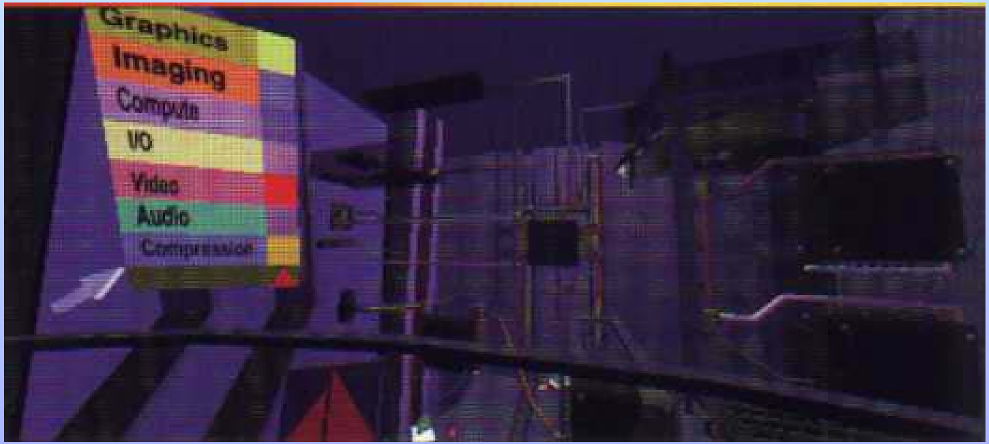
Realtime 3D Computer Graphics & Virtual Reality



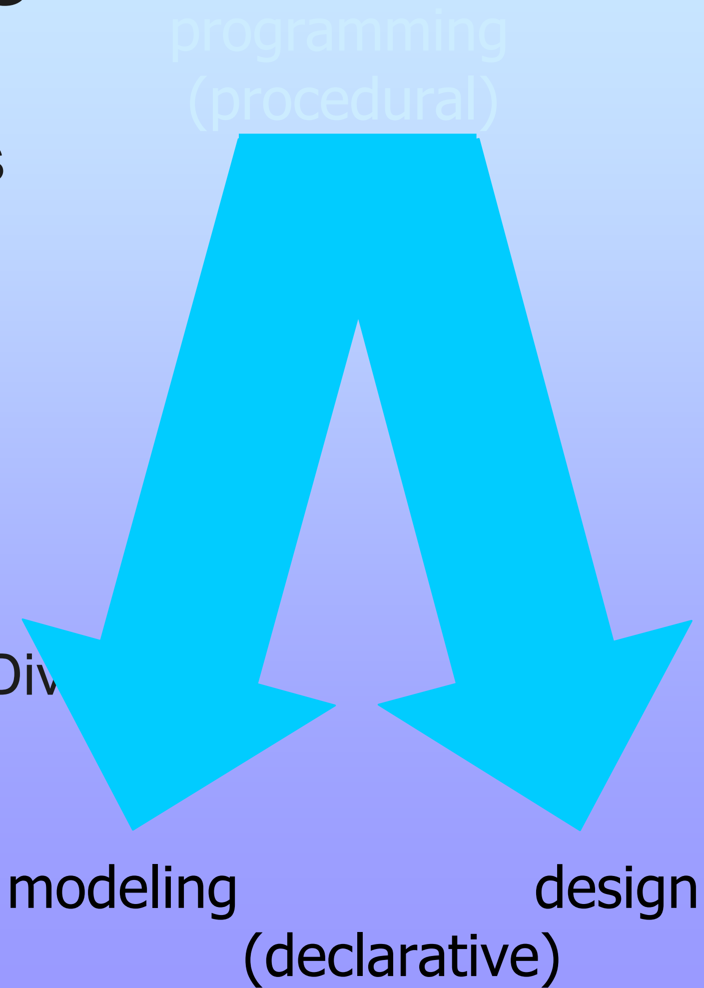
OpenGL Introduction

VR-programming

- Input and display devices are the main hardware interface to users
- Immersion embeds users through the generation of live-like sensory experiences
- *But how is the programmers/designers view?*

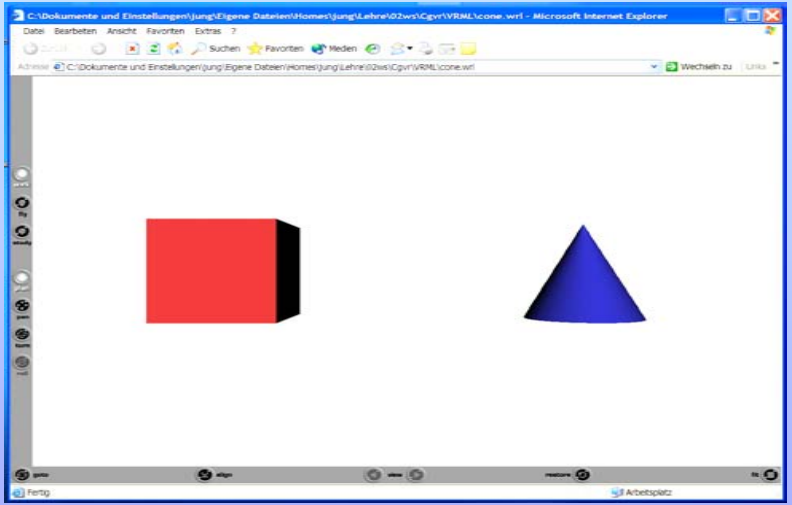
VR-programming tools

- Direct rendering and gfx packages
  - OpenGL, Direct3D, GKS (3D)
- Scene graph based tools
  - VRML, OpenGL Performer, OpenGL Optimizer, Open Inventor, PHIGS+
- VR modeling toolkits
  - AVANGO, World toolkit, Masive1-3, Div
    Lightning, game engines



A Scene Graph Language: VRML

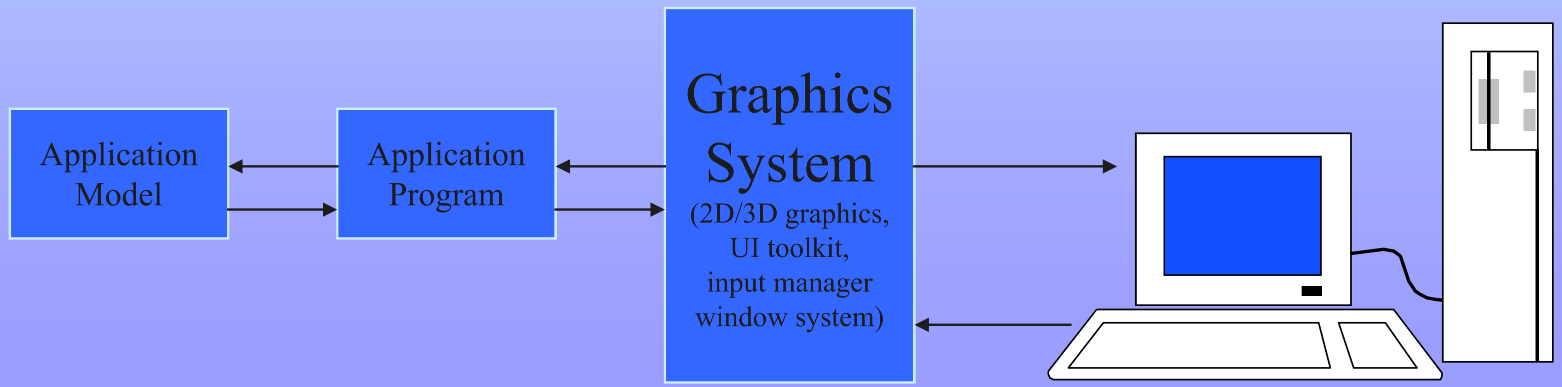
```
#VRML V2.0 utf8
Transform {
  translation -3 0 0
  children Shape {
    geometry Box { }
    appearance Appearance {
      material Material { diffuseColor .8 .2 .2 } }
  }
}
Transform {
  translation 3 0 0
  children Shape {
    geometry Cone { }
    appearance Appearance {
      material Material { diffuseColor .2 .2 .8 } }
  }
}
```



More VRML later in this course!

