Week 5

- Rotation about One Axis
- Simple Transformations
- Arbitrary Rotation and Shear
- Input Devices
- Input Modes
- Callbacks

Rotation about One Axis

Simple Rotation

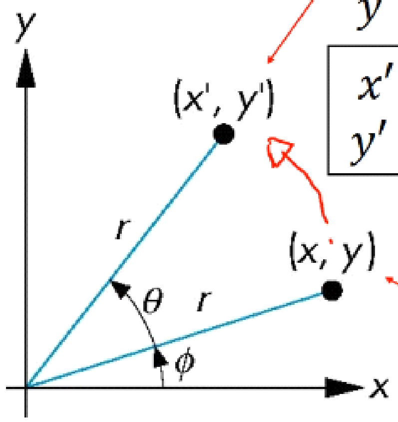
Consider a rotation about the origin by θ degrees

Radius stays the same, angle increases by θ

$$x' = r\cos(\phi + \theta)$$
$$y' = r\sin(\phi + \theta)$$

new point **p'**

$$\boxed{\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \end{aligned}}$$



$$x = r\cos\phi$$
$$y = r\sin\phi$$

original point **p**

Matrix Rotation

Formulae:

$$\cos(\theta + \phi) = \cos\theta\cos\phi - \sin\theta\sin\phi$$
$$\sin(\theta + \phi) = \sin\theta\cos\phi + \cos\theta\sin\phi$$

Gives us:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = R_z(\theta)\begin{bmatrix} r\cos\phi \\ r\sin\phi \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} r\cos(\theta+\phi) \\ r\sin(\theta+\phi) \\ 0 \\ 1 \end{bmatrix}$$

Thus,

- $x' = r\cos(\theta + \phi)$
- $y' = r\sin(\theta + \phi)$
- $z' = 0$

In general,

- $x' = x\cos\theta - y\sin\theta$
- $y' = x\sin\theta + y\cos\theta$
- $z' = z$

Rotation About The Z Axis

Rotation about the Z axis is *rotation is two dimensions*

$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	X and Y get multipliers Z is unchanged

Rotation About All Axes

Rotation	`Matrix × vPosition → gl_Position`
X	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ \cos\theta \cdot y - \sin\theta \cdot z \\ \sin\theta \cdot y + \cos\theta \cdot z \\ 1 \end{pmatrix}$
Y	$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x + \sin\theta \cdot z \\ y \\ -\sin\theta \cdot x + \cos\theta \cdot z \\ 1 \end{pmatrix}$
Z	$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x - \sin\theta \cdot y \\ \sin\theta \cdot x + \cos\theta \cdot y \\ z \\ 1 \end{pmatrix}$

Example: Code for Rotation About X Axis

In Display Callback:

```
float angle = 0.001 * glutGet(GLUT_ELAPSED_TIME);
mat3  xRot = mat3(
   vec3(1.0, 0.0, 0.0),
   vec3(0.0, cos(angle), -sin(angle)),
   vec3(0.0, sin(angle), cos(angle)));
```

$$\begin{bmatrix} \boxed{\begin{matrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{matrix}} & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \\ \begin{matrix} 0 & 0 & 0 \end{matrix} & 1 \end{bmatrix}$$

In Vertex Shader:

```
gl_Position = vec4(rotMatrix * vPosition, 1.0);
```

Note:

Different rotation matrices can be multiplied together, to give rotation in 2 or 3 axes

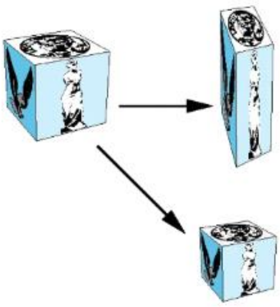
Simple Transformations

The four standard transformations are:

