We have two non-intersecting 3D lines in space.

Line 1 is defined by a point \( p_1 \) on the line and a ray \( r_1 \) in the direction of the line, and line 2 is defined by \( p_2 \) and \( r_2 \).

Find the closest distance between the two lines.
Example: Distance Between Lines

- We want to find the length of the vector between the two points projected onto the axis perpendicular to the two lines.

\[ \text{dist} = \frac{(p_2 - p_1) \cdot (r_1 \times r_2)}{|r_1 \times r_2|} \]
Example: Distance Between Lines

\[ \text{dist} = \frac{(p_2 - p_1) \cdot (r_1 \times r_2)}{|r_1 \times r_2|} \]

- Notice that the sign of the result might be positive or negative depending on the geometric configuration, so we should really take the absolute value if we want a positive distance.
- Also note that the equation could fail if \(|r_1 \times r_2|=0\), implying that the lines are parallel. We should really test for this situation and use a different equation if the lines are parallel.
Linear Transformations

\[ v'_x = a_1 v_x + b_1 v_y + c_1 v_z + d_1 \]
\[ v'_y = a_2 v_x + b_2 v_y + c_2 v_z + d_2 \]
\[ v'_z = a_3 v_x + b_3 v_y + c_3 v_z + d_3 \]

\[
\begin{bmatrix}
  v'_x \\
  v'_y \\
  v'_z
\end{bmatrix} =
\begin{bmatrix}
  a_1 & b_1 & c_1 \\
  a_2 & b_2 & c_2 \\
  a_3 & b_3 & c_3
\end{bmatrix}
\begin{bmatrix}
  v_x \\
  v_y \\
  v_z
\end{bmatrix} +
\begin{bmatrix}
  d_1 \\
  d_2 \\
  d_3
\end{bmatrix}
\]

\[ v' = M \cdot v + d \]
Multiple Transformations

- We can apply any sequence of transformations:
  \[ \mathbf{v}' = M_4 \cdot (M_3 \cdot (M_2 \cdot (M_1 \cdot \mathbf{v}))) \]

- Because matrix algebra obeys the *associative* law, we can regroup this as:
  \[ \mathbf{v}' = (M_4 \cdot M_3 \cdot M_2 \cdot M_1) \cdot \mathbf{v} \]

- This allows us to *concatenate* them into a single matrix:
  \[ M_{total} = M_4 \cdot M_3 \cdot M_2 \cdot M_1 \]
  \[ \mathbf{v}' = M_{total} \cdot \mathbf{v} \]

- Note: matrices do NOT obey the *commutative* law, so the order of multiplications is important.
Matrix Dot Matrix

$$L = M \cdot N$$

$$\begin{bmatrix}
  l_{11} & l_{12} & l_{13} \\
  l_{21} & l_{22} & l_{23} \\
  l_{31} & l_{32} & l_{33}
\end{bmatrix}
= \begin{bmatrix}
  m_{11} & m_{12} & m_{13} \\
  m_{21} & m_{22} & m_{23} \\
  m_{31} & m_{32} & m_{33}
\end{bmatrix}
\cdot \begin{bmatrix}
  n_{11} & n_{12} & n_{13} \\
  n_{21} & n_{22} & n_{23} \\
  n_{31} & n_{32} & n_{33}
\end{bmatrix}$$

$$l_{12} = m_{11}n_{12} + m_{12}n_{22} + m_{13}n_{32}$$
Multiple Rotations & Scales

- We can combine a sequence of rotations and scales into a single matrix.
- For example, we can combine a $y$-rotation, followed by a $z$-rotation, then a non-uniform scale, and finally an $x$-rotation:

$$M = R_x(\gamma) \cdot S(s) \cdot R_z(\beta) \cdot R_y(\alpha)$$

$$v' = M \cdot v$$
Multiple Translations

- We can also take advantage of the associative property of vector addition to combine a sequence of translations.
- For example, a translation along vector $t_1$ followed by a translation along $t_2$ and finally $t_3$ can be combined:

