Texture Mapping and Sampling

CPSC 314

The Rendering Pipeline
Scan Conversion of Lines - Digital Differential Analyzer

First Attempt:

```c
dda( float xs, ys, xe, ye ) {
    // assume xs < xe, and slope m between 0 and 1
    float m= (ye-ys)/(xe-xs);
    float y= round( ys );
    for( int x= round( xs ) ; x<= xe ; x++ ) {
        drawPixel( x, round( y ) );
        y= y+m;
    }
}
```

Scan Conversion of Lines

DDA:
Scan Conversion of Lines

Midpoint Algorithm

*Moving horizontally along x direction*
- Draw at current y value, or move up vertically to y+1?
  - Check if midpoint between two possible pixel centers above or below line

*Candidates*
- Top pixel: (x+1, y+1)
- Bottom pixel: (x+1, y)

*Midpoint: (x+1, y+.5)*

*Check if midpoint above or below line*
- Below: top pixel
- Above: bottom pixel

*Key idea behind Bresenham*

---

**Scan Conversion of Lines**

*Idea: decision variable*

```c
dda( float xs, ys, xe, ye ) {
    float d= 0.0;
    float m= (ye-ys)/(xe-xs);
    int y= round( ys );
    for( int x= round( xs ) ; x<= xe ; x++ ) {
        drawPixel( x, y );
        d= d+m;
        if( d>= 0.5 ) { d= d-1.0; y++; }
    }
}
```

© Wolfgang Heidrich
Scan Conversion of Lines
Bresenham Algorithm (’63)

• Use decision variable to generate purely integer algorithm
• Explicit line equation:
  \[ y = \frac{(y_e - y_s)}{(x_e - x_s)}(x - x_s) + y_s \]
• Implicit version:
  \[ L(x, y) = \frac{(y_e - y_s)}{(x_e - x_s)}(x - x_s) - (y - y_s) = 0 \]
• In particular for specific \( x, y \), we have
  – \( L(x, y) > 0 \) if \( (x, y) \) below the line, and
  – \( L(x, y) < 0 \) if \( (x, y) \) above the line

Scan Conversion of Lines
Bresenham Algorithm

• Decision variable: after drawing point \((x, y)\) decide whether to draw
  – \((x+1, y)\): case \( E \) (for “east”)
  – \((x+1, y+1)\): case \( NE \) (for “north-east”)
• Check whether \((x+1, y+1/2)\) is above or below line
  \[ d = L(x+1, y + \frac{1}{2}) \]
• Point above line if and only if \( d < 0 \)
Scan Conversion of Lines

**Bresenham Algorithm**

- Problem: how to update \( d \)?
  - Case E (point above line, \( d \leq 0 \))
    - \( x = x + 1 \);
    - \( d = L(x + 2, y + 1/2) = d + (y_e - y_s)/(x_e - x_s) \)
  - Case NE (point below line, \( d > 0 \))
    - \( x = x + 1; y = y + 1 \);
    - \( d = L(x + 2, y + 3/2) = d + (y_e - y_s)/(x_e - x_s) - 1 \)
- Initialization:
  - \( d = L(x_s + 1, y_s + 1/2) = (y_e - y_s)/(x_e - x_s) - 1/2 \)

---

**Scan Conversion of Lines**

**Bresenham Algorithm**

```c
Bresenham( int xs, ys, xe, ye ) {
    int y = ys;
    incrE = 2(ye - ys);
    incrNE = 2((ye - ys) - (xe - xs));
    for( int x = xs ; x <= xe ; x++ ) {
        drawPixel( x, y );
        if( d <= 0 ) d += incrE;
        else { d += incrNE; y++; }
    }
}
```
Scan Conversion of Polygons

One possible scan conversion
Scan Conversion of Polygons

**A General Algorithm**
- Intersect each scanline with all edges
- Sort intersections in x
- Calculate parity to determine in/out
- Fill the ‘in’ pixels

Edge Walking

**Past graphics hardware**
- Exploit continuous L and R edges on trapezoid

\[
\text{scanTrapezoid}(x_L, x_R, y_B, y_T, \Delta x_L, \Delta x_R)
\]
Edge Walking Triangles

