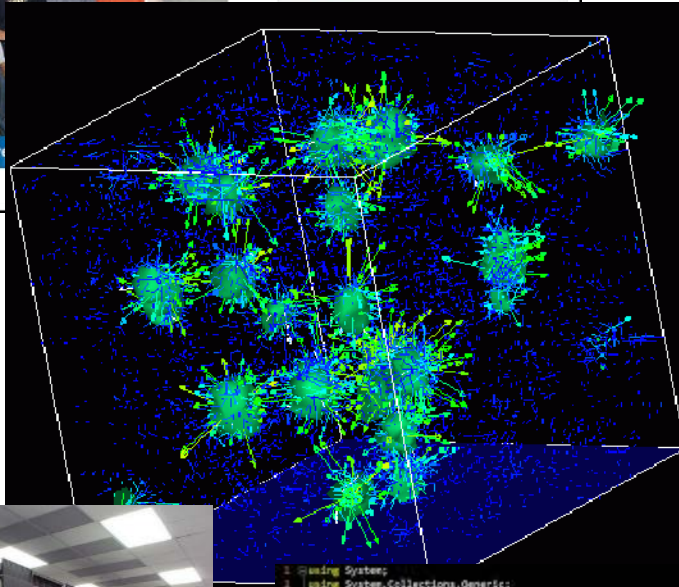




# Survey of Scientific Computing (SciComp 301)

Dave Biersach  
Brookhaven National  
Laboratory  
[dbiersach@bnl.gov](mailto:dbiersach@bnl.gov)



```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Windows.Forms;
9
10 namespace SimpleEvents
11 {
12     public partial class Form1 : Form
13     {
14         Person person = new Person();
15
16         public Form1()
17         {
18             InitializeComponent();
19             person.FirstName = "Christian";
20             person.LastName = "Pano";
21         }
22
23         private void button1_Click(object sender, EventArgs e)
24         {
25             person.MainColor = textBox1.Text;
26         }
27     }
28 }
```

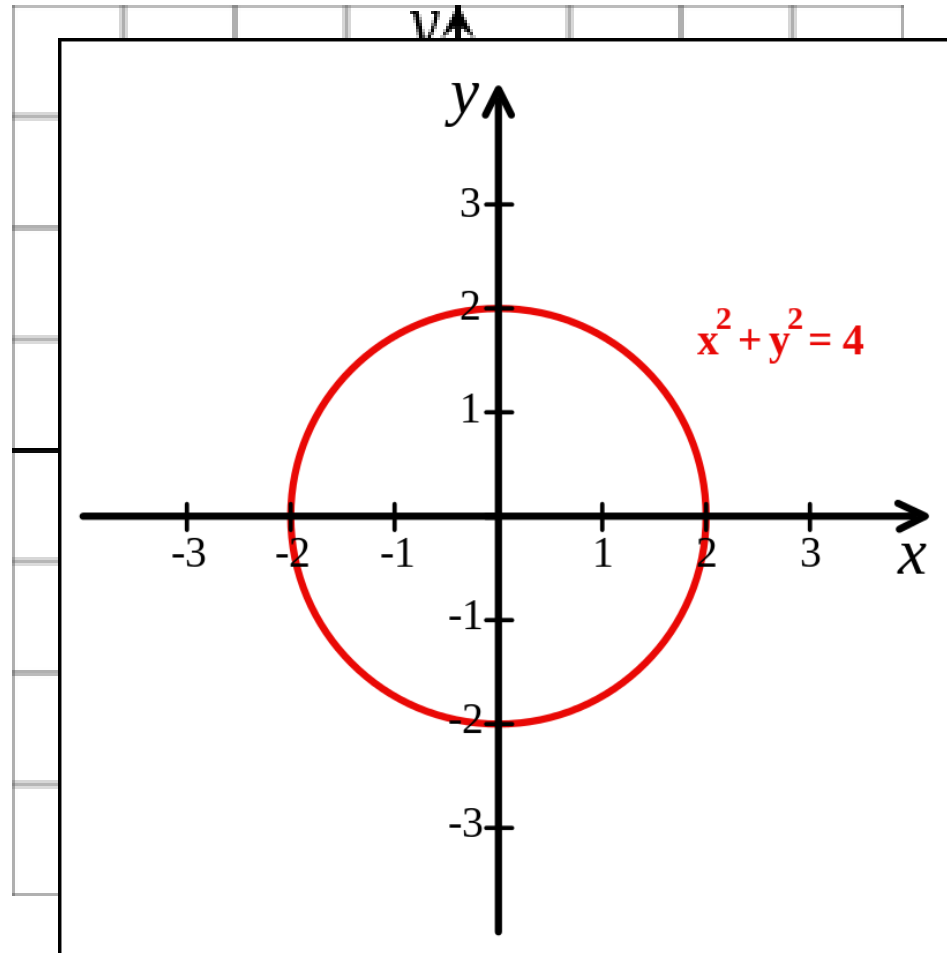
**Session 05**  
2D Graphics,  
Polar Coordinates

# Session Goals

- Learn the **SimpleScreen** custom class to draw graphics
- Understand the World **bounding rectangle**
- Use **Cartesian** coordinates to draw 2D lines
- Use **polar** coordinates to draw 2D circles
- Approximate circles by dividing  **$2\pi$  radians** into *intervals*
- Use a **delay** count to watch each interval being drawn
- Appreciate *the hidden geometry* of the **Olympic Rings**
- Draw fancy curves using **parametric equations**

# Cartesian Coordinates

Created by  
René Descartes  
in 1637

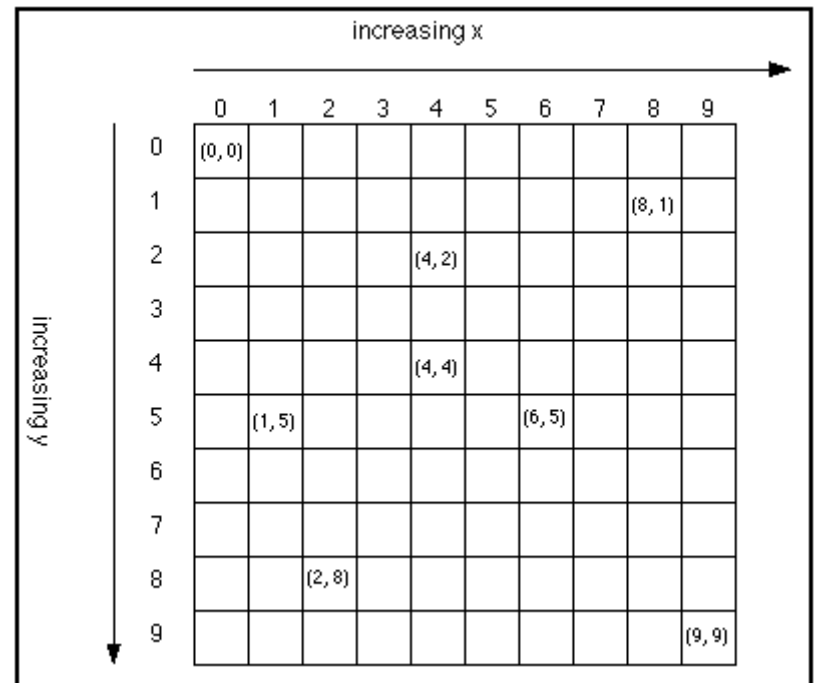


# Screen Pixels

- A computer screen is divided into **2D array** of small rectangles called **pixels**
- Every pixels has both an **X** and a **Y** coordinate
- Each pixel is set to a specific **color**
- The number of pixels in each dimension is the resolution
- Modern screens have resolutions greater than 1920 x 1080
- The **top left** screen pixel has coordinates **(0,0)** and there are no negative pixel coordinates
- Unfortunately there is an even greater **discrepancy** between Cartesian coordinates and pixel coordinates...

# Screen Y vs. World Y Coordinates

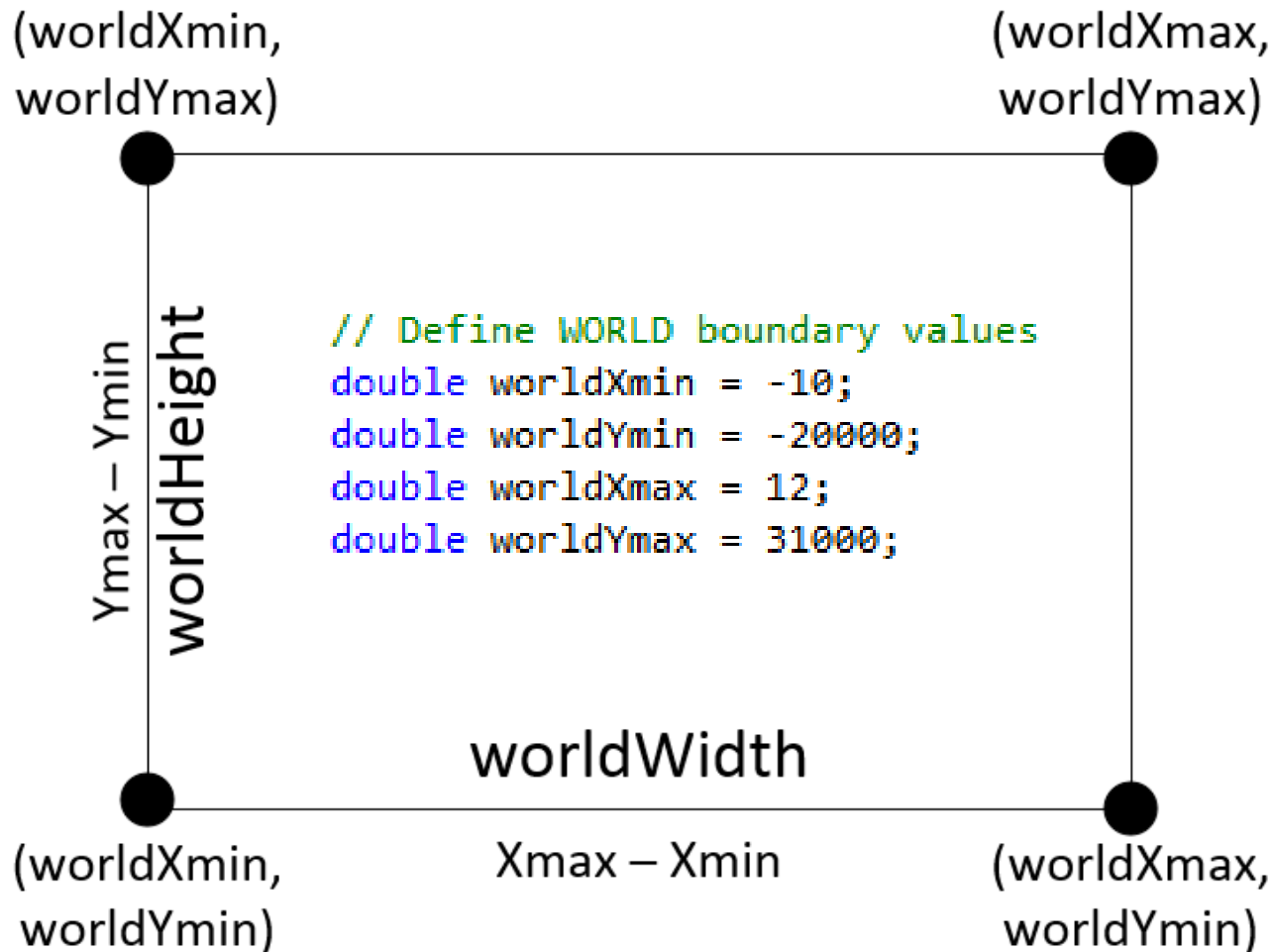
- In screen coordinates, as you move from the **top** of the screen *towards* the **bottom**, **the Y coordinate increases!**
- This is the exact opposite of world (Cartesian) coordinates
- Fortunately, the **X axis behaves the as expected** between the **screen** and **world** coordinate system
- This discrepancy will drive you nuts!



