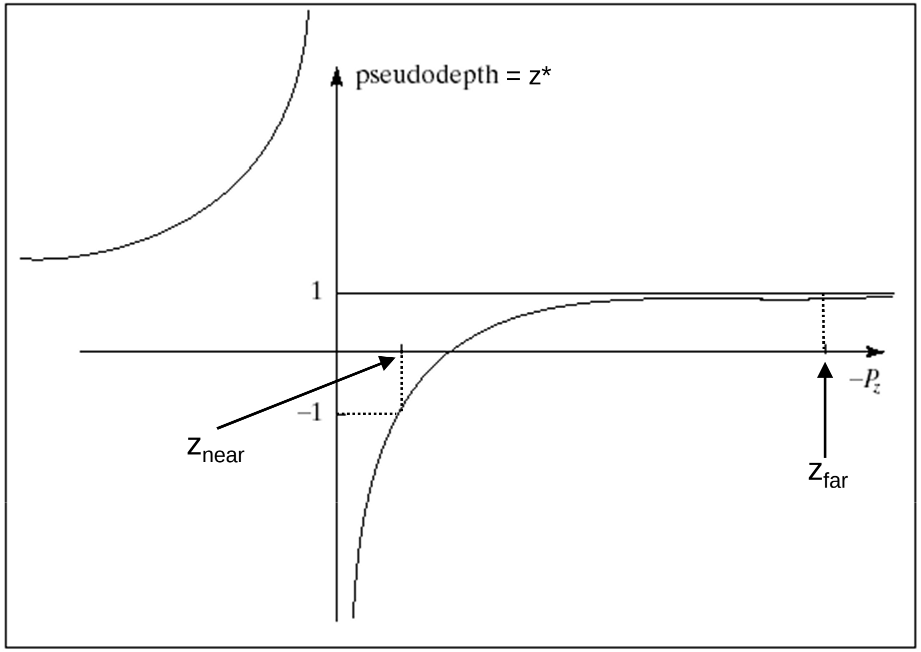
GRAPHICS GROUP

Computer Graphics: Clipping and Viewport Transformation

Part 2 – Lecture 3

Today's Outline

- Pseudodepth
- Clipping
- Viewport Transformations



PSEUDODEPTH

Perspective Transformation

- Requirements:
  1. x and y values must be scaled by same factor as derived in perspective projection equations
  2. z values must maintain depth ordering (monotonic increasing)
  3. z values must map: $-\mathbf{z}_{\mathbf{near}}$ → -1 and $-\mathbf{z}_{\mathbf{far}}$ → +1, view volume → NDC cube
- So we need a transformation that given a point *P* results in a transformed point *P'* such that *P'*$_x$ and *P'*$_y$ meet requirement 1 and $f(p_z)$ meets requirements 2 and 3:

$$P' = \left(\frac{-near}{p_z} p_x, \quad \frac{-near}{p_z} p_y, \quad f(p_z) \right)$$

- We have already found such a transformation:
  - Multiply *P* with $\mathbf{M}_{\mathbf{proj}}$
  - Convert result to ordinary coordinates (perspective division)

Perspective Transformation (cont'd)

- Perspective division:

  $P_{homog} = (x, y, z, w) \rightarrow P_{ord} = (x/w, y/w, z/w)$

- Thus, for these transformed points,

$$P^* = \mathbf{P}\,P = \begin{pmatrix} near\ x \\ near\ y \\ a\,z + b \\ -z \end{pmatrix} \qquad P^*_{near} = \mathbf{P}\,P_{znear} = \begin{pmatrix} near\ x \\ near\ y \\ -a\,near + b \\ near \end{pmatrix} \qquad P^*_{far} = \mathbf{P}\,P_{zfar} = \begin{pmatrix} near\ x \\ near\ y \\ -a\ far + b \\ far \end{pmatrix}$$

- Using $a = -\dfrac{far + near}{far - near}, \quad b = \dfrac{-2\ \ far\ near}{far - near}$

Ordinary form of the <u>x and y components</u>:

$z_{near}\, x / z = (-z_{near}/z)\, x$

$z_{near}\, y / z = (-z_{near}/z)\, y$

Ordinary form of the <u>z components</u>:

$(a\,z + b) / (-z)$

$(-a\, z_{near} + b) / z_{near} = -1.0$

$(-a\, z_{far} + b) / z_{far} = +1.0$

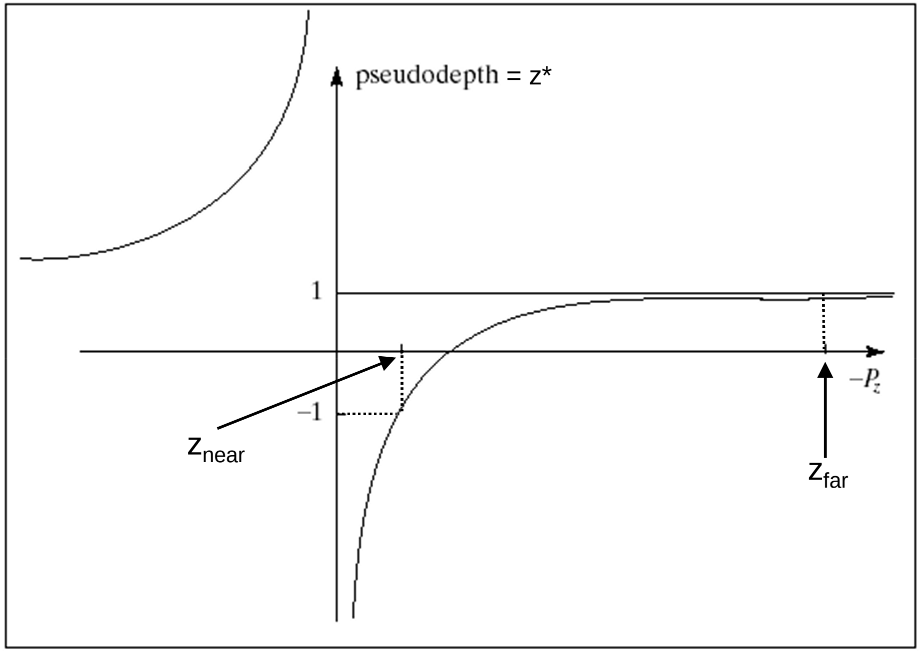
} *Check this out!*

Pseudodepth

- Transformed z* not linear function of z

$$z^* = \frac{az+b}{-z} = \frac{\left(-\frac{far+near}{far-near}\right)z + \frac{-2\,far*near}{far-near}}{-z}$$

$$z^* = \frac{(far+near)z + 2\,far*near}{(far-near)z}$$

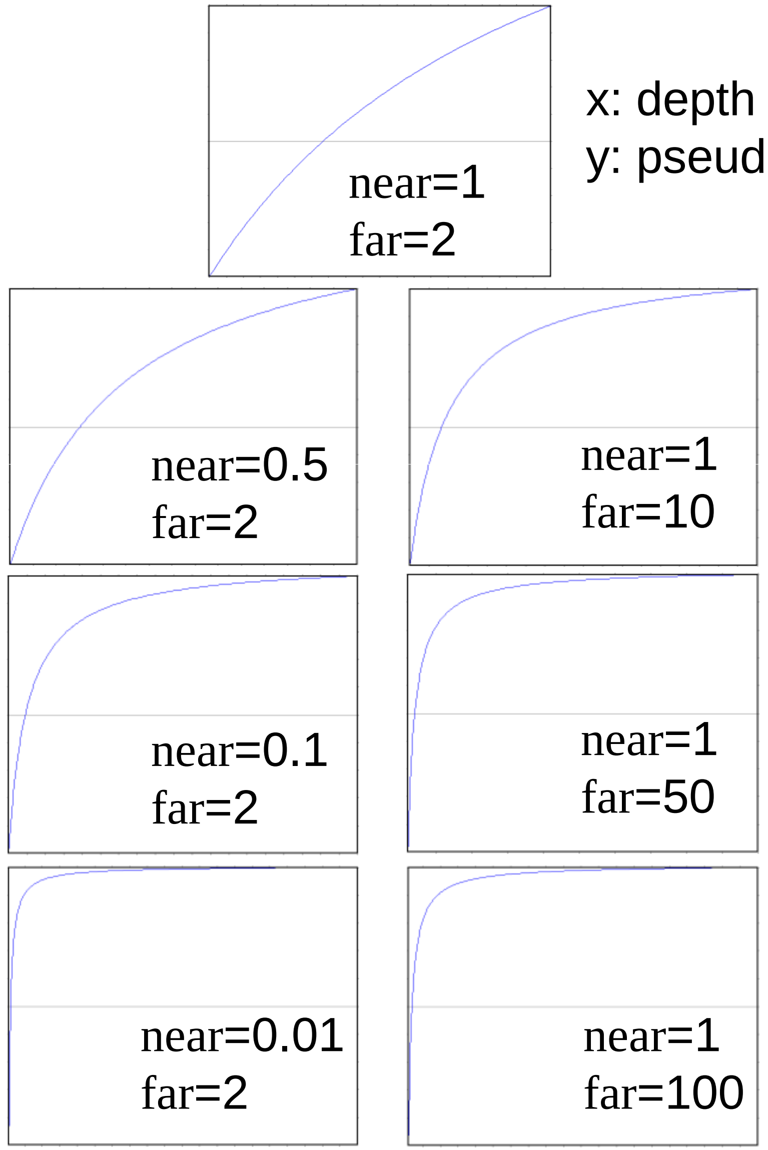


- This is OK (sort of) because z* meets our 2 requirements:
  1. monotonic increasing, and
  2. z* = -1 for $z = z_{near}$ = -near and z* = +1 for $z = z_{far}$ = -far
- But: can cause z-buffer precision problems! (z-buffer values are usually 32 bit integers)

