

# Primitives Drawing Algorithms

**Dr.E.A.Zanaty**

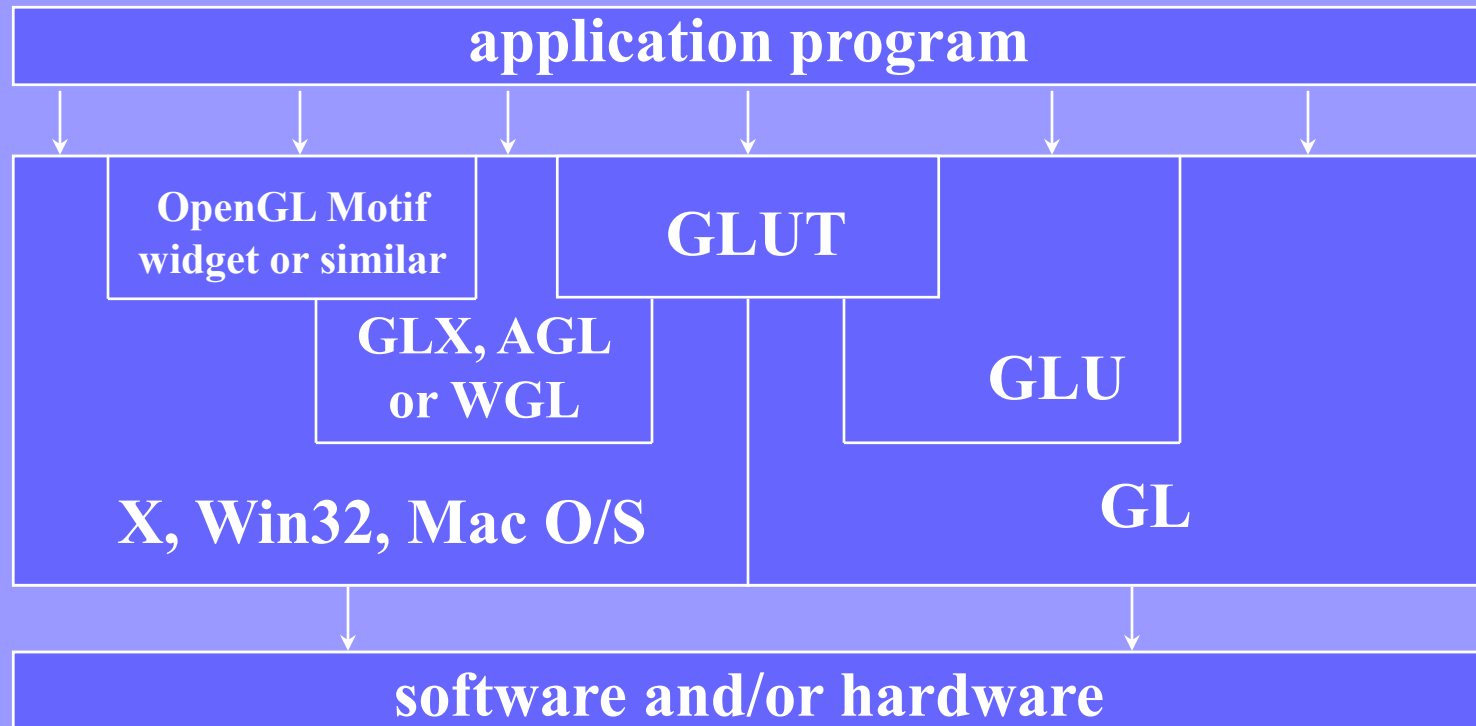
# What is OpenGL?

- A software interface to graphics hardware
- Graphics rendering API (Low Level)
  - High-quality color images composed of geometric and image primitives
  - Window system independent
  - Operating system independent

# OpenGL Libraries

- **GL (Graphics Library):** Library of 2-D, 3-D drawing primitives and operations
  - API for 3-D hardware acceleration
- **GLU (GL Utilities):** Miscellaneous functions dealing with camera set-up and higher-level shape descriptions
- **GLUT (GL Utility Toolkit):** Window-system independent toolkit with numerous utility functions, mostly dealing with user interface

# Software Organization



# OpenGL Syntax

- Functions have prefix **gl** and initial capital letters for each word
  - **glClearColor()**, **glEnable()**, **glPushMatrix()** ...
- **glu** for **GLU** functions
  - **gluLookAt()**, **gluPerspective()** ...
- Constants begin with **GL\_**, use all capital letters
  - **GL\_COLOR\_BUFFER\_BIT**, **GL\_PROJECTION**, **GL\_MODELVIEW** ...
- Extra letters in some commands indicate the number and type of variables
  - **glColor3f()**, **glVertex3f()** ...
- OpenGL data types
  - **GLfloat**, **GLdouble**, **GLint**, **GLenum**, ...
- Underlying storage mode is the same
- Easy to create overloaded functions in C++ but issue is efficiency

# GLU routines

## - Manage Quadrics and curves, surfaces

- *GLUquadricObj\* gluNewQuadric (void);*

*Creates a new quadrics object and returns a pointer to it. A null pointer is returned if the routine fails.*

- *void gluDeleteQuadric (GLUquadricObj \*qobj);*

*Destroys the quadrics object qobj and frees up any memory used by it.*

- *void gluQuadricCallback (GLUquadricObj \*qobj, GLenum which, void (\*fn)());*

- Use **gluNewNurbsRenderer()** to create a pointer to a NURBS object

# How to install GLUT?

- Download GLUT
  - <http://www.opengl.org/resources/libraries/glut.html>
- Copy the files to following folders:
  - glut.h → VC/include/gl/
  - glut32.lib → VC/lib/
  - glut32.dll → windows/system32/
- Header Files:
  - #include <GL/glut.h>
  - #include <GL/gl.h>
  - Include glut automatically includes other header files

# GLUT

- GLUT (OpenGL Utility Toolkit)
  - An auxiliary library
    - A portable windowing API
    - Easier to show the output of your OpenGL application
    - Not officially part of OpenGL
  - Handles:
    - Window creation,
    - OS system calls
      - Mouse buttons, movement, keyboard, etc...
    - Callbacks



# How to install GLUT?

- Download GLUT
  - <http://www.opengl.org/resources/libraries/glut.html>
- Copy the files to following folders:
  - glut.h → VC/include/gl/
  - glut32.lib → VC/lib/
  - glut32.dll → windows/system32/
- Header Files:
  - #include <GL/glut.h>
  - #include <GL/gl.h>
  - Include glut automatically includes other header files

# Glut Routines

- **Initialization:** **glutInit()** processes (and removes) commandline arguments that may be of interest to glut and the window system and does general initialization of Glut and OpenGL
  - Must be called before any other glut routines
- **Display Mode:** The next procedure, **glutInitDisplayMode()**, performs initializations informing OpenGL how to set up the frame buffer.

Display Mode	Meaning
– <b>GLUT_RGB</b>	Use RGB colors
– <b>GLUT_RGBA</b>	Use RGB plus alpha (for transparency)
– <b>GLUT_INDEX</b>	Use indexed colors (not recommended)
– <b>GLUT_DOUBLE</b>	Use double buffering (recommended)
– <b>GLUT_SINGLE</b>	Use single buffering (not recommended)
– <b>GLUT_DEPTH</b>	Use depthbuffer (for hidden surface removal.)

# Glut Routines

- Window Setup
  - **glutInitWindowSize**(int width, int height)
  - **glutInitWindowPosition**(int x, int y)
  - **glutCreateWindow**(char\* title)
  - **glutReshapeFunc**(void (\*func)(int w, int h)) indicates what action should be taken when the window is resized.

# Glut 3D Routines

cone	icosahedron	teapot
cube	octahedron	tetrahedron
dodecahedron	sphere	torus

GLUT includes several routines for drawing these three-dimensional objects:

```
void glutWireCube(GLdouble size);
```

```
void glutSolidCube(GLdouble size);
```

```
void glutWireSphere(GLdouble radius, GLint slices, GLint stacks);
```

```
void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);
```

# GLUT Callback Functions

- **Callback function** : Routine to call when an **event** happens
  - Window resize or redraw
  - User input (mouse, keyboard)
  - Animation (render many frames)
- **“Register” callbacks with GLUT**
  - `glutDisplayFunc( my_display_func );`
  - `glutIdleFunc( my_idle_func );`
  - `glutKeyboardFunc( my_key_events_func );`
  - `glutMouseFunc ( my_mouse_events_func );`

# Rendering Callback

- Callback function where all our drawing is done
- Every GLUT program must have a display callback
- `glutDisplayFunc( my_display_func );` */\* this part is in main.c \*/*

```
void my_display_func (void )
{
    glClear( GL_COLOR_BUFFER_BIT );
    glBegin( GL_TRIANGLE );
        glVertex3fv( v[0] );
        glVertex3fv( v[1] );
        glVertex3fv( v[2] );
    glEnd();
    glFlush();
}
```

# Idle Callback

- Use for animation and continuous update
  - Can use *glutTimerFunc* or *timed callbacks* for animations
- `glutIdleFunc( idle );`

```
void idle( void )
```

```
{  
    /* change something */  
    t += dt;  
    glutPostRedisplay();  
}
```

# User Input Callbacks

- Process user input
- `glutKeyboardFunc( my_key_events );`

```
void my_key_events (char key, int x, int y )
{
    switch ( key ) {
        case 'q' : case 'Q' :
            exit ( EXIT_SUCCESS);
            break;
        case 'r' : case 'R' :
            rotate = GL_TRUE;
            break;
    }
}
```



# Mouse Callback

- Captures mouse press and release events
- `glutMouseFunc( my_mouse );`

```
void myMouse(int button, int state, int x, int y)
```

```
{
```

```
    if (button == GLUT_LEFT_BUTTON && state ==  
        GLUT_DOWN)
```

```
    {
```

```
        ...
```

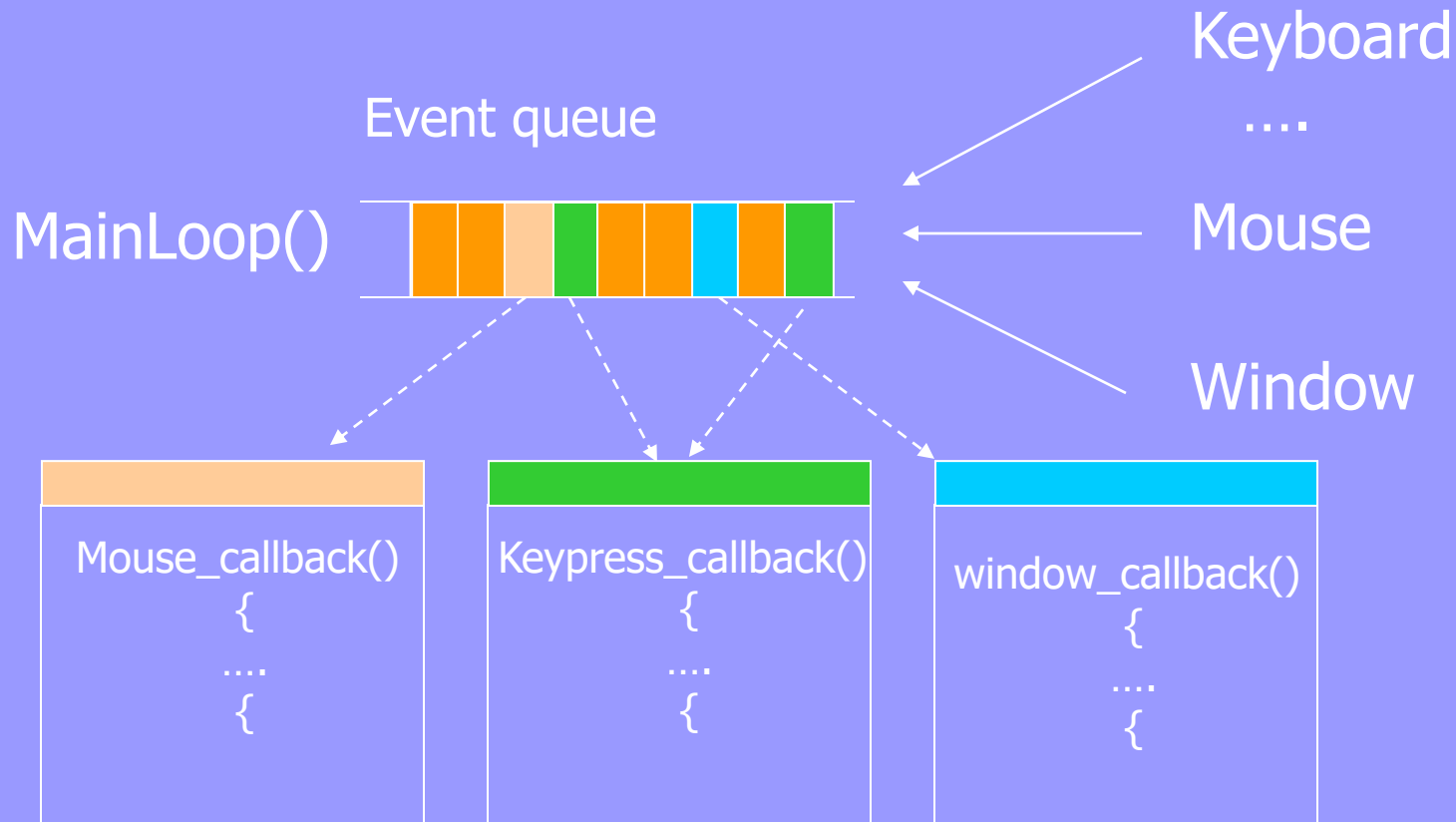
```
    }
```

```
}
```

# More Callbacks

- **glutReshapeFunc(void (\*func)(int w, int h))** indicates what action should be taken when the window is resized.
- **glutKeyboardFunc(void (\*func)(unsigned char key, int x, int y))** and **glutMouseFunc(void (\*func)(int button, int state, int x, int y))** allow you to link a keyboard key or a mouse button with a routine that's invoked when the key or mouse button is pressed or released.
- **glutMotionFunc(void (\*func)(int x, int y))** registers a routine to call back when the mouse is moved while a mouse button is also pressed.
- **glutIdleFunc(void (\*func)(void))** registers a function that's to be executed if no other events are pending - for example, when the event loop would otherwise be idle

# Event Queue



# simple.c

```
#include <GL/glut.h>

void mydisplay() {
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POLYGON);
        glVertex2f(-0.5, -0.5);
        glVertex2f(-0.5, 0.5);
        glVertex2f(0.5, 0.5);
        glVertex2f(0.5, -0.5);
    glEnd();
    glFlush();
}

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

# Man()

```
#include <GL/glut.h>
#include <GL/gl.h>

void main(int argc, char** argv)
{
    int mode = GLUT_RGB|GLUT_DOUBLE;
    glutInitDisplayMode( mode );
    glutInitWindowSize( 500,500 );
    glutCreateWindow( "Simple" );
    init();
    glutDisplayFunc(mydisplay );
    glutKeyboardFunc( key );
    glutMainLoop();
}
```

# Sample Program

