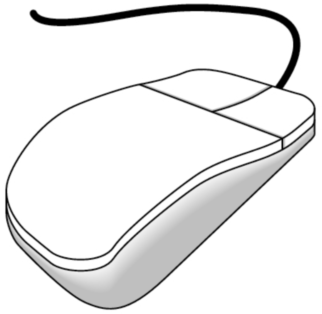
Lecture 4

Interaction / Graphical Devices

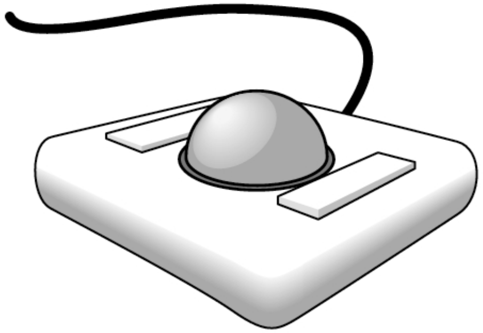
Graphical Input

- Devices can be described either by
  - Physical properties
    - Mouse
    - Keyboard
    - Trackball
  - Logical Properties
    - What is returned to program via API
      - A position
      - An object identifier
- Modes
  - How and when input is obtained
    - Request or event

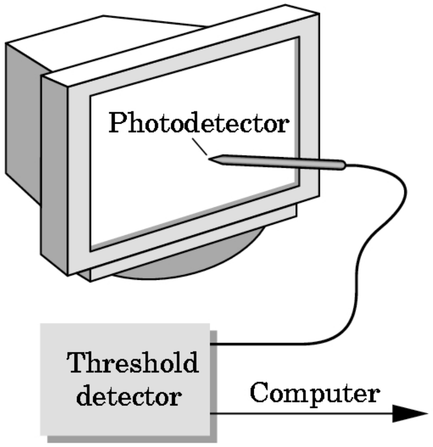
Physical Devices



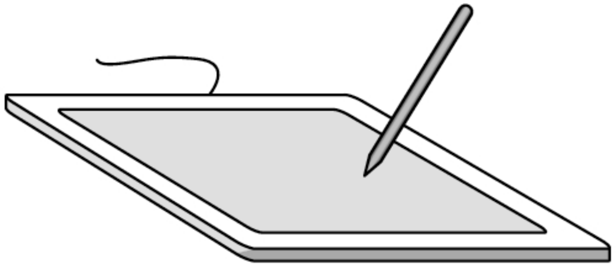
mouse



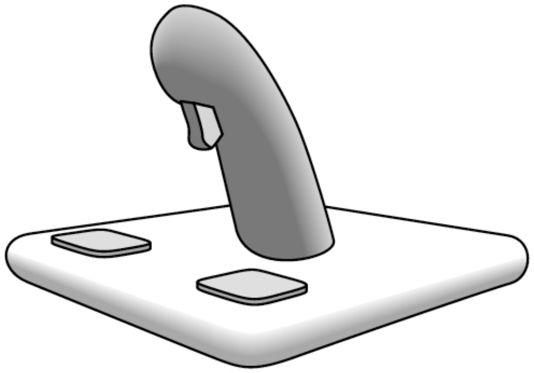
trackball



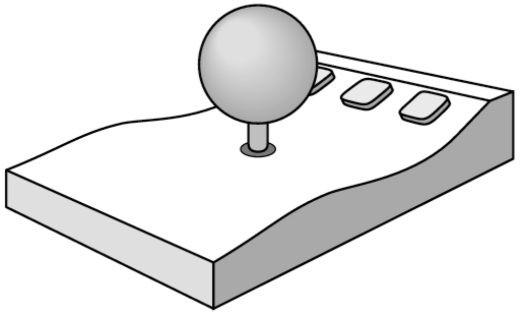
light pen



data tablet



joy stick



space ball

Incremental/Relative Devices

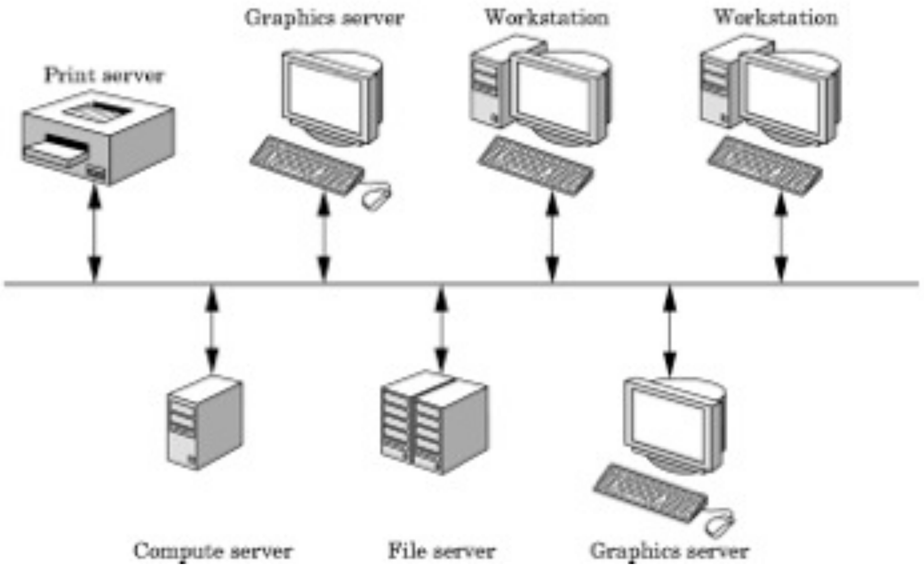
- Devices such as the data tablet return a position directly to the operating system
- Devices such as the mouse, trackball, and joy stick return incremental inputs (or velocities) to the operating system
  - Must integrate these inputs to obtain an absolute position
    - Rotation of cylinders in mouse
    - Roll of trackball
    - Difficult to obtain absolute position
    - Can get variable sensitivity

Logical Devices

- Consider the C and C++ code
  - C++: **`cin >> x;`**
  - C: **`scanf ("%d", &x);`**
