Computer Graphics

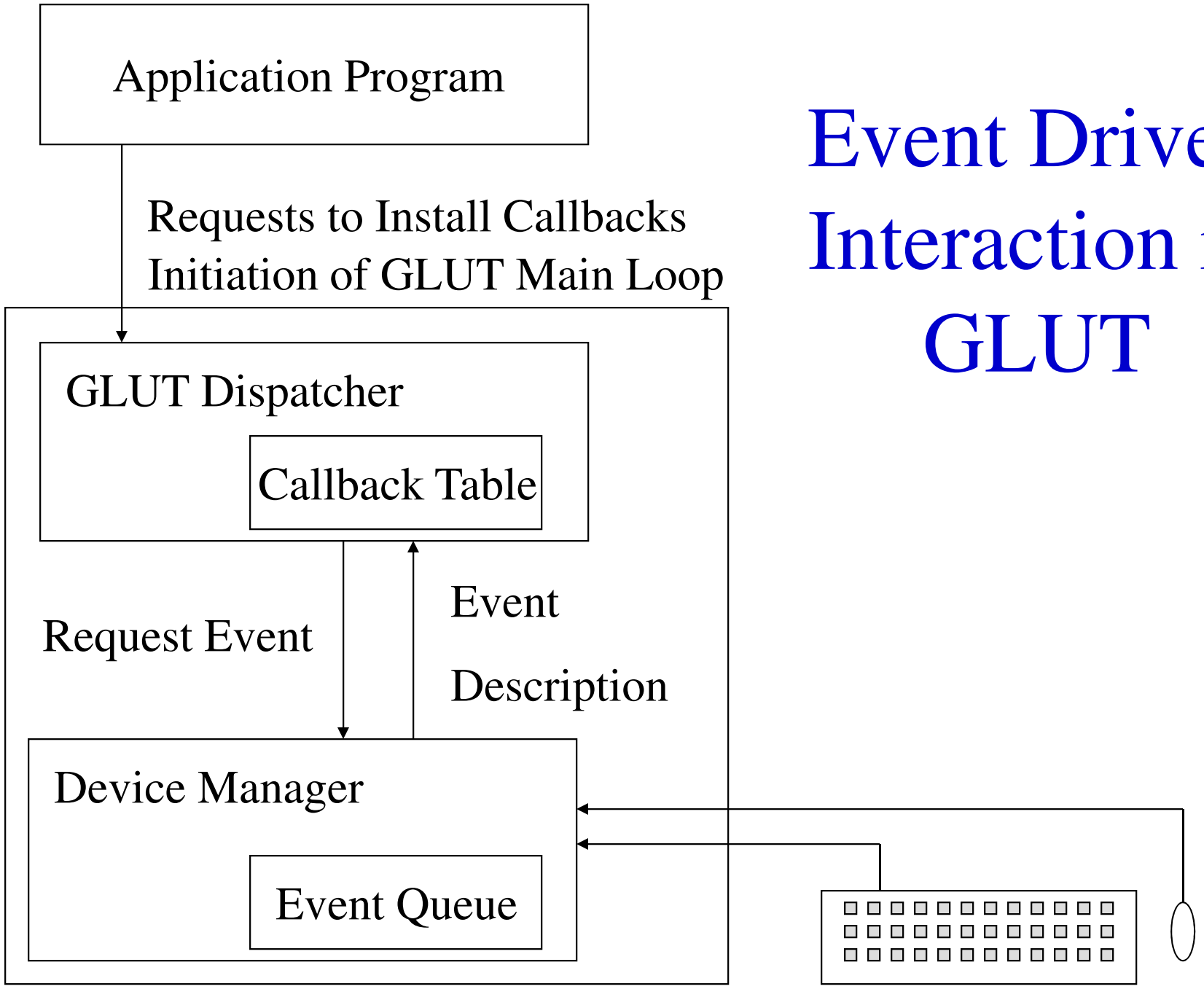
Tom Ellman

Lecture 2

Event Driven Interaction in GLUT

- Application Program:
  - Asks GLUT to install functions into a “Callback Table”.
  - Initiates the GLUT main loop, which runs until the program terminates.
- Device Manager:
  - Is activated by interrupts from the mouse or keyboard.
  - Responds to interrupts by putting an event description in the event queue.
- GLUT Dispatcher Repeatedly:
  - Asks Device Manager for an event description.
  - Uses a function in the Callback Table to process the event.

Event Driven Interaction in GLUT



Callback Table

EVENT TYPE	CALLBACK
REDISPLAY	
MOUSE	
KEYBOARD	
RESHAPE	
...	
IDLE	

GLUT Main Loop (Simplified)

Repeat Forever:

If Empty?(EventQueue)

Then Invoke function CallbackTable[IDLE].

Else 1. Let Event = Dequeue(EventQueue).

2. Let Type = Type field of Event.

3. Let Parms = Parameters field of Event.

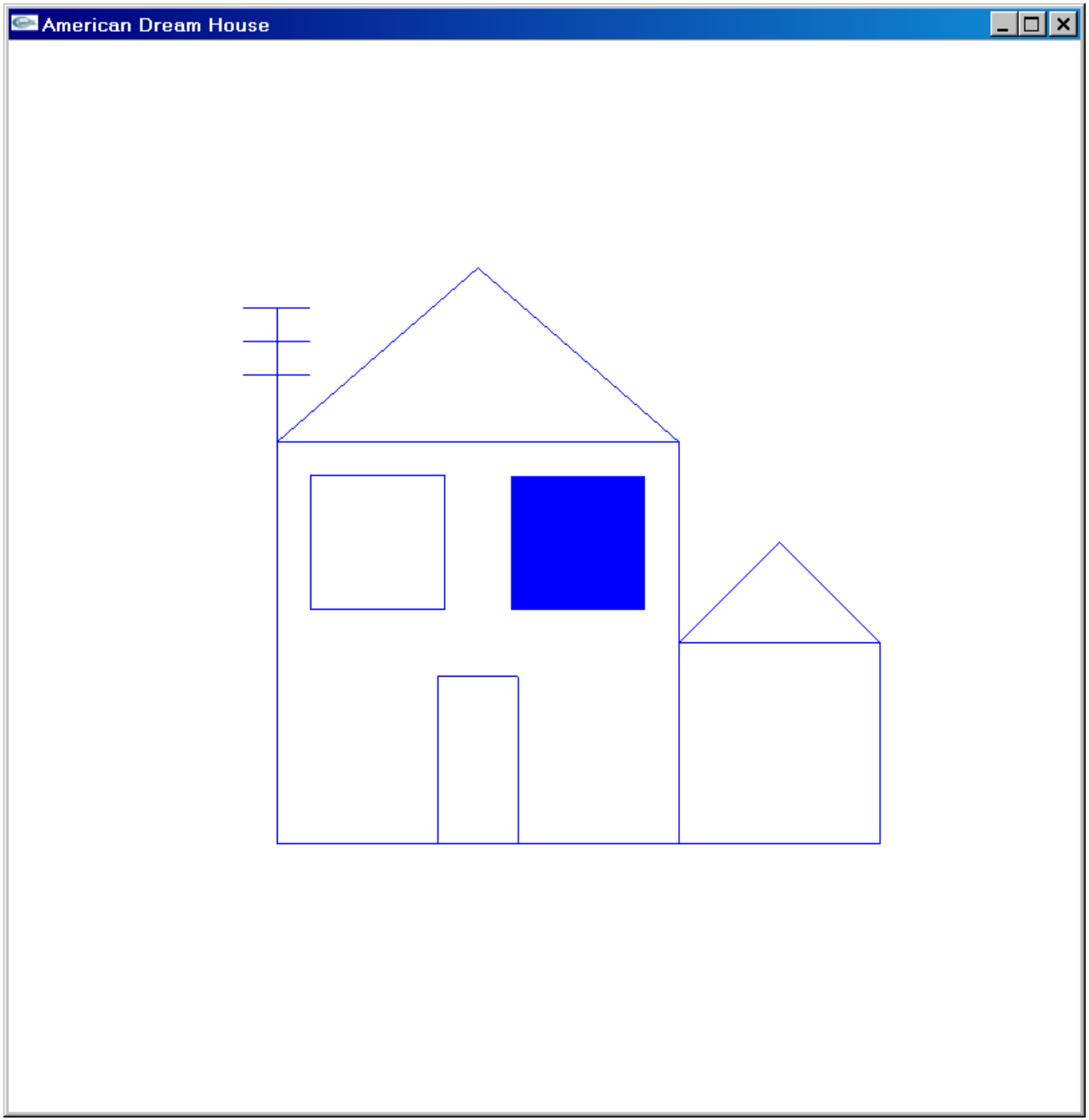
4. Invoke function CallbackTable[Type]

with Parms as parameters.