```
#include <GL/glut.h>
#include <GL/gl.h>

void main(int argc, char** argv)
{
    int mode = GLUT_RGB | GLUT_DOUBLE ;
    glutInitDisplayMode( mode );
    glutInitWindowSize( 500,500 );
    glutCreateWindow( "Simple" );
    init();
    glutDisplayFunc( mydisplay );
    glutKeyboardFunc( key );
    glutMainLoop();
}
```

← **Specify the display  
Mode – RGB or color  
Index, single or double  
Buffer**

# Sample Program

```
#include <GL/glut.h>
#include <GL/gl.h>

void main(int argc, char** argv)
{
    int mode = GLUT_RGB | GLUT_DOUBLE;
    glutInitDisplayMode( mode );
    glutInitWindowSize( 500,500 );
    glutCreateWindow( "Simple" );
    init();
    glutDisplayFunc( mydisplay );
    glutKeyboardFunc( key );
    glutMainLoop();
}
```

← **Create a window  
Named "simple"  
with resolution  
500 x 500**

# Sample Program

```
#include <GL/glut.h>
#include <GL/gl.h>

void main(int argc, char** argv)
{
    int mode = GLUT_RGB | GLUT_DOUBLE;
    glutInitDisplayMode( mode );
    glutInitWindowSize( 500,500 );
    glutCreateWindow( "Simple" );
    init();
    glutDisplayFunc( mydisplay );
    glutKeyboardFunc( key );
    glutMainLoop();
}
```

← **Your OpenGL initialization  
code (Optional)**



# Sample Program

```
#include <GL/glut.h>
#include <GL/gl.h>

void main(int argc, char** argv)
{
    int mode = GLUT_RGB | GLUT_DOUBLE;
    glutInitDisplayMode( mode );
    glutInitWindowSize( 500,500 );
    glutCreateWindow( "Simple" );
    init();
    glutDisplayFunc( mydisplay );
    glutKeyboardFunc (key) ;
    glutMainLoop();
}
```

← **Register your call back functions**

# glutMainLoop()

```
#include <GL/glut.h>
#include <GL/gl.h>

int main(int argc, char** argv)
{
    int mode = GLUT_RGB | GLUT_DOUBLE;
    glutInitDisplayMode(mode);
    glutInitWindowSize(500,500);
    glutCreateWindow("Simple");
    init();
    glutDisplayFunc(mydisplay);
    glutKeyboardFunc(key);
    glutMainLoop();
}
```

**The program goes into an infinite loop waiting for events**

## init.c

```
void init()  
{
```

```
    glClearColor (0.0, 0.0, 0.0, 1.0);
```

```
    glColor3f(1.0, 1.0, 1.0);
```

```
    glMatrixMode (GL_PROJECTION);
```

```
    glLoadIdentity ();
```

```
    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```

```
}
```

black clear color

opaque window

fill/draw with white

viewing volume

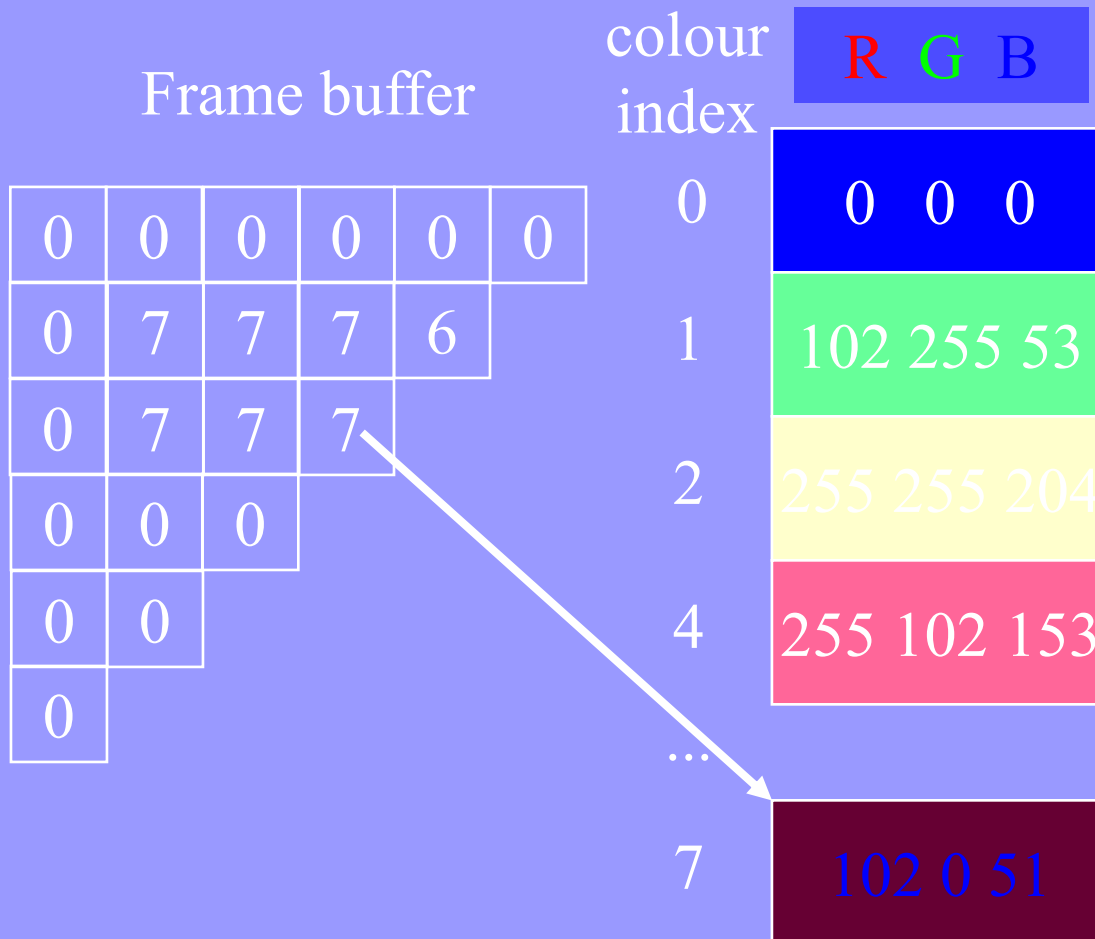
# Mydisplay()

```
void mydisplay() {  
    glClear(GL_COLOR_BUFFER_BIT);  
    glBegin(GL_POLYGON);  
        glVertex2f(-0.5, -0.5);  
        glVertex2f(-0.5, 0.5);  
        glVertex2f(0.5, 0.5);  
        glVertex2f(0.5, -0.5);  
    glEnd();  
    glFlush();  
}
```

# Color Attributes

- **glColor3f(GLfloat r, GLfloat g, GLfloat b)** sets the drawing color
  - **glColor3d()** , **glColor3ui()** can also be used
  - Remember OpenGL is a state machine
  - Once set, the attribute applies to all subsequent defined objects until it is set to some other value
  - **glColor3fv()** takes a flat array as input

# Color Lookup Table



CLUT:  
pixel = code

True colour:  
pixel = R,G,B

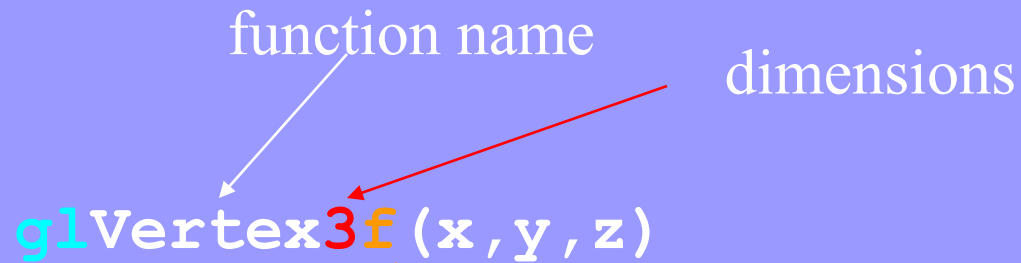
# Famous Color

- We will use **glColor3f()** or **glColor3fv()** with floating-point values specifying the colour. some examples are:
- **glColor3f(1.0, 0.0, 0.0)** red
- **glColor3f(0.0, 1.0, 0.0)** green
- **glColor3f(0.0, 0.0, 1.0)** blue
- **glColor3f(1.0, 1.0, 1.0)** white
- **glColor3f(0.0, 0.0, 0.0)** black
- **glColor3f(0.0, 1.0, 1.0)** cyan
- **glColor3f(1.0, 1.0, 0.0)** yellow
- **glColor3f(0.3, 0.25, 0.8)** a shade of blue

# OpenGL vertex format

function name      dimensions

`glVertex3f(x, y, z)`



belongs to GL library

`x, y, z` are floats

`glVertex3fv(p)`

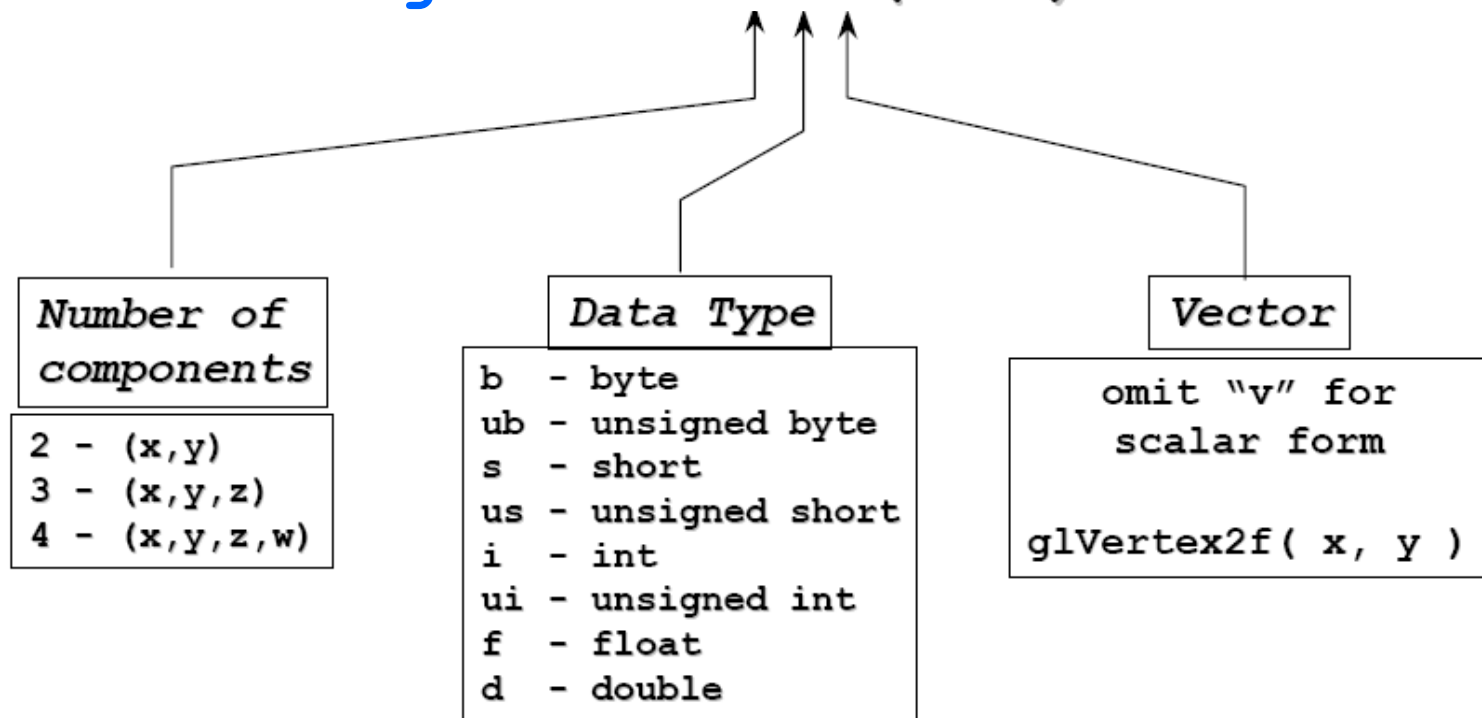


`p` is a pointer to an array



# OpenGL vertex format

`glVertex3fv( v )`



# Pixels Attributes

- `glSetPixel(?)`
- `glGetPixel(?)` *for scan line number – y, column number – x*
- Position (`glVertex 2f(x,y);`)
- Size (`glPointSize(5);`)
- Color (`glColor3f(1,0,0)`)
-

# Individual Point and multiple points

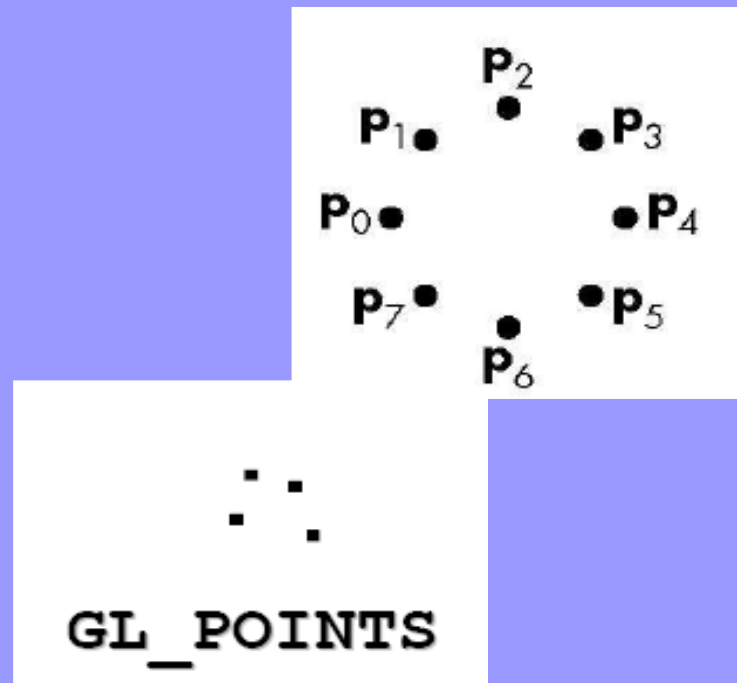
- In OpenGL, to specify a point:
  - `glVertex*()`;
- In OpenGL, some functions require both a dimensionality and a data type
  - `glVertex2i(80,100), glVertex2f(58.9, 90.3)`
  - `glVertex3i(20,20,-5), glVertex3f(-2.2,20.9,20)`
- Must put within a ‘glBegin/glEnd’ pair
  - `glBegin(GL_POINTS);`
  - `glVertex2i(50,50);`
  - `glVertex2i(60,60);`
  - `glVertex2i(60,50);`
  - `glEnd();`
- Let's draw points in our assignment #1
- Next up? Lines

