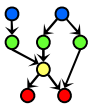


Lecture 5 of 41

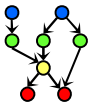
Viewing 3 of 4: Normalizing Transformation and Fixed-Function Graphics Pipeline

William H. Hsu
Department of Computing and Information Sciences, KSU

KSOL course pages: http://bit.ly/hGvXlH / http://bit.ly/eVizrE
Public mirror web site: http://www.kddresearch.org/Courses/CIS636
Instructor home page: http://www.cis.ksu.edu/~bhsu

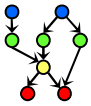
Readings:

Today: Section 2.3 (esp. 2.3.7), 2.6, 2.7, Eberly 2^e – see http://bit.ly/ieUq45
Next class:
This week: FVFH slides on Viewing



Lecture Outline

- Reading for Last Class: Chapters 2, 16, Eberly 2^e; Foley *et al.* Slides
- Reading for Today: Section 2.3 (esp. 2.3.7), 2.6, 2.7, Eberly 2^e
- Reading for Next Class: §2.5.1, 3.1 Eberly 2^e
- Last Time: Basic Viewing Principles
  - Projections: definitions, history
  - Perspective: optical principles, terminology
- Today: View Volume Specification and Viewing Transformation
  - View volumes: ideal *vs.* approximated
  - Frustum in computer graphics (CG)
  - Specifying view volume in CG: Look and Up vectors
  - Aspect ratio, view angle, front/back clipping planes
  - Focal length
  - Parallel (cuboid) view volume & perspective frustum
  - Normalizing transformation (NT) & viewing transformation (VT)
- Next Time: Fixed-Function Graphics Pipeline

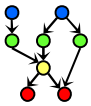


Where We Are

Lecture	Topic	Primary Source(s)
0	Course Overview	Chapter 1, Eberly 2e
1	**CG Basics: Transformation Matrices; Lab 0**	**Sections (§) 2.1, 2.2**
2	Viewing 1: Overview, Projections	§ 2.2.3 – 2.2.4, 2.8
3	Viewing 2: Viewing Transformation	§ 2.3 esp. 2.3.4; FVFH slides
4	**Lab 1a: Flash & OpenGL Basics**	**Ch. 2, 16[1], Angel *Primer***
5	Viewing 3: Graphics Pipeline	§ 2.3 esp. 2.3.7; 2.6, 2.7
6	Scan Conversion 1: Lines, Midpoint Algorithm	§ 2.5.1, 3.1, FVFH slides
7	**Viewing 4: Clipping & Culling; Lab 1b**	**§ 2.3.5, 2.4, 3.1.3**
8	Scan Conversion 2: Polygons, Clipping Intro	§ 2.4, 2.5 esp. 2.5.4, 3.1.6
9	Surface Detail 1: Illumination & Shading	§ 2.5, 2.6.1 – 2.6.2, 4.3.2, 20.2
10	**Lab 2a: Direct3D / DirectX Intro**	**§ 2.7, Direct3D handout**
11	Surface Detail 2: Textures; OpenGL Shading	§ 2.6.3, 20.3 – 20.4, *Primer*
12	Surface Detail 3: Mappings; OpenGL Textures	§ 20.5 – 20.13
13	**Surface Detail 4: Pixel/Vertex Shad.; Lab 2b**	**§ 3.1**
14	Surface Detail 5: Direct3D Shading; OGLSL	§ 3.2 – 3.4, Direct3D handout
15	Demos 1: CGA, Fun; Scene Graphs: State	§ 4.1 – 4.3, **CGA handout**
16	**Lab 3a: Shading & Transparency**	**§ 2.6, 20.1, *Primer***
17	**Animation 1: Basics, Keyframes; HW/Exam**	**§ 5.1 – 5.2**
	Exam 1 review; Hour Exam 1 (evening)	**Chapters 1 – 4, 20**
18	**Scene Graphs: Rendering; Lab 3b: Shader**	**§ 4.4 – 4.7**
19	**Demos 2: SFX; Skinning, Morphing**	**§ 5.3 – 5.5, CGA handout**
20	Demos 3: Surfaces; B-reps/Volume Graphics	§ 10.4, 12.7, **Mesh handout**

Lightly-shaded entries denote the due date of a written problem set; heavily-shaded entries, that of a machine problem (programming assignment); blue-shaded entries, that of a paper review; and the green-shaded entry, that of the term project.

Green, blue and red letters denote exam review, exam, and exam solution review dates.



Acknowledgements

Jim Foley
Professor, College of Computing & Stephen Fleming Chair in Telecommunications
Georgia Institute of Technology

James D. Foley
Georgia Tech
http://bit.ly/ajYf2Q

Andy van Dam
T. J. Watson University Professor of Technology and Education & Professor of Computer Science
Brown University

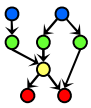
Andries van Dam
Brown University
http://www.cs.brown.edu/~avd/

Steve Feiner
Professor of Computer Science & Director, Computer Graphics and User Interfaces Laboratory
Columbia University

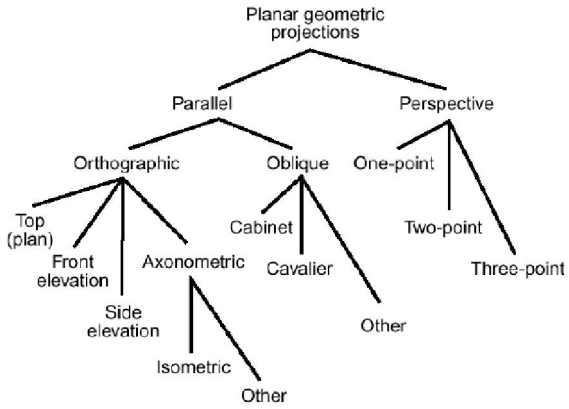
Steven K. Feiner
Columbia University
http://www.cs.columbia.edu/~feiner/

