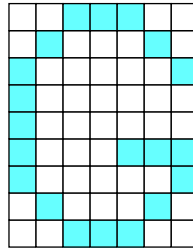
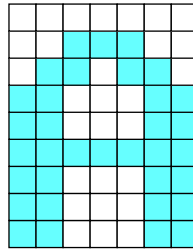


Text:

raster:



stroke:

Compute:
Graphics

Raster text:

- bit blocks
- simple and fast (e.g., `bitblt` transfer)
- awkward to magnify:



Raster fonts in OpenGL:

OpenGL provides a *few* bitmapped character sets:

```
glutBitmapCharacter(GLUT_BITMAP_8_BY_13, int  
character);
```

Placed at “current raster position,” which you can change with `glRasterPos*`.

Current raster position (CRP) moves one char right after `glutBitmapCharacter`.

Can keep explicit track of CRP if you want:

```
glRasterPos2i(rx, ry);  
glutBitmapCharacter(GLUT_BITMAP_8_BY_13, k);  
rx+=glutBitmapWidth(GLUT_BITMAP_8_BY_13, k);
```

Can query fontsize with `glutBitmapWidth`.

Stroke fonts in OpenGL:

OpenGL also provides a few *stroke* character sets:

```
glutStrokeCharacter(GLUT_STROKE_MONO_ROMAN,  
int character);
```

These are polygons, and are subject to all the transformations, etc., in the pipeline.

Arbitrary wierd size; use `glPushMatrix` and `glPopMatrix` to scale.

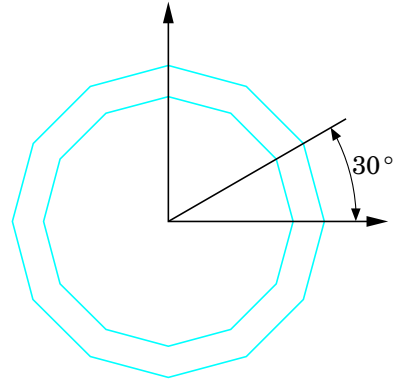
Positioning, needless to say, is a bear; CRP doesn't help...

(Can also use fonts provided by windowing system — but at the expense of portability.)

Display lists and stroke fonts:

```
void MyFont(char c)
{
    switch(c)
    {
        case 'a':
            ...
            break;
        case 'A':
            ...
            break;
        ...
    }
}
```

Display lists and stroke fonts, cont.:



```
case '0':
    /* move to center */
    glTranslatef(0.5,0.5,0.0);

    glBegin(GL_QUAD_STRIP);

    /* 12 vertices */
    for (i=0; i<=12; i++)
    {
        angle = M_PI / 6.0 * i;
        glVertex2f(0.4*cos(angle),
                  0.4*sin(angle));
        glVertex2f(0.5*cos(angle),
                  0.5*sin(angle));
    }
    glEnd();

    /* move to lower right */
    glTranslatef(0.5,-0.5,0.0);

    break;
```

Display lists and stroke fonts, cont.:

```
/* return index of first of 256
   consecutive available ids */
base = glGenLists(256);

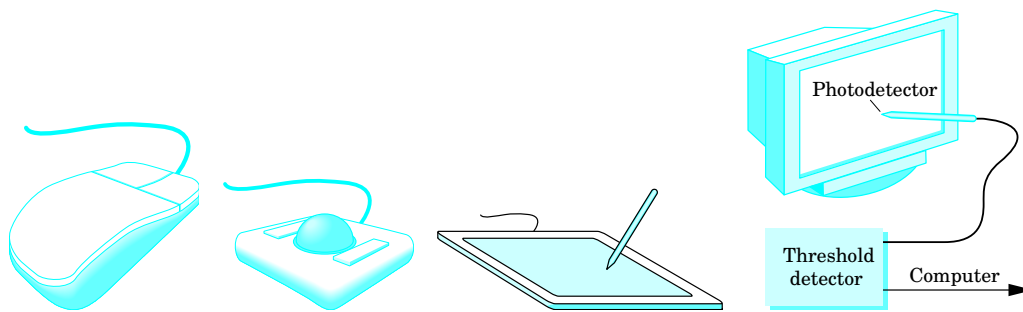
for (i=0; i<256; i++)
{
    glNewList(base+i, GL_COMPILE);
    myFont(i);
    glEndList();
}
```

Logical input devices:

- string (*viz.*, keyboard)
- locator (*viz.*, mouse)
- **choice**
- **pick**
- (dial)
- (stroke)

Ivan Sutherland introduced the basic paradigm that has characterized interactive computer graphics ever since:

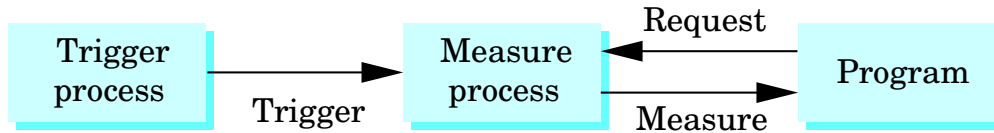
- user sees *object* on display
- user points to (*picks*) the object with an input device:



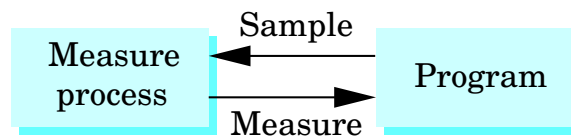
- object changes in response (moves, rotates, morphs, ...)
- repeat

Modes:

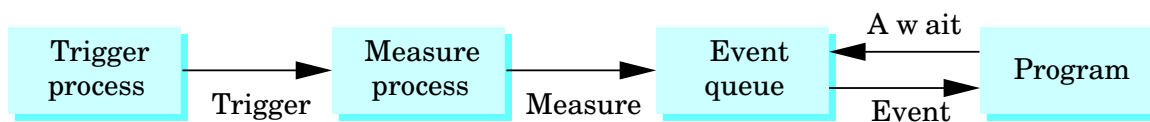
- request: measure not returned until triggered



- sample: immediate – measured when fcn is called; no trigger involved

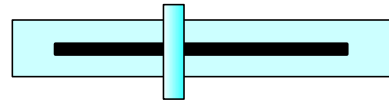


- event: what we've been playing with — event, event queue, callbacks



Choice:

- select one of discrete # of options
- widgets are ubiquitous
- could draw your own:



- but that's a pain, and windows systems generally provide a bunch for you
- and so does GLUT: "pop-up" menus

Menus in OpenGL:

- create menu: `glutCreateMenu(demo_menu);`
- define entries:

```
glutAddMenuEntry("quit",1);  
glutAddMenuEntry("increase size",2);  
glutAddMenuEntry("decrease size",3);
```

- link menu to mouse button:

```
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

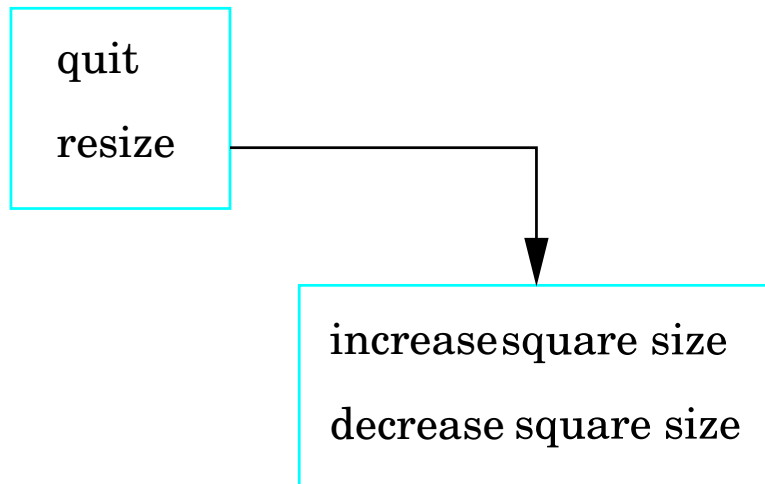
...all in main

Menus in OpenGL, cont.:

- finally, define callback for each menu entry:

```
void demo_menu(int id)
{
    if(id == 1) exit(0);
    else if (id == 2) size = size*2;
    else size = size/2;
    glutPostRedisplay();
}
```

Hierarchical menus in OpenGL:



```
sub_menu = glutCreateMenu(size_menu);}
glutAddMenuEntry("increase size",2);
glutAddMenuEntry("decrease size",3);
glutCreateMenu(top_menu);}
glutAddMenuEntry("quit",1);
glutAddSubMenu("resize",sub_menu);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

...and write `size_menu` and `top_menu` callbacks.

