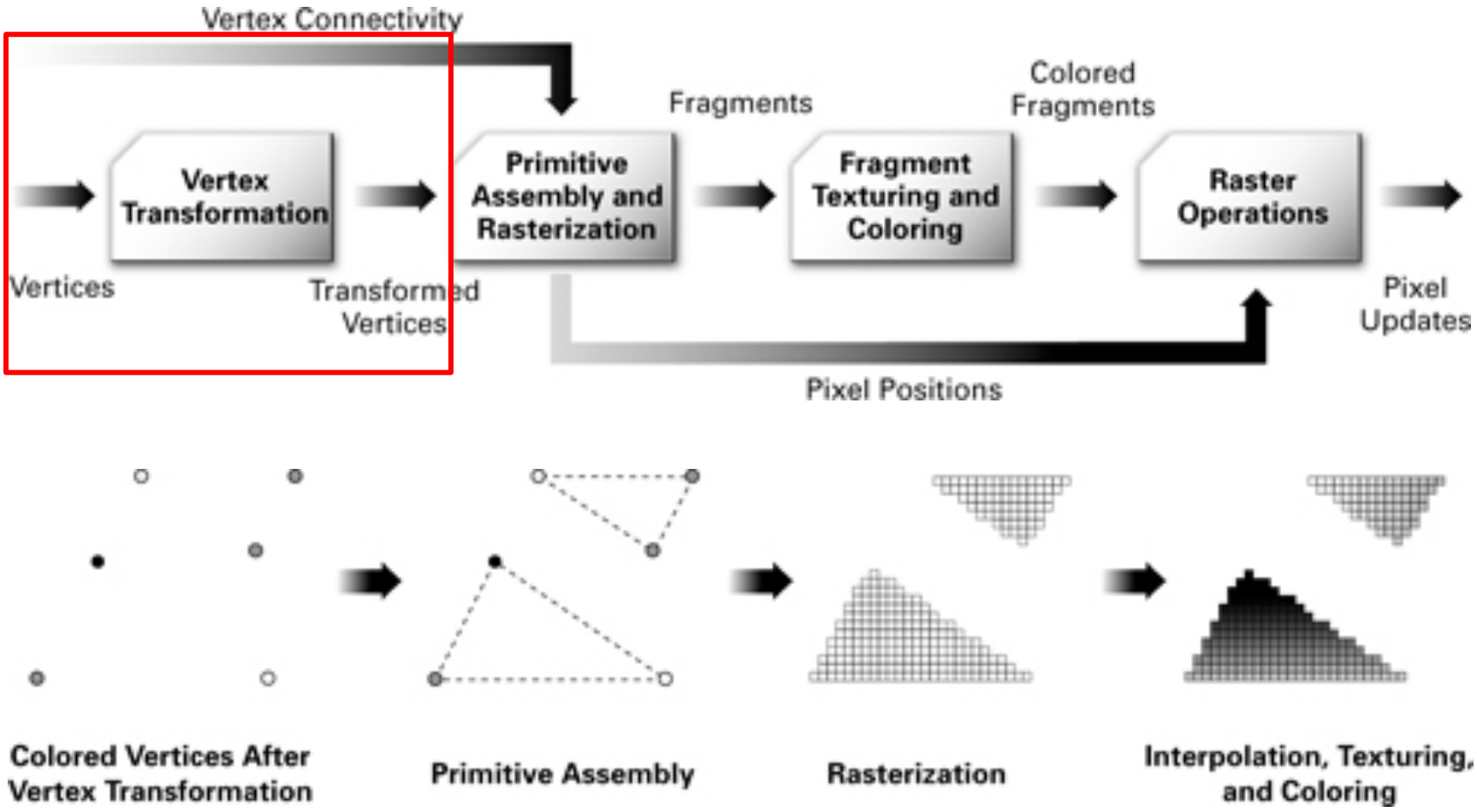
# CSC 4356
# Interactive Computer Graphics
## Lecture 6: Transformation in OpenGL
## Rasterization: Line Drawing

