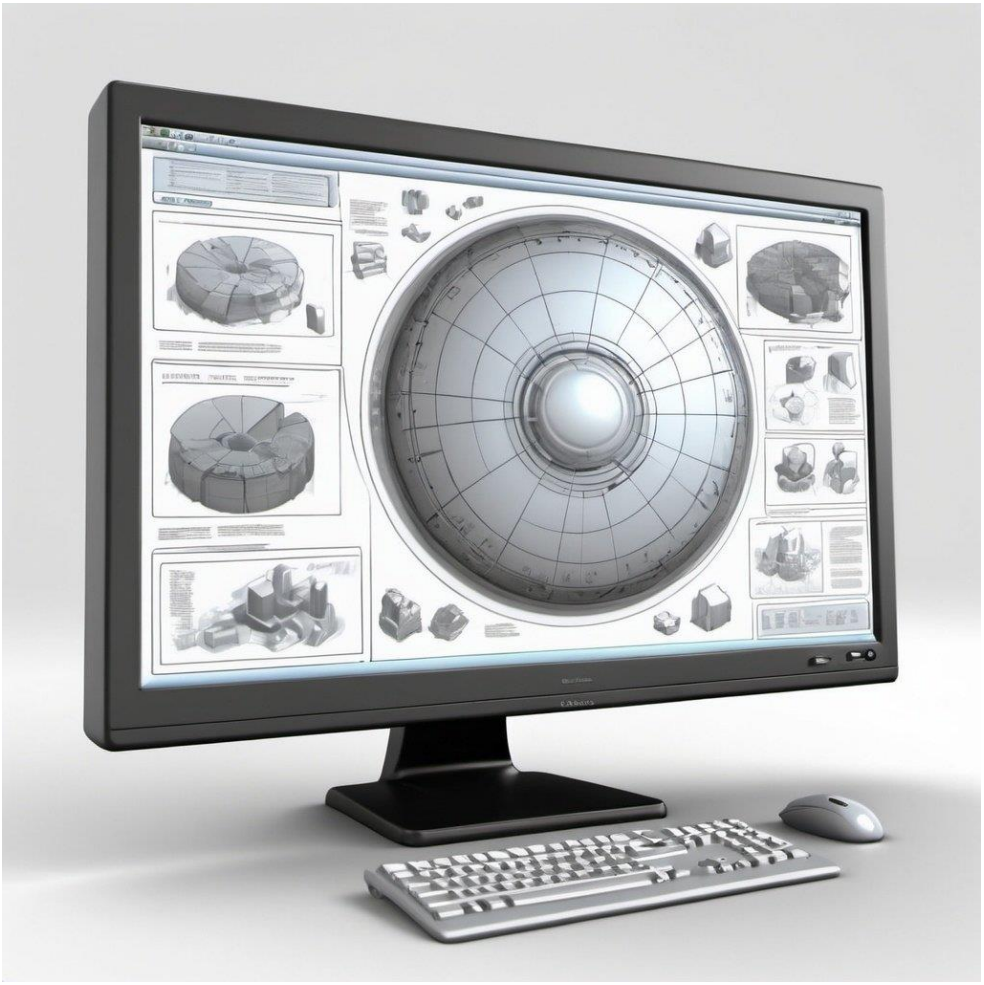
Chapter 8

X11 Windows and Graphics

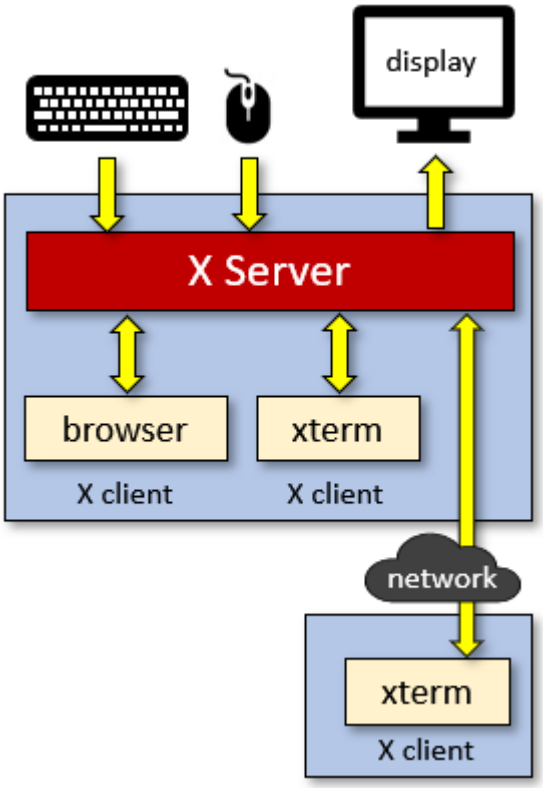
What is in This Chapter ?

This chapter discusses how to create **X11 Windows** in the Linux environment. These windows are basic, in that they allow drawing of graphics and event handling, but they do not contain window components (e.g., buttons, text boxes, lists, menus, scroll bars, etc..). We begin with a discussion of the basics of getting a window up and then we discuss how to draw **graphics** on the window with standard colors. There is a brief discussion of **event loops** and simple animation. Finally, we discuss how to do **event handling** for common events such as **window closing**, **mouse pointer entering/leaving** a window, **key presses/releases**, **mouse button presses/releases**, **mouse pointer motion** and **window resizing**. There is way too much information about the X11 window management system, event handling and graphics to cover in this course. Feel free to investigate further to do more fancy things.



8.1 X11 Windows (i.e., X Window System, version 11)

In Linux/Unix, if we want to create windows and do graphics, a very common framework for doing so is through the X Window System (also known as X11). It is a windowing system for bitmap displays. X11 provides basic GUI support by allowing the creation and manipulation of windows as well as interaction with the mouse and keyboard. It also allows basic graphics. The framework has a main X server that our programs can interact with via xterms, browsers, etc… even over a network.



The "look" and styling of the windows themselves may vary greatly from system to system as nobody has mandated the user interface. So, you cannot be sure what your application will look like as you port it to other Linux/Unix systems.

We will discuss just a few basics here of getting some windows opened and drawing on them. There is much information to learn, but we will not get into all the details. Keep in mind that this is a basic framework that does not have advanced features. Other libraries and frameworks can be used to enhance the GUI aspects of your applications in Linux such as **Motif** for windowing components and **OpenGL** for doing 3D graphics.

You will find that there are many parameters and options in the X window functions. It may be frustrating at times because some parameters may actually be ignored, depending on certain configurations.

Let's see how to create a simple window. To begin, we need to connect to the X server by using the **XOpenDisplay()** function which is defined in the **<X11/Xlib.h>** header file. The function takes a single parameter representing the display name (this is not the title of the window). We will leave it blank in our applications by using **NULL**. This function returns a **Display** structure which contains necessary information about the X server in order to communicate with it. If it cannot connect, then **NULL** is returned:

```
Display *display;

// Opens a connection to the X server
display = XOpenDisplay(NULL);
if (display == NULL) {
  printf("Error: Unable to open connection to XServer\n");
  exit(-1);
}
```

This code does not create a window, it just connects to the X server. To create a window, we can use the **XCreateSimpleWindow()** function which will return a Window structure. It has this format:

```
XCreateSimpleWindow(<display>, <parent>, <x>, <y>, <width>, <height>,
                    <border_width>, <border_color>, <background_color>)
```

Here, the display is the value returned from **`XOpenDisplay()`** and <parent> is the parent window (which can be used when one window opens another). In our simple examples we will use just one window, so we will set the parent to the root window of the system. We can do this by using **`RootWindow(<display>, 0)`**.

