Scan Conversion of Lines

Raster devices

Most devices that are used to produce images are raster devices, that is, use rectangular arrays of dots (pixels) to display the image. This includes CRT monitors, LCDs, laser and dot-matrix printers.

Examples of non-raster output devices include vector displays (not used anymore) and plotters still widely used.

Scan conversion = converting a continuous object such as a line or a circle into discrete pixels
Scan conversion of lines

Given two points with integer coordinates $p_1 = [x_1, y_1]$ and $p_2 = [x_2, y_2]$ the algorithm has to find a sequence of pixels approximating the line.

Slope: $(y_2 - y_1)/(x_2 - x_1)$

We can always reorder $p_1$ and $p_2$ so that $x_2 - x_1$ is nonnegative. It is convenient to look at only nonnegative slopes; if the slope is negative, change the sign of $y$.

Slope

Slope reduction: it is convenient to have the slope of the line between 0 and 1; then we are able to step along $x$ axis.

slope $> 1$, cannot step along $x$  
slope $< 1$, can step along $x$

To handle slope $> 1$, swap $x$ and $y$
**DDA**

Assume that the slope is between 0 and 1

Simplest algorithm (pompously called differential digital analyzer):

Step along x, increment y by slope at each step.

Round y to nearest pixel.

```plaintext
float y = y1;
float slope = (y2-y1)/(float)(x2-x1);
int x;
for(x = x1; x <= x2; x++) {
    drawpixel(x, floor(y));
    y += slope;
}
```

---

**Bresenham Algorithm**

What is wrong with DDA?

It requires floating-point operations.

These operations are expensive to implement in hardware.

They are not really necessary if the endpoints are integers.

Idea: instead of incrementing y and rounding it at each step, decide if we just go to the right, or to the right and up using only integer quantities.
Increment decision

- Pixel corners on different sides of the line: increment both x and y
- Pixel corners on the same side of the line: increment only x

Need: fast way to determine on which side of a line a point is.

Half-plane test

Implicit equation can be used to perform the test.

\[(n \cdot (q - p)) > 0\]  \(\rightarrow\) the point on the same side with the normal

\[(n \cdot (q - p)) < 0\]  \(\rightarrow\) the point on the other side
Implicit line equation

The implicit equation of the line through \( p_1 = [x_1, y_1] \) and \( p_2 = [x_2, y_2] \) is
\[
(n, q - p_1) = 0, \text{ with } n = [y_2, -y_1, x_1, -x_2]
\]

We need to test on which side of the line is the point
\( q+d_1 = [x, y] + [1/2, 1/2] \)

To do this, we need to determine the sign of
\[
F = (n, 2q + 2d_1 - 2p_1)
\]
Note that multiplication by two makes everything integer again!
Key idea: compute this quantity incrementally.

Incremental computation

At each step \( q = [x, y] \) changes either to \( [x+1, y] \) (step to the right) or to \( [x+1, y+1] \) (step to the right and up); in vector form, the new value of \( q \) is

either \( q + D_1 \) or \( q + D_2 \), with \( D_1 = [1, 0] \) and \( D_2 = [1, 1] \)

\[
F_{\text{next}} = (n, 2q + 2D + 2d_1 - 2p_1) = (n, 2q + 2d_1 - 2p_1) + 2(n, D)
\]

\( = F + 2(n, D) \), where \( D \) is \( D_1 \) or \( D_2 \)

At each step, to get new \( F \) we have to increment old \( F \) either by \( (n, D_1) \) or \( (n, D_2) \)

\[
(n, D_1) = y_2 - y_1
\]

\[
(n, D_2) = (y_2 - y_1) - (x_2 - x_1)
\]
Bresenham algorithm

Assume the slope to be between 0 and 1.

```c
int y = y1; int dy = y2-y1;
int dxdy = y2-y1+x1-x2;
int F = y2-y1+x1-x2; int x;
for( x = x1; x <= x2; x++ ) {
    drawpixel(x,y);
    if( F < 0 ) {
        F += dy;
    } else {
        y++; F+= dxdy;
    }
}
```

Bresenham algorithm

In your implementation you need to handle all slopes!
First, reorder endpoints so that x₁, <= x₂

Then consider 4 cases:

- \( y₂ - y₁ \geq 0, \ x₂ - x₁ \geq 0 \) \( y₂ - y₁ \) positive slope <= 1
- \( y₂ - y₁ \geq 0, \ x₂ - x₁ < 0 \) \( y₂ - y₁ \) positive slope > 1
- \( y₂ - y₁ < 0, \ x₂ - x₁ \geq 0 \) \( y₁ - y₂ \) negative slope >= -1
- \( y₂ - y₁ < 0, \ x₂ - x₁ < 0 \) \( y₁ - y₂ \) negative slope < -1

In each case, make appropriate substitutions in the algorithm.
Scan converting polygons

Polygons

- convex
- non-convex
- with holes
- with self-intersections

We focus on the convex case
Scan Conversion of Convex Polygons

General idea:
- decompose polygon into tiles
- scan convert each tile, moving along one edge

Convex Polygons

Scan convert a convex polygon:

```c
void ScanY( Vertex2D v[], int num_vertices, int bottom_index)
```

1. Find left edge of a tile:
   - go around **clockwise**, starting from v[bot], until find an edge such that it is not contained inside a scan line:

2. Similarly, find the right edge of a tile.
3. Scan convert all scan lines going from left to right edges
void ScanY( Vertex2D v[], int num_vertices, int bottom_index) {
    Initialize variables
    remaining_vertices = num_vertices;
    while(remaining_vertices > 0) {
        Find the left top row candidate
        Determine the slope and starting x location for the left tile edge
        Find the right top row candidate
        Determine the slope and starting x location for the right tile edge
        for(row = bottom_row; row < left_top_row &&
            row < right_top_row; row++) {
            ScanX(ceil(left_pos),ceil(right_pos),row);
            left_pos += left_step;
            right_pos += right_step;
        }
        bottom_row = row;
    }
}

Initialization

Keep track of the numbers of the vertices on the left and on the right:
    int left_edge_end = bottom_index;
    int right_edge_end= bottom_index;

This is the first row of a tile:
    int bottom_row = ceil(v[bottom_index].y);

These are used to store the candidates for the top row of a tile:
    int left_top_row = bottom_row;
    int right_top_row = bottom_row;

Keep track of the intersections of left and right edges of a tile with
    horizontal integer lines:
    float left_pos, right_pos, left_step, right_step;

Number of remaining vertices:
    int remaining_vertices;

A couple of auxilary variables: int edge_start; int row;
Find a tile

Compute increment in y direction and starting/ending (left/right) point for the first scan of a tile

Find the left top row candidate

while( left_top_row <= bottom_row && remaining_vertices > 0)
{
  Move to next edge:
  edge_start = left_edge_end;
  Be careful with C % operator, (N-1) % M will give -1 for N = 0, need to use (N+M-1) % M to get (N-1) mod M = N-1
  left_edge_end = (left_edge_end+num_vertices-1)%num_vertices;
  left_top_row = ceil(v[left_edge_end].y);
  remaining_vertices--;

  We found the first edge that sticks out over bottom_row
  determine the slope and starting x location for the left tile edge.
  if(left_top_row > bottom_row )
  {
    left_step = (v[left_edge_end].x - v[edge_start].x)/
    (v[left_edge_end].y - v[edge_start].y);
    left_pos = v[edge_start].x +
    (bottom_row-v[edge_start].y)*left_step;
  }
}
Find a tile

Find the right top row candidate; determine the slope and starting x location for the right tile edge. Exactly as for the left edge.

Scan convert a single row:

```c
void ScanX(int left_col, int right_col, int row, int R, int G, int B) {
    if( left_col < right_col) {
        for( int x = left_col; x < right_col; x++) {
            draw_pixel(x,y);
        }
    }
}
```

Texture mapping

Texture slides are based on E. Angel’s slides

```
\begin{itemize}
\item y
\item x
\item geometry
\item screen
\item image
\end{itemize}
```
**Sampling texture maps**

The back row is a very poor representation of the true image.

