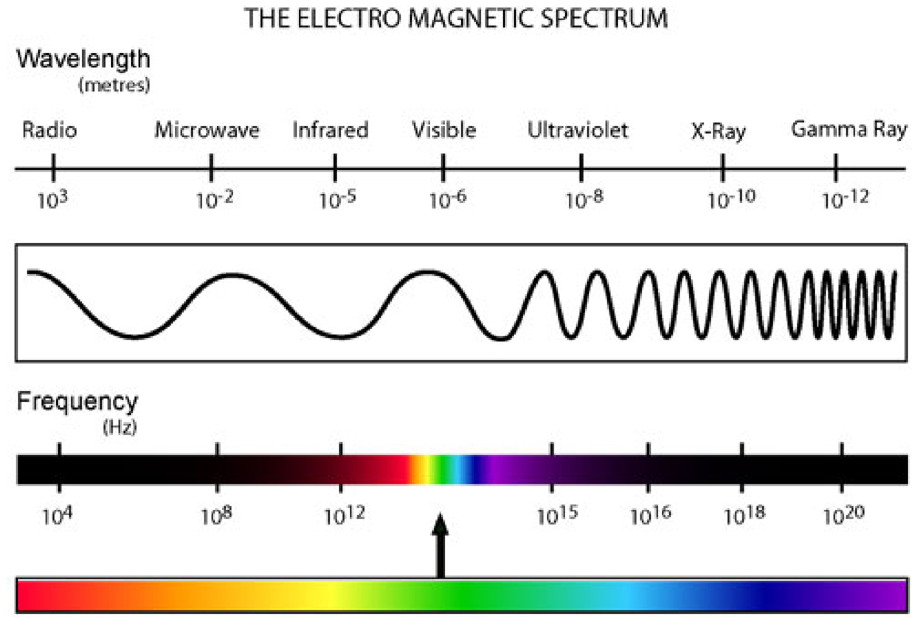
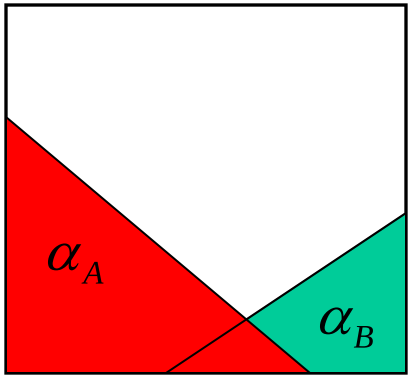


Image Compositing

Introduce a new alpha channel in addition to RGB channels

- the α value of a pixel indicates its transparency
  - if $\alpha=0$, pixel is totally transparent
  - if $\alpha=1$, pixel is totally opaque

$$P = \begin{bmatrix} r_p \\ g_p \\ b_p \\ \alpha_p \end{bmatrix} \Longrightarrow P' = \begin{bmatrix} \alpha_p r_p \\ \alpha_p g_p \\ \alpha_p b_p \\ \alpha_p \end{bmatrix}$$



Given images *A* & *B*, we can compute *C* = *A* over *B*

$$C_{rgb} = \alpha_A A_{rgb} + (1 - \alpha_A)\alpha_B B_{rgb}$$

- if we pre-multiply α values, this simplifies to

$$C' = A' + (1 - \alpha_A)B'$$

Pipeline Architecture of A Graphics System

All steps can be implemented in hardware on the graphics card

- **Vertex processing**
  - Geometrical transformation
  - Color transformations
- **Primitive assembly**
  - Vertices must be collected into geometric objects before clipping and rasterization
- **Clipping**
- **Rasterization**
  - produces a set of fragments for each object
  - Fragments are “potential pixels” in frame buffer with color and depth
- **Fragment processing**
  - Determine colors

Introduction to OpenGL

OpenGL is an Application Programmer Interface (API) and a standard graphics library for 2-D & 3-D drawing

- maps fairly directly to graphics hardware
- doesn't address windows or input events (we'll use GLUT)
- platform-independent

OpenGL 3.1 Totally shader-based

- Each application must provide both a vertex and a fragment shader

OpenGL 4.1 and 4.2

- Add geometry shaders and tessellator

OpenGL Libraries

OpenGL core library

- Available when you install the graphics driver

OpenGL Utility Toolkit (GLUT/FreeGLUT)

- Provides functionality for all window systems

OpenGL Extension Libraries

- Links with window system
- OpenGL Extension Wrangler Library (GLEW)

Shader-based OpenGL

- Vertex shading stage: receiving and process primitives separately
  - E.g., specifying the colors and positions
- Tessellation shading stage: specifying a patch, i.e., an ordered list of vertices and generating a mesh of primitives
- Geometry shading stage: enabling multivertex access, changing primitive type
- Fragment shading stage: processing color and depth

GLSL

OpenGL Shading Language

Like a complete C program

Code sent to shaders as source code

Entry point is the main function main()

Need to compile, link and get information to shaders

Type Qualifier

Define and modify the behavior of variables

- **Storage qualifiers: where the data come from**
  - const: read-only, must be initialized when declared
  - in: vertex attributes or from the previous stage
  - out: output from the shader
  - uniform: a global variable shared between all the shader stages
  - buffer: share buffer with application (r/w)

  Copy in/out data

- **Layout qualifiers: the storage location**

- **Invariant/precise qualifiers: enforcing the reproducibility**

Vertex Shader

Basic task: Sending vertices positions to the rasterizer

Advanced tasks:

- Transformation
  - Projection
- Moving vertices
  - Morphing
  - Wave motion
  - Fractals
- Processing color

A Simple Vertex Shader: triangles.vert (Shreiner et al)

```
#version 430 core

in vec4 vPosition;

void main()

{

        gl_Position = vPosition;

}
```

Specify it is an input to the shader (`in`)

Global variable, copied from the application to the shader (`vPosition`)

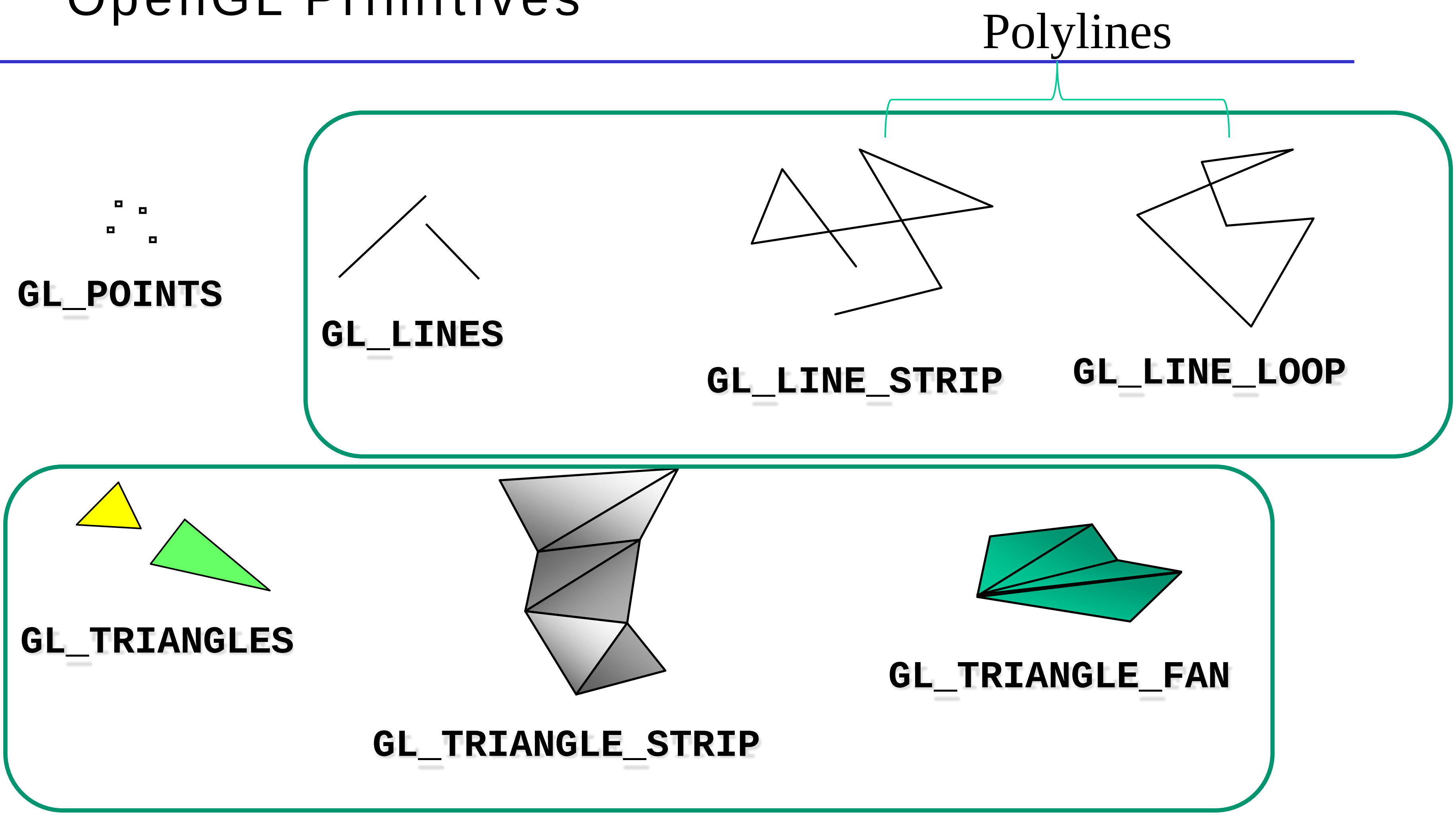
A built-in variable, passing data to the rasterizer (`gl_Position`)

Simple Fragment Program

```
#version 400

out vec4  fColor;

void main(void)

{

  fColor = vec4(1.0, 0.0, 0.0, 1.0);

}
```

Specify this is an output to the application

OpenGL Primitives

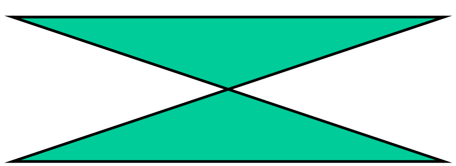


Polygon Issues

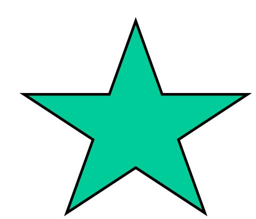
OpenGL only displays triangles

- Simple: edges cannot cross, i.e., only meet at the end points
- Convex: All points on line segment between two points in a polygon are also in the polygon
- Flat: all vertices are in the same plane