# World Bounding Rectangle

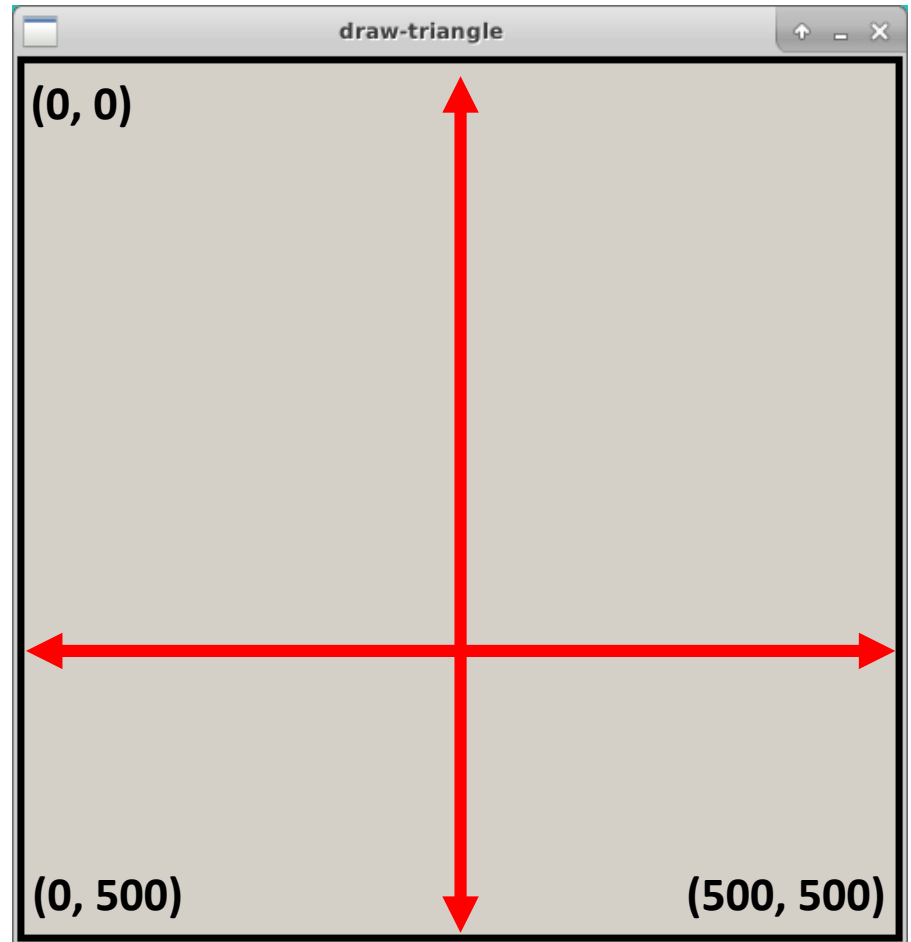
- World coordinates are what we use when doing pure math
- Screen coordinates are what we have to use when drawing
- We need a way to *map* **World X,Y** to **Screen X,Y**
- The world is framed by a “**bounding rectangle**” which is comprised of four variables of type **double**
  - **worldXmin, worldYmin**
  - **worldXmax, worldYmax**
- The values for all four World variables are **our choice** – they can be as big or as small as we want them

# World Bounding Rectangle



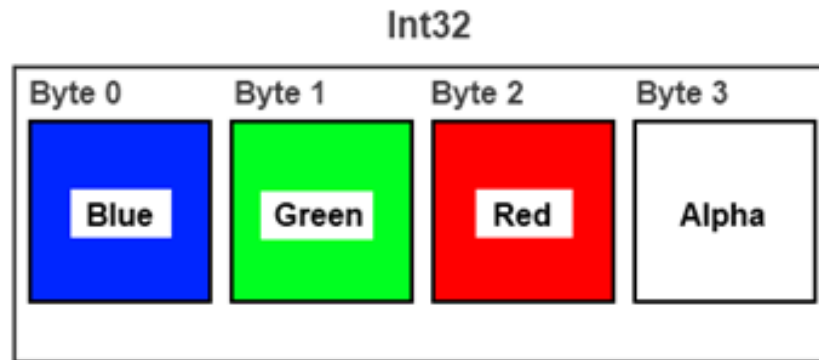
# Screen Bounding Rectangle

- The Screen bounding rectangle is the size of the **Allegro** display bitmap
- Due to your smaller laptop screens, we set the size to be **501 x 501**
- Remember Screen Y coordinates are **inverted** as **(0,0)** is in the **top** left



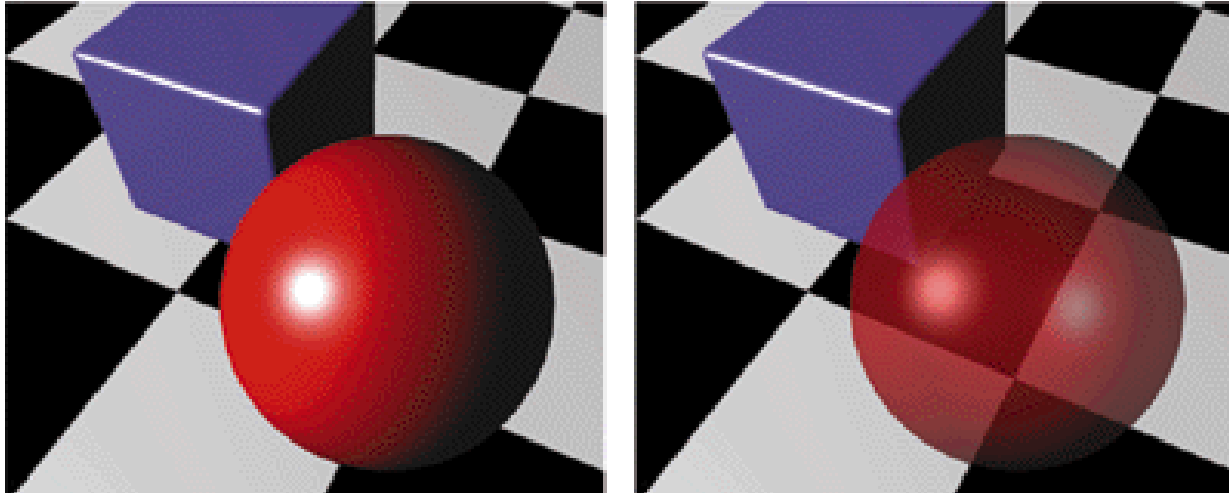


# Argb Color Encoding



```
ALLEGRO_COLOR clrRed = al_map_rgb(255, 0, 0);    // Red
ALLEGRO_COLOR clrGreen = al_map_rgb(0, 255, 0);  // Green
ALLEGRO_COLOR clrBlue = al_map_rgb(0, 0, 255);   // Blue
```

# Alpha Blending = Opacity



Alpha = 0 means fully transparent (see through)  
Alpha = 255 means fully opaque

We will be using **Alpha = 255** in our code

# Various Colors in **RGB** Values


<b>94.28.13</b>  11° 86% 37% 01 Dark Skin	<b>241.148.108</b>  18° 55% 95% 02 Light Skin	<b>97.119.171</b>  222° 43% 67% 03 Blue Sky	<b>90.103.39</b>  72° 62% 40% 04 Foliage	<b>164.131.196</b>  270° 33% 77% 05 Blue Flower	<b>140.253.153</b>  127° 46% 99% 06 Bluish Green
<b>255.116.21</b>  24° 92% 100% 07 Orange	<b>7.47.122</b>  219° 94% 48% 08 Purplish Blue	<b>222.29.42</b>  356° 87% 87% 09 Moderate Red	<b>69.0.68</b>  301° 100% 27% 10 Purple	<b>187.255.19</b>  77° 93% 100% 11 Yellow Green	<b>255.142.0</b>  33° 100% 100% 12 Orange Yellow
<b>0.0.142</b>  240° 100% 56% 13 Blue	<b>64.173.38</b>  108° 78% 68% 14 Green	<b>203.0.0</b>  0° 100% 80% 15 Red	<b>255.217.0</b>  51° 100% 100% 16 Yellow	<b>207.3.124</b>  324° 99% 81% 17 Magenta	<b>0.148.189</b>  193° 100% 74% 18 Cyan
<b>255.255.255</b>  0° 0% 100% 19 White	<b>249.249.249</b>  0° 0% 98% 20 Neutral 8	<b>180.180.180</b>  0° 0% 71% 21 Neutral 6.5	<b>117.117.117</b>  0° 0% 46% 22 Neutral 5	<b>53.53.53</b>  0° 0% 21% 23 Neutral 3.5	<b>0.0.0</b>  0° 0% 0% 24 Black

