TA (Amy Platt) office hours

M and W, 1-2
GS 929b

CRT Displays

- Phosphors glow when hit by electron beam.
- Color is adjusted via intensity of beam delivered to each of R.G, and B phosphor.
- CRT display phosphors glow for limited time—need to be refreshed.
- Raster displays refresh by scanning from top to bottom in left right order.
- Timing is used to make screen elements correspond to memory elements.

CRT Displays

- Typical refresh rate is 75 per second.
- May have many phosphor dots corresponding to one memory element (old stuff), but more usually one per phosphor trio.
- Memory elements called pixels.
- Refresh method creates architectural and programming issues (e.g. double buffering), defines "real-time" in animation.

[Diagram of CRT display]

[Diagram of CRT display components: Timing, Addressing, Frame Buffer, D/A converter, Electron Gun Intensity, CRT]
Flat Panel TFT* Displays

*Thin film transistor

3D displays

Use some scheme to control what each eye sees
Color, temporal + shutter glasses, polarization + glasses

OpenGL and GLUT

Demo and discussion of example program

http://www.cs.arizona.edu/classes/cs433/fall05/triangle.c
OpenGL and GLUT

• Layer between your program and lower levels (hardware, low level display issues)
• Provides primitives
  - points
  - lines
  - polygons
  - bitmaps, fonts
• Provides standard graphics facilities
  - We will learn how some of these work. Some assignments will therefore have some routines "out of bounds"
  - GLUT simplifies interactive program development with intuitive callbacks and additional facilities (menus, window management).

OpenGL and GLUT

• Initialization code from the example

  /* initialize glut system */
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
  glutInitWindowSize(400,500);
  win = glutCreateWindow("Triangle");
  /* create window */

  /* set background to black */
  glClearColor((GLclampf)0.0,(GLclampf)0.0,(GLclampf)0.0,(GLclampf)0.0);
  gluOrtho2D(0.0,400.0,0.0,500.0);
  /* how object is mapped to window */

  static void display_CB(void)
  {
    glutDisplayFunc(display_CB);
    glClear(GL_COLOR_BUFFER_BIT); /* clear the display */
    /* set current color */
    glColor3d(triangle_red, triangle_green, triangle_blue);
    /* draw filled triangle */
    glBegin(GL_POLYGON);
    /* specify each vertex of triangle */
    glVertex2i(200 + displacement_x, 125 - displacement_y);
    glVertex2i(100 + displacement_x, 375 - displacement_y);
    glVertex2i(300 + displacement_x, 375 - displacement_y);
    glEnd(); /* OpenGL draws the filled triangle */
    glFlush(); /* Complete any pending operations */
  }

  static void mouse_move_CB(int x, int y)
  {
    /* See example on-line for sample code. */
  }

  static void key澂assigned char key, int x, int y)
  {
    if(key == 'q') exit(0);

    /* Function called on mouse click */
    static void mouse_click(int button, int state, int x, int y) {
      /* Code which responds to the button, the state (press, release), and where
       * the pointer was when the mouse event occurred (x, y). */
      /* See example on-line for sample code. */
    }

    /* Function called on mouse move while depressed. */
    static void mouse_move_click(int x, int y) {
      /* See example on-line for sample code. */
    }

  static void display_CB(void)
  {
    glClear(GL_COLOR_BUFFER_BIT); /* clear the display */
    /* set current color */
    glColor3d(triangle_red, triangle_green, triangle_blue);
    /* draw filled triangle */
    glBegin(GL_POLYGON);
    /* specify each vertex of triangle */
    glVertex2i(200 + displacement_x, 125 - displacement_y);
    glVertex2i(100 + displacement_x, 375 - displacement_y);
    glVertex2i(300 + displacement_x, 375 - displacement_y);
    glEnd(); /* OpenGL draws the filled triangle */
    glFlush(); /* Complete any pending operations */
  }
OpenGL and GLUT

• GLUT makes pop-up menus easy. We will save development time by using (perhaps abusing) this facility.

