Comp 410/510

Computer Graphics
Spring 2023

Input & Interaction

Objectives

- Introduce the basic input devices
  - Physical Devices
  - Logical Devices
- Input Modes (*Request* vs *Event*)
- Event-driven programming with GLFW
- Interaction

Basic interaction

- Ivan Sutherland (MIT 1963) established the basic interaction paradigm that characterizes interactive computer graphics:
  - User sees an *object* on the display
  - User points to (*picks*) the object with an input device (light pen, mouse, trackball, etc.)
  - Object changes (moves, rotates, morphs, etc.)
  - Repeat

User Input

- Input devices can be characterized by their
  - Physical properties
    - Mouse
    - Keyboard
    - Trackball
    - etc.
  - Logical Properties
    - What is returned to program via API
      - A position
      - An object identifier
      - An action identifier
      - etc.
- Modes
  - How and when input is obtained
    - Request or event

Graphical Logical Devices

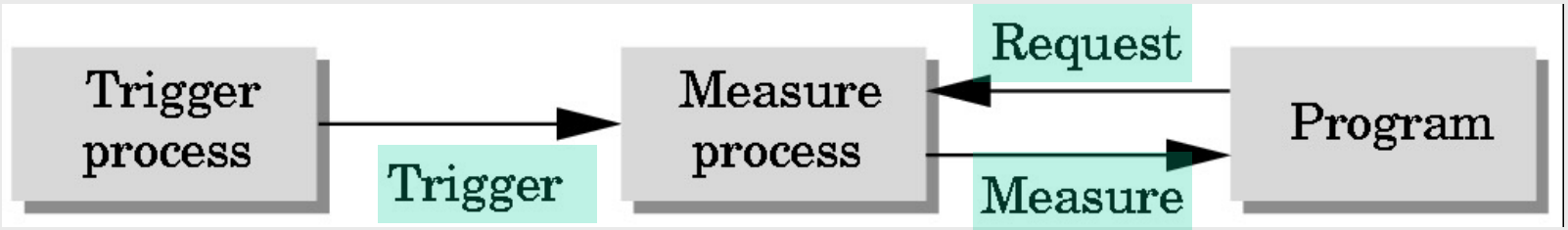
- Graphical input is more varied than input to standard programs (which is usually numbers, characters, or bits)
- Examples to types of graphical input:
  - **Locator**: return a position
  - **Pick**: return ID of an object
  - **Keyboard**: return strings of characters
  - **Stroke**: return array of positions
  - **Valuator**: return floating point number (such as slidebars)
  - **Choice**: return one of n items (such as menus)

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 on the keyboard
- When triggered, input devices return information (their *measure*) to the system
  - Mouse returns position information
  - Keyboard returns ASCII code
- *Request* or *Event* Mode

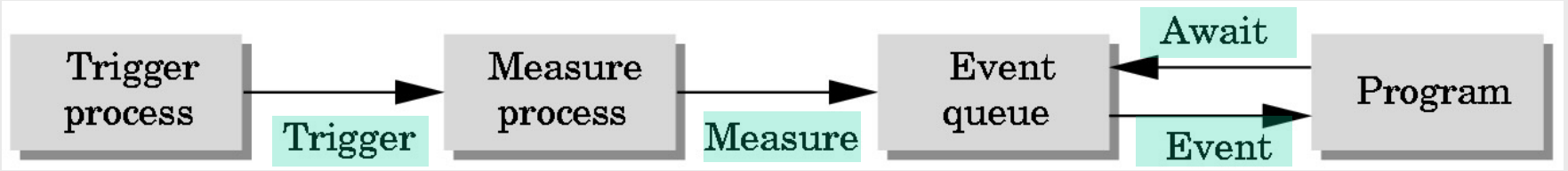
Request Mode

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



Event Mode

- Program waits for inputs that user may possibly provide
- 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**: non-event
  - 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 that the graphics system recognizes
- This user-supplied function is executed when the event occurs
- GLWF example: `glfwSetMouseButtonCallback(mywindow, mouse_button_callback)`

`mouse_button_callback` is set to be the user-defined function which specifies what to do when a mouse button is pressed.

GLFW callbacks

GLFW recognizes a subset of the events recognized by any particular window system (Windows, X, Mac OS, etc.)