Jinwei Ye

http://www.csc.lsu.edu/~jye/CSC4356/

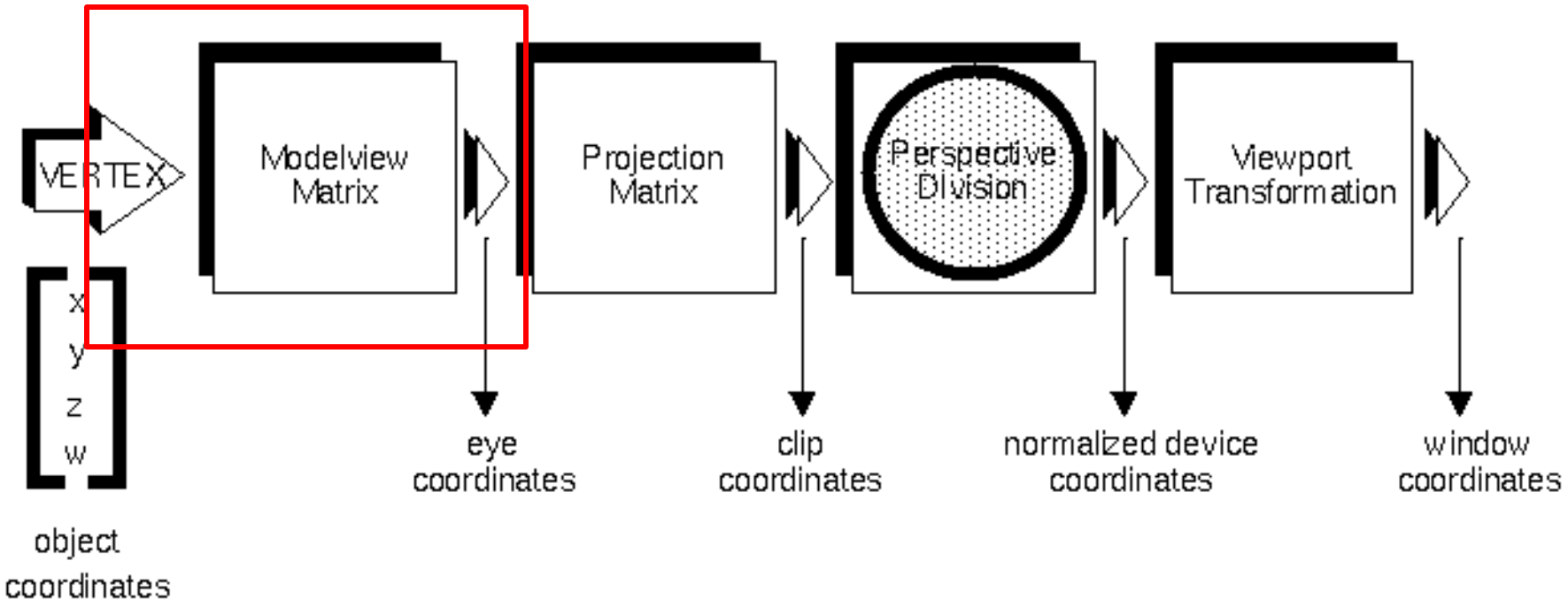
Tue & Thu: 10:30 - 11:50am
218 Tureaud Hall