# Individual Point and multiple points

- Points, **GL\_POINTS**

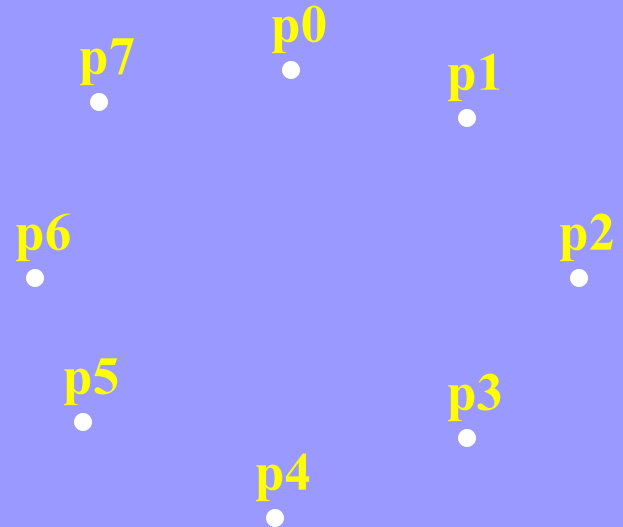
- Individual points
- Point size can be altered
  - *glPointSize (float size)*

```
glBegin (GL_POINTS);  
glColor3fv( color );  
glVertex2f( P0.x, P0.y );  
glVertex2f( P1.x, P1.y );  
glVertex2f( P2.x, P2.y );  
glVertex2f( P3.x, P3.y );  
glVertex2f( P4.x, P4.y );  
glVertex2f( P5.x, P5.y );  
glVertex2f( P6.x, P6.y );  
glVertex2f( P7.x, P7.y );  
glEnd();
```



# Points in OpenGL

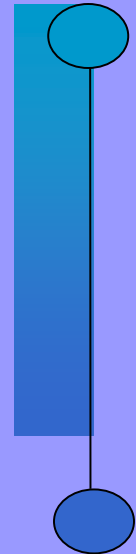
```
glBegin(GL_POINTS) ;  
    glVertex2fv(p0) ;  
    glVertex2fv(p1) ;  
    glVertex2fv(p2) ;  
    glVertex2fv(p3) ;  
    glVertex2fv(p4) ;  
    glVertex2fv(p5) ;  
    glVertex2fv(p6) ;  
    glVertex2fv(p7) ;  
glEnd() ;
```



# OpenGL Line Format

## Single Line

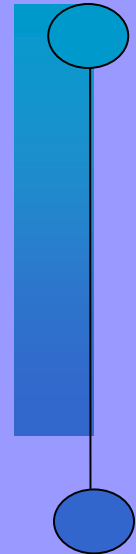
```
DrawLine();  
{  
    glBegin(GL_LINES)  
    glColor3f (0,0,0);  
    glVertex2f (0,5);  
    glVertex2f (0,0);  
    glEnd()  
}
```



# OpenGL Line Format

## Line as Vector

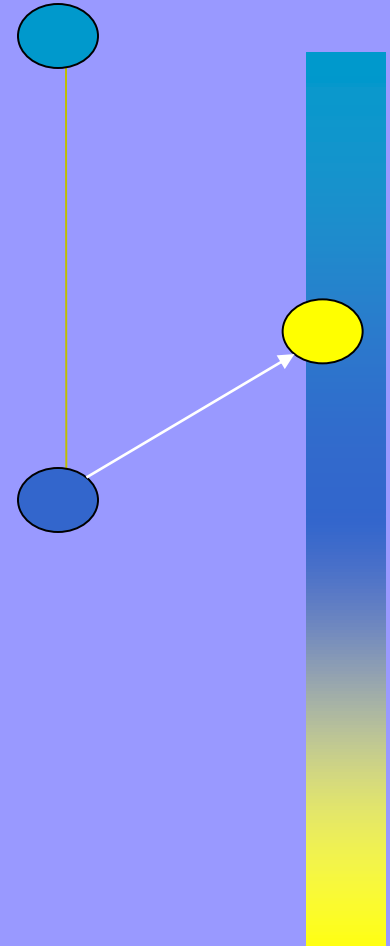
```
DrawLine(ps, pe , cs)  
{  
    glBegin(GL_LINES)  
    glColor3fv(cs);  
    glVertex3fv(ps);  
    glColor3fv(ce);  
    glEnd()  
}
```



# OpenGL Line Format

## Two Lines

```
DrawLine()  
{  
    glBegin(GL_LINES)  
    glVertex2f (0,0);  
    glVertex2f(0,-5);  
    glVertex2f(0,-5);  
    glVertex2f (5,5);  
    glColor3f(0,0,0);  
    glEnd()  
}
```





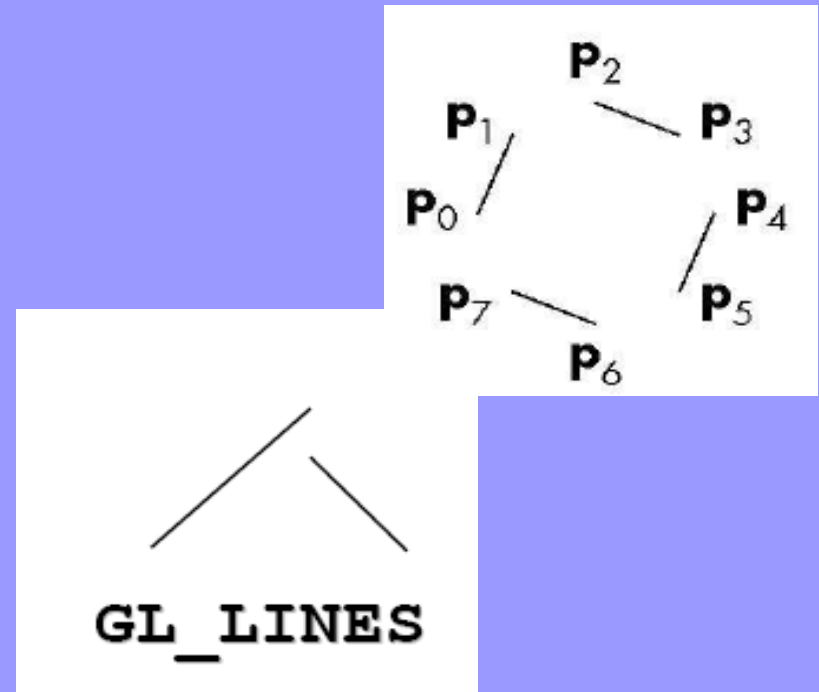
# OpenGL Line Format

## Multiple Lines

- Lines, **GL\_LINES**

- Pairs of vertices interpreted as individual line segments
- Can specify line width using:
  - *glLineWidth (float width)*

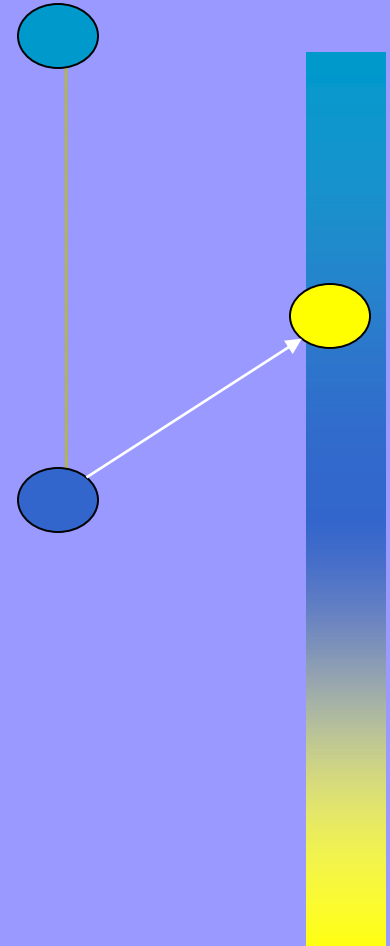
```
glBegin( GL_LINES );  
glColor3fv( color );  
glVertex2f( P0.x, P0.y );  
glVertex2f( P1.x, P1.y );  
glVertex2f( P2.x, P2.y );  
glVertex2f( P3.x, P3.y );  
glVertex2f( P4.x, P4.y );  
glVertex2f( P5.x, P5.y );  
glVertex2f( P6.x, P6.y );  
glVertex2f( P7.x, P7.y );  
glEnd();
```



# OpenGL Line Format

## GL line Strip

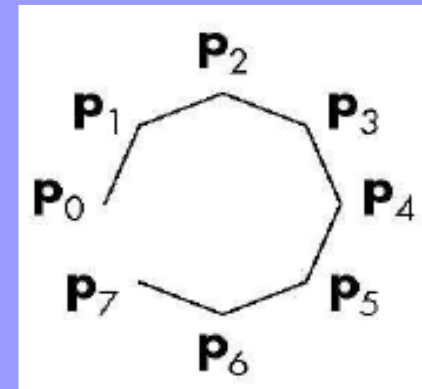
```
DrawLine()  
{  
    glBegin(GL_LINE_STRIP)  
    glVertex2f (0,0);  
    glVertex2f(0,-5);  
    glVertex2f (5,5);  
    glColor3f(0,0,0);  
    glEnd()  
}
```



# OpenGL Line Format

## GL line Strip

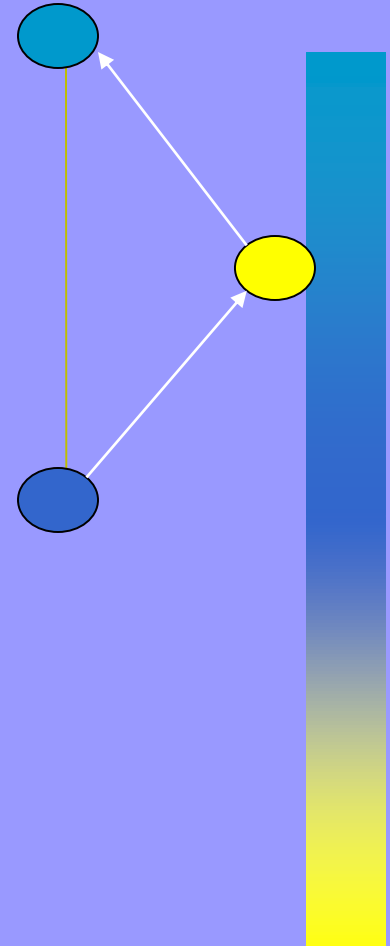
- Line Strip, **GL\_LINE\_STRIP**
  - series of connected line segments



# OpenGL Line Format

## GL lines Loop

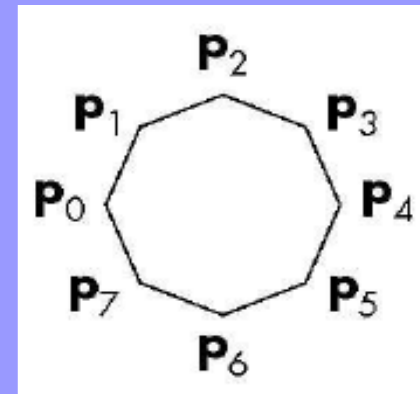
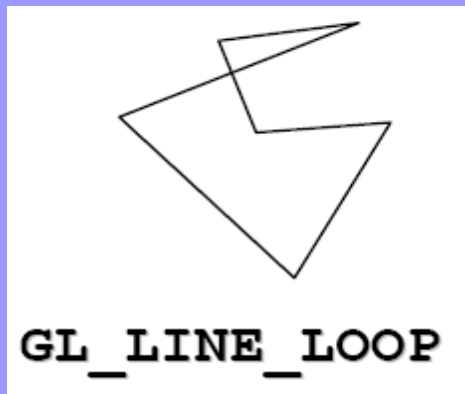
```
DrawLine()  
{  
    glBegin(GL_LINE_LOOP)  
    glVertex2f (0,0);  
    glVertex2f(0,-5);  
    glVertex2f (5,5);  
    glColor3f(0,0,0);  
    glEnd()  
}
```



# OpenGL Line Format

## GL lines Loop

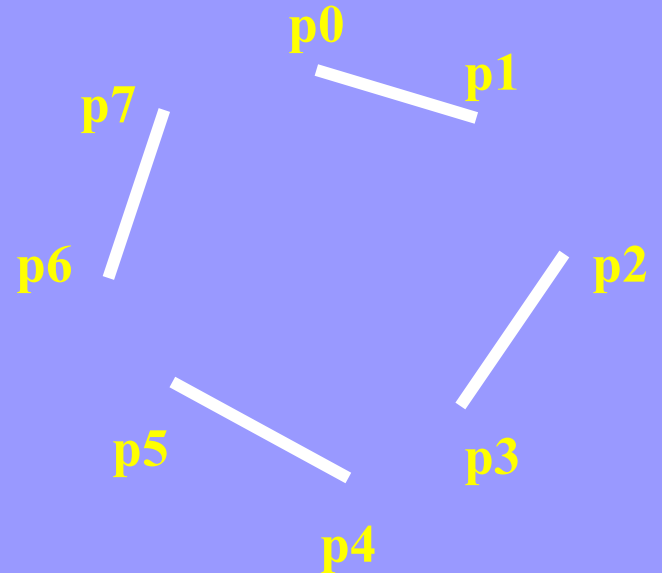
- Line Loop, **GL\_LINE\_LOOP**
  - Line strip with a segment added between last and first vertices



# OpenGL Line Format

## Showing Difference

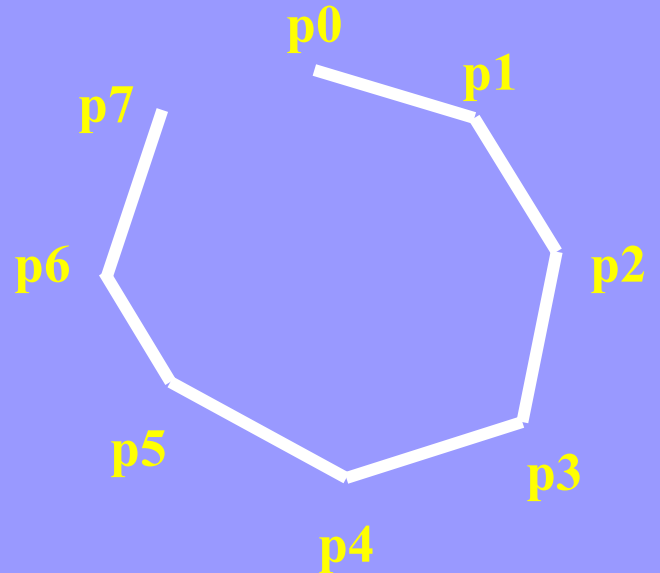
```
glBegin(GL_LINES) ;  
    glVertex2fv(p0) ;  
    glVertex2fv(p1) ;  
    glVertex2fv(p2) ;  
    glVertex2fv(p3) ;  
    glVertex2fv(p4) ;  
    glVertex2fv(p5) ;  
    glVertex2fv(p6) ;  
    glVertex2fv(p7) ;  
glEnd() ;
```



# OpenGL Line Format

## Showing Difference

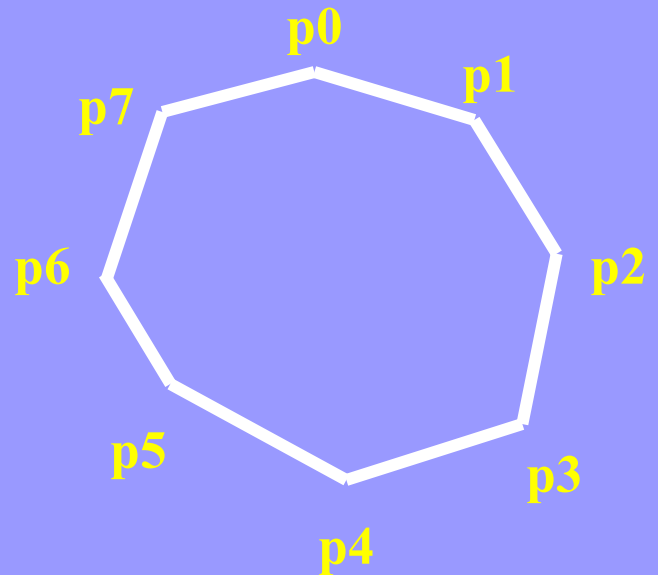
```
glBegin(GL_LINE_STRIP);  
    glVertex2fv(p0);  
    glVertex2fv(p1);  
    glVertex2fv(p2);  
    glVertex2fv(p3);  
    glVertex2fv(p4);  
    glVertex2fv(p5);  
    glVertex2fv(p6);  
    glVertex2fv(p7);  
glEnd();
```