The <x>, <y>, <width>, <height> values specify the position of the top left corner of the window (with respect to its parent) as well as the width and height of the window in pixels. Since we won't have a parent window, we'll set the <x> and <y> to be **0**. The <border_width> specifies the width (in pixels) of the window's borders. However, the window will take on this value from its parent window, so it will be ignored in our examples. Finally, the <border_color> and <background_color> allows us to specify the border and background color of the window. Again, the <border_color> will be largely ignored as it is defined by the parent window by default. These values are **`unsigned long`** integers that represent a color. There is a file with some standard X11 colors defined which is called **rgb.txt** and is located in our system at **/usr/share/X11/rgb.txt**. You can go here as well to find out more:

https://en.wikipedia.org/wiki/X11_color_names

The color names are not defined anywhere, but you can obtain the RGB values and use that:

FFFFFF	000000	333333	666666	999999	CCCCCC	CCCC99	9999CC	666699
660000	663300	996633	003300	003333	003399	000066	330066	660066
990000	993300	CC9900	006600	336666	0033FF	000099	660099	990066
CC0000	CC3300	FFCC00	009900	006666	0066FF	0000CC	663399	CC0099
FF0000	FF3300	FFFF00	00CC00	009999	0099FF	0000FF	9900CC	FF0099
CC3333	FF6600	FFFF33	00FF00	00CCCC	00CCFF	3366FF	9933FF	FF00FF
FF6666	FF6633	FFFF66	66FF66	66CCCC	00FFFF	3399FF	9966FF	FF66FF
FF9999	FF9966	FFFF99	99FF99	66FFCC	99FFFF	66CCFF	9999FF	FF99FF
FFCCCC	FFCC99	FFFFCC	CCFFCC	99FFCC	CCFFFF	99CCFF	CCCCFF	FFCCFF

The **XCreateSimpleWindow**() function returns a **Window** structure. Once we have that, we need to make the window visible. To do this, we need to map it to the display by using **`XMapWindow`**`(<display>, <window>)`.

To ensure it has been displayed, we can use the **XFlush**(<display>) function. Once we are all done, we can unmap the window, destroy it and then close the display.

Here is an example of opening a window, which will then wait for the user to press the ENTER key on the keyboard and then close the window. Be aware though, when we run the program, the keyboard focus moves to the opened window. To close the window, we would have to go back to our terminal window to press the ENTER key.

Code from **basicWindow.c**

```c
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>

int main() {
  Display *display;
  Window   win;

  // Opens connection to X server
  display = XOpenDisplay(NULL);

  // Create a simple window
  win = XCreateSimpleWindow(display,    // our connection to server
              RootWindow(display, 0),// parent window (none in this example)
              0, 0,                  // x,y (w.r.t. parent ... ignored here)
              300,150,               // width, height
              0,                     // border width
              0x000000,              // border color (ignored in this example)
              0xFFDD00);             // background color (mustard yellow)

  // Set the title of the window
  XStoreName(display, win, "My First X Window");

  // Make it visible
  XMapWindow(display, win);
  XFlush(display);

  // Wait until user presses a key on keyboard
  getchar();

  // Clean up and close the window
  XUnmapWindow(display, win);
  XDestroyWindow(display, win);
  XCloseDisplay(display);
}
```

To compile this code, we need to include the X11 library:

```
student@COMPBase:~$ gcc -o basicWindow basicWindow.c -lX11
student@COMPBase:~$ ./basicWindow
student@COMPBase:~$
```

8.2 X11 Graphics

Now that we know how to create a window, we'd like to do something useful. Let's see how to draw something on the window. In order to draw something, we need to create a ***graphics context***. This is like getting a "pen" that we can start using to draw with. We do this by using the **XCreateGC()** function which has this format:

```
XCreateGC(<display>, <win>, <value_mask>, <values>)
```

We will not get into the details of <value_mask> and <values> in this course. Instead, we'll set them to **0** and NULL. The function returns a **GC** structure that represents the graphics context. Once we have this graphics context, there are many functions that we could use to start drawing:

- **XDrawPoint**(<display>, <window>, <gc>, <x>, <y>)
- **XDrawPoints**(<display>, <window>, <gc>, <points>, <num_points>, <mode>)

- **XDrawLine**(<display>, <window>, <gc>, <x1>, <y1>, <x2>, <y2>)
- **XDrawLines**(<display>, <window>, <gc>, <points>, <num_points>, <mode>)
- **XDrawSegments**(<display>, <window>, <gc>, <segments>, <num_segments>)

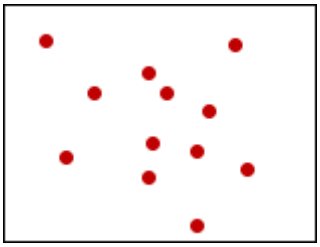
- **XDrawRectangle**(<display>, <window>, <gc>, <x>, <y>, <width>, <height>)
- **XFillRectangle**(<display>, <window>, <gc>, <x>, <y>, <width>, <height>)
- **XDrawRectangles**(<display>, <window>, <gc>, <rectangles>, <num_rectangles>)
- **XFillRectangles**(<display>, <window>, <gc>, <rectangles>, <num_rectangles>)

- **XDrawArc**(<display>, <window>, <gc>, <x>, <y>, <width>, <height>, <angle1>, <angle2>)
- **XFillArc**(<display>, <window>, <gc>, <x>, <y>, <width>, <height>, <angle1>, <angle2>)
- **XDrawArcs**(<display>, <window>, <gc>, <arcs>, <num_arcs>)
- **XFillArcs**(<display>, <window>, <gc>, <arcs>, <num_arcs>)

- **XFillPolygon**(<display>, <window>, <gc>, <points>, <num_points>, <shape>, <mode>)

- **XDrawString**(<display>, <window>, <gc>, <x>, <y>, <string>, <length>)

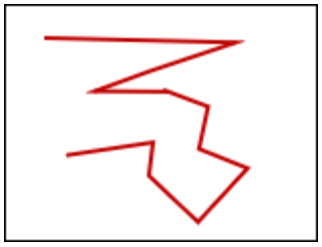
Each function makes use of the <display>, <window> and <gc> that have described earlier. The **XDrawPoint()** and **XDrawLine()** functions should be straight forward. For the **XDrawPoints()** and **XDrawLines()** functions, the <points> parameter is a pointer to a bunch of XPoint objects that must already exist in memory. The XPoint structure is defined as follows:



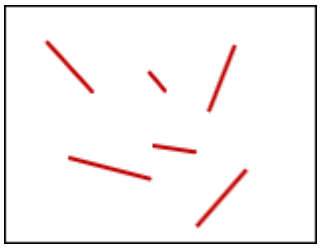
```
typedef struct {
    short x, y;
} XPoint;
```

The <mode> parameter can be either CoordModeOrigin (in which case all points are relative to the origin) or CoordModePrevious (in which all point coordinates are relative to the previous point (except for the first point)).

The **XDrawLines()** function draws lines to connect successive points, in a "connect the dot" fashion, where each point (except the first) is connected to the previous one.



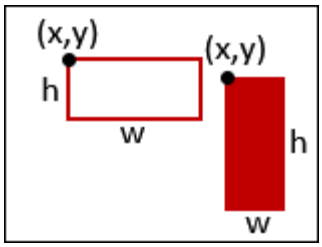
The **XDrawSegments()** function takes a bunch of XSegment structures which have this format:



```
typedef struct {
    short x1, y1, x2, y2;
} XSegment;
```

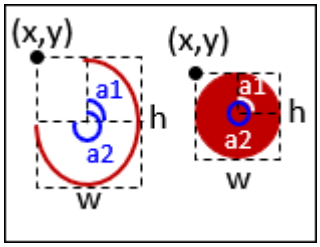
It then draws each segment one at a time. Hence the segments are assumed to be unrelated and disconnected.