# Graphics Rendering pipeline

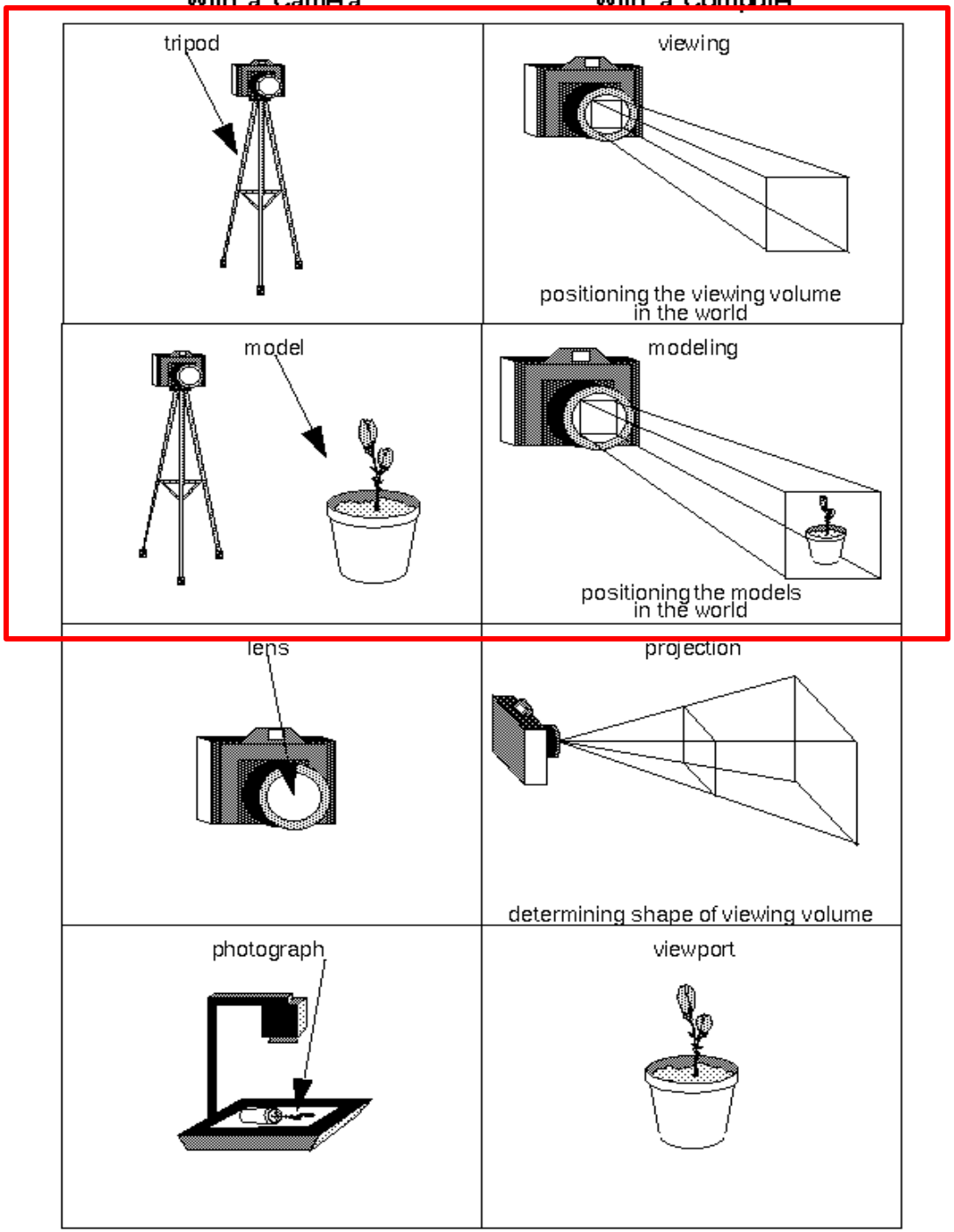



*Images from Nvdia CG tutorial: http://developer.download.nvidia.com/CgTutorial/*

# Vertex Transformation



**ModelView Transformation**



# ModelView Transformations

- Modeling Transformation
  - Position and orient the model in your scene
  - rotate, translate, scale
  - glRotatef(), glTranslatef(), glScalef()
- Viewing Transformation
  - Equivalent to position the camera
  - OpenGL always assume camera at (0,0,0)
  - Instead moving the camera, we have to move the scene
  - gluLookAt()