Program for Drawing a House

- Create a window.
- Define callbacks:
  - Redisplaying the house image.
  - Reshaping the window.
  - Exiting on keyboard ESC command.
- Initiate the GLUT main loop.

Project: House


American Dream House

```
int main()
{
   // Mask floating point exceptions.
   _control87(MCW_EM,MCW_EM);

  // Choose RGB display mode for normal screen window.
  glutInitDisplayMode(GLUT_RGB);

  // Set initial window size, position, and title.
  glutInitWindowSize( INITIAL_WIN_W, INITIAL_WIN_H );
  glutInitWindowPosition( INITIAL_WIN_X, INITIAL_WIN_Y );
  glutCreateWindow("American Dream House");

  // You don't (yet) want to know what this does.
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity( );
  gluOrtho2D( 0.0, (double) INITIAL_WIN_W, 0.0, (double) INITIAL_WIN_H );
  glMatrixMode( GL_MODELVIEW );
  glLoadIdentity( );

  // This is a hack.
  glTranslatef( 0.375, 0.375, 0.0 );

  // ... Continued on next slide ...

}
```

```
int main()
{
  // ... Continued from previous slide ...

  // Set the color for clearing the normal screen window.
  glClearColor( 1.0, 1.0, 1.0, 0.0 );

  // Set the color for drawing the house.
  glColor3f(0.0, 0.0, 1.0);

  // Set the callbacks for the normal screen window.
  glutDisplayFunc( drawHouse );

  // Set the callback for reshape events.
  glutReshapeFunc( reshape );

  // Set the callback for keyboard events.
  glutKeyboardFunc( escExit );

  // Start the GLUT main loop.
  glutMainLoop( );
}
```

Display Callback

```
void drawHouse( )
{
  // Clear the window.
  glClear(GL_COLOR_BUFFER_BIT);

  // Draw the base of the house.
  glBegin( GL_LINE_LOOP );
  glVertex2i( 200, 200 );  glVertex2i( 200, 500 );
  glVertex2i( 500, 500 );  glVertex2i( 500, 200 );
  glEnd( );

  // Draw the roof of the house.
  glBegin( GL_LINE_STRIP );
  glVertex2i( 200, 500 );
  glVertex2i( 350, 630 );
  glVertex2i( 500, 500 );
  glEnd( );

  // … Continued on next slide …
}
```

```
void drawHouse( )
{
  // … Continued from previous slide …

  // Draw the base of the garage.
  glBegin( GL_LINE_LOOP );
  glVertex2i( 500, 200 );
  glVertex2i( 500, 350 );
  glVertex2i( 650, 350 );
  glVertex2i( 650, 200 );
  glEnd( );

  // Draw the roof of the garage.
  glBegin( GL_LINE_STRIP );
  glVertex2i( 500, 350 );
  glVertex2i( 575, 425 );
  glVertex2i( 650, 350 );
  glEnd( );

  // … Continued on next slide …

}
```

```
void drawHouse( )
{

  // … Continued from previous slide …

  // Draw the door of the house.
  glBegin( GL_POINTS );
  for (int i=0; i<125; i++) glVertex2i(320,200+i);
  for (int i=0; i<60; i++) glVertex2i(320+i,325);
  for (int i=0; i<125; i++) glVertex2i(380,200+i);
  glEnd( );

  // Draw the antenna on the house.
  glBegin( GL_LINES );
  glVertex2i( 200, 500 );  glVertex2i( 200, 600 );
  glVertex2i( 175, 600 );  glVertex2i( 225, 600 );
  glVertex2i( 175, 575 );  glVertex2i( 225, 575 );
  glVertex2i( 175, 550 );  glVertex2i( 225, 550 );
  glEnd( );

}
```

```
void drawHouse( )
{

  // … Continued from previous slide …

  // Set polygon fill/line mode.
  glPolygonMode(GL_FRONT,GL_FILL);
  glPolygonMode(GL_BACK,GL_LINE);

  // Draw a window clockwise in line mode.
  glBegin(GL_POLYGON);
  glVertex2i( 225, 375 );
  glVertex2i( 225, 475 );
  glVertex2i( 325, 475 );
  glVertex2i( 325, 375 );
  glEnd();

  // Draw a window counter-clockwise in fill mode
  glBegin(GL_POLYGON);
  glVertex2i( 375, 375 );
  glVertex2i( 475, 375 );
  glVertex2i( 475, 475 );
  glVertex2i( 375, 475 );
  glEnd();

}
```

The Reshape Event Callback

```
void reshape(int w, int h)
// Callback for processing reshape events.
{
  glViewport(0,0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0.0, w, 0.0, h);
}
```

This callback sets up OpenGL to display images in a window of the new size. The value **`w`** is the new width. The value **`h`** is the new height.

The Keyboard Event Callback

```
void escExit( GLubyte key, int, int )
{
  if ( key == 27 /* ESC */ )
  exit( 0 );
}
```

This callback simply causes the program to exit when the user hits the ESC key.

Specifying Vertices in OpenGL

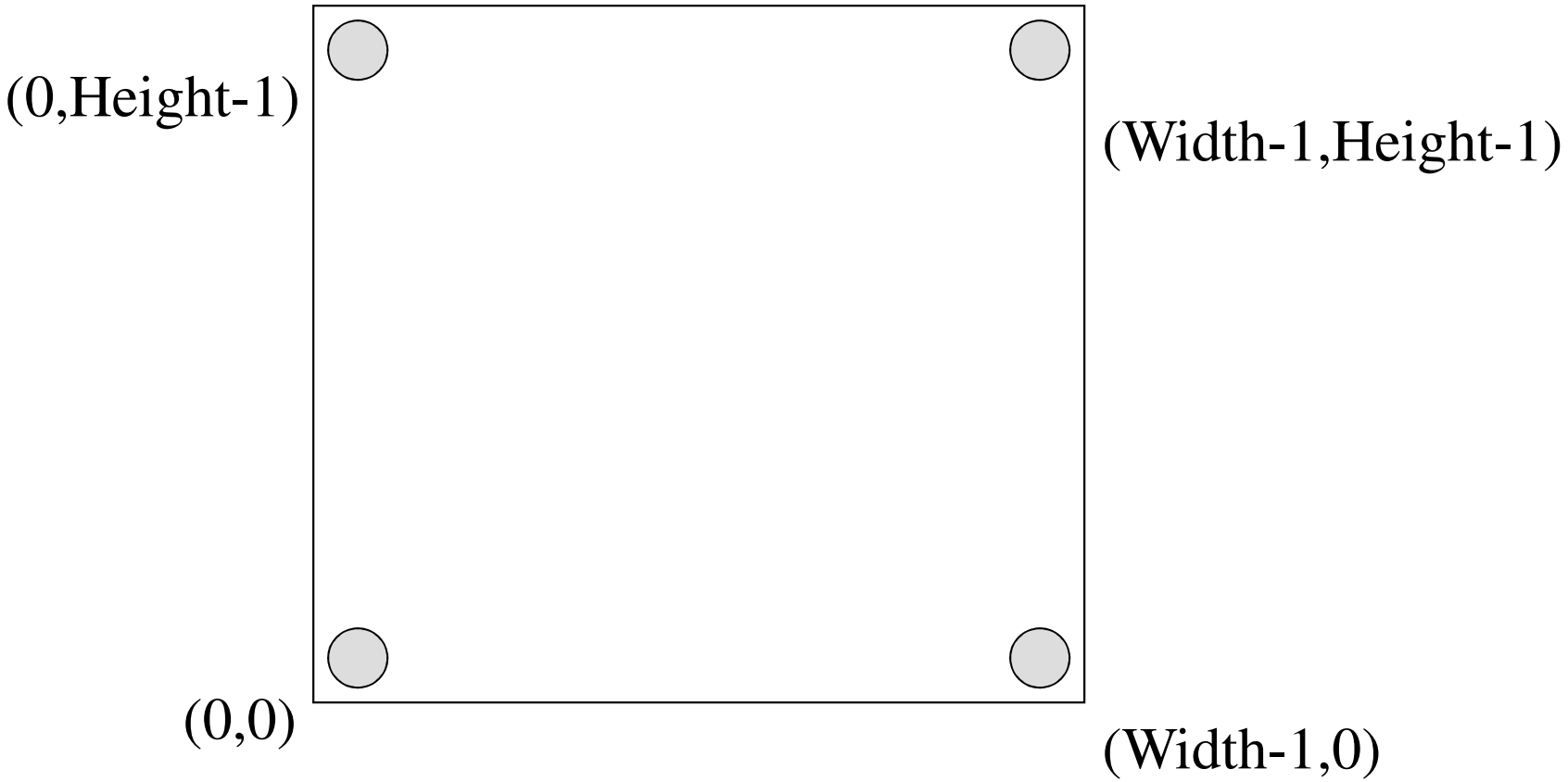
```
void glVertex2i(int x, int y);

void glVertex2f(float x, float y);

void glVertex2d(double x, double y);
```

