Computer Graphics

P. Healy

CS1-08
Computer Science Bldg.
tel: 202727
patrick.healy@ul.ie

Spring 2021–2022

Outline

Outline

Transforming from World to View Co-ordinates, §10-4

- We want to convert vertices in ☐ co-ordinates to ☐ co-ordinates
- This comprises two steps:
- firstly, take account of the separation between the ☐ origin and the ☐ origin (T)
- then, take account of mis-alignment of two sets of axes (R)

1. Translate viewing-coordinate origin (located at $p_0 = (x_0, y_0, z_0)$ in world coords) to world origin

$$T = \begin{pmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Transforming from World to View Co-ordinates, §10-4

1. Apply $\boxed{}$ to align the view axes (x_v, etc.) with world axes (x_w, etc)
   We can use the viewing unit vectors – which are given in terms of world coordinates – we calculated in prev. lect.

$$R = \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

World to Viewing Co-ordinates (contd.)

Reminder of notation:
we combine u_x, u_y and u_z into the vector u (and likewise for v and w) and write it as

$$u = \begin{pmatrix} u_x \\ u_y \\ u_z \\ 1 \end{pmatrix}$$

Also, u^T "u transpose" is u turned on its side;
$u^T = (u_x, u_y, u_z, 1)$
To multiply two vectors using inner product:

$$u^T v = u_x v_x + u_y v_y + u_z v_z + 1 \times 1$$

World to Viewing Co-ordinates (contd.)

- Then the matrix that transforms world coords to viewing coords is (see also pp. 246-248 (and p. 234))

$$\begin{aligned} M_{w,v} &= R \cdot T \\ &= \begin{pmatrix} u_x & u_y & u_z & -u^T p_0 \\ v_x & v_y & v_z & -v^T p_0 \\ n_x & n_y & n_z & -n^T p_0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

where, *e.g.*, $u^T p_0 = u_x x_0 + u_y y_0 + u_z z_0 + 1 \times 1$

World to Viewing Co-ordinates (contd.)

- What does the point $p_0 = (x_0, y_0, z_0)$ in world coords get mapped to by $\boxed{}$ transformation?
- Transforming the point corresponds to multiplying by the composite matrix $M_{w,v}$

$$M_{w,v}p_0 = \begin{pmatrix} u_x & u_y & u_z & -u^T p_0 \\ v_x & v_y & v_z & -v^T p_0 \\ n_x & n_y & n_z & -n^T p_0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{pmatrix}$$

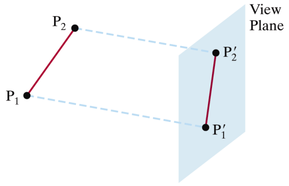
$$= \begin{pmatrix} u_x x_0 + u_y y_0 + u_z z_0 - u^T p_0 \\ v_x x_0 + v_y y_0 + v_z z_0 - v^T p_0 \\ n_x x_0 + n_y y_0 + n_z z_0 - n^T p_0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

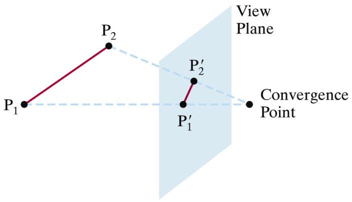
- So $p_0 = (x_0, y_0, z_0)$ in world coords $\rightarrow (0, 0, 0)$ in view coords, the centre of our universe – as we would expect

Outline

Projection Transformations – §10-5

- Once the view and view are decided upon we need to **project** scene on to the plane





- We can generate **elevation**, **end elevation** and **plan** views with appropriate choices of N
- The location of view plane along z_v-axis may affect the projection in **perspective** (right) case