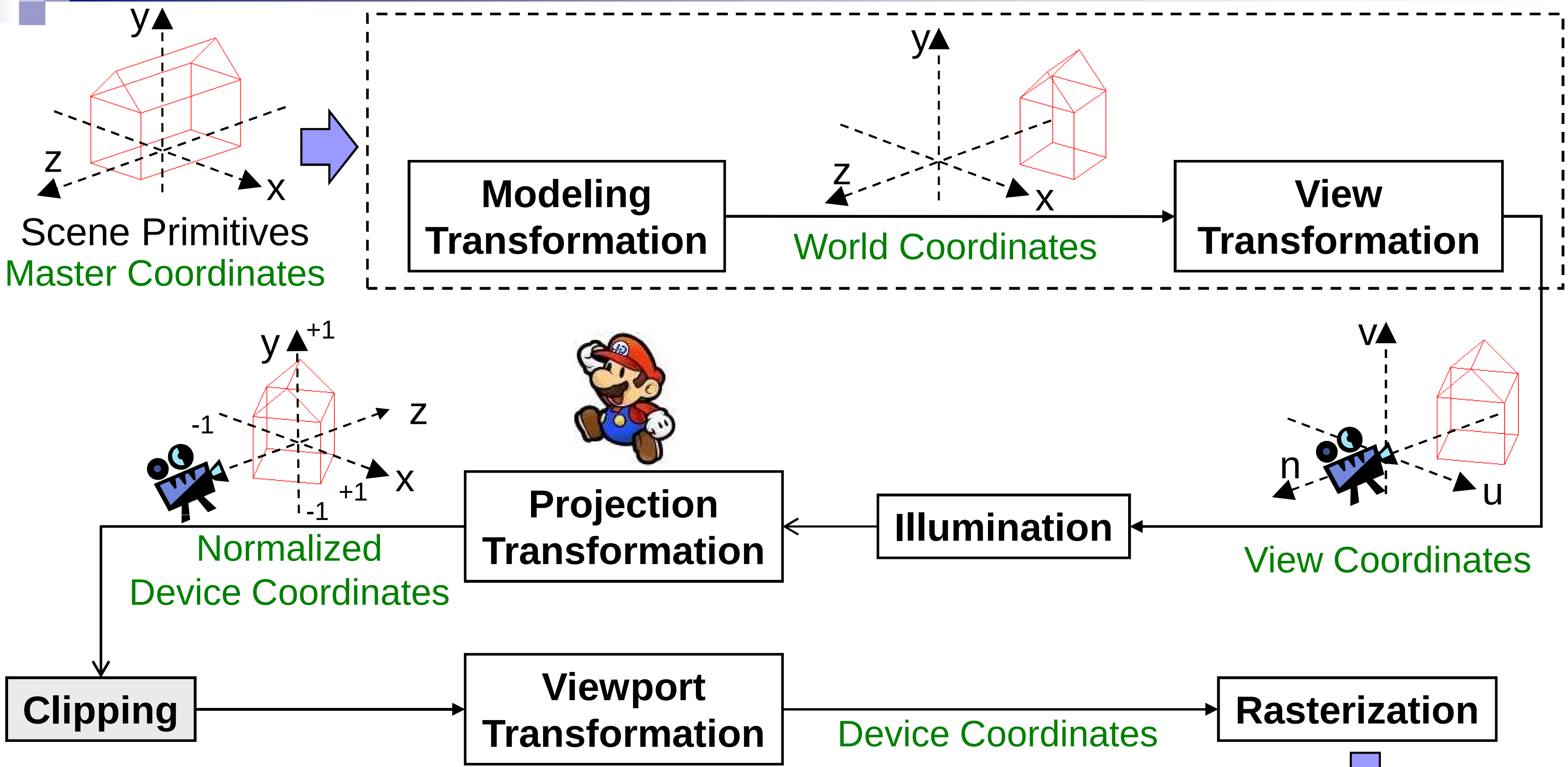
Problems of Pseudodepth

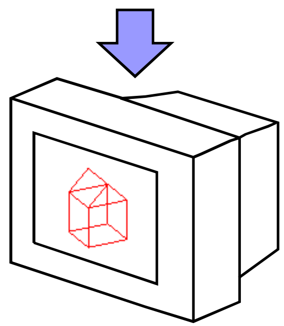
- Points closer to near plane have highest pseudodepth resolution
- Points closer to far plane have lowest pseudodepth resolution

- Never use near = 0
  $\rightarrow$ division by zero
- Avoid very small near and very large far
  $\rightarrow$ resolution too low for points
  that are further away