/* Create a menu which is accessed by the right button. */
submenu = glutCreateMenu(select_triangle_color);
glutAddMenuEntry("Red", KJB_RED);
glutAddMenuEntry("Green", KJB_GREEN);
glutAddMenuEntry("Blue", KJB_BLUE);
glutAddMenuEntry("White", KJB_WHITE);
glutCreateMenu(add_object_CB);
glutAddMenuEntry("Triangle", KJB_TRIANGLE);
glutAddMenuEntry("Square", KJB_SQUARE);
glutAddSubMenu("Color", submenu);
glutAttachMenu(GLUT_RIGHT_BUTTON);

Ready for the user!

/* start processing events... */
glutMainLoop();

For the rest of the code see http://www.cs.arizona.edu/classes/cs433/fall05/triangle.c

Displaying lines

• Assume for now:
  – lines have integer vertices
  – lines all lie within the displayable region of the frame buffer
• Other algorithms will take care of these issues.

Variety of naive (poor) algorithms:
  – step x, compute new y at each step by equation, rounding
  – step x, compute new y at each step by adding m to old y, rounding

Bresenham’s algorithm

[ H&B, pp 95-99]

• Plot the pixel whose y-value is closest to the line
• Given \((x_k, y_k)\), must choose from either \((x_k+1, y_k+m)\) or \((x_k+1, y_k)\)—recall we are working on case \(0<m<1\)
• Idea: compute value that will determine this choice that is easy to update and cheap to compute (no floating point operations if endpoints are integral).
Bresenham’s algorithm

- Determiner is $d_1 - d_2$
  
  $d_1 - d_2 < 0 \Rightarrow \text{plot at } y_k$ (same level as previous)
  
  $d_1 - d_2 \geq 0 \Rightarrow \text{plot at } y_{k+1}$

From the previous slide

$$d_1 \triangleleft d_2 = 2m(x_k + 1) \triangleleft 2y_k + 2b \triangleleft 1$$

Recall that,

$$m = (y_{\text{end}} - y_{\text{start}}) / (x_{\text{end}} - x_{\text{start}}) = dy / dx$$

So, for integral endpoints we can avoid floating point if we scale by a factor of $dx$. Use determiner $P_k$.

$$P_k = (d_1 \triangleleft d_2) dx$$

\[
= (2m(x_k + 1) \triangleleft 2y_k + 2b \triangleleft 1) dx \\
= 2(x_k + 1) dy \triangleleft 2y_k (dx) + 2b(dx) \triangleleft dx \\
= 2(x_k dy) \triangleleft 2y_k (dx) + \text{constant}
\]

Avoiding Floating Point

Incremental Update

From previous slide

$$P_k = 2(x_k dy) \triangleleft 2y_k (dx) + \text{constant}$$

Finally, express the next determiner in terms of the previous, and in terms of the decision on the next $y$.

$$P_{k+1} = 2(x_k + 1) dy \triangleleft 2y_{k+1} (dx) + \text{constant}$$

$$= P_k + 2 dy \triangleleft 2(y_{k+1} - y_k)$$

Either 1 or 0 depending on decision on $y$.
Bresenham algorithm

- \( p_{k+1} = p_k + 2 \ dy - 2 \ dx (y_{k+1}, y_k) \)
- Exercise: check that \( p_0 = 2 \ dy - dx \)
- Algorithm (for the case that 0<m<1):
  - \( x=x_\text{start}, y=y_\text{start}, p=2 \ dy - dx, \text{mark} (x, y) \)
  - until \( x=x_\text{end} \)
  - \( p<0 ? \ y=y+1, \text{mark} (x, y), \text{pp=2dy-2dx} \)
  - else \( y=y, \text{mark} (x, y), \text{pp=2dy} \)
- Some calculations can be done once and cached.

Issues

- End points may not be integral due to clipping (or other reasons)
- Brightness is a function of slope.
- Discretization problems “aliasing” (related to previous point).

Line drawing--simple line (Bresenham) brightness issues

8 pixels per 8*sqrt(2) length

8 pixels for 8 length
(Brighter)

Line drawing--discretization artifacts (often called aliasing)