What is a gfx package?

- software
  - that takes user input and passes it to applications
  - that displays graphical output for applications



An Interactive Introduction to OpenGL Programming

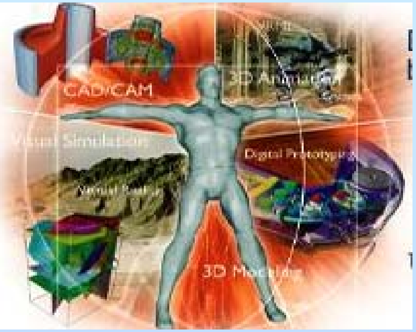
Partly based on SIGGRAPH course notes by Dave Shreiner, Ed Angel and Vicki Shreiner

What You'll See

- General OpenGL Introduction
- Rendering Primitives
- Rendering Modes
- Lighting
- Texture Mapping
- Additional Rendering Attributes
- Imaging

Goals

- Demonstrate enough OpenGL to write an interactive graphics program with
  - custom modeled 3D objects or imagery
  - lighting
  - texture mapping
- Introduce advanced topics for future investigation
- Generate knowledge to understand high-level scene graph based engines for VE-design



OpenGL and GLUT Overview

OpenGL and GLUT Overview

- What is OpenGL & what can it do for me?
- OpenGL in windowing systems
- Why GLUT
- A GLUT program template

What Is OpenGL?

- OpenGL – Open Graphics Library
- Graphics rendering API
  - high-quality color images composed of geometric and image primitives
  - window system independent
  - operating system independent
  - hardware independent layer to different acceleration designs (supporting software modes as well)

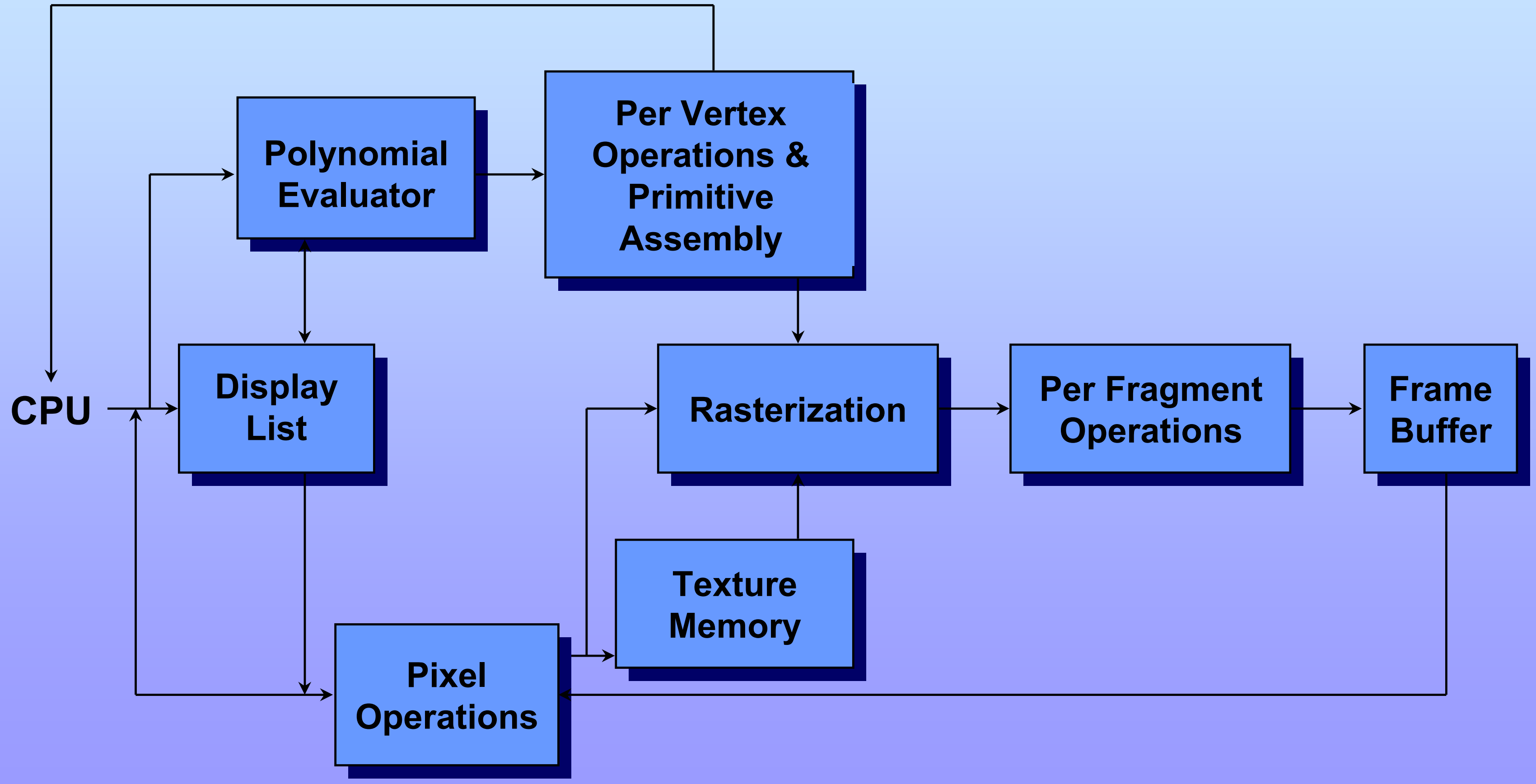
What Is OpenGL?

- Introduced 1992 by SGI
- Based on IRIS GL, an API for the SGI personal IRIS workstation and follow-ups
- Now an open standard that is widely adopted for all types of applications
- Under the supervision of the OpenGL architecture review board

OpenGL Design Goals

- SGI's design goals for OpenGL:
  - High-performance (hardware-accelerated) graphics API
  - Some hardware independence
  - Natural, terse API with some built-in extensibility
- OpenGL has become a standard because:
  - It doesn't try to do too much
    - Only renders the image, doesn't manage windows, etc.
    - No high-level animation, modeling, sound (!), etc.
  - It does enough
    - Useful rendering effects + high performance
  - It is promoted by SGI (& Microsoft, half-heartedly)