The commands differ in the type of data used to specify vertex coordinates.

GLUT Coordinates Versus OpenGL Coordinates

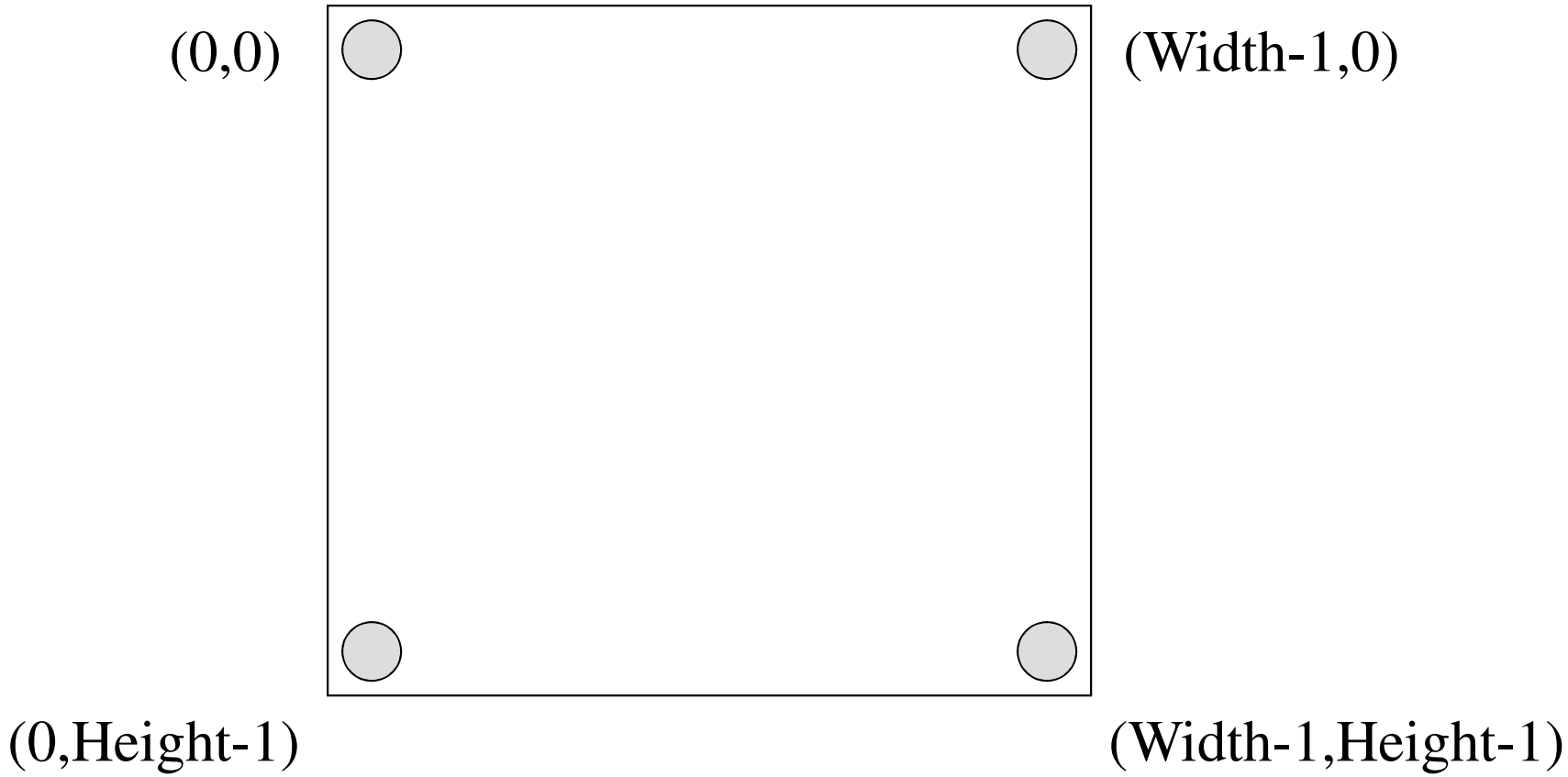
- In OpenGL, the vertical coordinate increases moving from the bottom of the screen toward the top of the screen. Zero is at the bottom.
- In GLUT, the vertical coordinate increases moving from the top of the the screen toward the bottom of the screen. Zero is at the top.

Addressing Pixels in OpenGL



Width and Height are set by the **glutInitWindowSize** function.

Addressing Pixels in GLUT



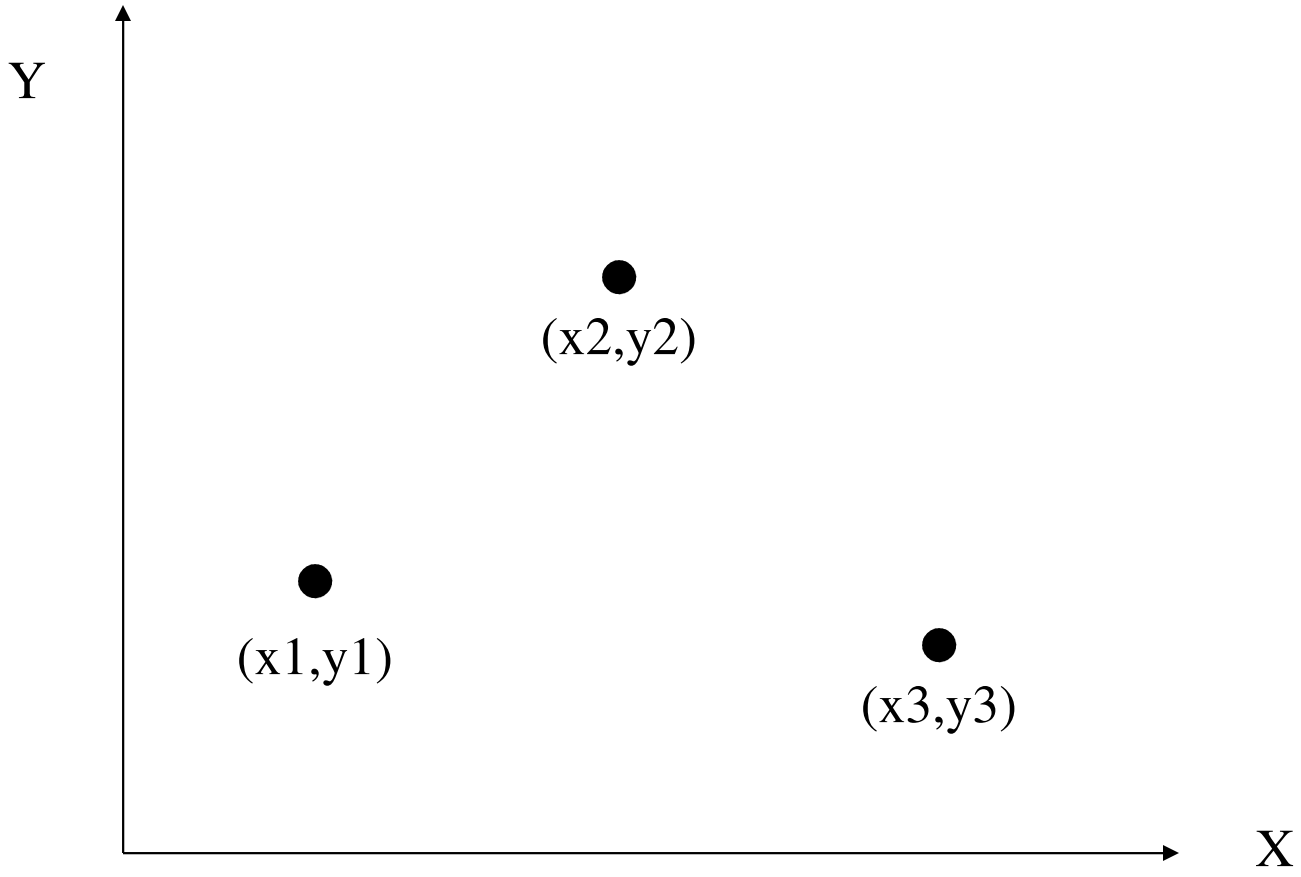
Width and Height are set by the **glutInitWindowSize** function.

