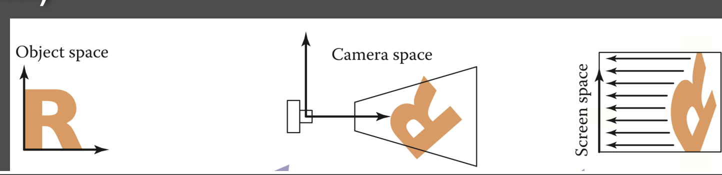
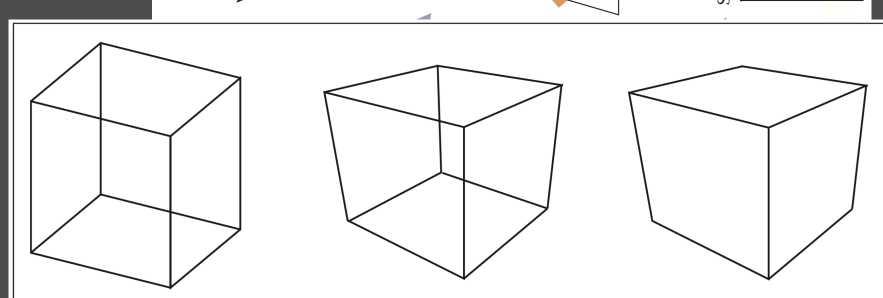
Hidden Surface Removal

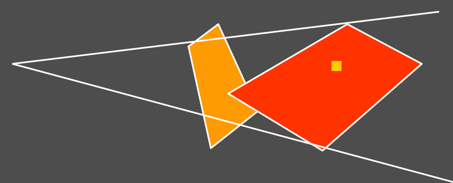
- In this dack of slides, we will assume that `depths' is along the z-axis. That is, camera is at (0,0,0), LookAt at (0,0,-1) and $\overrightarrow{w} = (0,0,1)$
- Furthermore, we assume that the projection is orthographic. $(x, y, z) \to (x, y, 0)$.
- That is, not a perspective projection
- Our hardware likes this assumptions.
- But if the camera's parameters don't agree - will discuss transformations (later)



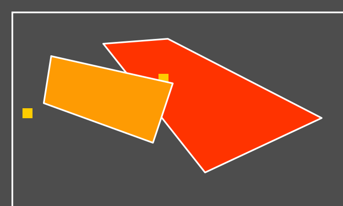


Hidden Surface Removal for Polygonal Scenes

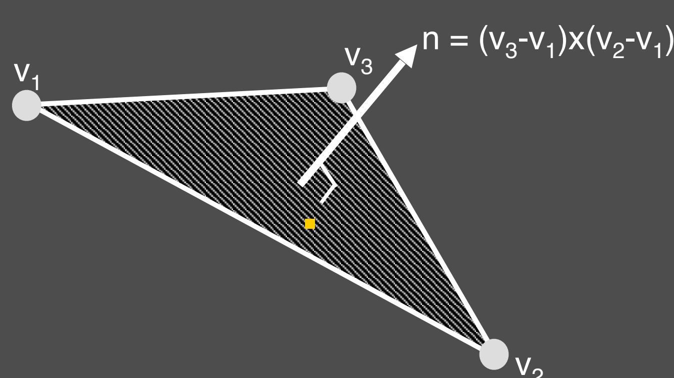
- Input: Set of polygons in three-dimensional space + a viewpoint



- **Output:** A two-dimensional image of projected polygons, containing only visible portions

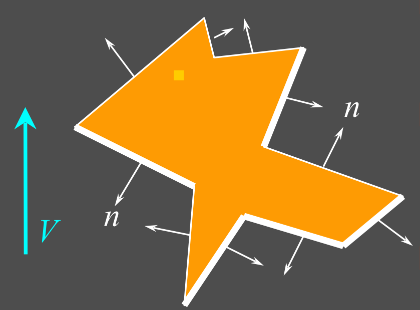


The Normal Vector

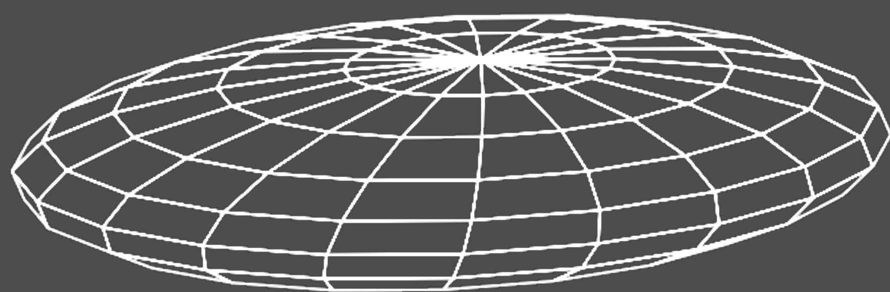


$n(1,2,3) = n(2,3,1) = -n(2,1,3)$

Back Face Culling (object space)



- In closed polyhedron you don't see object "back" faces
- Assumption
  - Normals of faces point *out* from the object
- Object space algorithm

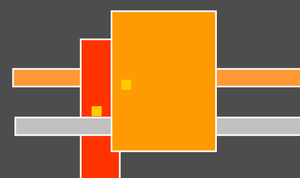


Back Face Culling

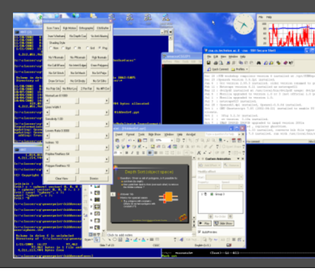
- Determine back & front faces using sign of inner product $< n, V >$
- In a convex object :
  - Invisible back faces
  - All front faces entirely visible $\Rightarrow$ solves hidden surfaces problem
- In non-convex object:
  - Invisible back faces
  - Front faces can be visible, invisible, or partially visible

Depth Sort (object space)

- **Question:** Given a set of polygons, is it possible to:
  - sort them by depth. The order is not necessarily unique.
  - Then paint them back to front (over each other) to remove the hidden surfaces ?(each polygon fully rendered)
  - This is called the **painter algorithm.**



- **Answer:** Usually not
- Works for special cases
  - E.g. polygons with constant z (where do we have polygons with constant z!?)

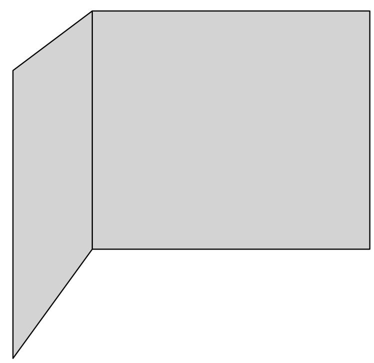


Painter’s Algorithm

- Draw one primitive at a time.

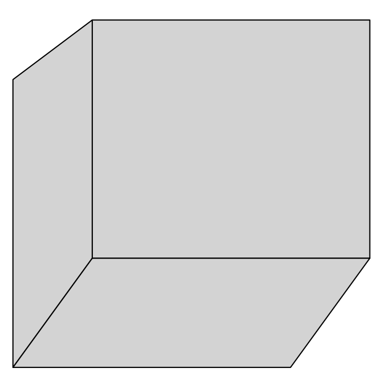
Painter’s Algorithm

- Draw one primitive at a time.



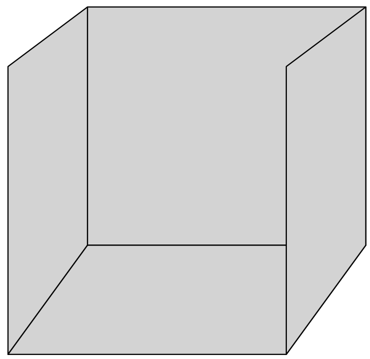
Painter’s Algorithm

- Draw one primitive at a time.



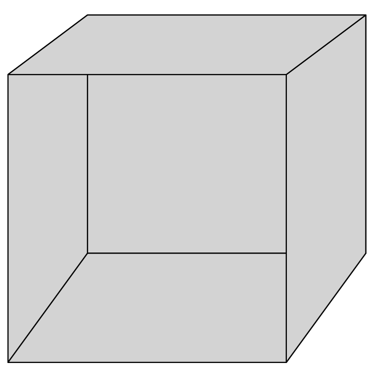
Painter's Algorithm

- Draw one primitive at a time.