# Predefined Color “Names”

AliceBlue	AntiqueWhite	Aqua	Aquamarine	Azure	Beige	Bisque
Black	BlanchedAlmond	Blue	BlueViolet	Brown	BurlyWood	CadetBlue
Chartreuse	Chocolate	Coral	CornflowerBlue	Cornsilk	Crimson	Cyan
DarkBlue	DarkCyan	DarkGoldenrod	DarkGray	DarkGreen	DarkKhaki	DarkMagenta
DarkOliveGreen	DarkOrange	DarkOrchid	DarkRed	DarkSalmon	DarkSeaGreen	DarkSlateBlue
DarkSlateGray	DarkTurquoise	DarkViolet	DeepPink	DeepSkyBlue	DimGray	DodgerBlue
Feldspar	Firebrick	FloralWhite	ForestGreen	Fuchsia	Gainsboro	GhostWhite
Gold	Goldenrod	Gray	Green	GreenYellow	Honeydew	HotPink
IndianRed	Indigo	Ivory	Khaki	Lavender	LavenderBlush	LawnGreen
LemonChiffon	LightBlue	LightCoral	LightCyan	LightGoldenrodYellow	LightGray	LightGreen
LightPink	LightSalmon	LightSeaGreen	LightSkyBlue	LightSlateBlue	LightSlateGray	LightSteelBlue
LightYellow	Lime	LimeGreen	Linen	Magenta	Maroon	MediumAquamarine
MediumBlue	MediumOrchid	MediumPurple	MediumSeaGreen	MediumSlateBlue	MediumSpringGreen	MediumTurquoise
MediumVioletRed	MidnightBlue	MintCream	MistyRose	Moccasin	NavajoWhite	Navy
OldLace	Olive	OliveDrab	Orange	OrangeRed	Orchid	PaleGoldenrod
PaleGreen	PaleTurquoise	PaleVioletRed	PapayaWhip	PeachPuff	Peru	Pink
Plum	PowderBlue	Purple	Red	RosyBrown	RoyalBlue	SaddleBrown
Salmon	SandyBrown	SeaGreen	SeaShell	Sienna	Silver	SkyBlue
SlateBlue	SlateGray	Snow	SpringGreen	SteelBlue	Tan	Teal
Thistle	Tomato	Transparent	Turquoise	TVBlack	TVWhite	Violet
VioletRed	Wheat	White	WhiteSmoke	Yellow	YellowGreen	

# Allegro Graphics Library

<http://liballeg.org>

*A game programming library*

**Allegro**  
About  
Git repository  
License  
Language bindings

**Downloads**  
Latest version  
Older versions  
Extra addons

**Documentation**  
Latest version  
All versions  
Tutorials  
Books  
Wiki

**Support**  
Forums  
Mailing lists  
Bug tracker  
IRC

**Community**  
Allegro.cc  
Misc

## Welcome to Allegro!

Allegro is a cross-platform library mainly aimed at video game and multimedia programming. It handles common, low-level tasks such as creating windows, accepting user input, loading data, drawing images, playing sounds, etc. and generally abstracting away the underlying platform. However, Allegro is *not* a game engine: you are free to design and structure your program as you like.

Allegro 5 has the following additional features:

- Supported on Windows, Linux, Mac OSX, iPhone and Android
- User-friendly, intuitive C API usable from C++ and many other languages
- Hardware accelerated bitmap and graphical primitive drawing support (via OpenGL or Direct3D)
- Audio recording support
- Font loading and drawing
- Video playback
- Abstractions over shaders and low-level polygon drawing
- And [more!](#)

## News

```

// SimpleScreen
class SimpleScreen
{
public:
    SimpleScreen(void(*draw)(SimpleScreen& ss) = nullptr,
                  void(*eventHandler)(SimpleScreen& ss, ALLEGRO_EVENT& ev)= nullptr);

    ~SimpleScreen();

    void SetWorldRect(double xMin, double yMin,
                     double xMax, double yMax);

    void GetWorldRect();

    void SetProjection(double degrees = 45, double correction = 1);

    void SetCameraLocation(double x, double y, double z);

    void SetZoomFrame(string clr, double width = 2);

    void LockDisplay();

    void UnlockDisplay();

    void Clear();

    void Update();

    void Redraw();

    void Exit();

    void HandleEvents();

    bool Contains(double x, double y);

    void DrawAxes(string clr = "black", float width = 1);

    void DrawLine(Point2D &a, Point2D &b,
                  string clr = "black", float width = 1);

    void DrawLines(PointSet* ps, string clr, float width = 1,
                   bool close = true, bool fill = false, long delay = 0);

    void DrawLines(FacetSet* facets, string clr, float width = 1,
                   bool fill = false, long delay = 0);

```

# The SimpleScreen Class

Your App

SimpleScreen

Allegro

# The **PointSet** Class

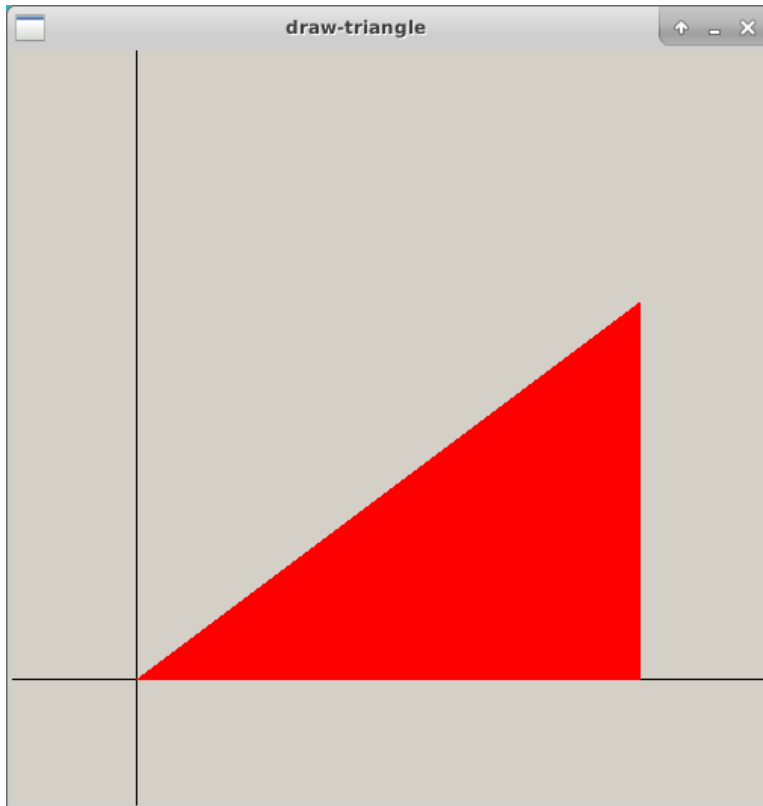
```
// PointSet
class PointSet
{
public:
    PointSet();
    ~PointSet();
    Point2D* at(size_t i);
    void clear();
    void add(double x, double y);
    size_t size();
private:
    vector<Point2D*>* points;
};
```

# Open Lab 1 – Draw Triangle

- No coding required – just follow along `main()`
- Draw a 3-4-5 right triangle in the *first* quadrant
- Paint the entire triangle RED including the interior
- Set the correct *world rectangle* dimensions



# View Lab 1 – Draw Triangle



```
#include "stdafx.h"
#include "simplescreen.h"

using namespace std;

int main()
{
    SimpleScreen ss;
    ss.SetWorldRect(-1, -1, 5, 5);
    ss.DrawAxes();

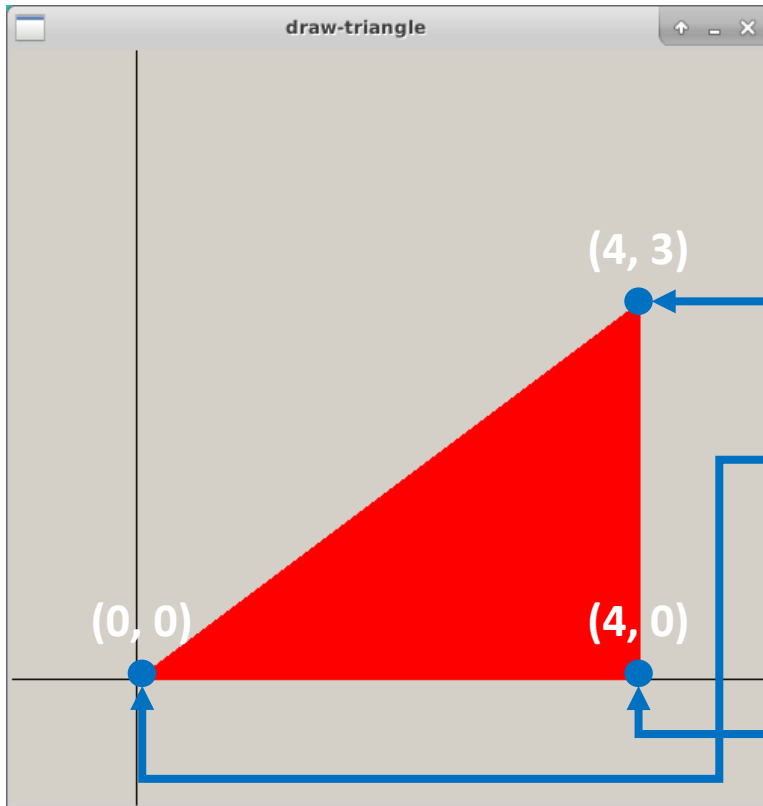
    PointSet ps;
    ps.add(0, 0);
    ps.add(4, 0);
    ps.add(4, 3);

    ss.DrawLines(&ps, "red", 1, true, true);

    ss.HandleEvents();

    return 0;
}
```

# View Lab 1 – Draw Triangle



```
#include "stdafx.h"
#include "simplescreen.h"

using namespace std;

int main()
{
    SimpleScreen ss;
    ss.SetWorldRect(-1, -1, 5, 5);
    ss.DrawAxes();

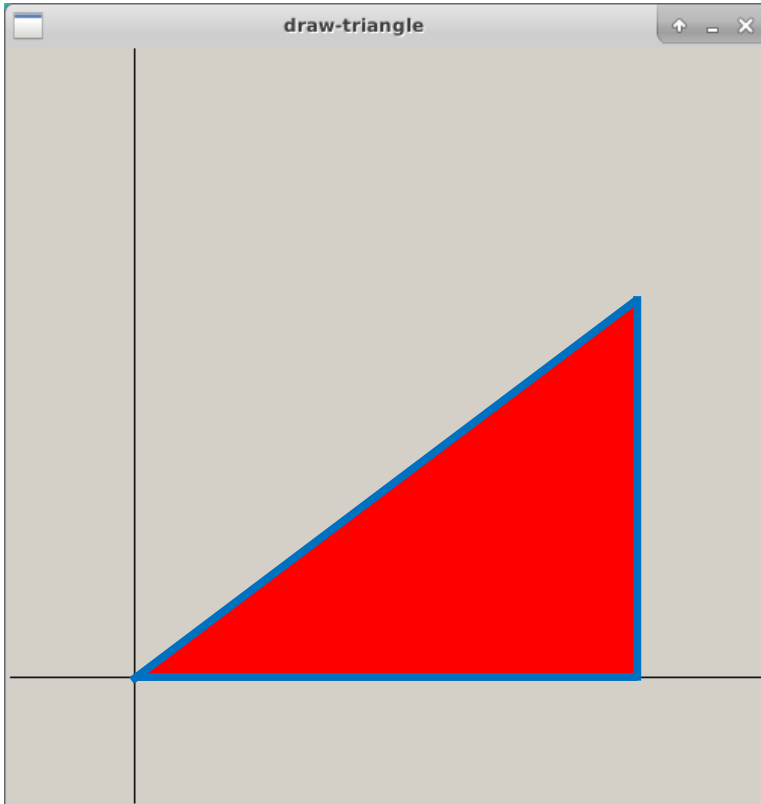
    PointSet ps;
    ps.add(0, 0);
    ps.add(4, 0);
    ps.add(4, 3);

    ss.DrawLines(&ps, "red", 1, true, true);

    ss.HandleEvents();

    return 0;
}
```