# OpenGL Line Format

## Showing Difference

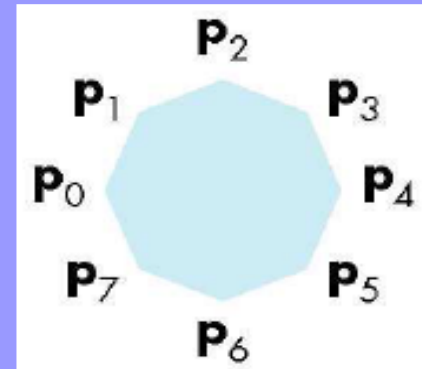
```
glBegin(GL_LINE_LOOP);  
    glVertex2fv(p0);  
    glVertex2fv(p1);  
    glVertex2fv(p2);  
    glVertex2fv(p3);  
    glVertex2fv(p4);  
    glVertex2fv(p5);  
    glVertex2fv(p6);  
    glVertex2fv(p7);  
glEnd();
```





# Vertices and Primitives

- Polygon , **GL\_POLYGON**
  - boundary of a simple, convex polygon



# OpenGL: Drawing Triangles

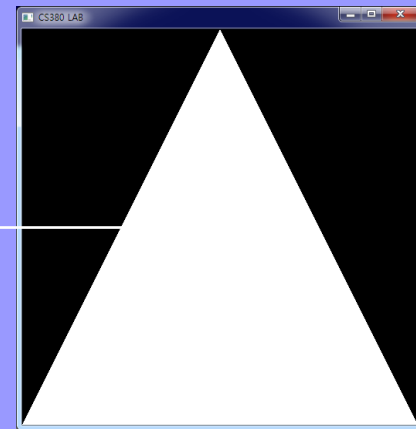
- You can draw multiple triangles between `glBegin(GL_TRIANGLES)` and `glEnd()`:
  - `float v1[3], v2[3], v3[3], v4[3];`
  - ...
  - `glBegin(GL_TRIANGLES);`
  - `glVertex3fv(v1); glVertex3fv(v2); glVertex3fv(v3);`
  - `glVertex3fv(v1); glVertex3fv(v3); glVertex3fv(v4);`
  - `glEnd();`
- The same vertex is used (sent, transformed, colored) many times (6 on average)

# Draw Polygon

## OpenGL Triangles

- `void display() {`
- `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);`
- `glLoadIdentity();` // Reset our view
- `glBegin(GL_TRIANGLES);` // Draw a triangle
- `glVertex3f( 0.0f, 1.0f, 0.0f);`
- `glVertex3f(-1.0f,-1.0f, 0.0f);`
- `glVertex3f( 1.0f,-1.0f, 0.0f);`
- `glEnd();`
- `glFlush();`
- `}`

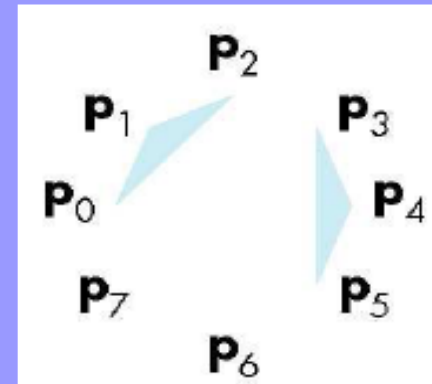
(0,0,0)



# Draw Polygon

## OpenGL Triangles

- Triangles , **GL\_TRIANGLES**
  - triples of vertices interpreted as triangles



# Draw Polygon

## OpenGL Triangles Strips

- An OpenGL *triangle strip* primitive reduces this redundancy by sharing vertices:

```
glBegin(GL_TRIANGLE_STRIP);
```

```
    glVertex3fv(v0);
```

```
    glVertex3fv(v1);
```

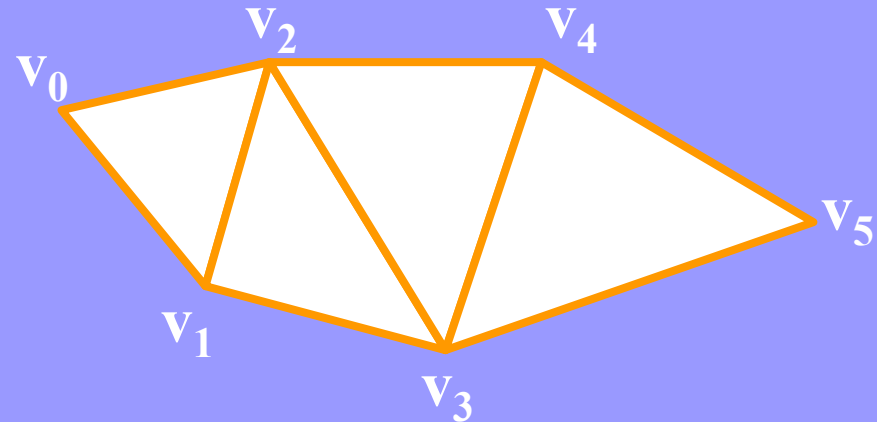
```
    glVertex3fv(v2);
```

```
    glVertex3fv(v3);
```

```
    glVertex3fv(v4);
```

```
    glVertex3fv(v5);
```

```
glEnd();
```



triangle 0 is v0, v1, v2

triangle 1 is v2, v1, v3 (*why not v1, v2, v3?*)

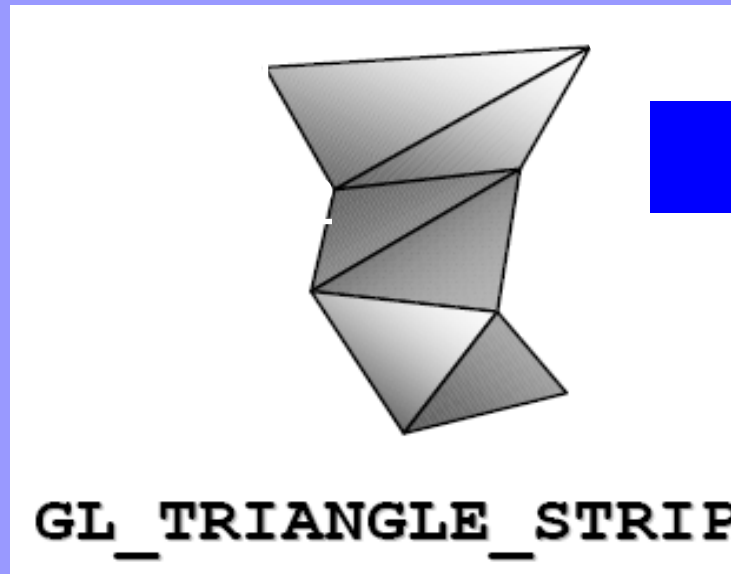
triangle 2 is v2, v3, v4

triangle 3 is v4, v3, v5 (again, **not** v3, v4, v5)

# Draw Polygon

## OpenGL Triangles Strip

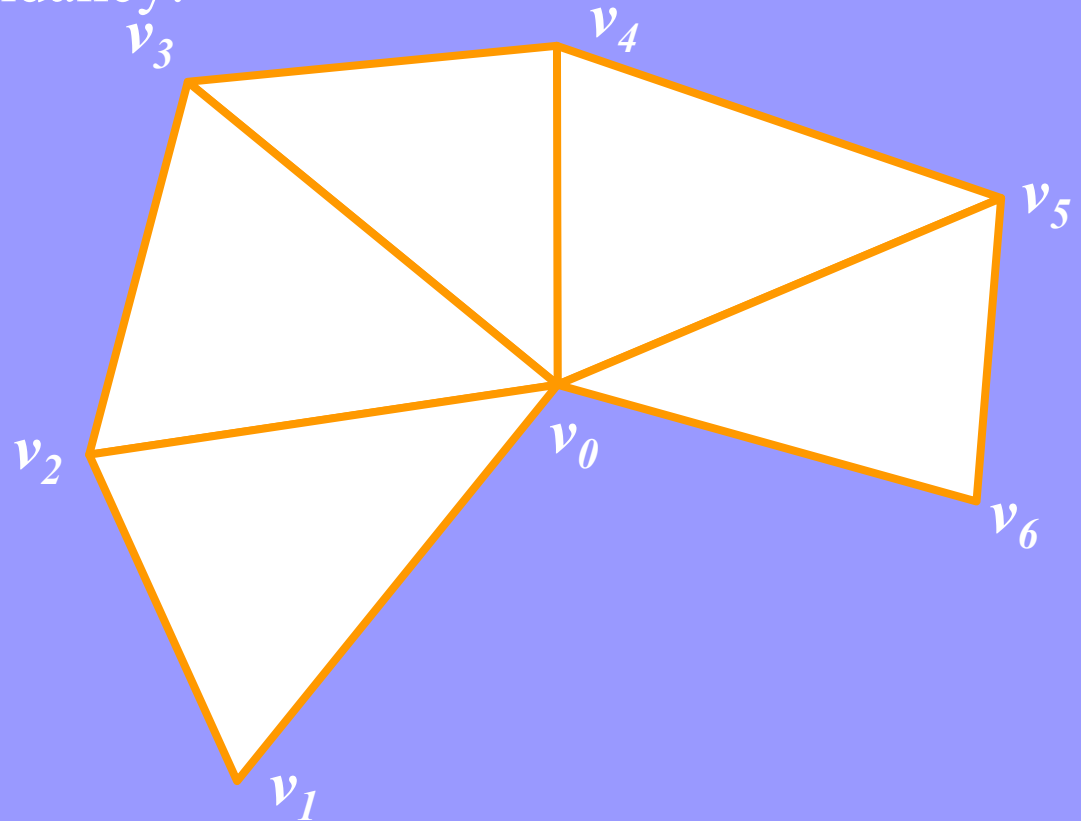
- Triangle Strip , **GL\_TRIANGLE\_STRIP**
  - linked strip of triangles



# Draw Polygon

## OpenGL Triangles Fan

- The **GL\_TRIANGLE\_FAN** primitive is another way to reduce vertex redundancy:



# Draw Polygon

## OpenGL Triangles Fan

- Triangle Fan ,  
**GL\_TRIANGLE\_FAN**
  - linked fan of triangles

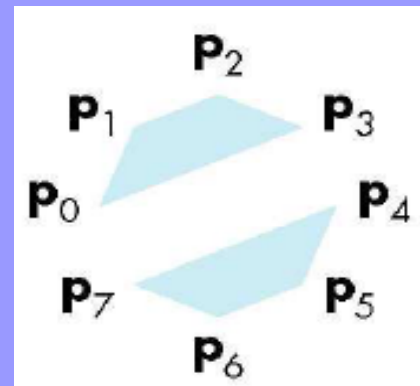




# Draw Polygon

## OpenGL Quads

- Quads , **GL\_QUADS**
  - quadruples of vertices interpreted as four-sided polygons



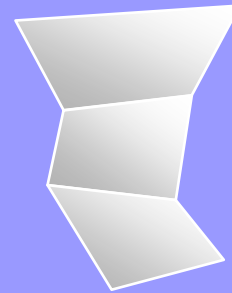
# Draw Polygon

## OpenGL Quads

- Quadrilaterals (quads)
  - Individual quads
    - Type is `GL_QUADS`
    - A quad is defined by 4 vertices
    - Quads can be decomposed into two triangles
    - Quads are not necessarily plane or convex
      - Be careful with the vertex sequence
  - Strips of connected quads
    - Type is `GL_QUAD_STRIP`
    - Uses the most recent 2 vertices and 2 new vertices



**GL\_QUADS**

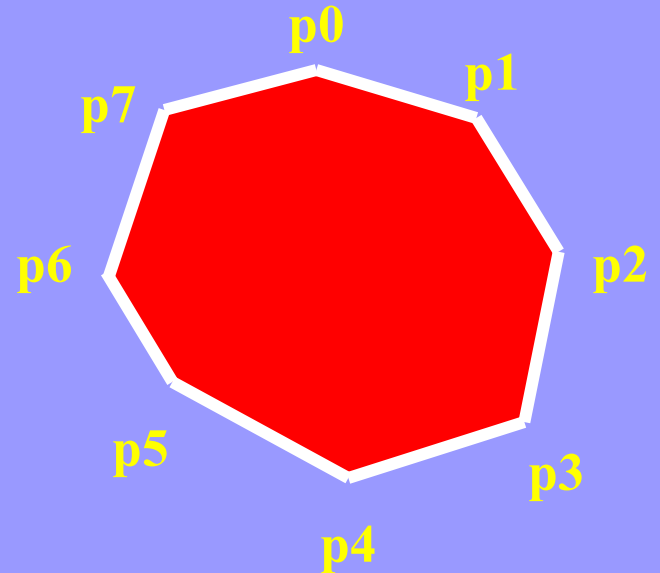


**GL\_QUAD\_STRIP**

# Draw Polygon

## Showing Difference

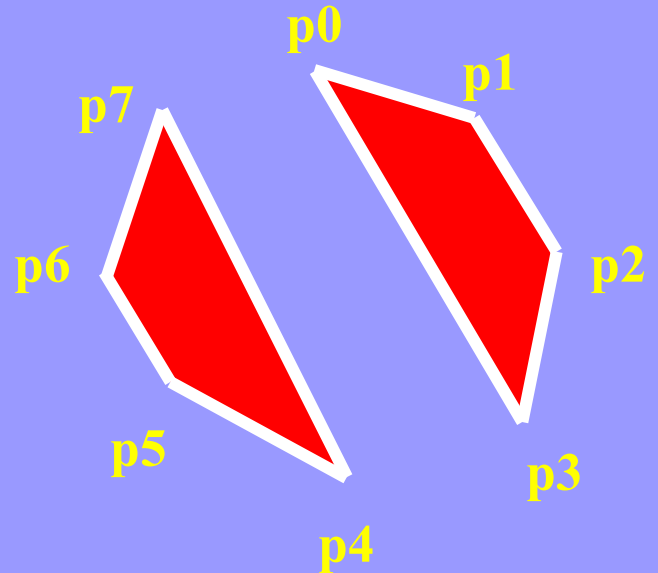
```
glBegin(GL_POLYGON) ;  
    glVertex2fv(p0) ;  
    glVertex2fv(p1) ;  
    glVertex2fv(p2) ;  
    glVertex2fv(p3) ;  
    glVertex2fv(p4) ;  
    glVertex2fv(p5) ;  
    glVertex2fv(p6) ;  
    glVertex2fv(p7) ;  
glEnd() ;
```