**Texture Example**

The texture (below) is a 256 x 256 image that has been mapped to a rectangular polygon which is viewed in perspective.
Applying Textures I

Three steps

1. specify texture
   - read or generate image
   - assign to texture

2. assign texture coordinates to vertices

3. specify texture parameters
   - wrapping, filtering

Applying Textures II

- specify textures in texture objects
- set texture filter
- set texture function
- set texture wrap mode
- set optional perspective correction hint
- bind texture object
- enable texturing
- supply texture coordinates for vertex
  - coordinates can also be generated
Texture Objects

Like display lists for texture images
- one image per texture object
- may be shared by several graphics contexts

Generate texture names

```c
glGenTextures( n, *texIds );
```

Bind textures before using

```c
glBindTexture( target, id );
```

Specify Texture Image

Define a texture image from an array of texels in CPU memory

```c
glTexImage2D( target, level, components, w, h, border, format, type, *texels );
```

- dimensions of image must be powers of 2

Texel colors are processed by pixel pipeline
- pixel scales, biases and lookups can be done
Converting A Texture Image

If dimensions of image are not power of 2

\[
\text{gluScaleImage}( \text{format}, \text{w}_\text{in}, \text{h}_\text{in}, \\
\quad \text{type}_\text{in}, *\text{data}_\text{in}, \text{w}_\text{out}, \text{h}_\text{out}, \\
\quad \text{type}_\text{out}, -*\text{data}_\text{out} );
\]

- *\text{in} is for source image
- *\text{out} is for destination image

Image interpolated and filtered during scaling

Specifying a Texture:
Other Methods

Use frame buffer as source of texture image
- uses current buffer as source image

\[
\text{glCopyTexImage2D}(...) \\
\text{glCopyTexImage1D}(...)
\]

Modify part of a defined texture

\[
\text{glTexSubImage2D}(...) \\
\text{glTexSubImage1D}(...)
\]

Do both with \text{glCopyTexSubImage2D}(...), etc.
Mapping a Texture

Based on parametric texture coordinates
\texttt{glTexCoord*()} specified at each vertex

Generating Texture Coordinates

Automatically generate texture coords
\texttt{glTexGen{ifd}[v]()}

specify a plane
- generate texture coordinates based upon distance from plane
generation modes
\[ Ax + By + Cz + D = 0 \]
- \texttt{GL_OBJECT_LINEAR}
- \texttt{GL_EYE_LINEAR}
- \texttt{GL_SPHERE_MAP}
Texture Application Methods

Filter Modes
- minification or magnification
- special mipmap minification filters

Wrap Modes
- clamping or repeating

Texture Functions
- how to mix primitive’s color with texture’s color
  - blend, modulate or replace texels

Example:
```c
glTexParameteri( target, type, mode );
```

Filter Modes

![Texture Polygon Magnification](image1)
![Texture Polygon Minification](image2)
Mipmapped Textures

Mipmap allows for prefiltered texture maps of decreasing resolutions
Lessens interpolation errors for smaller textured objects
Declare mipmap level during texture definition

```
  glTexImage*D( GL_TEXTURE_*D, level, ... )
```

GLU mipmap builder routines

```
gluBuild*DMipmaps( ... )
```

OpenGL 1.2 introduces advanced LOD controls

Wrapping Mode

Example:
```
  glTexParameteri( GL_TEXTURE_2D,
                 GL_TEXTURE_WRAP_S, GL_CLAMP )
  glTexParameteri( GL_TEXTURE_2D,
                 GL_TEXTURE_WRAP_T, GL_REPEAT )
```

© 2005, Denis Zorin
Texture Functions

Controls how texture is applied

\[ \text{glTexEnv}[f][v]( \text{GL_TEXTURE_ENV, prop, param } ) \]

\textit{GL_TEXTURE_ENV_MODE} modes

- \text{GL_MODULATE}
- \text{GL_BLEND}
- \text{GL_REPLACE}

Set blend color with \textit{GL_TEXTURE_ENV_COLOR}

Perspective Correction Hint

Texture coordinate and color interpolation

- either linearly in screen space
- or using depth/perspective values (slower)

Noticeable for polygons “on edge”

\[ \text{glHint( GL_PERSPECTIVE_CORRECTION_HINT, hint )} \]

where \textit{hint} is one of

- \text{GL_DONT_CARE}
- \text{GL_NICEST}
- \text{GL_FASTEST}
Bump Mapping

Bump mapped normals are inconsistent with actual geometry. Problems arise (shadows). Displacement mapping actually affects the surface geometry.
Mipmaps

*multum in parvo* — *many things in a small place*

A texture LOD technique

Prespecify a series of prefiltered texture maps of decreasing resolutions

Requires more texture storage

Eliminates shimmering and flashing as objects move

MIPMAPS

Arrange different versions into one block of memory
MIPMAPS

With versus without MIPMAP