- Rotation
- Translation
- Scaling
- Shear

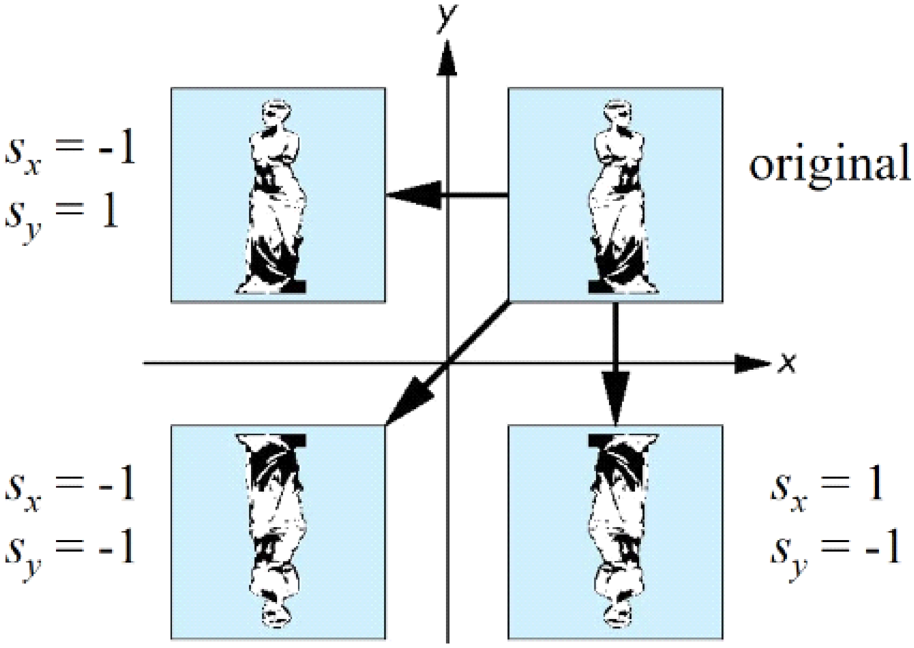
Scaling

Scaling is *expanding or contracting an object* **along each axis** (fixed point of origin)

$x' = s_x x$ $y' = s_y y$ $z' = s_z z$	Scaling each axis separately When we scale uniformly, we use the same *s* value for each
$\mathbf{p}' = \mathbf{S}\mathbf{p}$ $\mathbf{S} = \mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	This is a Transformation matrix in homogenous coordinate form Scaling coefficients are placed into matrix We multiply all points in the object by the matrix
	Result

Reflection

Reflection corresponds to negative scale factors



Inverse

Inverse Translation

- $\mathbf{T}^{-1}(d_x, d_y, d_z) = \mathbf{T}(-d_x, -d_y, -d_z)$
- We put negative values in the translation column

Inverse Rotation

- $\mathbf{R}^{-1}(\theta) = \boldsymbol{R}(-\theta)$
- Holds for any rotation matrix
  Note that since $\cos(-\theta) = \cos(\theta)$ and $\sin(-\theta) = -\sin(\theta)$
  - $\mathbf{R}^{-1}(\theta) = \mathbf{R}^{\mathrm{T}}(\theta)$
  - R inverse is R transposed - we exchange the rows with the columns

Scaling

- $\mathbf{S}^{-1}(s_x, s_y, s_z) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$
- We put 1 over each scaling factor

Concatenation

We can form arbitrary affine transformation matrices by *multiplying together rotation, translation, and scaling matrices*

Multiplying each point by each matrix separately would be costly

The most efficient way is to:

1. Multiply all the matrices needed together first ($M = ABCD$)
2. We compute Mp for each vertice (p)

The difficult part is how to form a desired transformation from the specifications in the application

Order of Transformations

The rightmost matrix is the first applied

$$\mathbf{p}' = \mathbf{ABCp} = \mathbf{A}(\mathbf{B}(\mathbf{Cp}))$$

Extra:

If we are using row vectors, the order is reversed

$$\mathbf{p}'^{\mathrm{T}} = \mathbf{p}^{\mathrm{T}}\mathbf{C}^{\mathrm{T}}\mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}$$

Matrices do not commute in general

Arbitrary Rotation and Shear

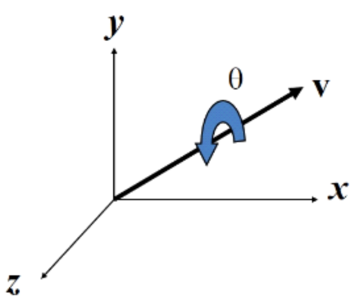
Sunday, 15 March 2020 11:39 PM

Rotation about an Arbitrary Axis

A rotation angle of θ about an arbitrary axis can be <u>decomposed into the concatenation of rotations</u> about the x,y and z axes

To rotate about an arbitrary vector (v), we need to multiply the 4x4 transformation matrices for each of the three axes in the **following order**:

- $R(\theta) = R_z(\theta_z)\, R_y(\theta_y)\, R_x(\theta_x)$
  $\theta_x, \theta_y,$ and θ_z are called the **Euler angles**