x: depth
y: pseudodepth

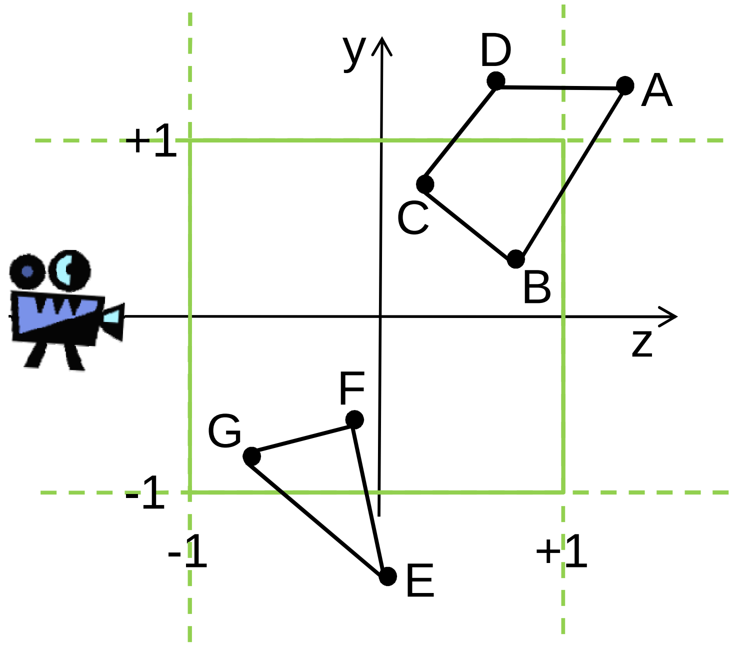

y
z
x
Scene Primitives
Master Coordinates
Modeling Transformation
y
z
x
World Coordinates
View Transformation
y
+1
-1
z
x
+1
-1
Normalized Device Coordinates
Projection Transformation
Illumination
v
n
u
View Coordinates
Clipping
Viewport Transformation
Device Coordinates
Rasterization



CLIPPING

Clipping

- Determine which lines are in the canonical view volume (using NDC)
- Outside of the view volume is given by:
  $p_x < -1$, $p_x > +1$, $p_y < -1$, $p_y > +1$,
  $p_z < -1$, $p_z > +1$
  ($\rightarrow$ **clip planes**)
- Each line is either…
  1. completely inside
     $\rightarrow$ **trivial accept**
  2. completely outside
     $\rightarrow$ **trivial reject**
  3. Partially in the view volume
     $\rightarrow$ need to find out which part is inside



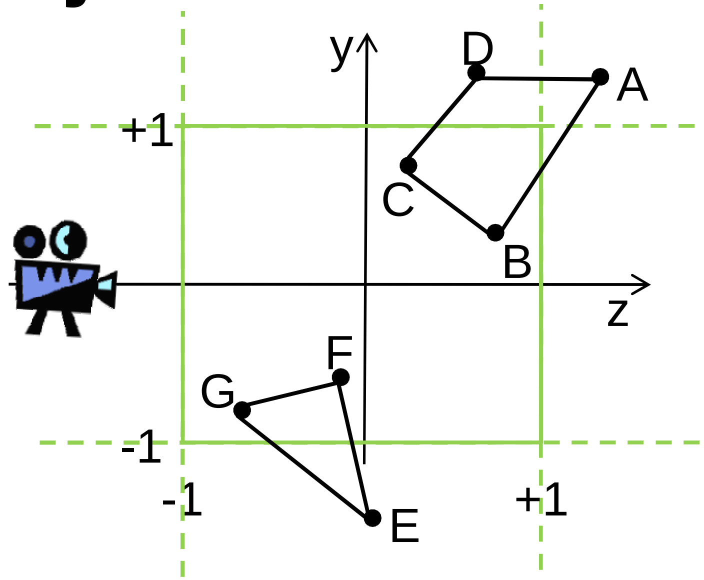
Trivial accept for:
CB and GF

Trivial reject for:
DA

Partially visible:
AB, CD, EF and EG

Trivial Accept and Reject Tests

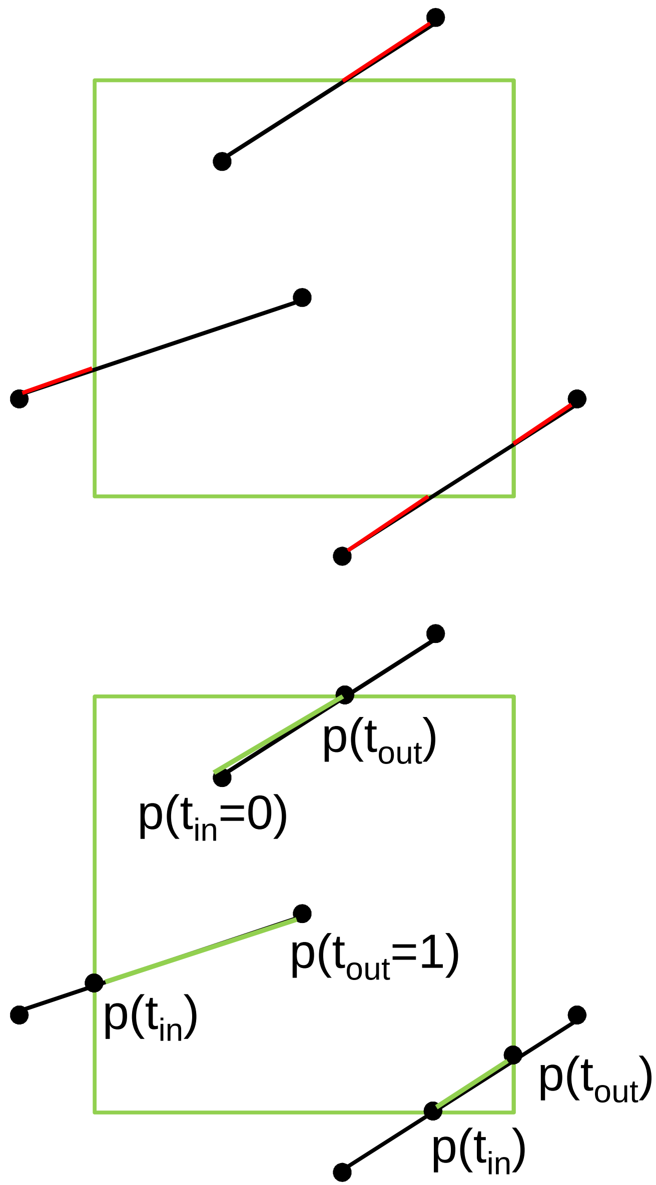
- For each point, check if it is outside of left (L), right (R), bottom (B), top (T), near (N) and far (F) clip plane
- Create table with **outcodes**: 1 if point is outside, 0 if inside
  - **Trivial reject** of a line PQ:
    = P and Q outside of the same clip plane
    = outcodes for same plane both 1
    = **`(outcode P & outcode Q)!=0`**
  - **Trivial accept** of a line PQ:
    = both endpoints inside of all clip planes
    = all outcodes 0
    = **`(outcode C | outcode D)==0`**