OpenGL Architecture



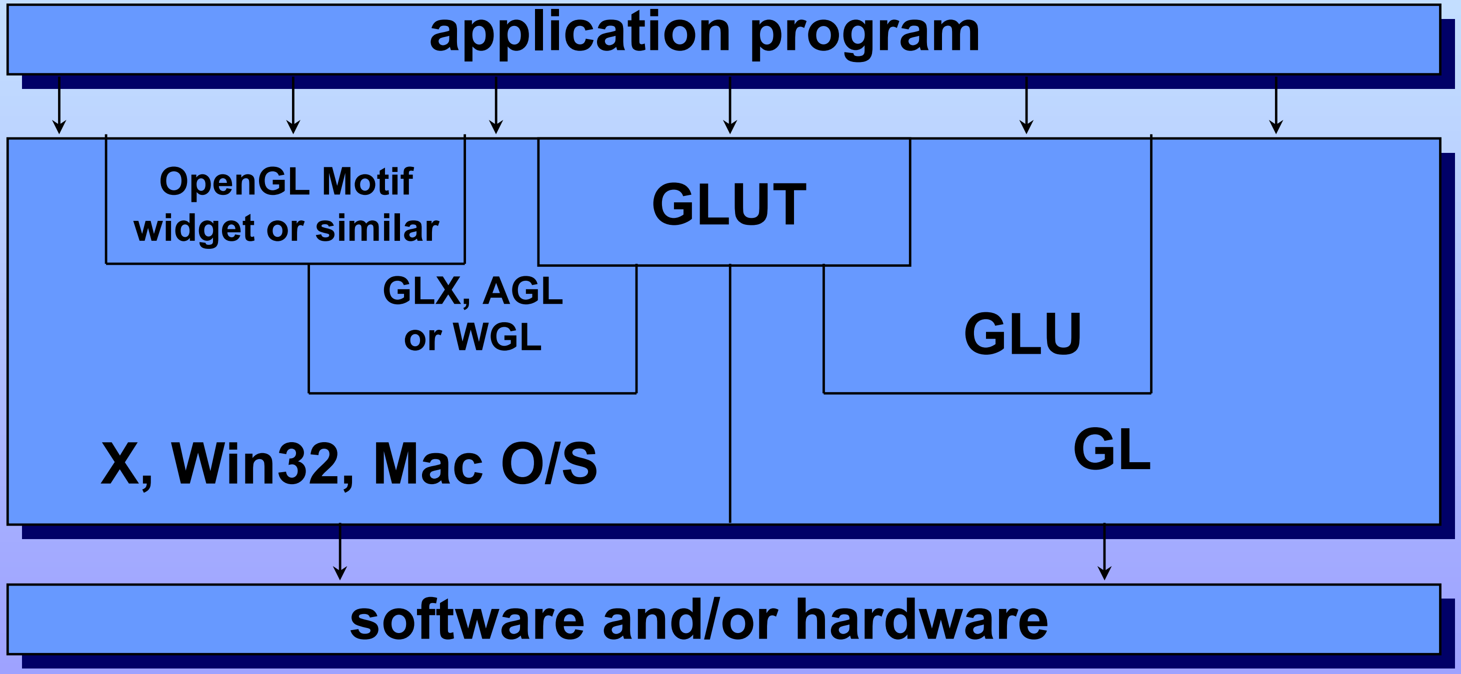
OpenGL as a Renderer

- Geometric primitives
  - points, lines and polygons
- Image Primitives
  - images and bitmaps
  - separate pipeline for images and geometry
    - linked through texture mapping
- Rendering depends on state
  - colors, materials, light sources, etc.

Related APIs

- AGL, GLX, WGL
  - glue between OpenGL and windowing systems
- GLU (OpenGL Utility Library)
  - part of OpenGL
  - NURBS, tessellators, quadric shapes, etc.
- GLUT (OpenGL Utility Toolkit)
  - portable windowing API
  - not officially part of OpenGL

OpenGL and Related APIs



OpenGL: Conventions

- Functions in OpenGL start with **`gl`**
  - Most functions just **`gl`** **`(`**e.g., **`glColor())`**
  - Functions starting with **`glu`** are utility functions **`(`**e.g., **`gluLookAt() )`**
  - Functions starting with **`glx`** are for interfacing with the X Windows system (e.g., in gfx.c)

OpenGL: Conventions

- Variables written in CAPITAL letters
  - Example: GLUT_SINGLE, GLUT_RGB
  - usually constants
  - use the bitwise or command (x | y) to combine constants

Preliminaries

- Headers Files
  - #include <GL/gl.h>
  - #include <GL/glu.h>
  - #include <GL/glut.h>
- Compile with libraries
  - cc myapp.c –o myapp –lgl –lglu –lglut –lm –lX11
  - Adopt different library places using e.g. –L/usr/…

Preliminaries

- Simple make looks like

```
CC = cc
CXX = gcc
LDLIBS = -lglut -lgl -lglu -lX11 -lm -L/usr/…
.c:
  $(CC)     $@.c $(LDLIBS) -o $@
.c++:
    $(CXX)     $@.c++ $(LDLIBS) -o $@
```

- Enumerated Types
  - OpenGL defines numerous types for compatibility between different systems
    - GLfloat, GLint, GLenum, etc.

Preliminaries

- Enumerated Types

Char	C-type	OpenGL type
b	signed char	GLbyte
s	short	GLshort
i	int	Glint, GLsizei
f	float	GLfloat, GLclampf
d	double	GLdouble, GLclampd
ub	unsigned char	GLubyte, GLboolean
us	unsigned char	GLushort
ui	unsigned int	GLuint, GLenum, GLbitfield
	void	GLvoid

OpenGL Command Formats

glVertex3fv(*v*)

Number of components

```
2 - (x,y)
3 - (x,y,z)
4 - (x,y,z,w)
```

Data Type

```
b  - byte
ub - unsigned byte
s  - short
us - unsigned short
i  - int
ui - unsigned int
f  - float
d  - double
```

Vector

omit "v" for scalar form

```
glVertex2f( x, y )
```

GLUT Basics

- Application Structure
  - Configure and open window
  - Initialize OpenGL state
  - Register input callback functions
    - render
    - resize
    - input: keyboard, mouse, etc.
  - Enter event processing loop

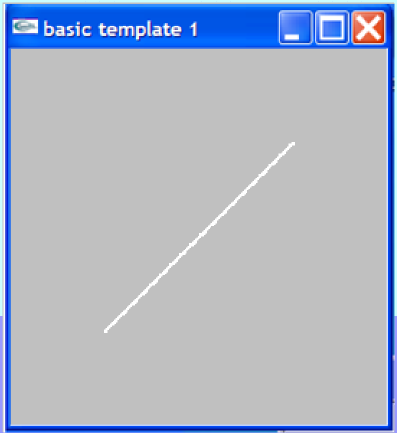
Basic OpenGL template

```
/* simple program template for OpenGL
   progs */

#include <GL/glut.h>

void myDisplay()
{
   /* clear the window */
   glClear(GL_COLOR_BUFFER_BIT);
   /* draw something */
   glBegin(GL_LINES);
         glVertex2f(-0.5, -0.5);
         glVertex2f(0.5, 0.5);
   glEnd();
   glFlush();
}
```

```
int main (int argc,
               char** argv)
{
   glutInit(&argc, argv);
   glutCreateWindow("basic
   template 1");
   glutDisplayFunc(myDisplay);
   glutMainLoop();
}
```



Sample Program

```
void main( int argc, char** argv )
{
  glutInit( argc, argv );
  int mode = GLUT_RGB|GLUT_SINGLE;
  glutInitDisplayMode( mode );
  glutCreateWindow( argv[0] );
  init();
  glutDisplayFunc( display );
  glutKeyboardFunc( key );
  glutMouseFunc( mouse );
  glutIdleFunc( idle );
  glutMainLoop();
}
```

OpenGL Initialization

- Set up whatever state you're going to use

```
void init( void )
{
  glClearColor( 0.0, 0.0, 0.0, 1.0 );
  glColor3f( 1.0, 1.0, 1.0);
  glClearDepth( 1.0 );
  glEnable( GL_LIGHT0 );
  glEnable( GL_LIGHTING );
  glEnable( GL_DEPTH_TEST );
}
```

GLUT Callback Functions

- A callback is a routine to call when something happens
  - window resize or redraw
  - user input
  - animation

GLUT Callback Functions

- "Register" callbacks with GLUT

```
glutDisplayFunc( display );
glutIdleFunc( idle );
glutResizeFunc( resize );
glutKeyboardFunc( keyboard );
glutSpecialFunction( special )
glutMouseFunc( mouse );
glutMotionFunc( mouse_motion );
glutPassiveMotionFunc( mouse_pmotion );
glutEntryFunc( on_focus_change );
```