# OpenGL Matrix Stack

- OpenGL store *stacks* of 4 X 4 matrices
  - Stack: first in, last out
  - Initially, each stack contains one matrix, an identity matrix.
- glMatrixMode() specifies which matrix is the current matrix
  - GL_MODELVIEW, GL_PROJECTION, GL_TEXTURE, etc.
- Use glGet(GL_MATRIX_MODE) to inquire the current matrix stack

# Matrix Stack Operations

- OpenGL manages the matrix stack by push, pop, multiply matrices on top of the stack
  - glLoadMatrix()/glLoadIdentity()
  - glPushMatrix()
  - glPopMatrix()
  - glMultiMatrix()
  - glRotatef(), glTranslatef(), glScalef()
- *All vertices are multiplied by the top of stack*

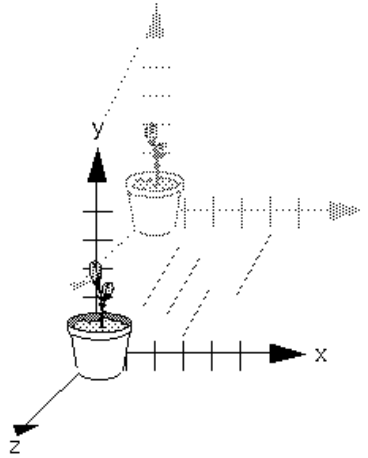
# Functions

- glLoadMatrix{fd}(m1,m2,…,m16)
  - Set matrix M as the current matrix in stack
- glLoadIdentity()
  - Clear the current matrix as a 4 x 4 identity matrix
- glMultMatrix{fd}(m1,m2,…,m16)
  - Multiply matrix M onto the top of stack

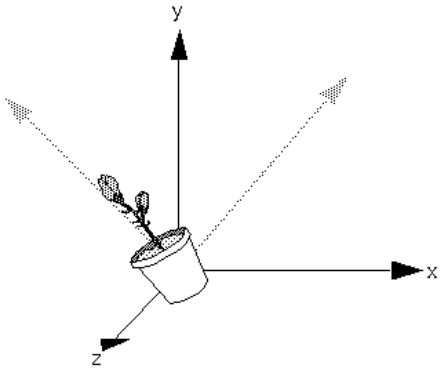
$$\mathbf{M} = \begin{bmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{bmatrix}$$

# Functions

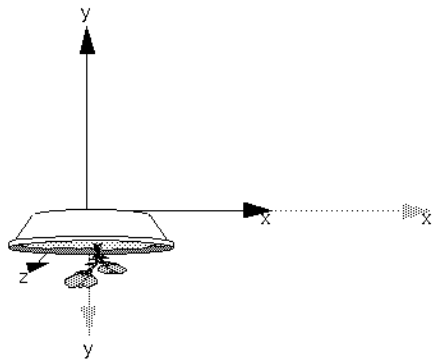
- glTranslate{fd}(x,y,z)
  - Move the object by (x,y,z)




- glRotate{fd}(angle,x,y,z)
  - Rotate the object by the angle about axis (x,y,z)
  - Direction: counter-clockwise




- glScale{fd}($s_x$,$s_y$,$s_z$)
  - Scale the x, y, z coordinate of the object by $s_x$,$s_y$,$s_z$
  - Reflection included



# Functions

- glPushMatrix()
  - pushes the current matrix stack down by one, duplicating the current matrix
  - the matrix on top of the stack is identical to the one below it
- glPopMatrix()
  - pops the current matrix stack
  - replacing the current matrix with the one below it on the stack
- Need to use in pair
- Similar to save(Push)/load(Pop)
- Useful when transforming multiple objects in different ways