# View Lab 1 – Draw Triangle



```
#include "stdafx.h"
#include "simplescreen.h"

using namespace std;

int main()
{
    SimpleScreen ss;
    ss.SetWorldRect(-1, -1, 5, 5);
    ss.DrawAxes();

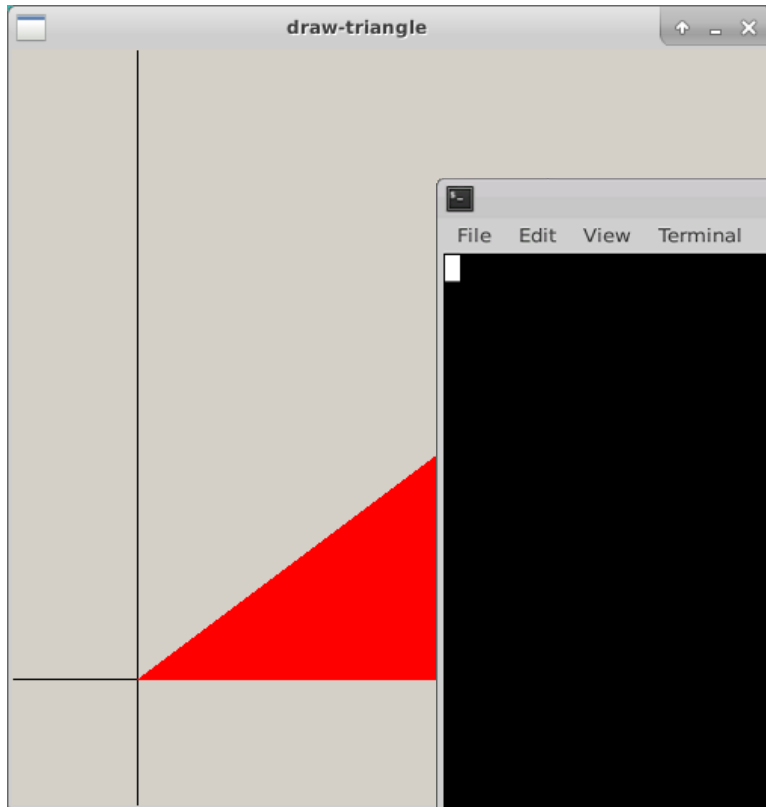
    PointSet ps;
    ps.add(0, 0);
    ps.add(4, 0);
    ps.add(4, 3);

    ss.DrawLines(&ps, "red", 1, true, true);

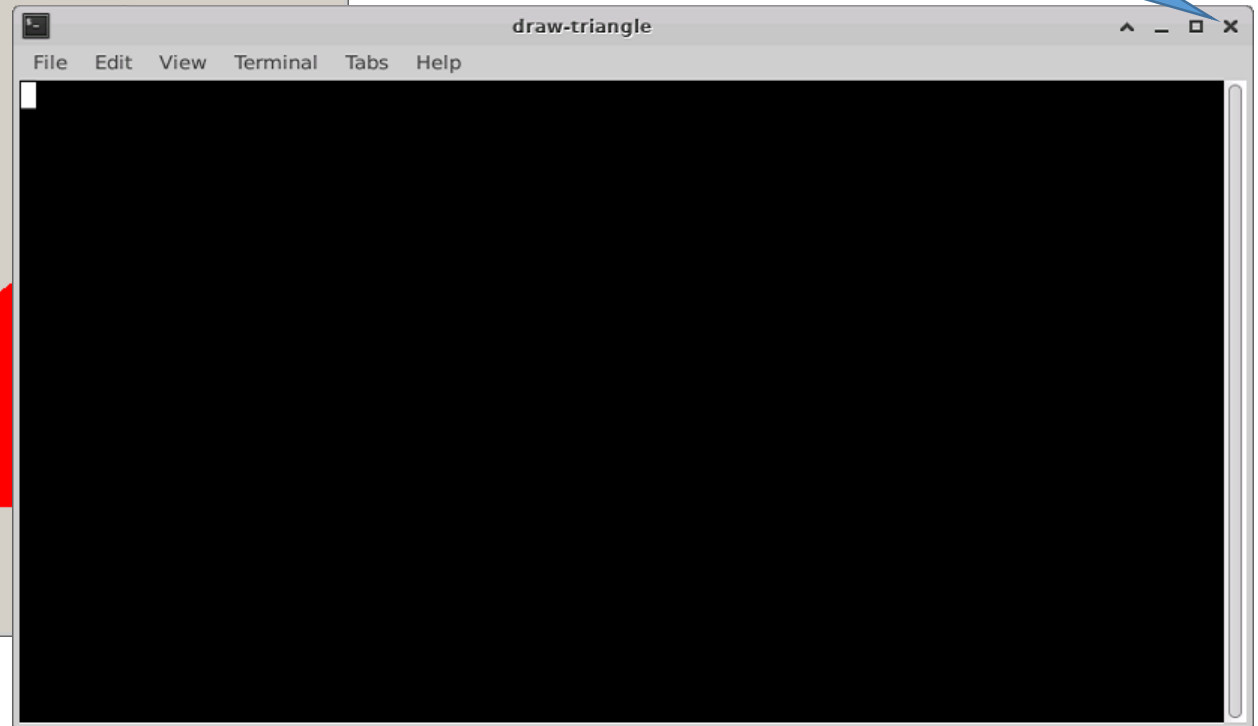
    ss.HandleEvents();

    return 0;
}
```

# Run Lab 1 – Draw Triangle



To stop an Allegro app,  
close the backing terminal  
session FIRST



## Open Lab 2 – Draw Rectangle

- Draw a **violet** rectangle in the first quadrant with the bottom left corner located at (0,0) and the longest side on the x-axis
- The dimensions are: **Area = 210** and **Perimeter = 62**
- Assume the world coordinates are (-30,-30) to (30,30)

```
int main()
{
    SimpleScreen ss;
    ss.SetWorldRect(-30, -30, 30, 30);
    ss.DrawAxes();

    PointSet ps;
    ps.add(0, 0);
    ps.add(5, 0);
    ps.add(5, 5);
    ps.add(0, 5);

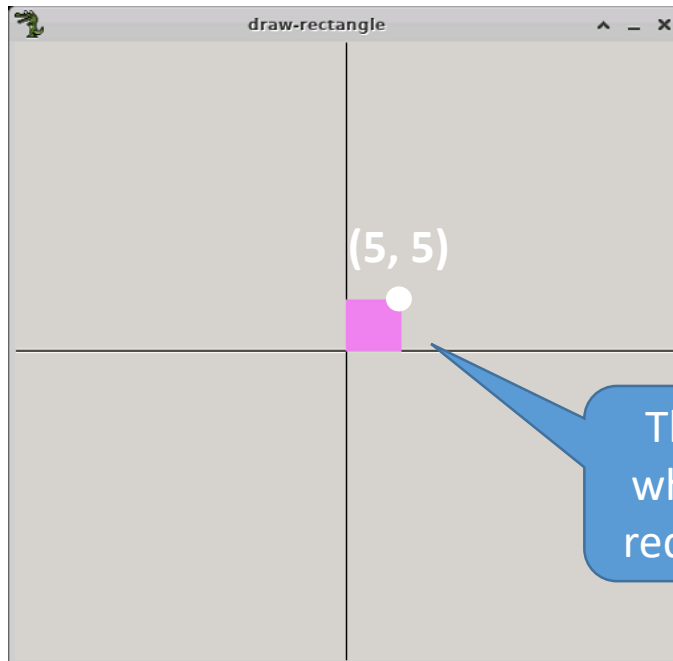
    ss.DrawLines(&ps, "violet", 1, true, true);

    ss.HandleEvents();

    return 0;
}
```

## Run Lab 2 – Draw Rectangle

- Draw a **violet** rectangle in the first quadrant with the bottom left corner located at (0,0) and the longest side on the x-axis
- The dimensions are: **Area = 210** and **Perimeter = 62**
- Assume the world coordinates are (-30,-30) to (30,30)



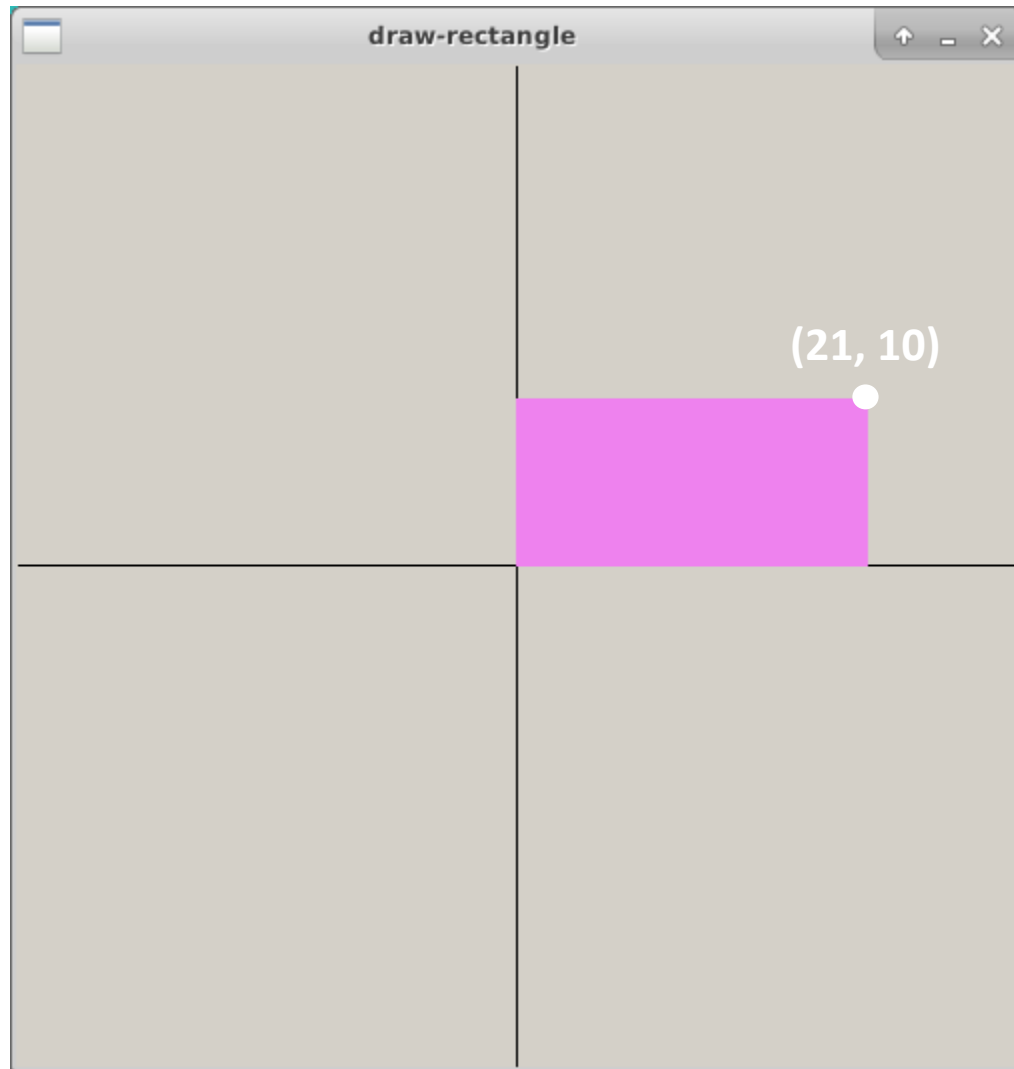
## Edit Lab 2 – Draw Rectangle

- Draw a **violet** rectangle in the first quadrant with the bottom left corner located at (0,0) and the longest side on the x-axis
- The dimensions are: **Area = 210** and **Perimeter = 62**
- Assume the world coordinates are (-30,-30) to (30,30)
- Edit the coordinates of the **three remaining** corner points with the correct values

```
PointSet ps;  
ps.add(0, 0);  
ps.add(5, 0);  
ps.add(5, 5);  
ps.add(0, 5);
```

