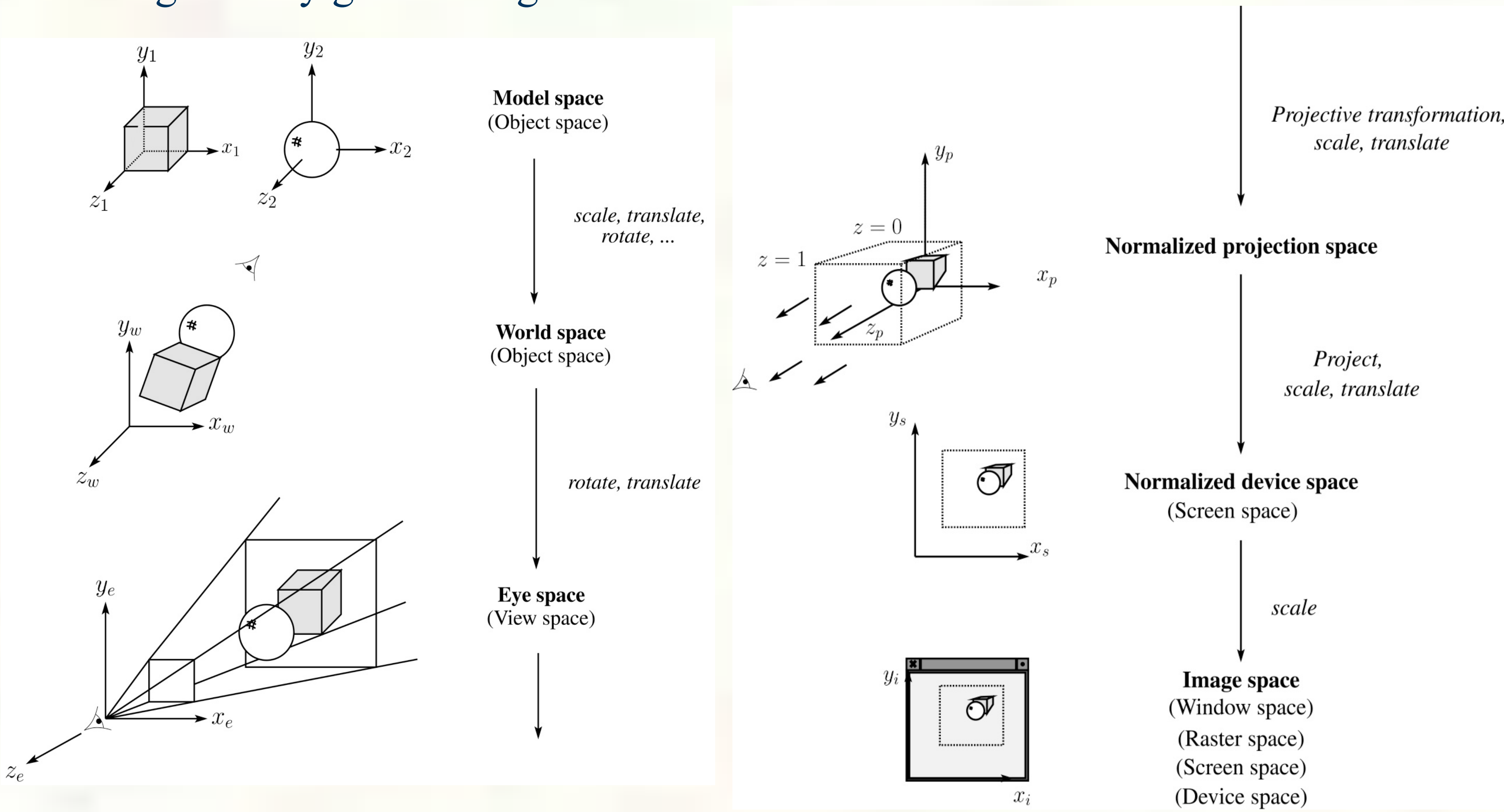
Projections and Z-buffers

Reading

- Required:
  - Watt, Section 5.2.2 – 5.2.4, 6.3, 6.6 (esp. intro and subsections 1, 4, and 8–10),
- Further reading:
  - Foley, et al, Chapter 5.6 and Chapter 6
  - David F. Rogers and J. Alan Adams, *Mathematical Elements for Computer Graphics*, 2nd Ed., McGraw-Hill, New York, 1990, Chapter 2.
  - I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, A characterization of ten hidden surface algorithms, *ACM Computing Surveys* 6(1): 1-55, March 1974.

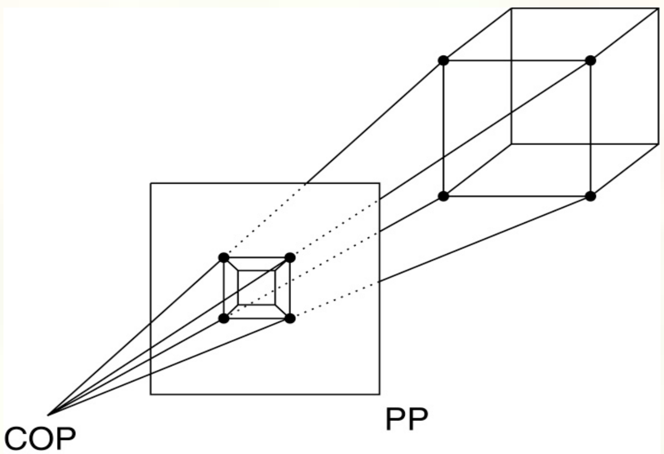
3D Geometry Pipeline

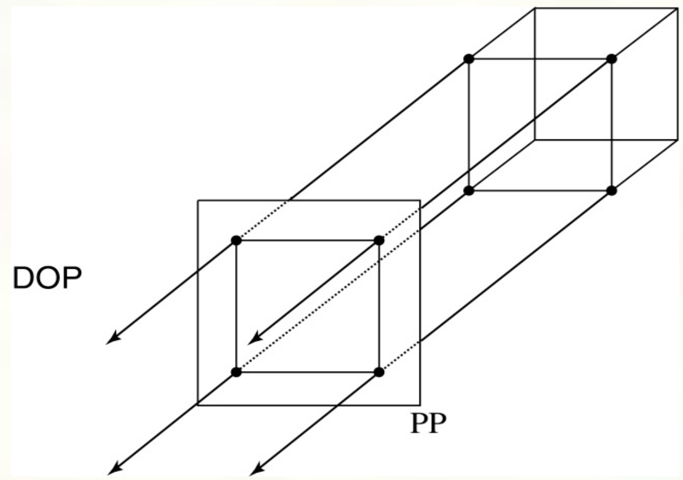
- Before being turned into pixels by graphics hardware, a piece of geometry goes through a number of transformations...