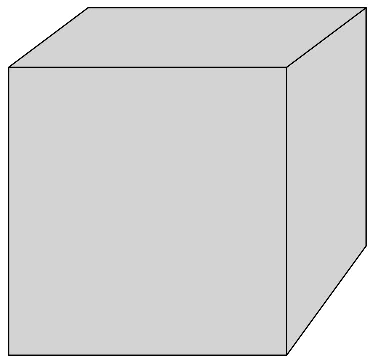
Painter's Algorithm

- Draw one primitive at a time.



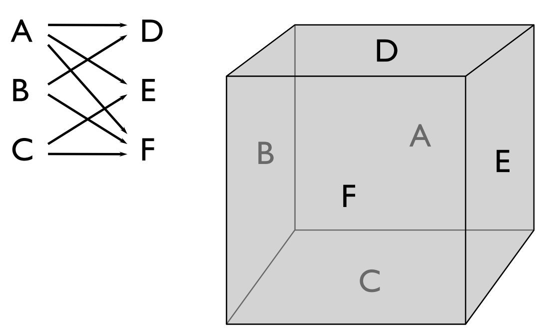
Painter's Algorithm

- Draw one primitive at a time.



Painter's algorithm

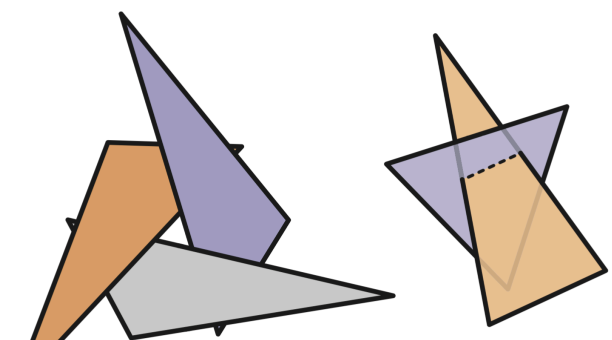
- **Amounts to a topological sort of the graph of occlusions**
  - that is, an edge from A to B means A sometimes occludes B
  - any sort is valid
    - ABCDEF
    - BADCFE
  - if there are cycles there is no sort



From

Painter's algorithm

- **Amounts to a topological sort of the graph of occlusions**
  – that is, an edge from A to B means A sometimes occludes B
  – any sort is valid
    - ABCDEF
    - BADCFE
  – if there are cycles there is no sort

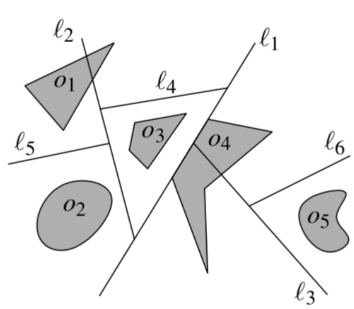


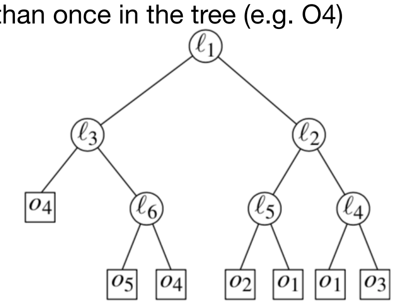
Cycles cannot be sorted

BSP (Binary Space Partition) Trees

- Partition a set of objects using a set of arbitrary half-spaces (for clarity, we only show segments, and not the full line separating objects).
- Each internal node contains a halfplane (in 3D) or a line (in 2D).
- For each half-space, divide the objects into two groups, those above and those below the half-space boundary
- Store the resulting divisions in a binary tree
  - Arbitrarily oriented split planes
  - Objects get fragmented into multiple pieces

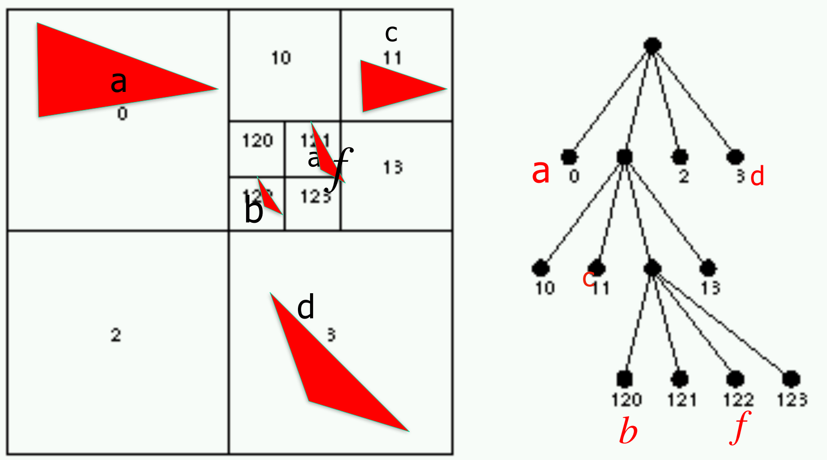
- Example BSP of 5 objects in the plane
- The same object might Possibly objects might be ``split". They are not physically split, but an object might be stored more than once in the tree (e.g. O4)





- BSP has many other applications We will revisit BSP when talking about accelerating

QuadTrees/OctTree for shape - also a BSP



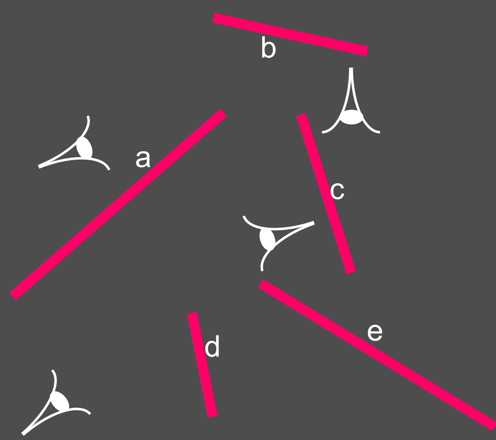
Input: Set S of triangles $S=\{t_{1\ldots}t_n\}$

Splitting policy: Split quadrant if it intersects more than 1 triangle of S.

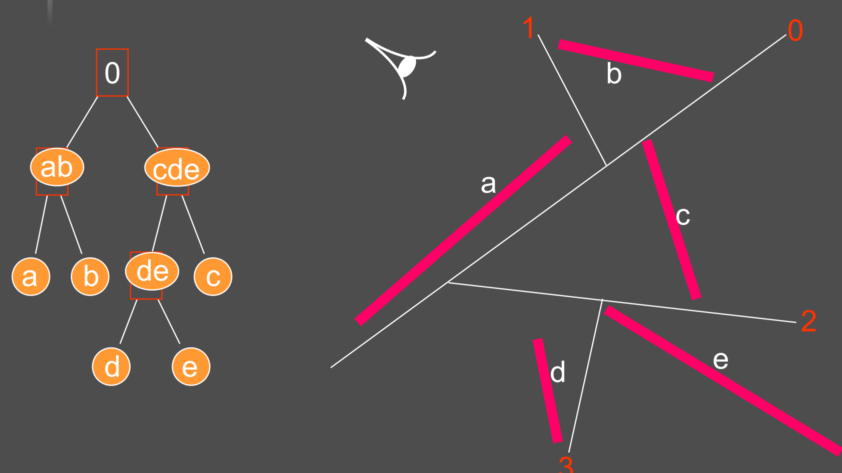
Depth Sort (object space)

- Will fail for:
  - Intersecting polygons
  - Mutually occluding polygons

The Visibility Problem



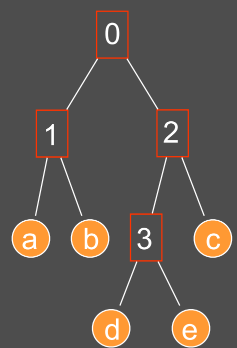
Binary Planar Partitions

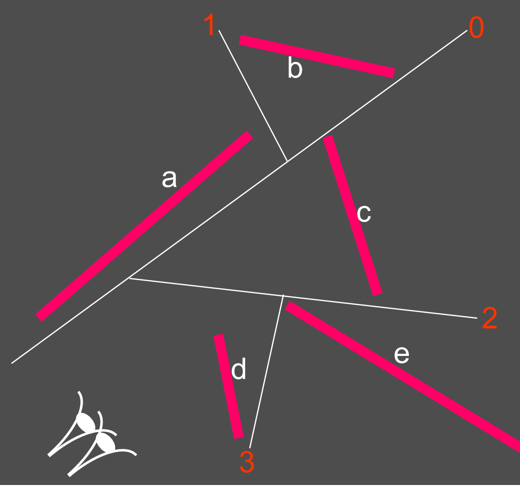


Draw everything below the line 0 (farther from the eye) before drawing the ones closer to the eye. Continue inside each subtree.
(the drawing actually takes place when visiting leaves of tree)

This structure, as any other hierarchical decomposition, is useful for range searching, point location etc.