# Example

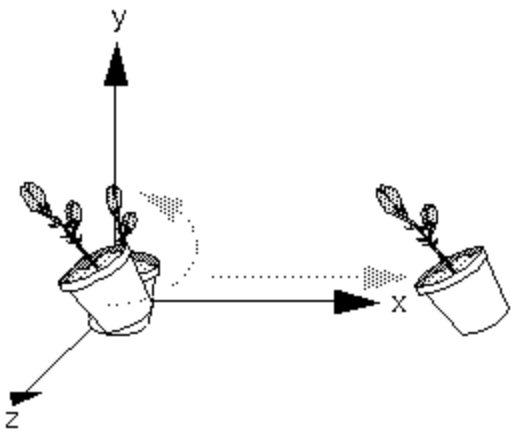
```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();       //load a 4x4 identity matrix
glMultMatrixf(N);       //apply transformation N
glMultMatrixf(M);       //apply transformation M
glMultMatrixf(L);       //apply transformation L

glBegin(GL_POINTS);
glVertex3f(v);          //draw transformed vertex v
glEnd();
```

**First apply transformation L, then M, and finally N**

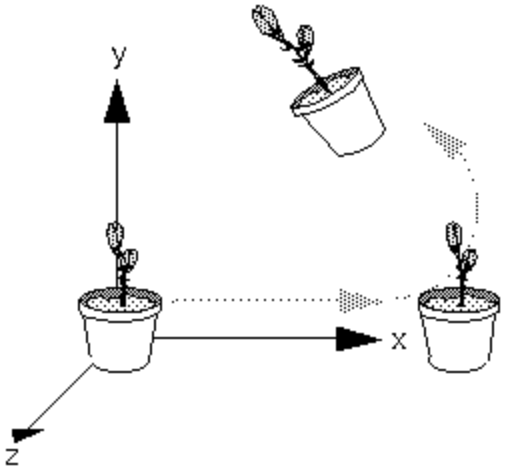
$$v' = NMLv$$

# Which Order?

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranlatef(xt,yt,zt);
glRotatef(θ,x,y,z);