John F. Hughes
Associate Professor of Computer Science
Brown University

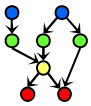
John F. Hughes
Brown University
http://www.cs.brown.edu/~jfh/



Review: Types of Projections



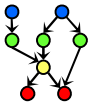
- Parallel projections used for engineering and architecture because they can be used for measurements
- Perspective imitates eyes or camera and looks more natural



Review: Synthetic Camera for 3-D Viewing

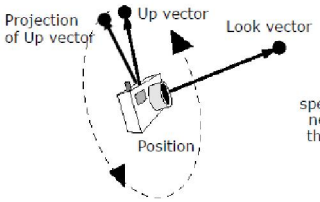
- The synthetic camera is the programmer's model to specify 3D view projection parameters to the computer
- General synthetic camera: each package has its own but they are all (nearly) equivalent. (PHIGS† Camera, *Computer Graphics: Principles and Practice*, ch. 6 and 7)
  - position of camera
  - orientation
  - field of view (wide angle, normal…)
  - depth of field (near distance, far distance)
  - focal distance
  - tilt of view/film plane (if not normal to view direction, produces oblique projections)
  - perspective or parallel projection? (camera near objects or an infinite distance away)
- CS123 uses a simpler, slightly less powerful model than the book's
  - omit tilt of view/film plane, focal distance (blurring)

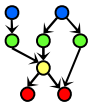
† This package is no longer in use but still has the most general synthetic camera model for perspective and parallel projections.



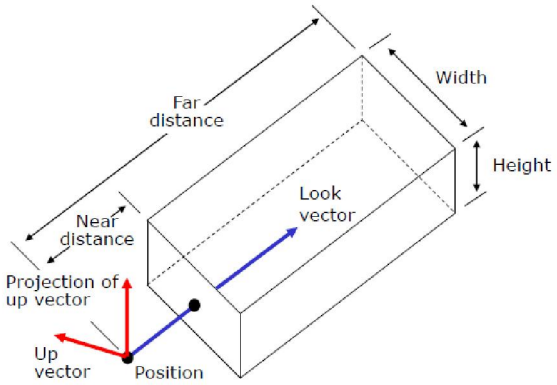
Review: Look and Up Vectors

- More concrete way to say the same thing as orientation
  - soon you'll learn how to express orientation in terms of *Look* and *Up* vectors
- *Look Vector*
  - the direction the camera is pointing
  - three degrees of freedom; can be any vector in 3-space
- *Up Vector*
  - determines how the camera is rotated around the *Look vector*
  - for example, whether you're holding the camera horizontally or vertically (or in between)
  - *Up vector* must not be parallel to *Look vector (Up vector* may be specified at an arbitrary angle to its *Look vector)*

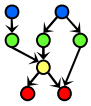




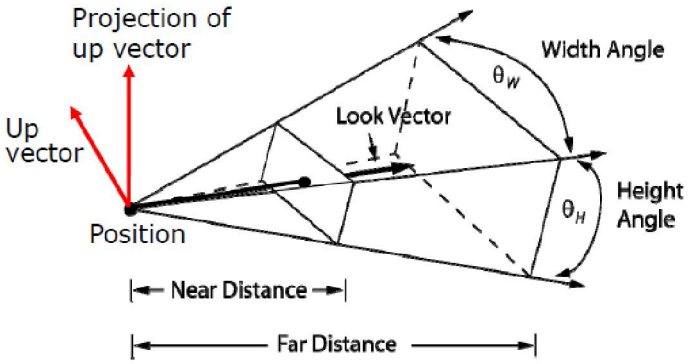
Review: View Volume for Parallel Projection



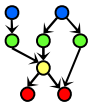
- Limiting view volume useful for eliminating extraneous objects
- Orthographic parallel projection has width and height view angles of zero



Review: View Volume for Perspective Projection



- Removes objects too far from *Position*, which otherwise would merge into "blobs"
- Removes objects too close to *Position* (would be excessively distorted)



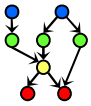
Viewing in Three Dimensions: Mathematics of Projections

- Reduce degrees of freedom; four steps to specifying view volume
  1. Position camera (and therefore its view/film plane)
  2. Orient camera to point at what you want to see
  3. Define field of view

     perspective: aspect ratio of film and angle of view: between wide angle, normal, and zoom

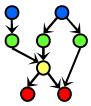
     parallel: width and height
  4. Choose perspective or parallel projection

(Optional: Specify a focal distance and exposure time. Our camera won't do this.)