- What is the input device?
  - Can't tell from the code
  - Could be keyboard, file, output from another program
- The code provides *logical input*
  - A number (an **`int`**) is returned to the program regardless of the physical device

X Window Input

- The X Window System introduced a client-server model for a network of workstations
  - **Client**: OpenGL program
  - **Graphics Server**: bitmap display with a pointing device and a keyboard

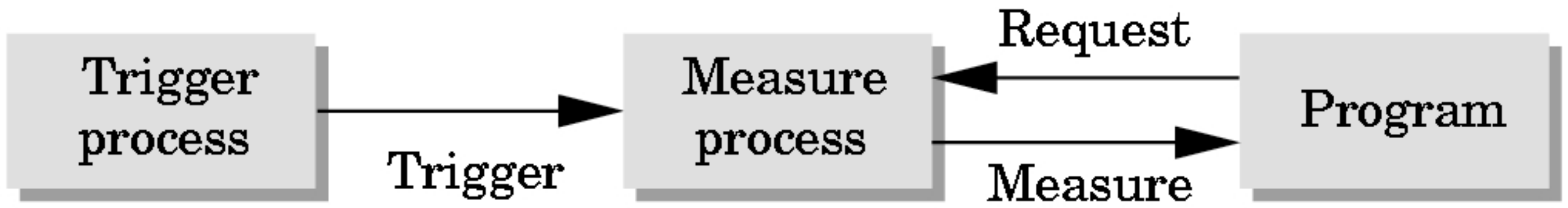


Input Modes

- Input devices contain a *trigger* which can be used to send a signal to the operating system
  - Button on mouse
  - Pressing or releasing a key
- When triggered, input devices return information (their *measure*) to the system
  - Mouse returns position information
  - Keyboard returns ASCII code

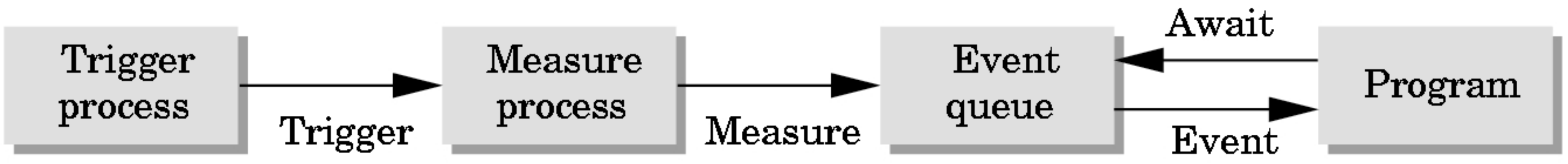
Request Mode

- Input provided to program only when user triggers the device
- Typical of keyboard input
  - Can erase (backspace), edit, correct until enter (return) key (the trigger) is depressed



Event Mode

- Most systems have more than one input device, each of which can be triggered at an arbitrary time by a user
- Each trigger generates an *event* whose measure is put in an *event queue* which can be examined by the user program