Drawing Points in OpenGL

```
glBegin( GL_POINTS );
            ...
  glVertex2i(x1,y1);
  glVertex2i(x2,y2);
  glVertex2i(x3,y3);
            ...
glEnd( );
```

Each **`glVertex2i`** command specifies a single point. In this example, points are drawn at (x1,y1), (x2,y2) and (x3,y3).

Drawing Points in OpenGL

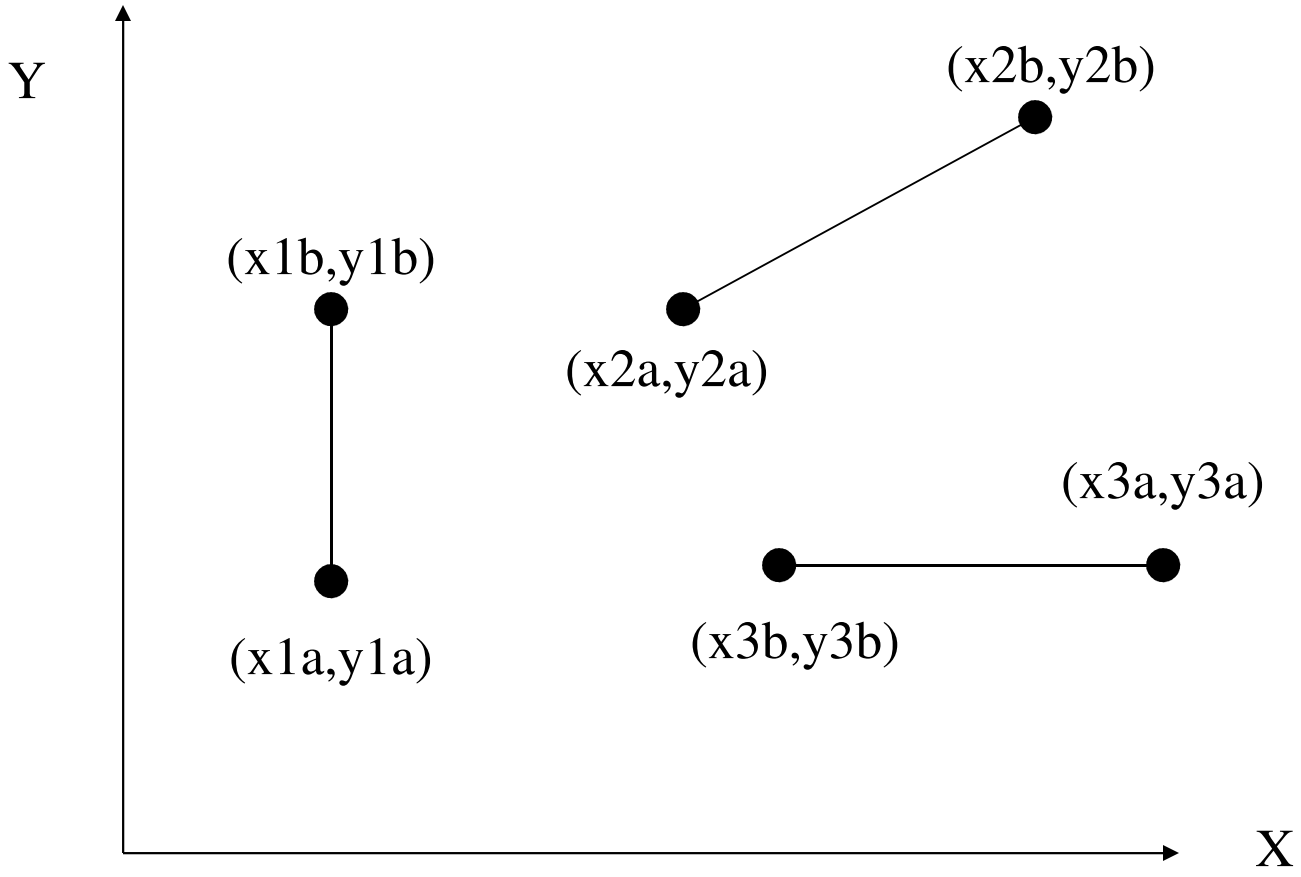


Drawing Lines in OpenGL

```
glBegin( GL_LINES );
            ...
 glVertex2i(x1a,y1a); glVertex2i(x1b,y1b);
 glVertex2i(x2a,y2a); glVertex2i(x2b,y2b);
 glVertex2i(x3a,y3a); glVertex2i(x3b,y3b);
            ...
glEnd( );
```

Each successive pair of **`glVertex2i`** commands specifies a single line segment. In this case, three line segments are drawn: (1) From (x1a,y1a) to (x1b,y1b); (2) From (x2a,y2a) to (x2b,y2b); (3) From (x3a,y3a) to (x3b,y3b).