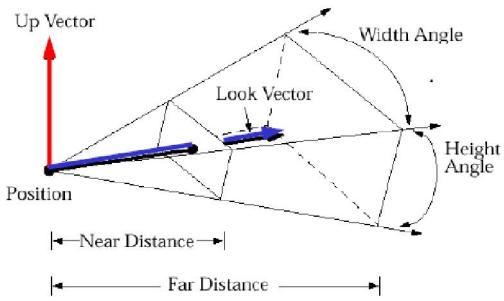
Stage 1: Specifying View Volume

- Lecture roadmap
  - mathematics of planar geometric projections
    - how to get from view specification to 2D image?
  - deriving 2D image from 3D view parameters is a hard problem
  - easier to take a picture from
    - **canonical view volume** (3D parallel projection cuboid)
    - **canonical view position** (camera at the origin, looking down the negative *z*-axis)
  - break into three stages:
    1. get parameters for view specification (covered in last lecture)
    2. transform from specified view volume into canonical view volume
    3. using canonical view, clip, project, and rasterize scene to make 2D image

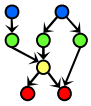


Examples of View Volume [1]: Perspective (Frustum)

- Perspective Projection: Truncated Pyramid – Frustum

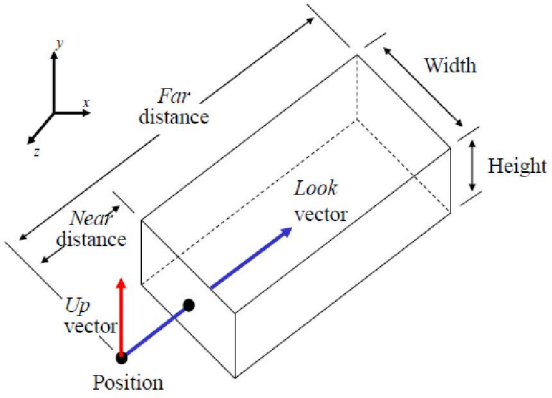


- *Look vector* is the center line of the pyramid, the vector that lines up with "the barrel of the lens"

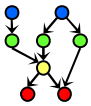


Examples of View Volume [2]: Parallel (Cuboid)

- Orthographic Parallel Projection: Truncated View Volume – Cuboid

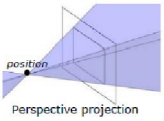


- Orthographic parallel projection has no view angle parameter



Viewing Transformation: World (x, y, z) to Camera (u, v, w)

- **Placement** of view volume (visible part of world) specified by camera's position and orientation
  - *Position* (a point)
  - *Look* and *Up vectors*
- **Shape** of view volume specified by
  - *horizontal* and *vertical view angles*
  - *front and back clipping planes*
- Perspective projection: projectors intersect at *Position*
- Parallel projection: projectors parallel to *Look vector*, but never intersect (or intersect at infinity)
- Coordinate Systems
  - **world coordinates** – standard right-handed *xyz* 3-space
  - **camera coordinates** – camera-space right handed coordinate system (*u*, *v*, *w*); origin at *Position* and axes rotated by orientation; used for transforming arbitrary view into canonical view



Perspective projection

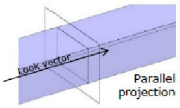
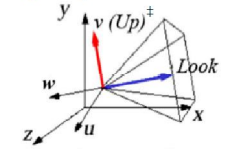
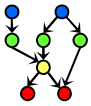


Image Source: Steve Marschner, Cornell



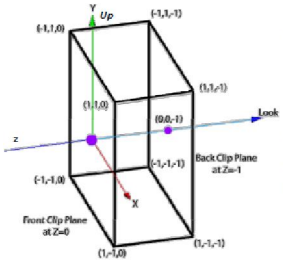
Arbitrary Perspective Frustum

‡ v isn't strictly the *Up vector* but the projection of *Up*



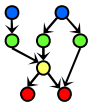
Arbitrary View Volume Too Complex

- We have specified arbitrary view with viewing parameters
- Problem: map arbitrary view specification to 2D image of scene. This is hard, both for clipping and for projection
- Solution: reduce to a simpler problem and solve
  - there is a view specification from which it is easy to take a picture. We'll call it the *canonical view*: from the origin, looking down the negative *z*-axis
  - think of the scene as lying behind a window and we're looking through that window



- parallel projection
- sits at origin:
  Position = (0, 0, 0)
- looks along negative *z*-axis:
  Look vector = (0, 0, –1)
- oriented upright:
  Up vector = (0, 1, 0)
- film plane extending from –1 to 1 in *x* and *y*

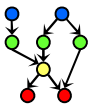
- Note: *Look vector* along negative, not positive, *z*-axis is arbitrary but makes math easier; ditto choosing $-1 \le x, y \le 1$



Stage 2: Normalizing to Canonical View Volume

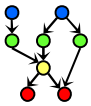
- Goal: transform arbitrary view and world to canonical view volume, maintaining relationship between view volume and world, then take picture
  - for parallel view volume, transformation is ***affine***† : made up of linear transformations (rotations and scales) and translation/shift
  - in case of a perspective view volume, it also contains a non-affine perspective transformation that turns a frustum into a parallel view volume, a cuboid
  - composite transformation to transform arbitrary view volume to canonical view volume, named the **normalizing transformation**, is still a 4x4 homogeneous matrix that typically has an inverse
  - easy to clip against this canonical view volume; clipping planes are axis-aligned!
  - projection using canonical view volume is even easier: just omit *z*-coordinate
  - for oblique parallel projection, a shearing transform is part of composite transform, to "de-oblique" view volume

† Affine transformations preserve parallelism but not lengths and angles. The perspective transformation is a type of non-affine transformation known as a *projective* transformation, which does not preserve parallelism



Viewing Transformation ★ Normalizing Transformation

- Problem of "taking a picture" has now been reduced to problem of finding correct normalizing transformation
- Finding rotation component of normalizing transformation is hard.
  - Easier to find inverse of rotational component (as you will see in a few slides.)
- Digression:
  - find inverse of normalizing transformation
  - called the **viewing transformation**
    - turns the canonical view into the arbitrary view
    - *(x, y, z)* to *(u, v, w)*