Event Types

- Window: resize, expose, iconify
- Mouse: click one or more buttons
- Motion: move mouse
- Keyboard: press or release a key
- Idle: nonevent
  - Define what should be done if no other event is in queue

Callbacks

- Programming interface for event-driven input
- Define a *callback function* for each type of event the graphics system recognizes
- This user-supplied function is executed when the event occurs
- GLUT example:
  `glutMouseFunc(mymouse)`

$$\uparrow$$

mouse callback function

GLUT Callbacks

GLUT recognizes a subset of the events recognized by any particular window system (Windows, X, Macintosh)

- **`glutDisplayFunc`**
- **`glutMouseFunc`**
- **`glutReshapeFunc`**
- **`glutKeyboardFunc`**
- **`glutIdleFunc`**
- **`glutMotionFunc, glutPassiveMotionFunc`**

GLUT Event Loop

- Recall that the last line in **`main.c`** for a program using GLUT must be

```
glutMainLoop();
```

which puts the program in an infinite event loop

- In each pass through the event loop, GLUT
  - looks at the events in the queue
  - for each event in the queue, GLUT executes the appropriate callback function if one is defined
  - if no callback is defined for the event, the event is ignored

Display Callback

- The display callback is executed whenever GLUT determines that the window should be refreshed, for example
  - When the window is first opened
  - When the window is reshaped
  - When a window is exposed
  - When the user program decides it wants to change the display
- In **`main.c`**
  - **`glutDisplayFunc(mydisplay)`** identifies the function to be executed
  - Every GLUT program must have a display callback

Posting Re-displays

- Many events may invoke the display callback function
  - Can lead to multiple executions of the display callback on a single pass through the event loop
- We can avoid this problem by instead using

```
glutPostRedisplay();
```

which sets a flag.

- GLUT checks to see if the flag is set at the end of the event loop
- If set then the display callback function is executed

Animating a Display

- When we redraw the display through the display callback, we usually start by clearing the window
  - `glClear()`

then draw the altered display

- Problem: the drawing of information in the frame buffer is decoupled from the display of its contents
  - Graphics systems use dual ported memory
- Hence we can see partially drawn display
  - See the program **`single_double.c`** for an example with a rotating cube

Double Buffering

- Instead of one color buffer, we use two
  - **Front Buffer**: one that is displayed but not written to
  - **Back Buffer**: one that is written to but not displayed
- Program then requests a double buffer in main.c
  - `glutInitDisplayMode(GL_RGB | GL_DOUBLE)`
  - At the end of the display callback buffers are swapped

```
void mydisplay()
{
      glClear(GL_COLOR_BUFFER_BIT|….)
.
/* draw graphics here */
.
      glutSwapBuffers()
}
```

Using the Idle Callback

- The idle callback is executed whenever there are no events in the event queue
  - **`glutIdleFunc(myidle)`**
  - Useful for animations

```
void myidle() {
/* change something */
      t += dt
      glutPostRedisplay();
}

Void mydisplay() {
      glClear();
/* draw something that depends on t */
      glutSwapBuffers();
}
```

Using Globals

- The form of all GLUT callbacks is fixed
  - void **`mydisplay()`**
  - void **`mymouse(GLint button, GLint state, GLint x, GLint y)`**
- Must use globals to pass information to callbacks

```
float t; /*global */

void mydisplay()
{
/* draw something that depends on t
}
```

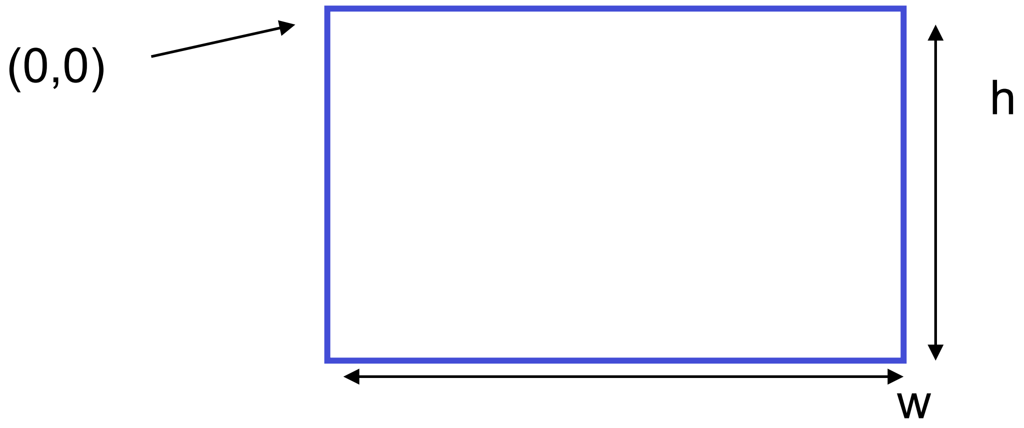
Mouse Callback

```
glutMouseFunc(mymouse)
void mymouse(GLint button, GLint
 state, GLint x, GLint y)
```