$$d = t_1 + t_2 + t_3$$

$$v' = v + d$$
Combining Transformations

- We see that we can combine a sequence of rotations and/or scales.
- We can also combine a sequence of translations.
- But what if we want to combine translations with rotations/scales?
Homogeneous Transformations

\[ \mathbf{v}' = \mathbf{M} \cdot \mathbf{v} \]

\[
\begin{bmatrix}
    v'_x \\
    v'_y \\
    v'_z \\
    1
\end{bmatrix} =
\begin{bmatrix}
    a_1 & b_1 & c_1 & d_1 \\
    a_2 & b_2 & c_2 & d_2 \\
    a_3 & b_3 & c_3 & d_3 \\
    0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
    v_x \\
    v_y \\
    v_z \\
    1
\end{bmatrix}
\]

\[
v'_x = a_1 v_x + b_1 v_y + c_1 v_z + d_1
\]

\[
v'_y = a_2 v_x + b_2 v_y + c_2 v_z + d_2
\]

\[
v'_z = a_3 v_x + b_3 v_y + c_3 v_z + d_3
\]

\[
1 = 0 v_x + 0 v_y + 0 v_z + 1
\]
Homogeneous Transformations

- So we’ve basically taken the 3x3 rotation/scale matrix and the 3x1 translation vector from our original equation and combined them into a new 4x4 matrix (for today, we will always have \([0 0 0 1]\) in the bottom row of the matrix).
- We also replace our 3D position vector \(v\) with its 4D version \([v_x, v_y, v_z, 1]\) (for today, we will always have a 1 in the fourth coordinate).
- Using 4x4 transformation matrices allows us to combine rotations, translations, scales, shears, and reflections into a single matrix.
- In the next lecture, we will see how we can also use them for perspective and other viewing projections.
Homogeneous Transformations

- For example, a translation by vector \( \mathbf{r} \) followed by a \( z \)-axis rotation can be represented as:

\[
\begin{bmatrix}
\mathbf{v}'_x \\
\mathbf{v}'_y \\
\mathbf{v}'_z \\
1
\end{bmatrix} =
\begin{bmatrix}
\cos \theta & -\sin \theta & 0 & 0 \\
\sin \theta & \cos \theta & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & r_x \\
0 & 1 & 0 & r_y \\
0 & 0 & 1 & r_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\mathbf{v}_x \\
\mathbf{v}_y \\
\mathbf{v}_z \\
1
\end{bmatrix}
\]
Rotation Matrices

\[ \mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

\[ \mathbf{R}_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

\[ \mathbf{R}_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]
Arbitrary Axis Rotation

- We can also rotate around an arbitrary unit axis \( \mathbf{a} \) by an angle \( \theta \)

\[
\mathbf{R}_a (\mathbf{a}, \theta) =
\begin{bmatrix}
    a_x^2 + c_\theta (1 - a_x^2) & a_x a_y (1 - c_\theta) - a_z s_\theta & a_x a_z (1 - c_\theta) + a_y s_\theta & 0 \\
    a_x a_y (1 - c_\theta) + a_z s_\theta & a_y^2 + c_\theta (1 - a_y^2) & a_y a_z (1 - c_\theta) - a_x s_\theta & 0 \\
    a_x a_z (1 - c_\theta) - a_y s_\theta & a_y a_z (1 - c_\theta) + a_x s_\theta & a_z^2 + c_\theta (1 - a_z^2) & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}
\]
Euler’s Theorem

- Euler’s Theorem: Any two independent orthonormal coordinate frames can be related by a sequence of rotations (not more than three) about coordinate axes, where no two successive rotations may be about the same axis.

- Not to be confused with Euler angles, Euler integration, Newton-Euler dynamics, inviscid Euler equations, Euler characteristic…

- Leonard Euler (1707-1783)
Euler’s Theorem

- There are two useful ways of looking at Euler’s theorem in relationship to orientations:
  1. It tells us that we can represent any orientation as a rotation about three axes, for example x, then y, then z
  2. It also tells us that any orientation can be represented as a single rotation about some axis by some angle
Orientation

- The term *orientation* refers to an object’s angular configuration.
- It is essentially the rotational equivalent of ‘position’, unfortunately however, there is no equally simple representation.
- Euler’s Theorem tells us that we should not need more than 3 numbers to represent the orientation.
- There are several popular methods for representing an orientation. Some of these methods use 3 numbers, and other methods use more than 3…
Orientation

- One method for storing an orientation is to simply store it as a sequence of rotations about principle axes, such as x, then y, then z. We refer to this as Euler angles. This is a compact method, as it uses exactly 3 numbers to store the orientation.