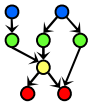


Building Viewing Transformation From View Specification

- We know the view specification: *Position*, *Look vector*, and *Up vector*
- Need to derive an affine transformation from these parameters to translate and rotate the canonical view into our arbitrary view
  - the scaling of the film (i.e. the cross-section of the view volume) to make a square cross-section will happen at a later stage, as will clipping
- Translation is easy to find: we want to translate the origin to the point *Position*; therefore, the translation matrix is

$$T(Position) = \begin{bmatrix} 1 & 0 & 0 & Pos_x \\ 0 & 1 & 0 & Pos_y \\ 0 & 0 & 1 & Pos_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rotation is harder: how do we generate a rotation matrix from the viewing specifications to turn *x, y, z*, into *u, v, w*?
  - a digression on rotation will help answer this

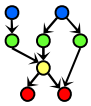


Rotation [1]

3 x 3 rotation matrices

- We learned about 3 x 3 matrices that "rigid-body rotate" the world (we're leaving out the homogeneous coordinate for simplicity)
- When they do, the three unit vectors that used to point along the *x*, *y*, and *z* axes are rotated to a new orientation
- Because it is a rigid-body rotation
  - resulting vectors are still unit length
  - resulting vectors are still perpendicular to each other
  - resulting vectors still satisfy the "right hand rule"
- *Any* matrix transformation with these three properties is a rotation about *some* axis by *some* amount!
- Let's call the three *x*-axis, *y*-axis, and *z*-axis-aligned unit vectors e_1, e_2, e_3
- Writing out:

$$e_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad e_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \qquad e_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$



Rotation [2]

- Let's call our rotation matrix *M*, and let's label its columns v_1, v_2, and v_3:

$$M = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix}$$

- When we multiply *M* by e_1, what do we get?

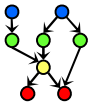
$$Me_1 = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = v_1 \longrightarrow Me_1 \text{ is the first column of } M$$

- Similarly for e_2 and e_3:

$$Me_2 = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = v_2 \longrightarrow Me_2 \text{ is the second column of } M$$

$$Me_3 = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = v_3 \longrightarrow Me_3 \text{ is the third column of } M$$

- Therefore M = [u v w] where u, v, and w are unit column vectors, will rotate the x, y, z axes into the u, v, w axes
- And M^{-1} would rotate the u, v, w axes into the x, y, z axes, which is what we actually want...
- Therefore we first find M by computing u, v and w from the viewing specification parameters, and then we need to find an easy way of getting the inverse of M.



Rotation [3]

- First, solve easier problem of finding the inverse of a rotation matrix
- As we learned in the transformations lecture, for a *rotation* matrix with columns v_i
  - columns must be unit vectors: $||v_i|| = 1$
  - columns are perpendicular: $v_i \cdot v_j = 0 \quad (i \neq j)$
- Therefore

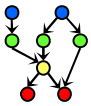
$v_i \cdot v_i = 1$ because $||v_i|| = 1$ but all other $v_i \cdot v_j = 0 \ (i \neq j)$

$$\begin{bmatrix} v_1 \cdot v_1 & v_1 \cdot v_2 & v_1 \cdot v_3 \\ v_2 \cdot v_1 & v_2 \cdot v_2 & v_2 \cdot v_3 \\ v_3 \cdot v_1 & v_3 \cdot v_2 & v_3 \cdot v_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- We can write this matrix dot products as $M^T M = I$ where M^T is a matrix whose *rows* are v_1, v_2, and v_3
- Also, for matrices in general, $M^{-1}M = I$, (actually, M^{-1} exists only for "well-behaved" matrices)
- Therefore, for rotation matrices, we have just shown that M^{-1} is simply M^T

$$M^T M = I \quad \wedge \quad M^{-1} M = I \quad \Rightarrow \quad M^T = M^{-1}$$

- M^T is trivial to compute, M^{-1} takes considerable work: big win!



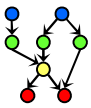
Rotation [4]

Summary

- If M is a rotation matrix, then its columns are pairwise perpendicular and have unit length
- Inversely, if the columns of a matrix are pairwise perpendicular and have unit length, then the matrix is a rotation
- For such a matrix,

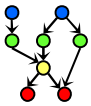
$$M^T M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Most importantly, then $M^T = M^{-1}$ and this will help us build the normalizing transformation from the easier to find viewing transformation.



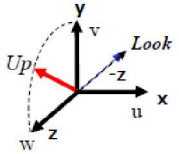
Construction of Orientation Matrix

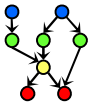
- Know how to invert a rotation matrix, but how do we build it from the viewing specification to to normalize the camera-space unit vector axes (u, v, w) located at the origin into the world-space axes (x, y, z).
  - rotation matrix M will turn (x, y, z) into (u, v, w) and has **columns (u, v, w) - viewing matrix**
  - conversely, $M^{-1}=M^T$ turns (u, v, w) into (x, y, z). M^T has **rows (u, v, w) - normalization matrix**
- Reduces the problem of finding the correct rotation matrix into finding the correct perpendicular unit vectors u, v, and w
- Restatement of rotation problem: Using *Position*, *Look vector*, and *Up vector*, compute viewing rotation matrix M with columns u, v, and w, then use its inverse, the transpose M^T, with row vectors u, v, w to get the normalization rotation matrix



Finding (u, v, w) from Position, Look, and Up [1]

- We know that we want the (u, v, w) axes to have the following properties:
  - our arbitrary *Look Vector* will lie along the negative *w-axis*
  - a projection of the *Up Vector* into the plane defined by the w-axis as its normal will lie along the v-axis
  - The u-axis will be mutually perpendicular to the v and w-axes, and will form a right-handed coordinate system
- Plan of attack: first find w from *Look*, then find v from the *Up* and w vector, then find u as a normal to the plane defined by w and v



