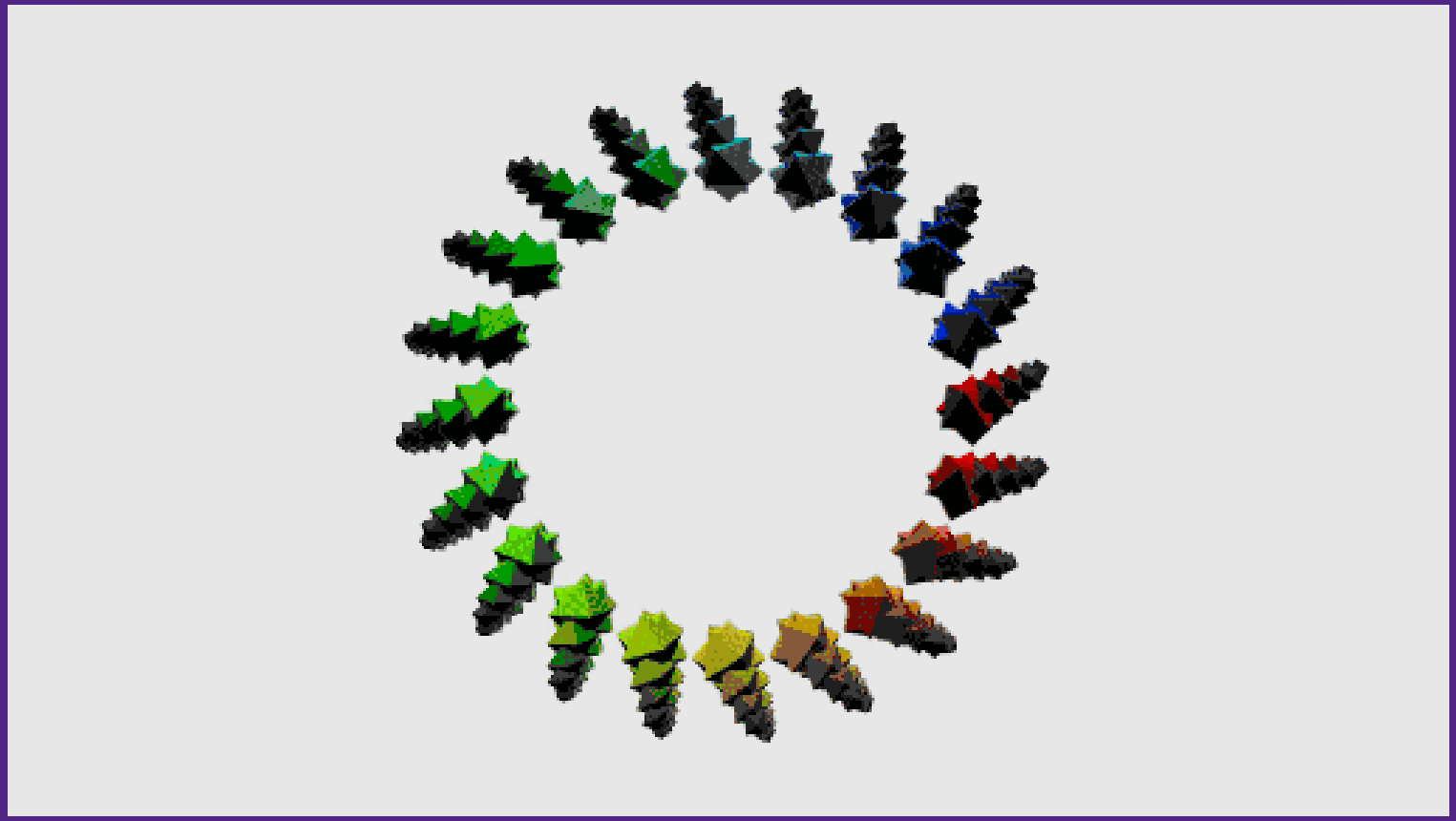
Computer Graphics I:

3D Transformations



Introduction to 3D Graphics

- We've made it past two dimensions.
- Enter, the third dimension.
- The hardest part about this is having to think about concepts written in two dimensions describing transformations in three dimensions.
- It's a brain workout to be sure.

Three Dimensional Affine Transformations

- Affine transformations in three dimensions allow us to manipulate 3D objects by altering their position, orientation and shape. We can express a 3D point as: P = X $\vec{i}$ + Y $\vec{j}$ + Z $\vec{k}$, where

$$\vec{i} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \vec{j} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \vec{k} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

- We use homogenous coordinates and column vectors such that points are written as follows:

$$P = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Three Dimensional Affine Transformations

- Generally, a 3D affine transformation is written in matrix form as:

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Such that transforming point P into point Q with matrix M is mathematically expressed as Q=MP

Elementary Transformations

- Translation:

$$M = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Elementary Transformations

- Scaling

$$M = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Elementary Transformations

- Rotation about the x-axis:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Elementary Transformations

- Rotation about the y-axis:

$$M = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Elementary Transformations

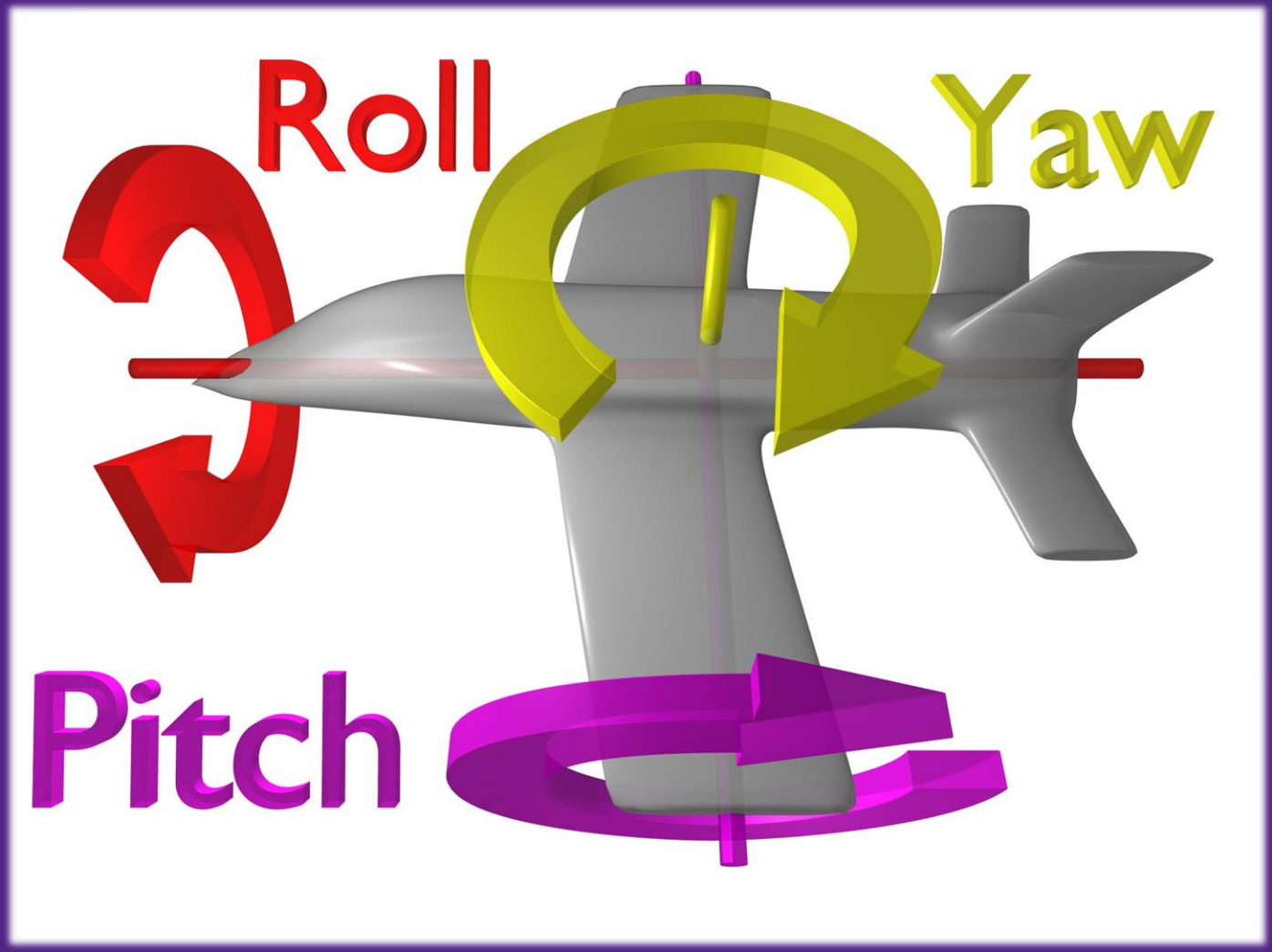
- Rotation about the z-axis:

$$M = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotations in 3D

- Three dimensional rotations are the biggest difference from the two-dimensional case.
- In two dimensions, rotations only have one degree of freedom. You can rotate clockwise or counter-clockwise in the plane.
- This is a single degree of freedom (indeed recall rotating clockwise by θ is the same as rotating counterclockwise by $-\theta$).
- On the other hand, consider now three dimensions. How many degrees of freedom do you have in rotation?

Rotations in 3D

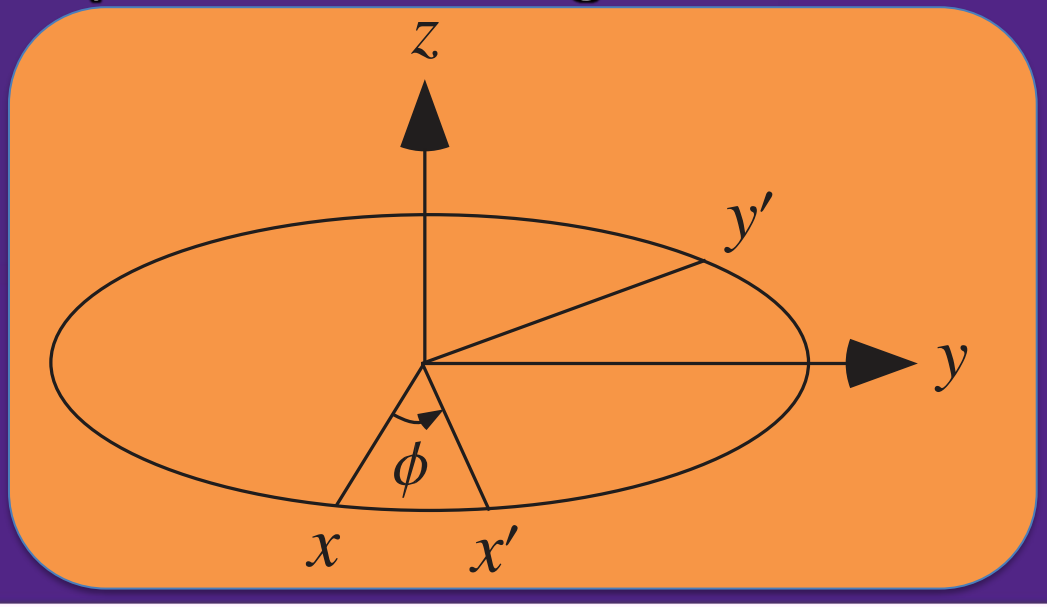


Rotations in 3D

- The building blocks of a three-dimensional rotation are individual rotations around each of the three individual axes.
- Let's think about two dimensions again for a moment. One way we though about two dimensional rotations is moving points around a circle of a particular radius.
- Thus, the point's distance to the origin never changes.