- Returns
  - which button (**GLUT_LEFT_BUTTON**, **GLUT_MIDDLE_BUTTON**, **GLUT_RIGHT_BUTTON**) caused event
  - state of that button (**GLUT_UP**, **GLUT_DOWN**)
  - Position in window

Positioning

- The position in the screen window is usually measured in pixels with the origin at the top-left corner
  - Consequence of refresh done from top to bottom
- OpenGL uses a world coordinate system with origin at the bottom left
  - Must invert *y* coordinate returned by callback by height of window
  - $y = h - y;$



Obtaining Window Size

- To invert the *y* position we need the window height
  - Height can change during program execution
  - Track with a global variable
  - New height returned to reshape callback that we will look at in detail soon
  - Can also use query functions
    - `glGetIntv`
    - `glGetFloatv`

  to obtain any value that is part of the state

Terminating a Program

- In our original programs, there was no way to terminate them through OpenGL
- We can use the simple mouse callback

```
void mouse(int btn, int state, int x, int y)
{
   if(btn==GLUT_RIGHT_BUTTON && state==GLUT_DOWN)
      exit(0);
}
```

Using Mouse Position

- In the next example, we draw a small square at the location of the mouse each time the left mouse button is clicked
- This example does not use the display callback but one is required by GLUT; We can use the empty display callback function

```
mydisplay(){}
```

Drawing squares at cursor location

```
void mymouse(int btn, int state, int x, int y)
{
   if(btn==GLUT_RIGHT_BUTTON && state==GLUT_DOWN)
       exit(0);
    if(btn==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
       drawSquare(x, y);
}
void drawSquare(int x, int y)
{
    y=w-y; /* invert y position */
    glColor3ub( (char) rand()%256, (char) rand )%256,
       (char) rand()%256); /* a random color */
    glBegin(GL_POLYGON);
        glVertex2f(x+size, y+size);
        glVertex2f(x-size, y+size);
        glVertex2f(x-size, y-size);
        glVertex2f(x+size, y-size);
     glEnd();
}
```

Using Motion Callback

- We can draw squares (or anything else) continuously as long as a mouse button is depressed by using the motion callback
  - `glutMotionFunc(drawSquare)`
- We can draw squares without depressing a button using the passive motion callback
  - `glutPassiveMotionFunc(drawSquare)`

Using the Keyboard

```
glutKeyboardFunc(mykey)
void mykey(unsigned char key,
        int x, int y)
```

- Returns ASCII code of key depressed and mouse location

```
void mykey()
{
        if(key == 'Q' | key == 'q')
                exit(0);
}
```

Special and Modifier Keys

- GLUT defines the special keys in **`glut.h`**
  - Function key 1: **`GLUT_KEY_F1`**
  - Up arrow key: **`GLUT_KEY_UP`**
    - **`if(key == 'GLUT_KEY_F1'`** ……
- Can also check of one of the modifiers
  - **`GLUT_ACTIVE_SHIFT`**
  - **`GLUT_ACTIVE_CTRL`**
  - **`GLUT_ACTIVE_ALT`**