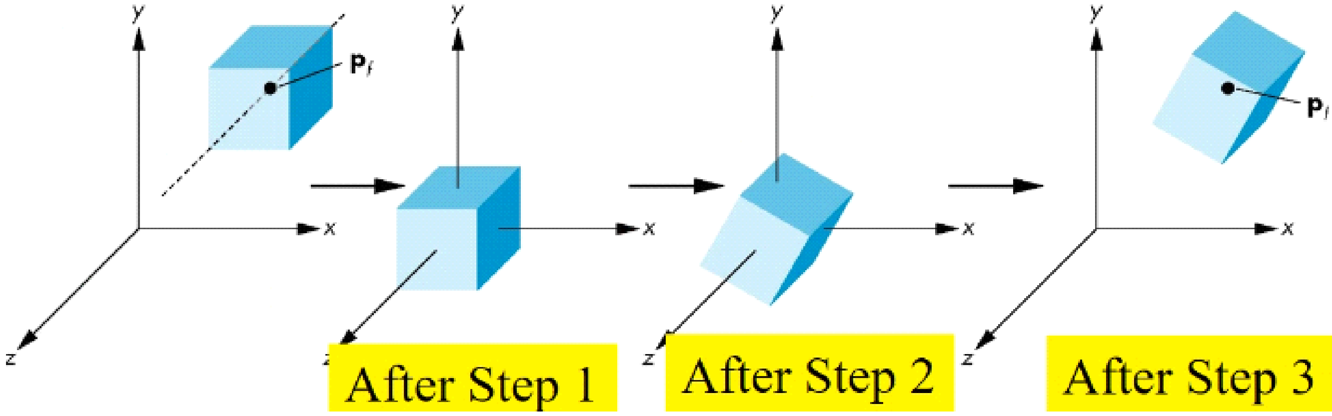
Note that rotations do not commute

We can use rotations in another order but with different angles

The order of rotations is important!

Rotation about an Arbitrary Point

$$\mathbf{M} = \mathbf{T}(p_f)\,\mathbf{R}(\theta)\,\mathbf{T}(-p_f)$$



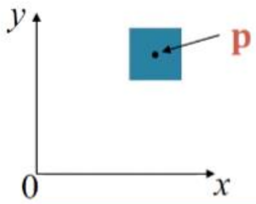
1. Translate to the origin
2. Rotate
3. Translate back to the point

Example

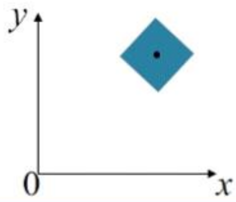
We want to rotate a square 45 degrees about its own center (p)

This is the same as rotating about the z-axis (pointing out of the page) in 3D.

Our aim is to construct a matrix M so that when the four vertices of the square are pre-multiplied by we get the desired output.