# Draw Polygon

## Showing Difference

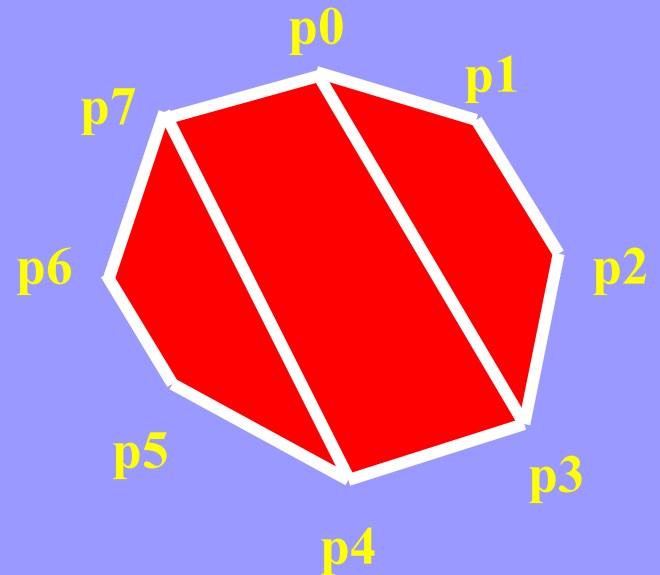
```
glBegin(GL_QUADS) ;  
    glVertex2fv(p0) ;  
    glVertex2fv(p1) ;  
    glVertex2fv(p2) ;  
    glVertex2fv(p3) ;  
    glVertex2fv(p4) ;  
    glVertex2fv(p5) ;  
    glVertex2fv(p6) ;  
    glVertex2fv(p7) ;  
glEnd() ;
```



# Draw Polygon

## Showing Difference

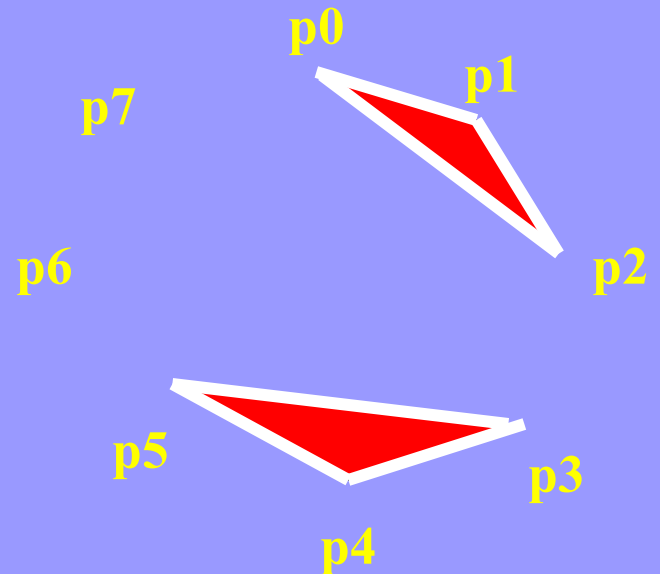
```
glBegin(GL_QUAD_STRIP);  
    glVertex2fv(p1);  
    glVertex2fv(p2);  
    glVertex2fv(p3);  
    glVertex2fv(p0);  
    glVertex2fv(p4);  
    glVertex2fv(p7);  
    glVertex2fv(p5);  
    glVertex2fv(p6);  
glEnd();
```



# Draw Polygon

## Showing Difference

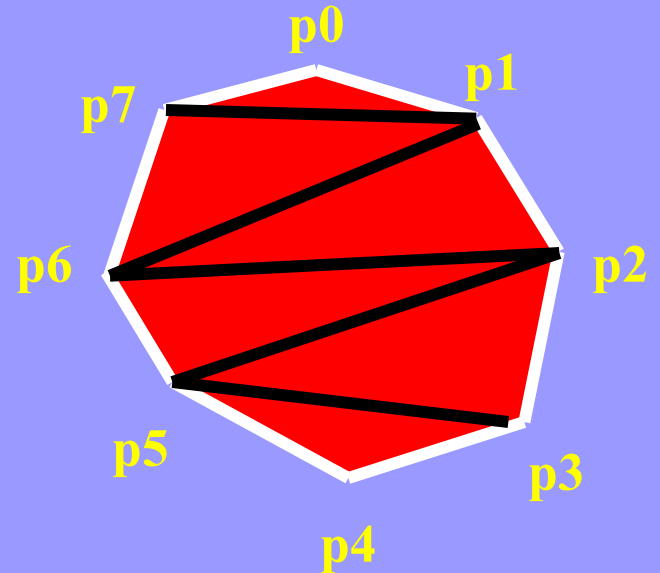
```
glBegin(GL_TRIANGLES) ;  
    glVertex2fv(p0) ;  
    glVertex2fv(p1) ;  
    glVertex2fv(p2) ;  
    glVertex2fv(p3) ;  
    glVertex2fv(p4) ;  
    glVertex2fv(p5) ;  
    glVertex2fv(p6) ;  
    glVertex2fv(p7) ;  
glEnd() ;
```



# Draw Polygon

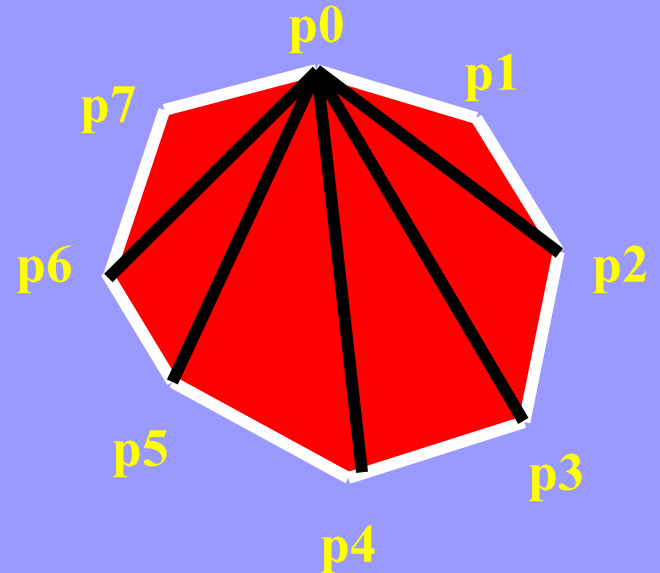
## Showing Difference

```
glBegin(GL_TRIANGLE_STRIP);  
    glVertex2fv(p0);  
    glVertex2fv(p7);  
    glVertex2fv(p1);  
    glVertex2fv(p6);  
    glVertex2fv(p2);  
    glVertex2fv(p5);  
    glVertex2fv(p3);  
    glVertex2fv(p4);  
glEnd();
```



# Draw Polygon Showing Difference

```
glBegin(GL_TRIANGLE_FAN);  
    glVertex2fv(p0);  
    glVertex2fv(p1);  
    glVertex2fv(p2);  
    glVertex2fv(p3);  
    glVertex2fv(p4);  
    glVertex2fv(p5);  
    glVertex2fv(p6);  
    glVertex2fv(p7);  
glEnd();
```

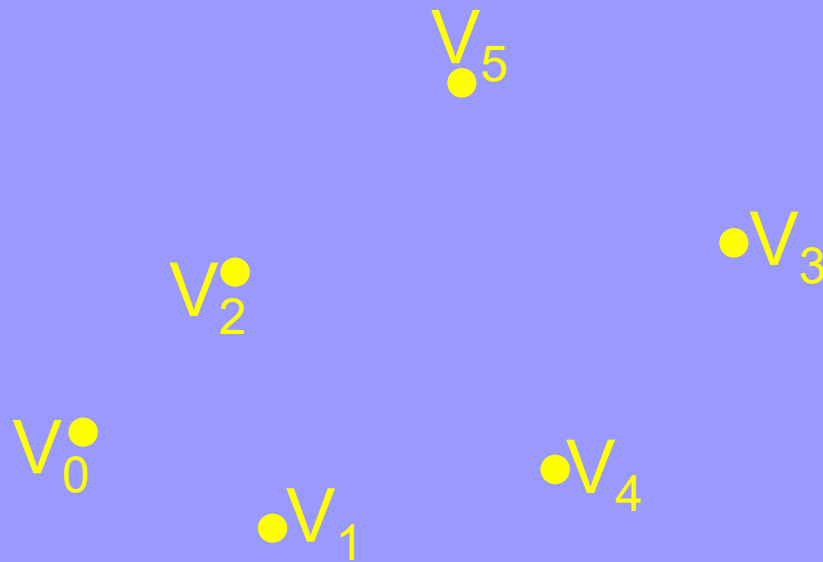




# Draw Primitives

## Showing Difference

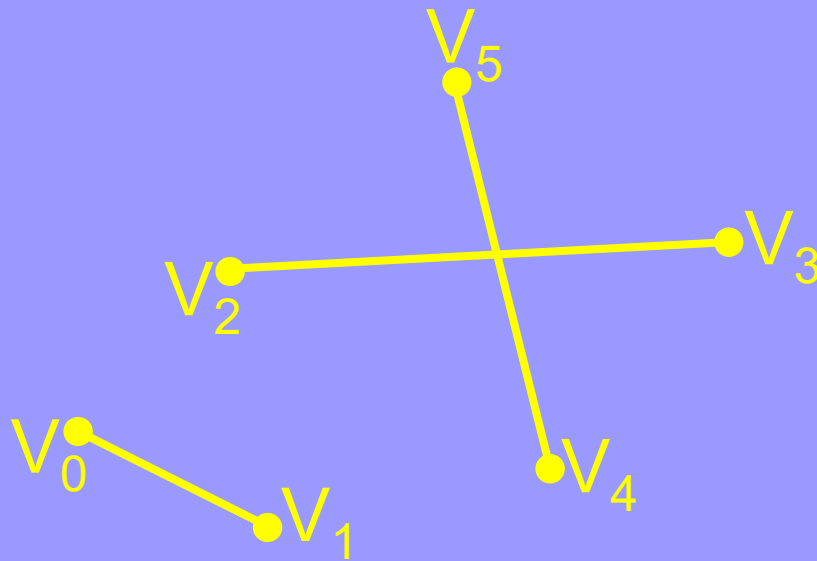
- Points



# Draw Primitives

## Showing Difference

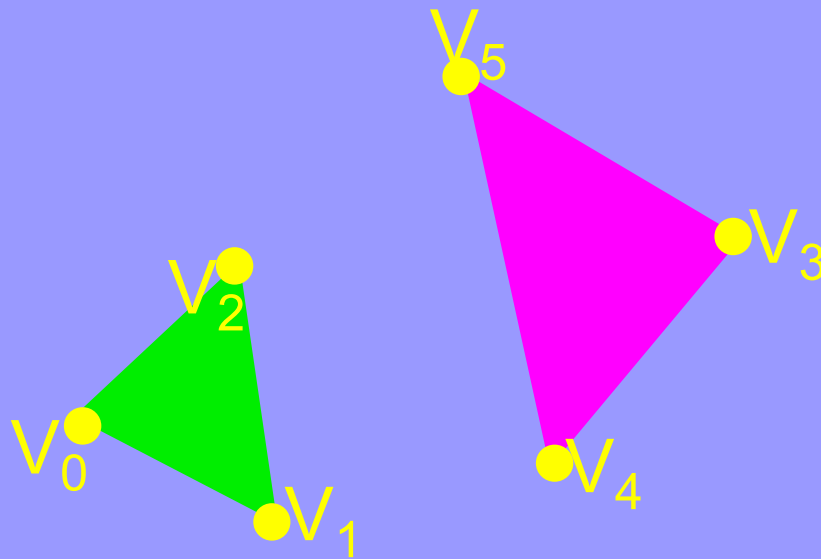
- Points
- Lines



# Draw Primitives

## Showing Difference

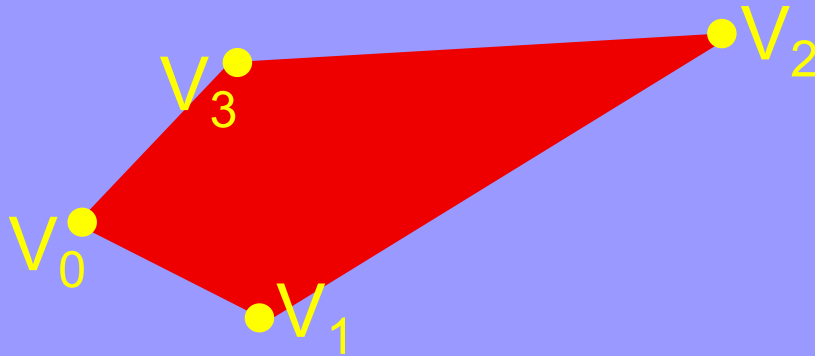
- Points
- Lines
- Triangles



# Draw Primitives

## Showing Difference

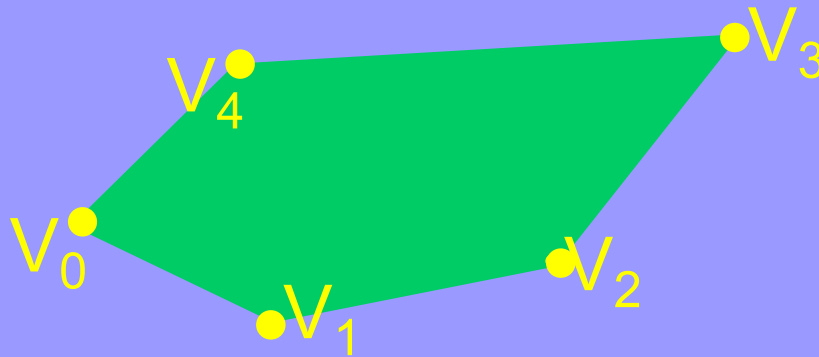
- Points
- Lines
- Triangles
- Quads



# Draw Primitives

## Showing Difference

- Points
- Lines
- Triangles
- Quads
- Polygons.