- Another option is to represent the orientation as a unit length axis and a scalar rotation angle, thus using 4 numbers in total. This is a conceptually straightforward method, but is of limited mathematical use, and so is not terribly common.

- A third option is to use a 3x3 matrix to represent orientation (or the upper 3x3 portion of a 4x4 matrix). This method uses 9 numbers, and so must contain some extra information (namely, the scaling and shearing info).

- A fourth option is to use a quaternion, which is a special type of 4-dimensional vector. This is a popular and mathematically powerful method, but a little bit strange… These are covered in Math155B and CSE169.

- No matter which method is used to store the orientation information, we usually need to turn it into a matrix in order to actually transform points or do other useful things.
Scale Transformations

- The uniform scaling matrix scales an entire object by scale factor $s$
  \[ S_u(s) = \begin{bmatrix}
  s & 0 & 0 & 0 \\
  0 & s & 0 & 0 \\
  0 & 0 & s & 0 \\
  0 & 0 & 0 & 1 
\end{bmatrix} \]

- The non-uniform scaling matrix scales independently along the $x$, $y$, and $z$ axes
  \[ S_n(s) = \begin{bmatrix}
  s_x & 0 & 0 & 0 \\
  0 & s_y & 0 & 0 \\
  0 & 0 & s_z & 0 \\
  0 & 0 & 0 & 1 
\end{bmatrix} \]
Volume Preserving Scale

- We can see that a uniform scale by factor $s$ will increase the volume of an object by $s^3$, and a non-uniform scale will increase the volume by $s_x s_y s_z$.
- Occasionally, we want to do a *volume preserving scale*, which effectively stretches one axis, while squashing the other two.
- For example, a volume preserving scale along the $x$-axis by a factor of $s_x$ would scale the $y$ and $z$ axes by $1/\sqrt{s_x}$.
- This way, the total volume change is

$$s_x \cdot \frac{1}{\sqrt{s_x}} \cdot \frac{1}{\sqrt{s_x}} = 1$$
Volume Preserving Scale

- A volume preserving scale along the $x$ axis:

$$
S_{vx}(s_x) =
\begin{bmatrix}
  s_x & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & \frac{1}{\sqrt{s_x}} & 0 & 0 \\
  0 & 0 & \frac{1}{\sqrt{s_x}} & 0 \\
  0 & 0 & 0 & 1 \\
\end{bmatrix}
$$
Shear Transformations

- A shear transformation matrix looks something like this:

\[
Z(z_1...z_6) = \begin{bmatrix}
1 & z_1 & z_2 & 0 \\
z_3 & 1 & z_4 & 0 \\
z_5 & z_6 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

- With pure shears, only one of the constants is non-zero.
- A shear can also be interpreted as a non-uniform scale along a rotated set of axes.
- Shears are sometimes used in computer graphics for simple deformations or cartoon-like effects.
Translations

- A 4x4 translation matrix that translates an object by the vector \( \mathbf{r} \) is:

\[
T(\mathbf{r}) = \begin{bmatrix}
1 & 0 & 0 & r_x \\
0 & 1 & 0 & r_y \\
0 & 0 & 1 & r_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Pivot Points

- The standard rotation matrices pivot the object about an axis through the origin.
- What if we want the pivot point to be somewhere else?
- The following transformation performs a z-axis rotation pivoted around the point $\mathbf{r}$:

$$
M = T(\mathbf{r}) \cdot R_z(\theta) \cdot T(-\mathbf{r})
$$
General 4x4 Matrix

