News

- Midterm handed back
  - solutions posted
  - distribution posted
  - all grades so far posted
- P1 Hall of Fame posted
- P3 grading
  - after 3:20
- P4 proposals
  - email or conversation to all
H3 Corrections/Clarifications

- Q1 should be from +infinity, not -infinity
- Q2-4 correction for point B
- Q7 clarified: only x and y coordinates are given for P
- Q8 is deleted
Review: 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

```c
glTexCoord2f(s, t)
glVertexf(x, y, z, w)
```
Review: Tiled Texture Map

```
glTexCoord2d(1, 1);
glVertex3d (x, y, z);

(1,0) + (0,0) = (1, 0)  
(1,1) + (0,1) = (1, 1)  
(4,0) + (0,0) = (4, 0)  
(4,4) + (0,4) = (4, 4)
```

Texture  +  Object = Mapped Texture
Review: Fractional Texture Coordinates

(texture image)

(0,0) (1,0) (0,.5) (.25,.5)

(0,1) (1,1) (0,.5) (.25,.5)
Review: Texture

- action when s or t is outside [0…1] interval
  - tiling
  - clamping
- functions
  - replace/decal
  - modulate
  - blend
- texture matrix stack
  
  ```
glMatrixMode( GL_TEXTURE );
  ```
Review: Basic OpenGL Texturing

■ setup
  ■ generate identifier: glGenTextures
  ■ load image data: glTexImage2D
  ■ set texture parameters (tile/clamp/...):
    glTexParameter
  ■ set texture drawing mode (modulate/replace/...):
    glTexEnvf

■ drawing
  ■ enable: glEnable
  ■ bind specific texture: glBindTexture
  ■ specify texture coordinates before each vertex:
    glTexCoord2f
Review: Perspective Correct Interpolation

- screen space interpolation incorrect

\[ s = \frac{\alpha \cdot s_0 / w_0 + \beta \cdot s_1 / w_1 + \gamma \cdot s_2 / w_2}{\alpha / w_0 + \beta / w_1 + \gamma / w_2} \]
Review: Reconstruction

how to deal with:

- pixels that are much larger than texels?
  - apply filtering, “averaging”

- pixels that are much smaller than texels?
  - interpolate
Review: MIPmapping

- image pyramid, precompute averaged versions
Review: Bump Mapping: Normals As Texture

- create illusion of complex geometry model
- control shape effect by locally perturbing surface normal
Review: Environment Mapping

- cheap way to achieve reflective effect
  - generate image of surrounding
  - map to object as texture
Review: Sphere Mapping

- texture is distorted fish-eye view
  - point camera at mirrored sphere
  - spherical texture coordinates
Review: Cube Mapping

- 6 planar textures, sides of cube
  - point camera outwards to 6 faces
    - use largest magnitude of vector to pick face
    - other two coordinates for (s,t) texel location
Review: Volumetric Texture

- define texture pattern over 3D domain - 3D space containing the object
  - texture function can be digitized or procedural
  - for each point on object compute texture from point location in space
- 3D function $\rho(x,y,z)$
function marble(point)
    x = point.x + turbulence(point);
    return marble_color(sin(x))
Review: Perlin Noise

- coherency: smooth not abrupt changes
- turbulence: multiple feature sizes
Review: Generating Coherent Noise

- just three main ideas
  - nice interpolation
  - use vector offsets to make grid irregular
  - optimization
    - sneaky use of 1D arrays instead of 2D/3D one
Review: Procedural Modeling

- textures, geometry
  - nonprocedural: explicitly stored in memory
- procedural approach
  - compute something on the fly
    - not load from disk
  - often less memory cost
- visual richness
  - adaptable precision
- noise, fractals, particle systems
Review: Language-Based Generation

- **L-Systems**
  - F: forward, R: right, L: left
  - Koch snowflake: \( F = FLFRRFLF \)
  - Mariano’s Bush: 
    \[ F = FF - [-F + F + F] + [+F - F - F] \]
    - angle 16

http://spanky.triumf.ca/www/fractint/lsys/plants.html
Correction/Review: Fractal Terrain

