Basics in Computer Graphics

egon elbre
Why?

It's fun (but we're not going to talk about that part)
Making things faster
Balancing speed and quality
Getting things to the screen
Additional Reading
anyway...
this is an image
these are pixels
typedef struct {
    int width; int height;
    float gamma;
    Pixel[height*width] pixels;
} Bitmap;

typedef struct {
    unsigned char R;
    unsigned char G;
    unsigned char B;
    unsigned char A;
} Pixel;
other pixel formats

R8,G8,B8,A8
R8,G8,B8
R5,G6,B5,A4
R5,G5,B5,A1
B8,G8,R8,A8

R16,G16,B16,A16
R7,G9,B6,A9
...

**set/get pixel**

typedef struct {
    int width; int height;
    float gamma;
    Pixel[height*width] pixels;
} Bitmap;

set(int x, int y, Pixel v) {
    pixels[y*width + x] = v;
}

Pixel get(int x, int y) {
    return pixels[y*width + x];
}
alternative bitmap

typedef struct {
    int width; int height;
    float gamma;
    Pixel[height][width] pixels;
} Bitmap;
typedef struct {
    int width; int height;
    float gamma;
    Pixel[height][width] pixels;
} Bitmap;

set(int x, int y, Pixel v) {
    pixels[y][x] = v;
}

Pixel get(int x, int y) {
    return pixels[y][x];
}
also the monitor
little bit closer
gamma correction

gamma correction $1/2.2$

CRT gamma 2.2
meaning IRL

http://www.4p8.com/eric.brasseur/gamma.html
Scan Conversion / Rasterization
let's learn to draw lines
// how code is written on slides
hline(int x0, int x1, int y, Pixel v) {
    int x;
    for(x = x0; x <= x1; ++x)
        put(x, y, v);
}

// how it should be implemented
hline(int x0, int x1, int y, Pixel v) {
    Pixel *p = &pixels[y*width + x0];
    Pixel *p_last = &pixels[y*width + x1];
    for(; p <= p_last; ++p)
        *p = v;
}
THE LINE

- parameters:
  - \((x_0, y_0, x_1, y_1)\)
- \(y = mx + b\)
- \(m = \frac{(y_1 - y_0)}{(x_1 - x_0)}\)
- assuming
  - \(|m| < 1\)
  - \(x_0 < x_1\)
brute-force calculation

```c
line(int x0, int y0,
    int x1, int y1,
    Pixel v){
    int x;
    float m, y;
    m = (y1 - y0) / (x1 - x0);
    for (x = x0; x <= x1; ++x) {
        y = m * (x - x0) + y0;
        set(x, round(y), v);
    }
}
```
can we do any better?
incremental line

```c
line(int x0, int y0,
    int x1, int y1,
    Pixel v){
    int x;
    float m, y;
    m = (y1 - y0) / (x1 - x0);
    y = y0
    for (x = x0; x <= x1; ++x) {
        set(x, round(y), v);
        y += m;
    }
}
```
now the circle
THE CIRCLE

- parameters:
  - $(x_0, y_0, R)$
- $y = R \sin(\theta)$, $x = R \cos(\theta)$
- $y = \pm \sqrt{R^2 - x^2}$

assuming
- $x_0 = 0$
- $y_0 = 0$
phase circle

void circle(int R, Pixel v) {
    float a, d;
    d = 1 / R;
    for (a = 0; a <= TAU; a += d) {
        set8(R * cos(a), R * sin(a), v);
    }
}

// arcsin( b/R ) ≈ b/R, iff b«R

// sin, cos are slow !!!
8 fold symmetry
8 times less work

```c
void set8(int x, int y, pixel v) {
    set( x, y, v)
    set( y, x, v)
    set( y,-x, v)
    set( x,-y, v)
    set(-x, y, v)
    set(-y,-x, v)
    set(-y, x, v)
    set(-x, y, v)
    set(-x, y, v)
}
```
void circle(int R, Pixel v) {
    int x, r2;
    float y;
    r2 = R*R;
    for(x = 0; y < x; ++x) {
        y = sqrt(r2 - x*x);
        set8(x, round(y), v);
    }
}
graphics programming is maximizing SPEED  QUALITY  STYLE
speed

so you can draw more stuff

animating 1 million particles
video
animating 1e6 particles

FPS = 30 frames/sec
1 / 30 = 0.033s = 1 frame
0.033 / 1,000,000 = 33ns, for one particle
2GHz CPU cycle ≈ 1ns