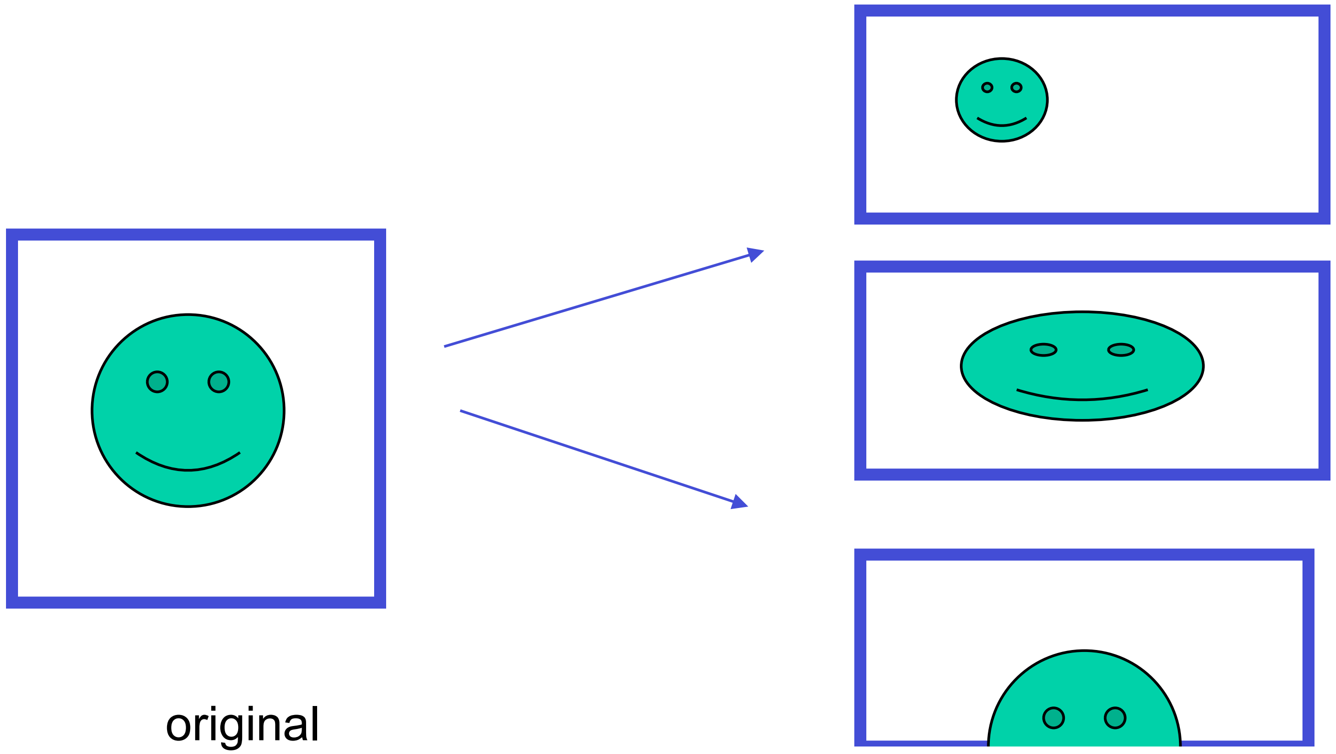
  is depressed by

  `glutGetModifiers()`
  - Allows emulation of three-button mouse with one- or two-button mice

Reshaping the Window

- We can reshape and resize the OpenGL display window by pulling the corner of the window
- What happens to the display?
  - Must redraw from application
  - Two possibilities
    - Display part of world
    - Display whole world but force to fit in new window
      - Can alter aspect ratio

Reshape Possibilities



original

reshaped

Reshape Callback

```
glutReshapeFunc(myreshape)
void myreshape( int w, int h)
```

- Returns width and height of new window (in pixels)
- A redisplay is posted automatically at end of execution of the callback
- GLUT has a default reshape callback but you probably want to define your own

- The reshape callback is good place to put viewing functions because it is invoked when the window is first opened

Example Reshape

- This reshape preserves shapes by making the viewport and world window have the same aspect ratio

```
void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION); /* switch matrix mode */
    glLoadIdentity();
    if (w <= h)
        gluOrtho2D(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w,
            2.0 * (GLfloat) h / (GLfloat) w);
    else  gluOrtho2D(-2.0 * (GLfloat) w / (GLfloat) h, 2.0 *
            (GLfloat) w / (GLfloat) h, -2.0, 2.0);
    glMatrixMode(GL_MODELVIEW); /* return to modelview mode */
}
```

Toolkits & Widgets

- Most window systems provide a toolkit or library of functions for building user interfaces that use special types of windows called *widgets*
- Widget sets include tools such as
  - Menus
  - Slidebars
  - Dials
  - Input boxes
- But toolkits tend to be platform dependent
- GLUT provides a few widgets including menus

Menus in GLUT

- GLUT supports pop-up menus
  - A menu can have submenus
- Three steps
  - Define entries for the menu
  - Define action for each menu item
    - Action carried out if entry selected
  - Attach menu to a mouse button

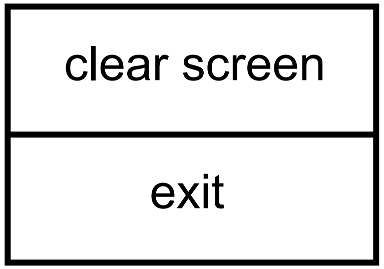
A simple menu example

- In **`main.c`**

```
menu_id = glutCreateMenu(mymenu);
glutAddmenuEntry(“clear Screen”, 1);

gluAddMenuEntry(“exit”, 2);

glutAttachMenu(GLUT_RIGHT_BUTTON);
```



entries that appear when
right button depressed

identifiers

Menu Actions

- Menu callback

```
void mymenu(int id)
{
      if(id == 1) glClear();
      if(id == 2) exit(0);
}
```