Drawing Lines in OpenGL

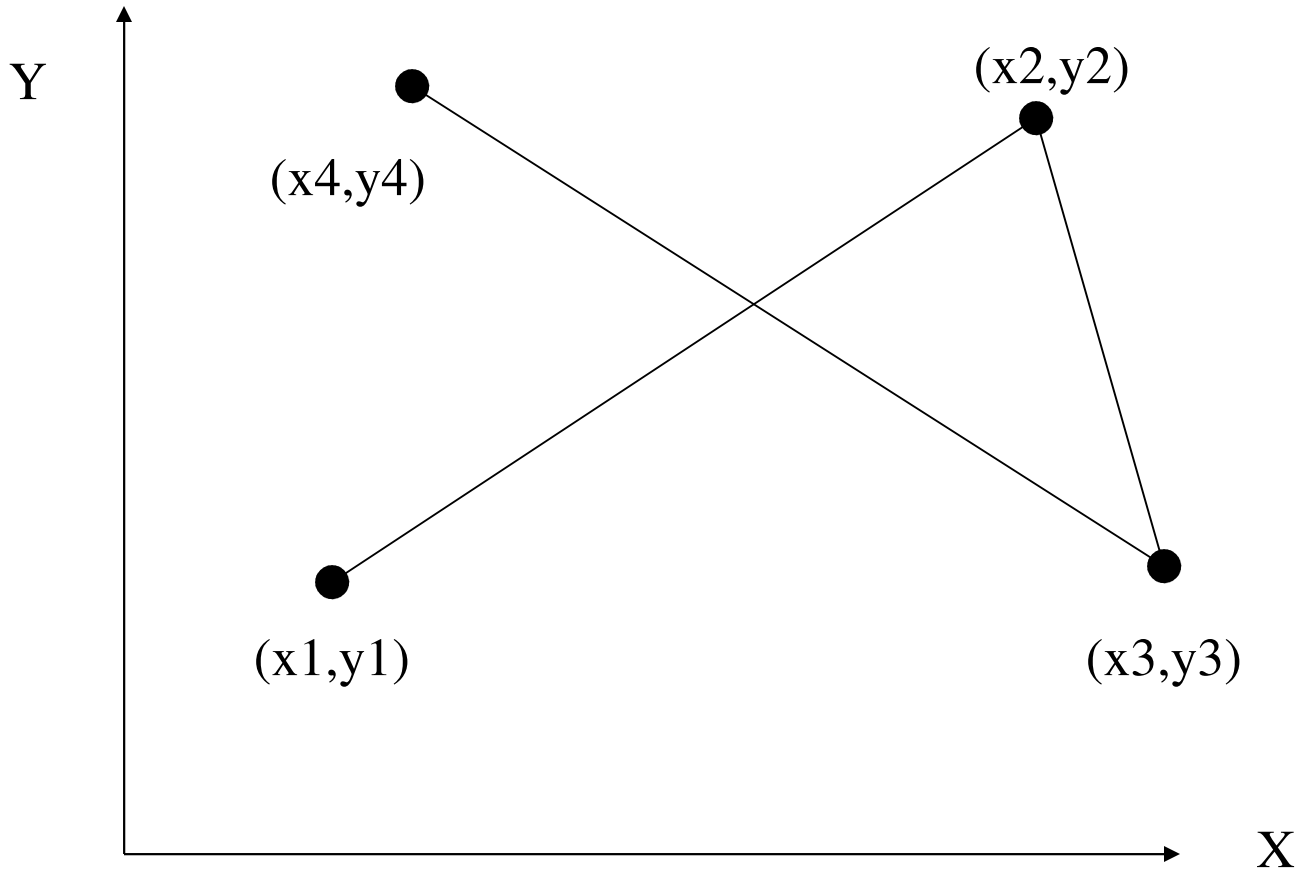


Drawing a Sequence of Connected Lines in OpenGL

```
glBegin( GL_LINE_STRIP );
            ...
 glVertex2i(x1,y1);
 glVertex2i(x2,y2);
 glVertex2i(x3,y3);
 glVertex2i(x4,y4);
            ...
glEnd( );
```

Each successive **`glVertex2i`** command specifies the next endpoint in a connected series of line segments. In this case, three line segments are drawn: (1) From (x1,y1) to (x1,y2); (2) From (x2,y2) to (x3,y3); (3) From (x3,y3) to (x4,y4).

Drawing a Sequence of Connected Lines in OpenGL

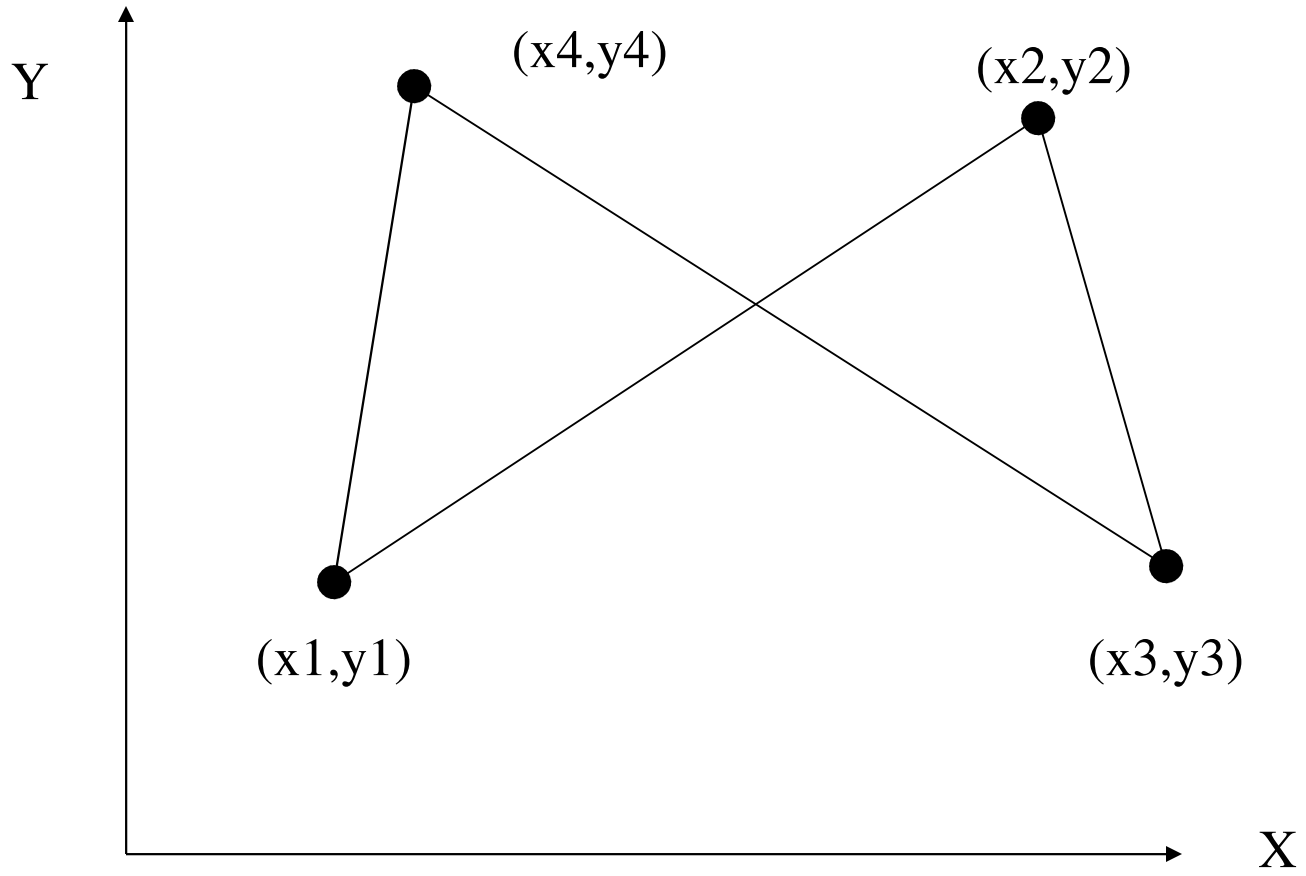


Drawing a Closed Sequence of Connected Lines in OpenGL

```
glBegin( GL_LINE_LOOP );
            ...
 glVertex2i(x1,y1);
 glVertex2i(x2,y2);
 glVertex2i(x3,y3);
 glVertex2i(x4,y4);
            ...
glEnd( );
```

Each successive **`glVertex2i`** command specifies the next endpoint in a connected, closed series of line segments. In this case, four line segments are drawn: (1) From (x1,y1) to (x1,y2); (2) From (x2,y2) to (x3,y3); (3) From (x3,y3) to (x4,y4); From (x4,y4) to (x1,y1).