The **XDrawRectangle()** and **XFillRectangle()** functions both take <x> and <y> which define the upper-left corner of the rectangle. The **XDrawRectangles()** and **XFillRectangles()** both take a bunch of XRectangle structures which have this format:




```
typedef struct {
    short           x, y;
    unsigned short width, height;
} XRectangle;
```

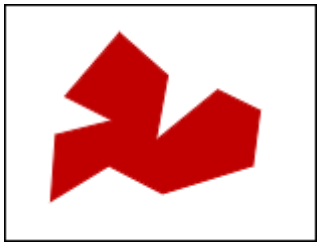
The **XDrawArc()**, **XFillArc()**, **XDrawArcs()** and **XFillArcs()** functions take the <x>, <y>, <width> and <height> that define the bounding box of the rectangle that the arc will be drawn within. angle1 represents the start of the arc relative to the 3 o'clock position from the center, in units of degrees*64. angle2 represents the path and extent of the arc relative to the start. These are defined using the following structure:



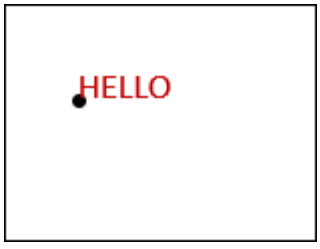

```
typedef struct {
    short                x, y;
    unsigned short       width, height;
    short                angle1, angle2;      /* Degrees * 64 */
} XArc;
```

You can use these functions to draw circles and ovals or opened arcs.

The **XFillPolygon()** function takes a bunch of **XPoint** structs just as with the **XDrawLines()** function. The <mode> parameter works the same way as before. The <shape> parameter is either Complex, Convex or Nonconvex which is used to improve performance. There is no **XDrawPolygon()** function because it is the same as **XDrawLines()**, as long as the last point in the list is the same as the first point.



The **XDrawString()** function takes <x> and <y> which define the bottom-left corner of the first character. The string is supplied, as well as the number of characters in the string.



Just one more thing to mention… to set the color for drawing, we need to use the **XSetForeground()** function which has this format:

```
XSetForeground(<display>, <gc>, <color>)
```

Recall that the `<color>` here is in the format 0xRRGGBB where RR, GG and BB are the amounts of red, green and blue in the color, respectively. These are each hex values from 00 to FF.

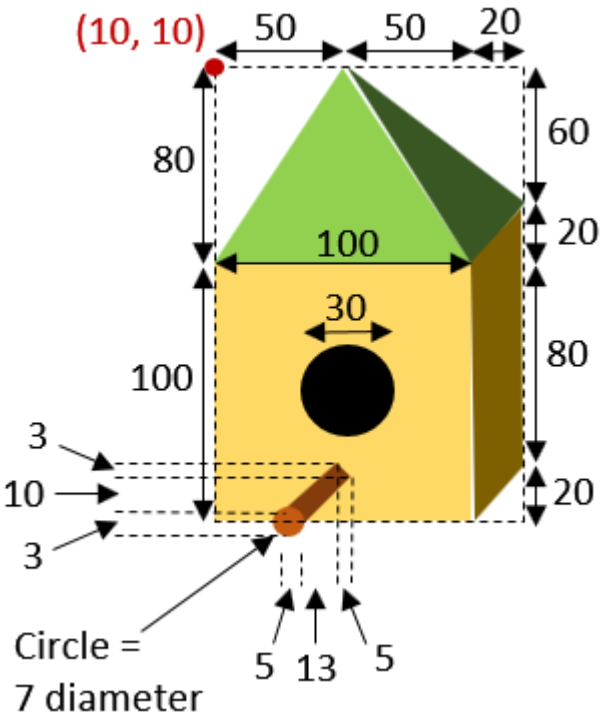
Example:

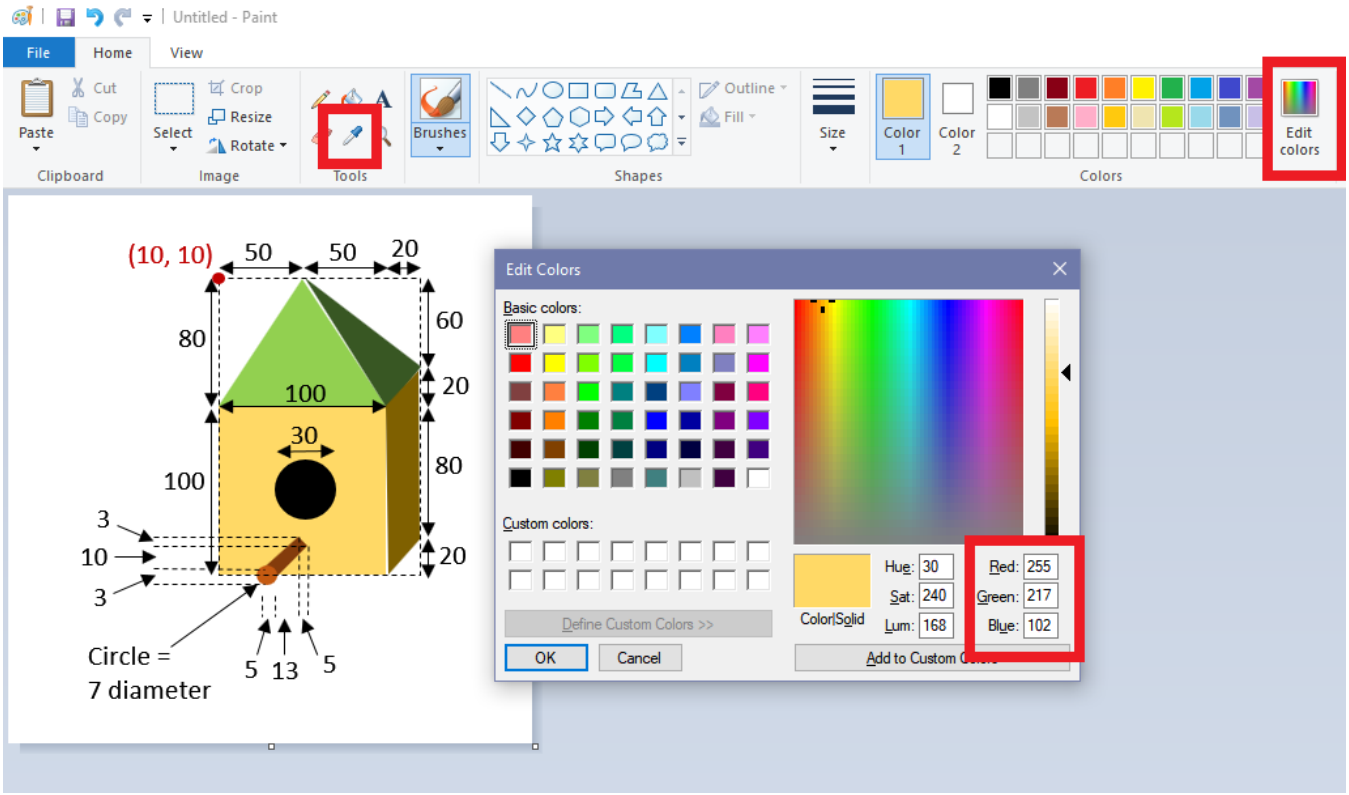
As an example, let's draw the following bird house onto a window →

How do we determine the color values?
In my case, I took the electronic image/drawing, pasted it into MSPaint and the used the color selector to determine the RGB values. I then calculated the hex values.

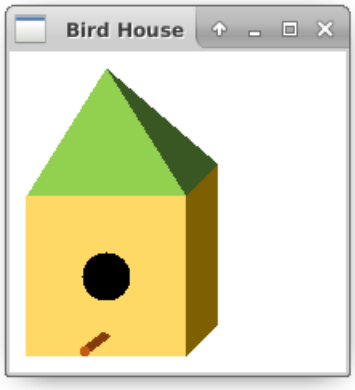
This is shown in the snapshot below.

The following page shows the code that draws this bird house.