Painter's Algorithm





Given the location of the camera (Eye), and a root of a BSP, the Eye is in one of the two half spaces that the root separates.

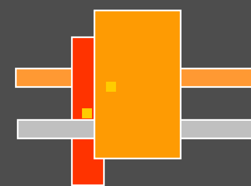
1. Render the objects in the subtree that is more remote from Eye (recursively)
2. Render objects in the root
3. Render objects in the nearer subtree

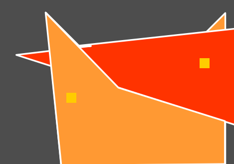
Creating a BSP might force splitting objects

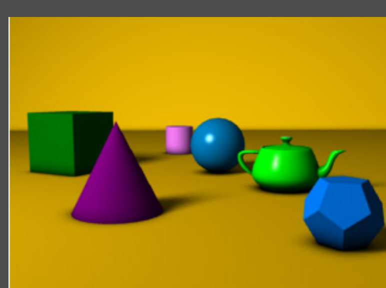
- **Can you think of 3 non-intersecting segments, where you cannot find a BSP without splitting?**

Z-Buffer Algorithm (image space)

- **Basic Idea: resolve the visibility at the pixel level, using depth sort.**
- For each image pixel - store both the color and the current z depth
- Instead of always painting the pixels while scan-converting a polygon, do so only if polygon's depth is less than current z depth at that pixel

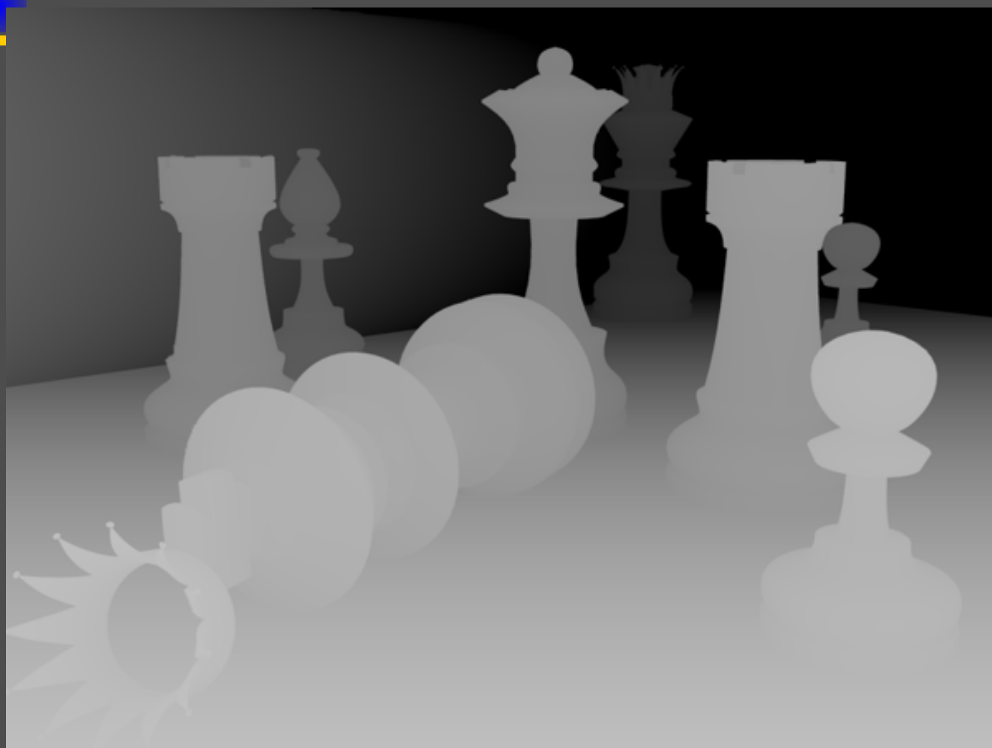






A simple three dimensional scene

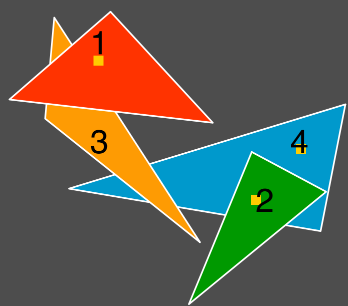
Z-buffer representation



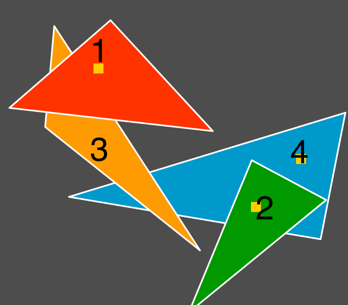
Z-Buffer

```
ZBuffer(Scene)
For every pixel (x,y) do PutZ(x,y,MaxZ);
For each polygon P in Scene do
   Q := Project(P);
   For each pixel (x,y) in Q do
      z := Depth(Q,x,y);
      if (z < GetZ(x,y)) then
         PutZ(x,y,z);
         PutColor(x,y,Col(P,x,y));
      end;
   end;
end;
```

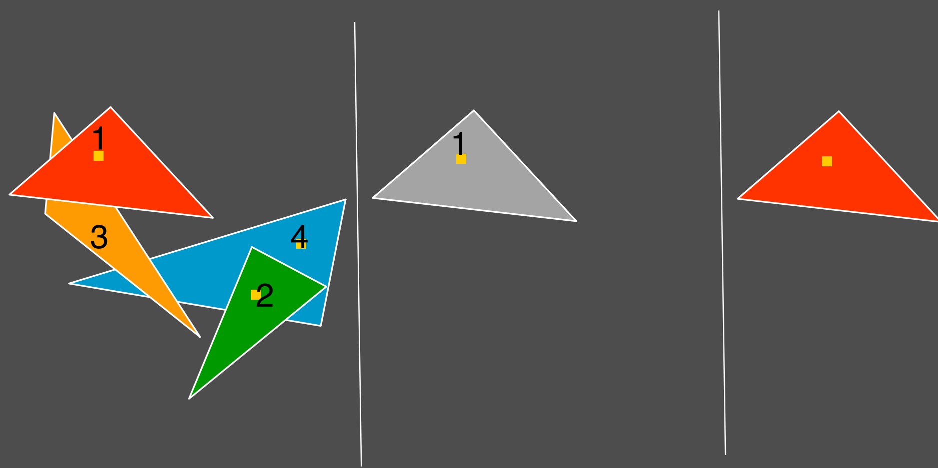
Questions: How can one compute **Project**(P) and **Depth**(Q,x,y)?