- `glfwSetMouseButtonCallback()`
- `glfwSetCursorPosCallback()`
- `glfwSetKeyCallback()`
- `glfwSetFramebufferSizeCallback()`
- `glfwSetCharCallback()`
- `glfwSetScrollCallback()`
- `glfwGetJoystickButtons()`
- `etc.`

OpenGL/GLFW program structure

```
#include <gl3.h>
#include <glfw3.h>


void init(){
....
}
void display(){
...
}
void update(){
...
}
void mouse_button_callback (GLFWwindow* window, int button, int action, int mods){
......
}
void key_callback (GLFWwindow* window, int key, int scancode, int action, int mods){
......
}

void framebuffer_size_callback(GLFWwindow* window, int width, int height){
}

int main()
{
  /* window intializations*/

  glfwSetKeyCallback(window, key_callback);
  glfwSetMouseButtonCallback(window, mouse_button_callback);
  glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

  init();

  while (!glfwWindowShouldClose(window)){
    update();
    display();
    glfwSwapBuffers(window);
    glfwPollEvents();
  }
}
```

GLFW Event Loop

- Remember that the last part of the main code is a while loop, which puts the program in an infinite event loop
- In each pass through the event loop,
  - GLFW looks at the events in the queue
  - For each event in the queue, GLFW executes the appropriate callback function (if one is defined)
  - If no callback is defined for the event, the event is ignored
  - The scene is redrawn into the framebuffer with the parameters that were possibly updated in the callbacks and/or in `update()`.

Using `glfwGetTime()`

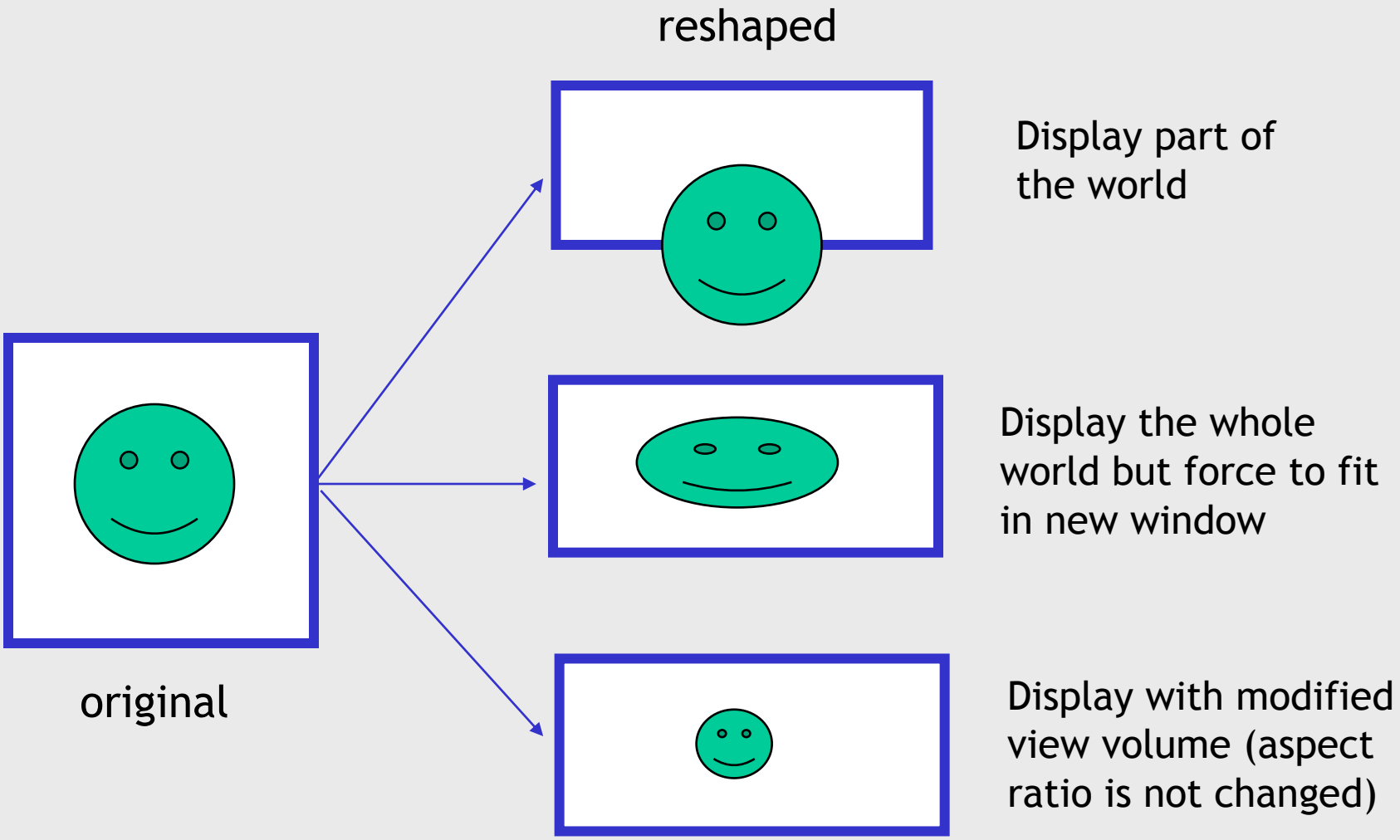
Can use `glfwGetTime()` to adjust the speed of animation:

```
double frameRate = 120, currentTime, previousTime = 0.0;
while (!glfwWindowShouldClose(window))
{
    currentTime = glfwGetTime();
    if (currentTime - previousTime >= 1/frameRate){
        previousTime = currentTime;
        update();
    }
    display();
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

Reshaping the window

- We can reshape and resize the OpenGL display window by pulling the corner of the window
- What happens to the display then?
  - Must redraw the scene from application
  - Three possibilities
    - Display part of the world
    - Display the whole world but force to fit in new window
      - Can alter aspect ratio
    - Display with modified view volume (or area)

Reshape possibilities



The framebuffer size callback

- `glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);`
- `void framebuffer_size_callback(GLFWwindow* window, int width, int height)`
  - is returned width and height of the new resized window (in pixels)
  - GLFW has a default reshape callback but you probably want to define your own
- The framebuffer size callback can be used to define your own reshape behaviour
- It is also a good place to put camera functions because it is invoked when the window is first opened

Example Reshape

The framebuffer size callback below preserves shapes by making the viewport and the view volume have the same aspect ratio:

```
void framebuffer_size_callback(GLFWwindow* window, int w, int h)
{
    glViewport( 0, 0, w, h );

    mat4 projection;
    if (w <= h)
        projection = Ortho(-1.0, 1.0, -1.0*(GLfloat) h / (GLfloat) w,
         1.0*(GLfloat) h / (GLfloat) w, -1.0, 1.0);
    else projection = Ortho(-1.0*(GLfloat) w / (GLfloat) h, 1.0*
           (GLfloat) w / (GLfloat) h, -1.0, 1.0, -1.0, 1.0);

    glUniformMatrix4fv( Projection, 1, GL_TRUE, projection );

}
```

See **spinCube (with reshape)** example

The mouse callback

- **`glfwSetMouseButtonCallback(window, mouse_button_callback)`**
- **`void mouse_button_callback (GLFWwindow* window, int button, int action, int mods)`**

- is returned
  - which button (**`GLFW_MOUSE_BUTTON_RIGHT, GLFW_MOUSE_BUTTON_LEFT, GLFW_MOUSE_BUTTON_MIDDLE`**) causes the event
  - action that was taken (**`GLFW_PRESS, GLFW_RELEASE`**)
  - any modifier keys that were pressed, such as **`GLFW_MOD_SHIFT`** or **`GLFW_MOD_CONTROL`**
    - e.g., **`(mods & GLFW_MOD_SHIFT)`** is true when the shift key is pressed

If cursor position needed in the callback, use:

```
void glfwGetCursorPos(GLFWwindow *window, double *xpos, double *ypos)
```

Terminating a program

We can use a simple mouse callback function to terminate the program execution through OpenGL:

```
void mouse_button_callback(GLFWwindow* window, int button, int action, int mods)
{
    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS)
    {
        exit(0);
    }
}
```

Using globals

- The form of all GLFW callbacks is fixed
  - `void mouse_button_callback(GLFWwindow* window, int button, int action, int mods)`
  - `void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)`

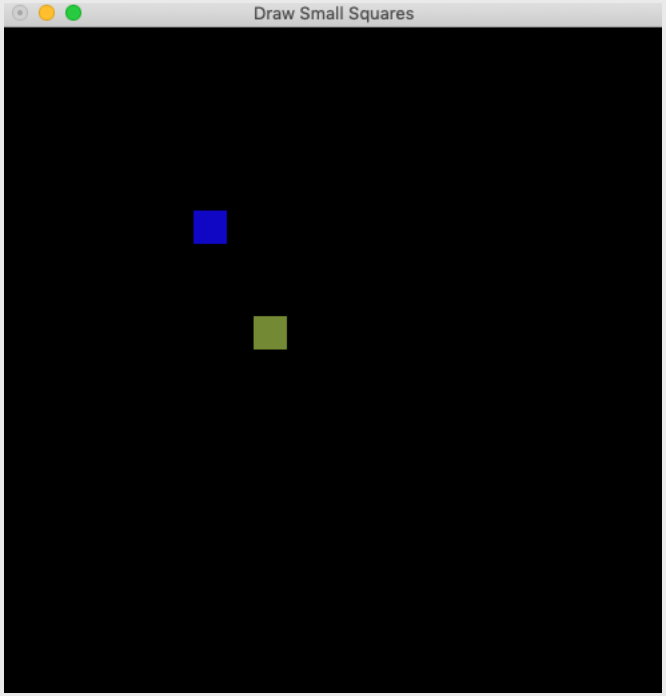
- Need to use global variables to pass information to callbacks:

```
float t; /*global */

void mouse_button_callback (….)
{
// do something that depends on t
}
```

Using the mouse position

In the next example, we'll draw a small square at the location of the mouse each time the left mouse button is clicked:



Drawing squares at cursor location

```
void drawSquare(GLFWwindow* window, double xpos, double ypos){

    glUniform2f(Cursor, (float)xpos, (float)ypos);
    glUniform3f(RandomColor, ((float)rand()) / RAND_MAX,
        ((float)rand()) / RAND_MAX, ((float)rand()) / RAND_MAX);

    int width, height;
    glfwGetWindowSize(window, &width, &height);
    glUniform2f(WindowSize, (float)width, (float)height);

    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
    glfwSwapBuffers(window);
}

void mouse_button_callback(GLFWwindow* window, int button, int
  action, int mods){

    if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_PRESS)
        exit(0);

    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS){
        double xpos, ypos;
        glfwGetCursorPos(window, &xpos, &ypos);
        drawSquare(window, xpos, ypos);
    }
}
```

See **drawSmallSquares** code

Shaders

Vertex shader

```
in vec4 vPosition;
uniform vec2 cursor;
uniform vec2 windowSize;

void main()
{
   gl_Position = vec4(vPosition.x +
       2.0*cursor.x/windowSize.x, vPosition.y +
       2.0*(windowSize.y-cursor.y)/windowSize.y, 0, 1);
}
```

Fragment shader

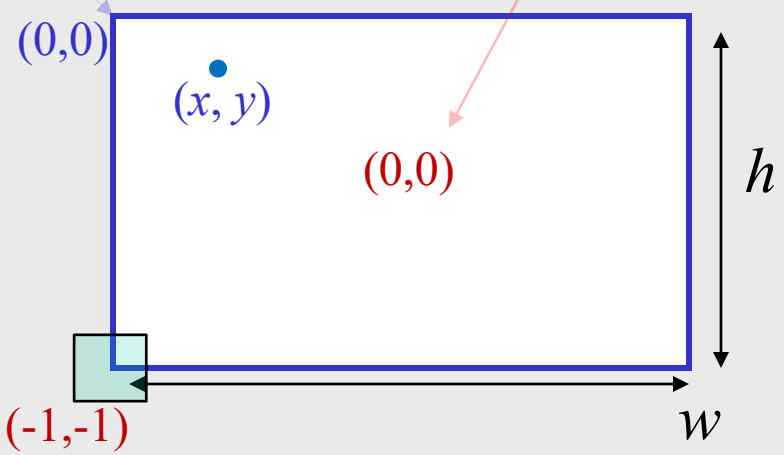
```
uniform vec3 randomColor;
out vec4 fragColor;

void main()
{
    vec4 color = vec4( randomColor.r, randomColor.g,
                randomColor.b, 1.0);
    fragColor = color;