	L	R	B	T	N	F
A	0	0	0	**1**	0	**1**
B	0	0	0	0	0	0
C	0	0	0	0	0	0
D	0	0	0	**1**	0	0
E	0	0	**1**	0	0	0
F	0	0	0	0	0	0
G	0	0	0	0	0	0

Nontrivial Clipping

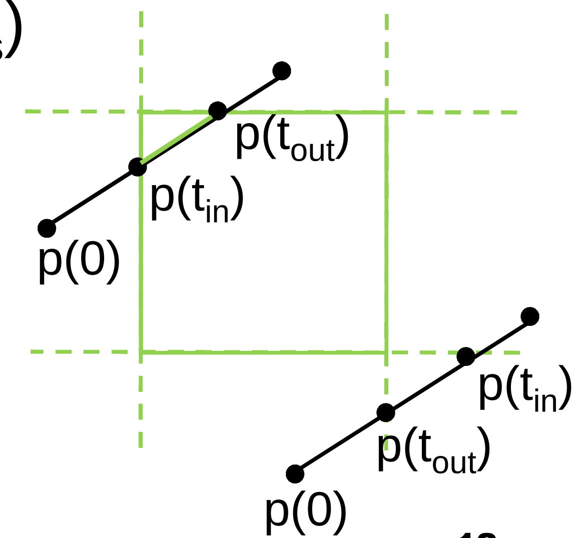
- **Idea**: find intersection point of line with each clipping plane
- Each line can only enter and leave the view volume once
- For each intersection X of line PQ with a clipping plane:
  - If P outside, then clip off PX
  - If P inside, the clip off XQ
- We use parametric line equation $p(t) = p_0 + t(p_1 - p_0)$ with $0 <= t <= 1$
- Clipping by finding t_{in} and t_{out} parameter values for line segment in view volume



Liang-Barsky Clipping Algorithm

Clip a line from point p_0 to p_1, represented as $p(t) = p_0 + t(p_1 - p_0)$

1. Perform trivial reject and accept tests, stop if trivial
2. Initialize $t_{in}=0$ and $t_{out}=1$
3. For each halfspace {x>– 1, x<+1, y>–1, y<+1, z>–1, z<+1} do
   1. Compute t_{cross} where (extended) line crosses halfspace
   2. If entering half-space then $t_{in} = \max(t_{in}, t_{cross})$ else $t_{out} = \min(t_{out}, t_{cross})$
   3. Stop if $t_{in} > t_{out}$
4. if $t_{in} > t_{out}$ then line is outside viewing volume else $p_0 = p(t_{in})$ and $p_1 = p(t_{out})$