- 33 ns ≈ 33 additions
- 33 ns ≈ 6 multiplications
- 33 ns ≈ 5 multiplication + 27 additions
- IRL, you've got multiple cores, GPU and other stufff

just read GPBB on basics, how to get most out of CPU/GPU
Intuition test

```c
int[] s_count = new int[1024];
void UpdateCounter(int position) {
    for (int j = 0; j < 100000000; j++) {
        s_count[position] = s_count[position] + 3;
    }
}
```

Case 1: From single processor with params 1, 2, 3, 4
Case 2: Run in parallel with params 1, 2, 3, 4

a) 1 is ~4x faster than 2
b) 1 is ~2x faster than 2
c) 2 is ~2x faster than 1
d) 2 is ~4x faster than 1
7 random things

- function calls are slow, if done 1e6 times
  - unless function call is negligible to the function time
- 1 multiplication ≈ 5 additions (used to be 10)
  - unless multiplications are independent
- floating point values accumulate errors
  - if used iteratively
- bad branching is 6 times slower
  - easily predictable branches cost nothing
- caches are your friends
  - unless working on multicore, and they are trying to use the same cache location
- reg ≤ 1, L1 ~ 3, L2 ~ 14, RAM ~ 240
  - predictable RAM usage makes things faster
- don't calculate stuff, you've already calculated
  - except when calculating is faster than retrieving from memory

branch-prediction

cache-effects

things-about-memory
our line drawing is still too slow

// set   <-- ignore that, for simplicity
// round  <-- really slow thing
// y += m <-- error accumulation

```c
line(int x0, int y0, int x1, int y1, Pixel v) {
    int x; float m, y;
    m = (y1 - y0) / (x1 - x0);
    y = y0
    for (x = x0; x <= x1; ++x) {
        set(x, round(y), v);
        y += m
    }
}
```
Bresenham line

- Bresenham's line algorithm (1965) uses only integer arithmetic.
- Computes \((x_{i+1}, y_{i+1})\) incrementally by using \((x_i, y_i)\)
- Generalizes to circles and "oriented" ellipses.
- Assume \(0 < m < 1\)
Choosing next pixel

- \((x_i, y_i)\) was chosen previously
- choose either E or SE based on \(Q\)'s relation to \(M\)

Use function for line:

\[ F(x, y) = ax + by + c \]

- \(F(M) > 0\), "above" the line
- \(F(M) < 0\), "below" the line
- \(F(M) = 0\), "on" the line
Decision variable \( d \)

\[
d = F(M) = F(x_i + 1, y_i + 1/2)
\]

- if \( d > 0 \), choose SE
- if \( d < 0 \), choose E
- if \( d = 0 \), choose one

update \( d \) after each iteration
Update $d$ using either $\Delta_E$ or $\Delta_{SE}$

if $E$ pixel is chosen, \( d_{\text{new}} = d_{\text{old}} + \Delta_E \)

\[
d_{\text{new}} = F(x_i + 2, y_i + 1/2) \\
= a(x_i + 2) + b(y_i + 1/2) + c \\
d_{\text{old}} = F(x_i + 1, y_i + 1/2) \\
= a(x_i + 1) + b(y_i + 1/2) + c \\
\Delta_E = d_{\text{new}} - d_{\text{old}} = a
\]

if SE pixel is chosen, \( d_{\text{new}} = d_{\text{old}} + \Delta_{SE} \)

\[
d_{\text{new}} = F(x_i + 2, y_i + 3/2) \\
= a(x_i + 2) + b(y_i + 3/2) + c \\
d_{\text{old}} = F(x_i + 1, y_i + 1/2) \\
= a(x_i + 1) + b(y_i + 1/2) + c \\
\Delta_{SE} = d_{\text{new}} - d_{\text{old}} = a + b
\]
Initializing $d$