Rendering Callback

- **Do all of your drawing here**

```
glutDisplayFunc( display );
```

```
void display( void )
{
  glClear( GL_COLOR_BUFFER_BIT );
  glBegin( GL_LINES );
    glVertex2f( 50.0, 50.0 );
    glVertex2f( 100.0, 100.0 );
    glVertex2f( 70.0, 10.0 );
    glVertex2f( 100.5, 70.1 );
  glEnd();
  glFlush();
}
```

Idle Callbacks

- Use for animation and continuous update

```
glutIdleFunc( idle );
```

```
void idle( void )
{
  t += dt;
  glutPostRedisplay();
}
```

“smart” update

```
glutPostRedisplay();
```

- Requests that the display callback be executed
- Allows the implementation to be smarter in deciding when to carry out the display callback
  - As GLUT goes through the event loop, more than one event can require a redraw which should only be carried out once during the loop

Idle callback and smart update

- Processing an animation should be done with respect to the elapsed time
  - t += dt;
- No hint when the update occurs
- How can we achieve a minimal simulation and frame rate using this application structure?

User Input Callbacks

- Process user keyboard input

glutKeyboardFunc(*keyboard*);

```
void keyboard( char key, int x, int y )
{
  switch( key ) {
    case 'q' : case 'Q' :
      exit( EXIT_SUCCESS );
      break;

    case 'r' : case 'R' :
      rotate = GL_TRUE;
      break;
  }
}
```

User Input Callbacks

- Process user special keyboard input

`glutSpecialFunction( special );`

```
void special( char key, int x, int y )
{
  if( key == GLUT_KEY_F1)    help();
  if( key == GLUT_KEY_UP)    up();
  if( key == GLUT_KEY_DOWN)  down();
  if( key == GLUT_KEY_LEFT)  left();
  if( key == GLUT_KEY_RIGHT) right();
}
```

User Input Callbacks

- Process user mouse input

glutMouseFunc(*mouse*);

```
void mouse( int button, int state, int
  x, int y )
{
  if (state == GLUT_DOWN &&
      button == GLUT_LEFT_BUTTON)
         exit(EXIT_SUCCESS);
}
```

User Input Callbacks

- Process user mouse motion input with a pressed button

glutMotionFunc(mouse_*motion*);

```
void mouse_motion( int x, int y )
{
  if (first_time_called)
     glBegin();
      …
      glEnd();
      first_time_called = GL_false;
}
```

User Input Callbacks

- Process user mouse motion input without a button pressed

```
glutPassiveMotionFunc( mouse_pmotion );
```

```
void mouse_pmotion( int x, int y )
{
  last_points_visited.push(pair(x,y));
  if( last_points_visited.size() > 100)
    last_points_visited.remove_last();
}
```

User Input Callbacks

- Process leaving and entering the OpenGL window with the mouse

```
glutEntryFunc( on_focus_change );
```

```
void on_focus_change( int state )
{
 if (state == GLUT_ENTERED)
   beep();
 if (state == GLUT_LEFT)
   exit(EXIT_SUCCESS);
}
```

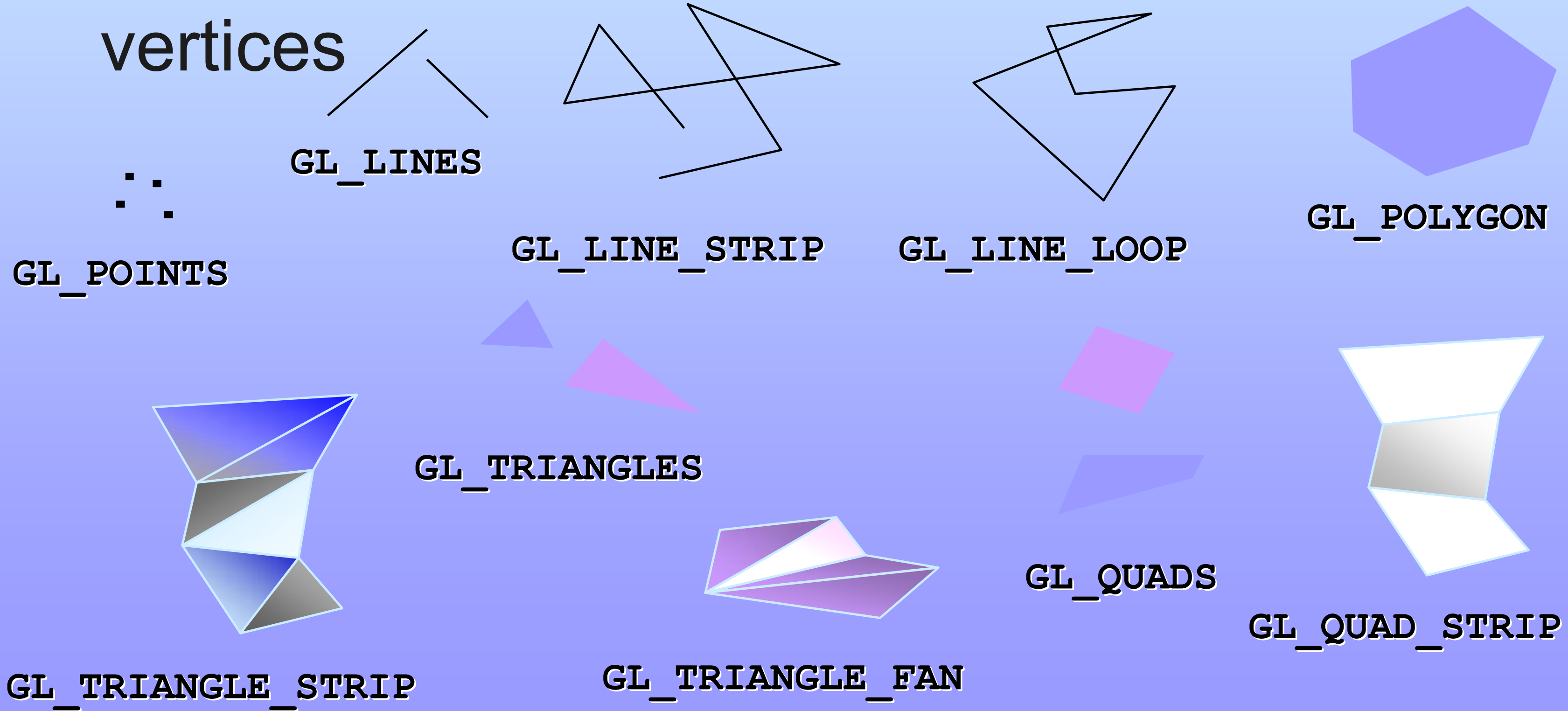
Elementary raster algorithms for fast rendering

Elementary Rendering

- Geometric Primitives
  - Line processing
  - Polygon processing
- Managing OpenGL State
- OpenGL Buffers