DrawObject;
```




Rotation First



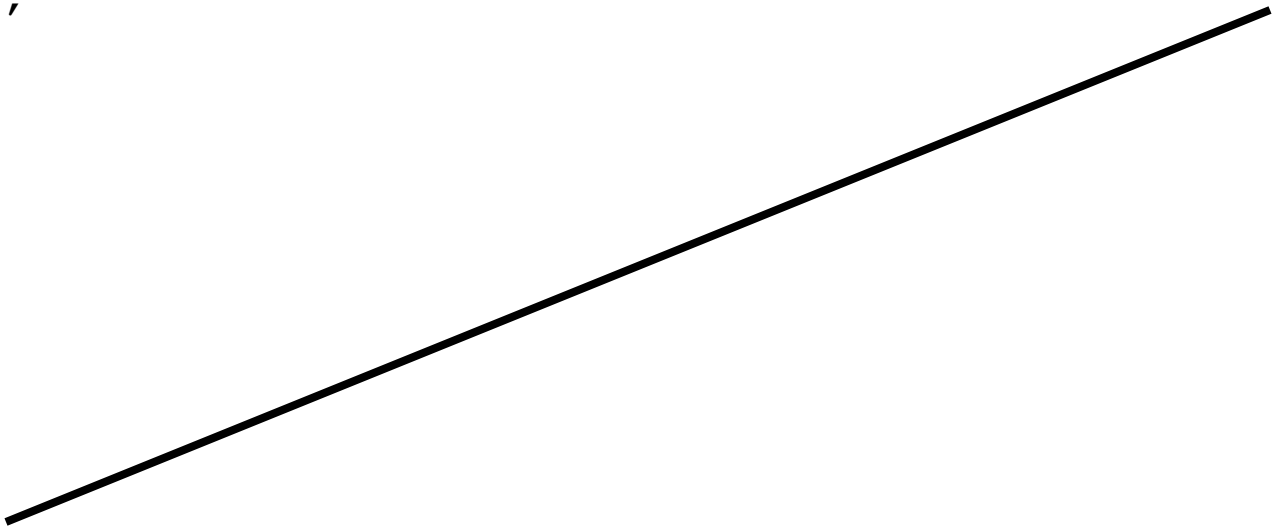

Translation First

# Transform Multiple Objects

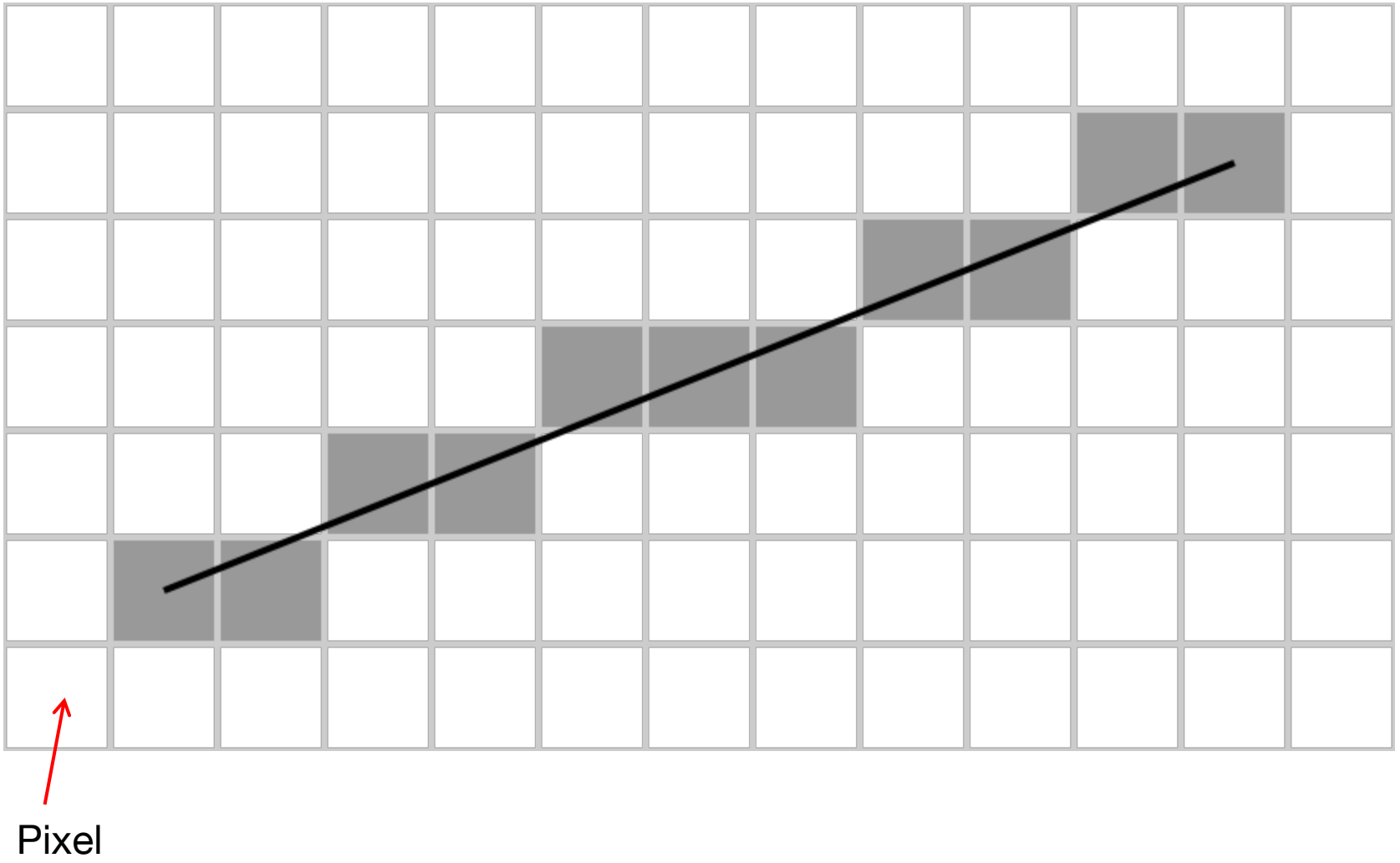
```
glPushMatrix();
Transformations for object one;
DrawObject(ONE);
glPopMatrix();