Initial value: $d = F(M)$ for the first midpoint $M$

$$d = F(x_o + 1, y_o + 1/2)$$
$$= a(x_o + 1) + b(y_o + 1/2) + c$$
$$= F(x_o, y_o) + a + b/2$$

We eliminate the fraction $b/2$ by using $2F$ instead of $F$

$$d_o = 2F(x_o + 1, y_o + 1/2)$$
$$= 2a(x_o + 1) + 2b(y_o + 1/2) + 2c$$
$$= 2F(x_o, y_o) + 2a + b$$

$\Delta_E = 2a$

$\Delta_{SE} = 2(a + b)$
Bresenham's Algorithm

\[ \begin{align*}
    b &= x_0 - x_1; \\
    a &= y_1 - y_0; \\
    d &= 2a + b; \\
    dE &= 2a; \\
    dSE &= 2(a + b); \\
    \text{set}(x_0, y_0, v); \\
    \text{for } (x = x_0, y = y_0; x < x_1; x++) \{ \\
        \text{if } (d <= 0) \\
        \quad d += dE; \\
        \text{else } \{ \\
        \quad d += dSE; \\
        \quad y++; \\
        \} \\
        \text{set}(x, y, v); \\
    \}
\]

\( 0 < m < 1, x_0 < x_1 !!! \)
void midpoint_line(int x0, int y0, int x1, int y1) {
    int x,y; /* current pixel coordinate */
    int a,b; /* major,minor line dimensions */
    int d; /* decision variable */
    int xdiag, ydiag; /* minor axis (diagonal) increment */
    int xdom, ydom; /* major axis (dominant) increment */
    int dd_dom, dd_diag; /* delta d for dom. and diag. increment */
    int count; /* number of pixels to set */

    a = x1 - x0; b = y1 - y0;
    xdiag = (a < 0) ? -1 : +1;
    ydiag = (b < 0) ? -1 : +1;
    a = abs(a); b = abs(b);
    if (a < b) {swap(a,b); xdom = xdiag; ydom = 0; }
    else   { xdom = 0; ydom = ydiag; }

    dd_dom = b << 1;
    d = dd_dom - a;
    dd_diag = (b - a) << 1;

    x = x0; y = y0;
    for (count = a+1; count; count--) {
        setPixel(x,y);
        if (d <= 0) {d += dd_dom; x += xdom; y += ydom; }
        else        {d += dd_diag; x += xdiag; y += ydiag; }
    }
}
Line is too easy... now the circle

// sqrt  <-- really, really bad
// x*x   <-- multiplication, why oh why..
// round <-- you again

void circle(int R, Pixel v){
    int x, r2;
    float y;
    r2 = R*R;
    for(x = 0; y < x; ++x){
        y = sqrt(r2 - x*x);
        set8( x, round(y), v);
    }
}
Circle Midpoint Algorithm

draw pixels using only one octant

\[ F(x, y) = x^2 + y^2 - R^2 \]

- \( F(x, y) = 0 \), "on" the circle
- \( F(x, y) > 0 \), "outside"
- \( F(x, y) < 0 \), "inside"
Choosing the next pixel

decision variable $d$

$$d = F(M) = F(x + 1, y + 1/2)$$

$d > 0$, choose E

$d \leq 0$, choose SE
Change of $d$ using $E$

\[
d_{\text{new}} = (x+2)^2 + (y + 1/2)^2 - R^2
\]
\[
d_{\text{old}} = (x+1)^2 + (y + 1/2)^2 - R^2
\]
\[
\Delta d = d_{\text{new}} - d_{\text{old}} = 2x + 3
\]
Change of $d$ using $E$