Projections

- **Projections** transform points in *n*-space to *m*-space, where $m<n$.
- In 3-D, we map points from 3-space to the **projection plane** (PP) along **projectors** emanating from the **center of projection** (COP):





- The center of projection is exactly the same as the pinhole in a pinhole camera.
- There are two basic types of projections:
  - Perspective – distance from COP to PP finite
  - Parallel – distance from COP to PP infinite

Parallel projections

- For parallel projections, we specify a **direction of projection** (DOP) instead of a COP.
- There are two types of parallel projections:
  - **Orthographic projection** – DOP perpendicular to PP
  - **Oblique projection** – DOP not perpendicular to PP
- We can write orthographic projection onto the z =0 plane with a simple matrix.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

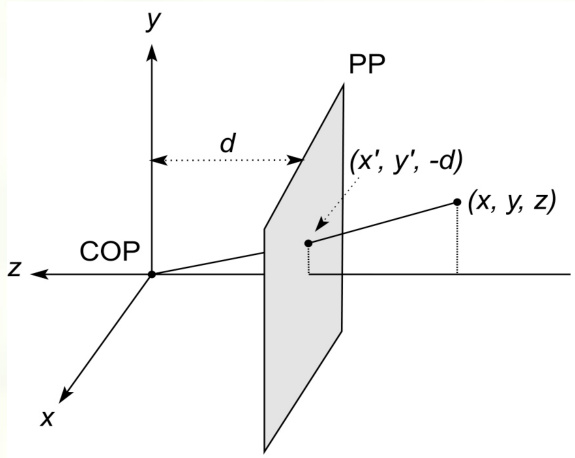
- But normally, we do not drop the z value right away. Why not?

Properties of parallel projection

- Properties of parallel projection:
  - Not realistic looking
  - Good for exact measurements
  - Are actually a kind of affine transformation
    - Parallel lines remain parallel
    - Angles not (in general) preserved
  - Most often used in CAD, architectural drawings, etc., where taking exact measurement is important

Derivation of perspective projection

- Consider the projection of a point onto the projection plane:



$$\frac{x'}{x} = -\frac{d}{z}$$

$$\frac{y'}{y} = -\frac{d}{z}$$

- By similar triangles, we can compute how much the *x* and *y* coordinates are scaled:

$$x' = -\frac{d}{z}x \qquad y' = -\frac{d}{z}y$$

- [Note: Watt uses a left-handed coordinate system, and he looks down the $+z$ axis, so his PP is at $+d$.]

Homogeneous coordinates revisited

- Remember how we said that affine transformations work with the last coordinate always set to one.
- What happens if the coordinate is not one?
- We divide all the coordinates by *W*:

$$\begin{bmatrix} X/W \\ Y/W \\ Z/W \\ W/W \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- If $W = 1$, then nothing changes.
- Sometimes we call this division step the "perspective divide."

Homogeneous coordinates and perspective projection

- Now we can re-write the perspective projection as a matrix equation:

$$\begin{bmatrix} X \\ Y \\ W \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ -z/d \end{bmatrix}$$

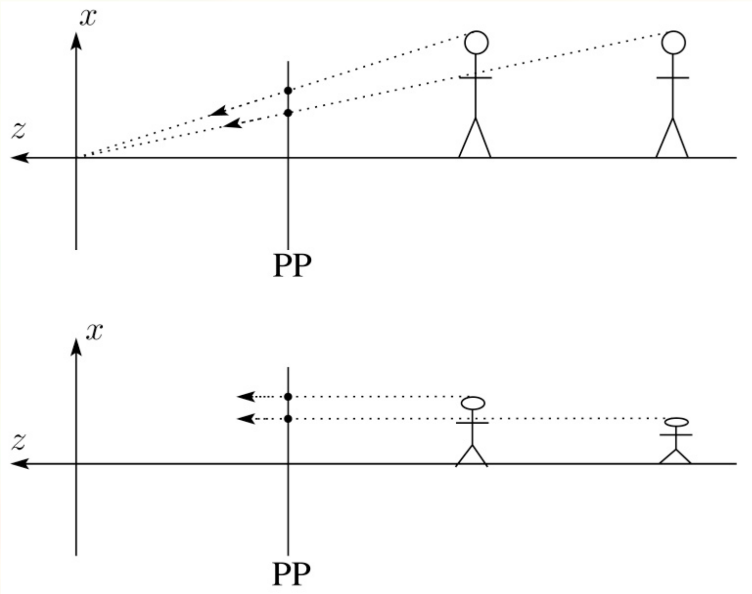
- After division by W, we get:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -\frac{x}{z} d \\ -\frac{y}{z} d \\ 1 \end{bmatrix}$$

- Again, projection implies dropping the z coordinate to give a 2D image, but we usually keep it around a little while longer.

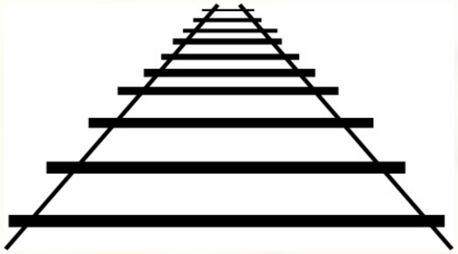
Projective normalization

- After applying the perspective transformation and dividing by *w*, we are free to do a simple parallel projection to get the 2D image.
- What does this imply about the shape of things after the perspective transformation + divide?