Picking:

- returns identifier of object on the display to user pgm
- generally implemented with same locating device, but with different software interface
- OpenGL: “selection”

Picking is hard:

- functionality is pretty obvious, but there are subtleties...
 - mapping from screen coords to world coords is not 1:1
 - pipeline is forward-only
 - how close before we proclaim something picked?

Solutions:

1. selection and the “hit list”
2. bounding rectangles or “extents”
3. “back buffer”
4. ...all require the application pgm to think

Selection:

- Support picking at the cost of an extra render each time you pick
- Render objects to separate buffer
- OpenGL keeps track of their location *w.r.t. specified volume*

Telling OpenGL what mode to use:

- `glRenderMode(GL_RENDER);` normal rendering
- `glRenderMode(GL_SELECT);` selection mode
- `glRenderMode(GL_FEEDBACK);` obtain list of primitives that were rendered
- use returned value to determine # hits

Selection mode:

- generally enter selection mode at beginning of mouse callback and exit at end of mouse callback
- set up special new clipping volume around the cursor (“close enough”)
- primitives rendered within that volume generate hit
- hits stored in name stack
- when you return to render mode, `glRenderMode` returns number of hits that have been processed
- then deal with them.

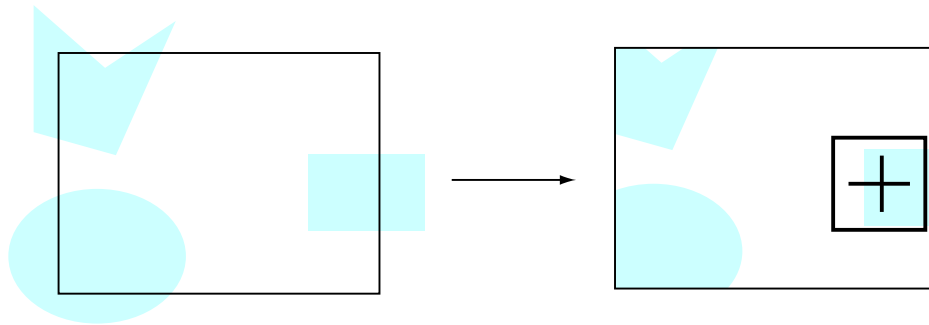
Selection mode, cont.:

Setting the “hit volume:”

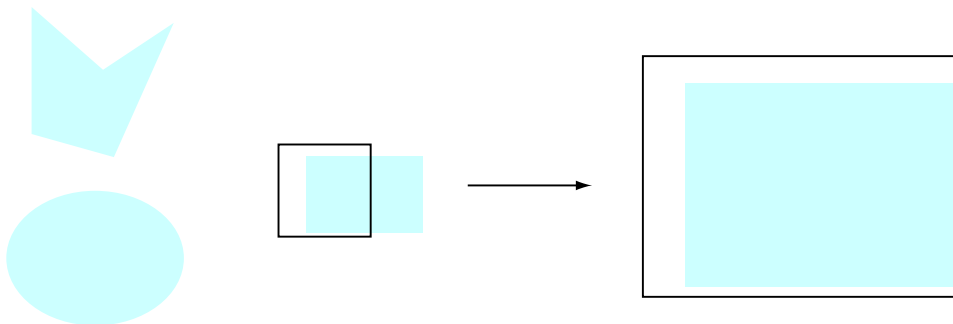
- save current view volume with `glPushMatrix`
- set new view volume with `gluPickMatrix` and `glOrtho`
- deal with picking
- restore original view volume with `glPopMatrix`

gluPickMatrix:

`gluPickMatrix(x,y,w,h,*vp)`; encapsulates a common bit of projection math: automatically creates a projection matrix that restricts “drawing” to a w by h rectangle centered at x,y in the viewport $*vp$:



(a)