Code from **birdHouse.c**




```c
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>
#include <unistd.h>

int main() {
  Display *display;
  Window   win;
  GC       gc;
  XPoint   polygon[4];

  // Opens connection to X server
  display = XOpenDisplay(NULL);

  // Create a simple window
  win = XCreateSimpleWindow(display, RootWindow(display, 0), 0, 0,
                            210,200, 0, 0x000000, 0xFFFFFF);

  // Set the name of the window
  XStoreName(display, win, "Bird House");

  // Get the graphics context
  gc = XCreateGC(display, win, 0, NULL);

  // Make it visible
  XMapWindow(display, win);
  XFlush(display);
  usleep(20000);  // sleep for 20 milliseconds.

  // Draw the main box
  polygon[0].x = 110;
  polygon[0].y = 90;
  polygon[1].x = 130;
  polygon[1].y = 70;
  polygon[2].x = 130;
  polygon[2].y = 170;
  polygon[3].x = 110;
  polygon[3].y = 190;
  XSetForeground(display, gc, 0x7F6000);
  XFillPolygon(display, win, gc, polygon, 4, Convex, CoordModeOrigin);
  XFlush(display);

  XSetForeground(display, gc, 0xFFD966);
  XFillRectangle(display, win, gc, 10, 90, 100, 100);
  XFlush(display);

  // Draw the roof
  XSetForeground(display, gc, 0x92D050);
  polygon[0].x = 10;
  polygon[0].y = 90;
  polygon[1].x = 60;
  polygon[1].y = 10;
  polygon[2].x = 110;
  polygon[2].y = 90;
  XFillPolygon(display, win, gc, polygon, 3, Convex, CoordModeOrigin);
  XFlush(display);
```

Window size = 210 x 200 with white background.

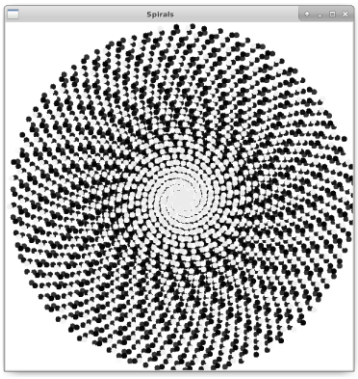
Need to sleep a bit until window is ready for drawing, otherwise image won't appear.

```
  XSetForeground(display, gc, 0x385723);
  polygon[0].x = 60;
  polygon[0].y = 10;
  polygon[1].x = 130;
  polygon[1].y = 70;
  polygon[2].x = 110;
  polygon[2].y = 90;
  XFillPolygon(display, win, gc, polygon, 3, Convex, CoordModeOrigin);
  XFlush(display);

  // Draw the hole
  XSetForeground(display, gc, 0x000000);
  XFillArc(display, win, gc, 45, 125, 30, 30, 0, 360*64);
  XFlush(display);

  // Draw the perch
  XSetForeground(display, gc, 0x843C0C);
  polygon[0].x = 58;
  polygon[0].y = 174;
  polygon[1].x = 63;
  polygon[1].y = 177;
  polygon[2].x = 50;
  polygon[2].y = 187;
  polygon[3].x = 45;
  polygon[3].y = 184;
  XFillPolygon(display, win, gc, polygon, 4, Convex, CoordModeOrigin);
  XFlush(display);

  XSetForeground(display, gc, 0xC55A11);
  XFillArc(display, win, gc, 43, 183, 7, 7, 0, 360*64);
  XFlush(display);

  // Wait until the user presses a key on the keyboard
  getchar();

  // Clean up and close the window
  XFreeGC(display, gc);
  XUnmapWindow(display, win);
  XDestroyWindow(display, win);
  XCloseDisplay(display);
}
```

Function requires us to multiply the degree value by 64.

Example:

Here is a more computational example that displays a spiraling sequence of circles:



Running this program could cause seizures because it contains flashing & spiraling animation.

Code from **spirals.c**

```c
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>
#include <unistd.h>
#include <math.h>

#define WIN_SIZE   600

int main() {
  Display *display;
  Window   win;
  GC       gc;

  int           radius = 0;
  double        angle = 0;
  int           grayLevel = 0;
  unsigned int  color;

  // Opens connection to X server
  display = XOpenDisplay(NULL);

  // Create a simple window, set the title and get the graphics context then
  // make is visible and get ready to draw
  win = XCreateSimpleWindow(display, RootWindow(display, 0), 0, 0,
                            WIN_SIZE, WIN_SIZE, 0, 0x000000, 0xFFFFFF);
  XStoreName(display, win, "Spirals");
  gc = XCreateGC(display, win, 0, NULL);

  // Make it visible
  XMapWindow(display, win);
  XFlush(display);
  usleep(20000);  // sleep for 20 milliseconds.

  // Go into infinite loop
  while(1) {
    color = (255-grayLevel)*65536 + (255-grayLevel)*256 + (255-grayLevel);
    XSetForeground(display, gc, color);
    XFillArc(display, win, gc, (int)(cos(angle*M_PI/180)*radius)+WIN_SIZE/2,
                               (int)(sin(angle*M_PI/180)*radius)+WIN_SIZE/2,
                               10, 10, 0, 360*64);
    angle += 137.51;
    if (angle > 360) angle -= 360;
    radius = (radius + 1)%300;
    if (radius == 0)
      grayLevel = (grayLevel + 5) % 255;
    XFlush(display);
    usleep(500);
  }

  // Clean up and close the window
  XFreeGC(display, gc);
  XUnmapWindow(display, win);
  XDestroyWindow(display, win);
  XCloseDisplay(display);
}
```

We will need to compile with **-lm** so that we link to the math library.

There are three variables declared at the top of the program. The **radius** represents the distance (from the center) that we are drawing the circles at, while the **angle** represents the angle that we are drawing them at. The **grayLevel** indicates the color that the spirals are drawn at, which begins with white and gets darker for each round of spirals.

You may notice that the **while** loop first draws the circle (with appropriate gray-level fill) and then simply updates the **angle** and **radius** for the next time through the loop. The **%** operator is the ***modulus operator*** that gives the remainer after dividing by a specified number. The modulus operator is great for ensuring that an integer does not exceed a certain value but that it wraps arpund to **0** again. The following two pieces of code do the same thing in our example:

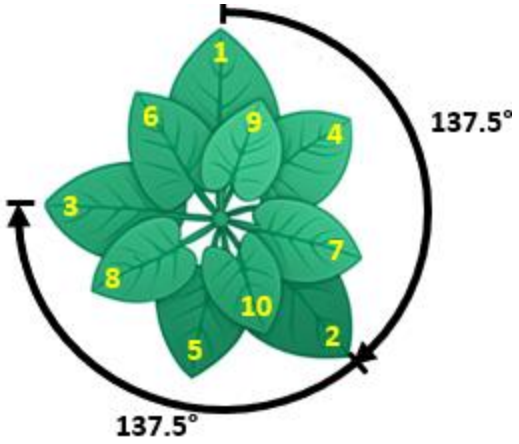
```
radius = (radius + 1) % 255;
```

```
radius = radius + 1;
if (radius == 255)
   radius = 0;
```

For example, if **x** was initially **0** and then we did **x = (x + 1)%5**, here would be the values for **x**:

0,1,2,3,4,0,1,2,3,4,0,1,2,3,4,0,1,2,3,4,0,1,2 … etc..

The angle of **137.51** is called the ***golden angle*** as it is found in nature as the ideal angle for producing spirals as shown in the design of seashells, flower petals, etc.. The angle is ideal as it minimizes overlap during multiple rounds of spiraling.



While this example illustrates the ability to vary the computational parameters during the processing loop of the simulation, it really does not serve any particular purpose other than to produce a nice picture.

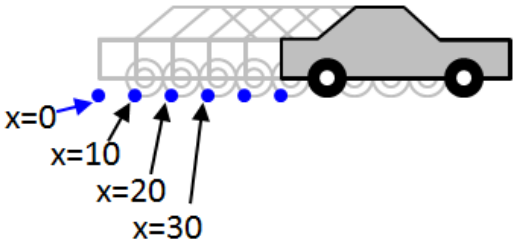
8.3 Simple Animation

For visually-appealing simulations, it is often necessary to show one or more objects moving on the screen. This is certainly the case in the area of game programming. We will discuss here some code for doing very simple motion.