Drawing a Closed Sequence of Connected Lines in OpenGL

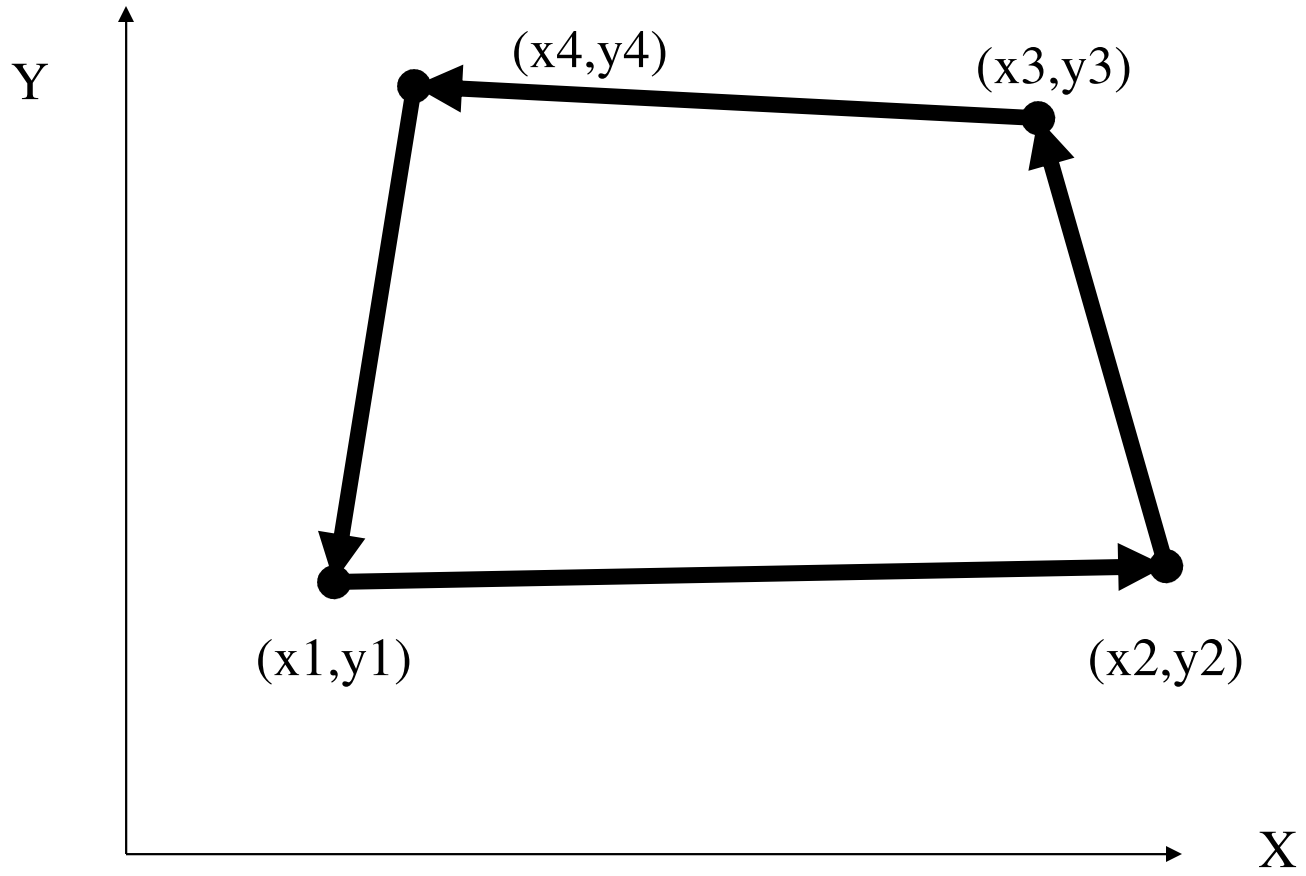


Drawing a Forward-Facing Polygon in OpenGL

```
glBegin( GL_POLYGON );
            ...
 glVertex2i(x1,y1);
 glVertex2i(x2,y2);
 glVertex2i(x3,y3);
 glVertex2i(x4,y4);
            ...
glEnd( );
```

Each successive **`glVertex2i`** command specifies the next endpoint in a connected, closed series of line segments that describe a convex polygon in counter-clockwise vertex-order. The polygon faces toward the viewer.

Drawing a Forward-Facing Polygon in OpenGL

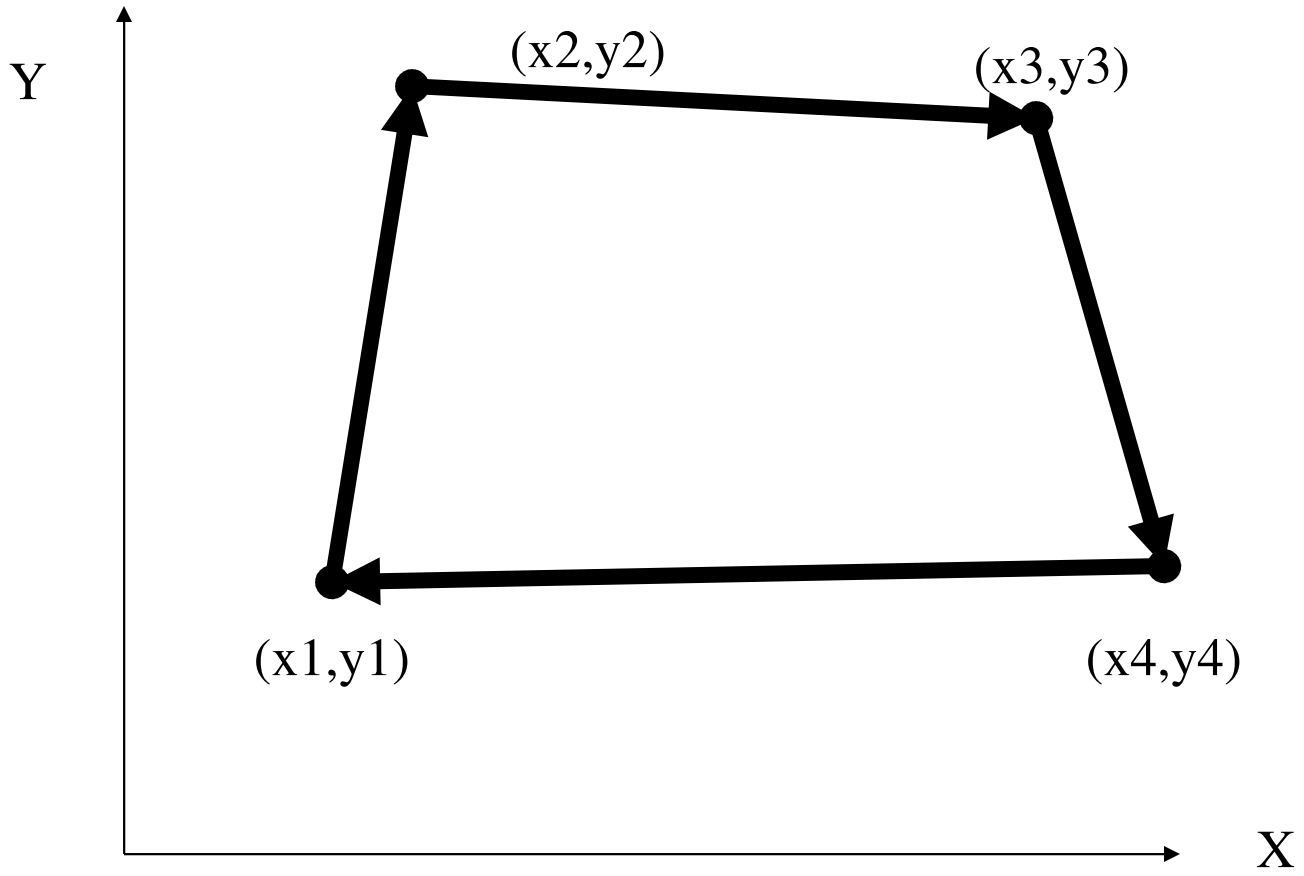


Drawing a Backward-Facing Polygon in OpenGL

```
glBegin( GL_POLYGON );
            ...
 glVertex2i(x1,y1);
 glVertex2i(x2,y2);
 glVertex2i(x3,y3);
 glVertex2i(x4,y4);
            ...
glEnd( );
```

Each successive **`glVertex2i`** command specifies the next endpoint in a connected, closed series of line segments that describe a convex polygon in clockwise vertex-order. The polygon faces away from the viewer.