Application program must tessellate a polygon into triangles (triangulation)



nonsimple polygon



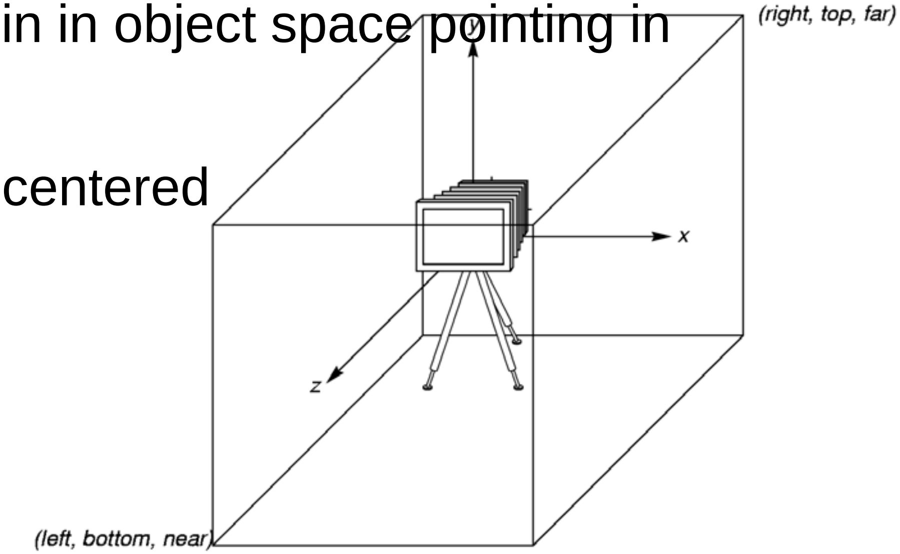
nonconvex polygon

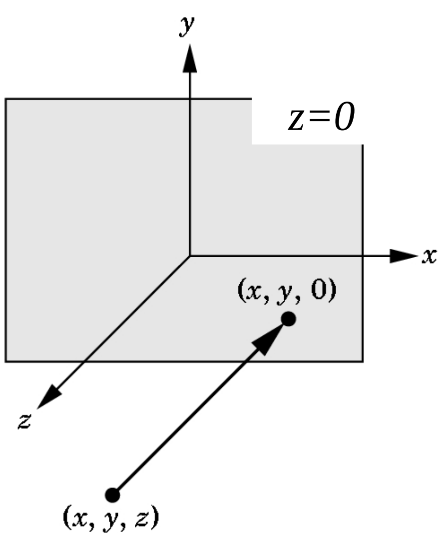
OpenGL Camera

OpenGL places a camera at the origin in object space pointing in the negative z direction

The default viewing volume is a box centered

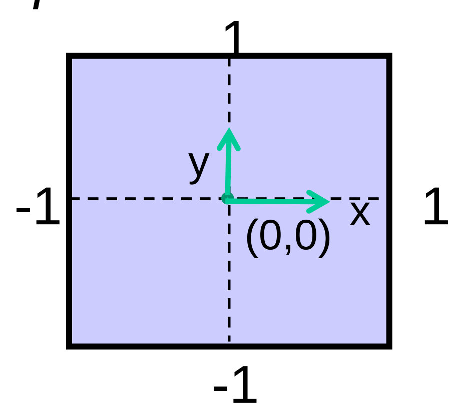
at the origin with sides of length 2





E. Angel and D. Shreiner:

OpenGL coordinates



Graphical Input

Devices can be described either by

- Physical properties
  - Mouse
  - Keyboard
  - Trackball
- Logical Properties
  - What is returned to program via API

Modes

- How and when input is obtained
  - Request mode, e.g., keyboard input
    - Input provided to program only when user triggers the device
    - Application and input cannot work at the same time
  - Event mode, e.g., mouse clicking
    - Each trigger generates an event whose measure is put in an event queue examined by the user program

Geometric Objects and Transformations

Described by a minimum set of primitives including

- Points
  - Associated with location
  - No size & shape
- Scalars
  - have no geometric properties
- Vectors,
  - a quantity with two attributes: direction and magnitude
  - No position information

Operations allowed between points and vectors

- Point-point subtraction yields a vector
- Point-vector addition yields a new point

Spaces

(Linear) vector space: scalars and vectors

- Mathematical system for manipulating vectors
- Operations including scalar-vector multiplication and vector-vector addition

Affine space: vector space + points

- Operations including vector-vector addition, scalar-vector multiplication, scalar-scalar operations, point-vector addition, point-point addition, and scalar-point multiplication

Euclidean space: vector space + distance

- Operations including vector-vector addition, scalar-vector multiplication, scalar-scalar operations, and inner (dot) products

Dimension, Basis, and Representation

Dimension of the space: the maximum number of linearly independent vectors

In an *n*-dimensional space, any set of n linearly independent vectors form a *basis* for the space

Given a basis $v_1, v_2, \ldots, v_n$, any vector v can be written as

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$$

where the $\{\alpha_i\}$ are unique and form the representation of the vector

Affine Spaces

Point + a vector space

Operations

- Vector-vector addition
- Scalar-vector multiplication
- Scalar-scalar operations
- Point-vector addition
- Point-point addition
- Scalar-Point multiplication