OpenGL Geometric Primitives

- All geometric primitives are specified by vertices

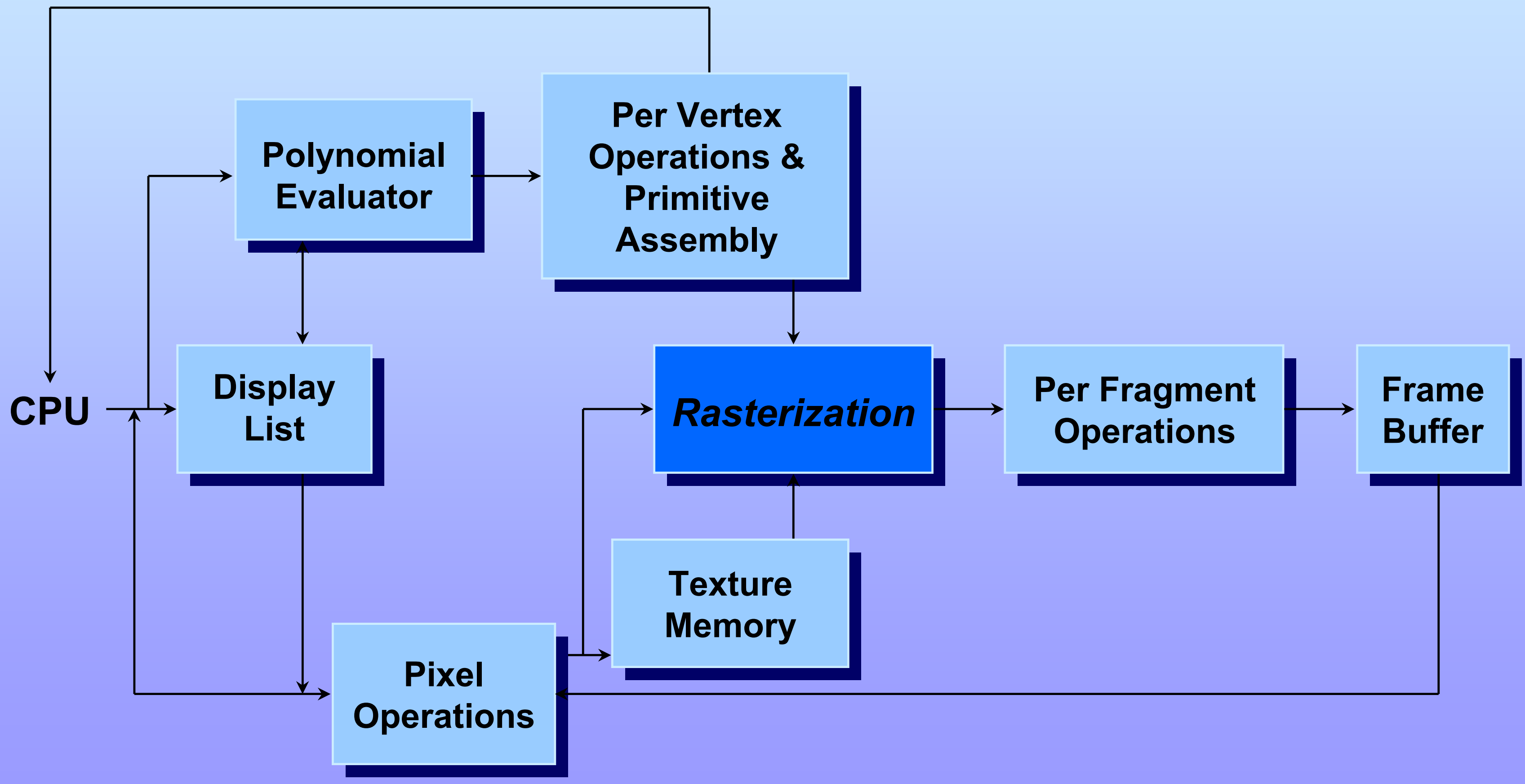


Design of Line Algorithms

Why Lines?

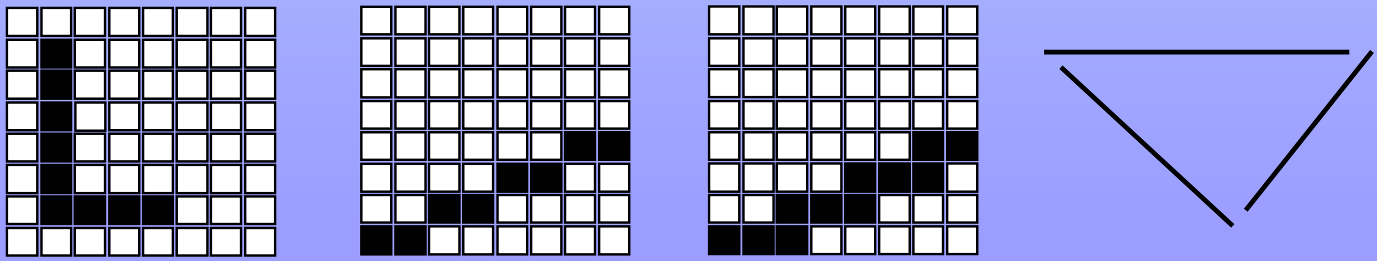
- Lines:
  - Most common 2D primitive - done 100s or 1000s of times each frame, even 3D wireframes are eventually 2D lines!
  - Lines are *compatible* with vector displays but nowadays most displays are raster displays. Any render stage before viz might need discretization.
  - Optimized algorithms contain numerous tricks/techniques that help in designing more advanced algorithms for line processing.

Line Algorithms in the OpenGL Architecture



Line Requirements

- Must compute integer coordinates of pixels which lie on or near a line or circle.
- Pixel level algorithms are invoked hundreds or thousands of times when an image is created or modified – must be fast!
- Lines must create visually satisfactory images.
  - Lines should appear straight
  - Lines should terminate accurately
  - Lines should have constant density
- Line algorithm should always be defined.



Basic Math Review

Point-slope Formula For a Line

Given two points (X_1,Y_1), (X_2, Y_2)

Consider a third point on the line:

$P = (X,Y)$

Slope = $(Y_2 - Y_1)/(X_2 - X_1)$

$= (Y - Y_1)/(X - X_1)$

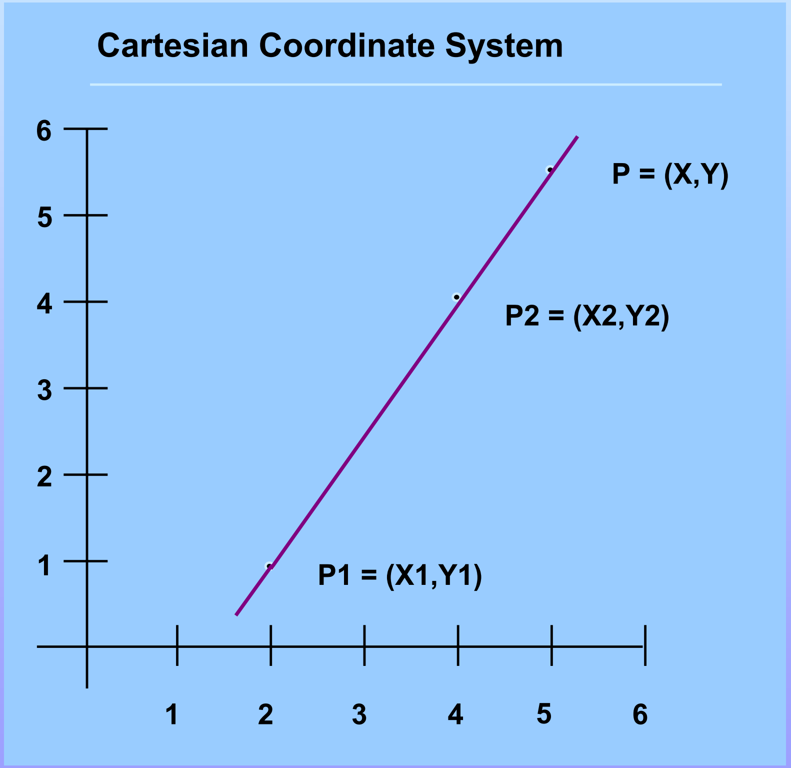
Solving For Y

$Y = [(Y_2-Y_1)/(X_2-X_1)]*(X-X_1)+ Y_1$

or, plug in the point (0, b) to get the

Slope-intercept form:

$Y = mx + b$



SLOPE = $\frac{\text{RISE}}{\text{RUN}} = \frac{\text{Y2-Y1}}{\text{X2-X1}}$

Other Helpful Formulas

- **Length of line segment between P_1 and P_2:**
  L = $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

- **Midpoint of a line segment between P_1 and P_3:**
  $P_2 = ((X_1+X_3)/2 , (Y_1+Y_3)/2)$

- Two lines are **perpendicular** iff
  1) $M_1 = -1/M_2$
  2) Cosine of the angle between them is 0.