Drawing a Backward-Facing Polygon in OpenGL



Filled v. UnFilled Polygons

Specify that forward facing polygons are filled:

```
glPolygonMode(GL_FRONT,GL_FILL);
```

Specify that backward facing polygons are outlined:

```
glPolygonMode(GL_BACK,GL_LINE);
```

Rotating Rectangles Program

- Middle mouse button brings up a menu:
  - DRAW RECTANGLE: User may draw a rectangle using the mouse.
  - ANIMATE: Starts rotation of rectangle. User may stop rotation with a left or right mouse click.
  - EXIT.
- Program uses the idle callback to animate the rectangle rotation.
- Program uses the mouse motion callback to draw rubber bands indicating the user's evolving rectangle.

Project: Rotating-Rectangles

Callbacks for Rotating Rectangles Program

```
glutDisplayFunc( drawRectangle );
glutMouseFunc( mouse );
glutMotionFunc( rubberBand );
glutReshapeFunc( reshape );
glutKeyboardFunc( escExit );
glutIdleFunc(NULL);
```

```
void setMenu( )
// Routine for creating menus.
{
  glutCreateMenu( mainMenu );
  glutAddMenuEntry( "Draw Rectangle", 0 );
  glutAddMenuEntry( "Animate", 1 );
  glutAddMenuEntry( "Exit", 2 );
  glutAttachMenu( GLUT_MIDDLE_BUTTON );
}
```

Create a menu to be processed by the **`mainMenu`** function. Give it three menu items. Attach the menu to the middle mouse button.

```
void mainMenu( int item )
// Callback for processing main menu.
{
  switch ( item )
  {
    case 0 : drawing = true;
             animating = false;
             glutIdleFunc(NULL);
             break;
    case 1 : animating = true;
             drawing = false;
             glutIdleFunc(spinDisplay);
             break;
    case 2 : exit(0);
  }
}
```

The **`mainMenu`** function is passed an integer indicating the menu item that was selected.

The Display Callback

```
void drawRectangle()
// Callback for drawing the rectangle at its current orientation.
{
  glClear(GL_COLOR_BUFFER_BIT);
  glPushMatrix();
  glTranslatef(rectCenterX,rectCenterY,0.0);
  glRotatef(rectTheta, 0.0, 0.0, 1.0);
  glColor3f(1.0, 0.0, 0.0);
  glRectf(-rectHalfWidth,-rectHalfHeight,rectHalfWidth, rectHalfHeight);
  glPopMatrix();

  glutSwapBuffers();
}
```

First the hidden buffer is cleared. Then the rectangle is drawn into the hidden buffer. Then the buffers are swapped so that the hidden buffer is displayed.

The IDLE Callback

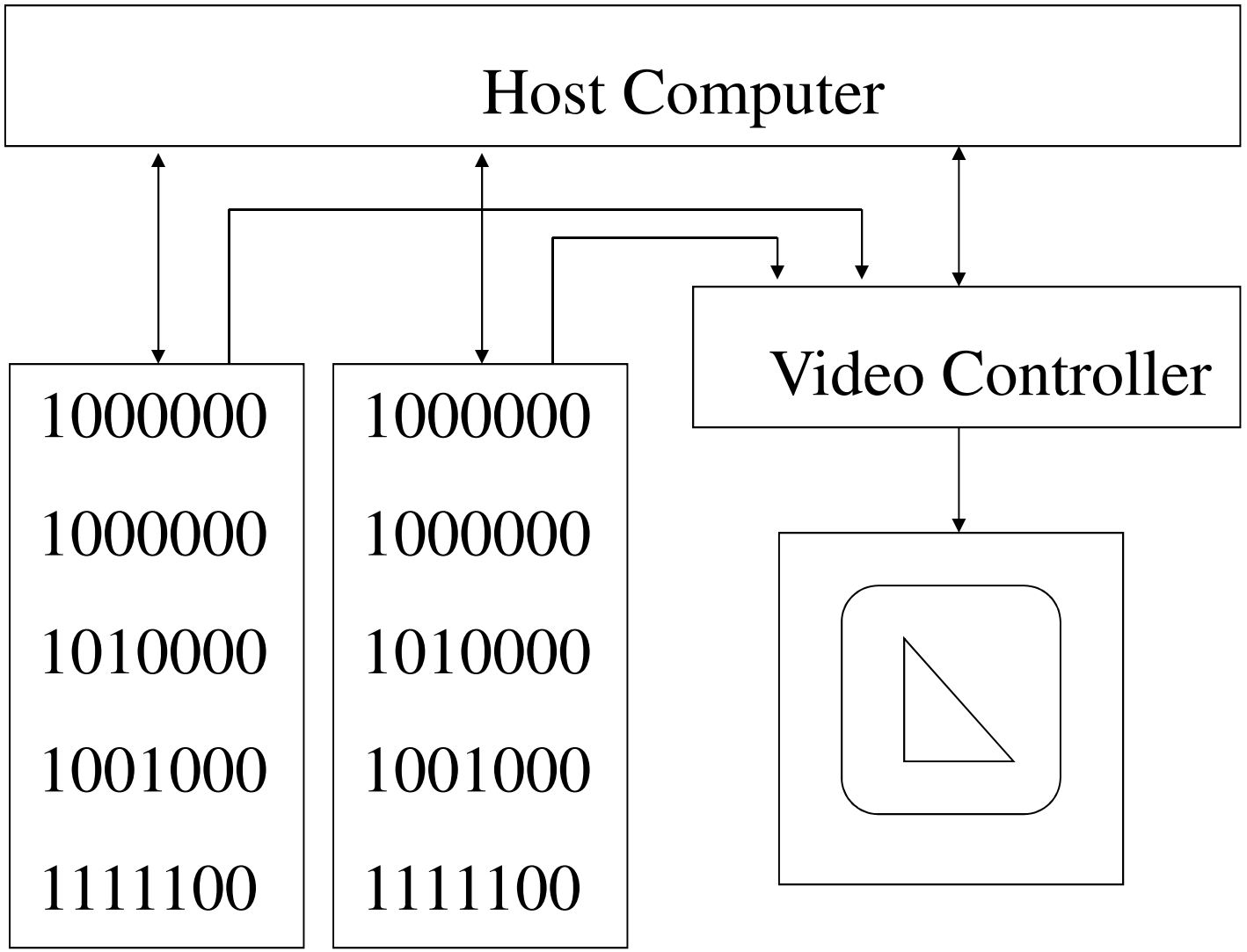
```
void spinDisplay()
{
  clock_t t=clock();
  do {} while( (float)(clock()-t)/CLOCKS_PER_SEC < DELAY);

  rectTheta += 1.0;
  if (rectTheta > 360.0) rectTheta -= 360.0;
  glutPostRedisplay();
}
```

The IDLE callback waits for a short period and then rotates the rectangle by one degree. Then it posts a redisplay event into the event queue, so that the display callback will be called to display the rectangle in its new position, when GLUT regains control.

Double Buffering

- The hardware includes two copies of the refresh buffer.
- Only one buffer is displayed at a time.
- While one buffer is displayed, the user draws a new image in the undisplayed buffer.
- When the new image is ready, the user issues the **`glutSwapBuffers`** command to display the undisplayed buffer.
- Used to implement real-time animation.

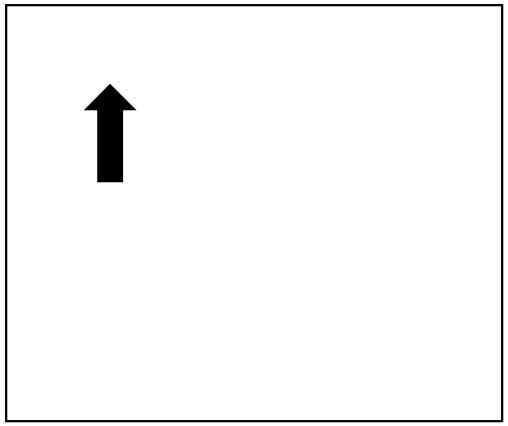
Double Buffer Architecture



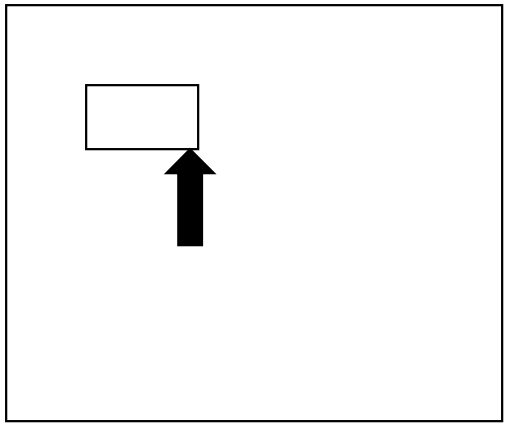
Rubber Bands

- Each time the mouse moves, the motion callback must redraw the rubber band.
- One corner of the rectangle is located at the point where the mouse button was first depressed.
- The other corner is located at the current mouse coordinates.