Example:

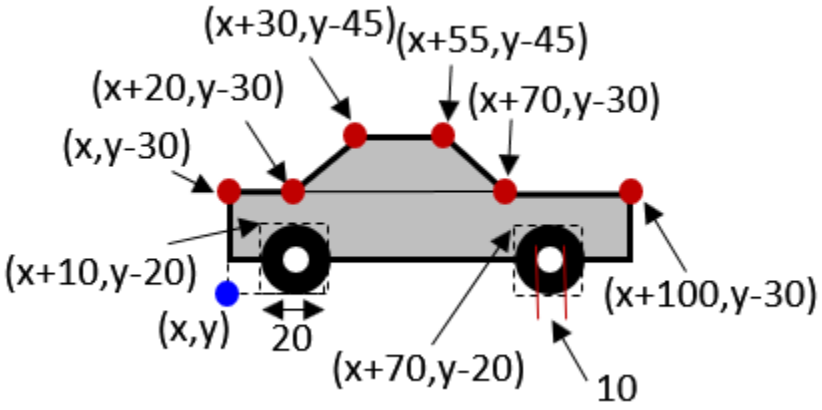
Consider a basic algorithm for moving a car horizontally across the window:

```
for successive x locations from 0 to WIN_SIZE {
        draw the car at position x
        x ← x + 10
}
```



To be able to do this, we should create a function that draws a car.

Here are the dimensions:



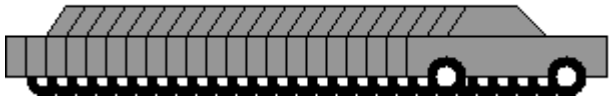
The code is straight forward since it follows from our knowledge of basic drawing functions:

```
void drawCar(int x, int y) {
  XPoint   polygon[4];

  // Draw the body
  XSetForeground(display, gc, 0x969696);
  XFillRectangle(display, win, gc, x, y-30, 100, 20);
  polygon[0].x = x+20;
  polygon[0].y = y-30;
  polygon[1].x = x+30;
  polygon[1].y = y-45;
  polygon[2].x = x+55;
  polygon[2].y = y-45;
  polygon[3].x = x+70;
  polygon[3].y = y-30;
  XFillPolygon(display, win, gc, polygon, 4, Convex, CoordModeOrigin);

  XSetForeground(display, gc, 0x000000);
  XDrawRectangle(display, win, gc, x, y-30, 100, 20);
  polygon[0].x = x+20;
  polygon[0].y = y-30;
  polygon[1].x = x+30;
  polygon[1].y = y-45;
  polygon[2].x = x+55;
  polygon[2].y = y-45;
  polygon[3].x = x+70;
  polygon[3].y = y-30;
  XDrawLines(display, win, gc, polygon, 4, CoordModeOrigin);

  // Draw the wheels
  XSetForeground(display, gc, 0x000000); // black
  XFillArc(display, win, gc, x+10, y-20, 20, 20, 0, 360*64);
  XFillArc(display, win, gc, x+70, y-20, 20, 20, 0, 360*64);

  XSetForeground(display, gc, 0xFFFFFF); // white
  XFillArc(display, win, gc, x+15, y-15, 10, 10, 0, 360*64);
  XFillArc(display, win, gc, x+75, y-15, 10, 10, 0, 360*64);
}
```

To draw this car going across the screen, we could use a simple **while** loop as before:

```
while(1) {
  drawCar(x, y);
  x += 10;
  XFlush(display);
  usleep(100000);
}
```

With this code, the car would simply be re-drawn with the old car position not being erased:



We could fix this by drawing a white rectangle beforehand … inserting the following code before the `drawCar(x, y);` function call:

```
XSetForeground(display, gc, 0xFFFFFF);
XFillRectangle(display, win, gc, 0, 0, WIN_SIZE, WIN_SIZE/2);
```

The code above, however, does not stop the car at the edge of the screen. In order to stop the car, we need to stop changing the **x** value so that the car is redrawn at the same spot once we reach the edge of the window. We will just need check in the **while** loop to determine whether we have reached the end and only update when we are not there yet:

```
while(1) {
  XSetForeground(display, gc, 0xFFFFFF);
  XFillRectangle(display, win, gc, 0, 0, WIN_SIZE, WIN_SIZE/2);

  drawCar(x, y);
  if (x+110 < WIN_SIZE)
    x += 10;
  XFlush(display);
  usleep(100000);
}
```

Notice that we check for the position of the front bumper of the car (i.e**., x +** 100), not the back bumper (i.e., **x**).

The above code shows our car moving rather quickly across the screen. If we adjusted the increment from **10** to a smaller value, the car would move much slower. Consider an algorithm for accelerating the car until it reaches the middle of the window and then decelerating until it reached the right side of the window again using this pseudocode:

```
speed ← 0
for x locations from 0 to WIN_SIZE by speed {
        draw the car at position x
        if ((x+50) < (WIN_SIZE /2)      // x + 50 is the middle of the car
              speed ← speed + 0.10
        otherwise
              speed ← speed – 0.10
}
```

We can adjust our code to do this by adding a `float speed` variable, initially set to **0**. Here is the final code:

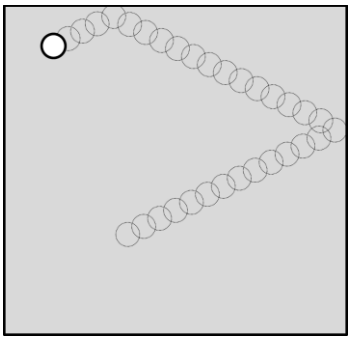
Code from **carAnimation.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>
#include <unistd.h>

#define WIN_SIZE   600

// global variables
Display *display;
Window   win;
GC       gc;

void drawCar(int x, int y) {
  XPoint   polygon[4];

  // Draw the body
  XSetForeground(display, gc, 0x969696);
  XFillRectangle(display, win, gc, x, y-30, 100, 20);
  polygon[0].x = x+20;
  polygon[0].y = y-30;
  polygon[1].x = x+30;
  polygon[1].y = y-45;
  polygon[2].x = x+55;
  polygon[2].y = y-45;
  polygon[3].x = x+70;
  polygon[3].y = y-30;
  XFillPolygon(display, win, gc, polygon, 4, Convex, CoordModeOrigin);

  XSetForeground(display, gc, 0x000000);
  XDrawRectangle(display, win, gc, x, y-30, 100, 20);
  polygon[0].x = x+20;
  polygon[0].y = y-30;
  polygon[1].x = x+30;
  polygon[1].y = y-45;
  polygon[2].x = x+55;
  polygon[2].y = y-45;
  polygon[3].x = x+70;
  polygon[3].y = y-30;
  XDrawLines(display, win, gc, polygon, 4, CoordModeOrigin);

  // Draw the wheels
  XSetForeground(display, gc, 0x000000); // black
  XFillArc(display, win, gc, x+10, y-20, 20, 20, 0, 360*64);
  XFillArc(display, win, gc, x+70, y-20, 20, 20, 0, 360*64);

  XSetForeground(display, gc, 0xFFFFFF); // white
  XFillArc(display, win, gc, x+15, y-15, 10, 10, 0, 360*64);
  XFillArc(display, win, gc, x+75, y-15, 10, 10, 0, 360*64);
}

int main() {
  int      x = 0, y = 300;
  float    speed = 0;
```

```
  // Opens connection to X server
  display = XOpenDisplay(NULL);

  // Create a simple window, set the title and get the graphics context then
  // make is visible and get ready to draw
  win = XCreateSimpleWindow(display, RootWindow(display, 0), 0, 0,
                            WIN_SIZE, WIN_SIZE/2, 0, 0x000000, 0xFFFFFF);
  XStoreName(display, win, "Car Animation");
  gc = XCreateGC(display, win, 0, NULL);
  XMapWindow(display, win);
  XFlush(display);
  usleep(20000);  // sleep for 20 milliseconds.