Using this information, what are some possible algorithms for line drawing?

Parametric Form

Given points $P_1 = (X_1, Y_1)$ and $P_2 = (X_2, Y_2)$

$$\mathbf{X = X_1 + t(X_2-X_1)}$$
$$\mathbf{Y = Y_1 + t(Y_2-Y_1)}$$

t is called the parameter. When

$t = 0$ we get (X_1,Y_1)

$t = 1$ we get (X_2,Y_2)

As $0 < t < 1$ we get all the other points on the line segment between (X_1,Y_1) and (X_2,Y_2).

New algorithm ideas based on parametric form?

Simple DDA* Line Algorithm

```
void DDA(int X1,Y1,X2,Y2)
{
   int    Length, I;
   float X,Y,Xinc,Yinc;

   Length = ABS(X2 - X1);
   if (ABS(Y2 - Y1)  > Length)
        Length = ABS(Y2-Y1);
   Xinc = (X2 - X1)/Length;
   Yinc = (Y2 - Y1)/Length;
```

```
   X = X1;
   Y = Y1;
   while(X<X2){
      Plot(Round(X),Round(Y));
      X = X + Xinc;
      Y = Y + Yinc;
   }
}
```

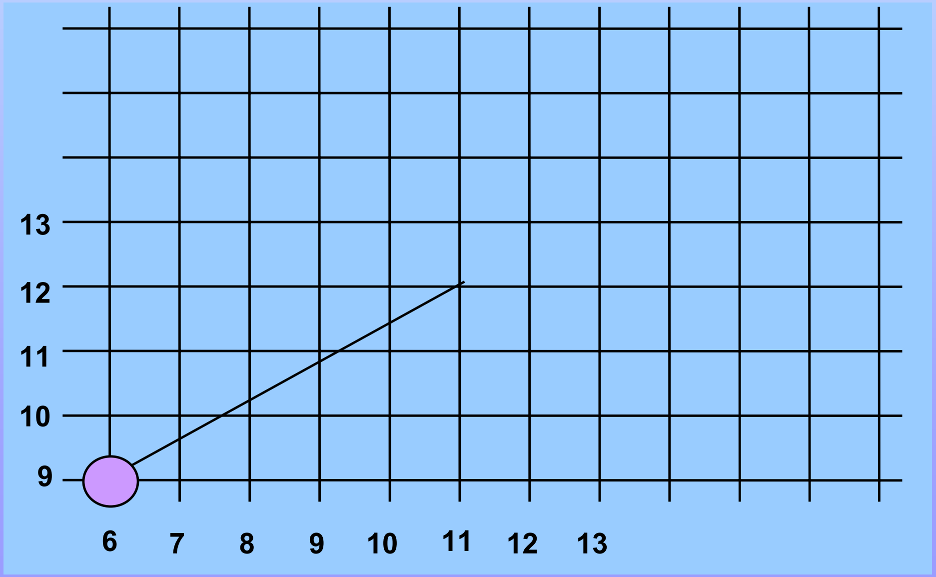
DDA creates good lines but it is too time consuming due to the round function and long operations on real values.

*DDA: Digital Differential Analyzer

DDA Example

Compute which pixels should be turned on to represent the line from (6,9) to (11,12).

Length = ?
Xinc = ?
Yinc = ?

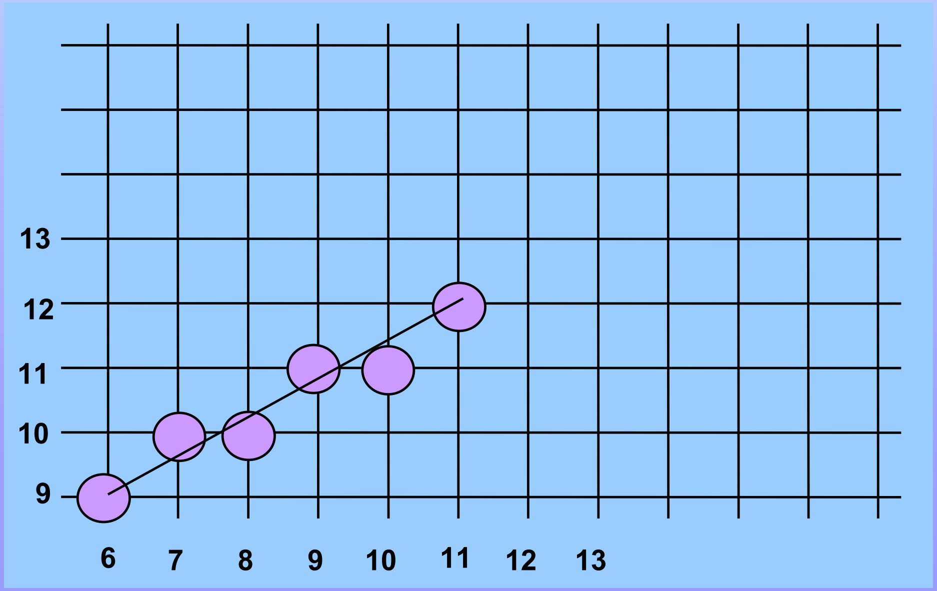


DDA Example

Line from (6,9) to (11,12).

Length := Max of (ABS(11-6), ABS(12-9)) = 5
Xinc := 1
Yinc := 0.6

Values computed are:
(6, 9)
(7, 9.6)
(8, 10.2)
(9, 10.8)
(10, 11.4)
(11, 12)

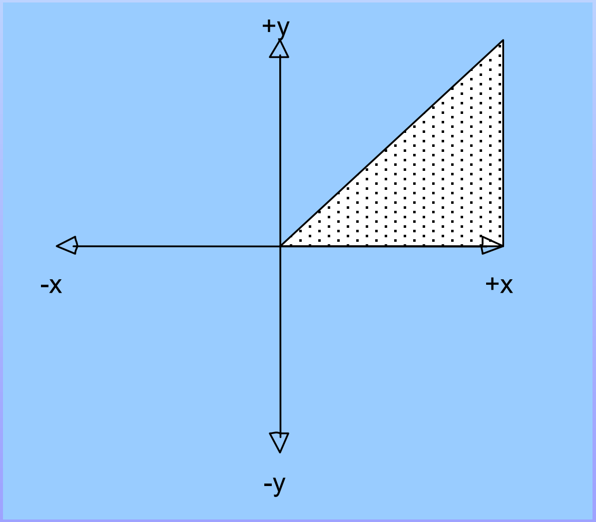


Fast Lines – Midpoint Method

- Simplifying assumptions: Assume we wish to draw a line between points (0,0) and (a,b) with slope m between 0 and 1 (i.e. line lies in first quadrant).