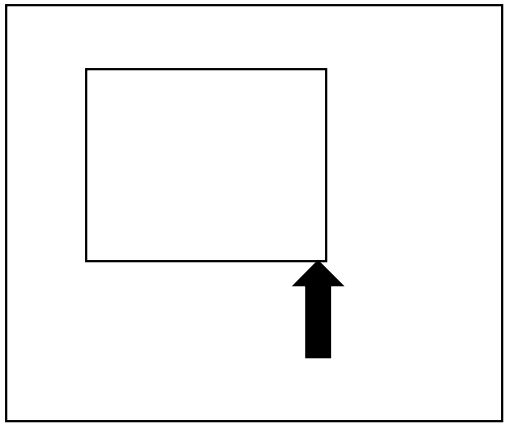
Rubber Bands



Mouse Button
Depressed



Mouse Moves



Mouse Moves

Implementing Rubber Bands

- Brute Force.
- Overlays.
- XOR Drawing Mode.

Brute Force Rubber Bands

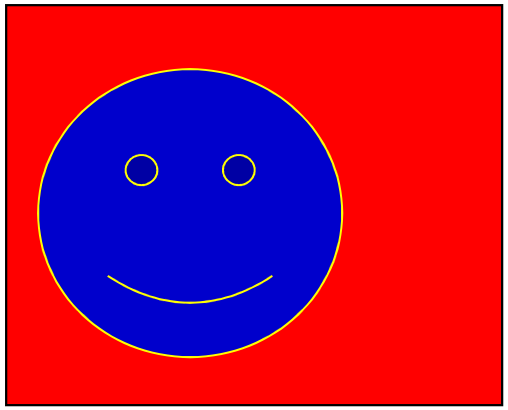
- Save clean image in an array of pixels.
- Each time the mouse moves:
  1. Write the clean image to the frame buffer.
  2. Draw the rubber band in a new position.

Requires very fast hardware for copying the clean image to the frame buffer.

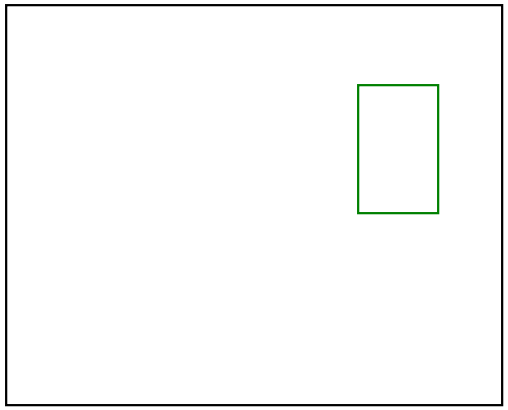
Rubber Bands with Overlays

- Program draws separately in two different layers:
  - The *normal* layer holds the main image.
  - The *overlay* layer holds things that are superimposed on the main image.
- The overlay layer uses a color map M:
  - For each index i, the map entry M[i] is a color.
  - One special index value represents transparency.
- Overlays can be implemented in hardware or software.

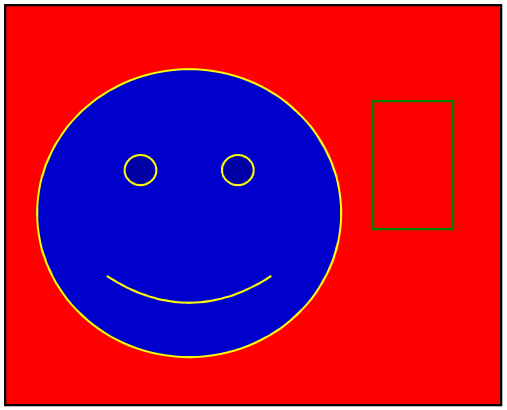
Normal and Overlay Layers



Normal Layer



Overlay Layer



Displayed Image

Sample Color Map

0	Transparent
1	Black
2	Blue
3	Green
4	Cyan
5	Red
6	Magenta
7	Yellow
8	White

XOR Drawing Mode

- New pixel value is combined with current pixel value using logical XOR operation.
- First XOR rendering puts the rubber band on the image.
- Second XOR rendering restores the original image.

Project: Rubber-Bands

XOR Truth Table

N	C	$N \oplus C$
0	0	0
0	1	1
1	0	1
1	1	0

$$N \oplus (N \oplus C) = C$$

This sets **`rubberBand`** to be the callback function for mouse motion when a button is pressed:

```
glutMotionFunc( rubberBand );
```

This sets **`rubberBand`** to be the callback function for mouse motion when a button is *not* pressed:

```
glutPassiveMotionFunc( rubberBand );
```

```
void rubberBand( int x, int y )
// Callback for processing mouse motion.
{
  if ( rubberBanding )
  {
    drawRubberBand(xAnchor,yAnchor,xStretch,yStretch);
    y = windowHeight - y;
    xStretch = x;
    yStretch = y;
    drawRubberBand(xAnchor,yAnchor,xStretch,yStretch);
    glFlush();
  }
}
```

Each time the mouse moves, we erase the old rubber band and then draw the new one.

```
void drawRubberBand(int xA, int yA, int xS, int yS)
{
    glEnable(GL_COLOR_LOGIC_OP);
    glLogicOp(GL_XOR);
    glBegin( GL_LINES );
    glVertex2i( xA, yA );
    glVertex2i( xS, yS );
    glEnd();
    glDisable(GL_COLOR_LOGIC_OP);
    glFlush();
}
```

Before we draw a rubber band, we put the system in XOR drawing mode. After drawing the rubber band, we restore the original drawing mode.

```
void mouse( int button, int state, int x, int y )
 // Function for processing mouse events.
 {
   if ( button == GLUT_LEFT_BUTTON )
     switch ( state )
     {
       case GLUT_DOWN: processLeftDown( x, y ); break;
       case GLUT_UP: processLeftUp( x, y ); break;
     }
 }
```

The mouse click callback function takes the button and the mouse coordinates as parameters. If the event is associated with the left button, the function checks whether the button just went down or just came up, and calls an appropriate routine in either case.

```
void processLeftDown( int x, int y )
 // Function for processing mouse left botton down events.
 {
   if (!rubberBanding)
   {
     int xNew = x;
     int yNew = windowHeight - y;
     xAnchor = xNew;
     yAnchor = yNew;
     xStretch = xNew;
     yStretch = yNew;
     drawRubberBand(xAnchor,yAnchor,xStretch,yStretch);
     rubberBanding = true;
   }
 }
```

When the mouse button goes down, we put up the first rubber band and set a flag indicating that rubber banding is in progress.

```
void processLeftUp( int x, int y )
 // Function for processing mouse left botton up events.
 {
   if (rubberBanding)
   {
     int xNew, yNew;
     drawRubberBand(xAnchor,yAnchor,xStretch,yStretch);
     rubberBanding = false;
     xNew = x;
     yNew = windowHeight - y;
     drawLine(xAnchor, yAnchor, xNew, yNew);
     glFlush();
   }
 }
```

When the button goes up, we erase the rubber band; draw the line (permanently) in its final position and turn off the flag indicating that rubber banding is no longer in progress.

Pixel Level vs. Object Level Graphics Packages

- Pixel level package:
  - Models the application as an array of pixels.
  - Does not support copy, translate, rotate, scale of objects such as lines, circles or polygons.
- Object level package:
  - Models the application in terms of line, circle or polygon commands.
  - Does support copy, translate, rotate, scale of objects such as lines, circles or polygons.