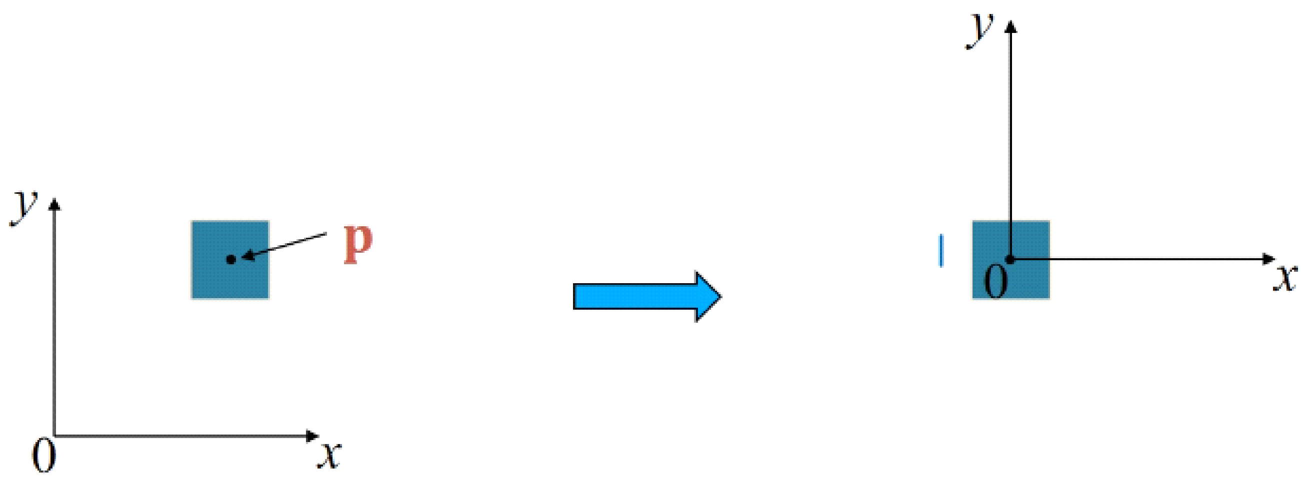
Before rotation



Output wanted after rotation

Step 1

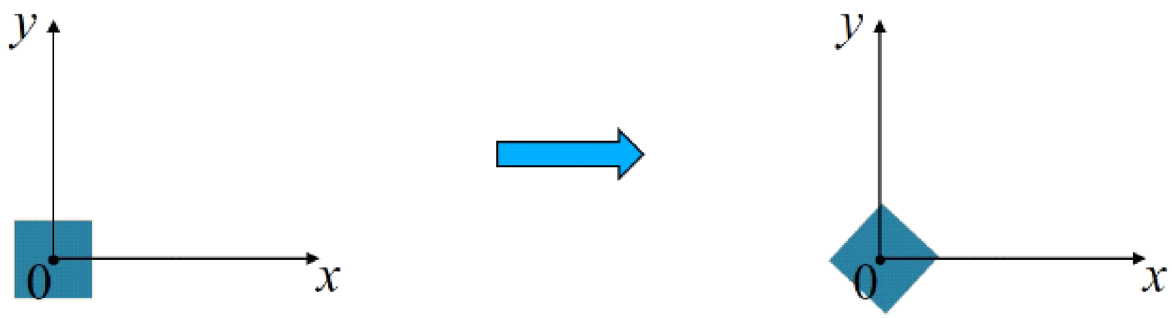
Apply a translation so that the origin is at p.



$$\mathbf{M} = \mathbf{T}(-\mathbf{p})$$

Step 2

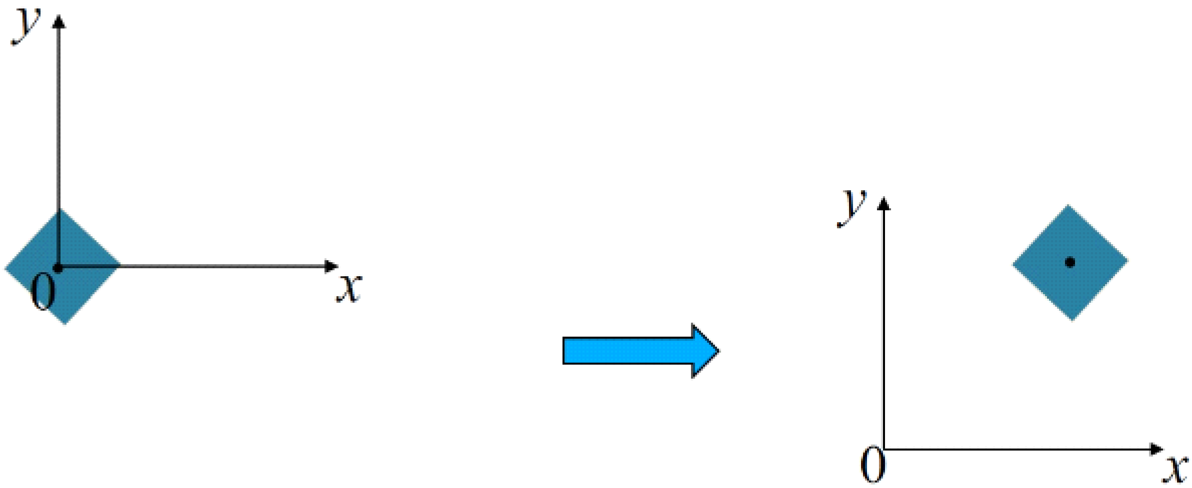
We apply a 45 degree rotation about the z-axis at the origin



$$\mathbf{M} = \mathbf{R}_z(\theta)\,\mathbf{T}(-\mathbf{p})$$

Step 3

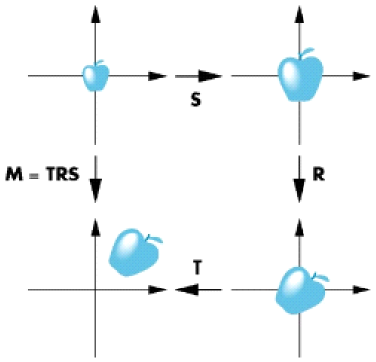
We move the origin back to where it was before



$$\mathbf{M} = \mathbf{T}(\mathbf{p})\,\mathbf{R}_z(\theta)\,\mathbf{T}(-\mathbf{p})$$