- All of the matrices we’ve see so far have \([0 \ 0 \ 0 \ 1]\) in the bottom row
- The product formed by multiplying any two matrices of this form will also have \([0 \ 0 \ 0 \ 1]\) in the bottom row
- We can say that this set of matrices forms a multiplicative group of 3D linear transformations
- We can construct any matrix in this group by multiplying a sequence of basic rotations, translations, scales, and shears

\[
\begin{bmatrix}
a_1 & b_1 & c_1 & d_1 \\
a_2 & b_2 & c_2 & d_2 \\
a_3 & b_3 & c_3 & d_3 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]
General 4x4 Matrix

- We will later see that we can change the bottom row to perform viewing projections, but for now, we will only consider the standard 3D transformations.
- We see that there are 12 different numbers in the upper 3x4 portion of the 4x4 matrix.
- There are also 12 degrees of freedom for an object undergoing a linear transformation in 3D space.
- 3 of those are represented by the three translational axes.
- 3 of them are for rotation in the 3 planes (xy, yz, xz).
- 3 of them are scales along the 3 main axes.
- and the last 3 are shears in the 3 main planes (xy, yz, xz).

- The 3 numbers for translation are easily decoded \((d_x, d_y, d_z)\).
- The other 9 numbers, however, are encoded into the 9 numbers in the upper 3x3 portion of the matrix.
Affine Transformations

- All of the transformations we’ve seen so far are examples of *affine* transformations.
- If we have a pair of parallel lines and transform them with an affine transformation, they will remain parallel.
- Affine transformations are fast to compute and very useful throughout computer graphics.
Object Space

- The space that an object is defined in is called object space or local space.
- Usually, the object is located at or near the origin and is aligned with the xyz axes in some reasonable way.
- The units in this space can be whatever we choose (i.e., meters, etc.).
- A 3D object would be stored on disk and in memory in this coordinate system.
- When we go to draw the object, we will want to transform it into a different space.
We will define a new space called *world space* or *global space*. This space represents a 3D world or scene and may contain several objects placed in various locations. Every object in the world needs a matrix that transforms its vertices from its own object space into this world space. We will call this the object’s *world matrix*, or often, we will just call it the object’s matrix. For example, if we have 100 chairs in the room, we only need to store the object space data for the chair once, and we can use 100 different matrices to transform the chair model into 100 locations in the world.
ABCD Vectors

- We mentioned that the translation information is easily extracted directly from the matrix, while the rotation information is encoded into the upper 3x3 portion of the matrix.
- Is there a geometric way to understand these 9 numbers?
- In fact there is! The 9 constants make up 3 vectors called \( \mathbf{a}, \mathbf{b}, \) and \( \mathbf{c}. \) If we think of the matrix as a transformation from object space to world space, then the \( \mathbf{a} \) vector is essentially the object’s x-axis rotated into world space, \( \mathbf{b} \) is its y-axis in world space, and \( \mathbf{c} \) is its z-axis in world space. \( \mathbf{d} \) is of course the position in world space.
Orthonormality

- If the a, b, and c vectors are all unit length and perpendicular to each other, we say that the matrix is *orthonormal*.
- Technically speaking, only the upper 3x3 portion is orthonormal, so the term is not strictly correct if there is a translation in the matrix.
- A better term for 4x4 matrices would be to call it *rigid* implying that it only rotates and translates the object, without any non-rigid scaling or shearing distortions.
Orthonormality

- If a 4x4 matrix represents a rigid transformation, then the upper 3x3 portion will be orthonormal

\[
\begin{align*}
|a| &= |b| = |c| = 1 \\
\mathbf{a} &= \mathbf{b} \times \mathbf{c} \\
\mathbf{b} &= \mathbf{c} \times \mathbf{a} \\
\mathbf{c} &= \mathbf{a} \times \mathbf{b}
\end{align*}
\]
Example: Move to Right

- We have a rigid object with matrix $M$. We want to move it 3 units to the object’s right.
Example: Move to Right

- We have a rigid object with matrix $M$. We want to move it 3 units to the object’s right.

$$M \cdot d = M \cdot d + 3 \cdot M \cdot a$$
Example: Target ‘Lock On’

For an airplane to get a missile locked on, the target must be within a 10 degree cone in front of the plane. If the plane’s matrix is $M$ and the target position is $t$, find an expression that determines if the plane can get a lock on.