Vanishing points

- What happens to two parallel lines that are not parallel to the projection plane?
- Think of train tracks receding into the horizon...



- The equation for a line is: $\ell = \mathbf{p} + t\mathbf{v} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} + t \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix}$
- After perspective transformation we get: $\begin{bmatrix} X \\ Y \\ W \end{bmatrix} = \begin{bmatrix} p_x + tv_x \\ p_y + tv_y \\ -(p_z + tv_z)/d \end{bmatrix}$

Vanishing points (cont'd)

- Dividing by W:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -\frac{p_x + tv_x}{p_z + tv_z} d \\ -\frac{p_y + tv_y}{p_z + tv_z} d \\ \frac{-(p_z + tv_z)/d}{-(p_z + tv_z)/d} \end{bmatrix}$$

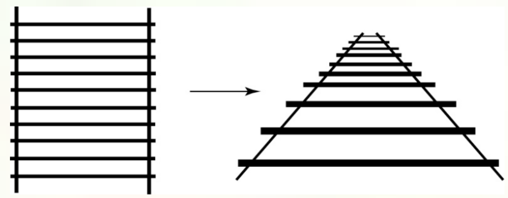
- Letting t go to infinity:

$$\begin{bmatrix} -\frac{v_x}{v_z} \\ -\frac{v_y}{v_z} \\ 1 \end{bmatrix}$$

- We get a point that depends only on **v**
- What happens to the line $\ell = \mathbf{q} + t\mathbf{v}$?
- Each set of parallel lines intersect at a **vanishing point** on the PP.
- **Q**: How many vanishing points are there?

Properties of perspective projections

- The perspective projection is an example of a **projective transformation**.



- Here are some properties of projective transformations:
  - Lines map to lines
  - Parallel lines do not necessarily remain parallel
  - Ratios are not preserved
- One of the advantages of perspective projection is that size varies inversely with distance – looks realistic.
- A disadvantage is that we can't judge distances as exactly as we can with parallel projections.
- **Q**: Why did nature give us eyes that perform perspective projections?
- **Q**: Do our eyes "see in 3D"?

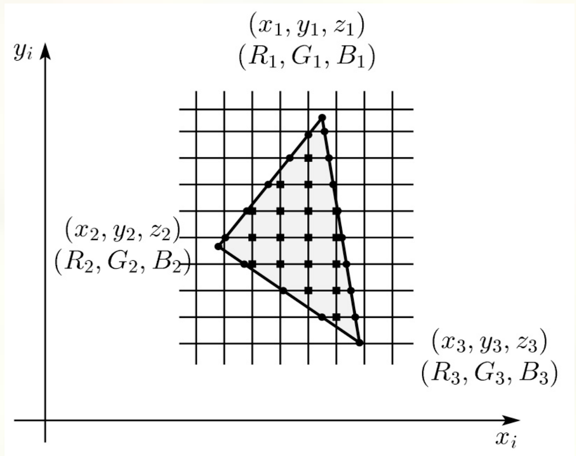
Z-buffer

- We can use projections for **hidden surface elimination**.
- The **Z-buffer'** or **depth buffer** algorithm [Catmull, 1974] is probably the simplest and most widely used of these techniques.
- Here is pseudocode for the Z-buffer hidden surface algorithm:

```
for each pixel (i,j) do
    Z-buffer [i,j]  ← FAR
    Framebuffer[i,j] ← <background color>
end for
for each polygon A do
    for each pixel in A do
        Compute depth z and shade s of A at (i,j)
        if z > Z-buffer [i,j] then
            Z-buffer [i,j] ← z
            Framebuffer[i,j] ← s
        end if
    end for
end for
```

Z-buffer, cont'd

- The process of filling in the pixels inside of a polygon is called **rasterization**.
- During rasterization, the *z* value and shade *s* can be computed incrementally (fast!).



Curious fact:

- Described as the "brute-force image space algorithm" by [SSS]
- Mentioned only in Appendix B of [SSS] as a point of comparison for huge memories, but written off as totally impractical.

Today, Z-buffers are commonly implemented in hardware.

Ray tracing vs. Z-Buffer

Ray tracing:

```
for each ray {
  for each object {
    test for intersection
  }
}
```

Z-Buffer:

```
for each object {
  project_onto_screen;
  for each ray {
    test for intersection
  }
}
```

In both cases, optimizations are applied to the inner loop.

Biggest differences:

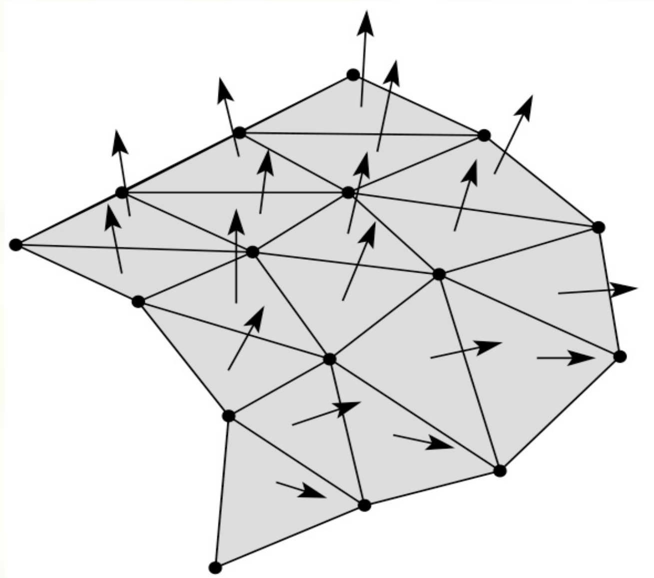
- ray order vs. object order
- Z-buffer does some work in screen space
- Z-buffer restricted to rays from a single center of projection!

Gouraud vs. Phong interpolation

- Does Z-buffer graphics hardware do a full shading calculation at every point? Not in the past, but this has changed!
- Smooth surfaces are often approximated by polygonal facets, because:
  - Graphics hardware generally wants polygons (esp. triangles).
  - Sometimes it easier to write ray-surface intersection algorithms for polygonal models.
- How do we compute the shading for such a surface?