- Split triangles into two regions with continuous left and right edges

\[
\text{scanTrapezoid}(x_3, m_3, y_3, y_m, \frac{1}{m_{13}}, \frac{1}{m_{12}})
\]

\[
\text{scanTrapezoid}(x_2, x_m, y_2, y_3, \frac{1}{m_{23}}, \frac{1}{m_{12}})
\]

Modern Rasterization

Define a triangle as follows:
Using Edge Equations

**Implicit equation of a triangle edge:**

\[ L(x, y) = \left( \frac{y_e - y_s}{x_e - x_s} \right) (x - x_s) - (y - y_s) = 0 \]

(see Bresenham algorithm)

- \( L(x,y) \) positive on one side of edge, negative on the other

**Question:**

- How do we know which side is in, and which side out?
  - And how do we make the \( L(x,y) \) positive for points inside?
Computing Edge Equations

Assumption:
• Triangle vertices given in counter-clockwise order

Then:
• If \( x_s < x_e \), then
  – Use \(-L(x,y)\) as edge equation
• Else
  – Use \(+L(x,y)\) as edge equation

Edge Equations: Code

```java
findBoundingBox(&xmin, &xmax, &ymin, &ymax);
setupEdges (&a0,&b0,&c0,&a1,&b1,&c1,&a2,&b2,&c2);

for (int y = yMin; y <= yMax; y++) {
    for (int x = xMin; x <= xMax; x++) {
        float e0 = a0*x + b0*y + c0;
        float e1 = a1*x + b1*y + c1;
        float e2 = a2*x + b2*y + c2;
        if (e0 > 0 && e1 > 0 && e2 > 0)
            Image[x][y] = TriangleColor;
    }
}
```
Triangle Rasterization Issues

Exactly which pixels should be lit?
A: Those pixels inside the triangle edges
What about pixels exactly on the edge?
  • Draw them: order of triangles matters (it shouldn’t)
  • Don’t draw them: gaps possible between triangles

We need a consistent (if arbitrary) rule
  • Example: draw pixels on left or top edge, but not on right or bottom edge

Triangle Rasterization Issues

Shared Edge Ordering
Value Interpolation: Plane Equation

Quantities vary linearly across image plane
- E.g.: \( r = Ax + By + C \)
  - \( r = \) red channel of the color
  - Same for \( g, b, Nx, Ny, Nz, z \)
- From info at vertices we know:
  \[
  r_1 = Ax_1 + By_1 + C \\
  r_2 = Ax_2 + By_2 + C \\
  r_3 = Ax_3 + By_3 + C
  \]
- Solve for \( A, B, C \)
- One-time set-up cost per triangle

Discussion

On old hardware:
- Use first scan-conversion algorithm
  - Scan-convert edges, then fill in scanlines
  - Compute interpolated values by interpolating along edges, then scanlines
- Requires clipping of polygons against viewing volume
- Faster if you have a few, large polygons
- Possibly faster in software
Discussion

Modern GPUs:
- Use edge equations
  - And plane equations for interpolation
  - No clipping of primitives required
- Faster with many small triangles

Additional advantage:
- Can control the order in which pixels are processed
- Allows for more memory-coherent traversal orders
  - E.g. space-filling curve rather than scanlines

The Rendering Pipeline
Texture Mapping

- Real life objects have nonuniform colors, normals
- To generate realistic objects, reproduce coloring & normal variations = \textit{texture}
- Can often replace complex geometric details

\textit{Introduced to increase realism}

\textit{Hide geometric simplicity}

- Images convey illusion of geometry
- Map a brick wall texture on a flat polygon
- Create bumpy effect on surface

\textit{Associate 2D information with 3D surface}

- Point on surface corresponds to a point in texture
- “Paint” image onto polygon
**Color Texture Mapping**