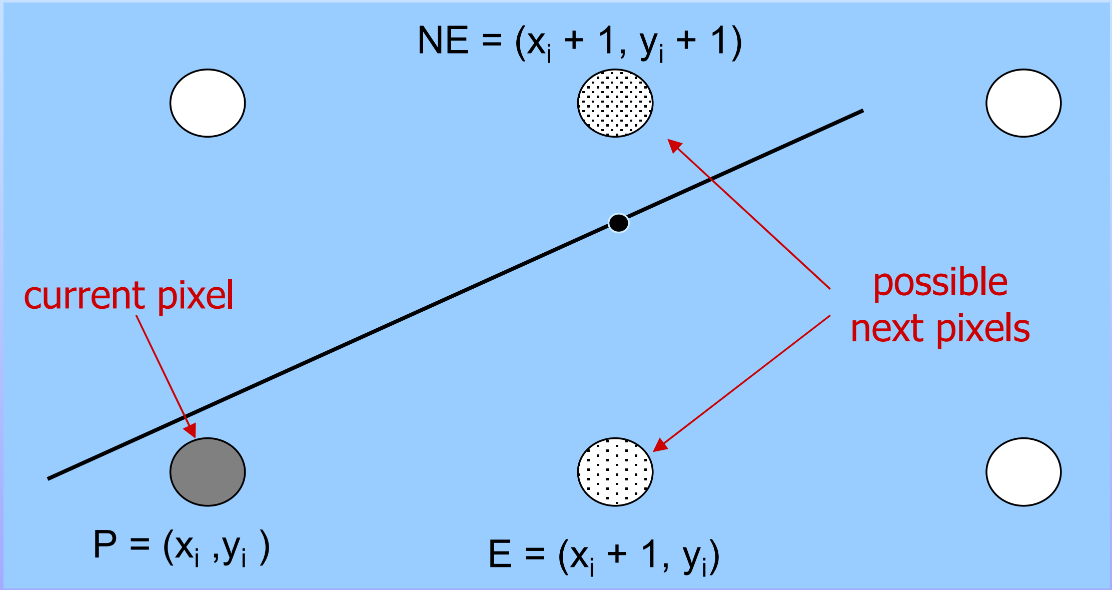
- The general formula for a line is
  y = mx + B where
  m is the slope of the line and
  B is the y-intercept. From our assumptions **m** = b/a and **B** = 0.

- y = (b/a)x + 0
  --> f(x,y) = bx - ay = 0
  is an equation for the line.



Fast Lines (cont.)

- For lines in the first quadrant, given one pixel on the line, the next pixel is to the right (E) or to the right and up (NE).

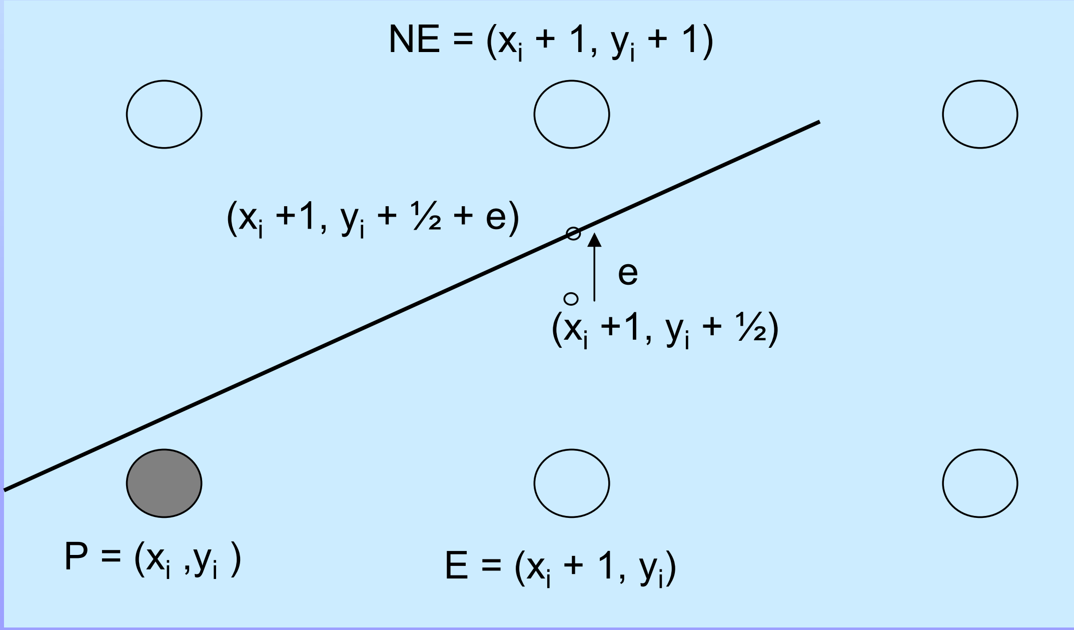


- Having turned on pixel P at (x_i, y_i), the next pixel is
  - NE at (x_i+1, y_i+1) or
  - E at (x_i+1, y_i).
- Choose the pixel closer to the line $f(x, y) = bx - ay = 0$.

Fast Lines (cont.)

- The midpoint between pixels E and NE is $(x_i + 1, y_i + ½)$.
- Let e be the “upward” distance between the midpoint and where the line actually crosses between E and NE.
- If <u>e is positive</u> the line crosses above the midpoint and is closer to NE.
- If <u>e is negative</u>, the line crosses below the midpoint and is closer to E.

  - *To pick the correct point we only need to know the <u>sign</u> of e.*



The Decision Variable

$f(x_i+1, y_i+\frac{1}{2} + e) = 0$ (point on line)

$= b(x_i + 1) - a(y_i + \frac{1}{2} + e)$

$= b(x_i + 1) - a(y_i + \frac{1}{2}) - ae$

$= f(x_i + 1, y_i + \frac{1}{2}) - ae$

→ $f(x_i + 1, y_i + \frac{1}{2}) = ae$

Let $\mathbf{d_i} = f(x_i + 1, y_i + \frac{1}{2}) = \mathbf{ae}$; $\mathbf{d_i}$ is known as the **decision variable**.

Since $a \geq 0$, d_i has the same sign as e.

Therefore, we only need to know the value of d_i to choose between pixels E and NE. If $d_i \geq 0$ choose NE, else choose E.

But, calculating d_i directly each time requires at least two adds, a subtract, and two multiplies -> too slow!

Decision Variable calculation

Algorithm:

Calculate d_0 directly, then for each i >= 0:

if $d_i \geq 0$ Then

Choose NE = $(x_i + 1, y_i + 1)$ as next point

$$d_{i+1} = f(x_{i+1} + 1, y_{i+1} + ½) = f(x_i + 1 + 1, y_i + 1 + ½)$$
$$= b(x_i + 1 + 1) - a(y_i + 1 + ½) = f(x_i + 1, y_i + ½) + b - a$$
$$= d_i + b - a$$

else

Choose E = $(x_i + 1, y_i)$ as next point

$$d_{i+1} = f(x_{i+1} + 1, y_{i+1} + ½) = f(x_i + 1 + 1, y_i + ½)$$
$$= b(x_i + 1 + 1) - a(y_i + ½) = f(x_i + 1, y_i + ½) + b$$
$$= d_i + b$$

➔ Knowing d_i, we need only add a **constant** term to find d_{i+1} !

Fast Line Algorithm

The initial value for the decision variable, d_0, may be calculated directly from the formula at point (0,0).

$d_0 = f(0 + 1, 0 + 1/2) = b(1) - a(1/2) = b - a/2$

Therefore, the algorithm for a line from (0,0) to (a,b) in the first quadrant is:

```
x = 0;
y = 0;
d = b - a/2;
for(i = 0; i < a; i++) {
     Plot(x,y);
     if (d ≥ 0) {
            x = x + 1;
            y = y + 1;
            d = d + b - a;
     }
     else {
            x = x + 1;
            d = d + b
     }
}
```

Note that the only non-integer value is a/2. If we then multiply by 2 to get d' = 2d, we can do all integer arithmetic. The algorithm still works since we only care about the sign, not the value of d.