Faceted shading

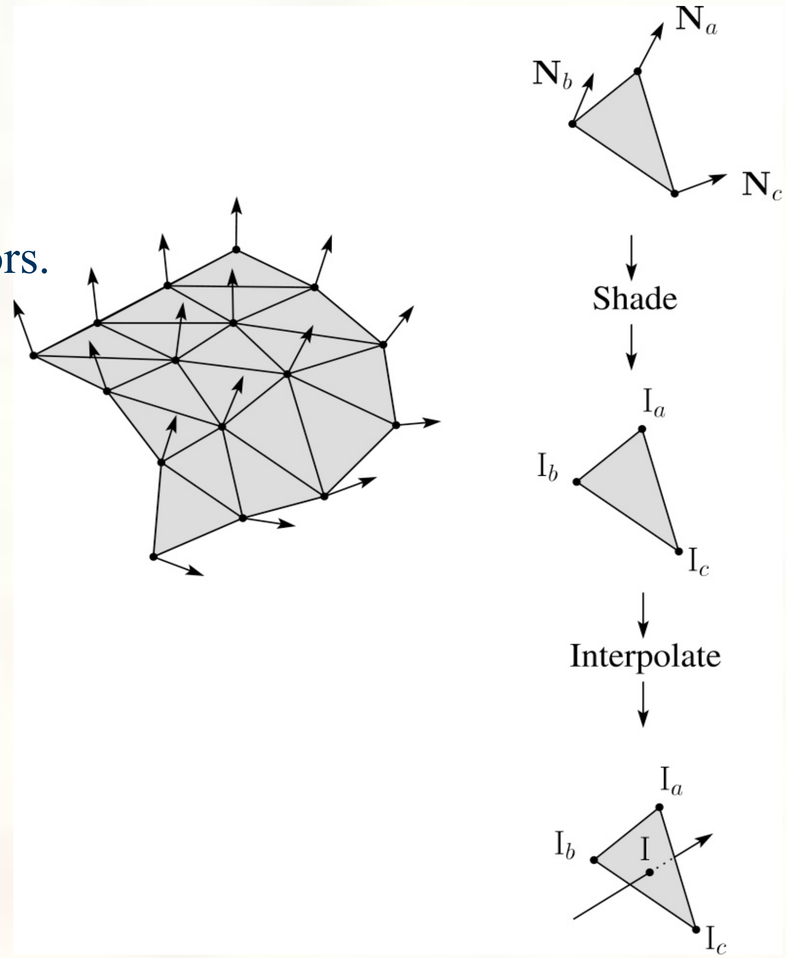
- Assume each face has a constant normal:



- For a distant viewer and a distant light source, how will the color of each triangle vary?
- Result: faceted, not smooth, appearance.

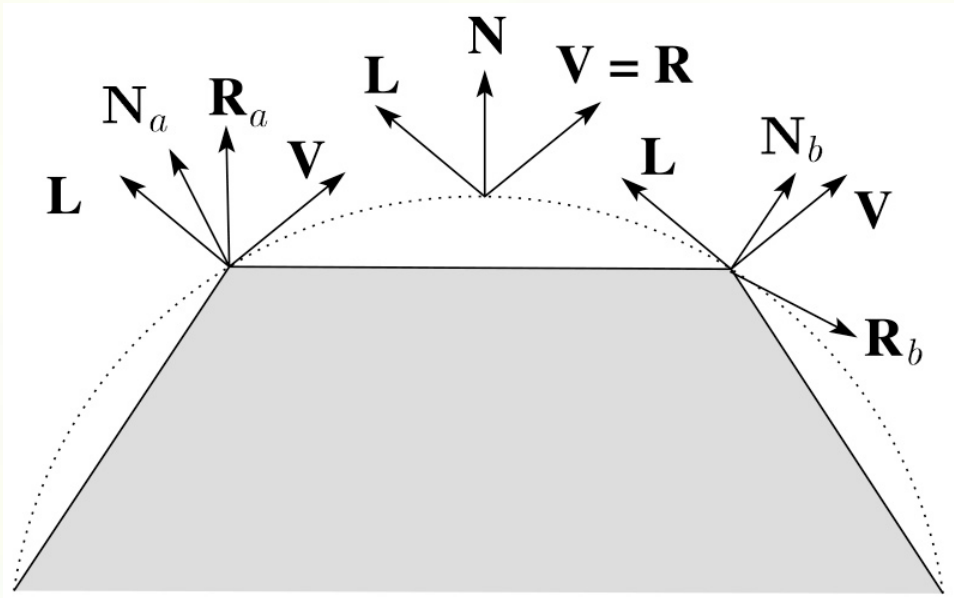
Gouraud interpolation

- To get a smoother result that is easily performed in hardware, we can do **Gouraud interpolation**.
- Here's how it works:
  - Compute normals at the vertices.
  - Shade only the vertices.
  - Interpolate the resulting vertex colors.



Gouraud interpolation, cont'd

- Gouraud interpolation has significant limitations.
  - If the polygonal approximation is too coarse, we can miss specular highlights.