$M \cdot t$
Example: Target ‘Lock On’

For an airplane to get a missile locked on, the target must be within a 10 degree cone in front of the plane. If the plane’s matrix is $\mathbf{M}$ and the target position is $\mathbf{t}$, find an expression that determines if the plane can get a lock on.

\[
\begin{align*}
\mathbf{M} \cdot \mathbf{t}
\end{align*}
\]
Example: Target ‘Lock On’

- We want to check the angle between the heading vector \((-c)\) and the vector from \(d\) to \(t\): 

\[
\text{if } \cos^{-1} \left( \frac{(d - t) \cdot c}{|d - t|} \right) < 10^\circ
\]

- We can speed that up by comparing the cosine instead (\(\cos(10^\circ) = 0.985\)) 

\[
\text{if } \frac{(d - t) \cdot c}{|d - t|} > 0.985
\]
Example: Target ‘Lock On’

- We can even speed that up further by removing the division and the square root in the magnitude computation:

\[ \text{if } \left( ( (d - t) \cdot c \right)^2 > 0.970 \cdot |d - t|^2 \) \]

- All together, this requires 8 multiplications and 8 adds
Position Vector Dot Matrix

\[ \mathbf{v}' = \mathbf{M} \cdot \mathbf{v} = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \\ \mathbf{v}_z \\ 1 \end{bmatrix} \]

\[ v'_x = v_x a_x + v_y b_x + v_z c_x + d_x \]

\[ v'_y = v_x a_y + v_y b_y + v_z c_y + d_y \]

\[ v'_z = v_x a_z + v_y b_z + v_z c_z + d_z \]
Position Vector Dot Matrix

\[ v' = v_x a + v_y b + v_z c + d \]

Local Space
\[ \mathbf{v'} = \nu_x \mathbf{a} + \nu_y \mathbf{b} + \nu_z \mathbf{c} + \mathbf{d} \]
Position Vector Dot Matrix

\[ \mathbf{v}' = v_x \mathbf{a} + v_y \mathbf{b} + v_z \mathbf{c} + d \]

Local Space

World Space
Position vs. Direction Vectors

- Vectors representing a position in 3D space are expanded into 4D as:
  \[
  \begin{bmatrix}
  v_x & v_y & v_z & 1
  \end{bmatrix}
  \]

- Vectors representing direction are expanded as:
  \[
  \begin{bmatrix}
  v_x & v_y & v_z & 0
  \end{bmatrix}
  \]
Direction Vector Dot Matrix

\[
v' = M \cdot v = \begin{bmatrix}
  a_x & b_x & c_x & d_x \\
  a_y & b_y & c_y & d_y \\
  a_z & b_z & c_z & d_z \\
  0 & 0 & 0 & 1
\end{bmatrix} \cdot \begin{bmatrix}
  v_x \\
  v_y \\
  v_z \\
  0
\end{bmatrix}
\]

\[
v'_x = v_x a_x + v_y b_x + v_z c_x
\]

\[
v'_y = v_x a_y + v_y b_y + v_z c_y
\]

\[
v'_z = v_x a_z + v_y b_z + v_z c_z
\]

\[
v' = v_x a + v_y b + v_z c
\]
The \textbf{abcd} vectors of $M'$ are the \textbf{abcd} vectors of $M$ transformed by matrix $N$.

- Notice that $a$, $b$, and $c$ transform as direction vectors and $d$ transforms as a position.
Identity

\[ I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

- Take one more look at the identity matrix
- It's a axis lines up with x, b lines up with y, and c lines up with z
- Position d is at the origin
- Therefore, it represents a transformation with no rotation or translation
Determinants

The determinant of a 4x4 matrix with no projection is equal to the determinant of the upper 3x3 portion

\[
\begin{vmatrix}
  a_x & b_x & c_x & d_x \\
  a_y & b_y & c_y & d_y \\
  a_z & b_z & c_z & d_z \\
  0 & 0 & 0 & 1
\end{vmatrix} = \begin{vmatrix}
  a_x & b_x & c_x \\
  a_y & b_y & c_y \\
  a_z & b_z & c_z
\end{vmatrix} = \mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})
\]
Determinants