- 1D: midpoint displacement
  - divide in half, randomly displace
  - scale variance by half
- 2D: diamond-square
  - generate new value at midpoint
  - average corner values + random displacement
    - scale variance by half each time

http://www.gameprogrammer.com/fractal.html
Review: Particle Systems

- changeable/fluid stuff
  - fire, steam, smoke, water, grass, hair, dust, waterfalls, fireworks, explosions, flocks
- life cycle
  - generation, dynamics, death
- rendering tricks
  - avoid hidden surface computations
Sampling
Samples

- most things in the real world are continuous
- everything in a computer is discrete
- the process of mapping a continuous function to a discrete one is called sampling
- the process of mapping a discrete function to a continuous one is called reconstruction
- the process of mapping a continuous variable to a discrete one is called quantization
- rendering an image requires sampling and quantization
- displaying an image involves reconstruction
Line Segments

- we tried to sample a line segment so it would map to a 2D raster display
- we quantized the pixel values to 0 or 1
- we saw stair steps, or jaggies
Line Segments

- instead, quantize to many shades
- but what sampling algorithm is used?
Unweighted Area Sampling

- shade pixels wrt area covered by thickened line
- equal areas cause equal intensity, regardless of distance from pixel center to area
  - rough approximation formulated by dividing each pixel into a finer grid of pixels
- primitive cannot affect intensity of pixel if it does not intersect the pixel
Weighted Area Sampling

- intuitively, pixel cut through the center should be more heavily weighted than one cut along corner
- weighting function, \( W(x,y) \)
  - specifies the contribution of primitive passing through the point \((x, y)\) from pixel center
Images

- an image is a 2D function $I(x, y)$ that specifies intensity for each point $(x, y)$
Image Sampling and Reconstruction

- convert **continuous** image to **discrete** set of samples
- display hardware **reconstructs** samples into continuous image
  - finite sized source of light for each pixel

![Diagram showing discrete input values and continuous light output](image)

- discrete input values
- continuous light output
Point Sampling an Image

- simplest sampling is on a grid
- sample depends solely on value at grid points
Point Sampling

- multiply sample grid by image intensity to obtain a discrete set of points, or samples.
Sampling Errors

- some objects missed entirely, others poorly sampled
  - could try unweighted or weighted area sampling
  - but how can we be sure we show everything?
- need to think about entire class of solutions!
Image As Signal

- image as spatial signal
- 2D raster image
  - discrete sampling of 2D spatial signal
- 1D slice of raster image
  - discrete sampling of 1D spatial signal

Examples from Foley, van Dam, Feiner, and Hughes
Sampling Theory

- how would we generate a signal like this out of simple building blocks?
- theorem
  - any signal can be represented as an (infinite) sum of sine waves at different frequencies
Sampling Theory in a Nutshell

- terminology
  - bandwidth – length of repeated sequence on infinite signal
  - frequency – 1/bandwidth (number of repeated sequences in unit length)

- example – sine wave
  - bandwidth = 2π
  - frequency = 1/ 2π

\[ \sin(t) \]
Summing Waves I

\[
\begin{align*}
sin(x) & \quad + \\
\frac{sin(3x)}{3} & \quad + \\
\frac{sin(5x)}{5} & \quad + \\
\frac{sin(7x)}{7} & \quad + \\
\frac{sin(9x)}{9} & \\
\end{align*}
\]
Summing Waves II

represent spatial signal as sum of sine waves (varying frequency and phase shift)

very commonly used to represent sound “spectrum”
1D Sampling and Reconstruction
1D Sampling and Reconstruction
1D Sampling and Reconstruction
1D Sampling and Reconstruction
1D Sampling and Reconstruction

- problems
  - jaggies – abrupt changes
1D Sampling and Reconstruction

- problems
  - jaggies – abrupt changes
  - lose data
Sampling Theorem

continuous signal can be completely recovered from its samples

iff

classic problem
	sampling rate greater than twice maximum frequency present in signal

- Claude Shannon
Nyquist Rate

- lower bound on sampling rate
  - twice the highest frequency component in the image’s spectrum
Falling Below Nyquist Rate