\[ d_{\text{new}} = (x+2)^2 + (y + 3/2)^2 - R^2 \]
\[ d_{\text{old}} = (x+1)^2 + (y + 1/2)^2 - R^2 \]
\[ \Delta d = d_{\text{new}} - d_{\text{old}} = 2x + 2y + 5 \]
Initial value of $d$

\[ d_o = F(M_o) \]
\[ d_o = F(1, -R + 1/2) \]
\[ d_o = (1)^2 + ( -R + 1/2 )^2 - R^2 \]
\[ d_o = 5/4 - R \]
Midpoint Circle Algorithm

\begin{verbatim}
x = 0;
y = -R;
d = 5/4 - R; /* real */
set8(x,y,v);
while (y > x) {
    if (d > 0) { /* E chosen */
        d += 2*x + 3;
        x++;
    } else {    /* SE chosen */
        d += 2*(x+y) + 5;
        x++; y++;
    }
set8(x,y,v);
}
\end{verbatim}
Not done yet

- still using real values
- Let's create new decision variable $h$
  - $h = d - \frac{1}{4}$
- Substitute $h + \frac{1}{4}$ for $d$ in the code
- Note $h > -\frac{1}{4}$ can be replaced with $h > 0$ since $h$ will always have an integer value
New circle algorithm

x = 0;
y = -R;
h = 1 – R;
set8(x,y,v);
while (y > x) {
    if (h > 0) { /* E chosen */
        h += 2*x + 3;
x++;
    } else { /* SE chosen */
        h += 2*(x+y) + 5;
x++; y++;
    }
    set8(x,y,v);
}
Second-Order Differences

- This idea can be applied iteratively until we have removed all multiplications.
- Currently no big benefit as
  - $2^x = x \ll 1$
- Left-shift is as quick as addition
// with second order differences
x = 0; y = -R;
h = 1 – R;
dE = 3; dSE = -2*R + 5;
setPixel(x,y,v);
while (y > x) {
    if (h > 0) { /* E chosen */
        h += dE;
        dE += 2; dSE += 2;
        x++;
    }
    else { /* SE chosen */
        h += dSE;
        dE += 2; dSE += 4;
        X++; y++;
    }
    set8(x,y,v);
}
Optimization using differences

problem:

- \( f(x_1, x_2, \ldots) \)
- \( x = <x_1, x_2, \ldots> \)
- calculate \( f(x) \) for \( x \in S \)
- \( S \) is large
  - \(|S| > 1e3\)

calculating \( f \) is slow

- calculating
  - \( \delta f(x, \Delta x) = f(x + \Delta x) - f(x) \)
  - it's faster for some \( \Delta x \)
  - e.g. \( \Delta x = <0, x_2, x_3, 0, \ldots> \)

optimization:

- find some \( \delta f \)
- sort based on speed of \( \delta f \)
- compose new \( f'(x, \Delta x) \) that uses previous values
Example

\[ f(x,y,z) = g(y,z) + x^2 + x \]

\[ \delta f = \delta f(x+\Delta x, y, z) = \]
\[ = f(x+\Delta x,y,z) - f(x,y,z) = \]
\[ = (x+\Delta x)^2 + (x + \Delta x) - (x^2 + x) \]
\[ = 2x\Delta x + \Delta x^2 + \Delta x \]

\[ f'(x,y,z,\Delta x,\Delta y,\Delta z) = \]
\[ if( \Delta y=\Delta z=0 ) { \]
\[ f[x,y,z] + 2x\Delta x + \Delta x^2 + \Delta x \]
\[ } else { \]
\[ f(x+\Delta x,y+\Delta y,z+\Delta z) \]
\[ } \]
onwards... to filling stuff
Scan Conversion

(0,0)

+X

scan lines

horizontal retrace

vertical retrace

+Y
Rectangles & Circles

rectangle(x0,x1,y0,y1,v) {
    for(y=y0; y < y1; ++y) {
        for(x=x0; x < x1; ++x) {
            set(x, y, v);
        }
    }
}

// similarly can be done for circle
// just calculate one side and other side
// draw a line between them
// next row
Polygons...