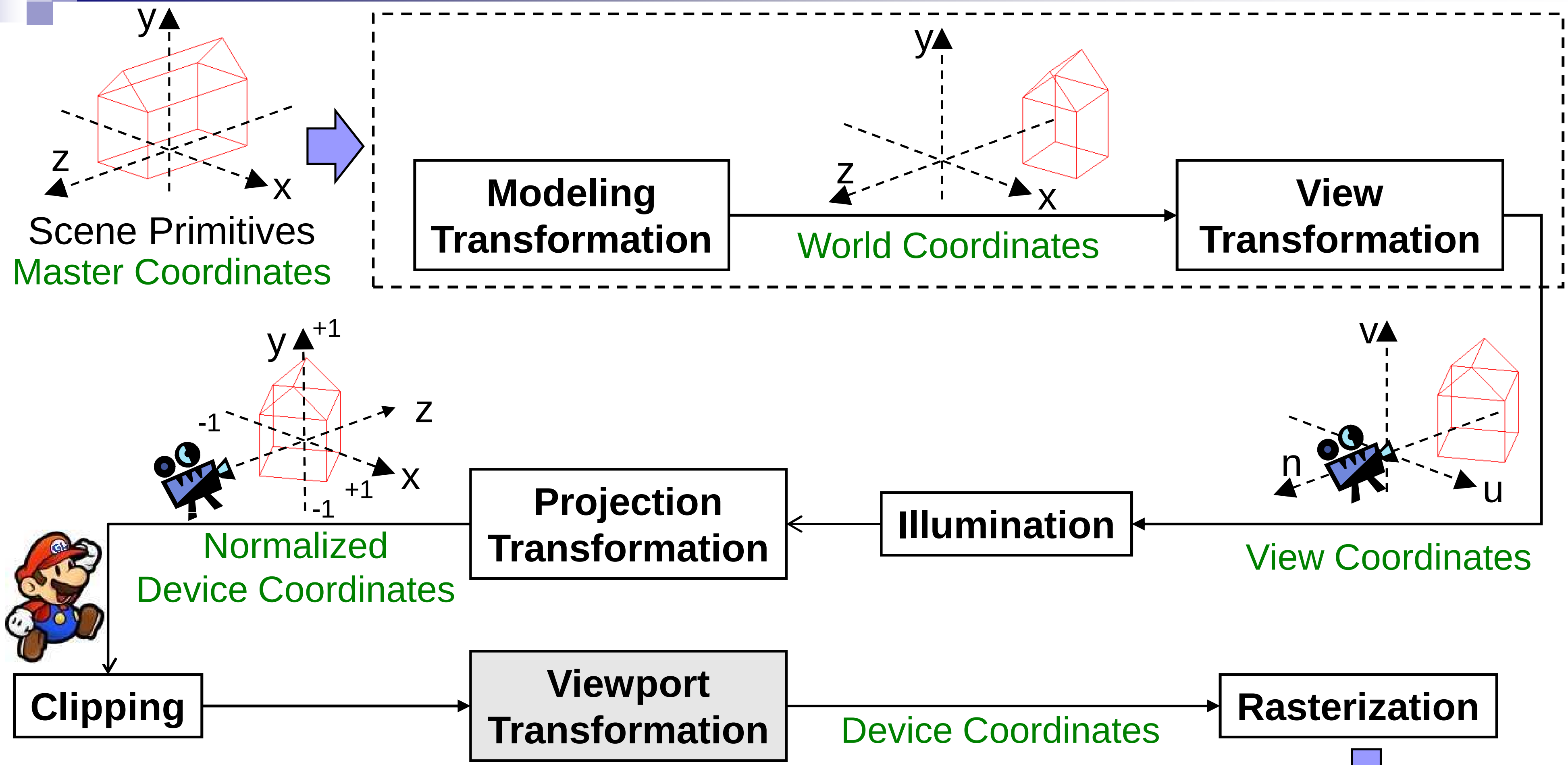
Clipping with Homogeneous Coordinates

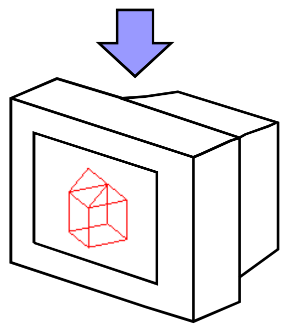
- OpenGL actually performs clipping before perspective division, i.e. using homogeneous coordinates
- One reason: perspective division only necessary for vertices that are in view volume
- Differences in clipping algorithm:
  - Point p is outside of view volume if

    $$p_x / p_w < -1 \iff p_x < -p_w \iff p_x + p_w < 0$$

    Other planes:

    $p_x - p_w > 1$, $p_y + p_w < 0$, $p_y - p_w > 0$, $p_z + p_w < 0$, $p_z - p_w > 0$
  - Compute $p_x(t)$, $p_y(t)$, $p_z(t)$, **<u>and</u> $p_w(t)$**