  // Go into infinite loop
  while(1) {
    XSetForeground(display, gc, 0xFFFFFF);
    XFillRectangle(display, win, gc, 0, 0, WIN_SIZE, WIN_SIZE/2);

    drawCar(x, y);

    x = (int)(x + speed);
    if (x+100 > WIN_SIZE)
      break;
    if (x+50 < WIN_SIZE/2)
      speed += 0.10;
    else
      speed -= 0.10;
    XFlush(display);
    usleep(100000);
  }
  getchar();

  // Clean up and close the window
  XFreeGC(display, gc);
  XUnmapWindow(display, win);
  XDestroyWindow(display, win);
  XCloseDisplay(display);
}
```

Example:

Now what about 2-dimensional motion? How could we get a ball to bounce around the window so that it remains within the window borders? To do this, we must understand the computational model.



To keep things simpler, let's assume that the ball is moving at a constant speed at all times. As the ball moves, we know that both its **x** and **y** locations will change. Also, the direction that the ball is facing should change. But when does the ball's direction change? We will assume that it only changes direction when it hits the window borders.

So, we will need to keep track of the **ball's (x,y)** *location* as well as the *direction* (i.e., **angle**).

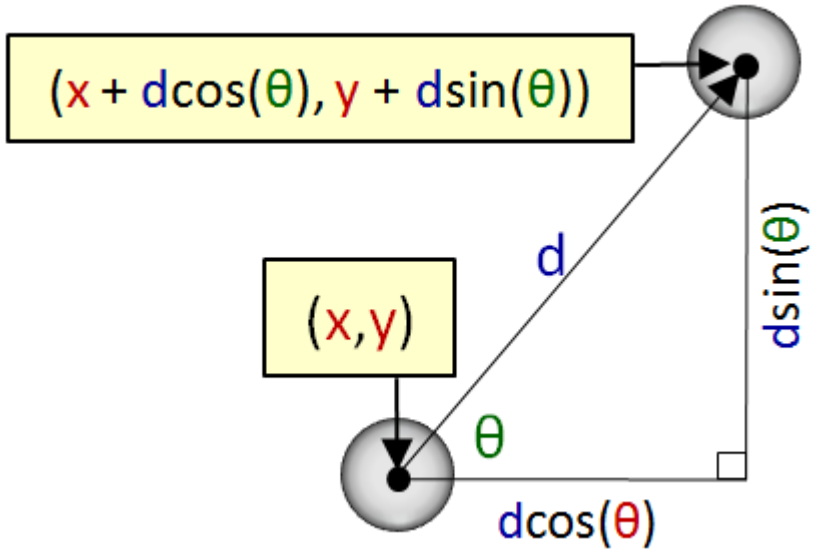
As with our moving car, we simply need to keep updating the ball's location and check to see whether or not it reaches the window borders. Here is the basic idea:

```
(x, y) ← center of the window
direction ← a random angle from 0 to 2π
repeat {
      draw the ball at position (x, y)
      move ball forward in its current direction
      if ((x, y) is beyond the window border) then
          change the direction accordingly
}
```

It seems fairly straight forward, but two questions arise:

1) How do we "*move the ball forward in its current direction*"?
2) How do we "*change the direction accordingly*"?

The first is relatively simple, since it is just based on trigonometry. Given that the ball at location **(x,y)** travels distance **d** in direction **θ**, the ball moves an amount of **d•*cos*(θ)** horizontally and **d•*sin*(θ)** vertically as shown in the diagram. So, to get the new location, we simply add the horizontal component to **x** and the vertical component to **y** to make this point: **(x + d*cos*(θ) , y + d*sin*(θ))**.

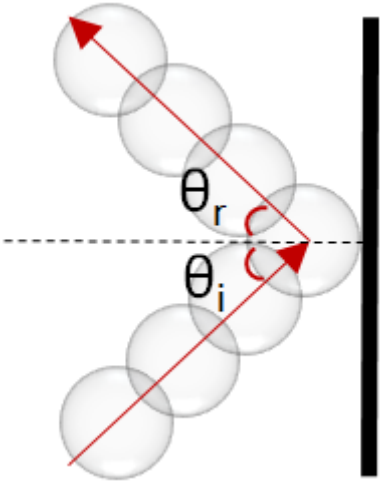


Line 5 in the above algorithm therefore can be replaced by this more specific code (assuming that the ball moves at a speed of **10** pixels per iteration):

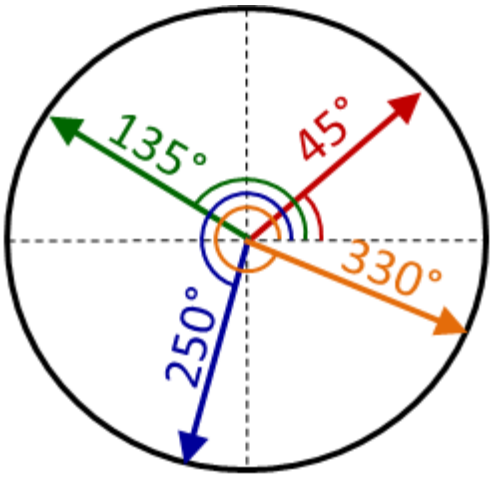
```
x ← x + 10 * cos(direction)
y ← y + 10 * sin(direction)
```

Now what about changing the direction when the ball encounters a window "wall"? Well, we would probably like to simulate a realistic collision. To do this, we must understand what happens to a real ball when it hits a wall.

You may recall the ***law of reflection*** from science/physics class. It is often used to explain how light reflects off of a mirror. The law states that the ***angle of reflection*** is the same as the ***angle of incidence***, under ideal conditions. That is, the angle at which the ball bounces off the wall (i.e., θ_r in the diagram), will be the same as the angle at which it hit the wall (i.e., θ_i in the diagram).



However, where do we get the angle of incidence from? Well, we have the direction of the ball stored in our **direction** variable. This direction will always be an angle from **0** to **360°** (or from **0** to **2π** radians).

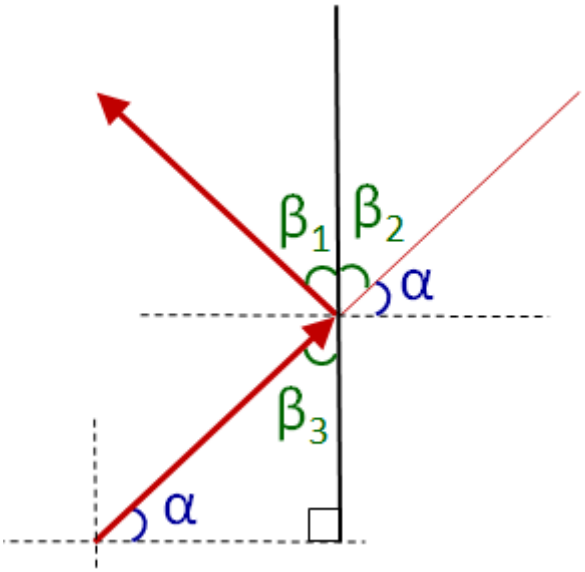


So, our ball's direction (called **α** for the purpose of this discussion) is always defined with respect to **0°** being the horizontal vector facing to the right. **360°** is the same as **0°**. As the direction changes counter-clockwise, the angle will increase. If the direction changes clockwise, the angle decreases. It is also possible that an angle can become negative. This is ok, since **330°** is the same as **-30°**.

Now, if you think back to the various angle theorems that you encountered in your math courses, you may remember these two:

1) the opposite angles of two straight crossing lines are equal

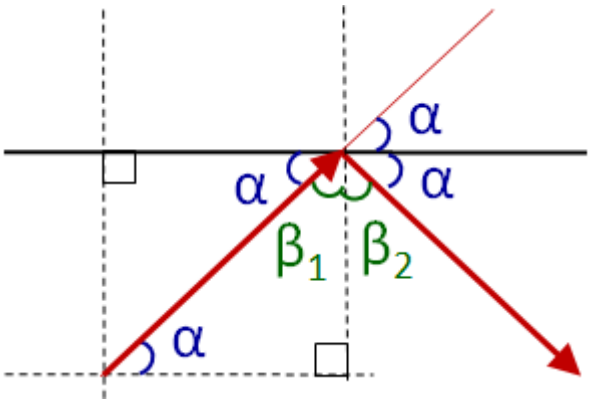
2) the interior angles of a triangle add up to **180°**



So, in the diagram on the right, for example, the 1st theorem above tells us that opposite angles **β2** and **β3** are equal. From the law of reflection, we also know that **β1** and **β3** are equal. Finally, **α** and **β3** add up to **90°**.
What does all this mean? Well, since **α** is the ball's direction, then to reflect off the wall, we simply need to add **β1** and **β2** to rotate the direction counter-clockwise. And since **β1**, **β2** and **β3** are all equal … and equal to **90° - α**, then to have the ball reflect we just need to do this:

direction = **direction** + (β_1 + β_2)
= **direction** + (90° - α + 90° - α)
= **direction** + (180° - 2 x **direction**)
= 180° - **direction**

The vertical bounce reflection is similar.
In the diagram here, it is easy to see that **β1** = **90° - α**. To adjust for the collision on the top of the window, we simply need to subtract **2α** from the direction:



direction = **direction** - 2 x **direction**)
= **- direction**

To summarize then, when the ball reaches the left or right boundaries of the window, we negate the direction and add **180°**, but when it reaches the top or bottom boundaries, we just negate the direction. Here is how we do it:

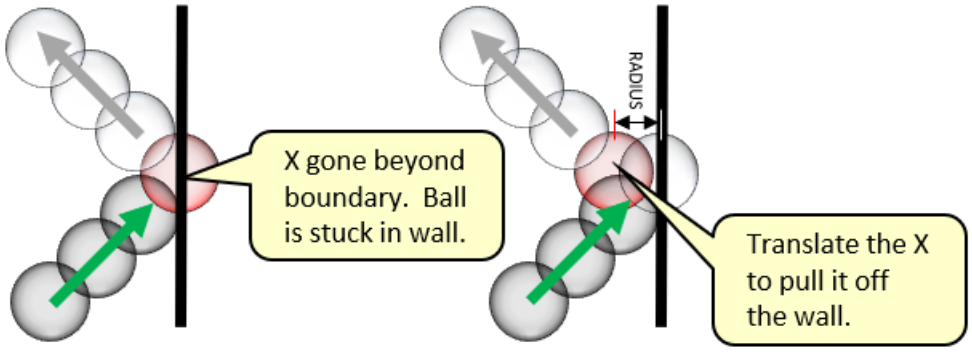
```
x ← windowWidth/2
y ← windowHeight/2
direction ← a random angle from 0 to 2π
repeat {
        draw the ball at position (x, y)
        x ← x + 10 * cos(direction)
        y ← y + 10 * sin(direction)
        if ((x >= windowWidth) OR (x <= 0)) then
            direction = 180° - direction
        if ((y >= windowHeight) OR (y <= 0)) then
            direction = - direction
}
```

Our calculations made the assumption that the window boundaries are horizontal and vertical. Similar (yet more complex) formulas can be used for the case where the ball bounces off walls that are placed at some arbitrary angle. Also, all of our calculations assumed that the ball was a point. In reality though, the ball has a shape. If, for example, the ball was drawn as a circle centered at **(x,y)**, then it would only detect a collision when the center of the ball reached the border.

How could we fix this? We just need to account for the ball's **radius** during our collision checks:

```
if (((x+radius) >= windowWidth) OR (x-radius)  <= 0)) then
        …
if ((y+radius)  >= windowHeight) OR (y-radius)  <= 0)) then
        …
```

At this point … it is still possible that our ball can get "stuck" in a wall. Why? Well, look at the picture below. We detect a collision by checking if the X value (i.e., center of ball) has gone beyond the boundary (i.e., when (x+radius) >= windowWidth). At this point, the ball is stuck in the wall. So merely changing the direction may not be successful if the ball is too deep in the wall. So, it is best to translate the ball in the x direction so that it is outside the wall. Then we can change the direction as before and the ball will not be stuck in the wall.



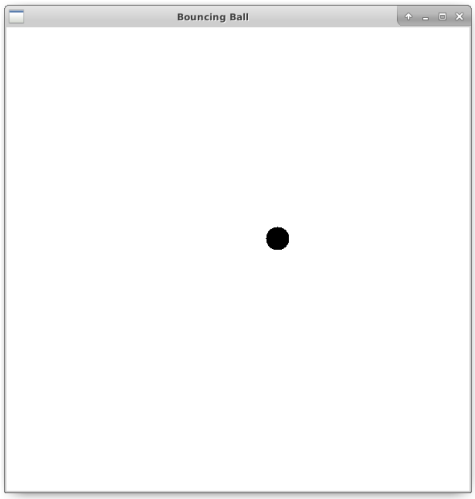
There are similar changes for when the ball hits the left, top and bottom boundaries.

So, we need to handle them separately as follows:

```
if ((x+radius) >= windowWidth) then {
        direction = 180° - direction
        x ← windowWidth - radius
} else if ((x-radius)  <= 0) then {
        direction = 180° - direction
        x ← radius
}
if ((y+radius)  >= windowHeight) then {
        direction =  - direction
        y ← windowHeight - radius
} else if ((y-radius)  <= 0) then {
        direction = - direction
        y ← radius
```

Here is the code:

Code from **ballBounce.c**




```c
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>
#include <unistd.h>
#include <math.h>
#include <time.h>

#define WIN_SIZE    600
#define SPEED       10
#define RADIUS      15
#define PI          3.14159

int main() {
  Display *display;
  Window   win;
  GC       gc;

  float    x = WIN_SIZE/2, y = WIN_SIZE/2;
  float    direction;

  // Opens connection to X server
  display = XOpenDisplay(NULL);

  // Create a simple window, set the title and get the graphics context then
  // make is visible and get ready to draw
  win = XCreateSimpleWindow(display,  RootWindow(display, 0), 0, 0,
                    WIN_SIZE, WIN_SIZE, 0, 0x000000, 0xFFFFFF);
  XStoreName(display, win, "Bouncing Ball");
  gc = XCreateGC(display, win, 0, NULL);
  XMapWindow(display, win);
  XFlush(display);
  usleep(20000);  // sleep for 20 milliseconds.

  srand(time(NULL));
  direction = (rand()/(double)(RAND_MAX))*2*PI;
```

```
  // Go into infinite loop
  while(1) {
    XSetForeground(display, gc, 0xFFFFFF);
    XFillRectangle(display, win, gc, 0, 0, WIN_SIZE, WIN_SIZE);

    XSetForeground(display, gc, 0x000000); // black
    XFillArc(display, win, gc, x-RADIUS, y-RADIUS,2*RADIUS,2*RADIUS,0, 360*64);

    // Move the ball forward
    x = x + SPEED * cos(direction);
    y = y + SPEED * sin(direction);