}
```

Positioning

- The cursor position (x, y) in the screen window is measured in pixels with the **origin** at the top-left corner
- OpenGL uses a default world coordinate system with **origin** located at the center of the window (hence the viewing rectangle)
- Thus without any offset, our small square is located around the point (-1,-1) which is at the bottom left corner
- To give cursor offset, we need to invert y coordinate (returned by callback) using the height h of window: $y = h - y$



Vertex Shader

Invert y-coordinate (`cursor.y`) by using the height of window (`windowSize.y`):

Vertex shader

```
in vec4 vPosition;
uniform vec2 cursor;
uniform vec2 windowSize;

void main()
{
   gl_Position = vec4(vPosition.x +
        2.0*cursor.x/windowSize.x, vPosition.y +
        2.0*(windowSize.y-cursor.y)/windowSize.y, 0, 1);
}
```

- Note that the small square is initially positioned at (-1,-1)
- Hence it needs to be moved to the cursor position with an offset
- Since the size of the view volume box is 2, we have to normalize vertex positions into the range [-1,1]
- So we divide the cursor coordinates by window size and then multiply by 2.0

Obtaining the window size

- To invert the y position, we need to know the window height
  - Height h can change during program execution
  - Can track with a global variable
  - New height is also returned to framebuffer size callback that we have seen
  - Can also use enquiry functions
    - `glfwGetWindowSize(window, &width, &height)`
    - `glGetIntv(…)`
    - `glGetFloatv(…)`

  to obtain any value that is part of the state

Drawing squares at cursor location

```
void drawSquare(GLFWwindow* window, double xpos, double ypos){

    glUniform2f(Cursor, (float)xpos, (float)ypos);
    glUniform3f(RandomColor, ((float)rand()) / RAND_MAX,
        ((float)rand()) / RAND_MAX, ((float)rand()) / RAND_MAX);

    int width, height;
    glfwGetWindowSize(window, &width, &height);
    glUniform2f(WindowSize, (float)width, (float)height);

    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
    glfwSwapBuffers(window);
}

void mouse_button_callback(GLFWwindow* window, int button, int
  action, int mods){

    if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_PRESS)
        exit(0);

    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS){
        double xpos, ypos;
        glfwGetCursorPos(window, &xpos, &ypos);
        drawSquare(window, xpos, ypos);
    }
}
```

See **drawSmallSquares** code

Using the motion callback

- We could draw squares (or anything else) continuously while the mouse moves by using the cursor callback:
  - `void mouse_cursor_callback(GLFWwindow* window, double xpos, double ypos)`
- The system returns to the callback function the x and y positions of the cursor.

Using the keyboard

- **glfwSetKeyCallback(window, key_callback)**
- **void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)**
  - is returned the ASCII code of the key
  - the scan code of the key
  - a flag indicating whether the key was released or pressed
  - any modifiers that were pressed at the time of the event