} Affine sum

Lines and Rays

The parametric form of line

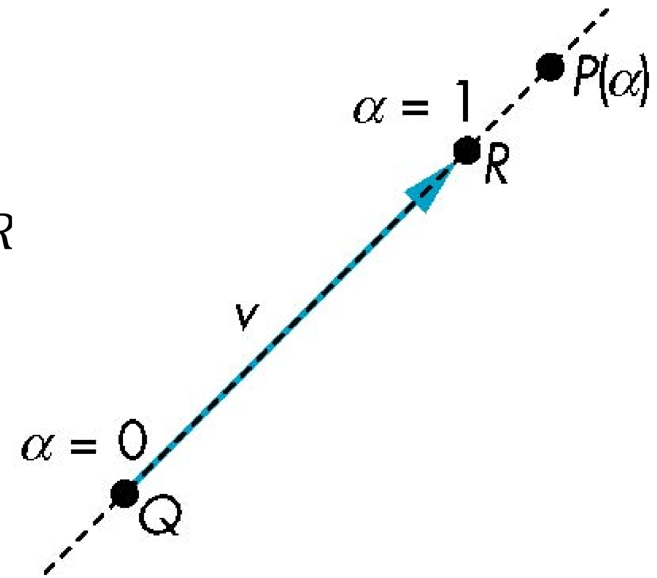
- $P(\alpha)=P_0 + \alpha\, \mathbf{d}$
- Set of all points that pass through P_0 in the direction of the vector $\mathbf{d}$
- If α >= 0, then $P(\alpha)$ is the *ray* leaving P_0 in the direction $\mathbf{d}$

Line segments

If we use two points to define v, then

$$P(\boldsymbol{\alpha}) = Q + \boldsymbol{\alpha} v = Q + \boldsymbol{\alpha}(R - Q) = \boldsymbol{\alpha} R$$

For $0 \leq \alpha \leq 1$ we get all the points on the *line segment* joining R and Q



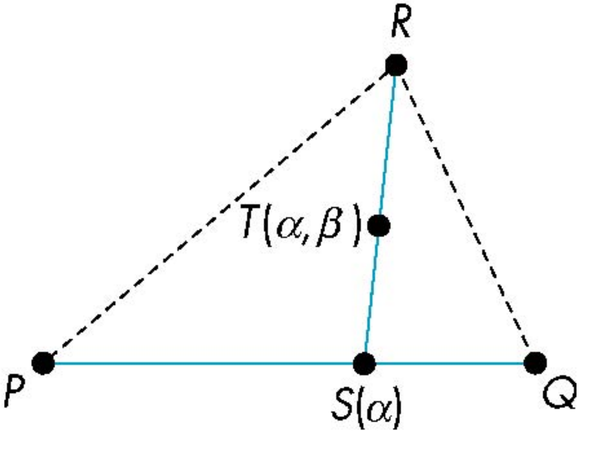
Planes

A plane can be defined by three non-collinear points

$$T(\alpha, \beta) = \beta[\alpha P + (1-\alpha)Q] + (1-\beta)\mathrm{R},\ 0 \le \alpha, \beta \le 1$$

A plane can be defined by a point and two vectors

$$T(\alpha', \beta') = P + \alpha' u + \beta' v$$ → Parametric form of planes

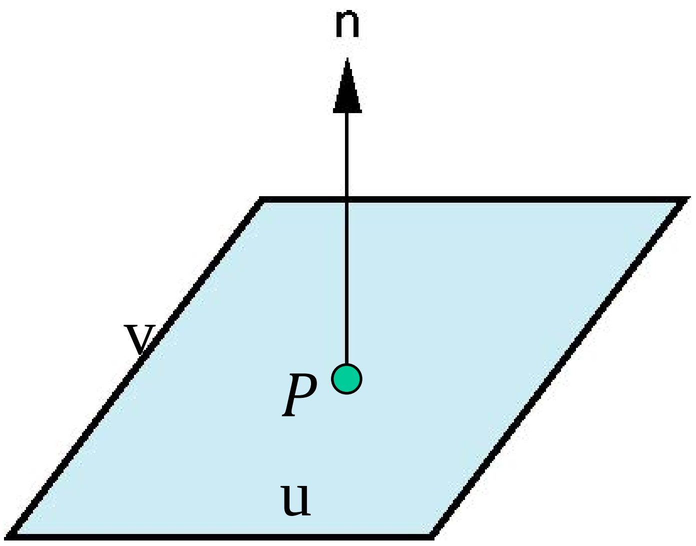


Planes

Every plane has a vector n normal (perpendicular) to it

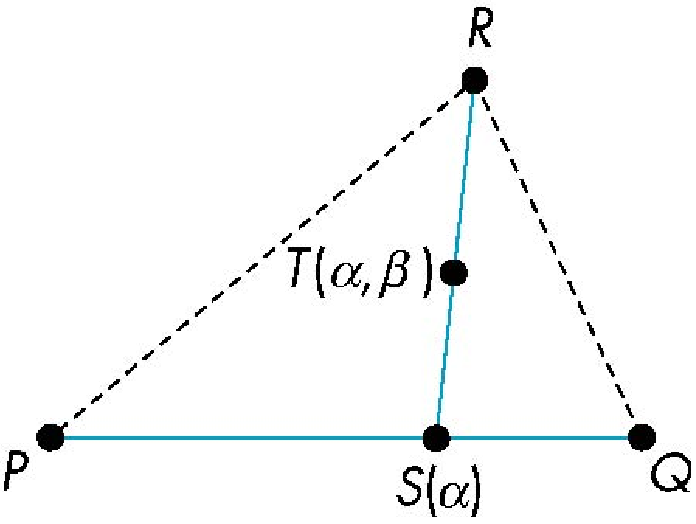
A plane can be represented by its point-normal form

$$(T - P) \cdot n = 0$$



Triangles

A triangle can be defined by an affine sum of three vertices



$$T(\alpha, \beta, \gamma') = \alpha P + \beta \quad Q + \gamma \quad R \quad \text{where} \quad \begin{array}{c} 0 \le \alpha \quad , \beta \quad , \gamma \quad \le 1 \\ \alpha \quad + \beta \quad + \gamma \quad = 1 \end{array}$$