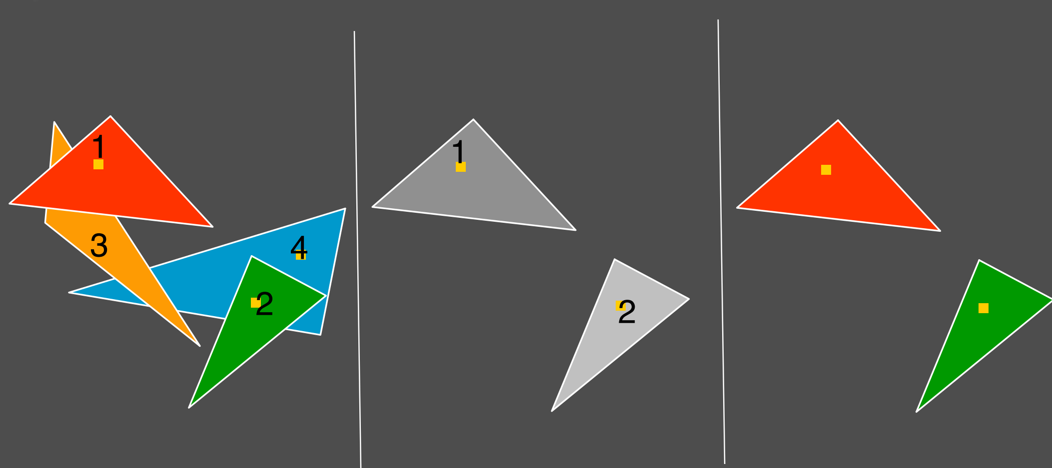
- Final image buffer
- Depth buffer
- Color buffer



- Final image buffer
- Depth buffer
- Color buffer

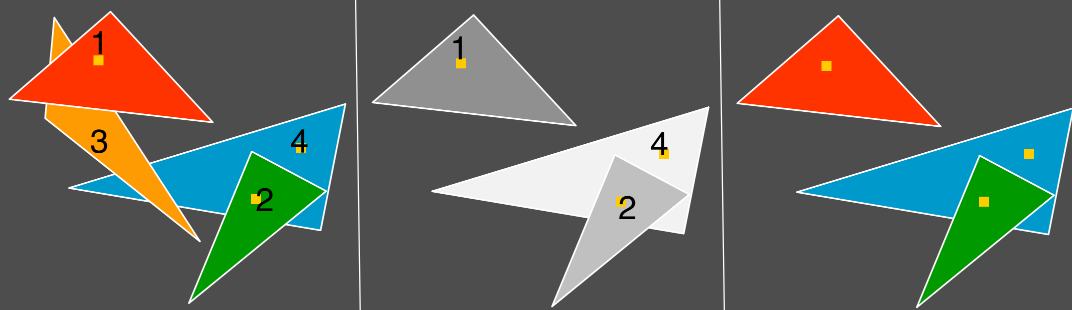


- Final image buffer
- Depth buffer
- Color buffer

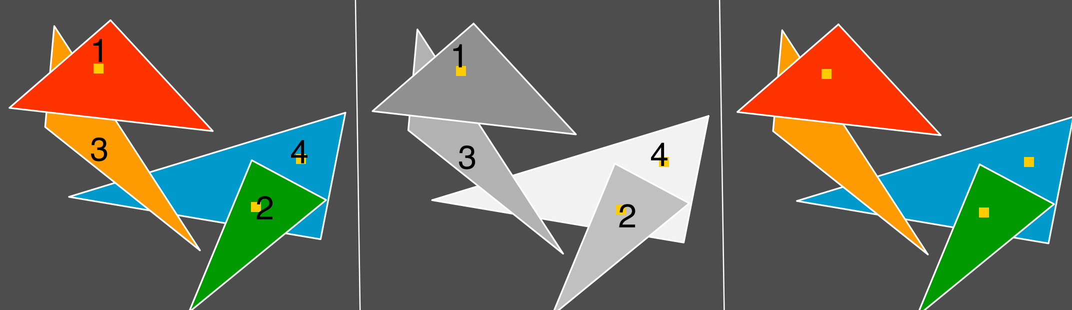


- Final image buffer
- Depth buffer
- Color buffer

Grey-level indicates z distance. Darker = closer to camera



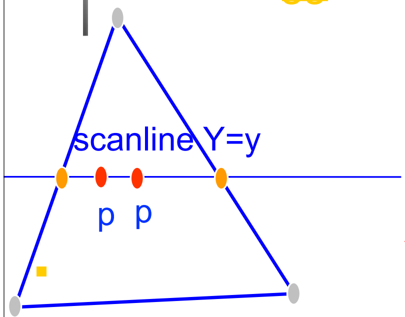
- Final image buffer
- Depth buffer
- Color buffer

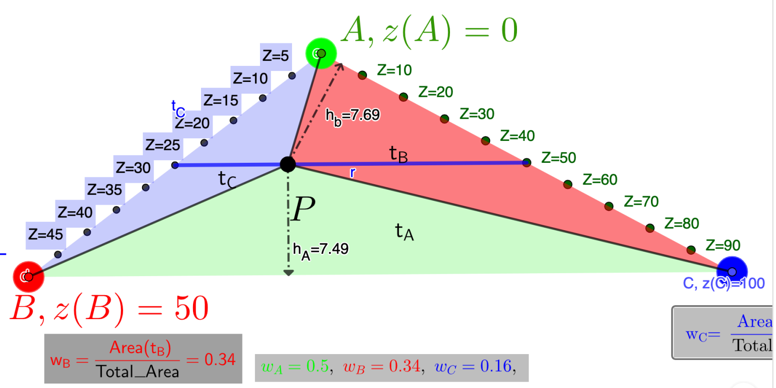


- Final image buffer
- Depth buffer
- Color buffer

Computing Depth(Q,x,y) efficiently

gg





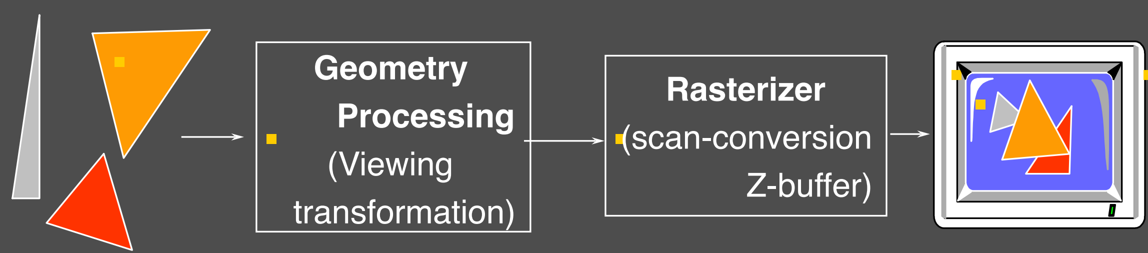
- For a pixel *p*, its depth is the (weighted) linear interpolation (using barycentric coordinates) of the depth of vertices
- Depending on the level of parallelism we have: Three cases: num_processors≈ num_pixels, ≤ num_rows, or is a constant.
  - If have a processor per pixel - they all compute their own barycentric coordinates.
  - If we have only one processor - do compute the interpolated depth along the edges AC, AB
  - To find the value at p, create a segment parallel to BC, via p. Linearly interpolate the values at its endpoint.
  - To compute the depth at p', only one addition operation

Z-Buffer Algorithm

- Image space algorithm
- Data structure: Array of depth values
- Common in hardware due to simplicity
- ***Scene may be updated on the fly, adding new polygons***

The Graphics Pipeline

- Hardware implementation of screen Z-buffer:
  - Polygons sent through pipeline one at a time
  - Display updated to reflect each new polygon



Transparency Z-Buffer

How can we emulate transparent objects?

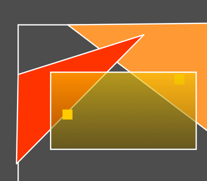
Transparency Buffer

- Extension to the basic Z-buffer algorithm
- Save *all* pixel values
- At the end – have list of polygons & depths (order) for each pixel
- Simulate transparency
  by weighting the
  different list elements,
  in order

The A - buffer

- For transparent surfaces and filter based anti-aliasing:

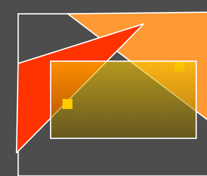
- Algorithm (1): filling buffer
  - at each pixel, maintain a pointer to a list of polygons sorted by depth. (only the few nearest ones)
  - when filling a pixel:
    - if polygon is opaque and covers pixel, insert into list, removing all polygons farther away
    - if polygon is opaque and only partially covers pixel, insert into list, but don't remove farther polygons



pixel

The A - buffer

- Algorithm (2): rendering pixels
  - at each pixel, traverse buffer using brightness values in polygons to fill.
  - values are used for either for calculations involving transparency or for filtering for aliasing



pixel

Viewing

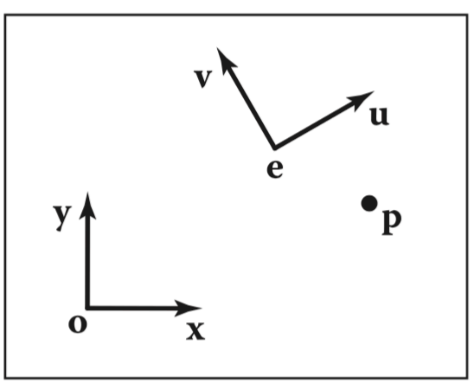
Coordinate Transformations

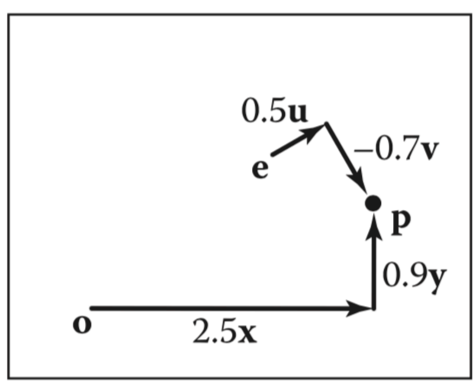
Recall: Coordinate Systems

- Points in space can be represented using an origin position and a set of orthogonal basis vectors:

$$\mathbf{p} = (x_p, y_p) \equiv \mathbf{0} + x_p\mathbf{x} + y_p\mathbf{y} \qquad \mathbf{p} = (u_p, v_p) \equiv \mathbf{e} + u_p\mathbf{u} + v_p\mathbf{v}$$

- Any point can be described in either coordinate system





Recall: Matrices for Converting Coordinate Systems

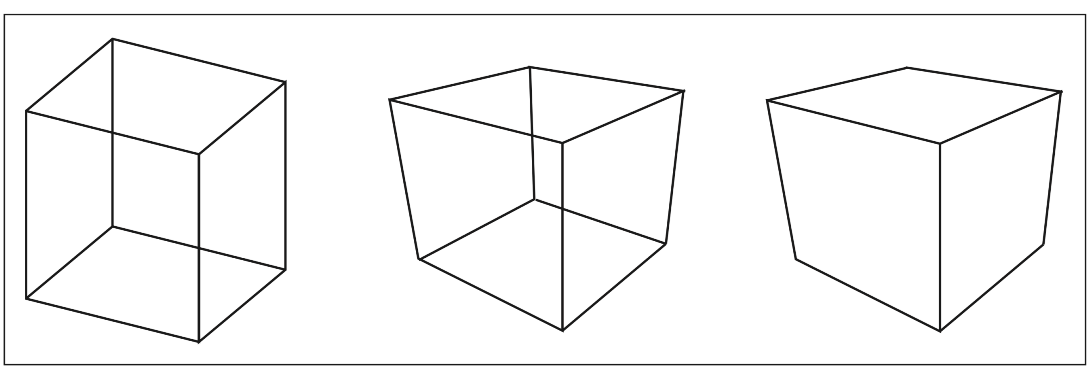
- Using homogenous coordinates and affine transformations, we can convert between coordinate systems:

$$\begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_e \\ 0 & 1 & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & 0 \\ y_u & y_v & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & x_v & x_e \\ y_u & y_v & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix}$$

- More generally, any arbitrary coordinate system transform:
- $$\mathbf{P}_{uv} = \begin{bmatrix} \mathbf{x}_{uv} & \mathbf{y}_{uv} & \mathbf{o}_{uv} \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p}_{xy}$$

Drawing by Transformation

- For now, we will consider drawing wireframe objects (collections of 3D line segments)

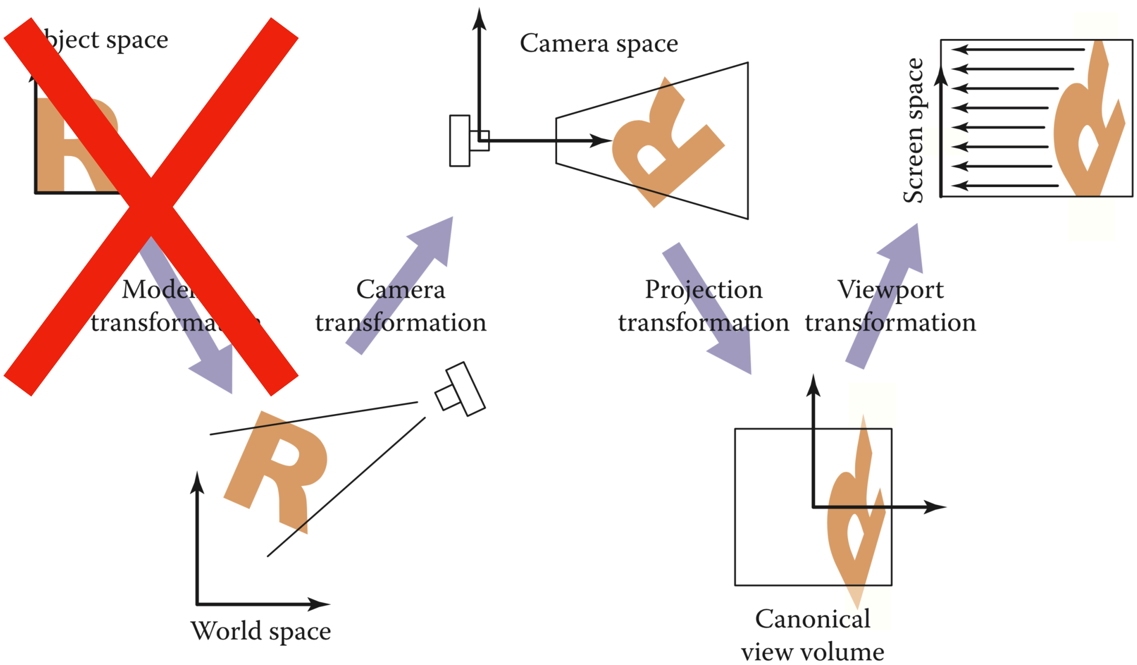


Orthographic **Perspective** **Perspective + Hidden Line Removal**

Step-by-Step Viewing Transformations

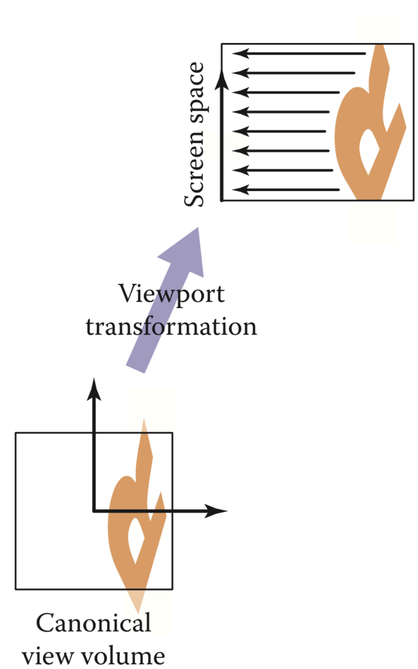
(Each arrow is a matrix)

We'll Discuss Later



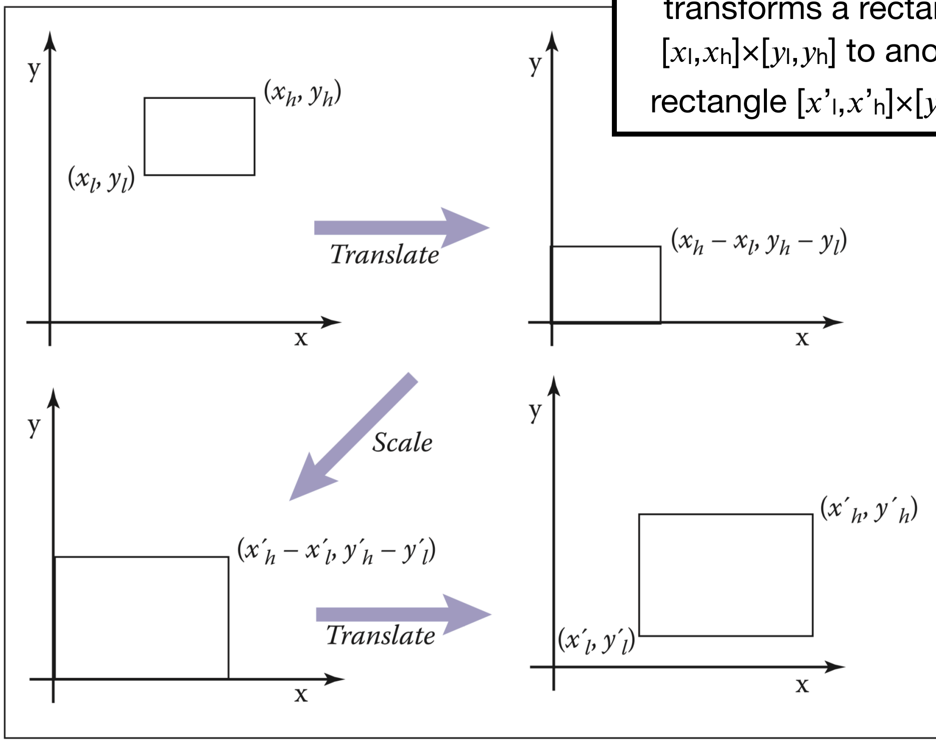
Viewport Transformation

- Goal: Transform from a canonical 2D space to pixel coordinates
  - Canonical space: $(x_{canonical}, y_{canonical}) \in [-1,1] \times [-1,1]$
  - Pixel space: $(x_{screen}, y_{screen}) \in [0.5, n_x - 0.5] \times [0.5, n_y - 0.5]$
- Initially, we will think of this as transformation of a 2D to 2D space