Bresenham's Line Algorithm

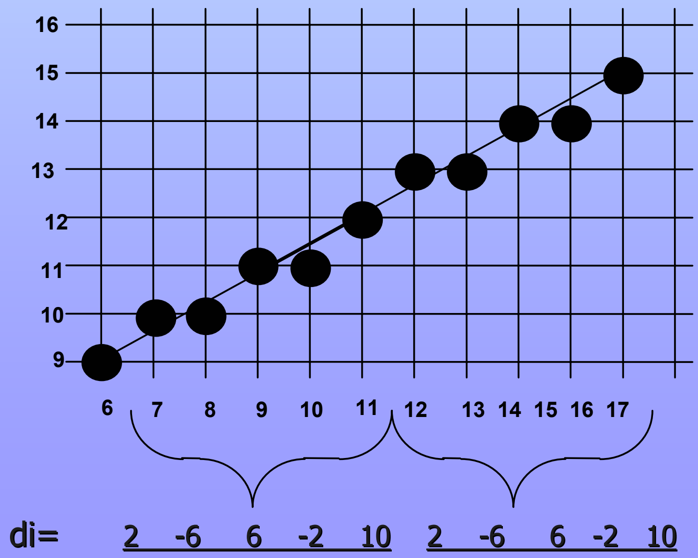
We can also generalize the algorithm to work for lines beginning at points other than (0,0) by giving x and y the proper initial values. This results in Bresenham's Line Algorithm.

```
{Bresenham for lines with slope between 0 and 1}
        a = ABS(xend - xstart);
        b = ABS(yend - ystart);
        d = 2*b - a;
        Incr1 = 2*(b-a);
        Incr2 = 2*b;
        if (xstart > xend) {
                x = xend;
                y = yend
        }
        else {
                x = xstart;
                y = ystart
        }
```

```
    for (i = 0; i<a; i++){
        Plot(x,y);
        x = x + 1;
        if (d ≥ 0) {
            y = y + 1;
            d = d + incr1;
        }
        else
            d = d + incr2;
    }
}
```

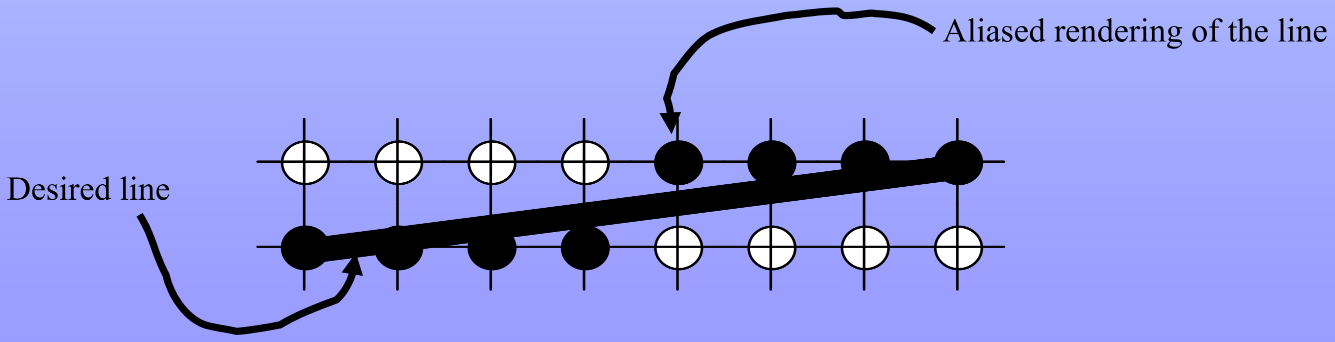
Optimizations

- Speed can be increased even more by detecting cycles in the decision variable. These cycles correspond to a repeated pattern of pixel choices.
- The pattern is saved and if a cycle is detected it is repeated without recalculating.



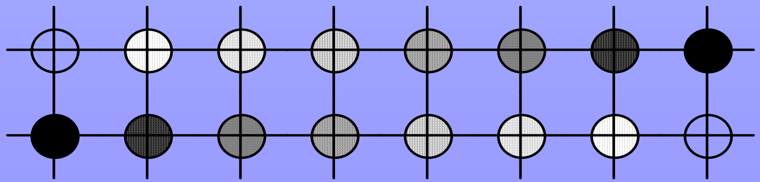
The aliasing problem

- Aliasing is caused by finite addressability of the display.
- Approximation of lines and circles with discrete points often gives a staircase appearance or "Jaggies".



Antialiasing - solutions

- Aliasing can be smoothed out by using higher addressability.
- If addressability is fixed but intensity is variable, use the intensity to control the address of a "virtual pixel". Two adjacent pixels can be be used to give the impression of a point part way between them. The perceived location of the point is dependent upon the ratio of the intensities used at each. The impression of a pixel located halfway between two addressable points can be given by having two adjacent pixels at half intensity.
- An antialiased line has a series of virtual pixels each located at the proper address.

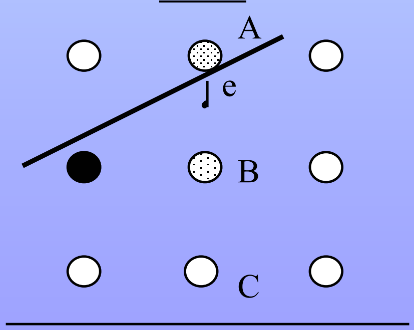


Aliasing / Antialiasing Examples

Antialiased Bresenham Lines

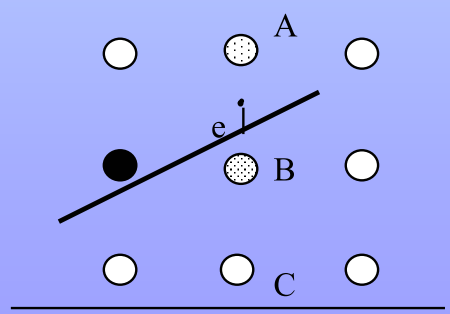
- Line drawing algorithms such as Bresenham's can easily be modified to implement virtual pixels. We use the distance ($e = d_i/a$) value to determine pixel intensities.
- Three possible cases which occur during the Bresenham algorithm:

$e > 0$



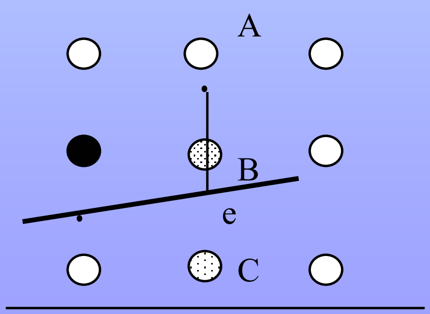
$A = 0.5 + e$
$B = 1 - abs(e+0.5)$
$C = 0$

$0 > e > -0.5$



$A = 0.5 + e$
$B = 1 - abs(e+0.5)$
$C = 0$

$e < -0.5$



$A = 0$
$B = 1 - abs(e+0.5)$
$C = -0.5 - e$

Line Rendering References

Bresenham, J.E., "Ambiguities In Incremental Line Rastering," IEEE Computer Graphics And Applications, Vol. 7, No. 5, May 1987.

Eckland, Eric, "Improved Techniques For Optimising Iterative Decision-Variable Algorithms, Drawing Anti-Aliased Lines Quickly And Creating Easy To Use Color Charts," CSC 462 Project Report, Department of Computer Science, North Carolina State University (Spring 1987).

Foley, J.D. and A. Van Dam, Fundamentals of Interactive Computer Graphics, Addison-Wesley 1982.

Newman, W.M and R.F. Sproull, Principles Of Interactive Computer Graphics, McGraw-Hill, 1979.