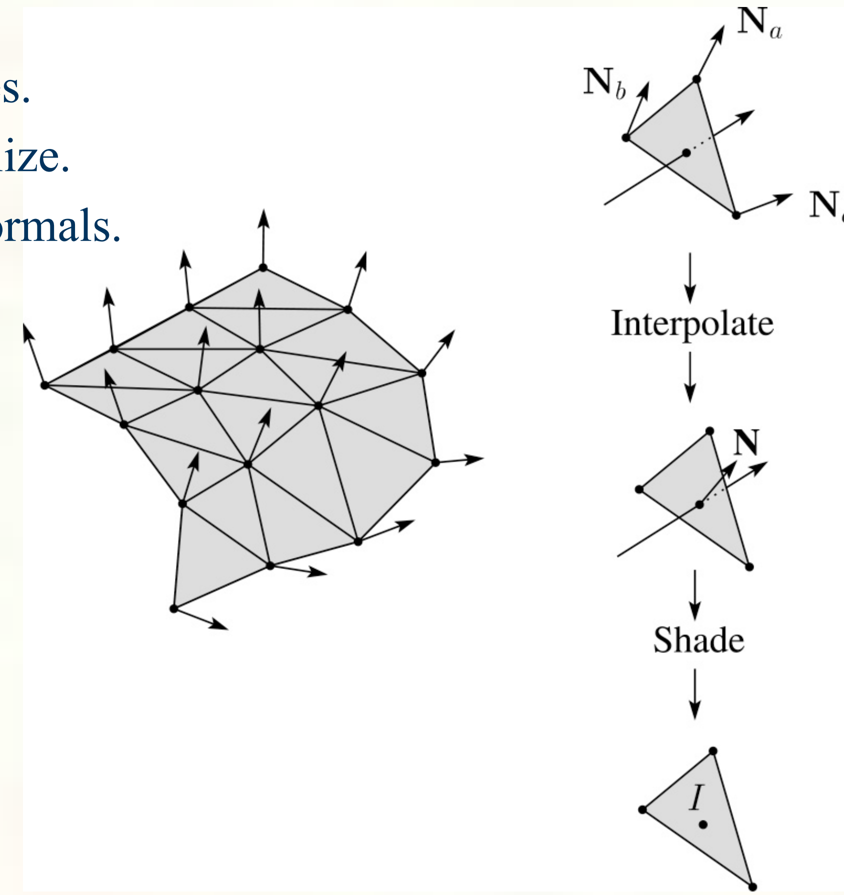


  - We will encounter **Mach banding** (derivative discontinuity enhanced by human eye).

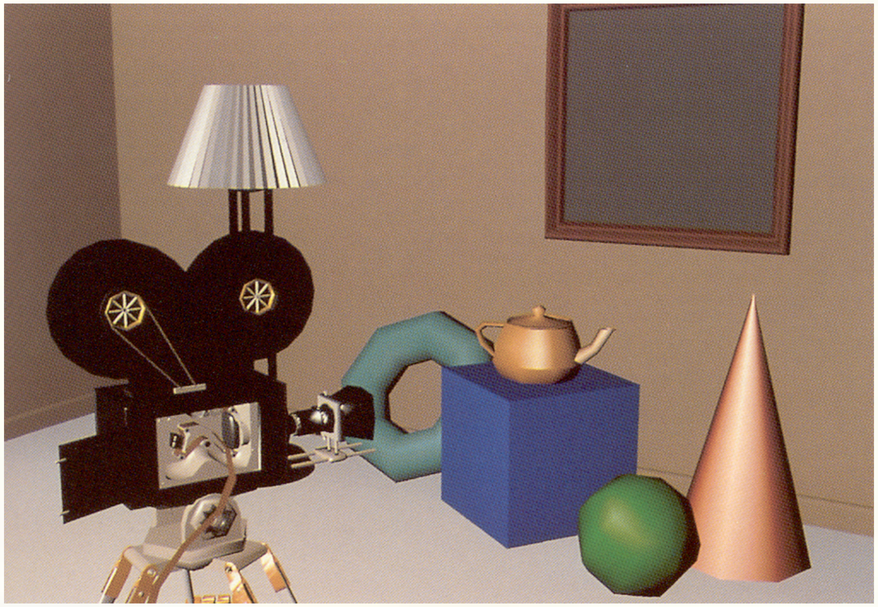
- Alas, this is usually what graphics hardware supported until very recently.
- But new graphics hardware supports…

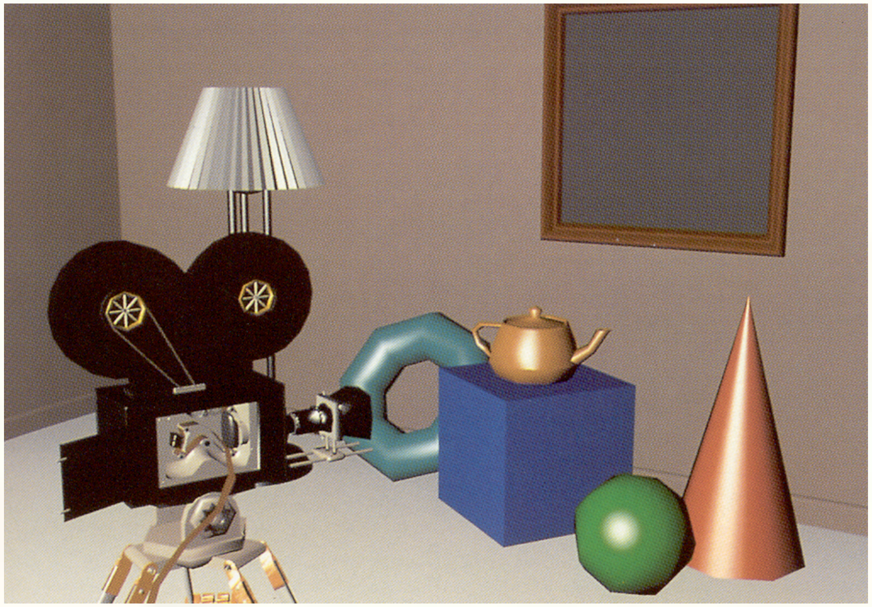
Phong interpolation

- To get an even smoother result with fewer artifacts, we can perform **Phong *interpolation***.
- Here's how it works:
  1. Compute normals at the vertices.
  2. Interpolate normals and normalize.
  3. Shade using the interpolated normals.