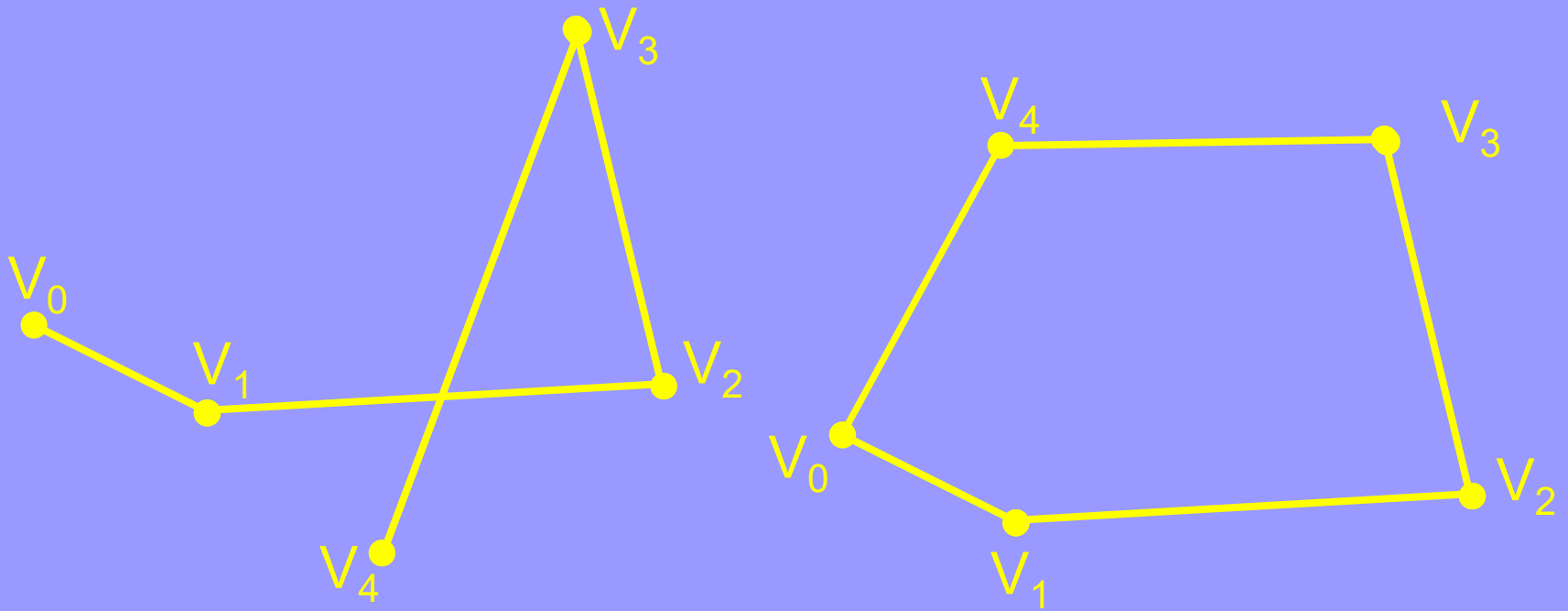
# Draw Primitives

## Showing Difference

# Draw Primitives

## Showing Difference

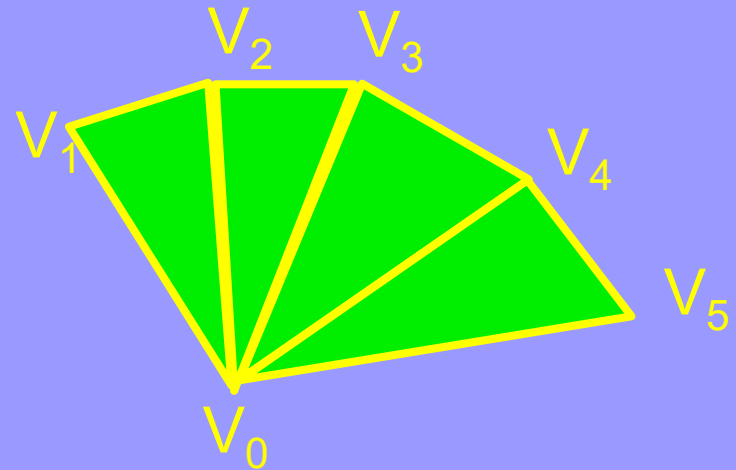
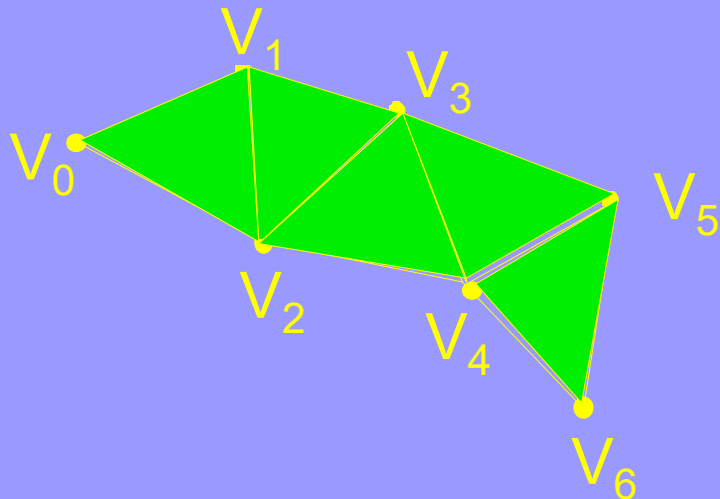
- Line strips and line loops.



# Draw Primitives

## Showing Difference

- Line strips and line loops.
- Triangle strips and fans.

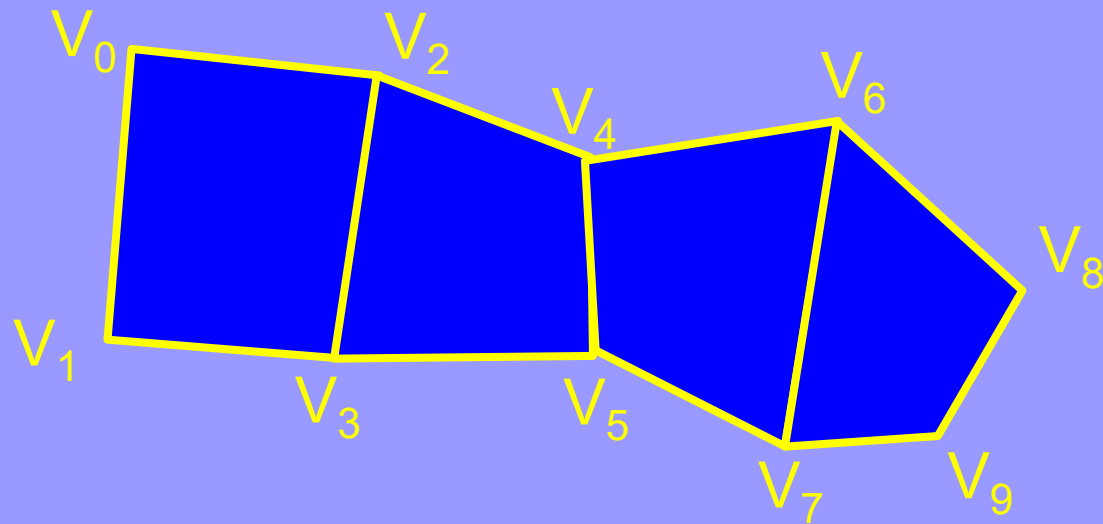




# Draw Primitives

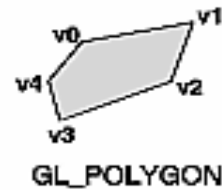
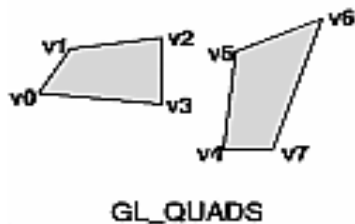
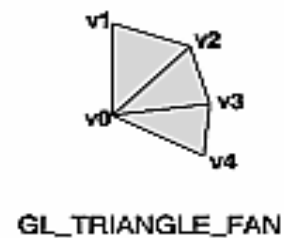
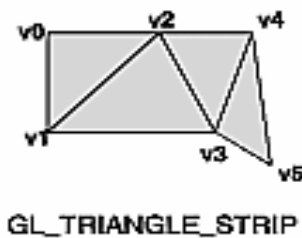
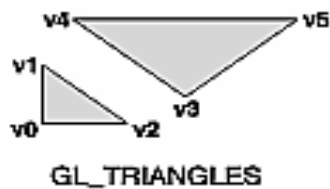
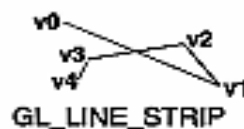
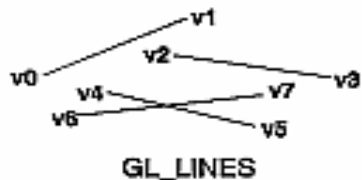
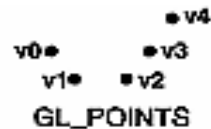
## Showing Difference

- Line strips and line loops.
- Triangle strips and fans.
- Quad strips.



# Exercise

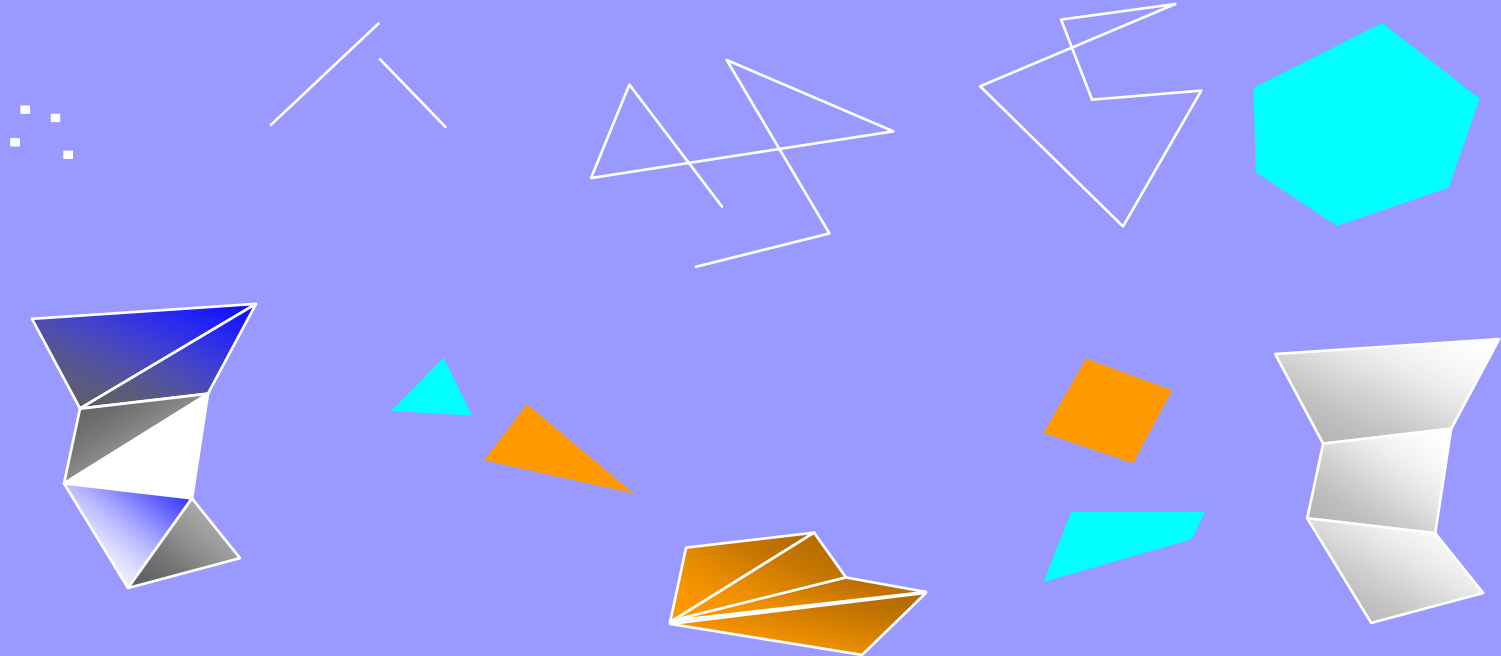
Draw the following:



# Exercise

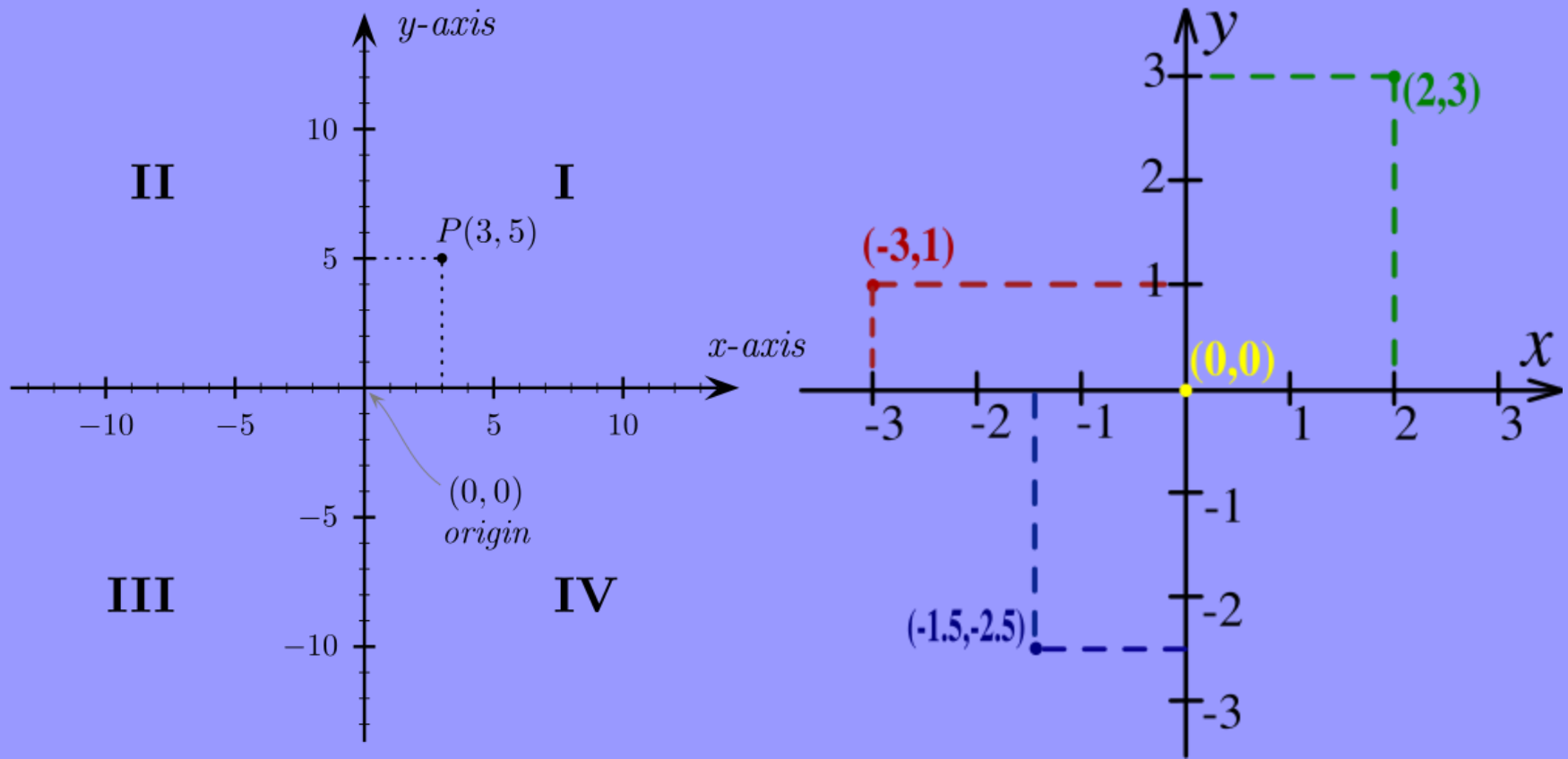


- Draw the following:



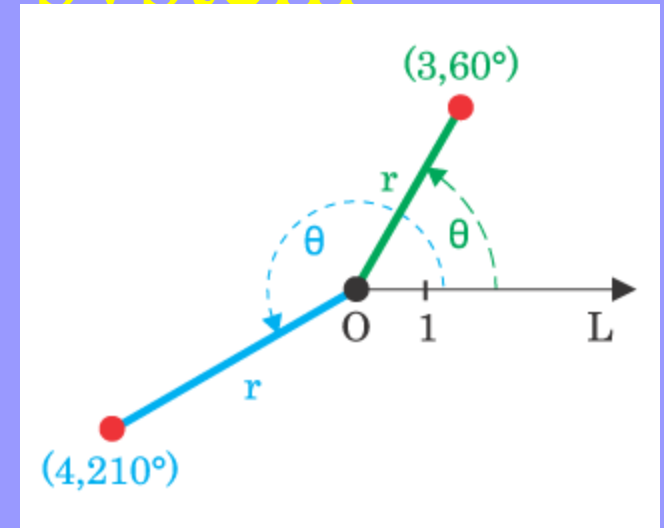
# 2D Coordinate System

- A way to associate points in a plane with numbers
- Each point can be represented as two real numbers, usually called x-coordinate and y-coordinate