glPushMatrix();
Transformations for object two;
DrawObject(TWO);
glPopMatrix();

…
```

# How does OpenGL draw a line?

```
glBegin(GL_LINES);
        glVertex3f (x1, y1, z1);
        glVertex3f (x2, y2, z2);
glEnd();
```

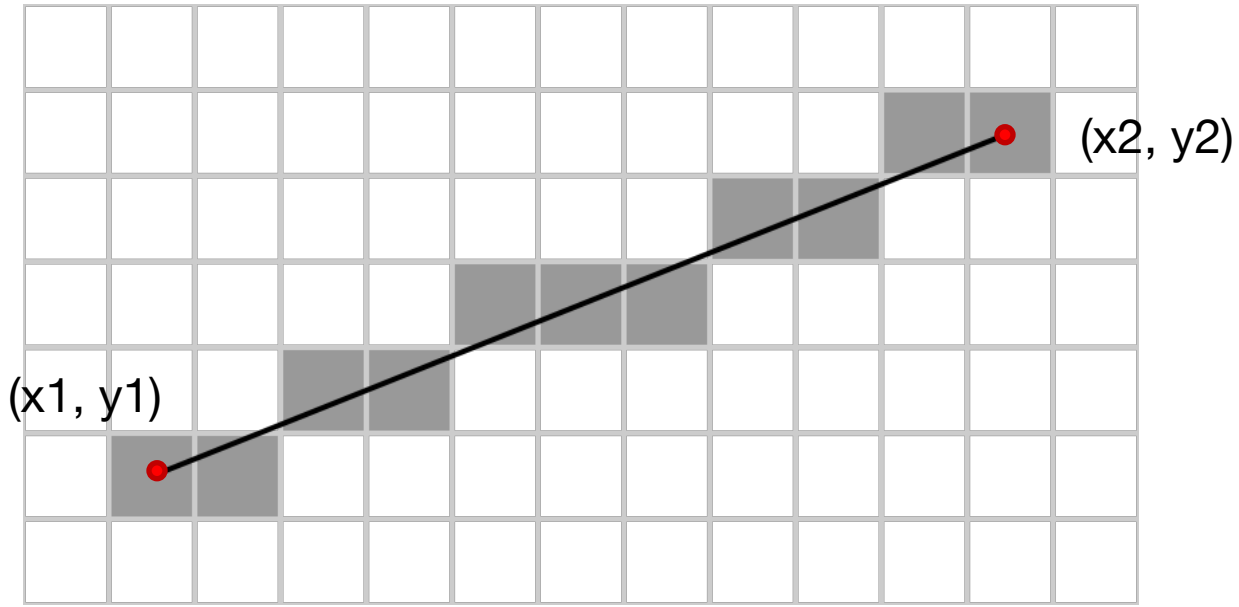


# Everything is rasterized!



# Line Rasterization Problem

- Given:
  - Two endpoints: integers (x1, y1) & (x2, y2)
- Identify:
  - Which pixels (x, y) to display for the line?



# Requirements

- Transform **continuous** primitive into **discrete** samples
- Uniform thickness & brightness
- Continuous appearance
- No gaps
- Accuracy
- Speed

# DDA Line Drawing

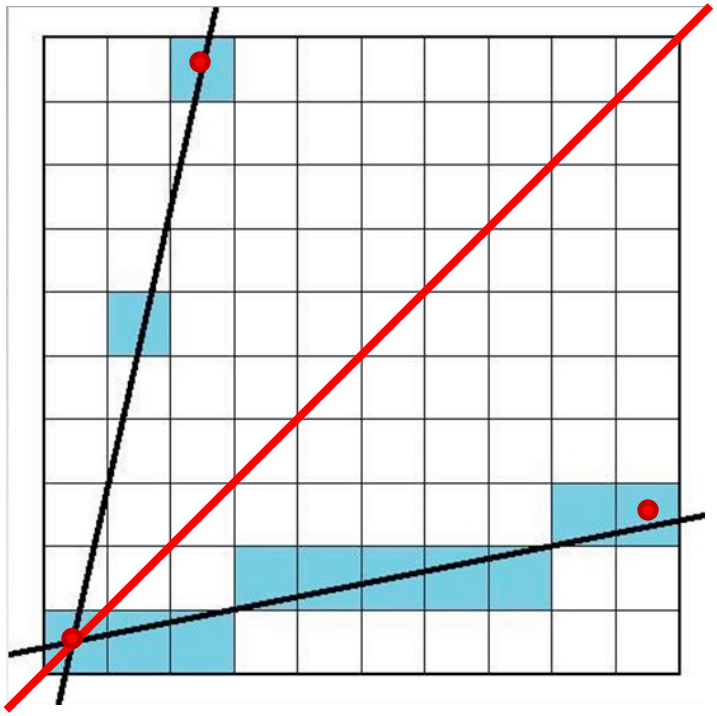