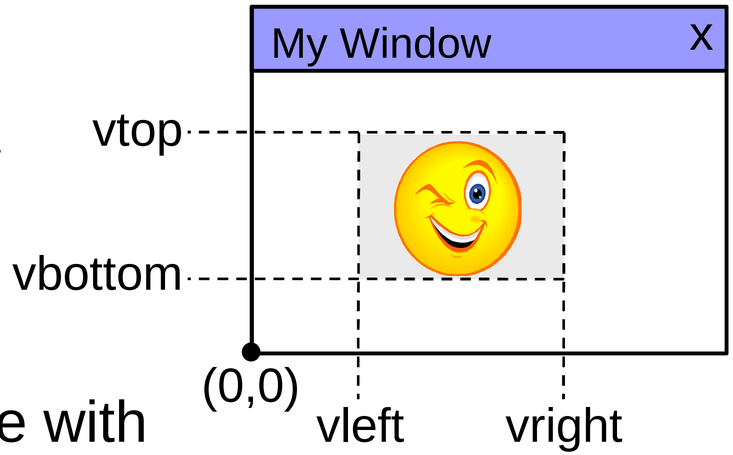




VIEWPORT TRANSFORMATIONS

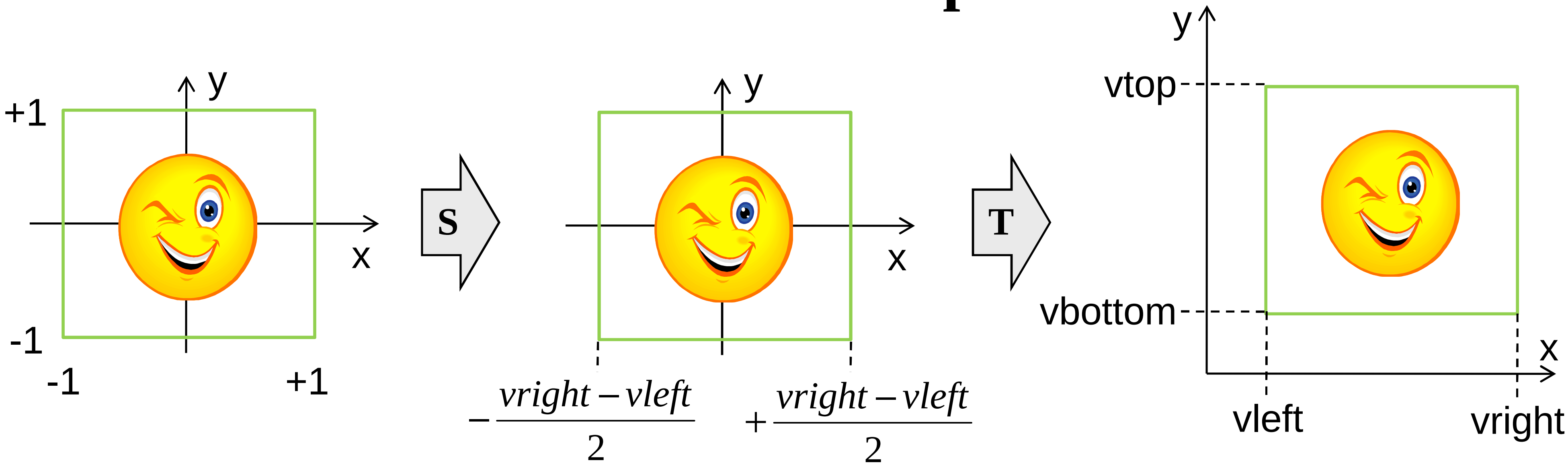
Viewport Transformation

- Mapping from Normalized Device Coordinates (NDC) to device coordinates (DC) aka viewport coordinates
- For NDC: $x,y,z \in (-1, +1)$
- For DC: $x \in (vleft, vright)$, $y \in (vbottom, vtop)$, $z \in (0, maxz)$
  - x and y are 2D window coordinates
  - vleft, vright, vbottom, vtop are the boundaries of the viewport in the window
  - maxz depends on type used for depth buffer values (e.g. uint32)
  - In OpenGL: set viewport position and size with `glViewport(x, y, width, height);`



- NDCs are multiplied with **viewport matrix** $\mathbf{M}_{\mathbf{viewport}}$ which maps NDC boundaries onto viewport boundaries

Viewport Matrix $\mathbf{M}_{\mathbf{viewport}}$



$$\mathbf{M}_{\text{viewport}} = \mathbf{T}\,\mathbf{S} = \begin{pmatrix} 1 & 0 & 0 & \frac{vright+vleft}{2} \\ 0 & 1 & 0 & \frac{vtop+vbottom}{2} \\ 0 & 0 & 1 & \frac{maxz}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{vright-vleft}{2} & 0 & 0 & 0 \\ 0 & \frac{vtop-vbottom}{2} & 0 & 0 \\ 0 & 0 & \frac{maxz}{2} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Multiple Viewports

- **Problem:** How to write a GL program that displays multiple views of a scene, each one in a different viewport?
- **Solution: Multiple viewports**

  Multiple views of a scene, e.g., architectural drawing front, side, and top views

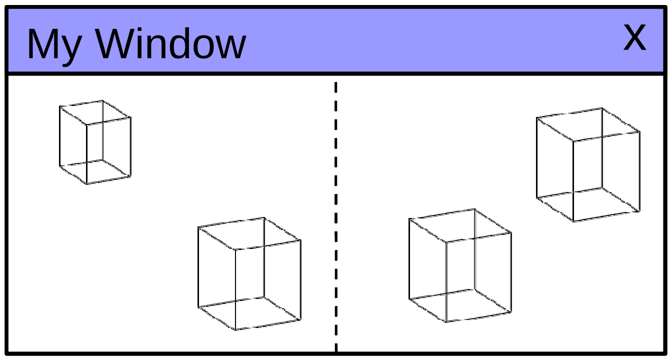
  Loop: repeat for each viewport

  1. Set this viewport:
     `glViewport( x, y, width, height );`
  2. Set view projection for this viewport (might be the same for all viewports, if so do this before loop):
     `glOrtho(left, right, bottom, top, zNear, zFar );`
     or other such as `gluPerspective( … );`
  3. Set camera view position and orientation for this viewport
     `gluLookAt(left, right, bottom, top, zNear, zFar );`
     or other such as `glTranslatef( … ); glRotatef( … );`
  4. Draw scene