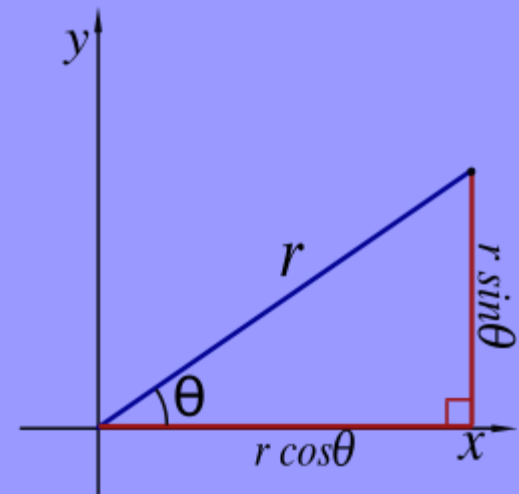
# Polar coordinate system

- Polar coordinate system



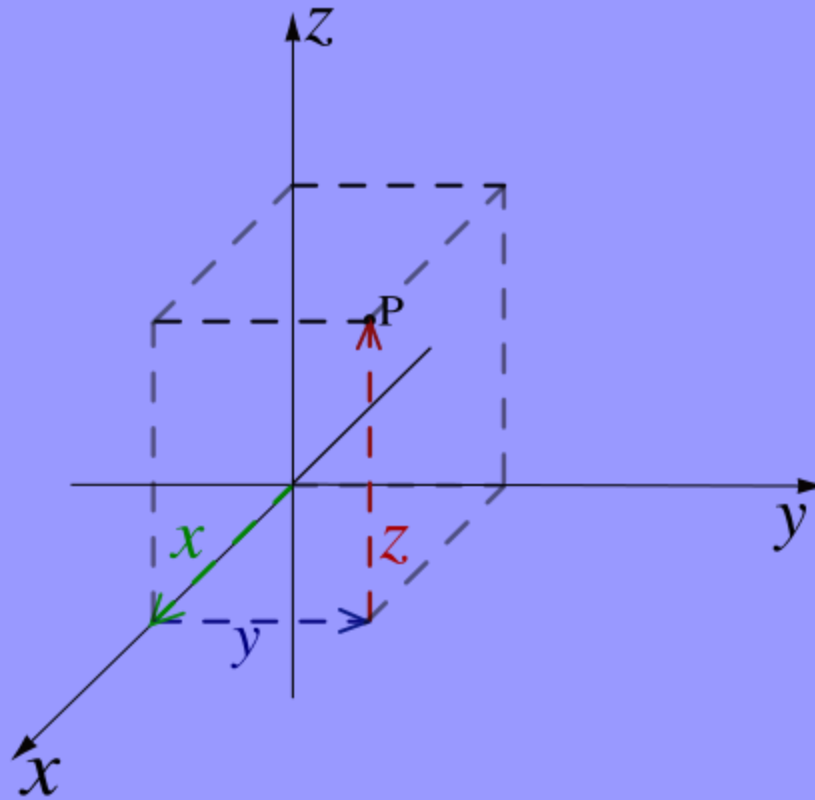
- Polar to Cartesian

$$\begin{aligned}x &= r \cos \theta \\y &= r \sin \theta,\end{aligned}$$



# 3D coordinate system

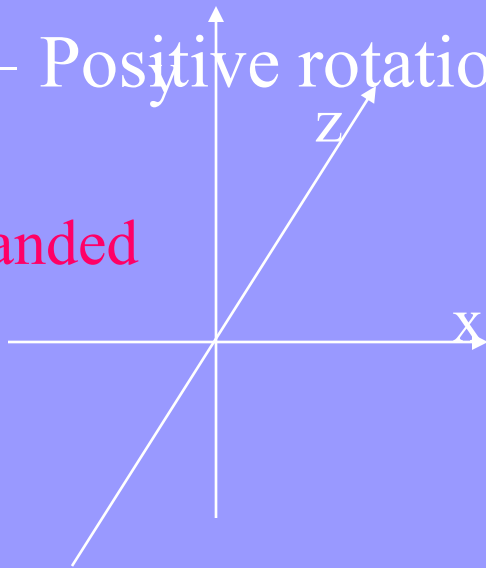
- Map points in our world to 3 real numbers.



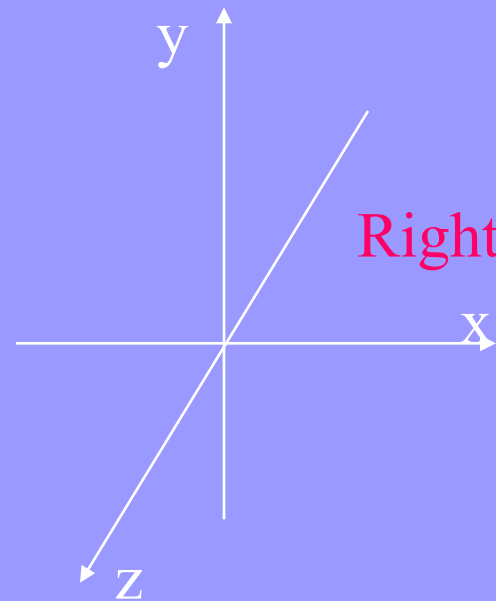
# Left-handed and right-handed Coordinate systems

- OpenGL is right-handed
  - Positive rotations: counter clockwise
- DirectX is left-handed
  - Positive rotations: clockwise

Left-handed



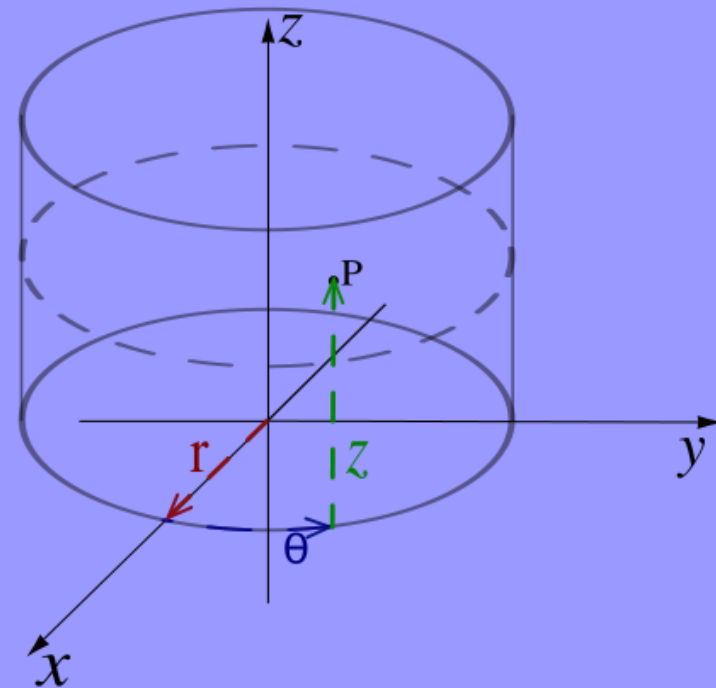
Right-handed



# Cylindrical coordinate system

- Radial projection, angle, and height

$$\begin{aligned}x &= r \cos \theta \\y &= r \sin \theta,\end{aligned}$$



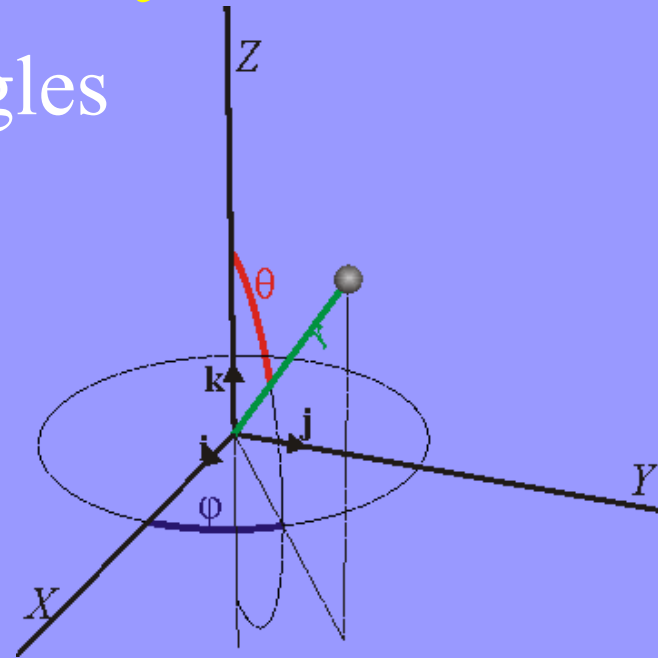
- From cylindrical to Cartesian (z is the same)



# Spherical Coordinate System

- Radial distance and two angles

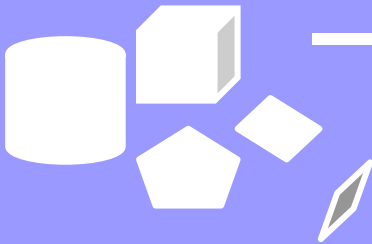
$$\begin{aligned}x &= r \sin \varphi \cos \theta \\y &= r \sin \varphi \sin \theta \\z &= r \cos \varphi.\end{aligned}$$



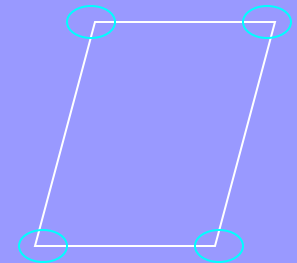
- From Spherical coordinate system to Cartesian:

# 3D scene

- Scene = list of objects
- Object = list of surfaces
- surface = list of polygons
- Polygon = list of vertices
- Vertex = a point in 3D



**scene**



**vertices**

# Line Drawing Algorithms

- Line drawn as pixels
- Graphics system
  - Projects the endpoints to their pixel locations in the frame buffer (screen coordinates as integers)
  - Finds a path of pixels between the two
  - Loads the color
  - Plots the line on the monitor from frame buffer (video controller)
  - Rounding causes all lines except horizontal or vertical to be displayed as jigsaw appearance (low resolution)
  - Improvement: high resolution, or adjust pixel intensities on the line path.

# Line Drawing Algorithms

- Line equation

- Slope-intercept form

$$y = m \cdot x + b$$

slope  $m$

Y-intercept  $b$

- Slope

$$m = \frac{y_{\text{end}} - y_0}{x_{\text{end}} - x_0} = \frac{\delta_y}{\delta_x}$$

- Y-intercept

$$b = y_0 - mx_0$$

# Line Drawing Algorithms

- DDA (Digital Differential Analyzer)
  - Scan conversion line algorithm
  - Line sampled at regular intervals of x, then corresponding y is calculated
  - From left to right

$$\text{if } m \leq 1, \quad y_{k+1} = y_k + m \quad (\delta_x = 1)$$

$$\text{if } m > 1.0, \quad x_{k+1} = x_k + \frac{1}{m} \quad (\delta_y = 1)$$

=

# Line Drawing Algorithms

```
void lineDDA (int x0, int y0, int xEnd, int yEnd)
{
    int dx = xEnd - x0, dy = yEnd - y0, steps, k;
    float xIncrement, yIncrement, x = x0, y = y0;
    if (fabs (dx) > fabs (dy))
        steps = fabs (dx);
    else
        steps = fabs (dy);
    xIncrement = float (dx) / float (steps);
    yIncrement = float (dy) / float (steps);
    setPixel (round (x), round (y));
    for (k = 0; k < steps; k++) {
        x += xIncrement;
        y += yIncrement;
        setPixel (round (x), round (y));
    }
}
```

# Line Drawing Algorithms

- Advantage
  - Does not calculate coordinates based on the complete equation (uses offset method)
- Disadvantage
  - Round-off errors are accumulated, thus line diverges more and more from straight line
  - Round-off operations take time
    - Perform integer arithmetic by storing float as integers in numerator and denominator and performing integer arithmetic.

# Line Drawing Algorithms

- Bresenham's line drawing
  - Efficient line drawing algorithm using only incremental integer calculations
  - Can be adapted to draw circles and other curves
- Principle
  - Vertical axes show scan line positions
  - Horizontal axes show pixel columns
  - At each step, determine the best next pixel based on the sign of an integer parameter whose value is proportional to the difference between the vertical separations of the two pixel positions from the actual line.



# Line Drawing Algorithms

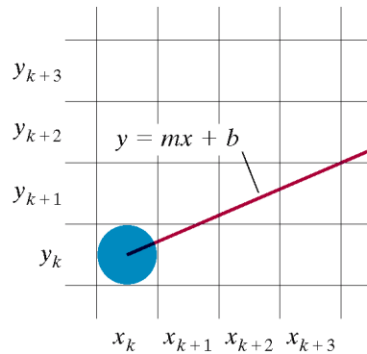


Figure 3-10

A section of the screen showing a pixel in column  $x_k$  on scan line  $y_k$  that is to be plotted along the path of a line segment with slope  $0 < m < 1$ .

(from Donald Hearn and Pauline Baker)

# Line Drawing Algorithms

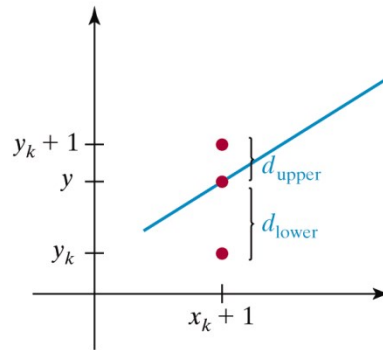


Figure 3-11

Vertical distances between pixel positions and the line  $y$  coordinate at sampling position  $x_k + 1$ .

(from Donald Hearn and Pauline Baker)

# Line Drawing Algorithms

- Bresenham's line drawing algorithm (positive slope less than 1)

$$d_{\text{lower}} = y - y_k$$

$$d_{\text{upper}} = (y_k + 1) - y$$

$$d_{\text{lower}} - d_{\text{upper}} = 2m(x_k + 1) - 2y_k + 2b - 1$$

decision parameter:  $p_k = \Delta x (d_{\text{lower}} - d_{\text{upper}})$

$$p_k = 2\Delta y x_k - 2\Delta x y_k + c$$

sign of  $p_k$  is the same as sign of  $d_{\text{lower}} - d_{\text{upper}}$

# Line Drawing Algorithms

- Bresenham's line drawing algorithm (positive slope less than 1)
  1. Input the two line endpoints and store the left endpoint in  $(x_0, y_0)$ .
  2. Set the color for the frame-buffer position  $(x_0, y_0)$  – i.e. plot the first point.
  3. Calculate the constant  $2\Delta y - \Delta x$ , and set the starting value for the decision parameter as  $p_0 = 2\Delta y - \Delta x$ .
  4. At each  $x_k$  along the line, from  $k=0$ , perform the following test:
    5. if  $p_k < 0$ , next point to plot is  $(x_k + 1, y_k)$  and  $p_{k+1} = p_k + 2\Delta y$
    - otherwise, next point to plot is  $(x_k + 1, y_k + 1)$  and  $p_{k+1} = p_k + 2\Delta y - 2\Delta x$
  5. Repeat step 4  $\Delta x$  times.