- Convex
- Concave
- Self-intersecting
- With holes
Brute-force filling...

for each pixel Point {
    if( IsInside(Poly, Point) ){
        set(Point.x, Point.y, v)
    }
}
is inside convex

test against each edge whether point is on the right side
is inside concave

- find sum over all angels
- if $\Sigma \alpha \approx 0$, then outside
is inside complex

- find horizontal line intersections
- sort based on x coordinate
- count how many to the point
void ScanPolygon(Polygon T, Pixel v) {
    sort edges by maxy
    make empty “active edge list”
    for each scanline (top-to-bottom) {
        insert/remove edges from “active edge list”
        update x coordinate of every active edge
        sort active edges by x coordinate
        for each pair of active edges (left-to-right)
            hline(xi, xi+1, y, v);
    }
}
Polygon clipping
Cohen-Sutherland Line Clipping

- clip lines to rectangle

calculate outcodes for each line ending

if both 0, can be accepted trivially

if share a 1, can be discarded
Cohen-Sutherland Line Clipping

calculate outcodes A and B

if( A||B == 0)
    ACCEPT
else(A&&B != 0)
    DECLINE
else
    INVESTIGATE
Cohen-Sutherland Line Clipping

clip the endings
side : one clip
corner : two clips
Sutherland-Hodgman's Polygon Clipping

Window is a convex polygon
Clipped polygon can be anything

Clip against each single edge (half-plane), repeat for each window edge.

let's pick a clipping edge
Clipping against one edge

half-plane

inside

outside

polygon
s inside, p inside

half-plane

inside

p

s

case 1

output p

outside

polygon
s inside, p outside
s outside, p outside

half-plane

inside

polygon

case 3

()
s outside, p inside

case 4
output i, p
done, for this edge
\[ s = v_{n-1} \]

for \( i = 0 \) to \( n - 1 \) do

\[ p = v_i \]

case 1: output \( p \)

case 2: output \( i \)

case 3: nothing

case 4: output \( i \); output \( p \)

\[ s = p; \]
now something different...
Scaling

to find \( ? \) pixels, map the point to \( p \) on the original picture
use \( p \) and it's surrounding pixels to calculate \( ? \)
upscaling
downscaling
of course, different ways of calculating new pixel

this is too boring....
Pixel-Art Scaling

we want something better, for some specific content [upscaling images with solid colors and thin lines]

just to play old games with better graphics :D
EPX

Eric's Pixel Expansion, LucasArts, 1992, for SCUMM games
Scale Image 2×, in real-time
n=0
1,2,3,4=P

C==A » 1=A, n++
A==B » 2=B, n++
B==D » 4=D, n++
D==C » 3=C, n++
A==D » n++
B==C » n++
if n>=3, 1=2=3=4=P
Scale2×/AdvMAME2×

n=0
1,2,3,4=P

C==A & C≠D & A≠B » 1=A
A==B & A≠C & B≠D » 2=B
B==D & B≠A & D≠C » 4=D
D==C & D≠B & C≠A » 3=C

optimization of EPX
Eagle

1,2,3,4=P

V==S==T » 1=S
T==U==W » 2=U
V==X==Y » 3=X
W==Z==Y » 4=Z

(has a bug when S=T=U=V=W=X=Y=Z≠C)
2×SaI, SuperSaI, SuperEagle

Scale and Interpolation engine by Derek Liauw Kie Fa uses some additional blending for target pixels
hqx
by Maxim Stepin
s = isClose(S - C)
t = isClose(T - C)
u = isClose(U - C)
v = isClose(V - C)
w = isClose(W - C)
x = isClose(X - C)
y = isClose(Y - C)
z = isClose(Z - C)
p = [s,t,u,v,w,x,y,z]
f = lookup(p)
1 = f(1,C,p)
2 = f(2,C,p)
3 = f(3,C,p)
4 = f(4,C,p)

use closenesses to determine strategy to fill the pixels

|P| = \(2^8\) different ways