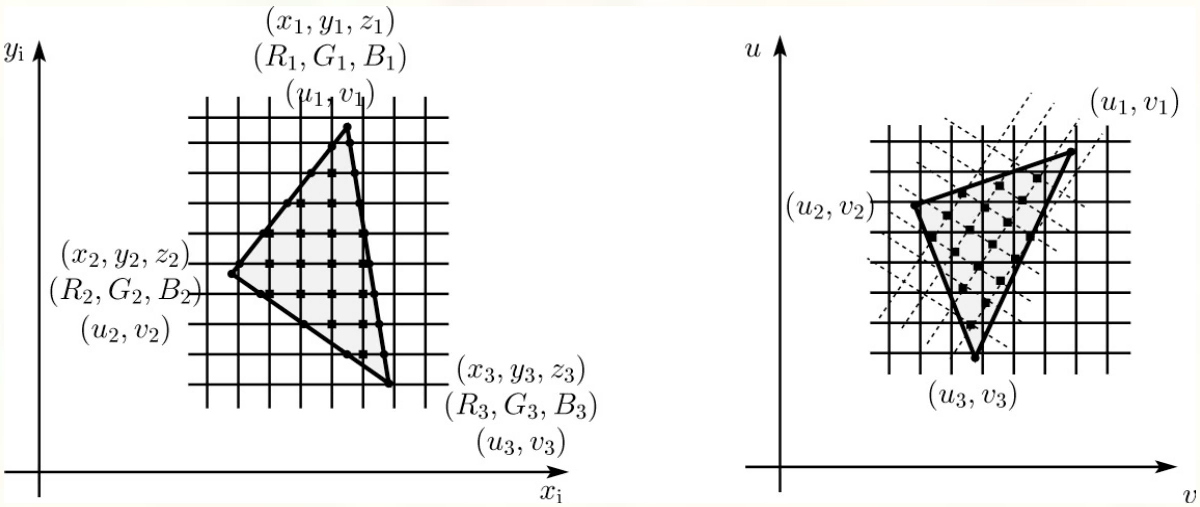
Gouraud vs. Phong interpolation





Texture mapping and the z-buffer

- Texture-mapping can also be handled in z-buffer algorithms.
- Method:
  - Scan conversion is done in screen space, as usual
  - Each pixel is colored according to the texture
  - Texture coordinates are found by Gouraud-style interpolation

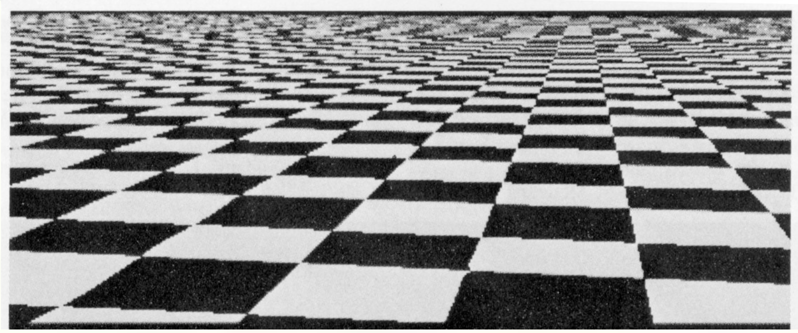


- Note: Mapping is more complicated if you want to do perspective right!

  - linear in world space != linear in screen space

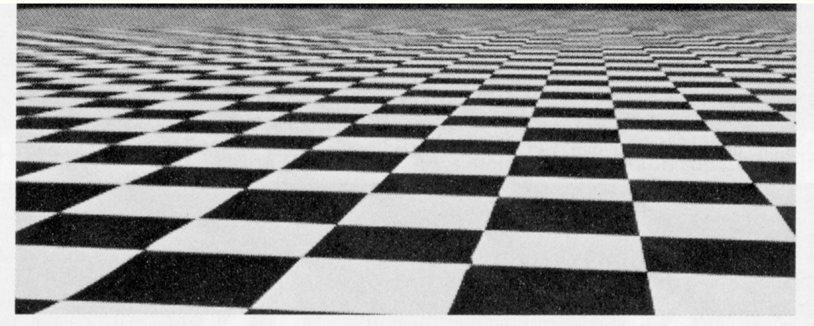
Antialiasing textures

- If you render an object with a texture map using point-sampling, you can get aliasing:



From Crow, SIGGRAPH '84

- Proper antialiasing requires area averaging over pixels:

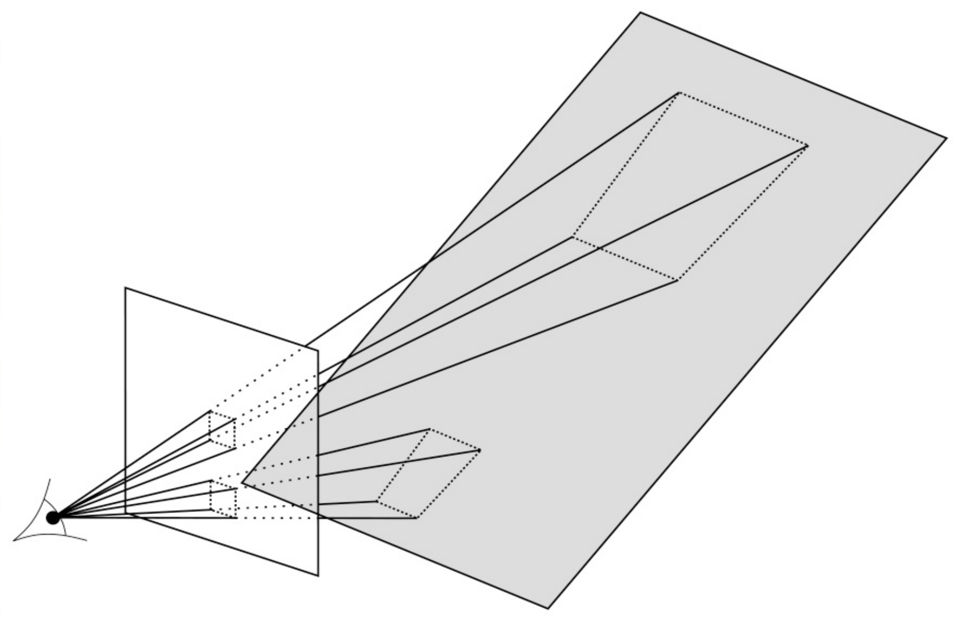


From Crow, SIGGRAPH '84

- In some cases, you can average directly over the texture pixels to do the anti-aliasing.