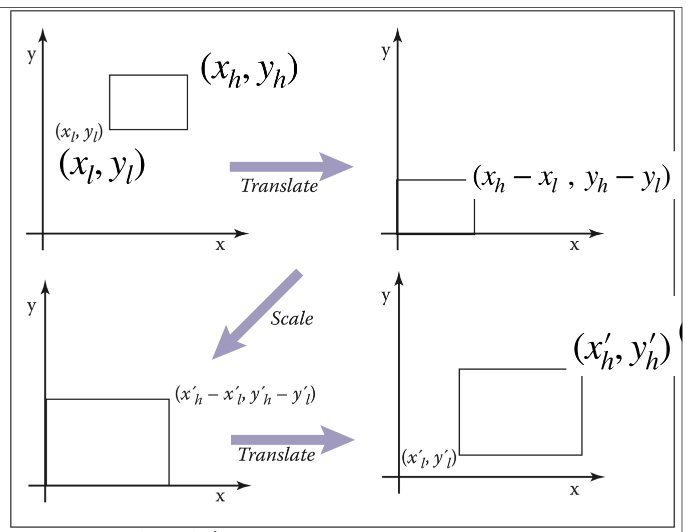


Viewports as Windowing

A **windowing operation** transforms a rectangle $[x_l,x_h]\times[y_l,y_h]$ to another rectangle $[x'_l,x'_h]\times[y'_l,y'_h]$



Viewports as Windowing

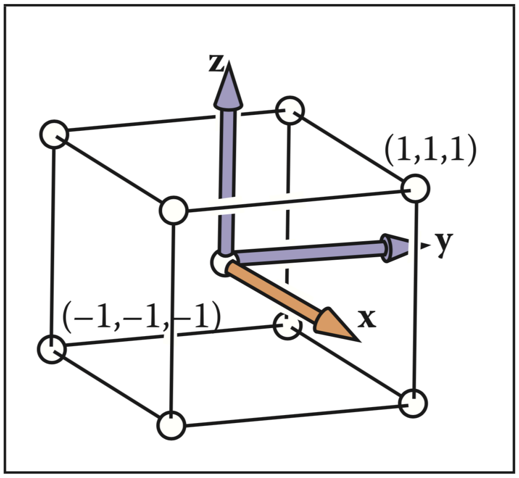


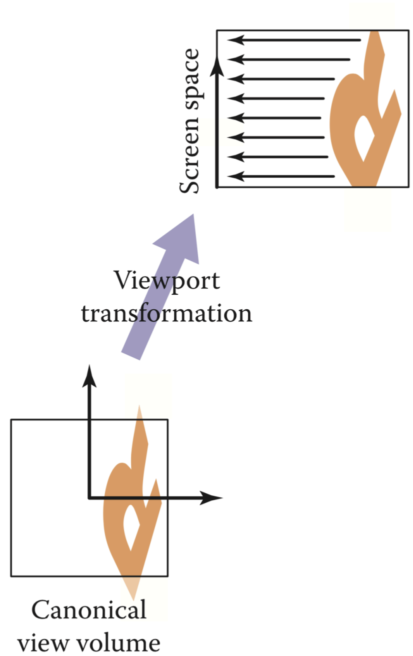
- Decompose windowing into three steps

$$\text{translate}\left(x'_l,\ y'_l\right)\ \text{scale}\left(\frac{x'_h - x'_l}{x_h - x_l},\ \frac{y'_h - y'_l}{y_h - y_l}\right)\ \text{translate}(-x_l, -y_l)$$

Canonical View Volume

- In actuality, our viewport transformation will work with the **canonical view volume**

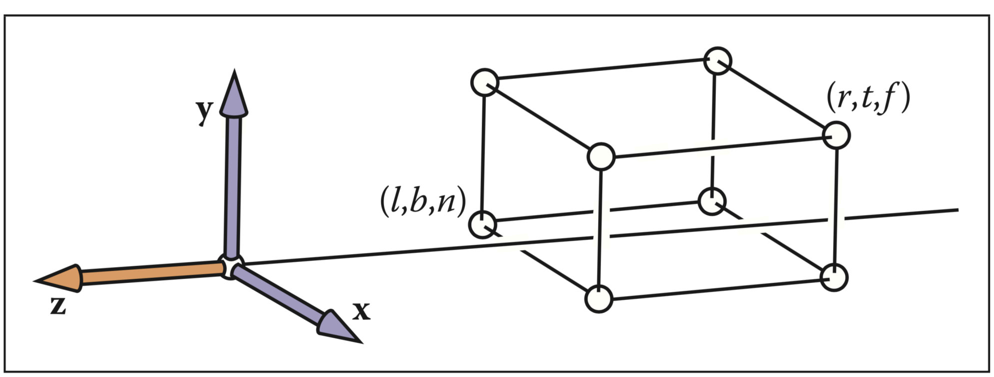




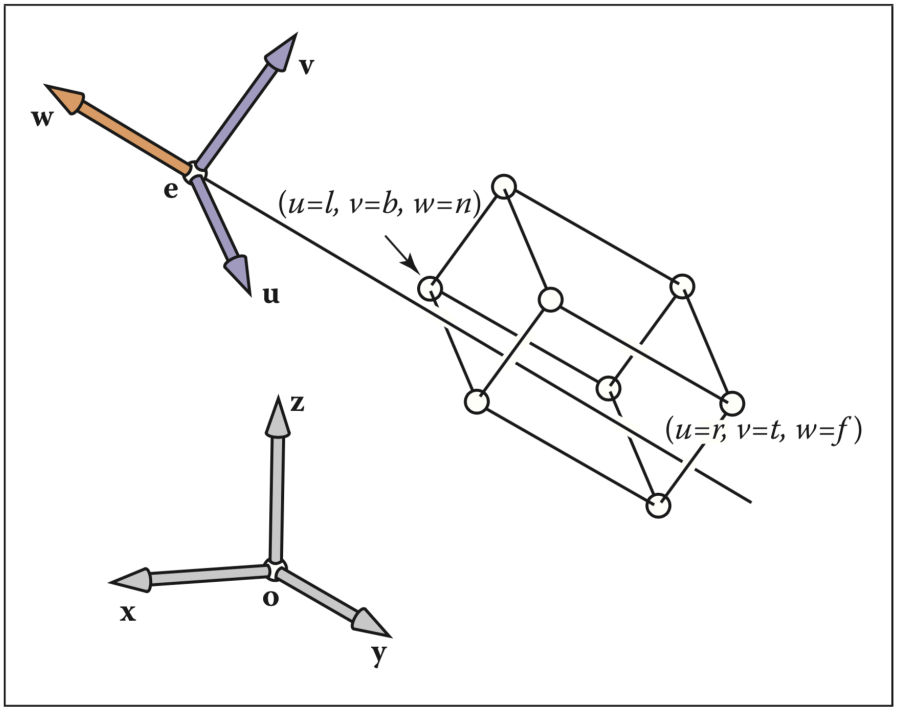
Orthographic Projection

- **Orthographic view volume** defined by six scalars:
- Convention: $n > f$, but note that both are *negative*

$$
\begin{aligned}
x = l &\equiv \text{left plane},\\
x = r &\equiv \text{right plane},\\
y = b &\equiv \text{bottom plane},\\
y = t &\equiv \text{top plane},\\
z = n &\equiv \text{near plane},\\
z = f &\equiv \text{far plane}.
\end{aligned}
$$