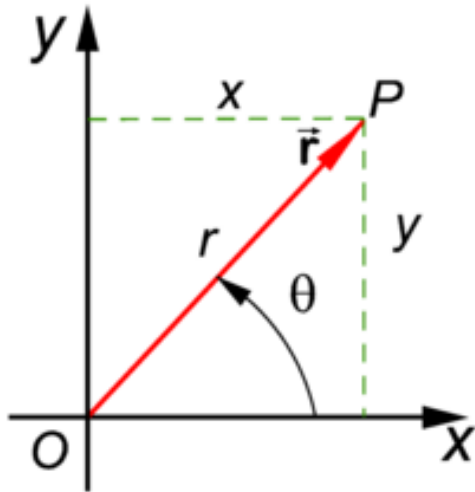
Fix these  
last 3 points

## Check Lab 2 – Draw Rectangle

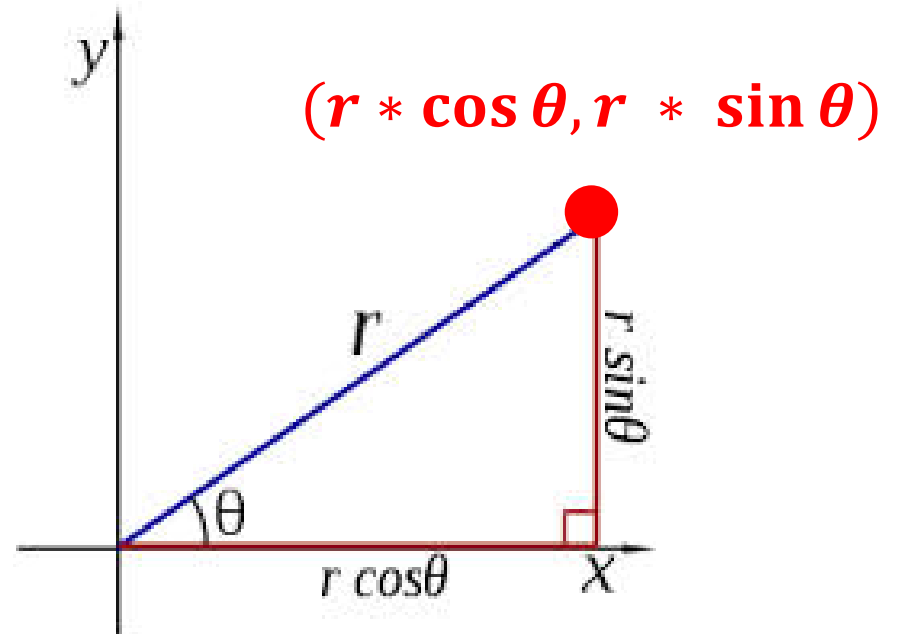




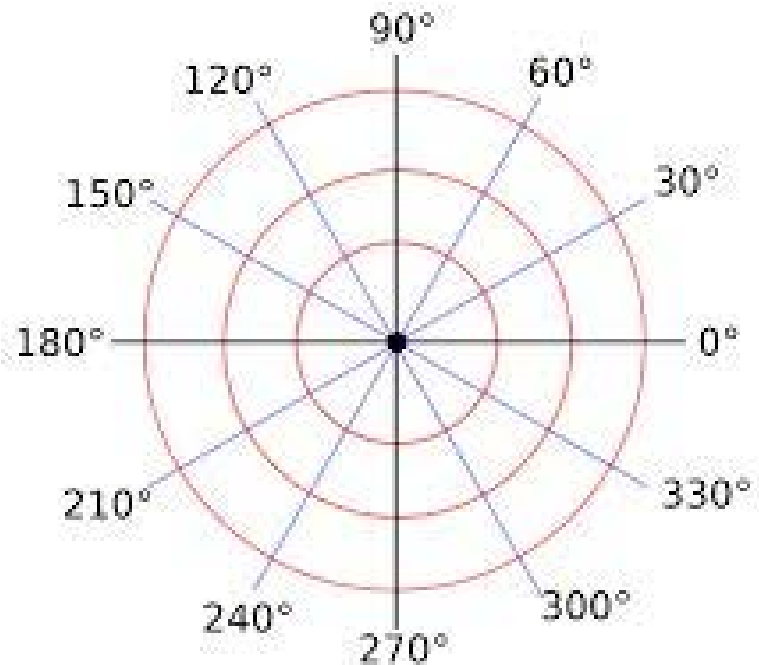
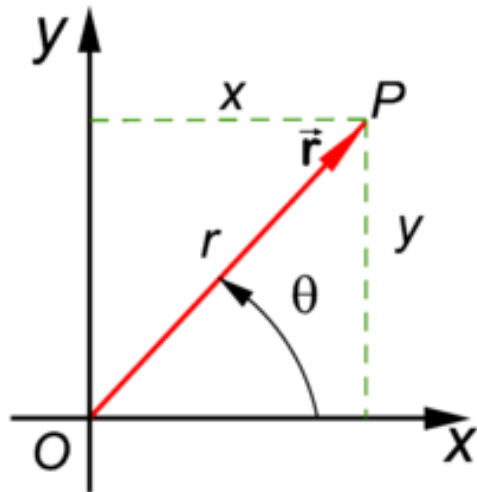
# Polar Coordinates



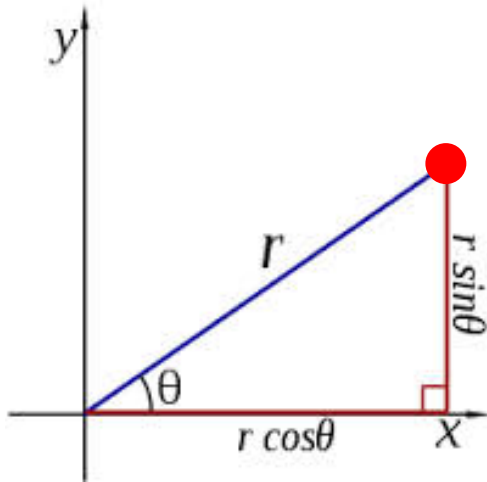
A radius and an angle (theta) make a 2D polar coordinate



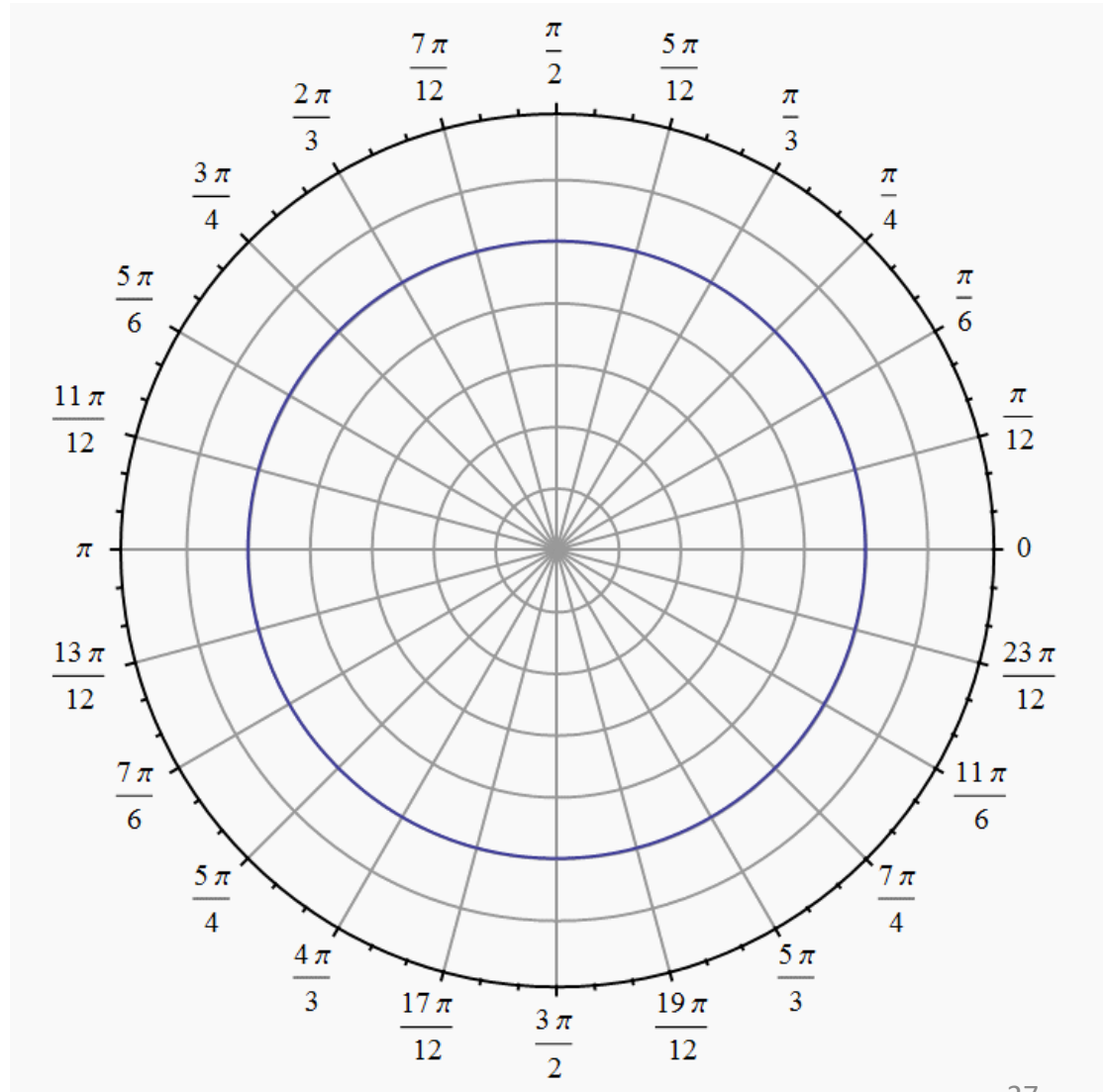
# Polar Coordinates



# Polar Coordinates



Angles are  
measured in  
**radians**  
( $0 \leq \theta \leq 2\pi$ )



