# Review: Rendering Pipeline

![Rendering Pipeline Diagram](diagram.png)

**Geometry Pipeline**
- set the state once, remains until overwritten
  - `glColor3f(1.0, 1.0, 0.0)` → set color to yellow
  - `glSetClearColor(0.0, 0.0, 0.2)` → dark blue bg
  - `glEnable(LIGHT0)` → turn on light
  - `glEnable(GL_DEPTH_TEST)` → hidden surf.
- tell it how to interpret geometry
  - `glBegin(GL_TRIANGLES)`
  - `glBegin(GL_POLYGON)`
- feed it vertices
  - `glVertex3f(-1.0, 1.0, -1.0)`
  - `glVertex3f(1.0, 0.0, -1.0)`
  - `glVertex3f(0.0, 1.0, -1.0)`
- set state as needed
  - `glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, 1)`
- timing
  - `glFlush()`
- idle processing, idle events
  - `glutIdleFunc()`

## OpenGL: Geometric Primitives

- `glPointSizex(float size);`
- `glLineWidthis(float width);`
- `glColor3f(float r, float g, float b);`
- `glLineWidth(float width);`
- `glPointSize(float size);`

## GLUT: OpenGL Utility Toolkit
- developed by Mark Kilgard (also from SGI)
- simple, portable window manager
- opening windows
- handling graphics contexts
- handling input with callbacks
- keyboard, mouse, window reshape events
- timing
- idle processing, idle events
- designed for small/medium size applications
- distributed as binaries
- free, but not open source

### OpenGL (briefly)
- API to graphics hardware
- based on RIS, GL by SGI
- designed to exploit hardware optimized for display and manipulation of 3D graphics
- implemented on many different platforms
- low level, powerful flexible
- pipeline processing
- set state as needed

### GLUT Callback Functions
- you supply these kind of functions
  - `void reshape(int w, int h);`
  - `void mouse(int btn, int state, int x, int y);`
  - `void idle();`
  - `void display();`
  - `// you supply these kind of functions`
  - `glutShapeFunc reshape)(w, h);`
  - `glutKeyboardFunc(keyboard);`
  - `glutMotionFunc(mouse);`
  - `glutMiddleFunc(idle);`
  - `glutDisplayFunc(display);`

### Event-Driven Programming
- main loop not under your control
  - vs. batch mode where you control the flow
  - control flow through event callbacks
  - redraw the window now
  - key was pressed
  - mouse moved
  - callback functions called from main loop when events occur
  - mouse/key state setting vs. redrawing

### Code Sample

```c
void display()
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0, 1.0, 0.0);
    glVertex3f(-1.0, 1.0, -1.0);
    glVertex3f(1.0, 0.0, -1.0);
    glVertex3f(0.0, 1.0, -1.0);
    glEnd();
    glutSwapBuffers();
}
```

### GLUT Example 1

```c
#include <GLUT/glut.h>
void display()
{
    glClearColor(0.0, 0.0, 0.0, 1);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_TRIANGLES);
        glVertex3f(-1.0, 1.0, -1.0);
        glVertex3f(1.0, 0.0, -1.0);
        glVertex3f(0.0, 1.0, -1.0);
    glEnd();
    glutSwapBuffers();
}
```

### GLUT Example 2

```c
#include <GLUT/glut.h>
void display()
{
    glRotatef(0.1, 0, 0, 1);
    glClearColor(0.0, 0.0, 0.0, 1);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_TRIANGLES);
        glVertex3f(-1.0, 1.0, -1.0);
        glVertex3f(1.0, 0.0, -1.0);
        glVertex3f(0.0, 1.0, -1.0);
    glEnd();
    glutSwapBuffers();
}
```

### GLUT Example 3

```c
#include <GLUT/glut.h>
void display()
{
    glRotatef(0.1, 0, 0, 1);
    glClearColor(0.0, 0.0, 0.0, 1);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 0.0, 0.0);
    glBegin(GL_TRIANGLES);
        glVertex3f(-1.0, 1.0, -1.0);
        glVertex3f(1.0, 0.0, -1.0);
        glVertex3f(0.0, 1.0, -1.0);
    glEnd();
    glutSwapBuffers();
}
```

### Redrawing Display
- display only redrawn by explicit request
  - `glutPostRedisplay();`
- default window resize callback does this
  - idle called from main loop when no user input
  - good place to request redraw
  - will call display next time through event loop
  - should return control to main loop quickly
  - continues to rotate even when no user action
Keyboard/Mouse Callbacks
- again, do minimal work
- consider keypress that triggers animation
- do not have loop calling display in callback!
  - what if user hits another key during animation?
  - instead, use shared/global variables to keep track of state
  - yes, OK to use globals for this!
- then display function just uses current variable value

