Z-Buffer Cons
- requires lots of memory
- (e.g. 1280x1024x32 bits)
- requires fast memory
- Read-Modify-Write in inner loop
- hard to simulate translucent polygons
- we throw away color of polygons behind closest one
- works if polygons ordered back-to-front
- extra work throws away much of the speed advantage

Hidden Surface Removal
- two kinds of visibility algorithms
  - object space methods
  - image space methods

Object Space Algorithms
- determine visibility on object or polygon level
  - using camera coordinates
  - resolution independent
  - explicitly compute visible portions of polygons
  - early in pipeline
  - after clipping
  - requires depth-sorting
  - painter’s algorithm
  - BSP trees

Image Space Algorithms
- perform visibility test for in screen coordinates
- limited to resolution of display
- Z-buffer: check every pixel independently
- performed late in rendering pipeline

Z-Buffer Algorithm Questions
- how much memory does the Z-buffer use?
- does the image rendered depend on the drawing order?
- how does Z-buffer load scale with visible polygons?
- with framebuffer resolution?

Review: Depth Test Precision
- reminder: perspective transformation maps eye-space (view) z to NDC z
- thus:
\[
z_{\text{NDC}} = \frac{C \cdot D}{z} + z_{\text{eye}}
\]

Correction: Ortho Camera Projection
- camera’s back plane parallel to lens
- infinite focal length
- no perspective convergence
- just throw away z values
- x and y coordinates do not change with respect to z in this projection

Z-Buffer Pros
- simple!!!
- easy to implement in hardware
  - hardware support in all graphics cards today
  - polygons can be processed in arbitrary order
  - easily handles polygon interpenetration
  - enables deferred shading
  - rasterize shading parameters (e.g., surface normal) and only shade final visible fragments

Z-Buffer Cons
- poor for scenes with high depth complexity
- need to render all polygons, even if most are invisible
- shared edges are handled inconsistently
- ordering dependent

Z-Buffer Algorithm
- augment color framebuffer with Z-buffer or depth buffer which stores Z value at each pixel
  - at frame beginning, initialize all pixel depths to \( \infty \)
  - when rasterizing, interpolate depth (Z) across polygon
  - check Z-buffer before storing pixel color in framebuffer and storing depth in Z-buffer
  - don’t write pixel if its Z value is more than the Z value already stored there

More: BSP Trees
- runtime: correctly traversing this tree enumerates objects from back to front
  - viewpoint dependent: check which side of plane viewpoint is on at each node
  - draw far, draw object in question, draw near

Review: BSP Trees
- preprocess: create binary tree
  - recursive spatial partition
  - viewpoint independent

Review: BSP Trees
- runtime: correctly traversing this tree enumerates objects from back to front
  - viewpoint dependent: check which side of plane viewpoint is on at each node
  - draw far, draw object in question, draw near

Hidden Surfaces III
- two kinds of visibility algorithms
  - object space methods
  - image space methods

How does Z-buffer load scale with visible polygons?
- with framebuffer resolution?

Z-Buffer Cons
- poor for scenes with high depth complexity
- need to render all polygons, even if most are invisible
- shared edges are handled inconsistently
- ordering dependent

Z-Buffer Pros
- simple!!!
- easy to implement in hardware
  - hardware support in all graphics cards today
  - polygons can be processed in arbitrary order
  - easily handles polygon interpenetration
  - enables deferred shading
  - rasterize shading parameters (e.g., surface normal) and only shade final visible fragments

Z-Buffer Cons
- requires lots of memory
  - (e.g. 1280x1024x32 bits)
- requires fast memory
  - Read-Modify-Write in inner loop
  - hard to simulate translucent polygons
  - we throw away color of polygons behind closest one
  - works if polygons ordered back-to-front
  - extra work throws away much of the speed advantage

Hidden Surface Removal
- two kinds of visibility algorithms
  - object space methods
  - image space methods

Object Space Algorithms
- determine visibility on object or polygon level
  - using camera coordinates
  - resolution independent
  - explicitly compute visible portions of polygons
  - early in pipeline
  - after clipping
  - requires depth-sorting
  - painter’s algorithm
  - BSP trees
Back-Face Culling

- not rendering backfacing polygons improves performance
- by how much?
  - reduces by about half the number of polygons to be considered for each pixel
  - optimization when appropriate

Invisible Primitives

- why might a polygon be invisible?
  - polygon outside the field of view / frustum
    - solved by clipping
  - polygon is back-facing
    - solved by backface culling
  - polygon is occluded by object(s) nearer the viewpoint
    - solved by hidden surface removal

Texturing

- examples of non-manifold objects:
  - a single polygon
  - a terrain or height field
  - polyhedron w/ missing face
  - anything with cracks or holes in boundary
  - one-polygon thick lampshade

Backface Culling: NDCS

- why might a polygon be invisible?
  - polygon outside the field of view / frustum
    - solved by clipping
  - polygon is back-facing
    - solved by backface culling
  - polygon is occluded by object(s) nearer the viewpoint
    - solved by hidden surface removal