# Open Lab 3 – Draw Circle

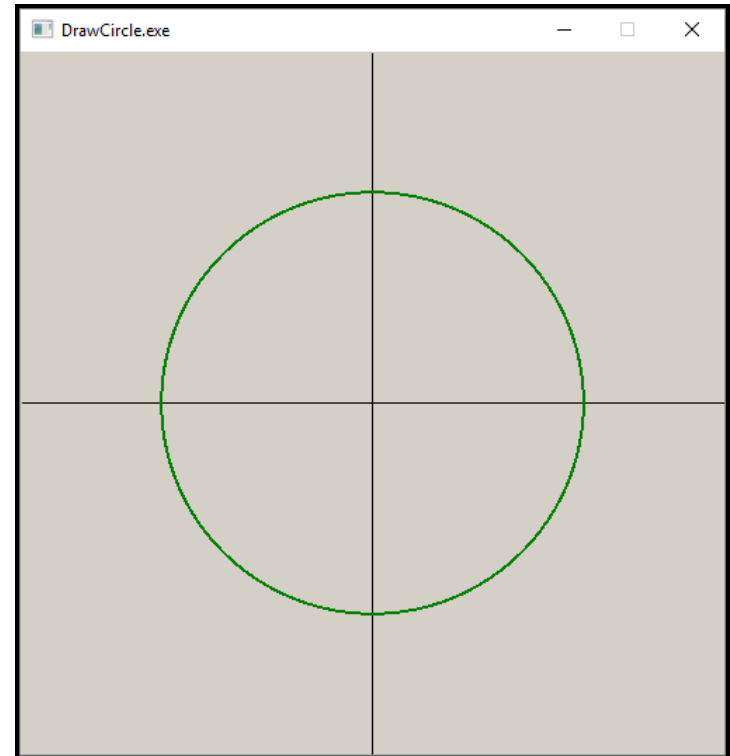
```
void draw(SimpleScreen& ss) {
    ss.DrawAxes();

    double radius{ 3 };
    int intervals{ 97 };
    double deltaTheta{ 2.0 * M_PI / intervals };
    PointSet psCircle;

    for (int i{}; i < intervals; ++i) {
        double theta = deltaTheta * i;
        double x = radius * cos(theta);
        double y = radius * sin(theta);
        psCircle.add(x, y);
    }

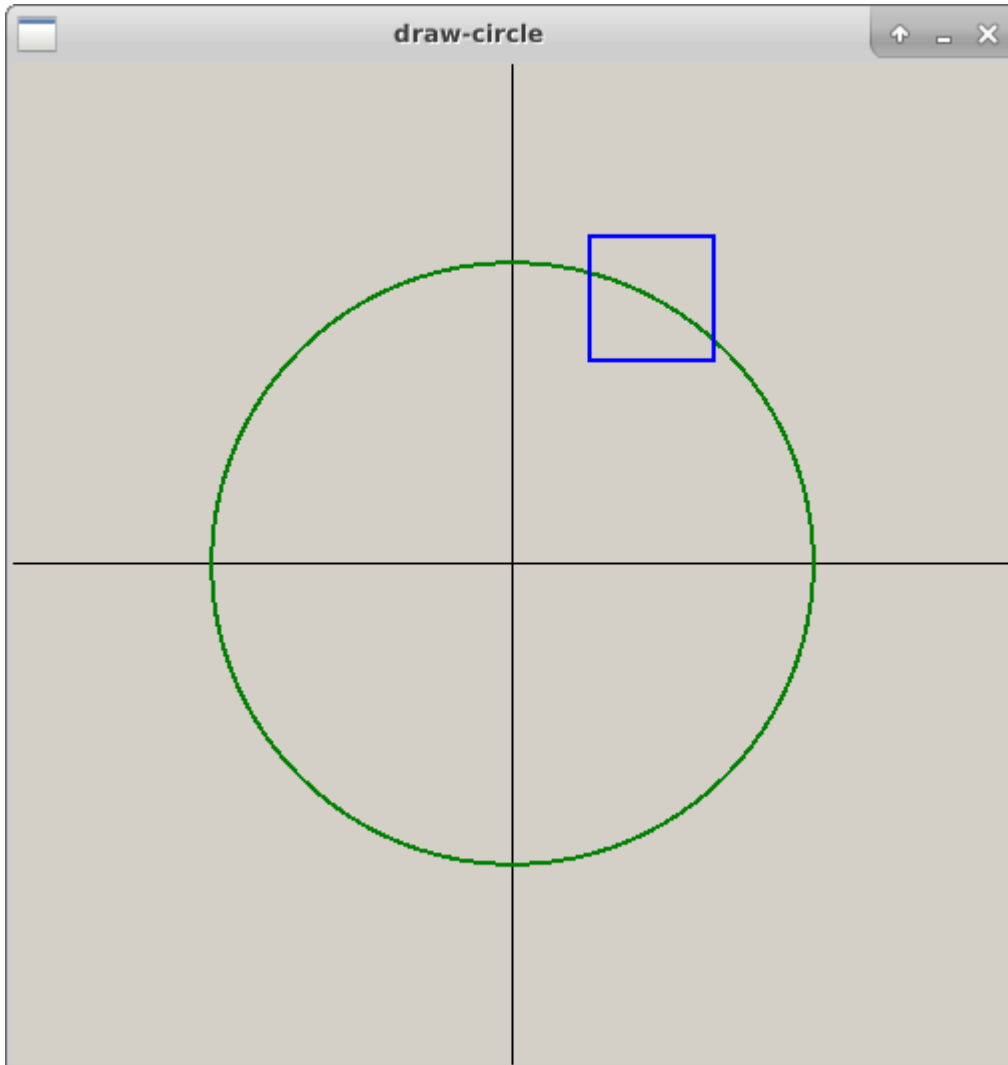
    ss.DrawLines(&psCircle, "green", 2);
}

int main()
{
    SimpleScreen ss(draw);
    ss.SetWorldRect(-5, -5, 5, 5);
    ss.HandleEvents();
    return 0;
}
```



**draw()** is a “callback function”  
It is your custom drawing  
code that is called by the  
**SimpleScreen** object

## Run Lab 3 – Draw Circle



By using the callback pattern, the framework can call your custom **draw()** function whenever it needs to repaint the screen. This enables features like **zooming**

You can **left click and drag** on the canvas to outline a zoom frame. When you release the left mouse button it will set the new **world rectangle** to the coordinates of the zoom frame and redraw the image

A **single right click** will restore each of the prior zoom levels

## Edit Lab 3 – Draw Circle

- First, change intervals to **8**, run the app, then leave it at 8 for the next steps – **what shape do you see?**
- Then change call to **ss.DrawLines()** as follows, and run the app **each time** after making these individual changes:

```
ss.DrawLines(psCircle, "blue", 2, false);
```

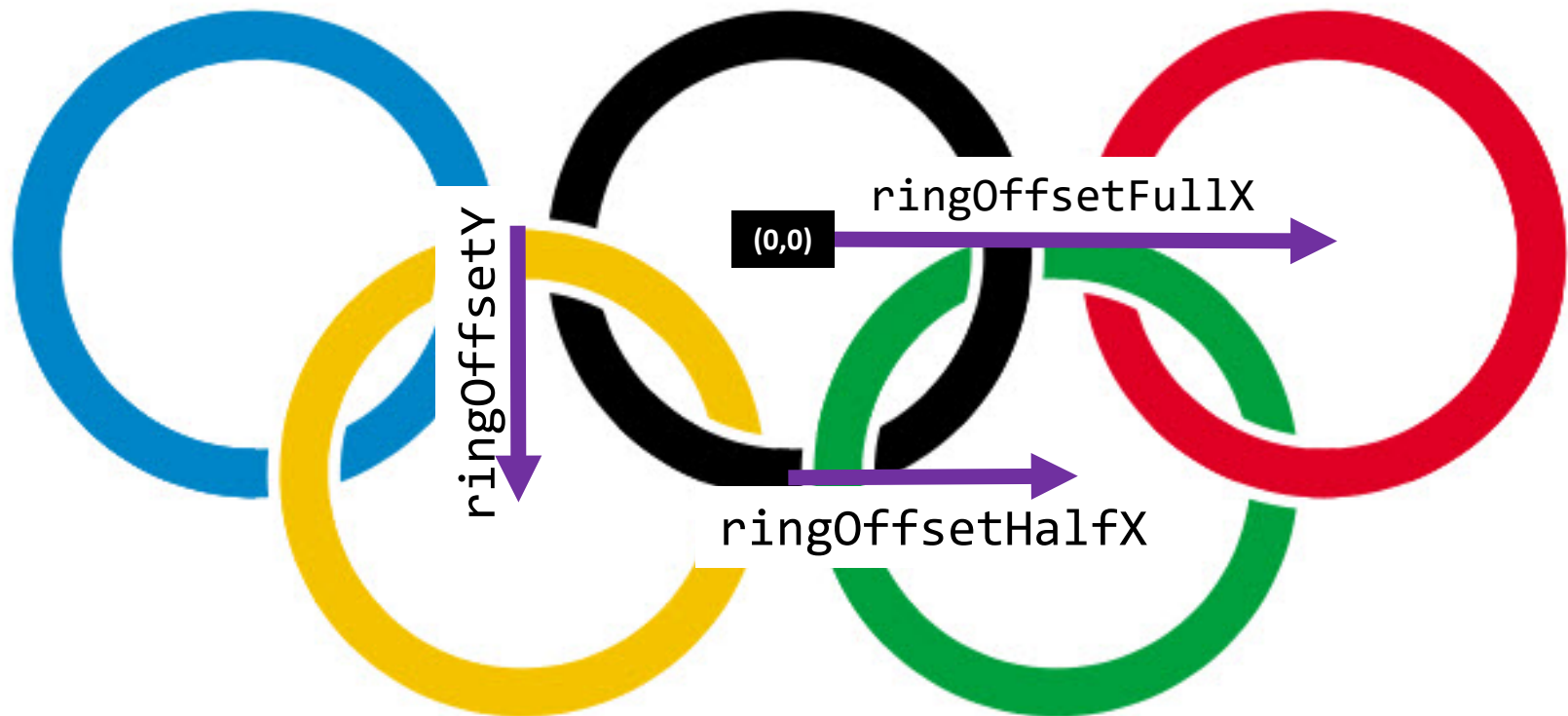
```
ss.DrawLines(psCircle, "red", 2, true, true);
```

```
void DrawLines(PointSet* ps, string clr, float width = 1,  
    bool close = true, bool fill = false, long delay = 0);
```