- DDA stands for Digital Differential Analyzer, the name of a class of old machines used for plotting functions

- Slope-intercept form of a line:

  $y = mx + b$

  slope: $m = dy/dx$

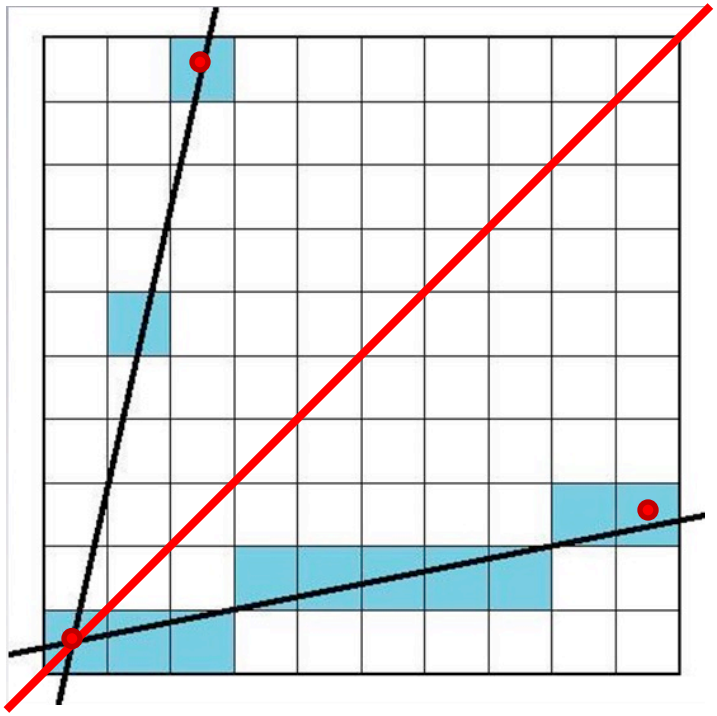
  intercept: b is where the line intersects the y-axis



# DDA Line Drawing

- Basic idea: If we increment the x coordinate by one pixel at each step, the slope of the line tells us how much to increment y per step
  - i.e., dx = 1, dy = m

  (because m = dy/dx)



# DDA Line Drawing

- This only works if m <= 1
  - otherwise there are gaps

- Solution: Reverse axes and step in y direction
  - Now dy = 1, dx = 1/m <1