- when sampling below Nyquist Rate, resulting signal looks like a lower-frequency one
  - this is aliasing!
Nyquist Rate

\[ f_s < 2f \]
\[ f_s = 2f \]
\[ f_s > 2f \]
Aliasing

- incorrect appearance of high frequencies as low frequencies
- to avoid: antialiasing
  - supersample
    - sample at higher frequency
  - low pass filtering
    - remove high frequency function parts
    - aka prefiltering, band-limiting
Supersampling
Low-Pass Filtering
Low-Pass Filtering

Fig. 14.20 The sampling pipeline with filtering. (Courtesy of George Wolberg, Columbia University.)
Filtering

- low pass
  - blur

- high pass
  - edge finding
Previous Antialiasing Example

- texture mipmapping: low pass filter
Virtual Trackball
Virtual Trackball

- interface for spinning objects around
  - drag mouse to control rotation of view volume
- rolling glass trackball
  - center at screen origin, surrounds world
  - hemisphere “sticks up” in z, out of screen
  - rotate ball = spin world
Virtual Trackball

- know screen click: \((x, 0, z)\)
- want to infer point on trackball: \((x,y,z)\)
  - ball is unit sphere, so \(||x, y, z|| = 1.0\)
  - solve for \(y\)
Trackball Rotation

- correspondence:
  - moving point on plane from \((x, 0, z)\) to \((a, 0, c)\)
  - moving point on ball from \(p_1 = (x, y, z)\) to \(p_2 = (a, b, c)\)

- correspondence:
  - translating mouse from \(p_1\) (mouse down) to \(p_2\) (mouse up)
  - rotating about the axis \(n = p_1 \times p_2\)
Trackball Computation

- user defines two points
  - place where first clicked $p_1 = (x, y, z)$
  - place where released $p_2 = (a, b, c)$
- create plane from vectors between points, origin
  - axis of rotation is plane normal: cross product
    - $(p_1 - o) \times (p_2 - o): p_1 \times p_2$ if origin = (0,0,0)
  - amount of rotation depends on angle between lines
    - $p_1 \cdot p_2 = |p_1| |p_2| \cos \theta$
    - $|p_1 \times p_2| = |p_1| |p_2| \sin \theta$
- compute rotation matrix, use to rotate world
Visibility
Reading

- FCG Chapter 7
Rendering Pipeline

Geometry Database → Model/View Transform. → Lighting → Perspective Transform. → Clipping

Scan Conversion → Texturing → Depth Test → Blending → Frame-buffer
Covered So Far

- modeling transformations
- viewing transformations
- projection transformations
- clipping
- scan conversion
- lighting
- shading

- we now know everything about how to draw a polygon on the screen, except visible surface determination
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*

- for efficiency reasons, we want to avoid spending work on polygons outside field of view or back-facing
- for efficiency and correctness reasons, we need to know when polygons are occluded
Hidden Surface Removal
Occlusion

- for most interesting scenes, some polygons overlap

- to render the correct image, we need to determine which polygons occlude which
Painter’s Algorithm

- simple: render the polygons from back to front, “painting over” previous polygons
  - draw blue, then green, then orange
  - will this work in the general case?
Painter’s Algorithm: Problems

- *intersecting polygons* present a problem
- even non-intersecting polygons can form a cycle with no valid visibility order:
Analytic Visibility Algorithms

- early visibility algorithms computed the set of visible polygon fragments directly, then rendered the fragments to a display:
Analytic Visibility Algorithms

- **question:** what is the minimum worst-case cost of computing the fragments for a scene composed of $n$ polygons?

- **answer:** $O(n^2)$
Analytic Visibility Algorithms

- so, for about a decade (late 60s to late 70s) there was intense interest in finding efficient algorithms for hidden surface removal
- we’ll talk about two:
  - *Binary Space Partition (BSP) Trees*
  - *Warnock’s Algorithm*
Binary Space Partition Trees (1979)

- BSP Tree: partition space with binary tree of planes
  - idea: divide space recursively into half-spaces by choosing splitting planes that separate objects in scene
  - preprocessing: create binary tree of planes
  - runtime: correctly traversing this tree enumerates objects from back to front
Creating BSP Trees: Objects
Creating BSP Trees: Objects
Creating BSP Trees: Objects
Creating BSP Trees: Objects
Creating BSP Trees: Objects
Splitting Objects

- no bunnies were harmed in previous example
- but what if a splitting plane passes through an object?
  - split the object; give half to each node
Traversing BSP Trees