Camera Coordinates



Changing Coordinates

- We need to both translate the origin and change coordinate systems

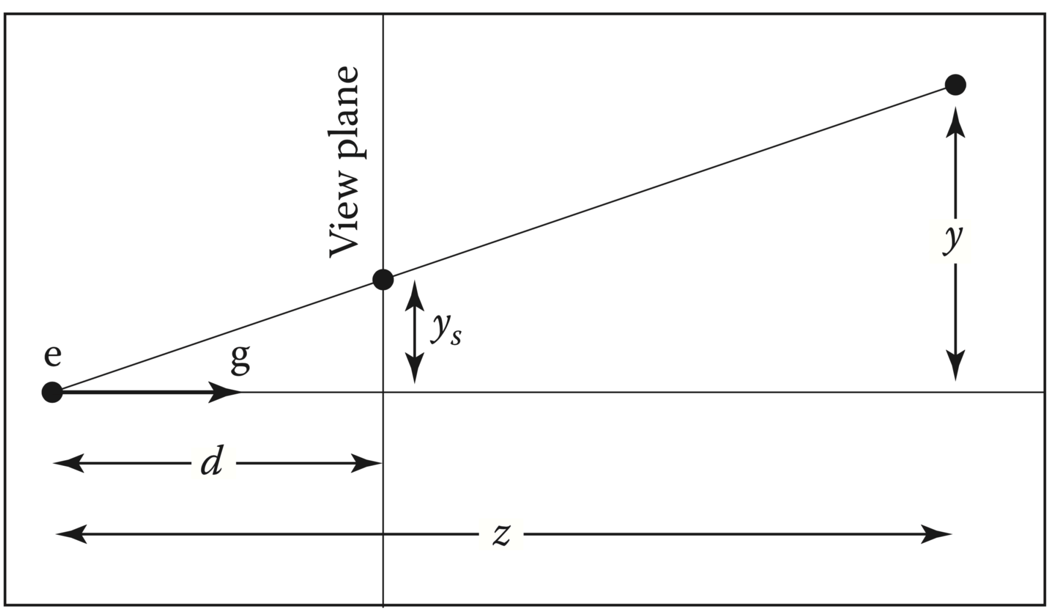
$$\mathbf{M}_{\text{cam}} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Projective Transformations (note: Transformation is not a Projection)

Relative Size Based on Distance

- Key idea of perspective: the size of an object on the screen is proportional to 1/z

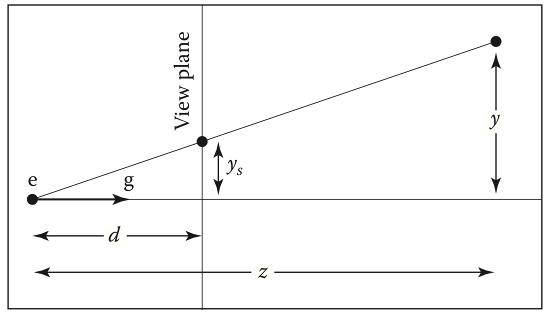
$$y_s = \frac{d}{z}\, y$$



Using Homogenious coord for Perspective
First attempt that will only partially work

- We can now replace:

$$y_s = \frac{d}{z}\, y$$



- With:

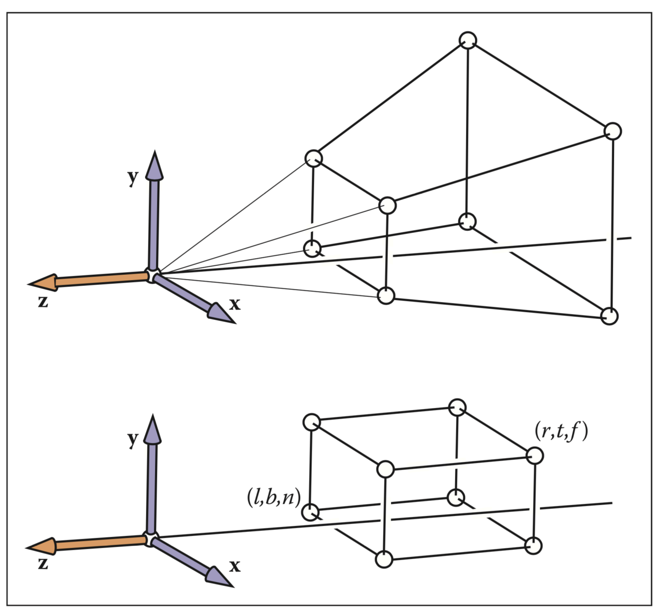
$$\begin{bmatrix} y_s \\ 1 \end{bmatrix} \sim \begin{bmatrix} d & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} y \\ z \\ 1 \end{bmatrix}$$

$$\begin{pmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = (dx, dy, z, z) = \left(\frac{x}{z}, \frac{y}{z}, \frac{z}{z}, 1\right)$$

Perspective Matrix

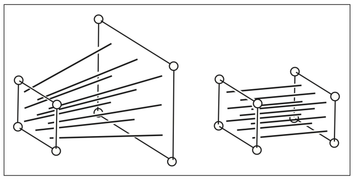
- Our matrix:

$$\mathbf{P} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}$$



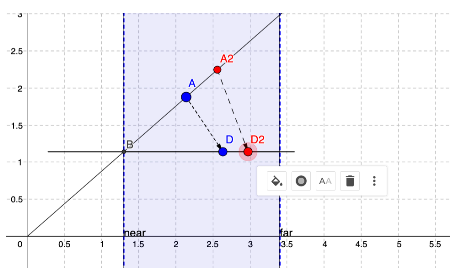
- Keeps near plane fixed, maps far plane to back of the box

Perspective Distortion



$$\mathbf{P} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

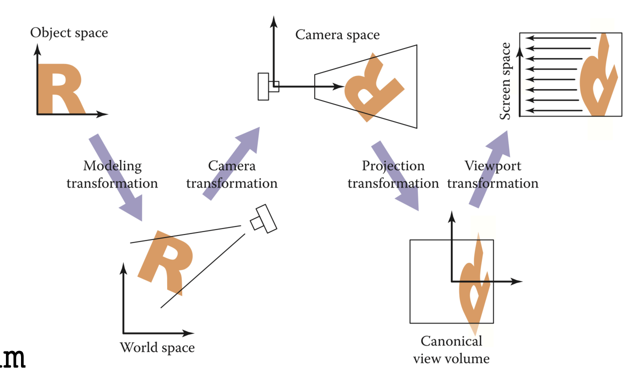
- Effect on view rays / lines:
- Note that affine transformation cannot do this because it keeps parallel lines parallel
  Perspective matrix effect on coordinates is nonlinear distortion in z:
- But it does, however, **preserve order** in the z-coordinate (which will become useful very soon)



GG

Putting it all together

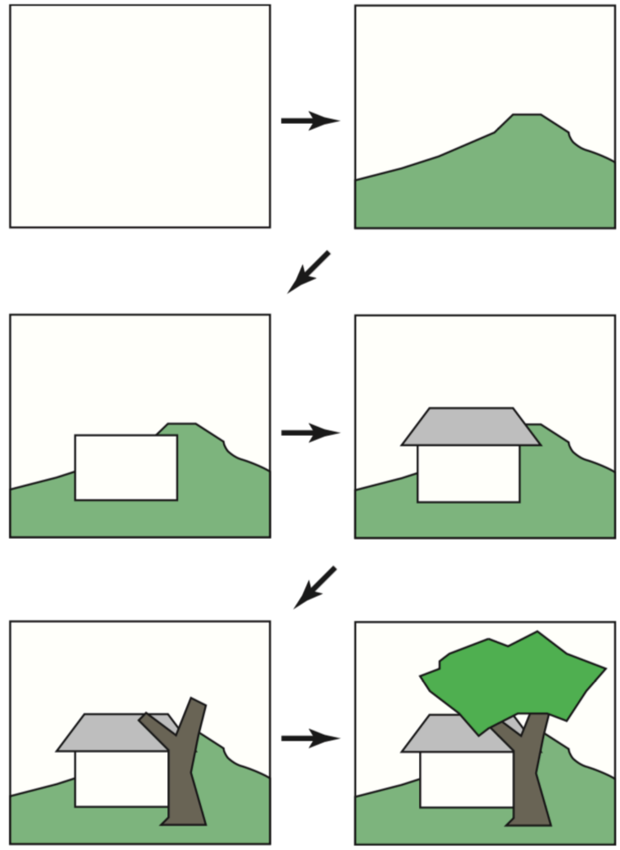
Equivalently: $\mathbf{M} = \mathbf{M}_{\text{vp}}\mathbf{M}_{\text{orth}}\mathbf{P}\mathbf{M}_{\text{cam}}$




```
construct M_vp
construct M_per
construct M_cam
M = M_vp * M_per * M_cam
for each 3D object O {
  O_screen = M * O
  draw(O_screen)
}
```