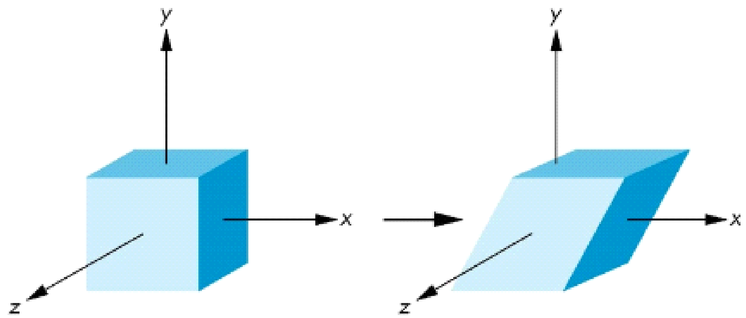
Instancing

In modeling, we often start with a simple object centered at the origin, oriented with the axis, and at a standard size. We can create **different instances of this same object**, each with different scales/orientations

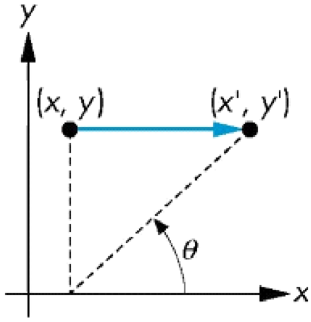
	We apply an instance transformation to its vertices to : • Scale it • Orient it • Locate (Translate) it

Shear

Shearing is equivalent to *pulling faces* in opposite directions



Shear Matrix

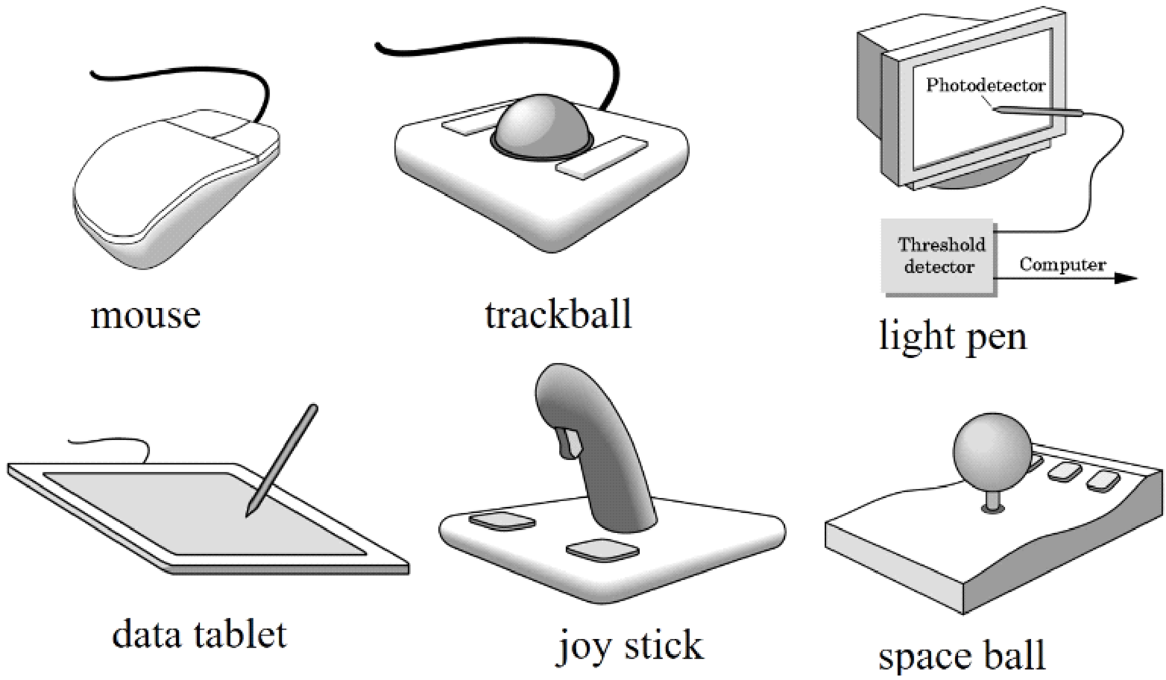
▪ $x' = x + y\cot\theta$ ▪ $y' = y$ ▪ $z' = z$		A simple shear along the x-axis Theta is **angle of shear** The points to move (dashed) points such that the angle is theta
$\Rightarrow \mathbf{H}(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$		The shear matrix

User Inputs control the program through input devices

Input devices can be viewed as:

- **Physical devices**
  - Described by their physical properties, e.g., mouse, keyboard, trackball, etc.
- **Logical devices**
  - Characterized by how they influence the application program, e.g., what is returned to the program via the API
    - An (x, y) position on the screen?
    - An object identifier for a menu item chosen?
    - Both are performed by the same physical device (the mouse, in this case) but what is returned to the program is different.