- tree creation independent of viewpoint
  - preprocessing step
- tree traversal uses viewpoint
  - runtime, happens for many different viewpoints
- each plane divides world into near and far
  - for given viewpoint, decide which side is near and which is far
    - check which side of plane viewpoint is on independently for each tree vertex
    - tree traversal differs depending on viewpoint!
- recursive algorithm
  - recurse on far side
  - draw object
  - recurse on near side
Traversing BSP Trees

query: given a viewpoint, produce an ordered list of (possibly split) objects from back to front:

renderBSP (BSPtree *T)
   BSPtree *near, *far;
   if (eye on left side of T->plane)
      near = T->left; far = T->right;
   else
      near = T->right; far = T->left;
   renderBSP (far);
   if (T is a leaf node)
      renderObject (T)
   renderBSP (near);
BSP Trees: Viewpoint A

- decide independently at each tree vertex
- not just left or right child!
BSP Trees : Viewpoint A
BSP Trees: Viewpoint A
BSP Trees : Viewpoint A
BSP Trees: Viewpoint A
BSP Trees: Viewpoint A
BSP Trees: Viewpoint A
BSP Trees: Viewpoint A
BSP Trees: Viewpoint A
BSP Trees: Viewpoint A
BSP Trees : Viewpoint A
BSP Trees : Viewpoint B
BSP Trees: Viewpoint B
BSP Tree Traversal: Polygons

- split along the plane defined by any polygon from scene
- classify all polygons into positive or negative half-space of the plane
  - if a polygon intersects plane, split polygon into two and classify them both
- recurse down the negative half-space
- recurse down the positive half-space
BSP Demo

- useful demo:
  [http://symbolcraft.com/graphics/bsp](http://symbolcraft.com/graphics/bsp)
Summary: BSP Trees

- **pros:**
  - simple, elegant scheme
  - correct version of painter’s algorithm back-to-front rendering approach
  - was very popular for video games (but getting less so)

- **cons:**
  - slow to construct tree: $O(n \log n)$ to split, sort
  - splitting increases polygon count: $O(n^2)$ worst-case
  - computationally intense preprocessing stage restricts algorithm to static scenes
Warnock’s Algorithm (1969)

- based on a powerful general approach common in graphics
  - if the situation is too complex, *subdivide*

- BSP trees was object space approach
- Warnock is image space approach
Warnock’s Algorithm

- start with root viewport and list of all objects
- recursion:
  - clip objects to viewport
  - if only 0 or 1 objects
    - done
  - else
    - subdivide to new smaller viewports
    - distribute objects to new viewpoints
    - recurse
Warnock’s Algorithm

- termination
  - viewport is single pixel
  - explicitly check for object occlusion
Warnock’s Algorithm

pros:
- very elegant scheme
- extends to any primitive type

cons:
- hard to embed hierarchical schemes in hardware
- complex scenes usually have small polygons and high depth complexity (number of polygons that overlap a single pixel)
  - thus most screen regions come down to the single-pixel case
The Z-Buffer Algorithm (mid-70’s)

- both BSP trees and Warnock’s algorithm were proposed when memory was expensive
  - first 512x512 framebuffer was >$50,000!
- Ed Catmull proposed a radical new approach called **z-buffering**.
- the big idea:
  - resolve visibility **independently at each pixel**
The Z-Buffer Algorithm

- we know how to rasterize polygons into an image discretized into pixels:
The Z-Buffer Algorithm

- what happens if multiple primitives occupy the same pixel on the screen?
- which is allowed to paint the pixel?
The Z-Buffer Algorithm

- idea: retain depth after projection transform
  - each vertex maintains z coordinate
    - relative to eye point
  - can do this with canonical viewing volumes
The 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 distant than the Z value already stored there
Interpolating Z

- edge equations: Z just another planar parameter:
  - \( z = (-D - Ax - By) / C \)
  - if walking across scanline by \( (D_x) \)
    \( z_{new} = z_{old} - (A/C)(D_x) \)

- total cost:
  - 1 more parameter to increment in inner loop
  - 3x3 matrix multiply for setup
Interpolating Z

- edge walking
  - just interpolate Z along edges and across spans
- barycentric coordinates
  - interpolate Z like other parameters
Z-Buffer