# Creating a New Primitive

```
void SimpleScreen::DrawCircle(double centerX, double centerY,  
                             double radius, string clr, int width)  
{  
    PointSet* psCircle = new PointSet();  
    const int intervals = 97;  
    const double deltaTheta = 2.0 * M_PI / intervals;  
    for (int i{}; i < intervals; ++i)  
    {  
        double theta = deltaTheta * i;  
        double x = centerX + radius * cos(theta);  
        double y = centerY + radius * sin(theta);  
        psCircle->add(x, y);  
    }  
    DrawLines(psCircle, clr, width);  
}
```

# Drawing the Olympic Rings





# Open Lab 4 – Draw the Olympic Rings




```
void draw(SimpleScreen& ss)
{
    double radius{ 5 };
    int width{ 15 };

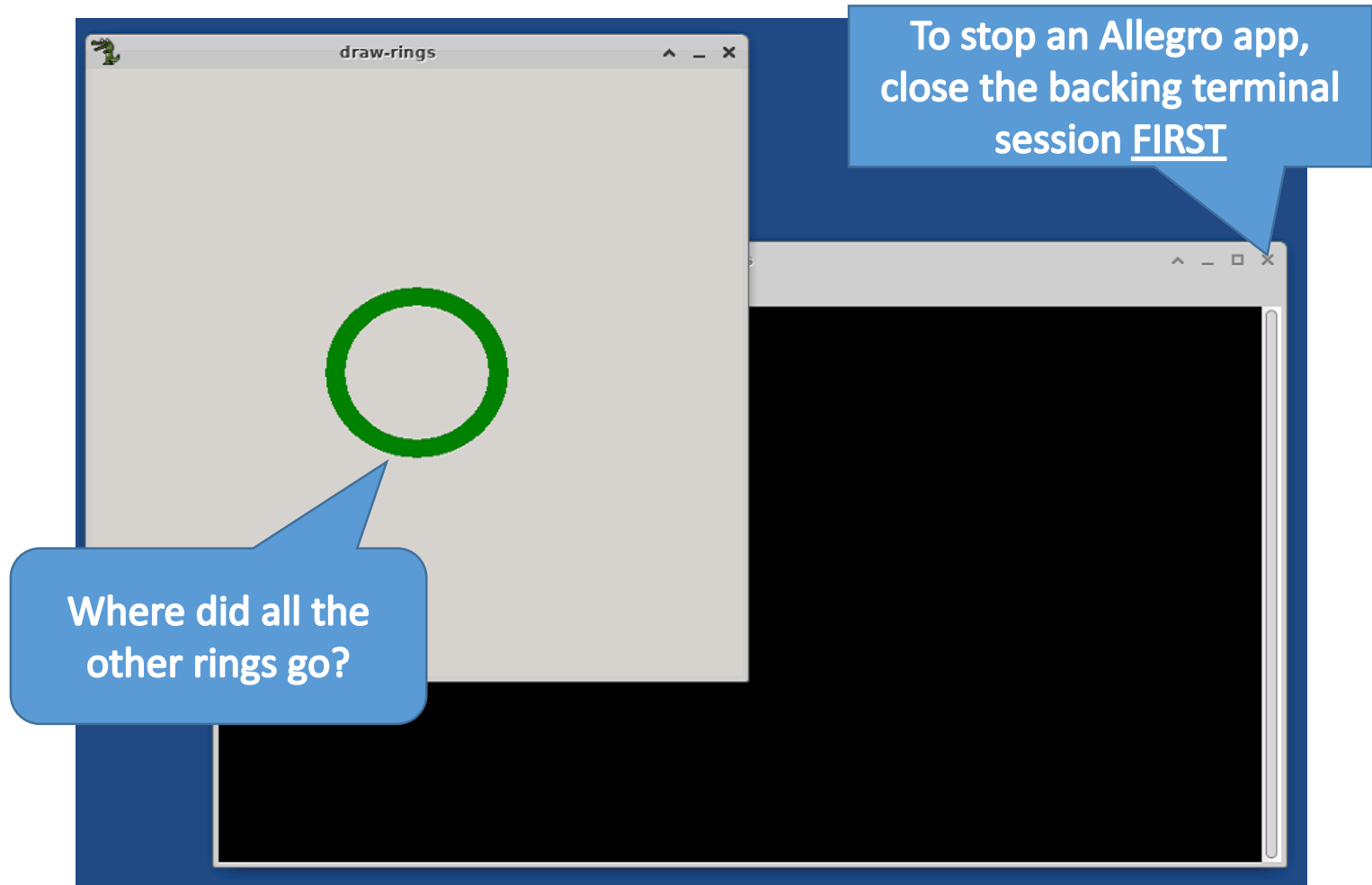
    // You must determine proper offsets
    double ringOffsetFullX{};
    double ringOffsetHalfX{};
    double ringOffsetY{};

    ss.DrawCircle(0, 0, radius, "black", width);
    ss.DrawCircle(-ringOffsetFullX, 0, radius, "blue", width);
    ss.DrawCircle(ringOffsetFullX, 0, radius, "red", width);
    ss.DrawCircle(-ringOffsetHalfX, -ringOffsetY, radius, "yellow", width);
    ss.DrawCircle(ringOffsetHalfX, -ringOffsetY, radius, "green", width);
}

int main()
{
    SimpleScreen ss(draw);
    ss.SetWorldRect(-20, -20, 20, 20);
    ss.HandleEvents();
    return 0;
}
```



# Run Lab 4 – Draw the Olympic Rings



# Open Lab 4 – Draw the Olympic Rings

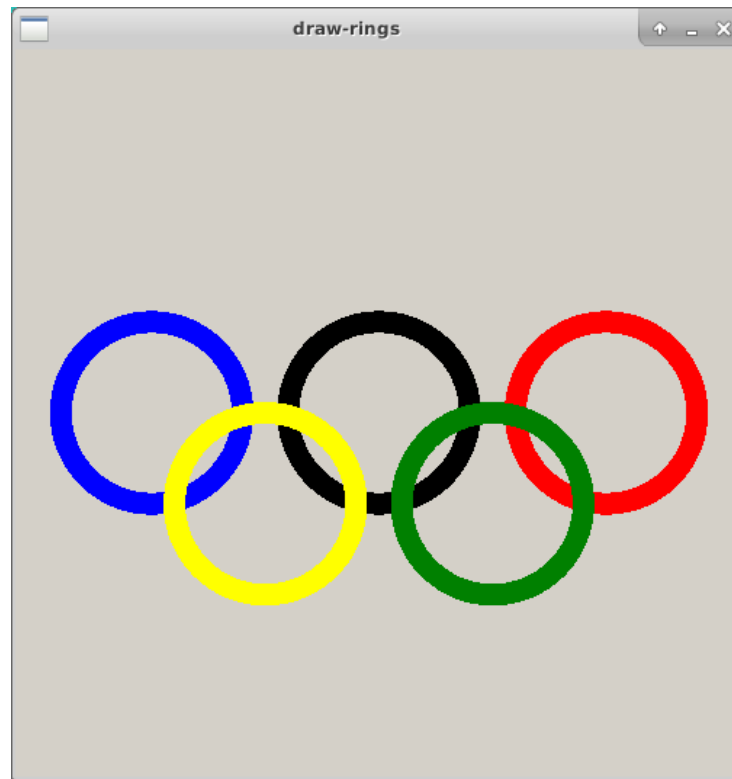
```
void draw(SimpleScreen& ss)
{
    double radius{ 5 };
    int width{ 15 };

    // You must determine proper offsets
    double ringOffsetFullX{};
    double ringOffsetHalfX{};
    double ringOffsetY{};

    ss.DrawCircle(0, 0, radius, "black", width);
    ss.DrawCircle(-ringOffsetFullX, 0, radius, "blue", width);
    ss.DrawCircle(ringOffsetFullX, 0, radius, "red", width);
    ss.DrawCircle(-ringOffsetHalfX, -ringOffsetY, radius, "yellow", width);
    ss.DrawCircle(ringOffsetHalfX, -ringOffsetY, radius, "green", width);
}

int main()
{
    SimpleScreen ss(draw);
    ss.SetWorldRect(-20, -20, 20, 20);
    ss.HandleEvents();
    return 0;
}
```

## Check Lab 4 – Draw the Olympic Rings

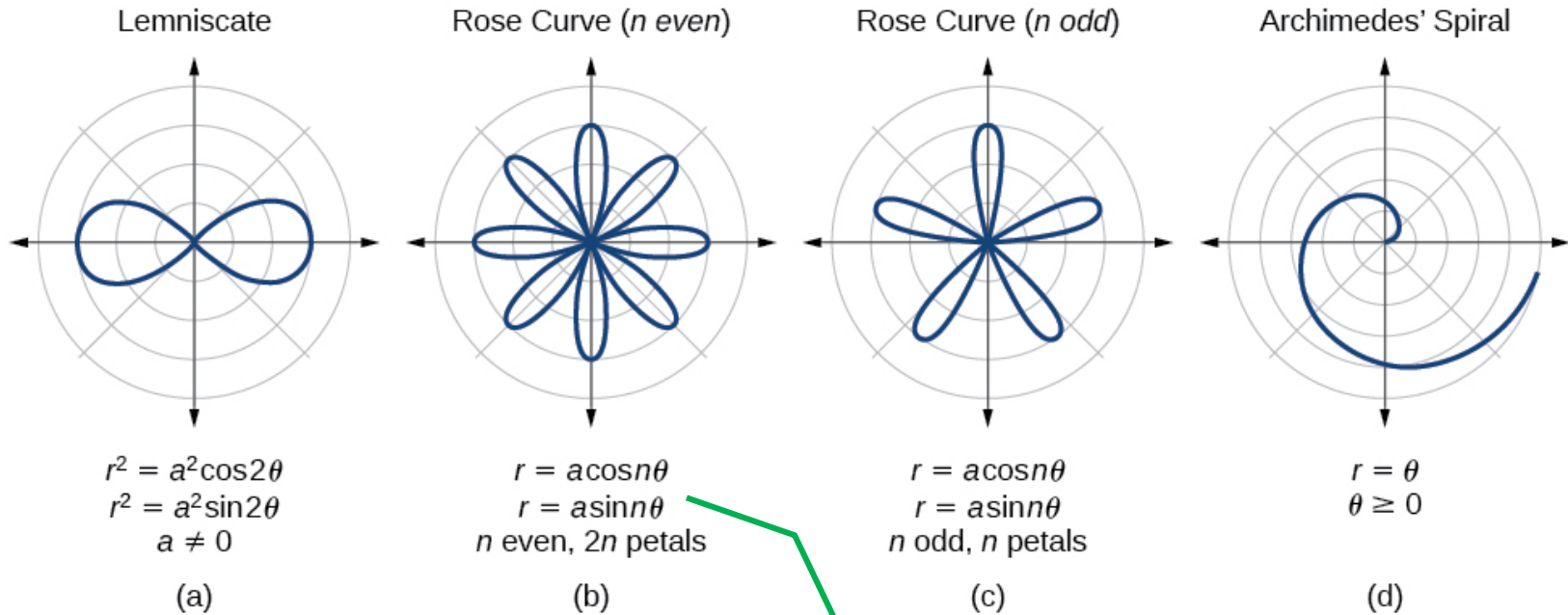


```
// You must determine proper offsets  
double ringOffsetFullX{radius * 5. / 2};  
double ringOffsetHalfX{radius * 5. / 4};  
double ringOffsetY{radius};
```