Physical Devices



Scanning Statements

Consider the C and C++ code

- `cin >> x; // C++:`
- `scanf ("%d", &x) ; //C`

We cannot determine the physical input device

Could be keyboard, file, output from another program

The code provides **logical input**

A number (an int) is returned to the program <u>regardless of the physical device</u>

Types of Logical Input

OpenGL and GLUT provide functions to handle **six types of logical input**

- **Locator:** Return a position, e.g., clicked at by a mouse pointer
- **Choice:** Return one of n discrete items, e.g., a menu item
- **String:** Return strings of characters, e.g., via key presses
- **Stroke:** Return array of positions
- **Valuator:** Return floating point number
- **Pick:** Return ID of an object

Incremental/Relative Devices

Devices such as the data tablet return an absolute position directly to the OS

Devices such as the mouse, track ball, 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
- But, gives us set **variable sensitivity**

Input Modes

Input devices contain a trigger which can be used to send a signal to the OS

- 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

Input modes concern how and when input is obtained

Two types of input modes:

- **Request mode**
- **Event mode**

Request Mode

For request mode input, the *input value is provided to program* only when the user triggers the device

A typical example is keyboard input:

- The application program request a keyboard input from the user
- The user can type, erase (backspace), correct.
- The application program hangs there until the enter (return) key (the trigger) is depressed

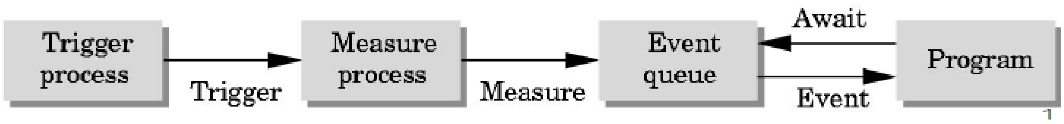
Request mode is not suitable for programs that need to interact with the user.

Most systems have more than one input device, each of which can be triggered at an arbitrary time by a user

Event Mode

In **event mode** input, the program specifies a number of input events that are of interest.

- The program gets into an event handling loop to deal with the events when they occur
- Each trigger generates an event whose measure is put in an event queue which can be examined by the user program



Event Types

Type	Description
Window event	Resize, expose, iconify
Mouse event	Click one or more mouse buttons
Motion event	This refers to the mouse move event (when the cursor is inside the window of the application program)
Keyboard	Press or release a key
Idle	Non-event Defines what should be done if no other event is in the event queued

Callbacks

We can program to handle event mode input

We can define a callback function for each type of event that the graphics system recognizes

This user-supplied function is executed when the event occurs

GLUT Callbacks

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

- glutDisplayFunc
- glutCreateMenu
- glutMouseFunc *(mouse clicking)*
- glutReshapeFunc *(window reshaped)*
- glutKeyboardFunc *(key pressed)*
- glutIdleFunc
- glutMotionFunc *(mouse click + motion)*
- glutPassiveMotionFunc *(mouse moves without click down)*

glutMouse Function

```
// Usage
// Params: Mouse button type, down/released state, location
void glutMouseFunc(void ( *func)(int button, int state, int x, int y));

// Defined in the main()
glutMouseFunc(myMouseFun);

// The function definition
void myMouseFun(int button, int state, int x, int y)
{
   // This is where you write code for what you want to do when a mouse
   "event" happens
}
```

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:

1. Looks at the events in the queue
2. For each event in the queue, GLUT executes the appropriate callback function if one is defined
3. If no callback is defined for the event, the event is ignored

The 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 *(closed window in front)*
- When the user program decides it wants to change the display

In main ()

- glutDisplayFunc (mydisplay) identifies the function to be executed
- Every GLUT program must have a display callback

Posting Redisplays

Many events may invoke the display callback function, lead into multiple executions of the display callback (glutDisplayFunc) 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
- So the display callback is called once only, improving performance