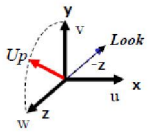


Finding (u, v, w) from Position, Look, and Up [2]

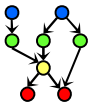
Finding *w*

- Finding *w* is easy. *Look vector* in canonical volume lies on –*z*. Since *z* maps to *w*, *w* is a normalized vector pointing in the opposite direction from our arbitrary *Look vector*

$$w = \frac{-Look}{\|Look\|}$$



- Note that *Up* and *w* define a plane, and that *u* is a normal to that plane, and that *v* is a normal to the plane defined by *w* and *u*



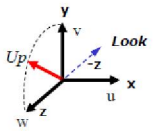
Finding (u, v, w) from Position, Look, and Up [3]

Finding *v*

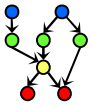
- Problem: find a vector, *v*, perpendicular to *w*
- Solution: project out the *w* component of the *Up* vector and normalize

$$v = Up - (w \bullet Up)w$$

$$v = \frac{v}{\|v\|}$$



- *w* is unit length, but *Up vector* might not be unit length or perpendicular to *w*, so we have to remove the *w* component and then normalize
- By removing the *w* component from the *Up* vector, the resulting vector is the component of *Up* in a direction perpendicular to *w*
- To create the orthogonal coordinate frame of the camera we need a third vector

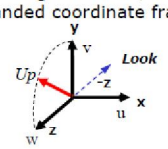


Finding (u, v, w) from Position, Look, and Up [4]

Finding *u*

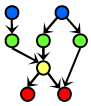
- We can use cross-product, but which one should we use?
  - *w* X *v* and *v* X *w* are both perpendicular to the plane, but in different directions . . .
- Answer: cross-products are right-handed, so use *v* X *w* to create a right-handed coordinate frame

$$u = \frac{v \times w}{\|v \times w\|}$$



- As a reminder, the cross product of two vectors *a* and *b* is:

$$a \times b = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$



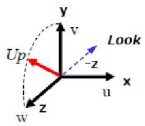
Finding (u, v, w) from Position, Look, and Up [5]

To summarize

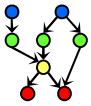
$$w = \frac{-Look}{\|Look\|}$$

$$v = \frac{Up - (w \bullet Up)w}{\|Up - (w \bullet Up)w\|}$$

$$u = \frac{v \times w}{\|v \times w\|}$$



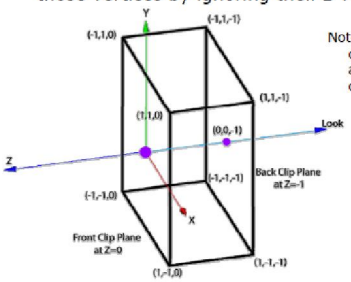
- The viewing transformation is now fully specified
  - knowing *u, v, and w*, we can rotate the canonical view into the user-specified orientation
  - we already know how to translate the view
- **Important Note:** we don't actually apply the forward viewing transformation. Instead, the inverse viewing transformation, namely the normalizing transformation, will be used to map the arbitrary view into the canonical view



Transforming to Canonical View

The Viewing Problem for Parallel Projection

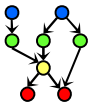
- Given a parallel view specification and vertices of a bunch of objects, we use the normalizing transformation, i.e., the inverse viewing transformation, to normalize the view volume to a cuboid at the origin, then clip, and then project those vertices by ignoring their z values



Note: it's a cuboid, not a cube (transformation arithmetic and clipping are easier)

The Viewing Problem for Perspective Projection

- Normalize the perspective view specification to a unit frustum at the origin looking down the –z axis; then transform the perspective view volume into a parallel (cuboid) view volume, simplifying both clipping and projection



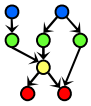
Normalizing View Volume: Overview

The Parallel Case

- Decomposes into multiple steps
- Each step defined by a matrix transformation
- The product of these matrices defines the whole transformation in one large, composite matrix. The steps are:
  - move the eye/camera to the origin
  - transform the view so that (u, v, w) is aligned with (x, y, z)
  - adjust the scales so that the view volume fits between –1 and 1 in x and y, the back clip plane lies at $z = -1$, the front plane at $z = 0$

The Perspective Case

- Same as parallel, but add one more step:
  - distort pyramid to cuboid to achieve perspective distortion to align the front clip plane with $z = 0$



Normalizing View Volume: Step 1

Move the eye to the origin

- We want a matrix to transform

  (Pos_x, Pos_y, Pos_z) to $(0, 0, 0)$

- Solution: it's just the inverse of the viewing translation transformation:

  $(t_x, t_y, t_z) = (-Pos_x, -Pos_y, -Pos_z)$

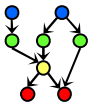
- We will take the matrix:

$$T_{trans} = \begin{bmatrix} 1 & 0 & 0 & -Pos_x \\ 0 & 1 & 0 & -Pos_y \\ 0 & 0 & 1 & -Pos_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  and we will multiply all vertices explicitly (and the camera implicitly) to preserve the relationship between camera and scene, i.e., for all vertices *p*