Multiple Viewports Code Example

```
// left: perspective
glViewport(0, 0, 100, 100);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(yfov, aspect,
   zNear, zFar);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
// do view transformations…
drawScene();
```

```
// right: orthographic
glViewport(100, 0, 100, 100);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(left, right, bottom,
   top, near, far);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
// do view transformations…
drawScene();
```



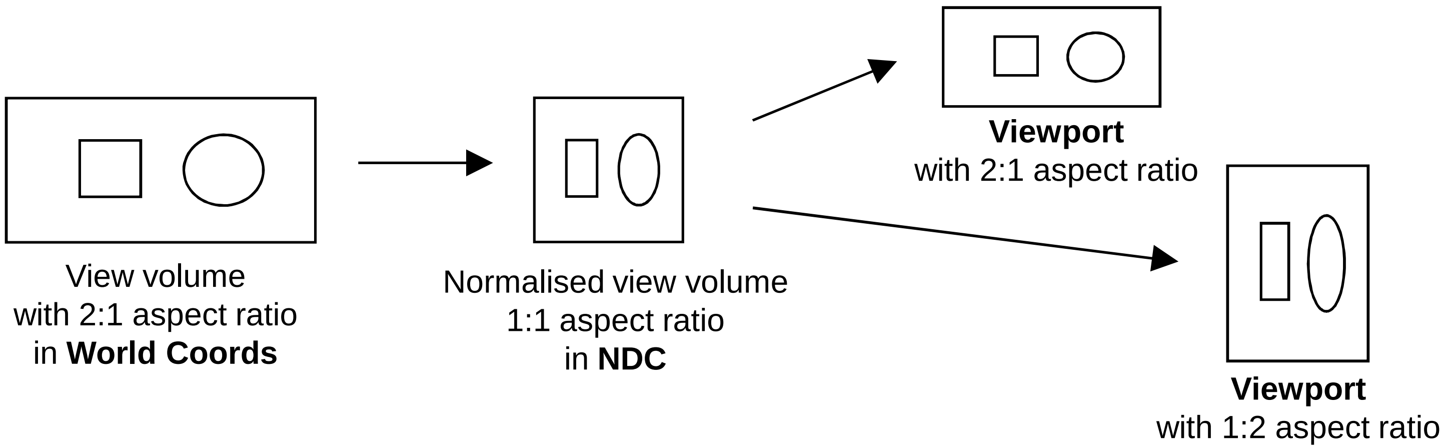
Aspect Ratio of View Volume and Viewport

- Final pipeline transformation step is viewport transformation

```
glViewport(GLint x, GLint y,
           GLsizei width, GLsizei height);
```

  Default viewport is entire drawing window, (0, 0, winWidth, winHeight).

- **Aspect ratio** of view volume and viewport should be same



- **Problem:** How to write a GLUT program that automatically resets the view volume aspect ratio when window (viewport) is resized?

Aspect Ratio: reshape callback function

Solution: in GLUT, use **reshape callback** to adjust viewport and view volume aspect ratio after a **window resize event**

- Register reshape callback function (in main at prog. init.)

```
void reshape(GLsizei width, GLsizei height); // prototype
glutReshapeFunc( reshape );      // callback registration
```

- Define reshape callback function (in main prog. module)

```
// left, right, bottom, top = class member or global variables
void reshape( GLsizei width, GLsizei height ) {
   glViewport(0, 0, width, height );    // set viewport size
   GLfloat aspect = (GLfloat)width /(GLfloat)height;  //NOT int!
   GLdouble center = (left + right) / 2.0;
   GLdouble newHalfWidth = aspect * (top - bottom) / 2.0;
   left = center - newHalfWidth;  right = center + newHalfWidth;
   glMatrixMode(GL_PROJECTION);  // reset proj matrix
   glLoadIdentity();
   glOrtho(left, right, bottom, top, near, far);
   drawSceneObjects();           // redraw all objects
}
```

SUMMARY

Summary

- Pseudodepth
  - Used to normalize z with matrix
  - For small near and large far resolution problems
- Clipping removes lines outside of view volume
  - Trivial accept and reject tests using outcodes
  - Check t_{in} and t_{out} values of parametric line equation
- Viewport Transformation: maps NDCs to DCs using $\mathbf{M}_{\mathbf{viewport}}$

References:

- Pseudeodepth: Hill, Chapter 7.4.3, pp. 349-351
- Clipping: Hill, Chapter 7.4.3, pp. 356-361
- Viewport Transformation: Hill, Chapter 7.4.3, p. 361

Quiz

1. Why isn't it a good idea to use a very small number for near or a very large number for far?
2. How is an outcode table constructed? How is it used for trivial reject/accept?
3. How do we find t_{in} and t_{out} during clipping? How does it help us to clip lines?