Rotations in 3D

- While that's not incorrect, another way to think about it is that we are rotating the X-Y plane around the Z-axis.
- Indeed, we can view two dimensions as being embedded inside three dimensional space but fixed to the z = 0 plane.
- Then, you are viewing that plane in a line of sight that is parallel to the z-axis.



Rotations in 3D

- In two dimensions, we represented this a la:

$$\begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} xcos\theta - ysin\theta \\ xsin\theta + ycos\theta \end{bmatrix}$$

- If we want this same rotation to occur, we need to modify *x* and *y* by the same amount, but we want to keep *z* unchanged. So, make that third row/column be that of the identity.

$$\begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotations in 3D

- This defines a 3D rotation around the z axis.
- In a right-handed coordinate system, we use the right-hand-rule to determine the direction of the rotation around this axis.
- Make the “thumbs up” gesture. Point your thumb in the direction of the axis of rotation. The direction of your fingers from knuckle to tip show the direction of the rotation.
- We can similarly define the other rotations around the x or y axes by fixing one of the axes to be unchanged and rotating around that axis.

Rotations in 3D

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\theta & -sin\theta \\ 0 & sin\theta & cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{bmatrix}$$

Elementary Transformations

- We can combine these matrices (through multiplication) to get a general rotation matrix in three dimensions.
- Indeed, any three-dimensional rotation can be described as the product of these three elementary rotations around the three coordinate axes.
- But how do you combine them? There's many possible choices.
- Naturally, the choice of order of the matrix multiplication changes the result.

Elementary Transformations

- The most common order is to first rotate around the z-axis, then rotate around the x-axis, then rotate around the y-axis.
- In this view, where the “world coordinates” remain fixed and we rotate around those fixed axes, the rotations are called extrinsic rotations.
- We are viewing objects as being rotated about some external frame of reference. The actual math of extrinsic rotations follows what we have been doing all along.

$$R_y R_x R_z v = v'$$

Elementary Transformations

- In many graphics libraries and other APIs, these three elementary rotations are combined into a single matrix with a function named something like “Euler Angles”.
- Euler angles describe a 3D rotation as the three different angles of rotation around the three coordinate axes.
- Note: each library may implement Euler angles differently, even though the inputs may appear the same.

Elementary Transformations

- EulerAngles(a,b,c) will describe a rotation of a degrees around the x-axis, b degrees around the y-axis, and c degrees around the z axis.
- However, the order in which those rotations ar combined will definitely change the output result.
- For example, the game engine Unity rotates around z first, then x, then y.
- See MVPTutorial step 0, substeps 1-8

Elementary Transformations

- **Euler's Rotation Theorem**
- Euler angles are certainly useful, but inherently ambiguous. Another option is given by Euler's rotation theorem.
- It says that any three-dimensional rotation can be described in terms of some axis of rotation, and a angle of rotation around that axis.
- This is called the axis-angle formulation of rotation.

Elementary Transformations

- The elementary rotations can be seen as special cases of the axis-angle formulation. For example, rotation around the z axis is a rotation around the axis ^k = (0, 0, 1).

- The hardest part of this formulation is to find the axis of rotation. But that itself is not that hard.

- Consider that, for any rotation, the axis of rotation is unchanged by the rotation. For the elementary rotation Rz, the z axis does not move, but the x and y axes do.

Elementary Transformations

- Therefore, given a general rotation matrix R, the axis of rotation is some direction vector ˆu such that Rˆu = ˆu.
- The important thing is to always **know what you're doing**.
- You can always build up a rotation matrix through multiplications of elementary rotations.
- You can also use glRotatef(theta, x, y, z) which uses the axis-angle formulation to define a 3D rotation.

Elementary Transformations

- This covers the most common 3D transformation matrices, but there exist others.
- All of these transformation matrices can be combined to form a single matrix encompassing many transformations
  - This is why we use homogenous coordinates.
- As an example, consider rotating a 3D point P around an arbitrary axis expressed with unit vector $\vec{v} = (v_x, v_y, v_z)^T$

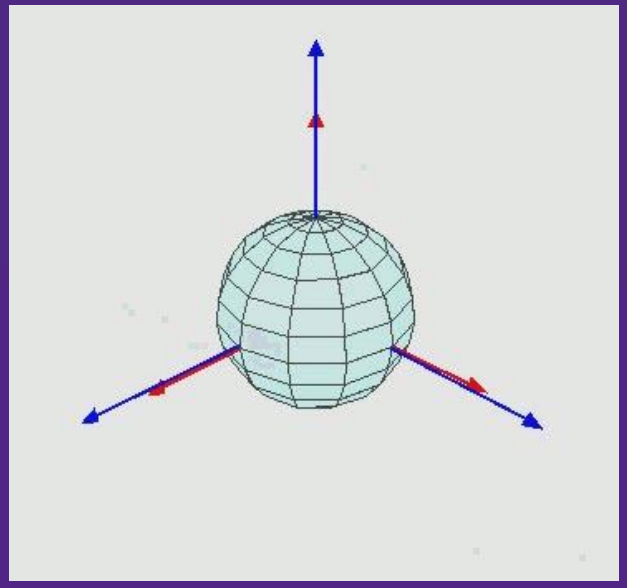
Elementary Transformations

- Working with $\vec{v} = (v_x, v_y, v_z)^T$
- First, project $\vec{v}$ onto the xy plane by setting its 3rd coordinate to 0
- Normalize the projected $\vec{v}$, and compute the angle ϕ between $\vec{v}$ and the x-axis as follows: $\phi = \arccos(\vec{i} \cdot \vec{v})$ where $\vec{i} = (1,0,0)$
- Apply a ϕ degree z-axis rotation to $\vec{v}$ as $R_z(\phi)\,\vec{v}$
- Now, $R_z(\phi)\,\vec{v}$ is located in the xz plane and we need to find the angle between it and the z-axis
- To find this angle, we compute $\psi = arccos\left(\frac{\vec{k} \cdot R_z(\phi)\vec{v}}{\|R_z(\phi)\vec{v}\|}\right)$
- where $\vec{k} = (0,0,1)$, and apply a ψ degree y axis roation to $\vec{v}$ and obtain $R_y(\psi)R_z(\phi)\vec{v}$
- Now, $R_y(\psi)R_z(\phi)\vec{v}$ is aligned with the z-axis and we can apply the desired rotation to point P around that axis.

Elementary Transformations

- Hence, to rotate a point P by an angle α around an arbitrary vector $\vec{v}$, we combine the following rotations: $R_y(\psi)R_z(\phi)$

$$Q = R_z(-\phi)R_y(-\psi)R_z(\alpha)R_y(\psi)R_z(\phi)P$$



Elementary Transformations

- Alternatively, we can construct a rotation matrix about any axis expressed with unit vector $\vec{v}$ directly. First we express the rotation axis in matrix form as:

$$J_{\vec{v}} = \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix}$$

- and compute the rotation matrix as:

$$R = I + \sin(\theta) J_{\vec{v}} + (1 - \cos(\theta)) J_{\vec{v}}^2$$

(I is the identity matrix)

Elementary Transformations

- It is also possible to derive both the angle of rotation and the axis vector when given matrix R:

$$\theta = cos^{-1}\left(\frac{tr(R)-1}{2}\right)$$

(tr is trace)

and

$$\frac{(R-R^T)}{2sin\theta} = J_{\vec{v}}$$

Direction of Positive Rotations Per Axis

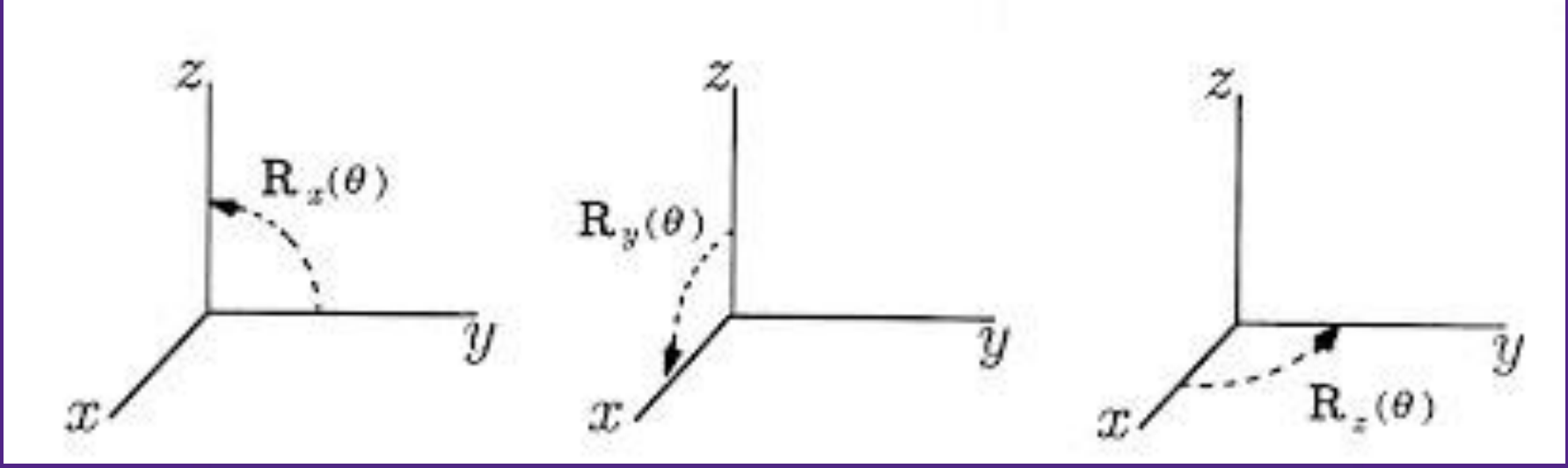


Illustration 1: Positive rotations per axis

3D Transformation Hierarchy

- The following table depicts the hierarchy of 3D transformations.
- Each transform also preserves the properties listed in the rows below it.
- For instance, the similarity transformation preserves not only angles but also parallelism and straight lines

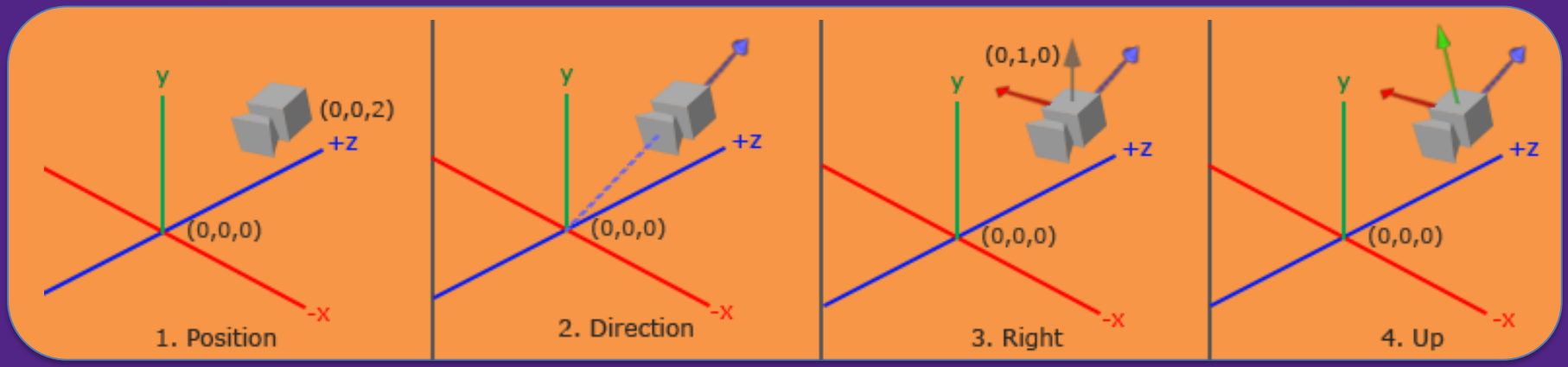
2D Transform	Degrees of Freedom	Preserves
Translation	3	Orientation
Rigid Body	6	Lengths
Similarity	7	Angles
Affine	12	Parallelism
Projective	15	Straight Lines

The Camera

- The camera can be thought of as existing at a particular point in the world and “looking at” another point.
- That is, a camera has a position and a direction.
- We have seen so far how, with some transformation matrix M we can “move” a vertex from its “local coordinates” to “world coordinates”.