# Parametric Curves



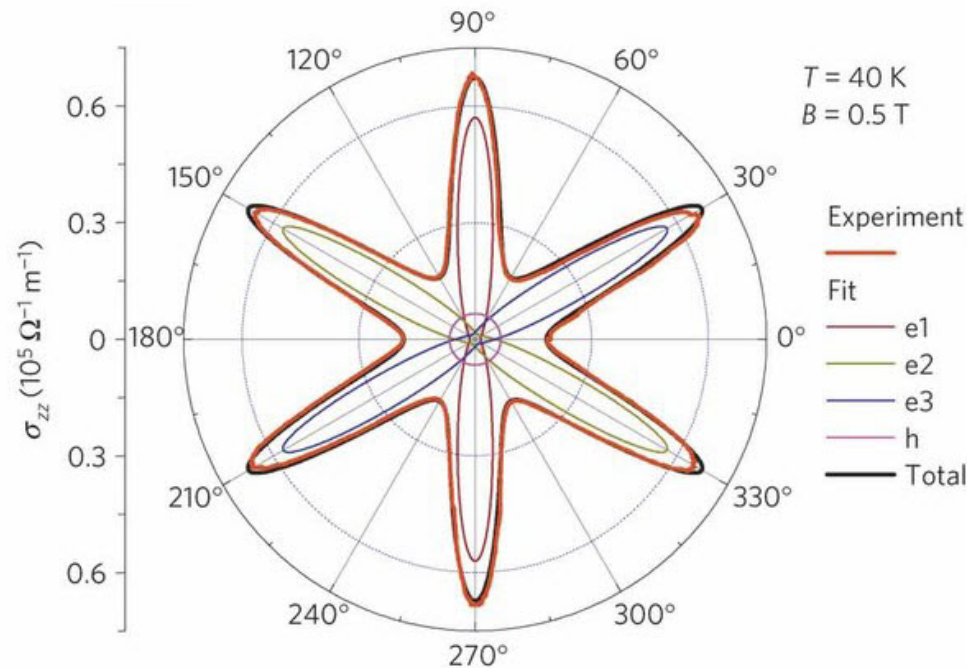
# Parametric Curves



The two parameters **a**, **n** are used to calculate the current radius **r** as **θ** sweeps the circle

# Parametric Curves

## Field Induced Polarization of Dirac Valleys in Bismuth\*

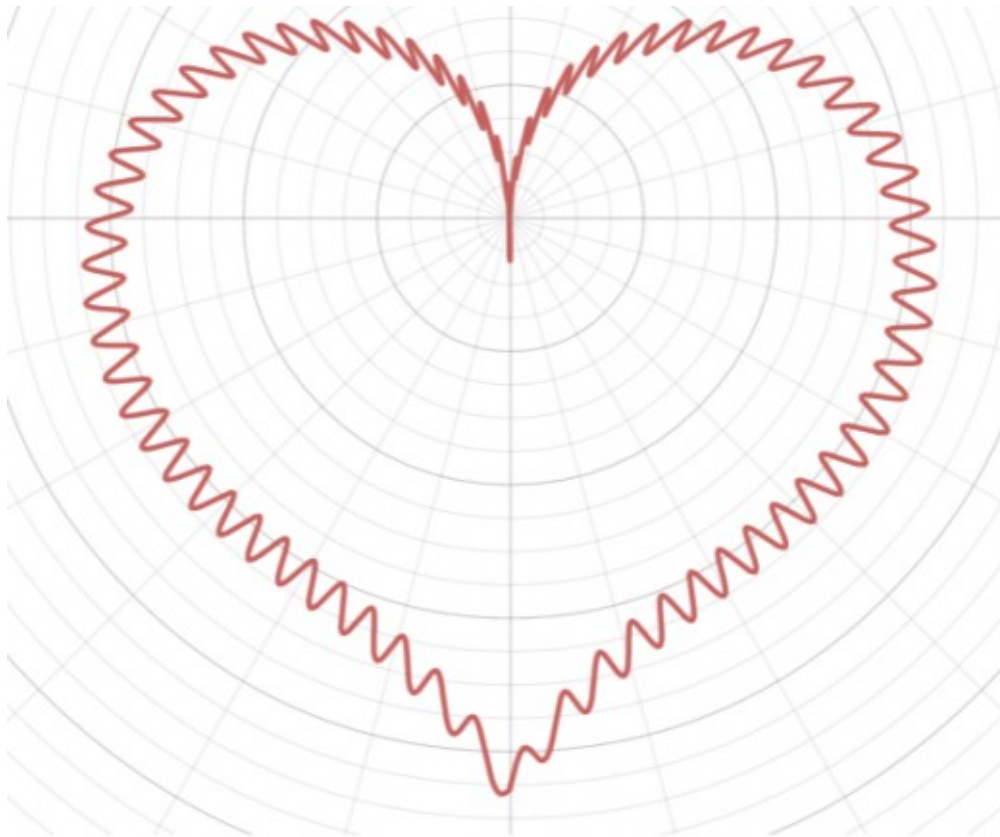


$$r = \sin^2\left(\frac{6}{5}\right) + \cos^2\left(\frac{6}{1}\right)$$

\*Bismuth is the element with the **highest** atomic mass that is **stable**



# Parametric Curves



$$r = 5 - \frac{2.5\sin(\theta)}{\sqrt{0.2 + |\cos(\theta)|}} + \sin(\theta) + \cos(2\theta) + 0.3\sin(70\theta)$$



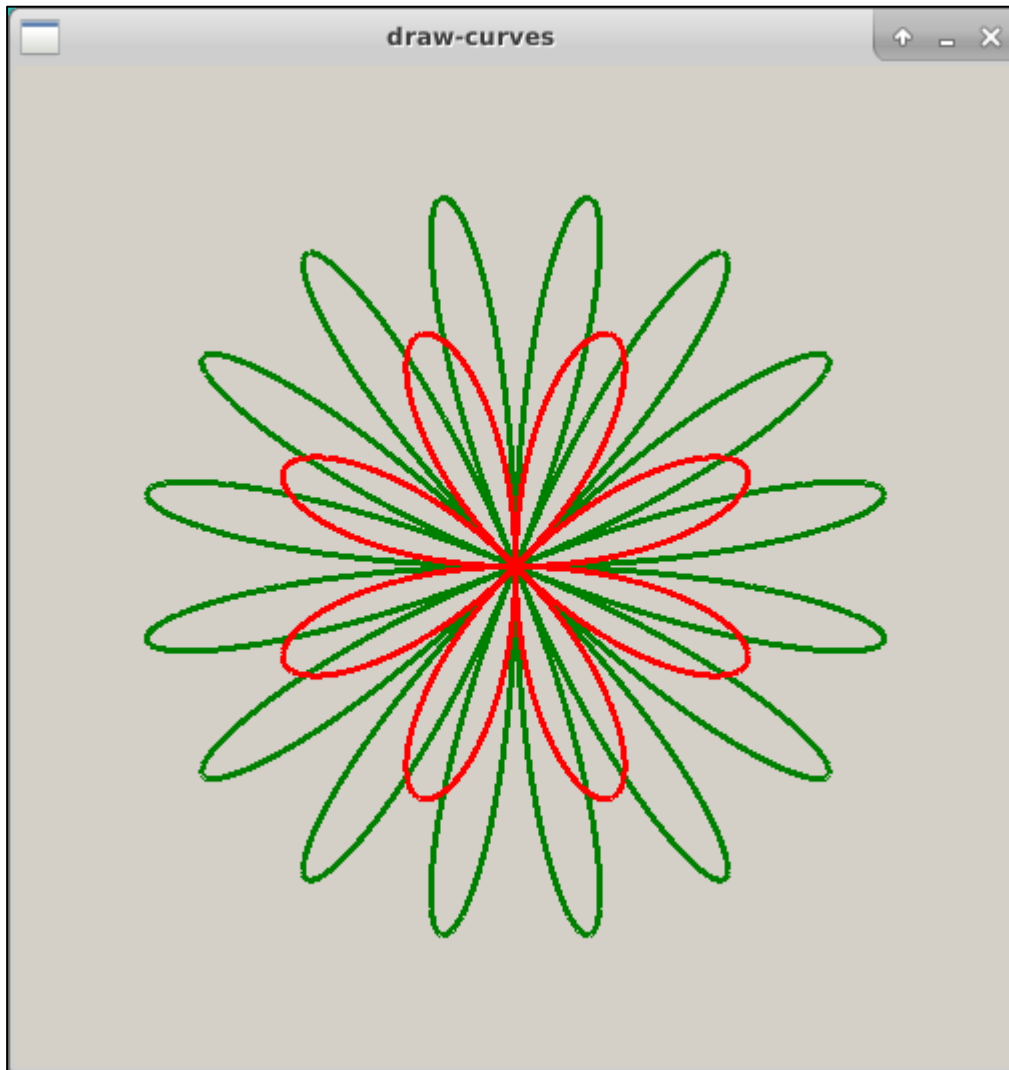
# Parametric Curves

```
void CalcPolarCurve(PointSet& ps,  
    double loops, double radius)  
{  
    int intervals{ 997 };  
    double deltaTheta{ 2.0 * M_PI / intervals };  
    for (int i{0}; i <= intervals; i++)  
    {  
        double theta{ i * deltaTheta };  
        double r{ radius * sin(loops * theta) };  
        double x{ r * cos(theta) };  
        // ...  
    }  
}
```

These two parameters are used to calculate the current radius  $r$  as  $\theta$  sweeps the circle

**Open and run Lab 5**

# Check Lab 5 - Parametric Curves



The key points are:

- The radius is changing while  $\theta$  is changing
- The shape of the curve depends on two *parameters*

## Now you know...

- **Allegro** is a free, open source, cross platform library for rendering 2D graphics
- A “bounding rectangle” is used to scale **world** (virtual) coordinates to **screen** (physical) coordinates
- How to create and populate elements of a **PointSet**
- How to “connect the dots” using **SimpleScreen.DrawLines()**
- How to use **polar coordinates** to draw 2D circles
- We approximate circles by drawing small **intervals**
- We can draw fancy curves by using **parametric equations**