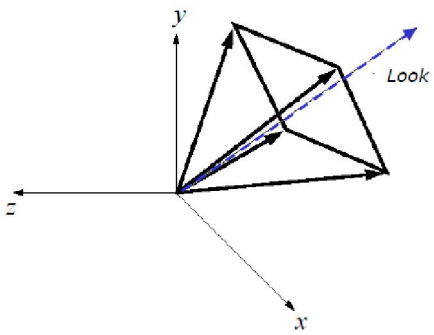
- This will move *Position* (the "eye point") to $(0, 0, 0)$

$$p' = T_{trans} p$$



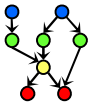
Normalizing View Volume: Step 2

- *Position* now at origin



But we're hardly done! Still need to:

- align orientation with *x,y,z* world coordinate system
- normalize proportions of the view volume



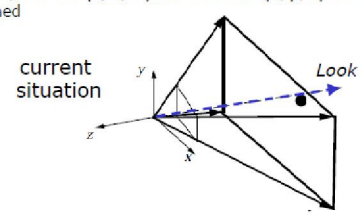
Normalizing View Volume: Step 3

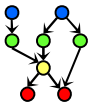
Rotate the view and align with the world coordinate system

- We found out that the view transformation matrix *M* with columns *u, v, and w* would rotate the *x, y, z* axes into the *u, v, and w* axes
- We now apply the inverse (transpose) of that rotation, M^T, to the scene. That is, a matrix with *rows u, v, and w* will rotate the axes *u, v, and w* into the axes *x, y,* and *z*
  - Define M_{rot} to be this rotation matrix transpose
- Now every vertex in the scene (and the camera implicitly) is multiplied by the composite matrix

$$M_{rot}T_{trans}$$

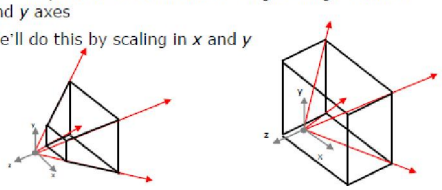
We've translated and rotated, so that the *Position* is at the origin, and the (*u, v, w*) axes and the (*x, y, z*) axes are aligned

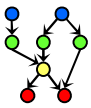




Normalizing View Volume: Step 4

- We've gotten things more or less to the right place, but the proportions of the view volume need to be normalized…
  - last affine transformation: **scaling**
- Need to be normalized to a square cross-section 2-by-2 units
  - why is that preferable to the unit square?
- Adjust so that the corners of far clipping plane eventually lie at (±1, ±1, –1)
- One mathematical operation works for both parallel and perspective view volumes
- Imagine vectors emanating from origin passing through corners of far clipping plane. For perspective view volume, these are edges of volume. For parallel, these lie inside view volume
- First step: force vectors into 45-degree angles with *x* and *y* axes
- We'll do this by scaling in *x* and *y*

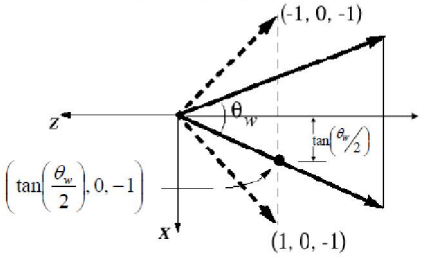




Normalizing View Volume: Step 5

Scaling Clipping Planes

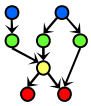
- Scale independently in x and y:
  looking down from above, we see this:



- Want to scale in x to make angle 90 degrees
- Need to scale in x by

$$\frac{1}{\left(\tan\frac{\theta_w}{2}\right)} = \cot\left(\frac{\theta_w}{2}\right)$$

- Similarly in y



Normalizing View Volume: Step 6

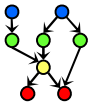
The xy scaling matrix

- The scale matrix we need looks like this:

$$S_{xy} = \begin{bmatrix} \cot\left(\frac{\theta_w}{2}\right) & 0 & 0 & 0 \\ 0 & \cot\left(\frac{\theta_h}{2}\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- So our current composite transformation looks like this:

$$S_{xy} M_{rot} T_{trans}$$

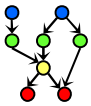


Normalizing View Volume: Step 7

One more scaling matrix

- Relative proportions of view volume planes are now correct, but the back clipping plane is probably lying at some $z \neq -1$, and we want all points inside view volume to have $0 \leq z \leq -1$
- Need to shrink the back (far) plane to be at $z = -1$
- The z distance from the eye to that point has not changed: it's still *far* (distance to far clipping plane)
- If we scale in z only, proportions of volume will change; instead we scale uniformly:

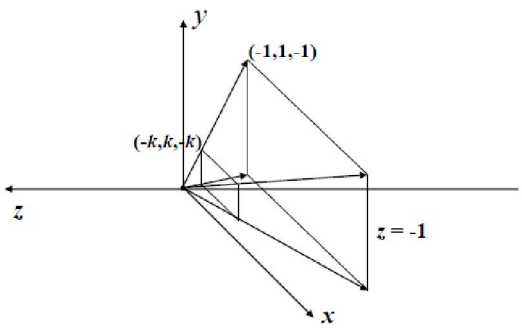
$$S_{far} = \begin{bmatrix} \frac{1}{far} & 0 & 0 & 0 \\ 0 & \frac{1}{far} & 0 & 0 \\ 0 & 0 & \frac{1}{far} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



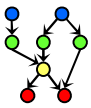
Normalizing View Volume: Step 8

The Current Situation

- Far plane at $z = -1$.