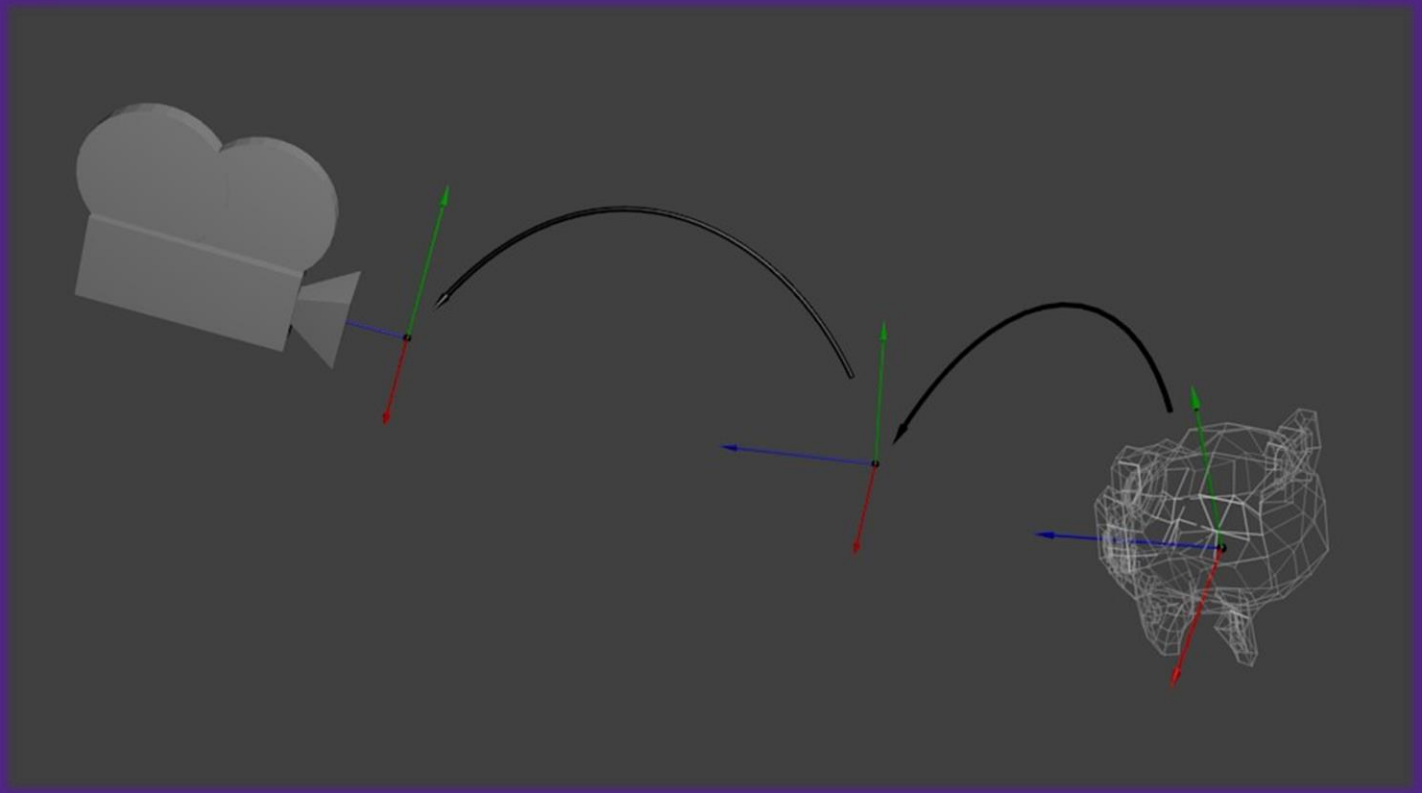
The Camera

- In any graphics library there is a concept of a camera.
- It is the “thing” from which we are viewing the scene.
- You can imagine the camera is somewhere in the world, and then what we render and see in the viewport is whatever the camera sees (after mapping to NDC and to the viewport).



The Camera

- In **camera space**, the camera is defined to exist at the origin (0, 0, 0) and looking down the negative z-axis.



The Camera

- Again, our view matrix tells us how to move from world coordinates to camera coordinates.
- If we want to “move the camera left” we actually “move the world right”.
- For example, if want the camera to be at (0, 0, 3) in world coordinates, we would define a view matrix V that moves the world 3 units in the negative z direction instead!

$$V = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

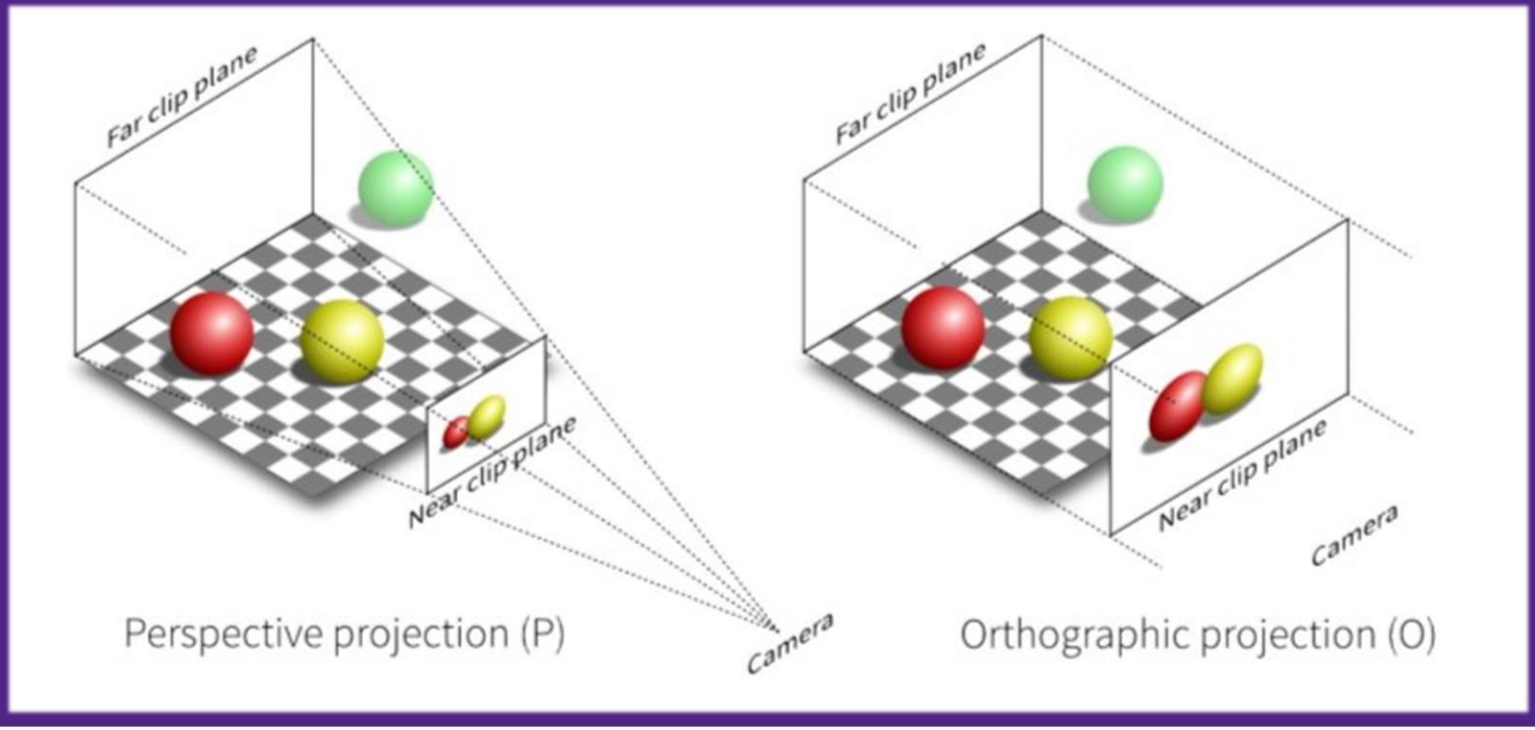
Perspective Projections

Perspective Projections

- Perspective projection is key to 3D graphics.
- They create a world that we are used to seeing!
- The lens in your eyes, the lenses in your cameras, they all use perspective projection.
- Objects that are closer to you appear larger.
- Parallel lines seem to meet at far distances.

Perspective Projections

- We previously saw orthographic projection. In this projection, the lines of projection were all *parallel*.
- In perspective projection, the projection lines are not parallel. They instead meet at a particular point.



Perspective Projections

- Assuming a viewing coordinate system defined by $\vec{u}$, $\vec{v}$, $\vec{n}$ in which the line of sight is given by $-\vec{n}$, projections are performed onto the near plane of the viewing volume, and N is the distance from the origin of the camera coordinate system to the near plane
(such that the point (0,0,−N) is contained in the near plane).

- Hence a perspective projection of a point $P=(X, Y, Z)^T$ is:

$$p = (x, y)^T = \left(-N\frac{X}{Z}, -N\frac{Y}{Z}\right)$$

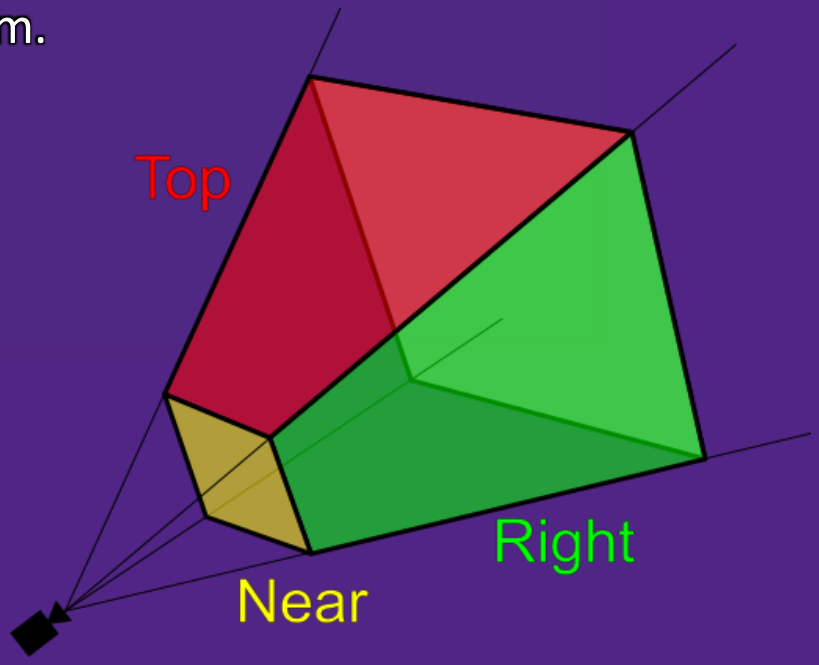
- Under this type of transformation, lines project onto lines but parallel lines do not project to parallel lines in general.