Outline

Orthogonal Projections – §10-6

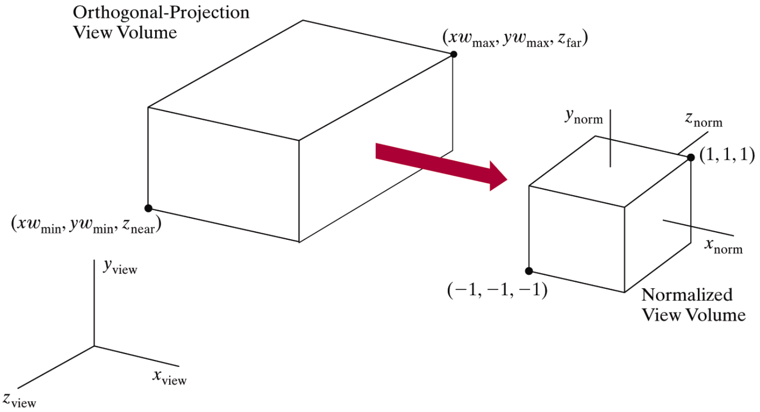
- In an orthogonal projection the scene gets projected on to view plane with lines perpendicular (orthogonal) to the view plane
- Although it gets mapped to a rectangle, in 3-D the clipping window corresponds to an infinitely long pipe of rectangular cross-section
- We place limits parallel to view plane along z_v axis to limit this size; this is called a **parallelepiped**
- So for orthogonal projections we talk about a **view volume**

Oblique Parallel Projections – §10-7

- Please read §10-7 of *H-B*

View Volume Transformation

What is the transformation matrix for transformation below?



View Volume Transformation

- View volume is RH coordinate system while normalized view is LH system
- This reversal is done to match "screen" view of depth where further-away objects have larger z-value ($z_{\text{near}} < z_{\text{far}}$); in our view coords, the opposite was case
- In 2-D we derived a matrix to transform from one rectangular window, $(c_x, c_y) \rightarrow (C_x, C_y)$ to another, $(n_x, n_y) \rightarrow (N_x, N_y)$ and used it to derive, $M_{C,NS}$

View Volume Transformation

- View volume is RH coordinate system while normalized view is LH system
- This reversal is done to match “screen” view of depth where further-away objects have larger *z*-value ($z_{\text{near}} < z_{\text{far}}$); in our view coords, the opposite was case
- In 2-D we derived a matrix to transform from one rectangular window, $(c_x, c_y) \rightarrow (C_x, C_y)$ to another, $(n_x, n_y) \rightarrow (N_x, N_y)$ and used it to derive, $M_{C,NS}$

View Volume Transformation

- View volume is RH coordinate system while normalized view is LH system
- This reversal is done to match “screen” view of depth where further-away objects have larger z-value ($z_{\text{near}} < z_{\text{far}}$); in our view coords, the opposite was case
- In 2-D we derived a matrix to transform from one rectangular window, $(c_x, c_y) \rightarrow (C_x, C_y)$ to another, $(n_x, n_y) \rightarrow (N_x, N_y)$ and used it to derive, $M_{C,NS}$

View Volume Transformation (contd.)

$$M_{C,NS} = \begin{pmatrix} \frac{2}{C_x - c_x} & 0 & -\frac{C_x + c_x}{C_x - c_x} \\ 0 & \frac{2}{C_y - c_y} & -\frac{C_y + c_y}{C_y - c_y} \\ 0 & 0 & 1 \end{pmatrix}$$

- In 3-D, the transformation from view volume $(xw_{\text{min}}, yw_{\text{min}}, z_{\text{near}}) \to (xw_{\text{max}}, yw_{\text{max}}, z_{\text{far}})$ to a normalized box $(-1, -1, 1) \to (1, 1, -1)$ is

$$M_{c,ns} = \begin{pmatrix} \frac{2}{xw_{\text{max}} - xw_{\text{min}}} & 0 & 0 & -\frac{xw_{\text{max}} + xw_{\text{min}}}{xw_{\text{max}} - xw_{\text{min}}} \\ 0 & \frac{2}{yw_{\text{max}} - yw_{\text{min}}} & 0 & -\frac{yw_{\text{max}} + yw_{\text{min}}}{yw_{\text{max}} - yw_{\text{min}}} \\ 0 & 0 & \frac{-2}{z_{\text{near}} - z_{\text{far}}} & \frac{z_{\text{near}} + z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Transformation from World Coordinates to Device