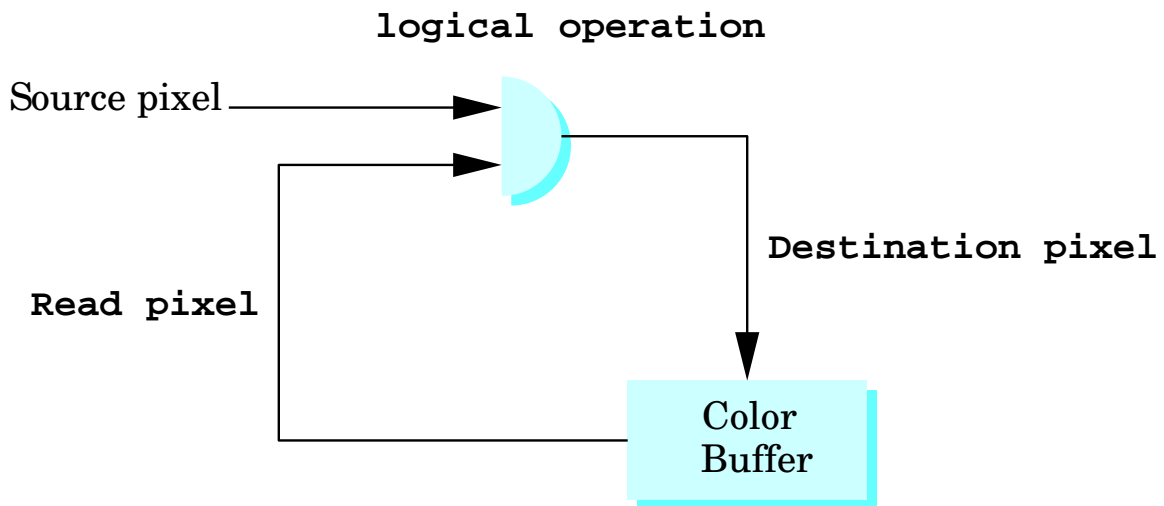
(b)

Selection mode, cont.:

Using the name stack:

- allocate memory for it with `glSelectBuffer`
- initialize it with `glInitNames`
- push things onto it with `glPushName`
- pop things off of it with `glPopName`
- *replace* top element with `glLoadName`

Pixelwise logic operations:

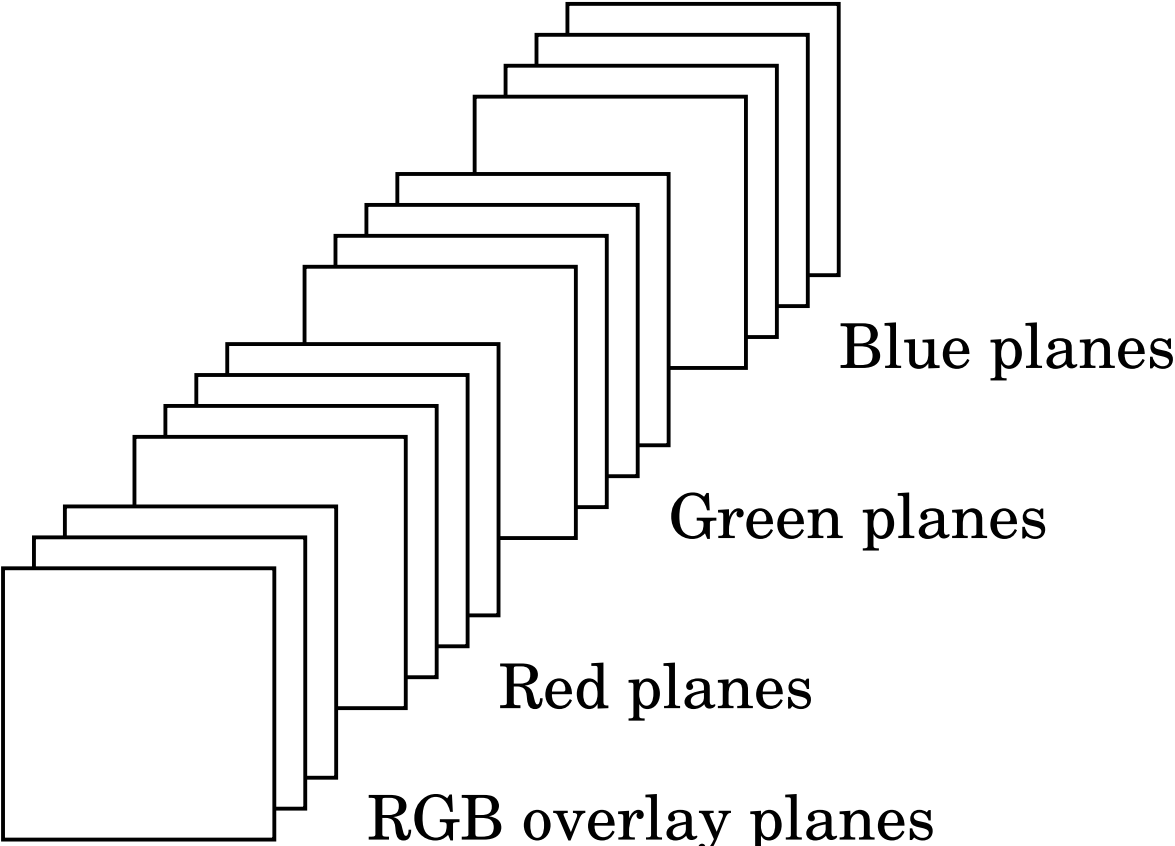


- enable: `glEnable(GL_COLOR_LOGIC_OP);`
- select mode: `glLogicOp([mode]);`
- XOR drawing mode: `glLogicOp(GL_XOR);`

Using logic operations:

```
if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
{
    glLogicOp(GL_COPY);
    glColor3f(0.0, 0.0, 1.0);
    glBegin(GL_LINES);
        glVertex2f(xm, ym);
        glVertex2f(xmm, ymm);
    glEnd();
    glFlush();
    glLogicOp(GL_XOR);
    glColor3f(0.0, 1.0, 0.0);
}
```

Dedicated color overlay planes:



(GLUT supports these, but only in indexed-color mode)

Double buffering:

- 60-100Hz is common; 30Hz minimal
- flicker if drawing overlaps screen refresh
- FRONT and BACK buffers
- work in latter and display former

Double buffering in OpenGL:

- swap with `glutSwapBuffers()`;

- set up with

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
```

- manipulate with `glDrawBuffer`:

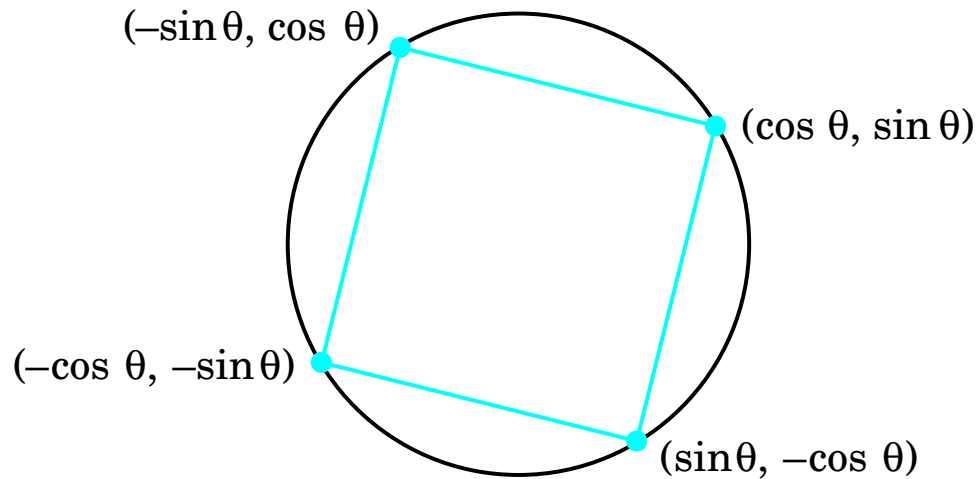
- tells OpenGL where to draw stuff

- `glDrawBuffer(GL_BACK);` – the default

- `glDrawBuffer(GL_FRONT_AND_BACK);`

- `glDrawBuffer(GL_FRONT);` – rare

The mathematics of rotation:



```
/* assumes a global var theta */  
void display(void)  
{  
    glClear (GL_COLOR_BUFFER_BIT);  
    glBegin(gl_polygon);  
        thetar = theta / ((2 * M_PI)/360.0);  
        glVertex2f(cos(thetar), sin(thetar));  
        glVertex2f(-sin(thetar), cos(thetar));  
        glVertex2f(-cos(thetar), -sin(thetar));  
        glVertex2f(sin(thetar), -cos(thetar));  
    glEnd();  
}
```

Animating the rotating square:

```
void idle(void)
{
    theta += 2;
    if (theta>=360.0 theta -= 360.0;
    glutPostRedisplay();
}
```

Turning the animation on and off:

```
void mouse(int btn, int state, int x, int y)
{
    if(btn==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
        glutIdleFunc(idle);
    if(btn==GLUT_MIDDLE_BUTTON && state==GLUT_DOWN)
        glutIdleFunc(NULL);
}
```