# Curve Functions

- Such primitive functions as to display circles and ellipses are not provided in OpenGL core library.
- GLU has primitives to display spheres, cylinders, rational B-splines, including simple Bezier curves.
- GLUT has some of these primitives too.
- Circles and ellipses can also be generated by approximating them using a polyline.
- Can also be generated by writing own circle or ellipsis display algorithm.

# Curve Functions

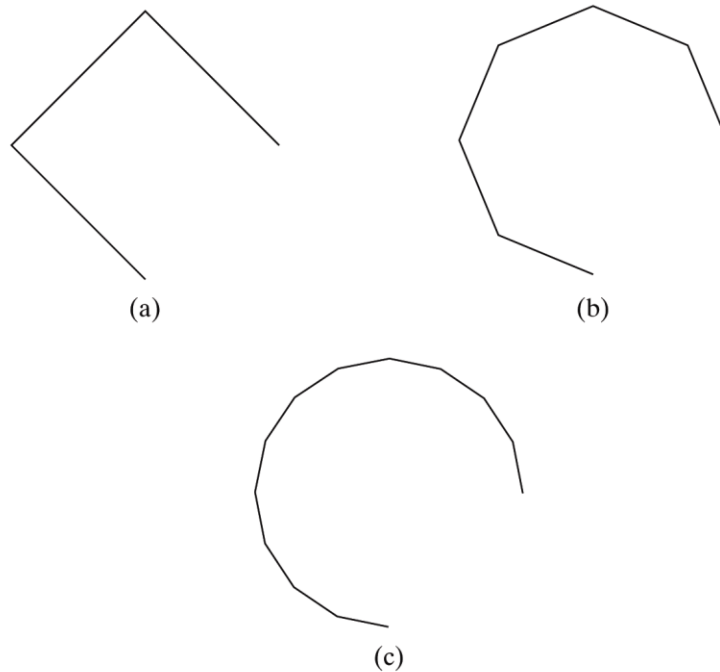


Figure 3-15

A circular arc approximated with (a) three straight-line segments, (b) six line segments, and (c) twelve line segments.

(from Donald Hearn and Pauline Baker)

# Circle Algorithms

- Properties of circles:

$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

steps from  $x_c - r$  to  $x_c + r$

$$y = y_c \pm \sqrt{r^2 - (x - x_c)^2}$$

# Circle Algorithms

- Polar coordinates
- Symmetry of circles

$$x = x_c + r \cos(\text{angle})$$
$$y = y_c + r \sin(\text{angle})$$

- All these methods have long execution time



# Curve Functions

## Circle

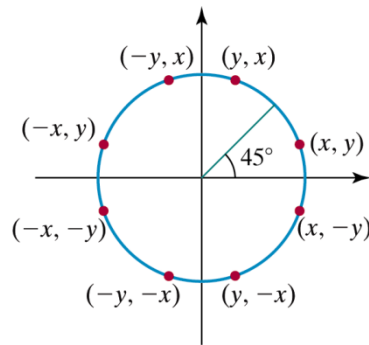


Figure 3-18

Symmetry of a circle. Calculation of a circle point  $(x, y)$  in one octant yields the circle points shown for the other seven octants.

(from Donald Hearn and Pauline Baker)

# Curve Functions

## Circle

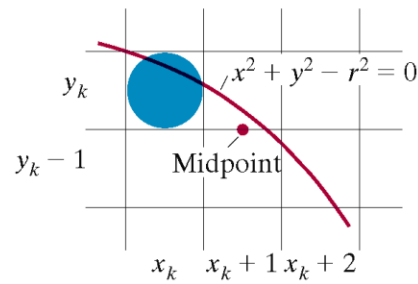


Figure 3-19

Midpoint between candidate pixels at sampling position  $x_k + 1$  along a circular path.

# Circle Algorithms

- Principle of the midpoint algorithm
  - Reason from octants of a circle centered in  $(0,0)$ , then find the remaining octants by symmetry, then translate to  $(x_c, y_c)$ .
  - The circle function is the decision parameter.
  - Calculate the circle function for the midpoint between two pixels.

# Circle Algorithms

- Midpoint circle drawing algorithm
  1. Input radius  $r$  and circle center  $(x_c, y_c)$ , then set the coordinates for the first point on the circumference of a circle centered on the origin as  $(x_0, y_0) = (0, r)$ .
  2. Calculate the initial value of the decision parameter as  $p_0 = 5/4 - r$  (1 -  $r$  if an integer)
  3. At each  $x_k$ , from  $k=0$ , perform the following test:  
if  $p_k < 0$ , next point to plot along the circle centered on  $(0,0)$  is  $(x_k + 1, y_k)$  and  $p_{k+1} = p_k + 2x_{k+1} + 1$   
otherwise, next point to plot is  $(x_k + 1, y_k - 1)$   
and  $p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$   
where  $2x_{k+1} = 2x_k + 2$ , and  $2y_{k+1} = 2y_k - 2$

# Circle Algorithms

- Midpoint circle drawing algorithm
  4. Determine symmetry points in the other seven octants.
  5. Move each calculated pixel position  $(x, y)$  onto the circular path centered at  $(x_c, y_c)$  and plot the coordinate values:  
$$x = x + x_c, y = y + y_c$$
  6. Repeat steps 3 through 5 until  $x \geq y$ .

# Curve Functions

- Ellipsis can be drawn similarly using a modified midpoint algorithm.
- Similar algorithms can be used to display polynomials, exponential functions, trigonometric functions, conics, probability distributions, etc.

# Curve Functions

## Ellipse

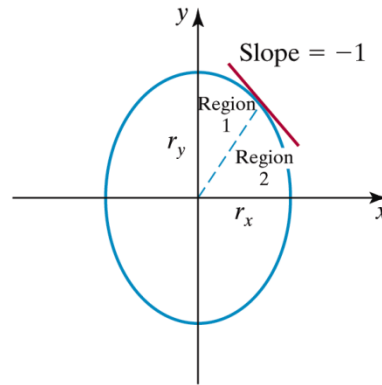


Figure 3-25

Ellipse processing regions. Over region 1, the magnitude of the ellipse slope is less than 1.0; over region 2, the magnitude of the slope is greater than 1.0.

(from Donald Hearn and Pauline Baker)

# Curve Functions

## Ellipse

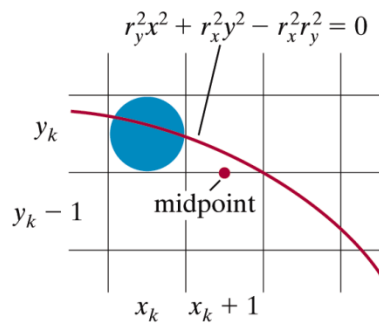


Figure 3-26

Midpoint between candidate pixels at sampling position  $x_k + 1$  along an elliptical path.

(from Donald Hearn and Pauline Baker)



# Curve Functions

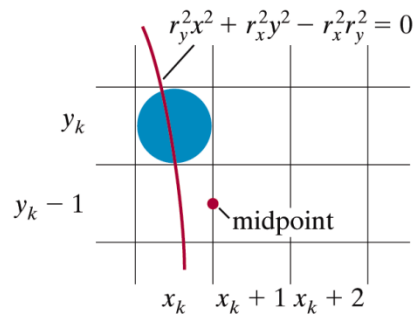


Figure 3-27

Midpoint between candidate pixels at sampling position  $y_k - 1$  along an elliptical path.

(from Donald Hearn and Pauline Baker)

# Specific Parametric Forms

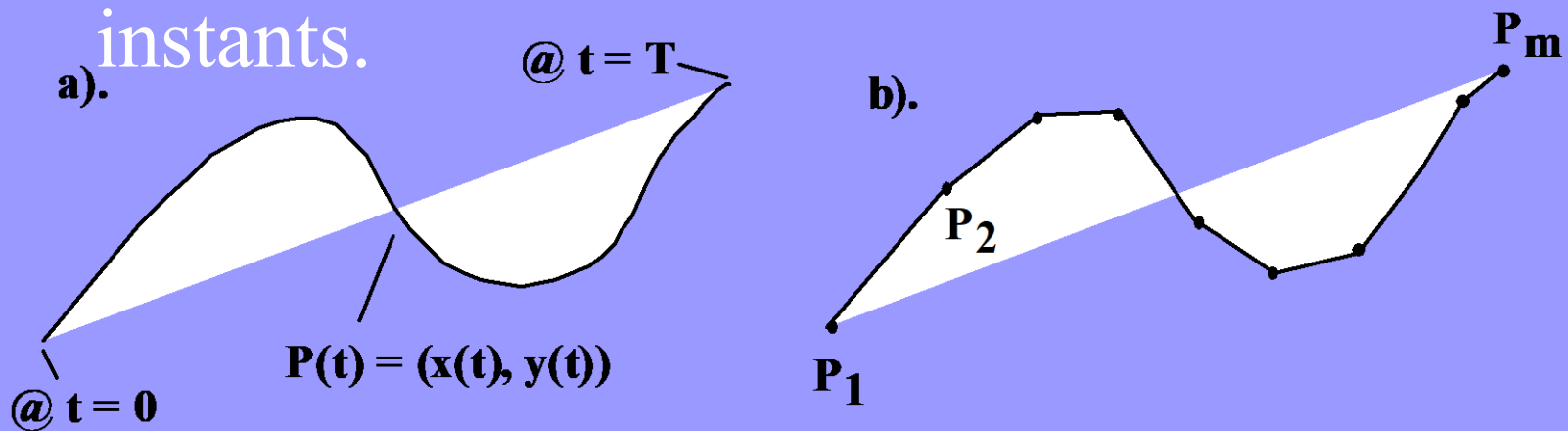
- line:
  - $x = x_1*(1-t) + x_2*t, y = y_1*(1-t) + y_2*t$
- circle:
  - $x = r*\cos(2\pi t), y = r*\sin(2\pi t)$
- ellipse:
  - $x = W*r*\cos(2\pi t), y = H*r*\sin(2\pi t)$
  - W and H are half-width and half-height.

# Finding Implicit Form from Parametric Form

- Combine the  $x(t)$  and  $y(t)$  equations to eliminate  $t$ .
- Example: ellipse:  $x = W \cdot \cos(2\pi t)$ ,  $y = H \cdot \sin(2\pi t)$ 
  - $X^2 = W^2 \cos^2(2\pi t)$ ,  $y^2 = H^2 \sin^2(2\pi t)$ .
  - Dividing by the  $W$  or  $H$  factors and adding gives  $(x/W)^2 + (y/H)^2 = 1$ , the implicit form.

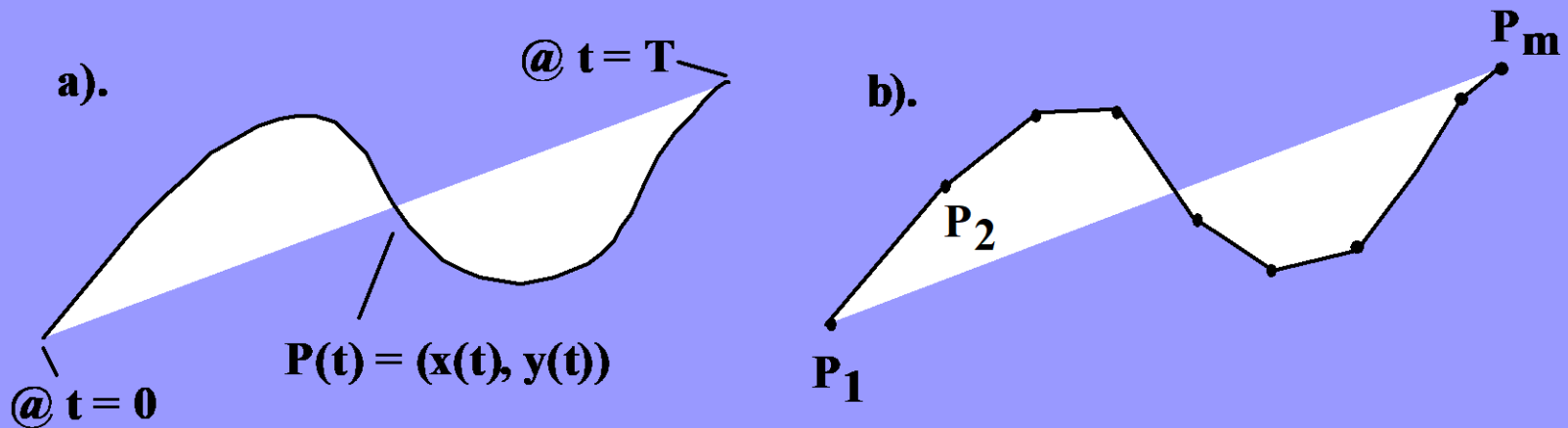
# Drawing Parametric Curves

- For a curve  $C$  with the parametric form  $P(t) = (x(t), y(t), z(t))$  as  $t$  varies from 0 to  $T$ , we use **samples** of  $P(t)$  at closely spaced instants.



# Drawing Parametric Curves (2)

- The position  $P_i = P(t_i) = (x(t_i), y(t_i), z(t_i))$  is calculated for a sequence  $\{t_i\}$  of times.
- The curve  $P(t)$  is approximated by the polyline based on this sequence of points  $P_i$ .



# Drawing Parametric Curves (3)

- Code (2D):

```
// draw the curve (x(t), y(t)) using
// the array t[0],...,t[n-1] of sample times
glBegin(GL_LINES);
    for(int i = 0; i < n; i++)
        glVertex2f((x(t[i]), y(t[i]));
glEnd();
```

# Parametric Curves: Advantages

- For drawing purposes, parametric forms circumvent all of the difficulties of implicit and explicit forms.
- Curves can be multi-valued, and they can self-intersect any number of times.
- Verticality presents no special problem:  $x(t)$  simply becomes constant over some interval in  $t$ .

# Polar Coordinates Parametric Form

- $x = r(\theta) \cos(\theta)$ ,  $y = r(\theta) \sin(\theta)$ 
  - cardioid:  $r(\theta) = K(1 + \cos(\theta))$ ,  $0 \leq \theta \leq 2\pi$
  - rose:  $r(\theta) = K \cos(n\theta)$ ,  $0 \leq \theta \leq 2n\pi$ , where  $n$  is number of petals ( $n$  odd) or twice the number of petals ( $n$  even)
  - spirals:
    - Archimedean:  $r(\theta) = K\theta$
    - logarithmic:  $r(\theta) = Ke^{a\theta}$
  - $K$  is a scale factor for the curves.



# Polar coordinates Parametric Form

## (2)

- conic sections (ellipse, hyperbola, circle, parabola):

$$r(\theta) = \frac{1}{1 \pm e \cos(\theta)}$$

– e is eccentricity:

- $e = 1$  : parabola
- $e = 0$  : circle
- $0 < e < 1$  : ellipse
- $e > 1$  : hyperbola