Coordinate Systems

Consider a basis $v_1, v_2, \ldots, v_n$ of $\mathcal{R}^n$

A vector in $\mathcal{R}^n$ is written $v = \alpha_1 v_1 + \alpha_2 v_2 + \cdots + \alpha_n v_n$

The list of scalars $\{\alpha_1, \alpha_2, \ldots . \alpha_n\}$ is the *representation* of v with respect to the given basis

We can write the representation as a row or column array of scalars

$$\mathbf{a} = [\alpha_1 \alpha_2 \cdots \alpha_n]^T = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \cdot \\ \alpha_n \end{bmatrix}$$

Frame of Reference

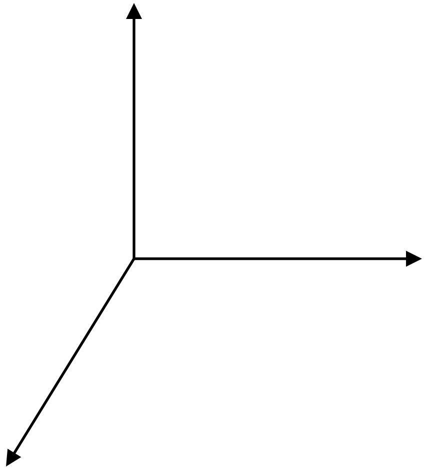
A coordinate system is insufficient to represent points

Need a frame of reference to relate points and objects to our physical world.

Adding a reference point (origin) to a coordinate system

→ **Frame defined in affine space**

- Frames used in graphics
  - World frame
  - Camera frame
  - Image frame



Representation in a Frame

Frame determined by (P_0, v_1, v_2, v_3)

Within this frame, every vector can be written as

$$v=\alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n$$

Every point can be written as

$$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_n v_n$$

Representation in a Frame- Homogeneous Coordinates

Frame determined by (P_0, v_1, v_2, v_3)

Define $0 \bullet P = 0$ and $1 \bullet P = P$, then

- **every vector can be written as**

$$v = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 = [\alpha_1\ \alpha_2\ \alpha_3\ 0]\ [v_1\ v_2\ v_3\ P_0]^T$$

- **every point can be written as**

$$P = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3 = [\beta_1\ \beta_2\ \beta_3\ 1]\ [v_1\ v_2\ v_3\ P_0]^T$$

Thus we obtain the four-dimensional *homogeneous coordinate* representation

$$v = [\alpha_1\ \alpha_2\ \alpha_3\ 0]^T$$

$$p = [\beta_1\ \beta_2\ \beta_3\ 1]^T$$

Representing One Frame in Terms of the Other

$$\mathbf{u}_1 = \gamma_{11}\mathbf{v}_1+\gamma_{12}\mathbf{v}_2+\gamma_{13}\mathbf{v}_3$$
$$\mathbf{u}_2 = \gamma_{21}\mathbf{v}_1+\gamma_{22}\mathbf{v}_2+\gamma_{23}\mathbf{v}_3$$
$$\mathbf{u}_3 = \gamma_{31}\mathbf{v}_1+\gamma_{32}\mathbf{v}_2+\gamma_{33}\mathbf{v}_3$$
$$\mathbf{Q}_0 = \gamma_{41}\mathbf{v}_1+\gamma_{42}\mathbf{v}_2+\gamma_{43}\mathbf{v}_3 +\mathbf{P}_0$$

defining a 4 x 4 matrix

$$\mathbf{M} = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & 1 \end{bmatrix} \Rightarrow \quad [\mathbf{U} \quad \mathrm{Q}_0] = [\mathbf{V} \quad P_0]\mathbf{M}^T$$

Changing Representations

Any point or vector has a representation in a frame

$\mathbf{a}=[\alpha_1\ \alpha_2\ \ \alpha_3\ \alpha_4\ ]$ in the first frame
$\mathbf{b}=[\beta_1\ \beta_2\ \ \beta_3\ \beta_4\ ]$ in the second frame

where $\alpha_4 = \beta_4 = 1$ for points and $\alpha_4 = \beta_4 = 0$ for vectors

We can change the representation from one frame to the other as

$$\mathbf{a}=\mathbf{M}^T\mathbf{b} \quad \text{and} \quad \mathbf{b}=(\mathbf{M}^T)^{-1}\mathbf{a}$$

The matrix $\mathbf{M}$ is 4 x 4 and specifies an affine transformation in homogeneous coordinates

Affine Transformations

Line preserving

Characteristic of many physically important transformations

- Translation
- Rotation
- Scaling
- Shearing

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{S} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{H} = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

General rotation about the origin

General rotation about an arbitrary vector

Rotation

$$\mathbf{R}_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma & 0 \\ 0 & \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_y(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