- Note each menu has an id that is returned when it is created
- Add submenus by

`glutAddSubMenu(char *submenu_name, submenu id)`

Additional GLUT functions

- Dynamic Windows
  - Create and destroy during execution
- Subwindows
- Multiple Windows
- Changing callbacks during execution
- Timers
- Portable fonts
  - **`glutBitmapCharacter`**
  - **`glutStrokeCharacter`**

More Sophisticated Interactivity

- Interactive CG programs using
  - Picking
    - Select objects from the display
    - Three methods
  - Rubberbanding
    - Interactive drawing of lines and rectangles
  - Display Lists
    - Retained mode graphics

Picking

- Identify a user-defined object on the display
- In principle, it should be simple because the mouse gives the position and we should be able to determine to which object(s) a position corresponds
- Practical difficulties
  - Pipeline architecture is feed forward, hard to go from screen back to world
  - Complicated by screen being 2D, world is 3D
  - How close do we have to come to object to say we selected it?

Three Approaches

- Hit list
  - Most general approach but most difficult to implement
- Use back or some other buffer to store object ids as the objects are rendered
- Rectangular maps
  - Easy to implement for many applications
  - See paint program in text (chap 3, pg 150 -)

Rendering Modes

- OpenGL can render in one of three modes selected by **`glRenderMode(mode)`**
  - **`GL_RENDER`**: normal rendering to the frame buffer (default)
  - **`GL_FEEDBACK`**: provides list of primitives rendered but no output to the frame buffer
  - **`GL_SELECTION`**: Each primitive in the view volume generates a *hit record* that is placed in a *name stack* which can be examined later

Selection Mode Functions

- `glSelectBuffer()`: specifies name buffer
- `glInitNames()`: initializes name buffer
- `glPushName(id)`: push id on name buffer
- `glPopName()`: pop top of name buffer
- `glLoadName(id)`: replace top name on buffer

- id is set by application program to identify objects

Using Selection Mode

- Initialize name buffer
- Enter selection mode (using mouse)
- Render scene with user-defined identifiers
- Reenter normal render mode
  - This operation returns number of hits
- Examine contents of name buffer (hit records)
  - Hit records include id and depth information

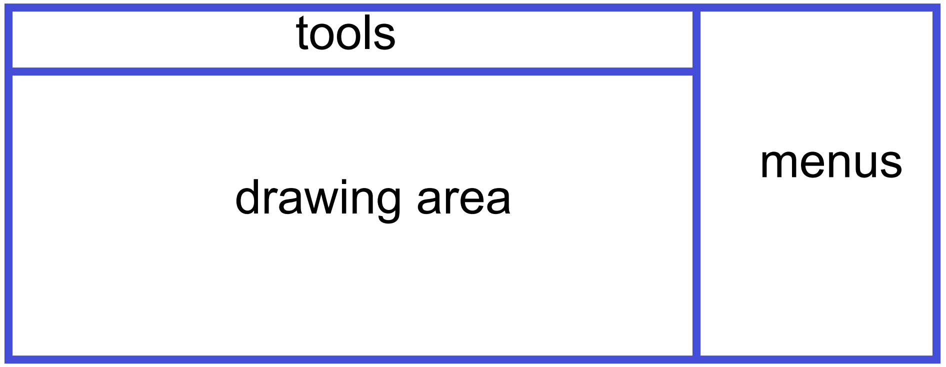
Selection Mode & Picking

- As we just described it, selection mode won't work for picking because every primitive in the view volume will generate a hit
- Change the viewing parameters so that only those primitives near the cursor are in the altered view volume
  - Use **`gluPickMatrix`** (See Text, Pg 785)

Creates a projection matrix for picking that restricts rendering to a w x h are centered at (x,y) in window coords within the viewport vp

Using Regions of the Screen

- Many applications use a simple rectangular arrangement of the screen
  - Example: paint/CAD program