Perspective Projections

- This is done to capture some ideas that become more obvious from looking at examples:
  - Further objects are smaller
  - Parallel lines follow different rules in perspective.

Perspective Projections

- We previously saw orthogonal projection as a way of defining the "viewing volume" of what gets mapped to from world coordinates to Normalized Device Coordinates.
- Perspective projection can be thought of similarly, except that the viewing volume is no longer a rectangular prism.
- Now, it is a **square frustum**.

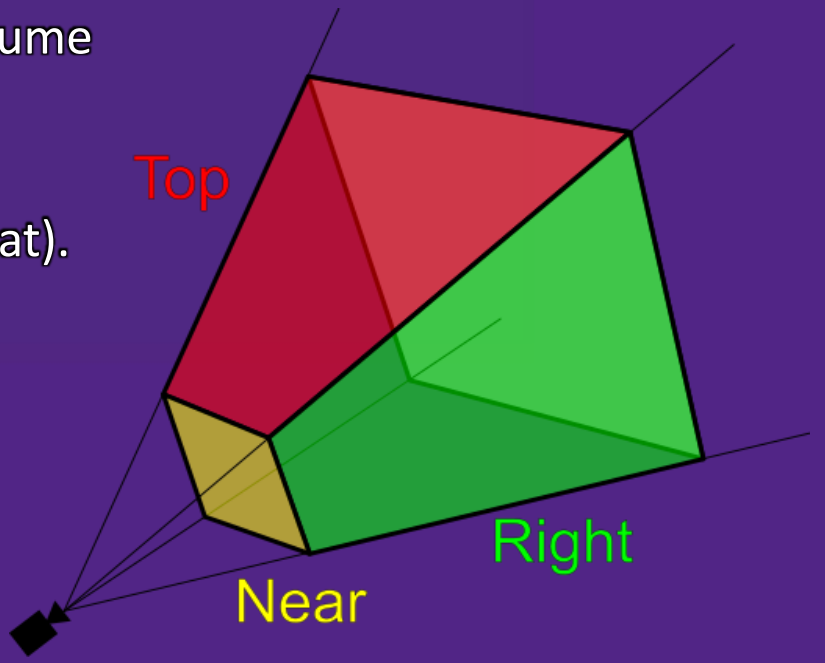


Perspective Projections

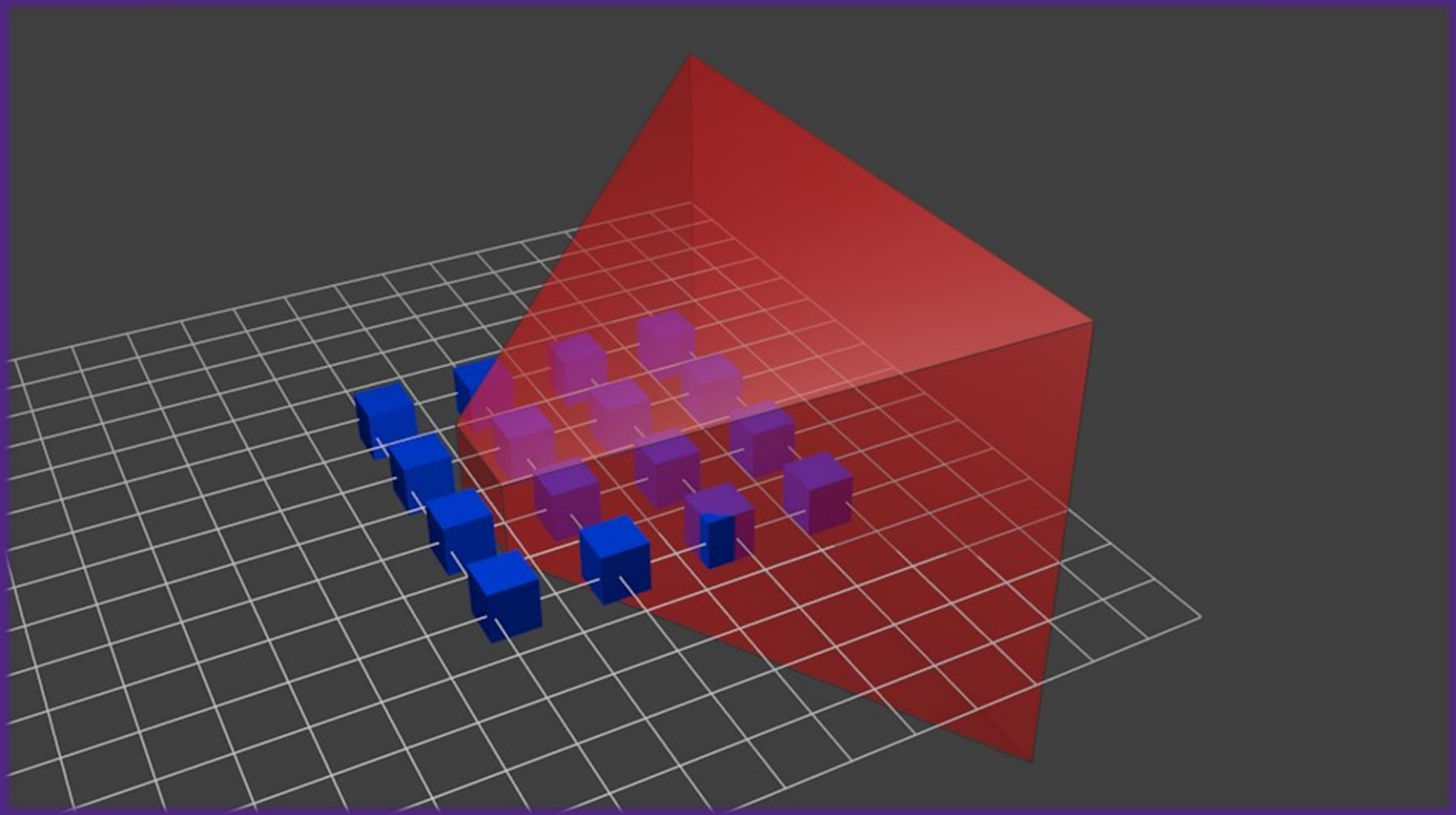
- We take this frustum, where the smaller side is nearer to the viewer, and transform it to NDC, forming a cube.

- The result is that objects closer to the viewer expand, and objects further away shrink.
- The matrix which defines this viewing volume is the projection matrix P.
- It transforms from camera space to NDC. (Actually clip space, but we'll return to that).

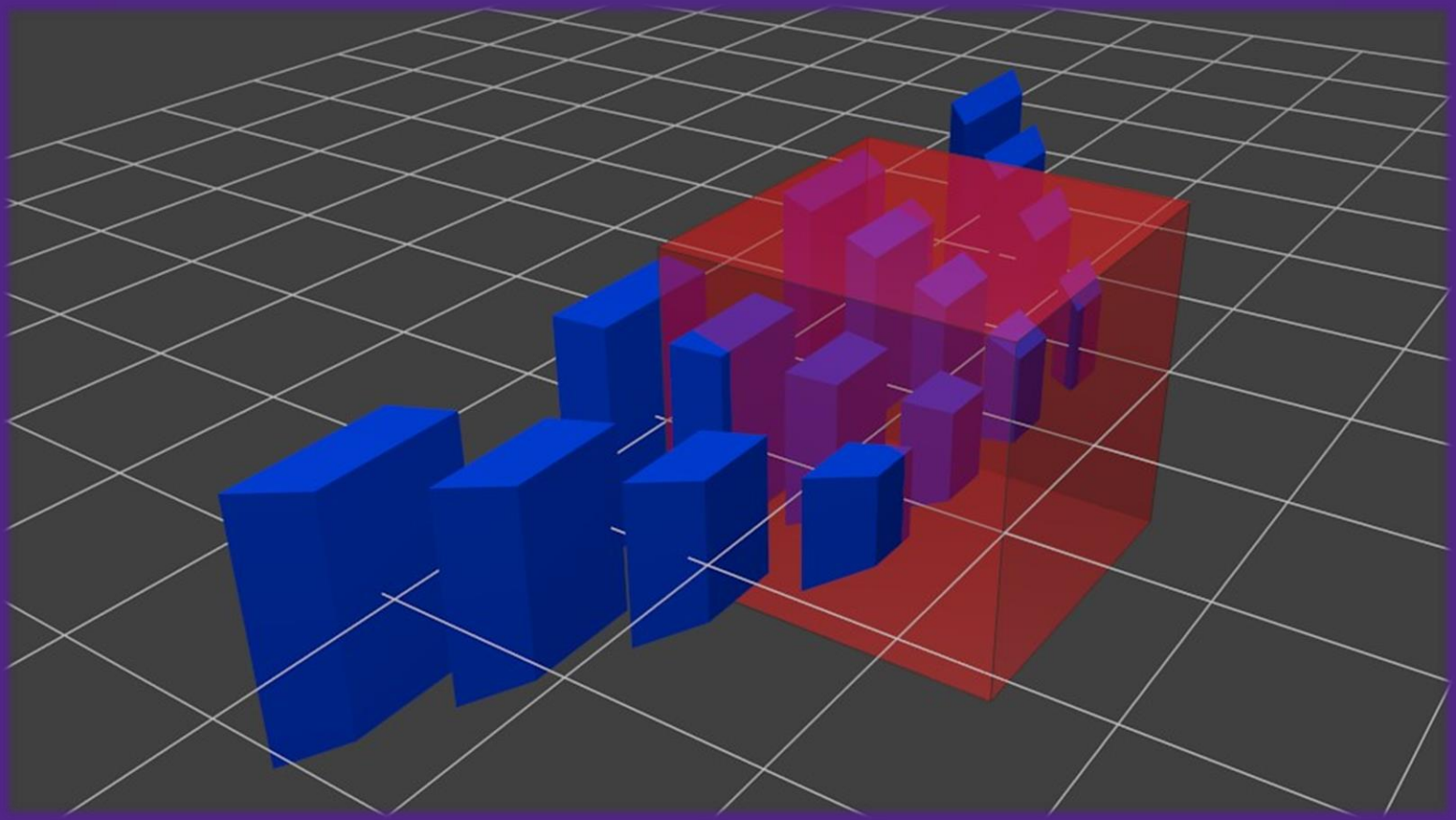
$$PVMv_{local} = v_{NDC}$$



Perspective Projections



Perspective Projections



Perspective Projections

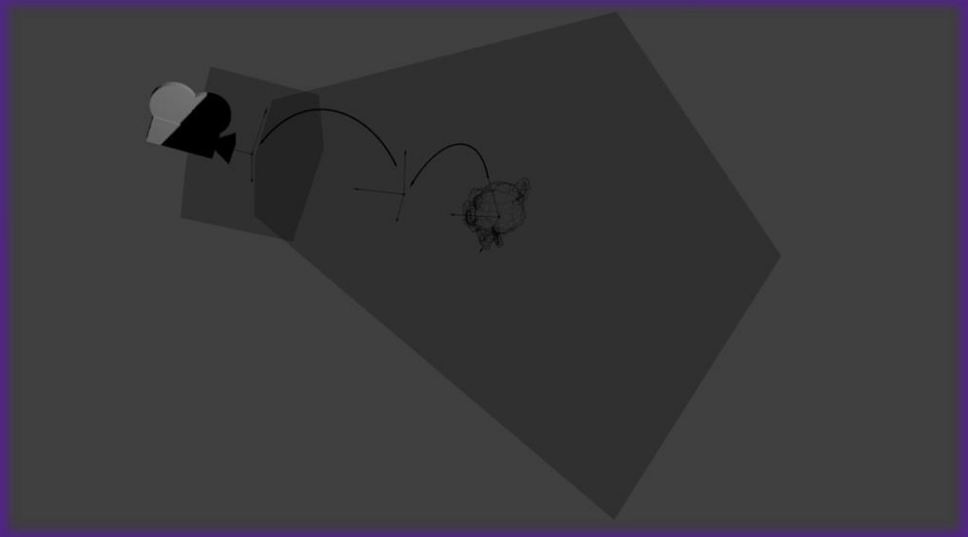
- To specify this frustum we require a few parameters:
- **Near clipping plane**: the plane perpendicular to the x-y plane at z = near which defines the front of the frustum.
- **Far clipping plane**: the plane perpendicular to the x-y plane at z = f ar which defines the back of the frustum.
- **Field of view**: The angle between the lines of projection. Typically this is measured in the vertical direction. 45°is a reasonable number.
- **Aspect ratio**: This defines the square/rectangle shape of the near and far ends of the frustum. Since we have a vertical field of view, this specifies what fraction the vertical view is compared to the horizontal view.