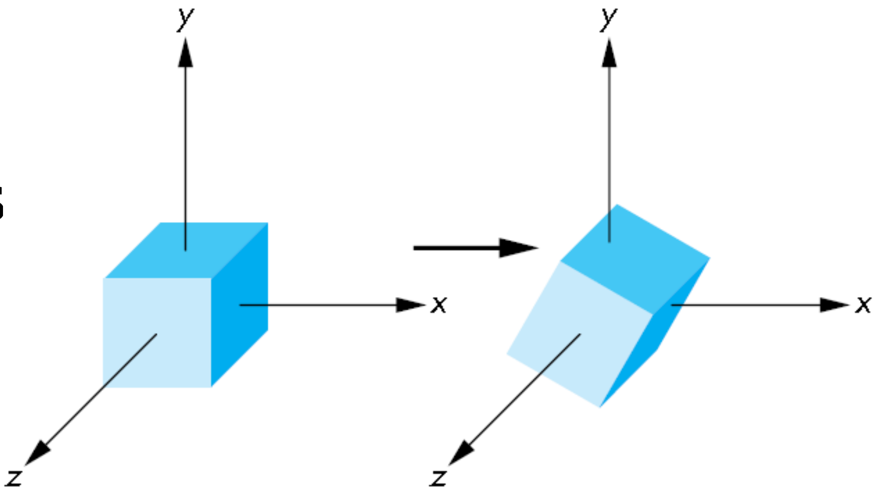
$$\mathbf{R}_Z(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

General Rotation About the Origin

A general rotation about the origin can be decomposed into successive of rotations about the *x*, *y*, and *z* axes

$$\mathbf{R} = \mathbf{R}_z(\alpha)\,\mathbf{R}_y(\beta)\,\mathbf{R}_x(\gamma)$$

α, β, γ are called the Euler angles



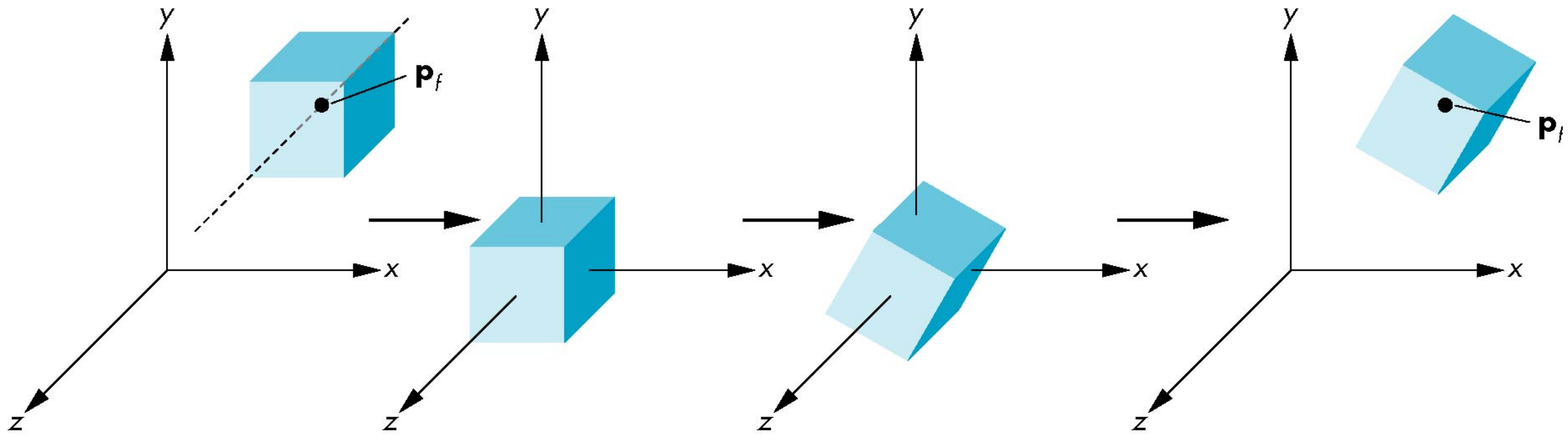
E. Angel and D. Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

Important:

- **R** is unique
- For a given order, rotations do not commute
- We can use rotations in another order but with different angles

Rotation About a Fixed Point Other than the Origin

- Move fixed point to origin
- Rotate around the origin
- Move fixed point back



General Rotation about An Arbitrary Vector

How do we achieve a rotation θ about an arbitrary vector?

Step 1: move the fixed point to the origin $\mathbf{M}_1 = \mathbf{T}(-\mathbf{p}_0)$

Step 2: align the arbitrary vector v=$\frac{\mathbf{p}_2-\mathbf{p}_1}{|\mathbf{p}_2-\mathbf{p}_1|}$ with the z-axis by two rotations about the x-axis and y-axis with θ_x and θ_y, respectively

$$\mathbf{M}_2 = \mathbf{R}_y(\theta_\mathrm{y})\mathbf{R}_x(\theta_\mathrm{x})$$

Step 3: rotate by q about the z-axis

$$\mathbf{M}_3 = \mathbf{R}_z(\theta)$$

Step 4: undo the two rotations for aligning z-axis

$$\mathbf{M}_4 = \mathbf{R}_x(-\theta_\mathrm{x})\mathbf{R}_y(-\theta_\mathrm{y})$$