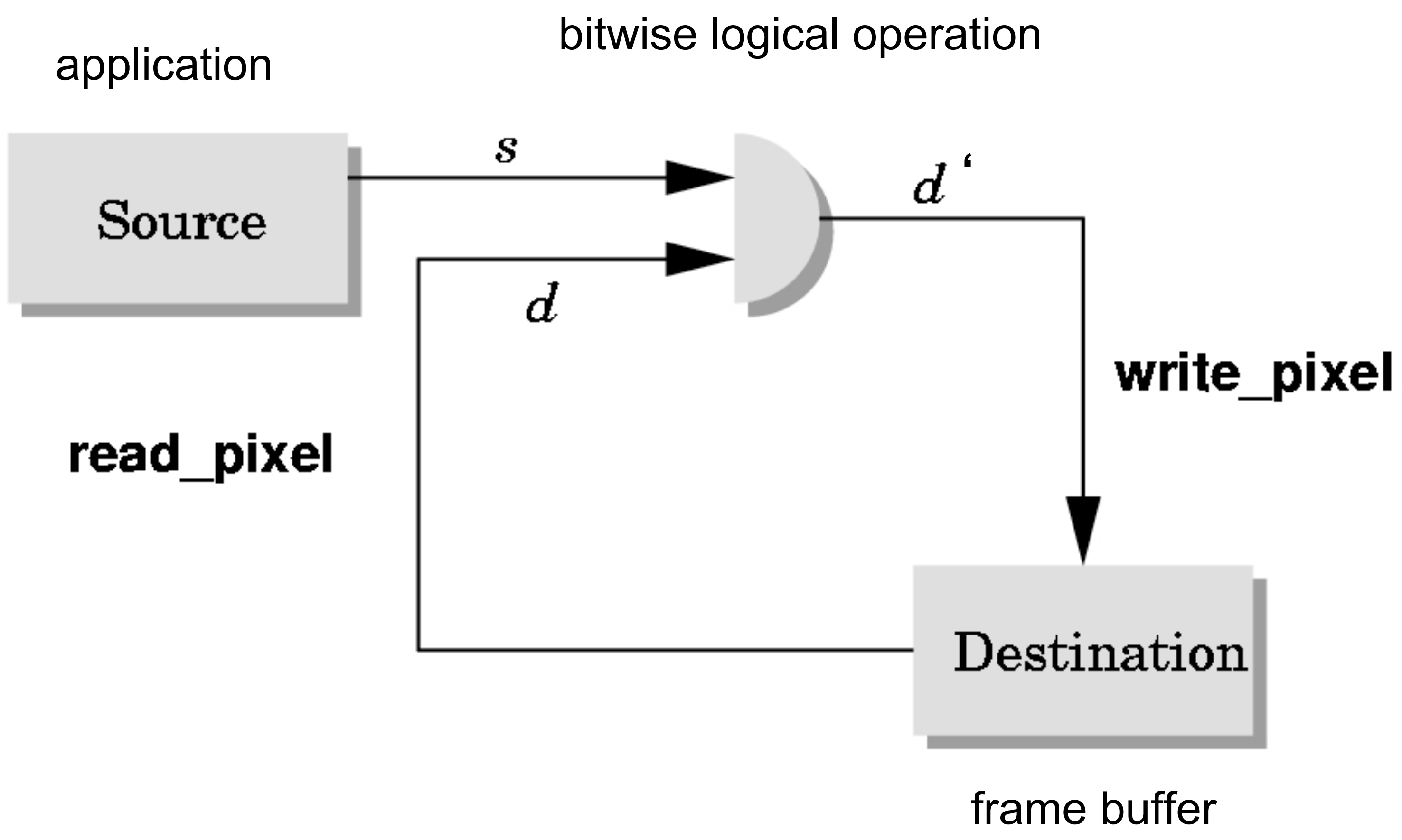


- Easier to look at mouse position and determine which area of screen it is in than using selection mode picking

Using another buffer and color for picking

- For a small number of objects, we can assign a unique color (often in color index mode) to each object
- We then render the scene to a color buffer other than the front buffer so the results of the rendering are not visible
- We then get the mouse position and use `glReadPixels()` to read the color in the buffer we just wrote at the position of the mouse
- The returned color gives the id of the object

