**Architecture:**

- pioneered by Silicon Graphics; picked up by graphics chip cos (Nvidia, 3dfx, S3, ATI, ...)
- OpenGL library was designed for this architecture (and vice versa)
- good for opaque textured polygons and lines

**Why pipeline?**

- trade longer latency for higher throughput
- tune stage design for associated task: multiple specialized processors
- tune stage *timing*, too...
Solve light transport through environment:
Ray tracing:

- pretend objects emit light
- still lousy memory locality
- SIMD is a good idea; keep copy of scene on each PU

Alternatives: radiosity, others. (Chs 6, 13)
Hardware:
How we sense light:

Pinhole camera:
We are differently sensitive to different colors:

(a) $R(\lambda)$

(b) $G(\lambda)$

(c) $B(\lambda)$
How real objects look:

What we actually pick up:

\[ A_i = \int S_i(\lambda)C(\lambda)d\lambda \]

...where \( S_i(\lambda) \) is the response of the \( i \) different receptors (cones)
Three-color theory:

aka “additive color model”

...fake that $C(\lambda)$ distribution with combination of R, G, and B:

\[
C = T_1R + T_2G + T_3B
\]

$T_i$ “tristimulus values”

“metameric pairs”
The “color solid” of the additive color model:

- valid for media where light is added to an initially black background: CRTs, transparency film, etc.

cf., args to glColor3f
Additive vs. subtractive color:

(a) Blue, Red, Magenta, Green, Yellow, Cyan, White

(b) Green, Cyan, Red, Black, Blue, Magenta

The subtractive color model:

• valid for media where pigments remove color components from light that is striking the surface: printing, painting, etc.

• primaries are usually the complementary colors: cyan, magenta, and yellow.
Cathode ray tube:

- tickles phosphor on screen, causing it to fluoresce
- “refresh”
- interlaced/not
- raster vs. vector (“random-scan” or “calligraphic”)
Color displays:

- three beams

- three dots of different phosphor at each pixel

- sometimes with “shadow mask” to keep beams confined to right pixels

- many other kinds of displays — e.g., CAVE!

- but CRTs are canonical
Pixellation and the “jaggies”:
Coordinate systems: world and screen

- set \( r_{min}, s_{max} \) with `glutInitWindowPosition`
  (note that 0,0 is top left... raster legacy...)

- set \( r_{max} - r_{min} \) and \( s_{max} - s_{min} \) with `glutInitWindowSize`

- set \( x_{max}, x_{min}, y_{max}, y_{min} \) with `glOrtho`
glOrtho:

- anything not in viewing volume is clipped out
- default: $2 \times 2 \times 2$ cube centered at $(0, 0, 0)$
- whole 3D viewing volume squashed flat to screen window:

(essentially tosses $z$; note that this can make stuff behind the camera visible!)
Setting up glOrtho:

```c
glMatrixMode(GL_PROJECTION);
gluOrtho2D(0.0, 500.0, 0.0, 500.0, -1.0, 1.0);
gluMatrixMode(GL_MODELVIEW);
```

Process:

1. specify target

2. perform operation

(remember: state machine)

Why load the identity and then do the glOrtho?

Why put the MODELVIEW matrix back on top?
Coordinate systems: The viewport

set with `glViewport(x, y, w, h)`

...with the predictable effects if you make its aspect ratio different than that of the screen window:
Last bits of hair:

```c
int main(int argc, char** argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(500,500);
    glutInitWindowPosition(0,0);
    glutCreateWindow("simple");
    init();
    glutDisplayFunc(display);
    glutMainLoop();

}

...where init() looks like this:

void init()
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glColor3f(1.0, 1.0, 1.0);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
    glMatrixMode (GL_MODELVIEW);
}
```
Summary:

main():

- defines callback functions
- opens one or more windows with reqd parameters
- enters event loop (the last executable statement)

init(): sets state variables

- viewing
- attributes

callbacks:

- display
- input
- window
Drawing triangles in 3D:

typedef GLfloat point3[3];

/* this defines a tetrahedron */
point3 vertices[4]=\{\{0,0,0\},\{250,500,100\},
    \{500,250,250\},\{250,100,250\}\};

void drawTriangle(point3 a, point3 b, point3 c)
{
    glBegin(GL_POLYGON);
    glVertex3fv(a); // 'v' for pointer arg
    glVertex3fv(b);
    glVertex3fv(c);
    glEnd();
}
Hidden surface removal with the Z buffer:

Request auxiliary storage:

```c
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
```

and enable the algorithm:

```c
glEnable(GL_DEPTH_TEST);
```

(both in main)

NB: have to clear it too, so glClear becomes:

```c
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```