GLUT Example 4
```c
#include <GLUT/glut.h>

void doKey(unsigned char key, int x, int y) {
    if (key == 'q') {
        glutPostRedisplay();
    } else if ('e' == key) {
        glutPostRedisplay();
    }
}
```

Transformations
- transforming an object = transforming all its points
- transforming a polygon = transforming its vertices

Matrix Representation
- represent 2D transformation with matrix
  - multiply matrix by column vector
  - apply transformation to point
    
    \[
    \begin{bmatrix}
    x' \\
    y'
    \end{bmatrix}
    =
    \begin{bmatrix}
    a & b \\
    c & d
    \end{bmatrix}
    \begin{bmatrix}
    x \\
    y
    \end{bmatrix}
    \]
- transformations combined by multiplication
  
  \[
  \begin{bmatrix}
  x'' \\
  y''
  \end{bmatrix}
  =
  \begin{bmatrix}
  a & b & e \\
  c & d & f
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y
  \end{bmatrix}
  \]
- matrices are efficient, convenient way to represent sequence of transformations!

Scaling
- scaling a coordinate means multiplying each of its components by a scalar
- uniform scaling means this scalar is the same for all components:
  
  \[
  \begin{bmatrix}
  x' \\
  y'
  \end{bmatrix}
  =
  \begin{bmatrix}
  ax \\
  by
  \end{bmatrix}
  \]
- how can we represent this in matrix form?

2D Rotation
- counterclockwise
- RHS
  
  \[
  x' = x \cos(\theta) - y \sin(\theta) \\
  y' = x \sin(\theta) + y \cos(\theta)
  \]

2D Rotation From Trig Identities
- easy to capture in matrix form:
  
  \[
  \begin{bmatrix}
  x' \\
  y'
  \end{bmatrix}
  =
  \begin{bmatrix}
  \cos(\theta) & -\sin(\theta) \\
  \sin(\theta) & \cos(\theta)
  \end{bmatrix}
  \begin{bmatrix}
  x \\
  y
  \end{bmatrix}
  \]
- even though \(\sin(q)\) and \(\cos(q)\) are nonlinear functions of \(q\):
  - \(x'\) is a linear combination of \(x\) and \(y\)
  - \(y'\) is a linear combination of \(x\) and \(y\)

2D Rotation: Another Derivation

Transformations
2D Rotation: Another Derivation

Shear

- shear along x axis
  - push points to right in proportion to height
  \[
  \begin{pmatrix}
  x' \\
  y'
  \end{pmatrix} = \begin{pmatrix}
  1 & s_x & 0 \\
  0 & 1 & 0
  \end{pmatrix}
  \begin{pmatrix}
  x \\
  y
  \end{pmatrix}
  \]

Shear

- shear along x axis
  - push points to right in proportion to height
  \[
  \begin{pmatrix}
  x' \\
  y'
  \end{pmatrix} = \begin{pmatrix}
  1 & s_x & 0 \\
  0 & 1 & 0
  \end{pmatrix}
  \begin{pmatrix}
  x \\
  y
  \end{pmatrix}
  \]

Reflection

- reflect across x axis
  - mirror
  \[
  \begin{pmatrix}
  x' \\
  y'
  \end{pmatrix} = \begin{pmatrix}
  1 & 0 & 0 \\
  0 & 1 & 0
  \end{pmatrix}
  \begin{pmatrix}
  x \\
  y
  \end{pmatrix}
  \]

2D Translation

- linear transformations are combinations of
  - shear
  - scale
  - rotate
  - reflect

- properties of linear transformations
  - satisfies T(ax+by) = aT(x) + bT(y)
  - origin maps to origin
  - lines map to lines
  - parallel lines remain parallel
  - ratios are preserved
  - closed under composition

Homogeneous Coordinates

- point in 2D cartesian

Homogeneous Coordinates Geometrically

- homogenize to convert homog. 3D point to cartesian 2D point:
  - divide by w to get (x/w, y/w, 1)
  - projects line to point onto w=1 plane
  - like normalizing, one dimension up
  - when w=0, consider it as direction
  - points at infinity
  - these points cannot be homogenized
  - lies on x-y plane
  - (0,0,0) is undefined

Affine Transformations

- affine transforms are combinations of
  - linear transformations
  - translations

- properties of affine transformations
  - origin does not necessarily map to origin
  - lines map to lines
  - parallel lines remain parallel
  - ratios are preserved
  - closed under composition
Homogeneous Coordinates Summary

- may seem unintuitive, but they make graphics operations much easier
- allow all affine transformations to be expressed through matrix multiplication
  - we’ll see even more later...
- use 3x3 matrices for 2D transformations
  - use 4x4 matrices for 3D transformations