*Define color (RGB) for each point on a surface*

*Two approaches*
- Surface texture map
- Volumetric texture

---

**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)
```
Texture Mapping Example

Example Texture Map

```c
glTexCoord2d(0, 0);
glVertex3d (0, -2, -2);

glTexCoord2d(1, 1);
glVertex3d (0, 2, 2);
```
Fractional Texture Coordinates

Texture image

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

Texture Lookup: Tiling and Clamping

What if s or t is outside the interval [0...1]?

Multiple choices

- Use fractional part of texture coordinates
  - Cyclic repetition of texture to tile whole surface
    \[
    \text{glTexParameteri}( ..., \text{GL_TEXTURE_WRAP_S, GL_REPEAT, GL_TEXTURE_WRAP_T, GL_REPEAT, ... })
    \]

- Clamp every component to range [0...1]
  - Re-use color values from texture image border
    \[
    \text{glTexParameteri}( ..., \text{GL_TEXTURE_WRAP_S, GL_CLAMP, GL_TEXTURE_WRAP_T, GL_CLAMP, ... })
    \]
Tiled Texture Map

\( \text{glTexCoord2d}(1, 1); \)
\( \text{glVertex3d} (x, y, z); \)

\( \text{glTexCoord2d}(4, 4); \)
\( \text{glVertex3d} (x, y, z); \)

Texture Coordinate Transformation

**Motivation**
- Change scale, orientation of texture on an object

**Approach**
- Texture matrix stack
- Transforms specified (or generated) tex coords
  
  \[
  \text{glMatrixMode}( \text{GL_TEXTURE} ); \\
  \text{glLoadIdentity}(); \\
  \text{glRotate}(); \\
  \ldots
  \]
- More flexible than changing \((s,t)\) coordinates
Texture Functions

*Once you 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**
  - Like replace, but supports texture transparency
- Blend surface color with another: **GL_BLEND**
  - New value controls which of 2 colors to use

*Specify desired behavior with glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, <mode>)*

---

Texture Pipeline

- **Object position** (-2.3, 7.1, 17.7)
- **Parameter space** (0.32, 0.29)
- **Transformed parameter space** (0.52, 0.49)
- **Texel space** (81, 74)
- **Texel color** (0.9, 0.8, 0.7)
- **Final color** (0.45, 0.4, 0.35)
- **Object color** (0.5, 0.5, 0.5)
Texture Objects and Binding

**Texture object**
- An OpenGL data type that keeps textures resident in memory and provides identifiers to easily access them
- Provides efficiency gains over having to repeatedly load and reload a texture
- You can prioritize textures to keep in memory
- OpenGL uses least recently used (LRU) if no priority is assigned

**Texture binding**
- Which texture to use right now
- Switch between preloaded textures

Basic OpenGL Texturing

*Create a texture object and fill it with texture data:*
- `glGenTextures(num, &indices)` to get identifiers for the objects
- `glBindTexture(GL_TEXTURE_2D, identifier)` to bind
  - *Following texture commands refer to the bound texture*
- `glTexParameter*(GL_TEXTURE_2D, ..., ...)` to specify parameters for use when applying the texture
- `glTexImage2D(GL_TEXTURE_2D, ..., ...)` to specify the texture data (the image itself)
Basic OpenGL Texturing (cont.)

Enable texturing:
- `glEnable(GL_TEXTURE_2D)`

State how the texture will be used:
- `glTexEnvf(...)`

Specify texture coordinates for the polygon:
- Use `glTexCoord2f(s,t)` before each vertex:
  - `glTexCoord2f(0,0);`
  - `glVertex3f(x,y,z);`

Low-Level Details

Large range of functions for controlling layout of texture data
- State how the data in your image is arranged
- e.g.: `glPixelStorei(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 have a size of power of 2
- Common sizes are 32x32, 64x64, 256x256
- But don’t need to be square, i.e. 32x64 is fine
- Smaller uses less memory, and there is a finite amount of texture memory on graphics cards
Texture Mapping

Texture coordinate interpolation

- Perspective foreshortening problem

Interpolation: Screen vs. World Space

Screen space interpolation incorrect

- Problem ignored with shading, but artifacts more visible with texturing
Texture Coordinate Interpolation

*Perspective correct interpolation*

- $\alpha$, $\beta$, $\gamma$:
  - Barycentric coordinates of a point $P$ in a triangle
- $s0$, $s1$, $s2$:
  - Texture coordinates of vertices
- $w0$, $w1$, $w2$:
  - Homogeneous coordinates of vertices

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

\[
\frac{\alpha}{w_0} + \frac{\beta}{w_1} + \frac{\gamma}{w_2}
\]

Reconstruction

(Image courtesy of Kiriakos Kutulakos, U Rochester)
Reconstruction

- How to deal with:
  - *Pixels that are much larger than texels?*