Perspective Projections

- The glm (OpenGL math) library does the work for us:

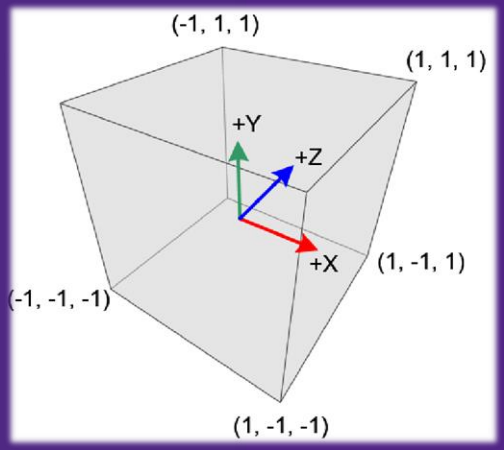
```
glm::mat4 P = glPerspective(fovy, aspect, near, far);
```

- Note: we specify the near and far clipping planes to have positive values. That is, they measure the distance of the plane from the camera.
- However, in reality, that are at negative z coordinates since the camera is looking down the negative z axis.



Clean-Up to 3D-Up

- This previous figure shows something interesting. Notice that in NDC the z-axis is now facing the opposite direction!
- Indeed, Normalized Device Coordinates is a left-handed system.



- We had a cube drawn with opposite corners (0,0,0) and (1, 1, -1).

- We saw that without perspective projection (i.e. the projection matrix was the identity), we saw the front face of the cube, at z = 0.

Clean-Up to 3D-Up

- We said this was because the camera sits at the origin and looks down the negative z axis.
- So, the z = 0 face is obviously closer to the camera, and is therefore the face seen, compared to the back side of the cube at z = –1.

- That was a *lie*.

Clean-Up to 3D-Up

- In reality, the Normalized Device Coordinates are rendered as follows. The x and y coordinates directly map to window coordinates using the viewport matrix.

- So, all values of x between [1,1] get mapped to window coordinates $[x_0, x_0 + w]$ (see the viewport lecture for notations).
  Similarly, y between [-1,1] get mapped to
  $[y_0, y_0 + h]$.

- What about z? The z = −1 plane is the “near” plane and the z = 1 plane is the “far plane”.

- So, the objects with z = −1 should get rendered “in front of” objects with z > −1.

Clean-Up to 3D-Up

- However, this is only true if you tell OpenGL to do depth testing.
- Otherwise, OpenGL ignores the z component of NDC entirely. Whatever is drawn last is rendered “on top”.
- So, return to the cube with one face at z = 0 and one face at z = −1.
- With a projection matrix being the identity matrix, the cube’s face at z = −1 is actually closer to the viewer than the cube’s face at z = 0.

Clean-Up to 3D-Up

- So the camera is not really at the origin and looking down the negative z-axis.
- If we think of the camera as what is “seen” and what will be drawn, then the camera is at (0, 0, –1) in NDC and looks down the positive z-axis
- Yet, in another point of view, the camera is indeed at the origin and looking down the z-axis.
- Why? It comes from the perspective frustum and clip space.

Clip Space

- It is not the case that

$$PVMv_{local} = v_{NDC}$$

- In fact, we are working with

$$PVMv_{local} = v_{clip}$$

Clip Space

- What is clip space you ask? Let us look more closely at the projection matrix and what actually happens.

- For a frustum which is symmetric (it's left and right sides are the same distance from the z-axis; it's top and bottom sides are the same distance from the z-axis), a perspective projection matrix is defined as:

$$\begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

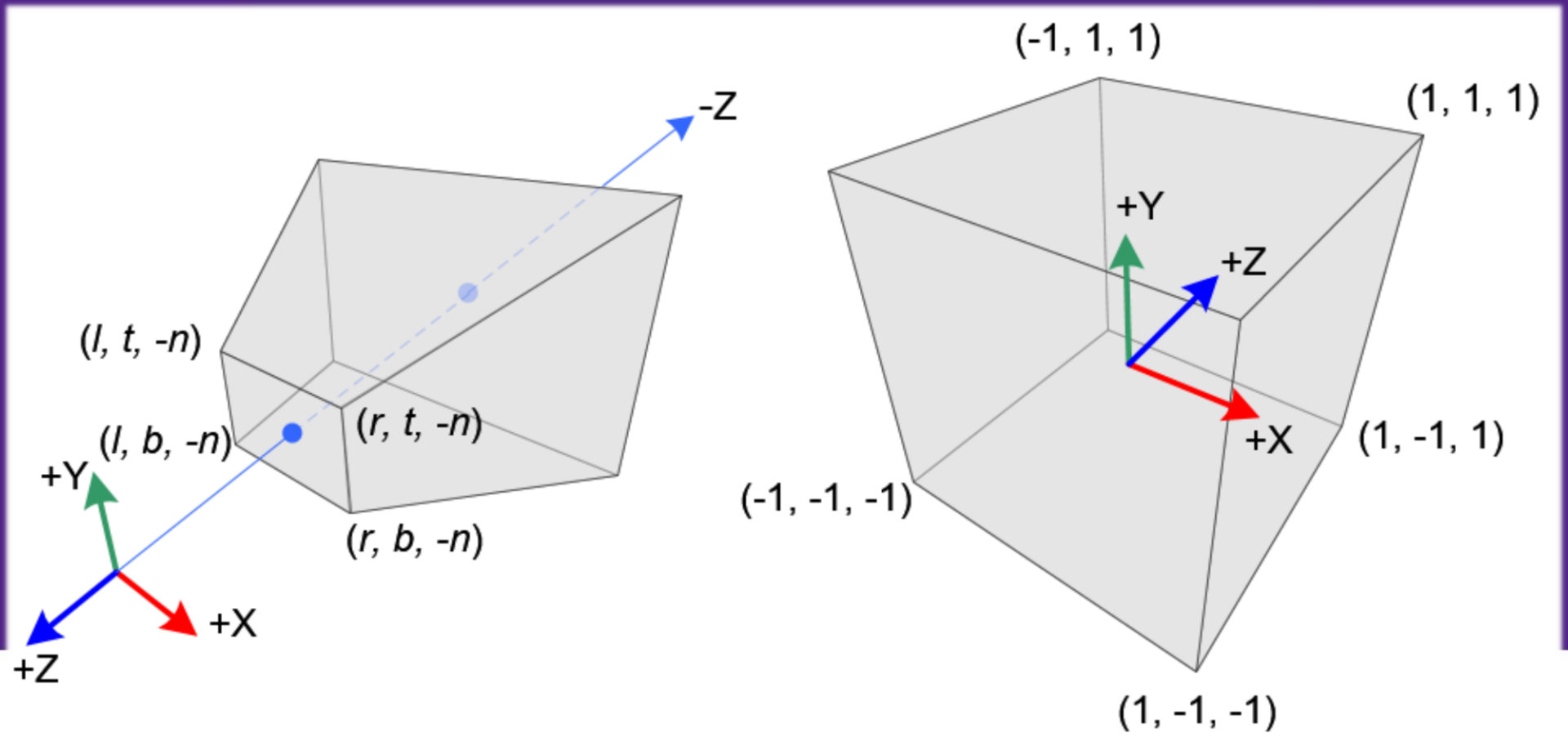
Clip Space

$$\begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- where the corners of the near side of the frustum has corners:
- *(–r, t, –n), (r, t, –n), (r, –t, –n), (–r, –t, n)*

Clip Space

- where the corners of the near side of the frustum has corners: *(−r, t, −n), (r, t, −n), (r, −t, −n), (−r, −t, n)*
- *In the below diagram, b = −t, l = −r.*



Clip Space

- With this view of the projective frustum, the camera is indeed at the origin and looking down the negative z-axis.
- Just as the diagram on the left of the previous figure shows.
- But, we have one automatic/invisible step between going from frustum to NDC.
- Notice that the perspective projection matrix we showed earlier is the first matrix which modifies the 4th homogeneous coordinate.

Clip Space

- In homogeneous coordinates, we have 4 coordinates: (x, y, z, w).
- Much like in two dimensional homogeneous coordinates w = 1 encodes a “point” and w = 0 encodes a “direction”.
- Thus, a point with w = 1 can be translated while a direction cannot.
- When we multiply a point (x, y, z, 1) by the perspective projection matrix, we get a point in clip space.

Clip Space

- In particular, notice that $w_{clip} = -z_{camera}$ (check the 4th row of the projection matrix above).
- Thus, a larger w means further away from the camera since, this time around, we are again thinking of the camera being at the origin and looking down the negative z axis.
- It is within clip space that clipping occurs.
- That is, we remove any vertices which would become outside NDC. The key is perspective division.

Clip Space

- In particular, notice that $w_{clip} = -z_{camera}$ (check the 4th row of the projection matrix above).
- Thus, a larger w means further away from the camera since, this time around, we are again thinking of the camera being at the origin and looking down the negative z axis.
- It is within clip space that clipping occurs.
- That is, we remove any vertices which would become outside NDC. The key is perspective division.

Clip Space

We calculate NDC from clip space like this:

$$\begin{bmatrix} x_{NDC} \\ y_{NDC} \\ z_{NDC} \end{bmatrix} = \begin{bmatrix} x_{clip} \,/\, w_{clip} \\ y_{clip} \,/\, w_{clip} \\ z_{clip} \,/\, w_{clip} \end{bmatrix}$$

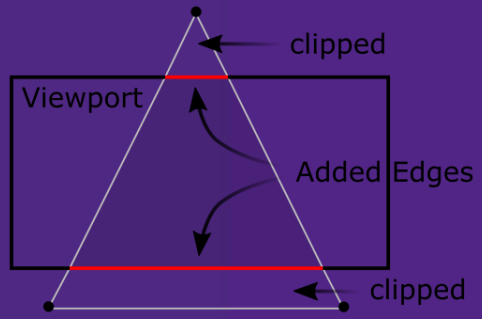
Dividing by w is called the perspective divide. Now, we know NDC is defined to be [-1,1]. And we compute coordinates in NDC by perspective division. Therefore, a coordinate x_{clip} is *inside* the [-1,1] NDC volume if $-w_{clip} < x_{clip} < w_{clip}$.