- store \((r, g, b, z)\) for each pixel
- typically 8+8+8+24 bits, can be more

\[
\begin{align*}
\text{for all } i, j \{ \\
\text{Depth}[i, j] &= \text{MAX DEPTH} \\
\text{Image}[i, j] &= \text{BACKGROUND COLOUR} \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{for all polygons } P \{ \\
\text{for all pixels in } P \{ \\
\text{if } (\text{Z\_pixel} < \text{Depth}[i, j]) \{ \\
\text{Image}[i, j] &= \text{C\_pixel} \\
\text{Depth}[i, j] &= \text{Z\_pixel} \\
\}
\}
\end{align*}
\]
Depth Test Precision

- reminder: projective transformation maps eye-space $z$ to generic $z$-range (NDC)
- simple example:

$$
T \begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix} = \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & a & b \\
  0 & 0 & -1 & 0
\end{bmatrix} \begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
$$

- thus:

$$
z_{NDC} = \frac{a \cdot z_{eye} + b}{z_{eye}} = a + \frac{b}{z_{eye}}$$
Depth Test Precision

- therefore, depth-buffer essentially stores $1/z$, rather than $z$!
- issue with integer depth buffers
  - high precision for near objects
  - low precision for far objects
Depth Test Precision

- low precision can lead to **depth fighting** for far objects
  - two different depths in eye space get mapped to same depth in framebuffer
  - which object “wins” depends on drawing order and scan-conversion
- gets worse for larger ratios \( f:n \)
  - *rule of thumb*: \( f:n < 1000 \) for 24 bit depth buffer
- with 16 bits cannot discern millimeter differences in objects at 1 km distance
Z-Buffer Algorithm Questions

- how much memory does the Z-buffer use?
- does the image rendered depend on the drawing order?
- does the time to render the image depend on the drawing order?
- how does Z-buffer load scale with visible polygons? with framebuffer resolution?
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 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
  - Warnock: check up to single pixels if needed
- performed late in rendering pipeline
Projective Rendering Pipeline

- **OCS** - object coordinate system
- **WCS** - world coordinate system
- **VCS** - viewing coordinate system
- **CCS** - clipping coordinate system
- **NDCS** - normalized device coordinate system
- **DCS** - device coordinate system

**glVertex3f(x,y,z)**

**modeling transformation**

**glTranslatef(x,y,z)**

**glRotatef(th,x,y,z)**

**gluLookAt(...)**

**viewing transformation**

**gluLookAt(...)**

**projection transformation**

**glFrustum(...)**

**viewport transformation**

**glutInitWindowSize(w,h)**

**glViewport(x,y,a,b)**

**alter w**

**/ w**

**perspective division**

**clipping CCS**

**normalized device NDCS**

**device DCS**
Rendering Pipeline

Geometry Database → Model/View Transform. → Lighting → Perspective Transform. → Clipping

Clipping (4D)

object world viewing

OCS WCS VCS

normalized
device
NDCS

device
DCS

(3D)

Scan Conversion → Texturing → Depth Test → Blending → Frame-buffer

screen

SCS

(2D)
Backface Culling
Back-Face Culling

- on the surface of a closed orientable manifold, polygons whose normals point away from the camera are always occluded:

note: backface culling alone doesn’t solve the hidden-surface problem!
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
Back-Face Culling

- most objects in scene are typically “solid”
- rigorously: orientable closed manifolds
  - orientable: must have two distinct sides
    - cannot self-intersect
    - a sphere is orientable since has two sides, 'inside' and 'outside'.
    - a Mobius strip or a Klein bottle is not orientable
  - closed: cannot “walk” from one side to the other
    - sphere is closed manifold
    - plane is not
Back-Face Culling

- most objects in scene are typically “solid”
- rigorously: orientable closed manifolds
  - manifold: local neighborhood of all points isomorphic to disc
  - boundary partitions space into interior & exterior
Manifold

- examples of manifold objects:
  - sphere
  - torus
  - well-formed CAD part
Back-Face Culling

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

first idea:
cull if $N_z < 0$

sometimes misses polygons that should be culled

better idea:
cull if eye is below polygon plane
Back-face Culling: NDCS

works to cull if $N_Z > 0$