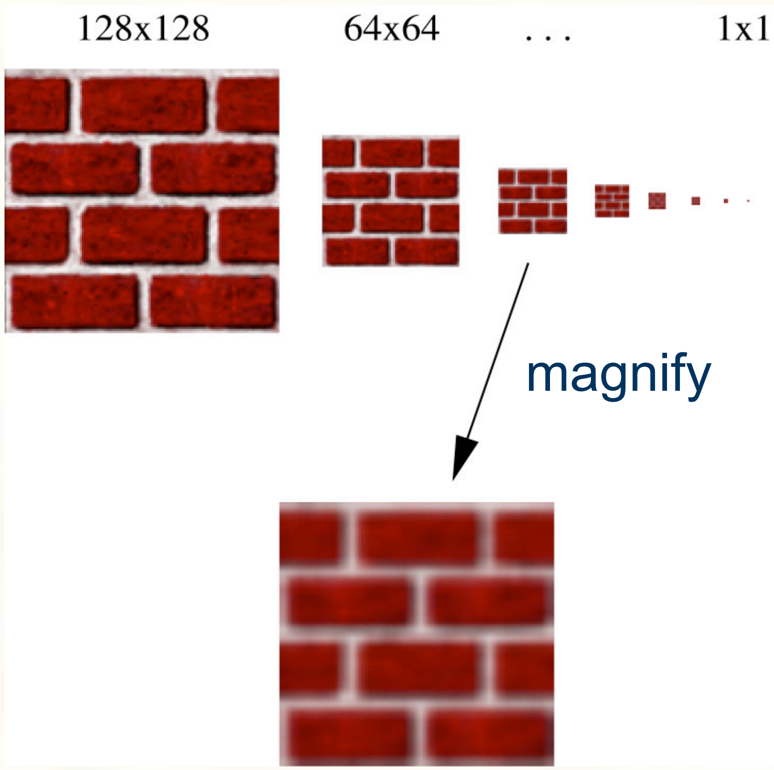
Computing the average color

- The computationally difficult part is summing over the covered pixels.
- Several methods have been used.
- The simplest is **brute force**:
  - Figure out which texels are covered and add up their colors to compute the average.

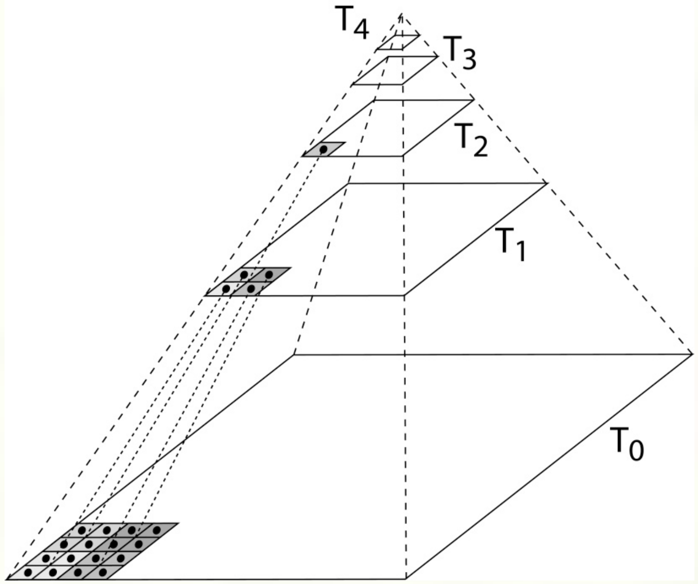


Mipmaps

- A faster method is **mip maps** developed by Lance Williams in 1983:
  - Stands for "multum in parvo" – many things in a small place
  - Keep textures prefiltered at multiple resolutions
  - Has become the graphics hardware standard

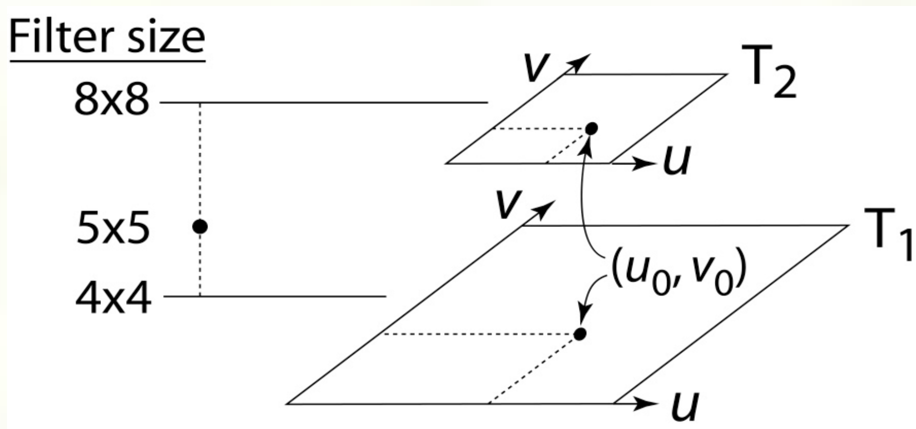


Mipmap pyramid



- The mip map hierarchy can be thought of as an image pyramid:
  - Level 0 ($T_0[i,j]$) is the original image.
  - Level 1 ($T_1[i,j]$) averages over 2x2 neighborhoods of original.
  - Level 2 ($T_2[i,j]$) averages over 4x4 neighborhoods of original
  - Level 3 ($T_3[i,j]$) averages over 8x8 neighborhoods of original
- What's a fast way to pre-compute the texture map for each level?

Mipmap resampling



- What would the mipmap return for an average over a 5 x 5 neighborhood at location (u_0, v_0)?

- How do we measure the fractional distance between levels?

- What if you need to average over a non-square region?