- The determinant is a scalar value that represents the volume change that the transformation will cause.
- An orthonormal matrix will have a determinant of 1, but non-orthonormal volume preserving matrices will have a determinant of 1 also.
- A flattened or degenerate matrix has a determinant of 0.
- A matrix that has been mirrored will have a negative determinant.
Inversion

If $\mathbf{M}$ transforms $\mathbf{v}$ into world space, then $\mathbf{M}^{-1}$ transforms $\mathbf{v}'$ back into object space

\[ \mathbf{v}' = \mathbf{M} \cdot \mathbf{v} \]

\[ \mathbf{v} = \mathbf{M}^{-1} \cdot \mathbf{v}' \]

\[ \mathbf{M} \cdot \mathbf{M}^{-1} = \mathbf{M}^{-1} \cdot \mathbf{M} = \mathbf{I} \]
Camera Matrix

- Just like any object (chair, desk...) can use a matrix to place it into world space, we can have a virtual camera and use a matrix to place it into world space.
- The camera matrix essentially transforms vertices from *camera space* into world space.
Example: Camera ‘Look At’

- Our eye is located at position $e$ and we want to look at a target at position $t$. Generate an appropriate camera matrix $\mathbf{M}$. 
Example: Camera ‘Look At’

- Our eye is located at position $e$ and we want to look at a target at position $t$. Generate an appropriate camera matrix $M$.

- Two possible approaches include:
  - Measure angles and rotate matrix into place
  - Construct $a, b, c,$ & $d$ vectors of $M$ directly
Example: Camera ‘Look At’
Method 1: Measure Angles & Rotate

- Measure Angles:
  - Tilt angle
  - Heading angle
  - Position

- Construct matrix by starting with the identity, then apply tilt and heading rotations
Example: Camera ‘Look At’
Method 2: Build Matrix Directly

- The $d$ vector is just the position of the camera, which is $e$:

$$d = e$$

- The $c$ vector is a unit length vector that points directly behind the viewer:

$$c = \frac{e - t}{|e - t|}$$
Example: Camera ‘Look At’
Method 2: Build Matrix Directly

- The a vector is a unit length vector that points to the right of the viewer. It is perpendicular to the c axis. To keep the camera from rolling, we also want the a vector to lay flat in the xz-plane, perpendicular to the y-axis.

\[ a = \frac{y \times c}{|y \times c|} \]

- Note that a cross product with the y-axis can be optimized as follows:

\[
\begin{bmatrix}
0 & 1 & 0
\end{bmatrix} \times c =
\begin{bmatrix}
c_z & 0 & -c_x
\end{bmatrix}
\]
Example: Camera ‘Look At’
Method 2: Build Matrix Directly

- The \( \mathbf{b} \) vector is a unit length vector that points up relative to the viewer. It is perpendicular to the both the \( \mathbf{c} \) and \( \mathbf{a} \) axes

\[ \mathbf{b} = \mathbf{c} \times \mathbf{a} \]

- Note that \( \mathbf{b} \) does not need to be normalized because it is already unit length. This is because \( \mathbf{a} \) and \( \mathbf{c} \) are unit length vectors 90 degrees apart.

\[ |\mathbf{b}| = |\mathbf{c}||\mathbf{a}| \sin 90^\circ = 1 \times 1 \times 1 \]
Example: Camera ‘Look At’
Method 2: Build Matrix Directly

Summary:

\[ d = e \]
\[ c = \frac{e - t}{|e - t|} \]
\[ a = \frac{y \times c}{|y \times c|} \]
\[ b = c \times a \]
Camera Space

Let’s say we want to render an image of a chair from a certain camera’s point of view.

The chair is placed in world space with matrix $M$.

The camera is placed in world space with matrix $C$.

The following transformation takes vertices from the chair’s object space into world space, and then from world space into camera space:

$$v' = C^{-1} \cdot M \cdot v$$

In order to actually render it, we must also project the point into 2D space, which we will learn about in the next lecture.