```
void key_callback(GLFWwindow* window, int key, int scancode,
int action, int mods)
{
    switch( key ) {
        case GLFW_KEY_ESCAPE: case GLFW_KEY_Q:
            exit( EXIT_SUCCESS );
            break;
}
```

Special and Modifier Keys

- GLFW defines the special keys in **`glfw.h`**
  - Function key 1: **`GLFW_KEY_F1`**
  - Up arrow key: **`GLFW_KEY_UP`**
    - **`if(key == GLFW_KEY_F1 ……`**

- Can check whether modifiers
  - **`GLFW_MOD_SHIFT`**
  - **`GLFW_MOD_CTRL`**
  - **`GLFW_MOD_ALT`**

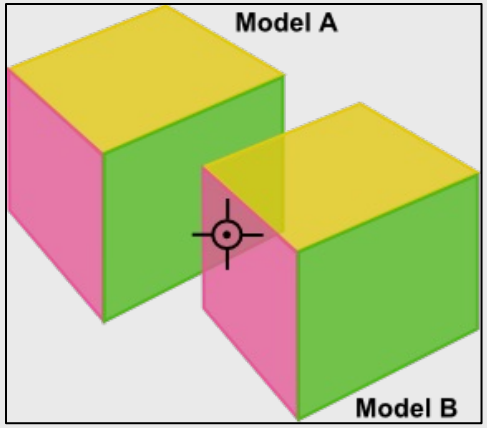
  are pressed via “mod” field in the keyboard or mouse callback

Other functions in GLFW

- Dynamic Windows
  - Create and destroy during execution
- Subwindows
- Multiple Windows
- Changing callbacks during execution
- etc

Picking

- Identifying 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 a position corresponds
- Practical difficulties
  - Pipeline architecture is fed forward, hard to go from screen back to world
  - Complicated by screen being 2D, world is 3D
  - How close should we come to object to say “we’ve selected it”?

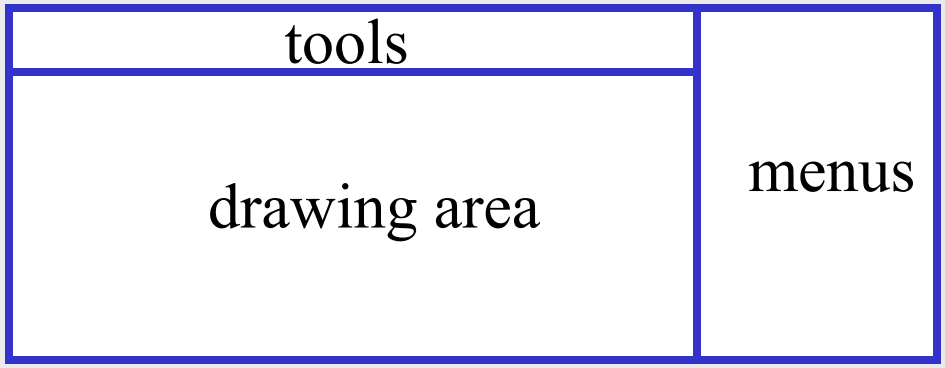


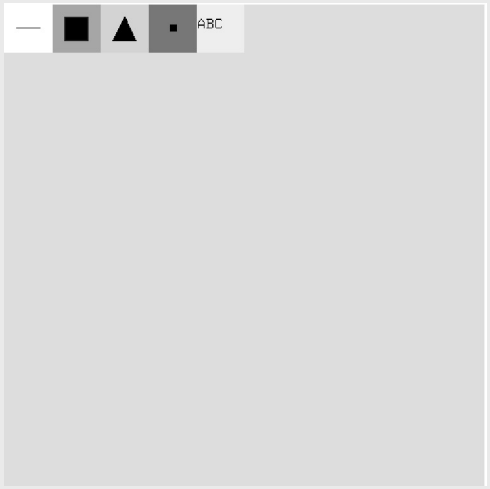
Two Approaches

1. Use of some other buffer to store object ids as the objects are rendered
2. Use of rectangular maps
   - Easy to implement for many applications

Using of Rectangular Maps

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





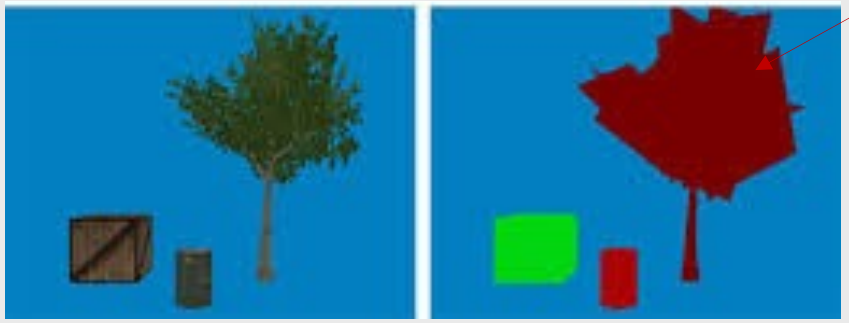
- Easy to look at mouse position and determine which region of screen it is in
- But it works only if the objects don't move on the screen

How to pick if objects are moving?

- We first render the scene as it is
- We then assign a unique color to each object (not necessarily the true color of the object)
- Next we render the scene to a color buffer other than the front buffer, so the results of the rendering are not visible
  - e.g., can specify buffer to write using `glDrawBuffer()`
  - e.g., can render to back buffer
- We then get the mouse position and use `glReadPixels()` function to read the color in the buffer, which is at the position of the mouse
- The returned color gives the id of the object

Clicking here picks tree object since cursor pixel is red

front buffer



back buffer

See **picking** example