- We saw previously world to view transformation

$$M_{w,v} = \begin{pmatrix} u_x & u_y & u_z & -u \cdot p_0 \\ v_x & v_y & v_z & -v \cdot p_0 \\ n_x & n_y & n_z & -n \cdot p_0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- So transformation from world to normalized cube is the product of these two:

$$M_{w,\mathrm{ns}} = M_{c,\mathrm{ns}} \cdot M_{w,v}$$

- As with 2-D case, clipping now takes place
- Followed by...
    - ...visibility testing,
    - ...surface rendering (to be covered later),
    - ...view port transformation (transform to device coordinates)

Outline

Perspective Transformations – §10-8

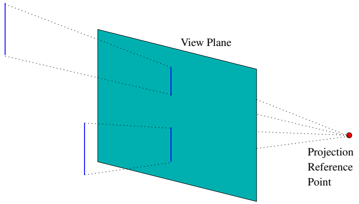
- For more realistic 3-D representations (e.g. simulating a camera) we must take account of fact that light rays from a scene follow a *converging* path to camera film plane
- Point where rays meet is called **projection reference point** or **convergence point**
- Similar length lines at different distances are projected to view plane differently
- Depending on positioning of view plane w.r.t. convergence point, image may be inverted (pinhole camera effect)

Perspective Transformations – §10-8

- For more realistic 3-D representations (e.g. simulating a camera) we must take account of fact that light rays from a scene follow a *converging* path to camera film plane
- Point where rays meet is called **projection reference point** or **convergence point**
- Similar length lines at different distances are projected to view plane differently
- Depending on positioning of view plane w.r.t. convergence point, image may be inverted (pinhole camera effect)

Perspective Transformations – §10-8

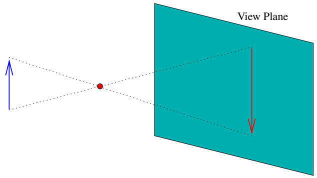
- For more realistic 3-D representations (e.g. simulating a camera) we must take account of fact that light rays from a scene follow a *converging* path to camera film plane
- Point where rays meet is called **projection reference point** or **convergence point**
- Similar length lines at different distances are projected to view plane differently



- Depending on positioning of view plane w.r.t. convergence point, image may be inverted (pinhole camera effect)

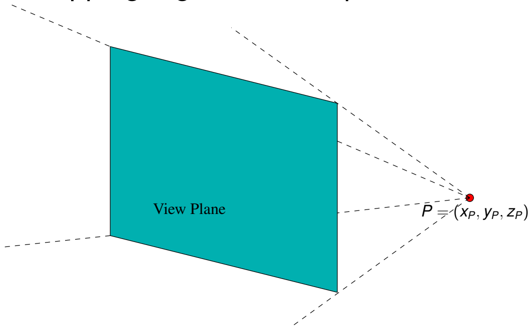
Perspective Transformations – §10-8

- For more realistic 3-D representations (e.g. simulating a camera) we must take account of fact that light rays from a scene follow a *converging* path to camera film plane
- Point where rays meet is called **projection reference point** or **convergence point**
- Similar length lines at different distances are projected to view plane differently
- Depending on positioning of view plane w.r.t. convergence point, image may be inverted (pinhole camera effect)



Perspective-Projection View Volume

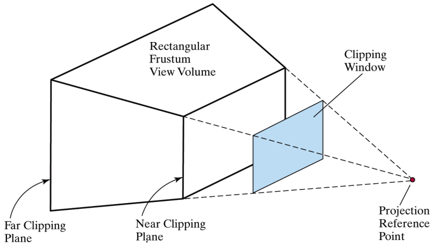
- We create a view volume by specifying the position of a rectangular clipping region on view plane



- In analogy to our “cone of vision” this is often called a “pyramid of vision”

Perspective-Projection View Volume (contd.)