Rendering Pipeline

- how might you combine multiple elements?
  - foreground color A, background color B

Blending/Compositing

- specify opacity with alpha channel (e.g., {r, g, b, a})
  - {r, g, b}: opaque, {0, 255}, transparent, {0, 0, 0}, transparent
  - A over B
    - C = ωA + (1 - ω)B
    - but what if B is also partially transparent?
      - C = ωA + (1 - ω)B
      - ω = B / (B + A) = 1 / (1 + B / A)
        - 3 equations, different equations for alpha vs. RGB
  - premultiplying by alpha
    - C' = ωC + (1 - ω)B
    - C = B' + A', A' = ωA
    - 1 multiply to find C, same equations for alpha and RGB
Texture Mapping

- introduced to increase realism
  - lighting/shading models not enough
  - hide geometric simplicity
  - images convey illusion of geometry
  - map a brick wall texture on a flat polygon
  - create blurry effect on surface
  - associate 2D information with 3D surface
  - point on surface corresponds to a point in texture
  - "paint" image onto polygon

Texture Functions

- once have value from the texture map, can:
  - directly use as surface color: GL_REPLACE
  - throw away old color, lose lighting effects
  - modulate surface color: GL_MODULATE
  - multiply old color by new value, keep lighting info
  - texturing happens after lighting, not relit
  - use as surface color, modulate alpha: GL_DECAL
    - new value controls which if 2 colors to use
    - indirect, new value not used directly for coloring
  - specify with glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, <mode>)
    - [demo]

Texture Coordinates

- texture image: 2D array of color values (texels)
- assigning texture coordinates (s,t) at vertex with object coordinates (x,y,z,w)
- use interpolated (s,t) for texel lookup at each pixel
- use value to modify a polygon's color
  - or other surface property
- specified by programmer or artist

Example Texture Map

```
1
5

(0, 0)
(1, 1)
(0, 0)
(1, 1)
```

Texture Mapping

- real life objects have nonuniform colors, normals
- to generate realistic objects, reproduce coloring & normal variations = texture
- can often replace complex geometric details

Basic OpenGL Texturing

- create a texture object and fill it with texture data:
  - glTexImage2D(GL_TEXTURE_2D, identifier) to bind
  - following texture commands refer to the bound texture
  - glTexImage2D(GL_TEXTURE_2D, ..., ...) to specify parameters for use when applying the texture
  - glTexImage2D(GL_TEXTURE_2D, ..., ...) to specify the texture data
    - the image itself
  - enable texturing: glEnable(GL_TEXTURE_2D)
  - state how the texture will be used:
    - glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, <mode>)
  - specify texture coordinates for the polygon:
    - use glTexCoord2f(s, t) before each vertex:
Low-Level Details

- large range of functions for controlling layout of texture data
- state how the data in your image is arranged
  - e.g.: `glPixelStoref(GL_UNPACK_ALIGNMENT, 1)` tells OpenGL not to skip bytes at the end of a row
- you must state how you want the texture to be put in memory: how many bits per "pixel", which channels...
- textures must be square and size a power of 2
- common sizes are 32x32, 64x64, 256x256
- smaller uses less memory, and there is a finite amount of texture memory on graphics cards
- ok to use texture template sample code for project 4

Texture Mapping

- texture coordinates
  - specified at vertices
    - `glTexCoord2f(s, t);` 
    - `glVertexf(x, y, z);`
  - interpolated across triangle (like R, G, B, Z)
  - ...well not quite!

Texture Coordinate Interpolation

- perspective correct interpolation
  - `α`, `β`, `γ`:
  - barycentric coordinates of a point `P` in a triangle
  - `s0, s1, s2`:
  - texture coordinates of vertices
  - `w0, w1, w2`:
  - homogeneous coordinates of vertices
    - `(x1,y1,z1,w1)`
    - `(x0,y0,z0,w0)`
    - `(x2,y2,z2,w2)`
    - `(α, β, γ)`
    - `(s, t)`?

MIPmaps

- multum in parvo -- many things in a small place
- prespecify a series of prefilled texture maps of decreasing resolutions
- requires more texture storage
- avoid shimmering and flashing as objects move
- `glBuild2DMipmaps` automatically constructs a family of textures from original texture size down to 1x1
- only 1/3 more space required

Texture Parameters

- in addition to color can control other material/object properties
- surface normal (bump mapping)
- reflected color (environment mapping)

Bump Mapping: Normals As Texture

- object surface often not smooth – to recreate correctly need complex geometry model
- can control shape "effect" by locally perturbing surface normal
- random perturbation
- directional change over region

Bump Mapping

- at transitions
  - rotate point's surface normal by θ or -θ

Displacement Mapping

- bump mapping gets silhouettes wrong
- shadows wrong too
- change surface geometry instead
- only recently available with realtime graphics
- need to subdivide surface