So, it is easy to test which vertices to keep. We only keep vertices which have all three:

$$-w_{clip} < x_{clip} < w_{clip}$$
$$-w_{clip} < y_{clip} < w_{clip}$$
$$-w_{clip} < z_{clip} < w_{clip}$$

Clip Space

- When primitives have vertices partly outside and partly inside, OpenGL clips the primitive so that its new vertex is directly fits in NDC.



- For the curious, here is a special detail...
  Each vertex belongs in its own clip space.

- What does that mean? Each vertex has its own value of w_{clip}.

Clip Space

- So the test for whether or not to clip a vertex is dependent on that vertex itself and its particular value of w_{clip}.
- Every vertex belongs in a “shared” camera space.
- Every vertex belongs in a “shared” NDC.
- But, to get from camera space to NDC, each vertex passes through its own clip space and checks against its own w_{clip} to find out if it gets clipped or not.

Vanishing Points

- Under perspective projection, parallel lines in 3D do not map to parallel lines in 2D, except in one case when the lines are parallel to the viewing plane.
- Otherwise, parallel lines, once perspective projected, will meet at a point called the vanishing point.
- Suppose a 3D line passes through a point $A=(X,Y,Z)^T$ with unit direction vector $\vec{n}=(n_x, n_y, n_z)^T$.
- In parametric form, this line is written as $P(t)=A+\vec{n}t$. If we project this line onto the viewing plane, we obtain:

$$p(t) = \left(N\frac{X+n_x t}{-(Z+n_z t}, N\frac{Y+n_y t}{-(Z+n_z t)} \right)^T$$

Vanishing Points

- Also suppose $P(t)$ is not parallel to the viewing plane and $n_z>0$, which make the line go away from the camera position as t increases.

- If we take the following limit:

$$\lim_{t\to\infty} p(t)$$

- then we find it is equal to:

$$\left(-N\frac{n_x}{n_z}, -N\frac{n_y}{n_z}\right)^T$$

- which is the vanishing point onto the viewing plane

Vanishing Points & Perspective, From an Artist's View

Pseudo-Depth

- When a perspective projection is performed, the depth information is lost.
- This prevents us from removing hidden surfaces while rendering objects.
- We choose to keep a quantity we call pseudo-depth, which is easy to obtain and is sufficient to sort surfaces with respect to their depth in the viewing volume.
- To accomplish this, we make any point P project to $(X', Y', Z')^T$, where Z' is pseudo-depth:

$$\begin{bmatrix} X' \\ Y' \\ Z' \end{bmatrix} = \begin{bmatrix} -N\frac{X}{Z} \\ -N\frac{Y}{Z} \\ \frac{-(aZ+b)}{Z} \end{bmatrix}$$

Pseudo-Depth

- Where a and b are constants chosen such that:

$$-1 \leq \frac{-(aZ + B)}{Z} \leq 1$$

- In other words, we keep the pseudo-depth values comprised between -1 and 1 for Z between –N and –F. In order to do this, we set the constants as:

$$a = -\frac{F + N}{F - N}, \qquad b = \frac{-2FN}{F - N}$$

Pseudo-Depth

- These constants are obtained as follows

When Z = -N, we require that:

$$\frac{-(aZ + B)}{Z} = -1$$

Therefore,

$$\frac{-(a(-N) + b)}{-N} = -1$$

Which in turn implies:

$$a = \frac{N + b}{N}$$

Pseudo-Depth

- These constants are obtained as follows

When Z = -F, we require that:

$$\frac{-(aZ + B)}{Z} = 1$$

Therefore,

$$\frac{-(a(-F) + b)}{-F} = 1$$

Which in turn implies:

$$a = \frac{b - F}{F}$$

Pseudo-Depth

- We now have two expressions for a that we equate in order to find b:

$$\frac{b-F}{F}=\frac{N+b}{N}$$

Isolating b, we obtain:

$$b=\frac{-2FN}{F-N}$$

and using

$$a=\frac{N+b}{N}$$

substituting in b yields

$$a=\frac{-(F+N)}{F-N}$$

Pseudo-Depth

- Now, this perspective transformation of a point p_v in homoegenous coordinates can be written as:

$$M_p p_v = \begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} NX \\ NY \\ aZ + b \\ -Z \end{bmatrix}$$

Dividing the result by the last element (-Z), we obtain:

$$\begin{bmatrix} -N\frac{X}{Z} \\ -N\frac{Y}{Z} \\ \frac{-(aZ+b)}{Z} \\ 1 \end{bmatrix}$$

Pseudo-Depth

- In general, with perspective projections, we have as many finite vanishing points as there are intersections between the coordinate axes and the extended viewing plane.

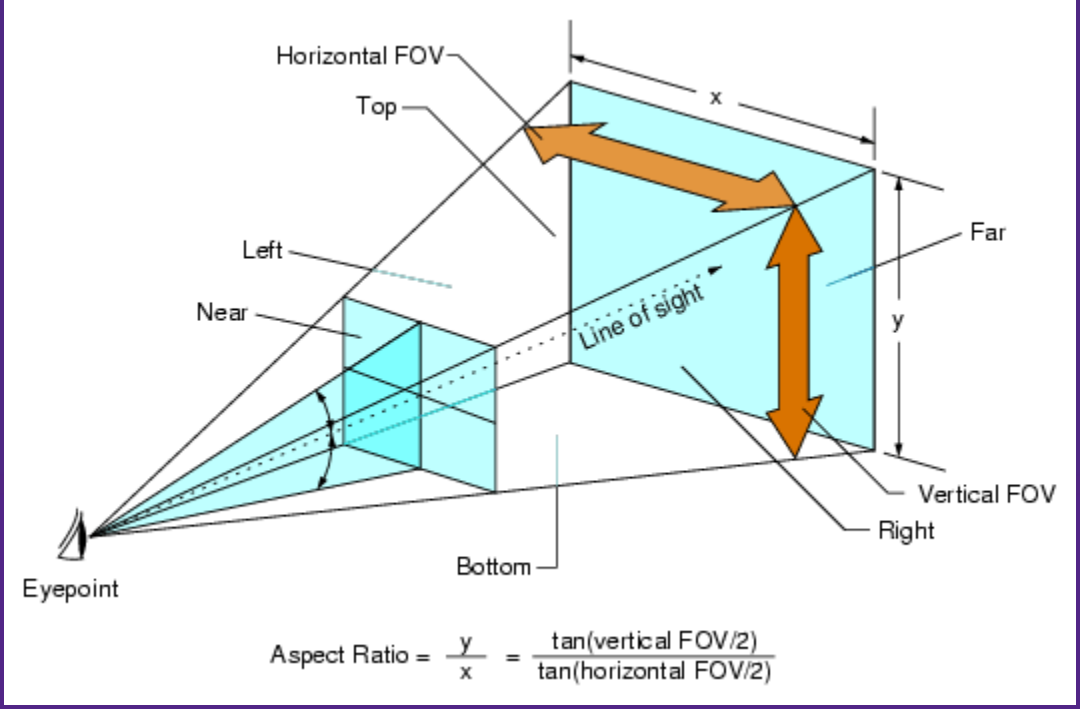


Illustration 2: The viewing volume with the camera as the eyepoint

Other Projections

- Perspective projections are most common because this is how humans see.
- However, for purposes other than realism, other forms of projections may be useful, such as these that preserve proportions, or display objects from particular angles, etc.
- Here is a brief description of some of these projections:
- **Orthographic Projections:** In this type of projection, the direction of the projection is perpendicular to the viewing plane. In other words, this projection is realized by simply removing the third coordinate of the point to be projected.

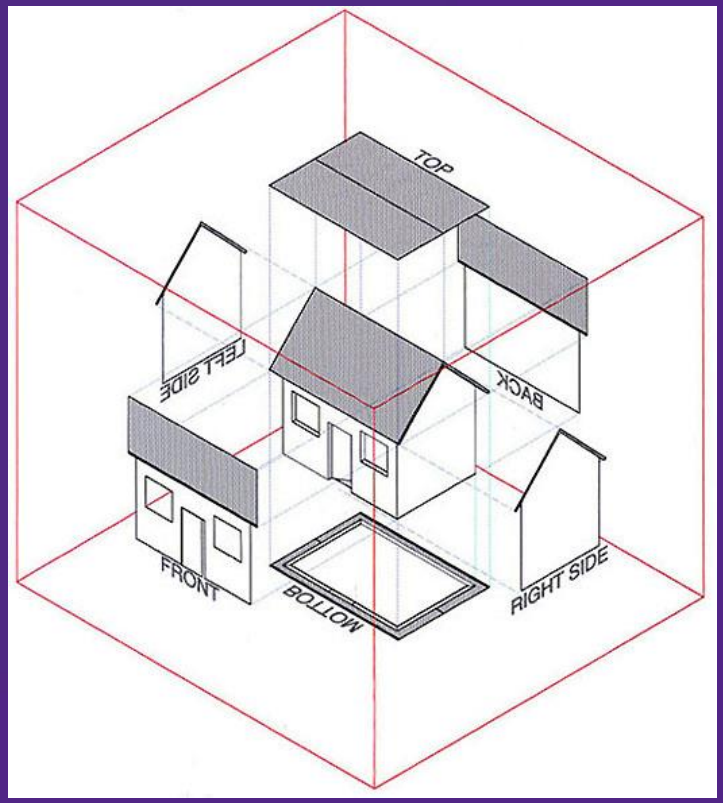


Illustration 3: Orthographic projection of a house

Other Projections

- **Isometric Projections:** This projection is obtained by aligning the viewing plane in away such that it intersects each coordinate axis at the same distance as the origin.

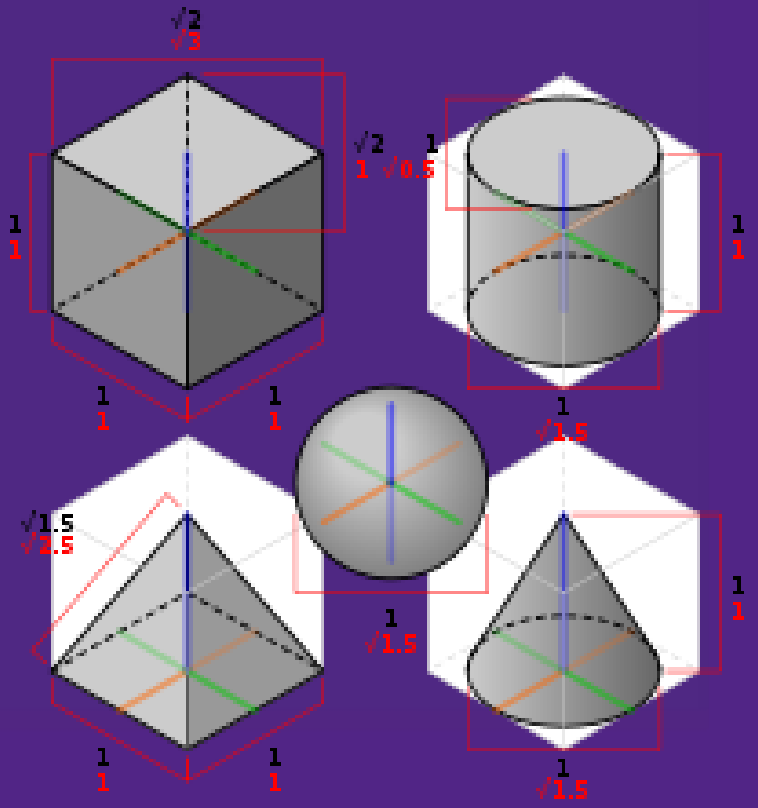


Illustration 4: Isometric projections of some common shapes

Other Projections

- **Oblique Projections:** Obtained by projecting points along parallel lines that are not perpendicular to the viewing plane.