    - Apply filtering, “averaging”
  - *Pixels that are much smaller than texels?*
    - Interpolate

Interpolating Textures

- Nearest neighbor
- Bilinear
- Hermite
**MIPmapping**

use “image pyramid” to precompute averaged versions of the texture

store whole pyramid in single block of memory

**MIPmaps**

*Multum in parvo -- many things in a small place*

- Prespecify a series of prefILTERed texture maps of decreasing resolutions
- Requires more texture storage
- Avoid shimmering and flashing as objects move

**gluBuild2DMipmaps**

- Automatically constructs a family of textures from original texture size down to 1x1 without with
MIPmap storage

*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

Original surface

A bump map
Bump Mapping

\[ O'(u) \]
Lengthening or shortening \( O(u) \) using \( B(u) \)

\[ N'(u) \]
The vectors to the ‘new’ surface

Displacement Mapping

**Bump mapping gets silhouettes wrong**
- Shadows wrong too

**Change surface geometry instead**
- Need to subdivide surface

**GPU support**
- Bump and displacement mapping not directly supported: require per-pixel lighting
- However: modern GPUs allow for programming both yourself
Environment Mapping

*Cheap way to achieve reflective effect*

- Generate image of surrounding
- Map back into a sphere

Sphere Mapping

*Texture is distorted fish-eye view*

- Point camera at mirrored sphere
- Spherical texture mapping creates texture coordinates that correctly index into this texture map
Cube Mapping

6 planar textures, sides of cube
- Point camera in 6 different directions, facing out from origin
Cube Mapping

**Direction of reflection vector r selects the face of the cube to be indexed**

- Co-ordinate with largest magnitude
  - e.g., the vector (-0.2, 0.5, -0.84) selects the –Z face

- Remaining two coordinates (normalized by the 3rd coordinate) selects the pixel from the face.
  - E.g., (-0.2, 0.5) gets mapped to (0.38, 0.80).

*Difficulty in interpolating across faces*

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

*Common for natural material/irregular textures (stone, wood, etc...)*
Volumetric Bump Mapping

Marble

Bump

Procedural Textures

*Generate “image” on the fly, instead of loading from disk*

- Often saves space
- Allows arbitrary level of detail
Procedural Textures

Several good explanations

- Text book Section 10.1
- http://www.noisemachine.com/talk1
- http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

http://mrd.nyu.edu/~perlin/planet/

Sampling

CPSC 314
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

![Weighting function graph](image)

**Images**

*An image is a 2D function \( I(x, y) \)**

- Specifies intensity for each point \((x, y)\)
- (we consider each color channel independently)
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

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 Geometry

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

---

**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**
- Wavelength – length of repeated sequence on infinite signal
- Frequency – 1/wavelength (number of repeated sequences in unit length)

**Example – sine wave**
- Wavelength = 2\(\pi\)
- Frequency = 1/2\(\pi\)

\[
\sin(t)
\]

Summing Waves I

\[
\begin{align*}
\sin(x) + \frac{\sin(3x)}{3} + \frac{\sin(5x)}{5} + \frac{\sin(7x)}{7} + \frac{\sin(9x)}{9} &= \frac{\sin(11x)}{11} \\
0 & \quad 2\pi & 4\pi & 6\pi & 8\pi & 10\pi & 0 & 2\pi & 4\pi & 6\pi & 8\pi & 10\pi
\end{align*}
\]
Summing Waves II

Summing Waves II represents 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

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**

- Sampling rate greater than twice highest 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: anti-aliasing*

- Supersample
  - Sample at higher frequency
- Low pass filtering
  - Remove high frequency function parts
  - Aka prefiltering, band-limiting
Supersampling

No antialiasing

3x3 supersampling
3x3 unweighted filter

Low-Pass Filtering

Original signal

Low-pass filtering

Low-pass filtered signal
Low-Pass Filtering

![Diagram showing sampled signal and reconstruction process.](image)

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

Previous Antialiasing Example

*Texture mipmapping: low pass filter*

![Texture mipmapping images.](image)

(a)  (b)
Coming Up...

**Wednesday:**
- Modern GPU Features

**Friday:**
- Miscellaneous topics, Rendering Pipeline wrap-up
- Quiz 2