- Near clip plane now at $z = -k$ (note $k > 0$)

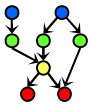


Normalizing View Volume: Results

- Our near-final composite normalizing transformation for canonical perspective view volume:

$$S_{far} S_{xy} M_{rot} T_{trans}$$

  - T_{trans} takes the camera's *Position* and moves the camera to the world origin
  - M_{rot} takes the *Look* and *Up* vectors and orients the camera to look down the *–z* axis
  - S_{xy} takes θ_w, θ_h and scales the clipping planes so that the corners are at (±1, ±1)
  - S_{far} takes the far clipping plane and scales it to lie on the *z=-1* plane



Perspective Transformation [1]

- We've put the perspective view volume into canonical position, orientation and size
- Let's look at a particular point on the original near clipping plane lying on the *Look vector*:

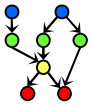
$$p = Position + near \cdot Look$$

It gets moved to a new location

$$p' = S_{far} S_{xy} M_{rot} T_{trans} p$$

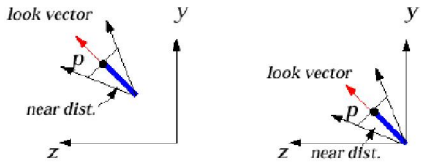
on the negative *z*-axis, say

$$p' = (0 \quad 0 \quad -k)$$

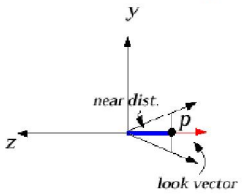


Perspective Transformation [2]

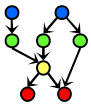
- What is the value of k? Trace through the steps.
  p first gets moved to just $near \cdot Look$



- This point is then rotated to $(near)(-e_3)$

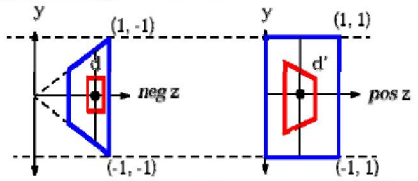


- The xy scaling has no effect, and the *far* scaling changes this to $\left(-\frac{near}{far}\right)e_3$, so $k = \frac{near}{far}$



Perspective Transformation [3]

- Transform points in standard *perspective* view volume between $-k$ and -1 to standard parallel view volume
- "z-buffer," used for visible surface calculation, needs z values to be [0 1], not [−1 0]. Perspective transformation must therefore transform scene to positive range $0 \le z \le 1$

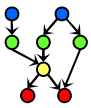


- The matrix that does this:
- (Remember that $0 < k < 1$ ...)

$$D = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{k-1} & \frac{k}{k-1} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Note: not 1!
Flip the z-axis and unhinge

- Why not originally align camera to +z axis?
  - Choice is perceptual, we think of looking through a display device into the scene that lies behind window

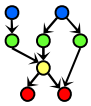


Perspective Transformation [4]

- Take a typical point prior to perspective transform $p=\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$ and rewrite it as $\begin{bmatrix} x \\ y \\ -k-d \\ 1 \end{bmatrix}$
- $0 \le d \le 1-k$; p is parameterized by distance along frustum
  - when $d = 0$, p is on near clip plane; when $d = (1\text{-}k)$, p is on far clip plane
  - depending on x, y, and z, p may or may not actually fall within frustum
- Apply D (from previous slide) to p to get new point p'

$$p' = D\begin{bmatrix} x \\ y \\ -k-d \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ \frac{(-k-d)}{k-1}+\frac{k}{k-1} \\ -1/(-k-d) \end{bmatrix} = \begin{bmatrix} x \\ y \\ -d/(k-1) \\ k+d \end{bmatrix} \rightarrow \begin{bmatrix} x/(k+d) \\ y/(k+d) \\ -d/((k-1)(k+d)) \\ 1 \end{bmatrix}$$

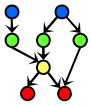
- Note: $w = k+d \neq 1$ so in the last step we must divide through by $k+d$!!!
  - this causes x and y to be "perspectivized", with points closer to the near clip plane being scaled up the most
  - this also transforms z, but z is tossed out when we project onto the film plane so it ultimately doesn't matter
  - understanding homogenous coordinates is essential to graphics
- Try values of d: 0, 1-k, ½(1-k), -1, 1



Perspective Transformation [5]

- Again consider $p' = \begin{bmatrix} x/(k+d) \\ y/(k+d) \\ -d/((k-1)(k+d)) \\ 1 \end{bmatrix}$
- What happens to x and y when d gets very large? $x \to 0$ and $y \to 0$ as $d \to \infty$
  - note: when we let d increase beyond $(1\text{-}k)$, p is now *beyond* the frustum (such lines will be *clipped*)
- This result provides *perspective foreshortening*: parallel lines converge to a vanishing point
- What happens when d is negative?

  - now p lies in front of the near clip plane, possibly behind the eye point
  - result: when $(k+d)$ becomes negative, the signs of x and y will be *flipped*: text would be upside-down and backwards
  - you won't see these points because they are clipped
- What happens after perspective transformation?
- Answer: parallel *projection* is applied to determine location of points onto the *film plane*
  - projection is easy: drop z!
  - however, we still need to keep the z ordering intact for visible surface determination



Perspective Transformation: End Result

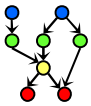
- Final transformation:

$$p' = D_{persp} S_{far} S_{xy} M_{rot} T_{trans} p$$

- Note that once the viewing parameters (*Position*, *Up vector*, *Look vector*, *Height angle*, *Aspect ratio*, *Near*, and *Far*) are known, the matrices

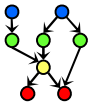
$$D_{persp}, S_{far}, S_{xy}, M_{rot}, T_{trans}$$

  can all be computed and multiplied together to get a single 4x4 matrix that is applied to all points of all objects to get them from "world space" to the standard parallel view volume.
- This is a huge win for homogeneous coordinates
- NOTE: Slide nomenclature differs from the book:
  - k is –c in the book
  - near is n, far is f