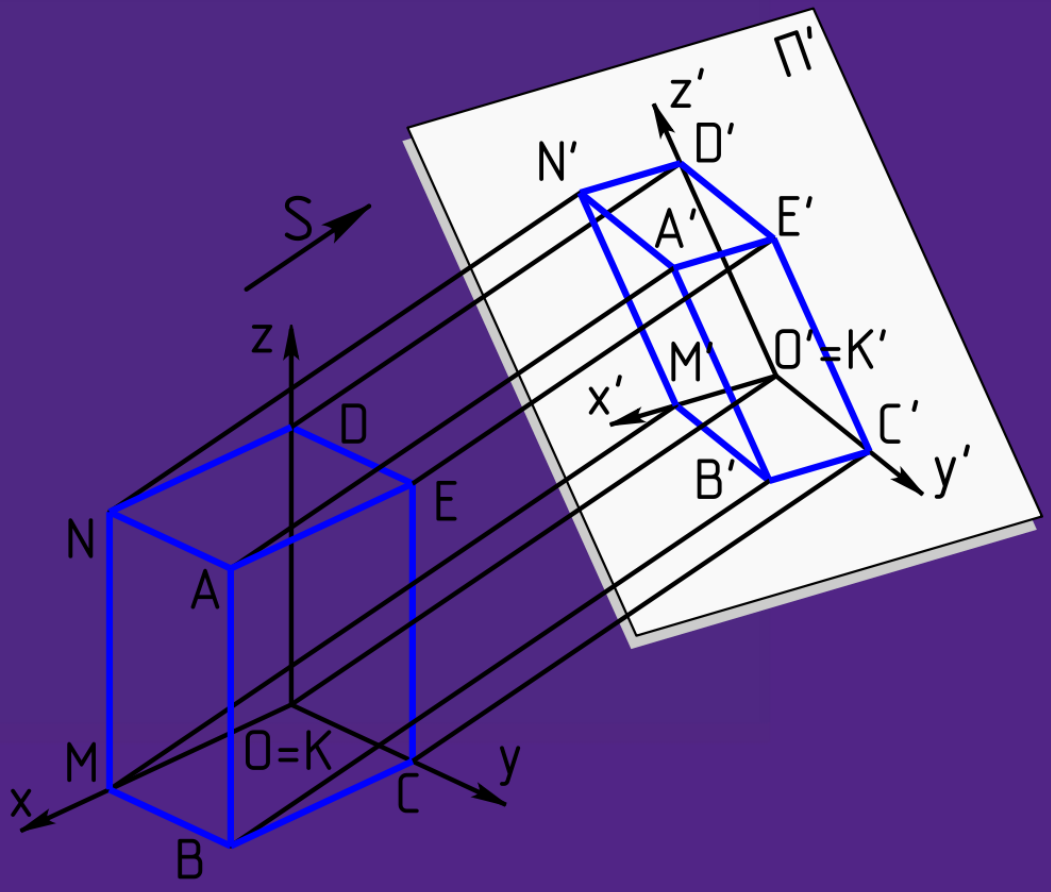


Illustration 5: Oblique projection

Camera pt 2

- The camera (or what we conceptually think of as the camera) is extremely important to graphics.
- It's worth revisiting. In one point of view the camera always exists at the origin and look down the negative z-axis.
- In another point of view, we can actually place the camera in world coordinates as if it was an object or any other model.

Camera pt 2

- The latter point of view is most practical.
- The only trick is that we have to remember that our viewmatrix is defining how to get from world coordinates to camera coordinates.
- So, if we want to think of the camera as an object in the world coordinates, it must be that the view matrix is the inverse matrix of the camera's world position.

Camera pt 2

- Let's say we want to position the camera so that it is placed at (e_x, e_y, e_z) , (*e* for "eye").
- The camera should then be rotated to look at (t_x, t_y, t_z) (*t* for "target").
- Our eventual *V* matrix needs to this have a translation, to put the camera at (e_x, e_y, e_z) in world coordinates, and then some sort of rotation to make it face the target.
- Let's start with translation, that's easy.
- Again, we want the *V* matrix to actually encode the inverse of the camera transformation. Whats the inverse of translation? Just multiply by –1.

Camera pt 2

$$\begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Now we need to specify the rotation. We could figure out all the elementary rotations needed to get the camera to point as the target.
- Or, we could consider another view point for transformation matrices.
- Recall from Lecture 4 that one way of understanding transformation matrices is not how they transform points from one coordinate system to another, but rather how it transforms the whole coordinate system of one space to become the coordinate system of the destination space.

Camera pt 2

- In this case, our destination space is camera space, where the camera is at the origin and looking down the negative *z* axis.
- In camera space, this negative *z* axis has an associated direction $-\hat{k} = (0,0,-1)$.
- Therefore, in camera space, the positive *z* axis is defined as $\hat{k} = (0,0,1)$.
- Relative to the world coordinate system, this $\hat{k}_{camera}$ has an associated vector cz_{world}, the camera's positive z-axis, specified in world coordinates.

Camera pt 2

- What is that direction?
- Well we have an eye position and a target position we want to look at.
- The direction the camera is looking is thus a displacement vector between the target and the correct eye position.

$$\vec{d} = \vec{e} - \vec{t} = \begin{bmatrix} e_x - t_x \\ e_y - t_y \\ e_z - t_z \end{bmatrix}$$

- This gives a vector from $^{->}$t towards $^{->}$e; which is what we want, as it makes the camera look down *negative z*. The difference between e and t describes the positive z axis of the camera space.
- We then normalize by dividing d by its magnitude.

Camera pt 2

- So we've now described the z axis of the camera's coordinate system relative to the world coordinates.
-
- We do the same for the x and y axes. It follows a simple trick that is a consequence of 3D rotations.

- To "look" in a particular direction leaves you still with one degree of freedom.

Camera pt 2

- To see why, pick any point on a the wall in the room you are currently in. Look at it.
- Rotate your head side-to side while still looking at that point.
- What happens? Your vision rotates around that point.
- This is exactly the axis-angle formulation of 3D rotations.
- The look direction gives us the axis.
- Now we have to specify the angle.

Camera pt 2

- One way to specify this angle is to specify the proper x and y axes of the camera coordinate system.
- We do so with a simple trick. Look at your point on the wall again. This time, put the crown of your head facing toward the ceiling.
- Call this “upwards”. Now, to look at your point you may have to move your eyes a little bit up or down in their sockets.
- That’s okay, as long as we keep the crown of your head pointing upwards.
- If your head is the camera, your eyes are looking in the look direction and your right ear is facing in the direction of the positive x axis. Your ear points out to the right.

Camera pt 2

- Now, we will let our head drift away from the crown pointing directly up. Bring your chin up or down so that you're still looking at your point on the wall, but now your eyes are looking "straight ahead", but maybe your chin is no longer pointing parallel to the floor.

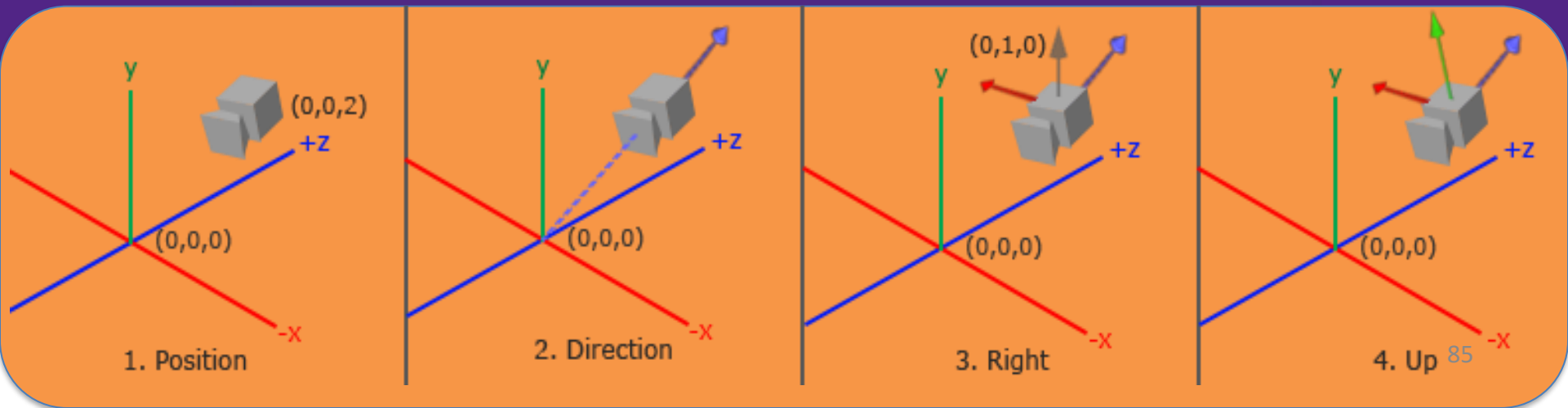
- That's okay. This is the key observation: your right ear hasn't moved! By tilting your chin up and down, you essentially rotate your head around the x axis (the axis coming out of your right ear).

- Once you've moved your chin so that the line of sight is "straight out of your head", the crown of your head now doesn't point directly up toward the ceiling, but the crown of your head still points directly "upwards" in the coordinate system relative to your eye balls.

Camera pt 2

- In summary:

1. Specify a position of the camera in world space.
2. Specify a look direction ^d as the normalized $\vec{e} - \vec{t}$.
3. Let the "up" direction be directly up toward the ceiling and then let your right ear point out toward the positive x axis.
4. Rotate around this x axis so that your line of sight returns to be ^d and that the crown of your head is no longer pointing directly upward.



Camera pt 2

- Now, how do we actually compute steps 3 and 4?
  Cross products!

- Given ˆd, we compute ˆr as the cross product between ˆd and (0, 1, 0), that is, the cross product between ˆd and “directly up” in world coordinates.

- Then, we compute ˆd × ˆr to get the “up” direction relative to the camera, call it ˆup .

- Phew. That was a lot. Put it all together now:

Camera pt 2

- Now, how do we actually compute steps 3 and 4?
  Cross products!

- Given ˆd, we compute ˆr as the cross product between ˆd and (0, 1, 0), that is, the cross product between ˆd and “directly up” in world coordinates.

- Then, we compute ˆd × ˆr to get the “up” direction relative to the camera, call it ˆup .

- Phew. That was a lot. Put it all together now:

Camera pt 2

$$\begin{bmatrix} \widehat{r_x} & \widehat{r_y} & \widehat{r_z} & e_x \\ \widehat{up_x} & \widehat{up_y} & \widehat{up_z} & e_y \\ \widehat{d_x} & \widehat{d_y} & \widehat{d_z} & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Ta-da! This is the view matrix that (conceptually) positions the camera at location (e_x, e_y, e_z) in world coordinates and looks toward $\vec{t}$. Right? Not quite!
- Recall that the order of a transformations matters. This matrix above actually corresponds to rotating first and then translating.
- Will that give us what we want from a "look at" function? Nope. we need to move the camera to its position in the world first and then rotate it to look at what it should be looking at.