Step 5: move the fixed point back

$$\mathbf{M}_5 = \mathbf{T}(\mathbf{p}_0)$$

Two Important Transformations in OpenGL

Object (or model) coordinates

World coordinates

Eye (or camera) coordinates

Clip coordinates

Normalized device coordinates

Window (or screen) coordinates

Model-view transformation

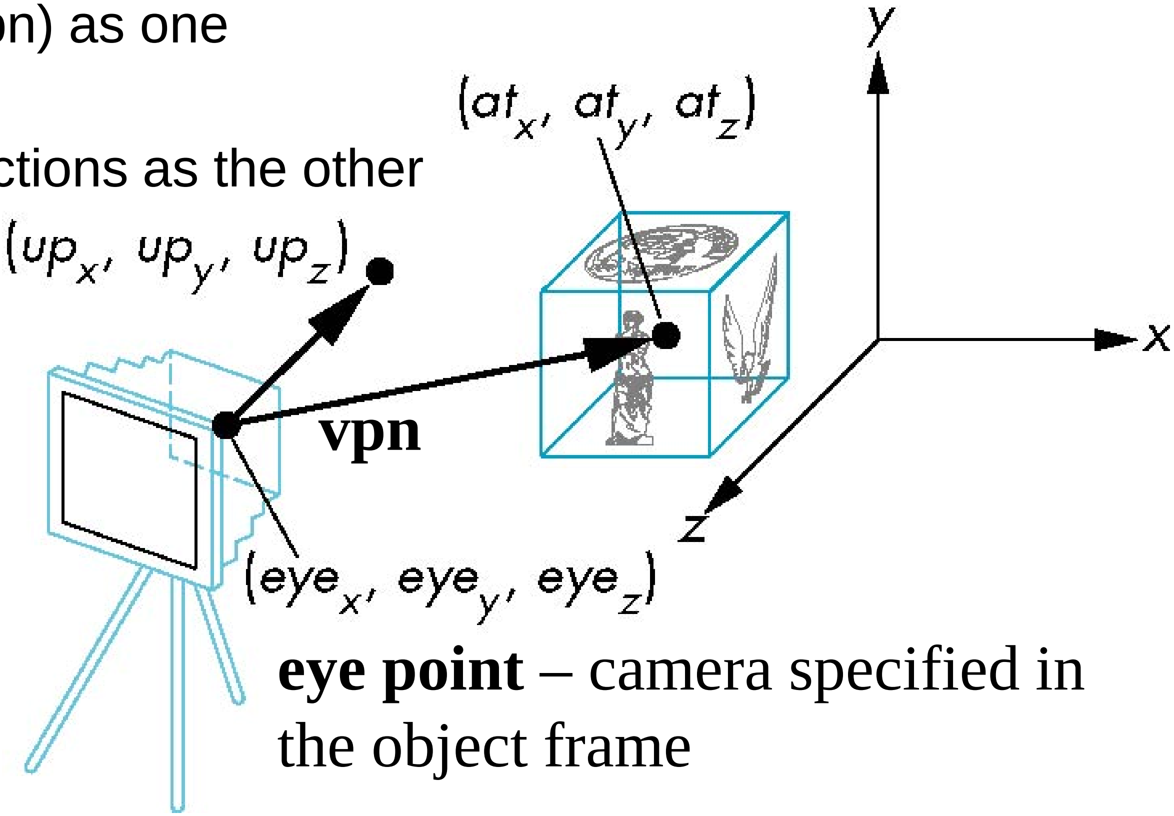
Projection transformation

Model View Transformation

Objective: construct a new frame with

- the origin at the eye point,
- The view plane normal (vpn) as one coordinate direction
- Two other orthogonal directions as the other two coordinate directions

at point – the point (e.g., the object center) the camera looks at



eye point – camera specified in the object frame

E. Angel and D. Shreiner

LookAt(eye, at, up)

$$\mathbf{vpn} = \mathbf{a} - \mathbf{e},$$

$$\mathbf{n} = \frac{\mathbf{vpn}}{|\mathbf{vpn}|}$$

$$\mathbf{u} = \frac{\mathbf{v}_{\mathbf{up}} \times \mathbf{n}}{|\mathbf{v}_{\mathbf{up}} \times \mathbf{n}|}$$

$$\mathbf{v} = \frac{\mathbf{n} \times \mathbf{u}}{|\mathbf{n} \times \mathbf{u}|}$$

⇒

$$\mathbf{M} = \begin{bmatrix} -u_x & -u_y & -u_z & -\mathbf{u} \cdot \mathbf{vpn} \\ v_x & v_y & v_z & \mathbf{v} \cdot \mathbf{vpn} \\ -n_x & -n_y & -n_z & -\mathbf{n} \cdot \mathbf{vpn} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Three Basic Elements in Viewing

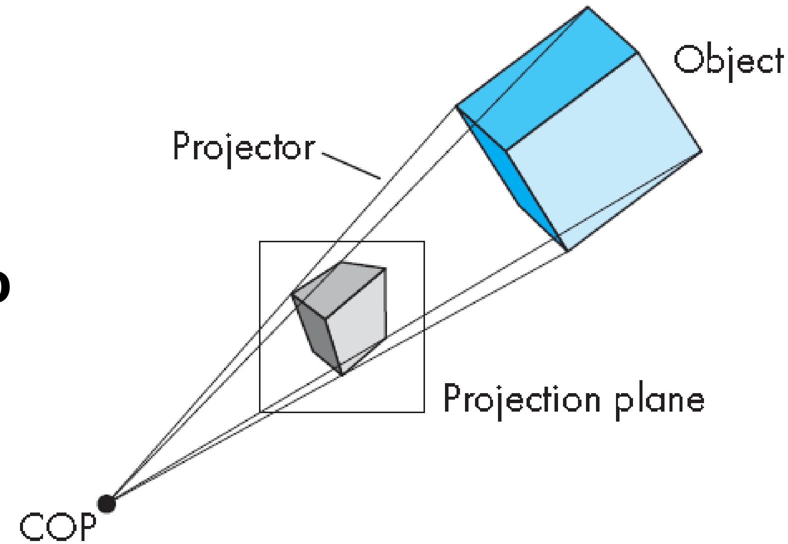
One or more objects

A viewer with a projection surface

- Planar geometric projections
- Nonplanar projections are needed for applications such as map construction