Writing Modes



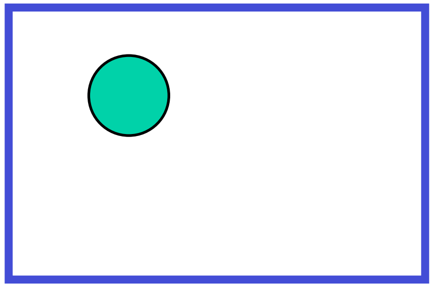
XOR Write

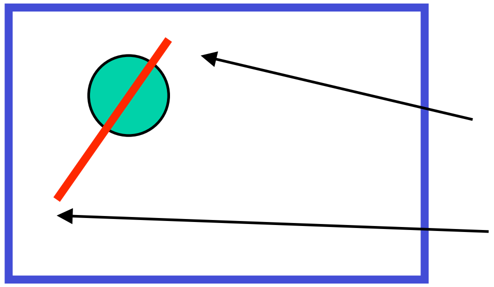
- Usual (default) mode: source replaces destination (d' = s)
  - Cannot write temporary lines this way because we cannot recover what was "under" the line in a fast simple way
- Exclusive OR mode (XOR) ($d' = d \oplus s$)
  - $(y \oplus x) \oplus x = y$ (applying XOR twice returns original)
  - Hence, if we use XOR mode to write a line, we can draw it a second time and line is erased!

Rubberbanding

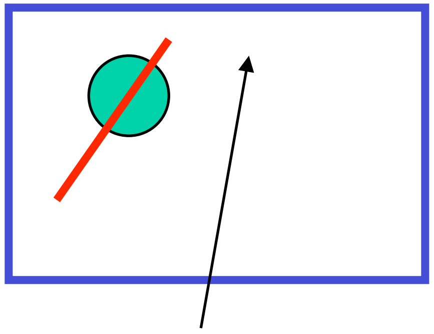
- Switch to XOR write mode
- Draw object
  - For line can use first mouse click to fix one endpoint and then use motion callback to continuously update the second endpoint
  - Each time mouse is moved, redraw line which erases it and then draw line from fixed first position to to new second position
  - At end, switch back to normal drawing mode and draw line
  - Works for other objects: rectangles, circles

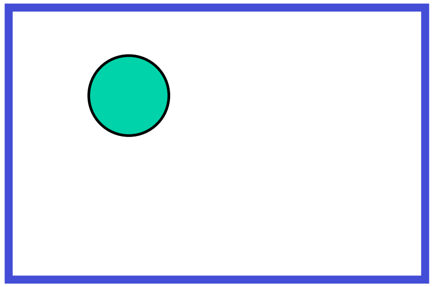
Rubberband Lines

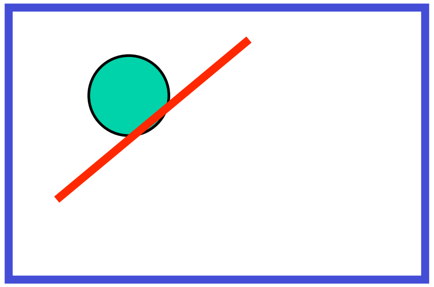

initial display



draw line with mouse in XOR mode


mouse moved to new position


original line redrawn with XOR


new line drawn with XOR

XOR in OpenGL

- There are 16 possible logical operations between two bits
- All are supported by OpenGL
  - Must first enable logical operations
    - **`glEnable(GL_COLOR_LOGIC_OP)`**
  - Choose logical operation
    - **`glLogicOp(GL_XOR)`**
    - **`glLogicOp(GL_COPY)`** (default)

Immediate & Retained Modes

- Recall that in a standard OpenGL program, once an object is rendered there is no memory of it and to redisplay it, we must re-execute the code for it
  - Known as *immediate mode graphics*
  - Can be especially slow if the objects are complex and must be sent over a network
- Alternative is define objects and keep them in some form that can be redisplayed easily
  - *Retained mode graphics*
  - Accomplished in OpenGL via *display lists*

Display Lists

- Conceptually similar to a graphics file
  - Must define (name, create)
  - Add contents
  - Close
- In client-server environment, display list is placed on server
  - Can be redisplayed without sending primitives over network each time

Display List Functions

- Creating a display list

```
GLuint id;

void init()
{
    id = glGenLists( 1 );
    glNewList( id, GL_COMPILE );
    /* other OpenGL routines */
    glEndList();
}
```

- Call a created list

```
void display()
{
    glCallList( id );
}
```

Display Lists and State

- Most OpenGL functions can be put in display lists
- State changes made inside a display list persist after the display list is executed
- Can avoid unexpected results by using **`glPushAttrib`** and **`glPushMatrix`** upon entering a display list and **`glPopAttrib`** and **`glPopMatrix`** before exiting

Hierarchy & Display Lists

- Consider model of a car
  - Create display list for chassis
  - Create display list for wheel

```
glNewList( CAR, GL_COMPILE );
 glCallList( CHASSIS );
 glTranslatef( … );
 glCallList( WHEEL );
 glTranslatef( … );
 glCallList( WHEEL );
       …
glEndList();
```