Camera pt 2

$$V = \begin{bmatrix} \widehat{r_x} & \widehat{r_y} & \widehat{r_z} & -\widehat{r_z}e_x - \widehat{r_y}e_y - \widehat{r_z}e_z \\ \widehat{up_x} & \widehat{up_y} & \widehat{up_z} & -\widehat{up_z}e_x - \widehat{up_y}e_y - \widehat{up_z}e_z \\ \widehat{d_x} & \widehat{d_y} & \widehat{d_z} & -\widehat{d_z}e_x - \widehat{d_y}e_y - \widehat{r_z}e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- What follows is very similar but a more math driven and perhaps less intuitive approach.

Synthetic Camera

- The synthetic camera allows the user to choose the viewing position in 3D.
- To create a camera, we define a coordinate system within the world coordinate system. This is accomplished by specifying a 3D position, a gaze vector, and an indication of where up is.
- With these three elements, it is possible to obtain the unit vectors of the coordinate system of the camera: $\vec{u}$, $\vec{v}$, $\vec{n}$.

Synthetic Camera

- To obtain $\vec{n}$, we need a few things. Pose *e* , the position of the origin of the camera coordinate system, *g* a point through which the gaze direction unit vector $\vec{n}$ points to, and $\vec{p}$, a unit vector in the direction in which up is.

- We can then obtain unit vector $\vec{n}$ of the coordinate system as:

$$\vec{n} = \frac{e - g}{\|e - g\|}$$

- It is natural to consider the vector $\vec{u}$ as a unit vector perpendicular to $\vec{n}$ and $\vec{p}$ and thus $\vec{u} = \vec{p} \times \vec{n}$. Further, $\vec{v} = \vec{n} \times \vec{u}$.

- One must normalize these vectors to unit length.

Camera Transformation Matrix

- The transformation matrix for the camera operates by transforming $\vec{i}$, $\vec{j}$, and $\vec{k}$ into $\vec{u}$, $\vec{v}$, and $\vec{n}$. Hence we can write:

$$M = \begin{bmatrix} \vec{u} & \vec{v} & \vec{n} & \vec{e} \\ \overrightarrow{0^T} & & & 1 \end{bmatrix}$$

- If this shorthand is unfamiliar, this is equivalent to:

$$M = \begin{bmatrix} u_x & v_x & n_x & e_x \\ u_y & v_y & n_y & e_y \\ u_z & v_z & n_z & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Camera Transformation Matrix

- The transformation matrix for the camera operates by transforming $\vec{}$i , $\vec{}$j , and $\vec{}$k into $\vec{}$u , $\vec{}$v , and $\vec{}$n . Hence we can write:

$$M = \begin{bmatrix} \vec{u} & \vec{v} & \vec{n} & \vec{e} \\ \overrightarrow{0^T} & & & 1 \end{bmatrix}$$

- Where $\vec{e}$ is the location of the origin of the camera coordinate system. Observe that M$\vec{}$i = $\vec{}$u + $\vec{}$e, M $\vec{}$j = $\vec{}$v+ $\vec{}$e and M $\vec{}$k = $\vec{}$n+ $\vec{}$e, where

$$\vec{\imath} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \qquad \vec{\jmath} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}, \qquad \vec{k} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Camera Transformation Matrix

- Hence a point p_v expressed in the camera (or viewing) coordinates can be transformed into a point in the world coordinate system p_w by computing
- $M\,p_v = p_w$. The inverse transformation also exists and is simply $p_v = M^{-1}\,p_w$.
- Note that M is a composite transform created with a rotation matrix and a translation matrix:

$$M = TR = \begin{bmatrix} [I] & \vec{e} \\ \vec{0}^T & 1 \end{bmatrix} \begin{bmatrix} \vec{u} & \vec{v} & \vec{n} & \vec{0} \\ & \vec{0}^T & & 1 \end{bmatrix} = \begin{bmatrix} u_x & v_x & n_x & e_x \\ u_y & v_y & n_y & e_y \\ u_z & v_z & n_z & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Camera Transformation Matrix

- The inverse matrices for the rotation and the translation are:

$$R^{-1} = \begin{bmatrix} \vec{u}^T & \\ \vec{v}^T & \vec{0} \\ \vec{n}^T & \\ \vec{0}^T & 1 \end{bmatrix}$$

and

$$T^{-1} = \begin{bmatrix} [I] & -\vec{e} \\ \vec{0}^T & 1 \end{bmatrix}$$

Camera Transformation Matrix

- Hence:

$$M_v = M^{-1} = (TR)^{-1}R^{-1}T^{-1} = \begin{bmatrix} \vec{u}^T & -\vec{e}\cdot\vec{u} \\ \vec{v}^T & -\vec{e}\cdot\vec{v} \\ \vec{n}^T & -\vec{e}\cdot\vec{n} \\ \vec{0}^T & 1 \end{bmatrix}$$

Camera Transformation Matrix

- Given a point in world coordinates, we now can transform it into the camera (viewing) coordinate system, and apply the perspective transformation (including pseudo-depth). If p_w is a point in world coordinates, then its representation in the camera coordinates including its perspective transformation is given by

$$p_v = M_p M_v p_w$$

- The near plane extends infinitely and so does the viewing volume. We need to specify four points on the near plane that will limit the extent of it. We define the points as

$$p_1 = (l, t, -N),\ p_2 = (r, t, -N),\ p_3 = (r, b, -N),\ p_4 = (l, b, -N)$$

- Generally, we let $l = -r$ and $b = -t$. The points form a rectangle on the near plane that, along with the far plane, define the extent of the viewing volume.

Camera Transformation Matrix

- Given a point in world coordinates, we now can transform it into the camera (viewing) coordinate system, and apply the perspective transformation (including pseudo-depth). If p_w is a point in world coordinates, then its representation in the camera coordinates including its perspective transformation is given by

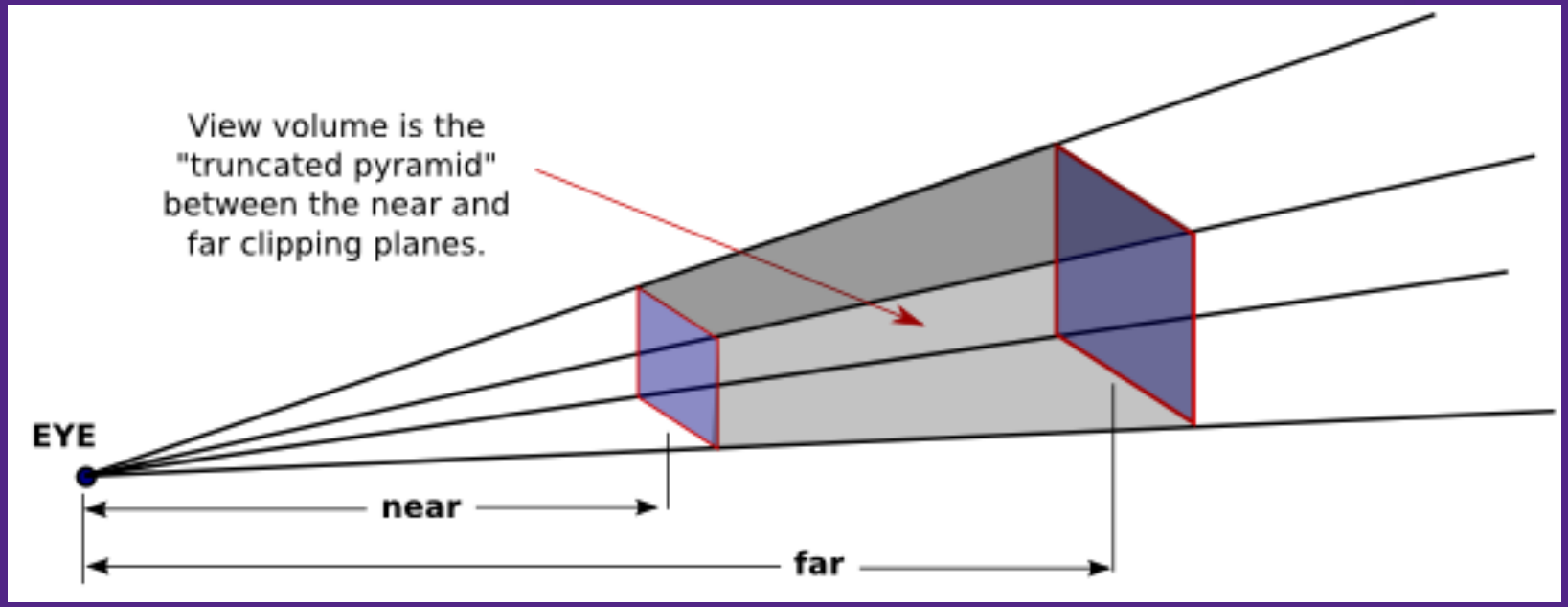
$$p_v = M_p M_v p_w$$

- The near plane extends infinitely and so does the viewing volume. We need to specify four points on the near plane that will limit the extent of it. We define the points as

$$p_1 = (l, t, -N),\ p_2 = (r, t, -N),\ p_3 = (r, b, -N),\ p_4 = (l, b, -N)$$

- Generally, we let $l = -r$ and $b = -t$. The points form a rectangle on the near plane that, along with the far plane, define the extent of the viewing volume.

Camera Transformation Matrix



Camera Transformation Matrix

- We already have the pseudo depth comprised between -1 and 1. We now need to find the transformations that will do the same for the rest of the coordinates within the viewing volume. A scaling and a translation are needed to accomplish this:

$$S_1 = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad T_1 = \begin{bmatrix} 1 & 0 & 0 & \frac{-(r+l)}{2} \\ 0 & 1 & 0 & \frac{-(t+b)}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Camera Transformation Matrix

- At this point, we can transform a point in world coordinates into the canonical view volume by applying the following transformations:

$$p_v = S_1 T_1 M_p M_v p_w$$

where

$$S_1 T_1 M_p = \begin{bmatrix} \frac{2N}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2N}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(F+N)}{F-N} & \frac{-2FN}{F-N} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Camera Transformation Matrix

- We can also define the viewing volume by specifying a viewing angle θ and an aspect ratio A. In this case, we write:

$$t = N tan\left(\frac{\pi}{180}\frac{\theta}{2}\right)$$

and

b = -t r = At, l = -r

Camera Transformation Matrix

- We now need to transform objects from the warped viewing volume into screen coordinates.
- A scaling and translation are needed to perform this operation. Suppose the screen window has its upper left corner located at $(0,0)$ with width w and height h.
- First, we need to translate the $(x\ ,\ y)$ coordinates to the positive quadrant:

$$T_2 = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Camera Transformation Matrix

- Next, we need to scale in such a way that these translated coordinates now fit in a space defined by (0…w, 0…h):

$$S_2 = \begin{bmatrix} \frac{w}{2} & 0 & 0 & 1 \\ 0 & \frac{h}{2} & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Camera Transformation Matrix

- These two transforms would be sufficient if the origin of the window was at its bottom left corner. However, in screen coordinates, the origin is at the top left corner and we need to compose a last transform which takes care of this:

$$W = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & h \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Camera Transformation Matrix

- Composing these transformations together yields:

$$WS_2T_2 = \begin{bmatrix} \frac{w}{2} & 0 & 0 & \frac{w}{2} \\ 0 & \frac{-h}{2} & 0 & \frac{-h}{2} + h \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- We are now in a position to transform a 3D point in world coordinates into the image coordinates of the window by performing the following suite of transformations:

$$p_s = WS_2T_2S_1T_1M_pM_vp_w$$

- The last thing is to perform the perspective projection in order to obtain the point in pixel coordinates

Euler (Gimbal) Lock