Projectors that go from the object(s) to the projection surface

- Perspective projection: projectors converge at a center of projection
- Parallel projection: projectors are parallel

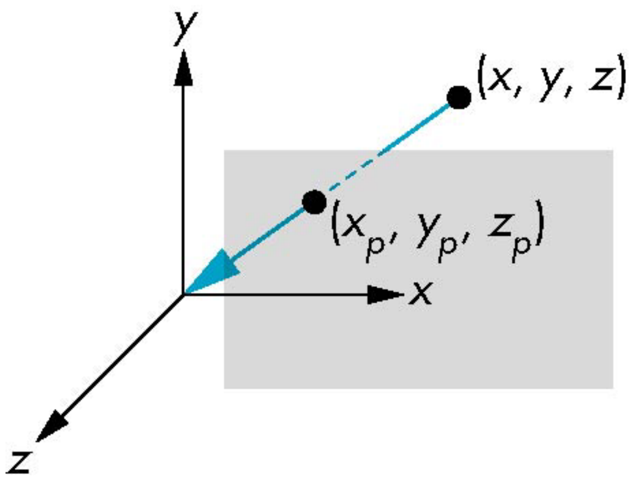


Parallel Projection

The default projection is **orthogonal (orthographic) projection**

For points within the view volume

$$x_p = x$$
$$y_p = y$$
$$z_p = 0$$



E. Angel and D. Shreiner

In homogeneous coordinates

$$\mathbf{p}_p = \mathbf{M}_{orth}\mathbf{p}$$

$$\mathbf{M}_{orth} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For general parallel projection

$$\mathbf{P} = \mathbf{M}_{orth}\ \mathbf{STH}(\theta,\phi)$$

Perspective Projection

Points project to points

Lines project to lines

Planes project to the whole or half image

- A plane may only has half of its area in the projection side

Scaling and foreshortening

Angles are not preserved

- Parallel lines may be not projected to parallel lines unless they are parallel to the image plane

Degenerate cases

- Line through focal point projects to a point.
- Plane through focal point projects to line

Simple Perspective with OpenGL

A point P (x, y ,z, 1) is projected to a new point Q

$$Q = \mathbf{M}P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ -z \end{bmatrix} = \begin{bmatrix} -x/z \\ -y/z \\ -1 \\ 1 \end{bmatrix}$$

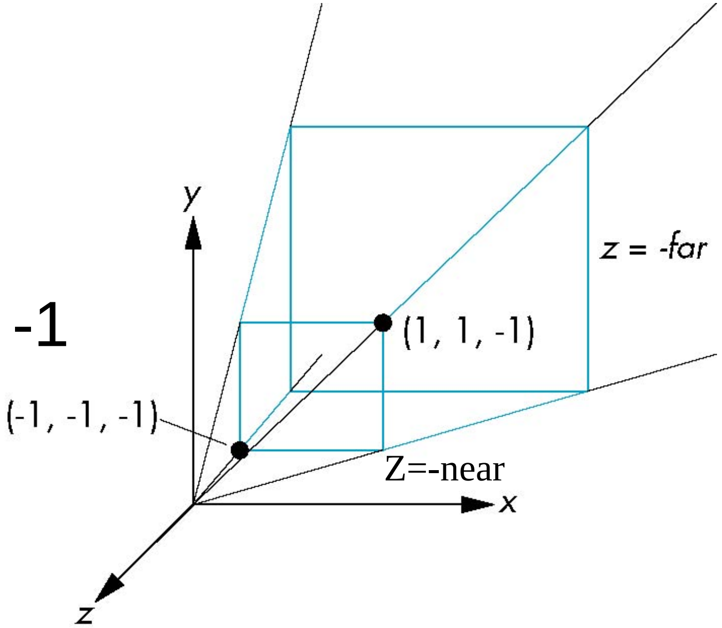
Simple Perspective with OpenGL

Consider a simple perspective with

- the COP at the origin,
- the near clipping plane at z = -1, and
- a 90 degree field of view determined by the planes $x = \pm z$, $y = \pm z$
- Perspective projection matrix is

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

where d = -1



E. Angel and D. Shreiner:

Perspective Projection and Normalization

The projection can be achieved by ***view normalization*** and an ***orthographic projection***

A point P=(x, y, z, 1) is project to a new point Q on the projection plane as

$$Q = \mathbf{M}_{\mathrm{orth}}\mathbf{N}\mathrm{P}$$

$$\mathbf{N} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

OpenGL Perspective

How do we handle the asymmetric frustum?

Convert the frustum to a symmetric one by performing a shear followed by a scaling to get the normalized perspective volume.

The final perspective matrix

$$\mathbf{M}_{\boldsymbol{p}} = \mathbf{NSH} = \begin{bmatrix} \frac{2near}{right-left} & 0 & \frac{left+right}{right-left} & 0 \\ 0 & \frac{2near}{top-bottom} & \frac{bottom+top}{top-bottom} & 0 \\ 0 & 0 & \frac{near+far}{near-far} & \frac{2near*far}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

A point P=(x, y, z, 1) is project to a new point Q on the projection plane as

$$Q = \mathbf{M}_{\text{orth}}\mathbf{M}_{\boldsymbol{p}}\text{P}$$