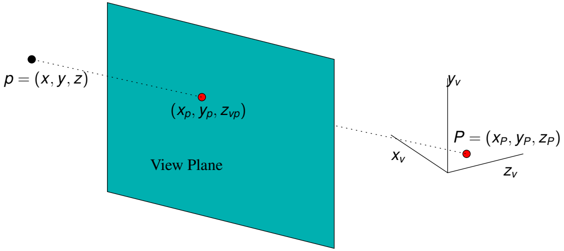
- When we add near and far clipping planes (perpendicular to z_v axis and parallel to view plane) we truncate this pyramid to get a **frustum**



- OpenGL requires that the near clipping plane be the view plane.

Transforming Perspective Coordinates

How does a point $p = (x, y, z)$ get projected on to view plane?



- Any point p' on the line from p to P can be written parametrically as $p' = p + u(P - p) = p(1 - u) + Pu$, where $0 \leq u \leq 1$
- On the view plane, $u = \frac{z_{vp} - z}{z_P - z}$, $1 - u = \frac{z_P - z_{vp}}{z_P - z}$, and this gives

$$x_p = x\left(\frac{z_P - z_{vp}}{z_P - z}\right) + x_P\left(\frac{z_{vp} - z}{z_P - z}\right)$$

Transforming Perspective Coordinates (contd.)

- On the view plane, $u = \frac{z_{vp} - z}{z_P - z}$ and this gives

$$x_p = x\left(\frac{z_P - z_{vp}}{z_P - z}\right) + x_P\left(\frac{z_{vp} - z}{z_P - z}\right)$$
$$y_p = y\left(\frac{z_P - z_{vp}}{z_P - z}\right) + y_P\left(\frac{z_{vp} - z}{z_P - z}\right)$$

- Some special cases to ease computational burden:
  - Limit PRP to be along z_v axis, that is, $P = (x_P, y_P, z_P) = (0, 0, z_P)$:

$$x_p = x\left(\frac{z_P - z_{vp}}{z_P - z}\right), \qquad y_p = y\left(\frac{z_P - z_{vp}}{z_P - z}\right)$$

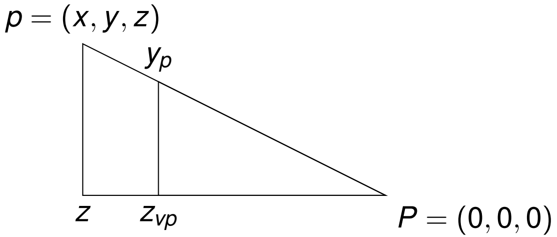
  - Limit PRP even further by forcing it at origin, $P = (0, 0, 0)$:

$$x_p = x\left(\frac{z_{vp}}{z}\right), \qquad y_p = y\left(\frac{z_{vp}}{z}\right)$$

OpenGL does this.

Transforming Perspective Coordinates (contd.)

With the OpenGL simplification of PRP being the origin we use the property of similar triangles that the ratios of the sides must be identical to argue:



The ratios of zs, $\frac{z_{vp}}{z}$, must match the ratios of ys, $\frac{y_p}{y}$, (similarly(!) for xs, too).
Refer back two slides for notation.

Transforming Perspective Coordinates (contd.)

- Other possible restrictions:
  - With restriction on PRP removed, place view plane at viewing origin, $z_{vp} = 0$
  - Use *uv* plane as view plane ($z_{vp} = 0$) and require PRP to be on z_v axis ($x_P = y_P = 0$)
- Note that if PRP is **on** view plane ($z_P = z_{vp}$) then everything projects to single point at (x_P, y_P) (Please verify this from perspective-transformation equations previously.)
- View plane is usually placed between PRP and scene, but doesn't have to be:
  - If PRP is between scene and view plane, inversion takes place (OpenGL doesn't allow this.)
  - If scene is between PRP and view plane "enlargement" onto view plane results