Summed area tables

- A more accurate method than mipmaps is **summed area tables** invented by Frank Crow in 1984.
- Recall from calculus:

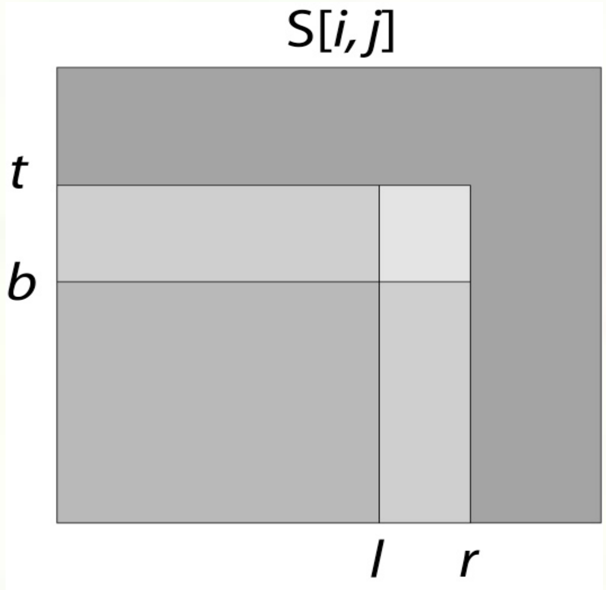
$$\int_a^b f(x)dx = \int_{-\infty}^b f(x)dx - \int_{-\infty}^a f(x)dx$$

- In discrete form: $$\sum_{i=k}^{m} f[i] = \sum_{i=0}^{m} f[i] - \sum_{i=0}^{k} f[i]$$

- **Q**: If we wanted to do this real fast, what might we pre-compute?

Summed area tables (cont'd)

- We can extend this idea to 2D by creating a table, S[i,j], that contains the sum of everything below and to the left.



- **Q**: How do we compute the average over a region from (l, b) to (r, t)?
- Characteristics:
  - Requires more memory and precision
  - Gives less blurry textures

Comparison of techniques

Point sampled

MIP-mapped

Summed area table

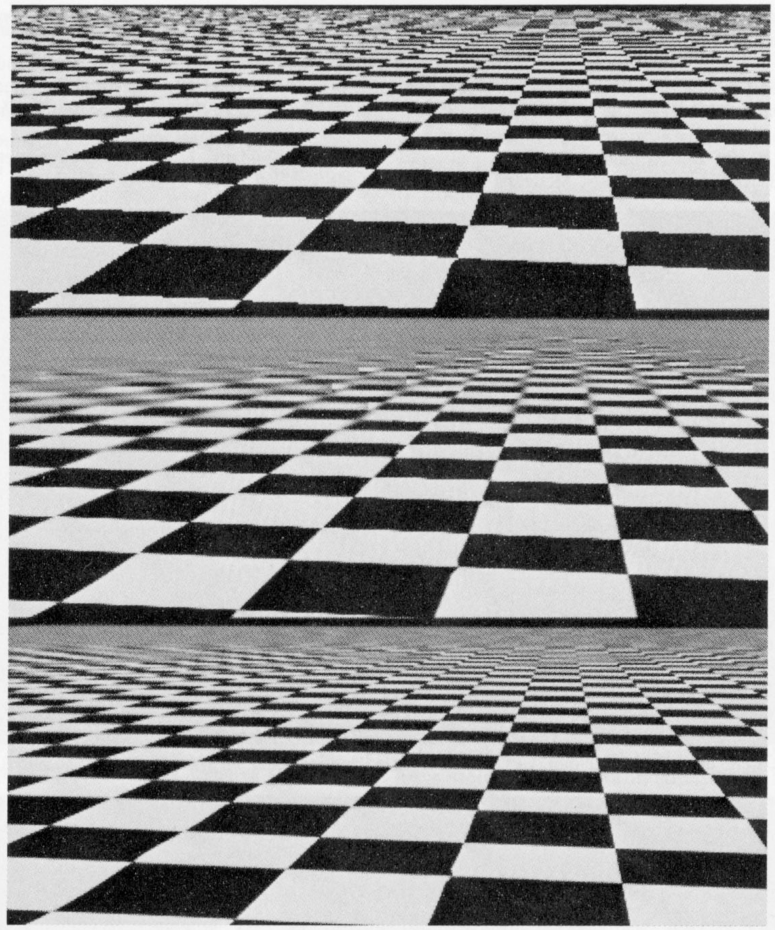


Figure 5: CheckerBoards mapped onto a square showing vertically compressed texture.

From Crow, SIGGRAPH '84

Cost of Z-buffering

- Z-buffering is ***the*** algorithm of choice for hardware rendering (today), so let's think about how to make it run as fast as possible…
- The steps involved in the Z-buffer algorithm are:
  1. Send a triangle to the graphics hardware.
  2. Transform the vertices of the triangle using the modeling matrix.
  3. Transform the vertices using the projection matrix.
  4. Set up for incremental rasterization calculations
  5. Rasterize
     (generate "fragments" = potential pixels)
  6. Shade at each fragment
  7. Update the framebuffer according to *z*.

- What is the overall cost of Z-buffering?

Cost of Z-buffering, cont'd

- We can approximate the cost of this method as:

$$k_{bus} v_{bus} + k_{xform} v_{xform} + k_{setup} t + k_{shade} (dm^2)$$

where:

k_{bus} = bus cost to send a vertex
v_{bus} = number of vertices sent over the bus
k_{xform} = cost of transforming a vertex
v_{xform} = number of vertices transformed
k_{setup} = cost of setting up for rasterization
t = number of triangles being rasterized
k_{shade} = cost of shading a fragment
d = depth complexity
(average times a pixel is covered)
m^2 = number of pixels in frame buffer

Accelerating Z-buffers

- Given this cost function:

$$k_{bus} v_{bus} + k_{xform} v_{xform} + k_{setup} t + k_{shade} (dm^2)$$

what can we do to accelerate Z-buffering?

Accel method	v_{bus}	v_{xform}	t	d	m

Next class: Visual Perception

- Topic:
  How does the human visual system?
  How do humans perceive color?
  How do we represent color in computations?
- Read:
  - Glassner, Principles of Digital Image Synthesis, pp. 5-32. [Course reader pp.1-28]
  - Watt , Chapter 15.
  - Brian Wandell. Foundations of Vision. Sinauer Associates, Sunderland, MA, pp. 45-50 and 69-97, 1995.
    [Course reader pp. 29-34 and pp. 35-63]