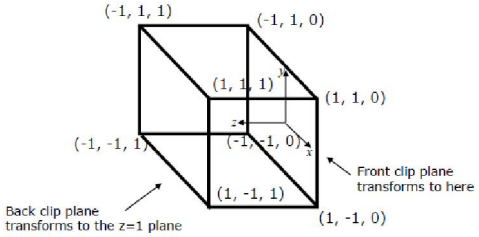
Stage 3: Making 2-D Image

- So far we've
  - Specified the view volume
  - Transformed from the specified view volume into the canonical view volume
- Last stage involves
  - Clipping
  - Projecting

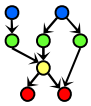


Clipping

- We said that taking the picture from the canonical view would be easy: final steps are clipping and projecting onto the film plane
- Need to clip scene against sides of view volume
- However, we've normalized our view volume into an axis-aligned cuboid that extends from –1 to 1 in *x* and *y* and from 0 to 1 in *z*



- Note: This is the flipped (in *z*) version of the canonical view volume
- Clipping is easy! Test *x* and *y* components of vertices against +/-1. Test *z* components against 0 and 1



Final Projection: Viewport Mapping

- Can make an image by taking each point and "ignoring *z*" to project it onto the *xy*-plane
- A point (*x*, *y*, *z*) where

- Entire problem can be reduced to a composite matrix multiplication of vertices, clipping, and a final matrix multiplication to produce screen coordinates.
- Final composite matrix (*CTM*) is composite of all modeling (instance) transformations (*CMTM*) accumulated during scene graph traversal from root to leaf, composited with the final composite normalizing transformation *N* applied to the root/world coordinate system:

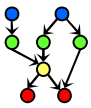
1) $N = D_{persp} S_{far} S_{xy} M_{rot} T_{trans}$

2) $CTM = N \cdot CMTM$

3) $P' = CTM \cdot P$ for every vertex P defined in its own coordinate system

4) $P_{screen} = 512 \cdot P' + 1$ for all **clipped** *P'*

1. Note that these functions are not exactly correct since if x or y is ever 1, then we will get x' or y' to be 1024 which is out of our range, so we need to make sure that we handle these cases gracefully. In most cases, making sure that we get the floor of 512(x+1) will address this problem since the desired x will be less than 1.



Fixed Function Graphics Pipeline: Composite Transformation Matrix

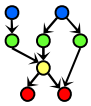
- Entire problem can be reduced to a composite matrix multiplication of vertices, clipping, and a final matrix multiplication to produce screen coordinates.
- Final composite matrix (*CTM*) is composite of all modeling (instance) transformations (*CMTM*) accumulated during scene graph traversal from root to leaf, composited with the final composite normalizing transformation *N* applied to the root/world coordinate system:

1) $N = D_{persp} S_{far} S_{xy} M_{rot} T_{trans}$

2) $CTM = N \cdot CMTM$

3) $P' = CTM \cdot P$ for every vertex P defined in its own coordinate system

4) $P_{screen} = 512 \cdot P' + 1$ for all **clipped** P'



Putting It All Together: Coordinate Spaces & Transformations

(See Eberly 2e § 2.3.2 – 2.3.7, pp. 48-66, especially p. 58)

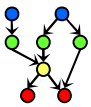
1. model coordinates / object coordinates
2. world coordinates / scene coordinates
3. camera coordinates / eye coordinates
4. (optional) view coordinates / clip coordinates
5. normalized device coordinates (NDC)
6. screen coordinates

$X_{model} \rightarrow (H_{world})$
$X_{world} \rightarrow (H_{view})$
$X_{view} \rightarrow (H_{proj})$
$X_{clip} \rightarrow$ (perspective division)
$X_{ndc} \rightarrow (H_{window})$
X_{window}

H_{world}: modelview transformation

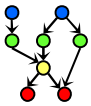
Normalizing transformation: $X_{world} \rightarrow X_{ndc}$ {
H_{view}: "view matrix" **(really NT!)**
H_{proj}: projection matrix
/w: perspective division

H_{window}: window matrix
(*aka* viewport transformation)



Summary

- Last Time: View Volume Specification and Viewing Transformation
  - View volumes: ideal *vs.* approximated
  - Specifying view volume in CG: Look and Up vectors
  - Aspect ratio, view angle, front/back clipping planes, focal length
- Today
  - Viewing transformation (NT)
  - Rotation
  - Finding camera coordinates (u, v, w) from Look, Up
  - Normalizing transformation (NT)
  - Perspective transformation *D*
  - Overall CTM of fixed-function pipeline
  - Intro to clipping
- Coming Week
  - Lab: OpenGL basics
  - Drawing lines, polygons
  - Parametric clipping



Terminology

- Today: View Volumes, Viewing Transformation
  - Parallel projection (cuboid) view volume *vs.* perspective projection frustum
  - Front/back (near/far) clipping planes
  - World coordinates (canonical): (*x*, *y*, *z*)
  - User coordinates (arbitary): (*u*, *v*, *w*), *aka* (*u*, *v*, *n*) or (*R*, *U*, *D*) in Eberly
  - Perspective transformation *D*: "morphs" frustum into cuboid
  - Normalizing transformation: inverse of viewing transformation (easier)
- Other Topics
  - Clipping
    - Given view volume
    - Problem of determining visible elements of scene
  - Cumulative Transformation Matrices (CTM)
    - Translation, Rotation, Scaling (TRS)
    - Used to map arbitrary view to canonical view
    - Then "ready to take picture"