For a given vertex $\mathbf{a} = (x,y,z)$, $\mathbf{p} = \mathbf{Ma}$ should result in drawing $(x_\mathbf{p}/w_\mathbf{p}, y_\mathbf{p}/w_\mathbf{p}, z_\mathbf{p}/w_\mathbf{p})$ on the screen

Painter's Algorithm



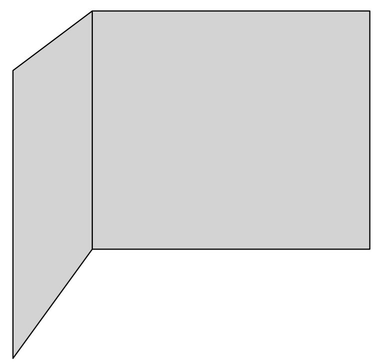
- Simple way to do hidden surfaces
- Draw from back-to-front, overwriting directly on the image

Painter's Algorithm

- Draw one primitive at a time.

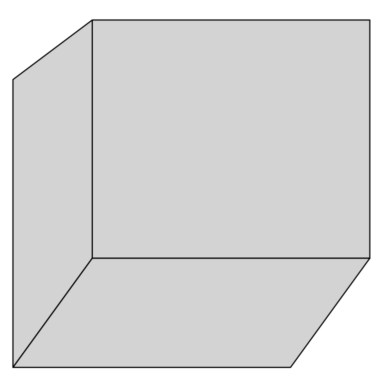
Painter’s Algorithm

- Draw one primitive at a time.



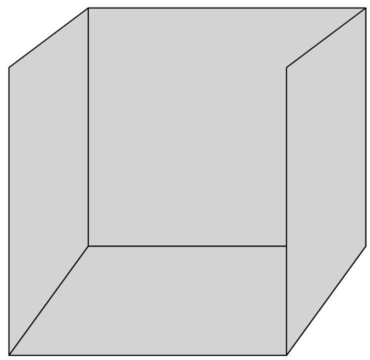
Painter’s Algorithm

- Draw one primitive at a time.



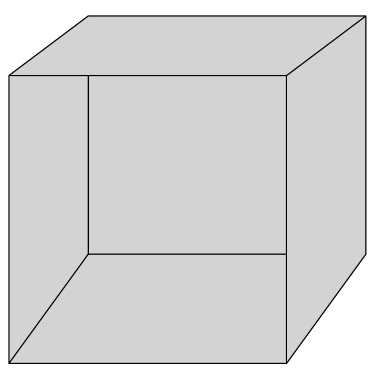
Painter's Algorithm

- Draw one primitive at a time.



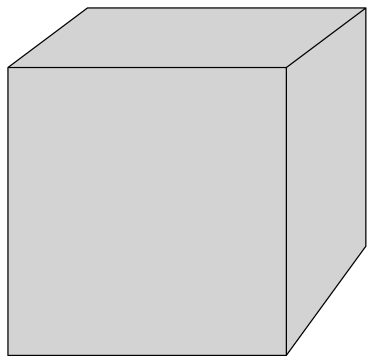
Painter's Algorithm

- Draw one primitive at a time.



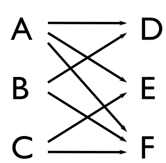
Painter's Algorithm

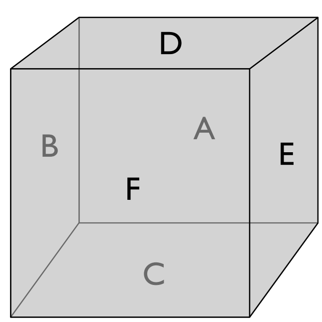
- Draw one primitive at a time.



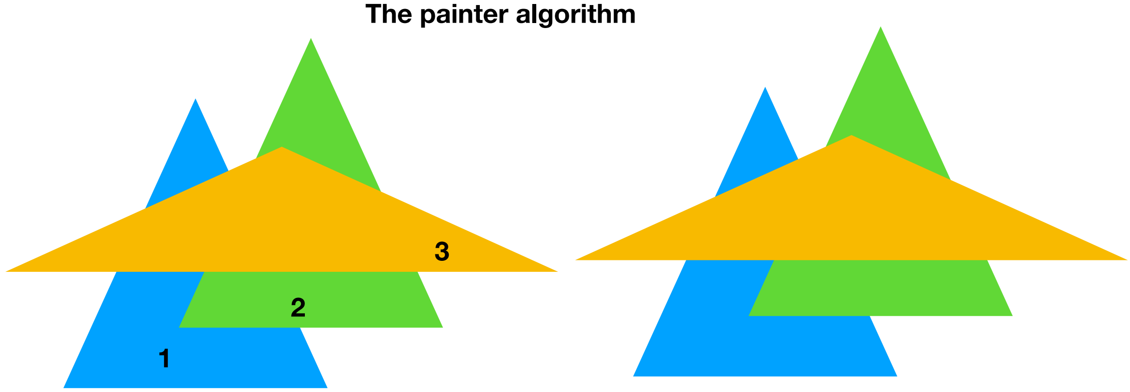
Painter's algorithm

- **Amounts to a topological sort of the graph of occlusions**
  - that is, an edge from A to B means A sometimes occludes B
  - any sort is valid
    - ABCDEF
    - BADCFE
  - if there are cycles there is no sort





The painter algorithm



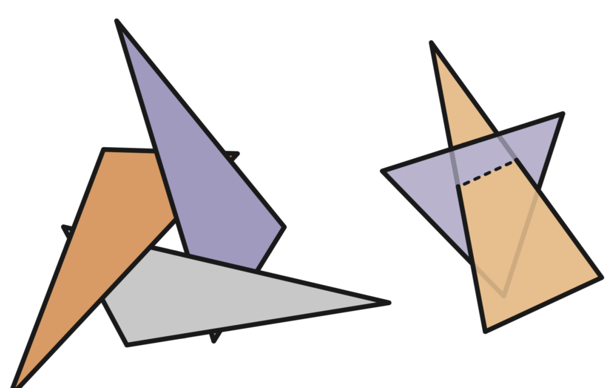
In this context, to ``render a triangle” $\mathbf{t}_i$ means
``set the color of each the pixel that $\mathbf{t}_i$ contains, to the color of $\mathbf{t}_i$

- Input - a set of triangles. Each with a different color
- Sort them, in decreasing distance from the viewer.

- (the order might not be unique)
- Render them, start from the one which is furthest away from the viewer, and end with the one closest to the viewer
- When rendering a triangle, we ignore all other triangles.
- In general, the order must satisfies: If **A occludes any part of B**, then B should be rendered **earlier**.

Painter’s algorithm

- **Amounts to a topological sort of the graph of occlusions**
  – that is, an edge from A to B means A sometimes occludes B
  – any sort is valid
    - ABCDEF
    - BADCFE
  – if there are cycles there is no sort



Cycles cannot be sorted

Using a z-Buffer for Hidden Surfaces

- Most of the time, sorting the primitives in z is too expensive and complex
- Solution: draw primitives in any order, but keep track of closest at the **fragment (pixel)** level
    - Method: use an extra data structure that tracks **the closest fragment in depth. (**this is the **depth buffer, or Z-buffer)**
    - When drawing, compare fragment's depth to closest current depth and discard the one that is further away