    // Check if the ball collides with borders and adjust accordingly
    if (x >= (WIN_SIZE-RADIUS)) {
      direction = PI - direction;
      x = WIN_SIZE-RADIUS;
    }
    else if (x <= RADIUS) {
      direction = PI - direction;
      x = RADIUS;
    }
    if (y >= (WIN_SIZE-RADIUS)) {
      y = WIN_SIZE-RADIUS;
      direction *= -1;
    }
    else if (y <= RADIUS) {
      y = RADIUS;
      direction *= -1;
    }
    XFlush(display);
    usleep(5000);
  }
  // Clean up and close the window
  XFreeGC(display, gc);
  XUnmapWindow(display, win);
  XDestroyWindow(display, win);
  XCloseDisplay(display);
}
```

8.4 Event-Handling

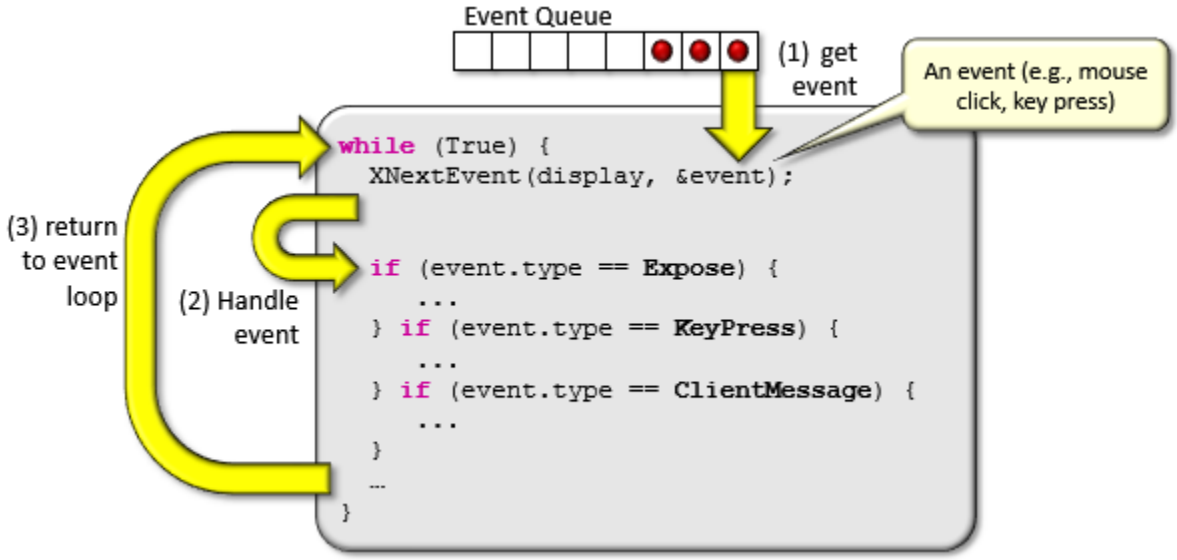
Now that we know how to display graphics and some animation, we need to understand interaction with the user in terms of event handling. One thing that you may have noticed is that when we close our X11 windows at the moment by using the X in the top left of the window, we end up with an error something like this:

```
X connection to :0.0 broken: (explicit kill or server shutdown).
```

We'd like to handle window-closing events properly. To do this, we need to tell the window manager that we are interested in "window deletion" events by calling **XSetWMProtocols()** and registering a **WM_DELETE_WINDOW** message with it. Then we'll get a *client message* from the window manager when the user tries to close the window. We need to add the following code any time after making the window:

```
Atom   WM_DELETE_WINDOW = XInternAtom(display, "WM_DELETE_WINDOW", False);
XSetWMProtocols(display, win, &WM_DELETE_WINDOW, 1);
```

Of course, we then need to handle events. To do this, we need an event loop to handle events one at a time as they come in. The **XNextEvent()** function waits for an incoming event and returns an **XEvent** structure. We can then access the **.type** field of the structure to determine what kind of event it was. We put all of this into an endless loop as shown in this diagram:



What are some of the events that we can handle?

- Keyboard key presses and releases
- Mouse button presses and releases
- Mouse motion/movement within a window
- Mouse entering and leaving a window
- Resizing of the window
- Closing of a window … and many more …

To handle events, we must ***register*** the event with the window manager by indicating which events we would like to handle. This is done with the **XSelectInput()** function, which takes the display, window and a set of ***masks*** that indicate the events to be handled.

Here is an example of how to call this function with many event masks combined using a bitwise OR (i.e., **|**) operation:

```
XSelectInput(display, win,
             KeyPressMask |          // When key is pressed on keyboard
             KeyReleaseMask |        // When key is released on keyboard
             ButtonPressMask |       // When a mouse button is pressed
             ButtonReleaseMask |     // When a mouse button is released
             EnterWindowMask |       // When mouse enters window
             LeaveWindowMask |       // When mouse leaves window
             PointerMotionMask |     // When mouse is moved within window
             ExposureMask |          // When window is exposed
             StructureNotifyMask     // When there is a change in window structure
             );
```

If we do not want to handle any events (or want to disable all temporarily), we use this:

```
XSelectInput(display, win, NoEventMask);
```

Once we do this, we just enter an infinite **while** loop, getting each event and handling it:

```
XEvent    event;

while(1) {
  XNextEvent(display, &event);
  switch(event.type) {
    case ButtonPress:
      …
      break;
    case ButtonRelease:
      …
      break;
    case ConfigureNotify:
      …
      break;
    //... etc ...
  }
}
```

Example:

Here is an example of a program that you can run to test out various X11 events. It displays the number and location of any mouse button presses/releases. It also displays keycodes of keys pressed/released. It indicates when the mouse enters/leaves a window. When the mouse moves within the window, it displays the current mouse location with respect to the window's top/left origin and also the location with respect to the screen's top/left origin. It allows the window to be closed by pressing it's **X** button or when pressing the **ESC** key. Finally, it allows resizing of the window and adjusts things by redrawing the window's background with the new window dimensions.

Code from **eventTest.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>
#include <unistd.h>

int  WINDOW_WIDTH = 600;
int  WINDOW_HEIGHT = 300;

void quit(Display *display, Window win, GC gc) {
  // Clean up and close the window
  XFreeGC(display, gc);
  XUnmapWindow(display, win);
  XDestroyWindow(display, win);
  XCloseDisplay(display);
  exit(0);
}
```

These will change when window is resized.

Call this when all done to release resources.

```
void redraw(Display *display, Window win, GC gc) {
  XSetForeground(display, gc, 0xFFFFFF);
  XFillRectangle(display, win, gc, 0, 0, WINDOW_WIDTH, WINDOW_HEIGHT);
  XFlush(display);
}

int main() {
  Display         *display;
  Window           win;
  GC               gc;
  XEvent           event;
  int              key, button;
  Window           window_returned;
  int              x, y, screenX, screenY, width, height, borderWidth, depth;
  unsigned int     mask_return;
  XConfigureEvent  cEvent;


  // Opens connection to X server
  display = XOpenDisplay(NULL);

  // Create a simple window, set the title and get the graphics context then
  // make is visible and get ready to draw
  win = XCreateSimpleWindow(display,  RootWindow(display, 0), 0, 0,
                            WINDOW_WIDTH, WINDOW_HEIGHT, 0, 0x000000, 0xFFFFFF);

  // Indicate which events we want to handle
  XSelectInput(display, win,
             KeyPressMask |          // When key is pressed on keyboard
             KeyReleaseMask |        // When key is released on keyboard
             ButtonPressMask |       // When Mouse Button is Pressed
             ButtonReleaseMask |     // When Mouse Button is Released
             EnterWindowMask |       // When Mouse Enters Window
             LeaveWindowMask |       // When mouse Leaves Window
             PointerMotionMask |     // When mouse is moved within window
             ExposureMask |          // When window is exposed
             StructureNotifyMask     // When there is a change in window structure
             );

  // Create the window and display it
  XStoreName(display, win, "Event Handler");
  gc = XCreateGC(display, win, 0, NULL);
  XMapWindow(display, win);
  XFlush(display);

  // Indicate that we'd like to be able to gracefully handle window closing
  Atom WM_DELETE_WINDOW = XInternAtom(display, "WM_DELETE_WINDOW", False);
  XSetWMProtocols(display, win, &WM_DELETE_WINDOW, 1);

  // Go into infinite loop handling X11 events
  while(1) {
    XNextEvent(display, &event);

    switch(event.type) {

    case EnterNotify:
      printf("Mouse entered window\n");
      break;
```

This procedure gets called upon startup and then on every

We will handle all these events in this program.

Needed to prevent error upon closing.

```
case LeaveNotify:
  printf("Mouse left window\n");
  break;

case Expose:
  printf("Window has been exposed\n");
  redraw(display, win, gc);
  break;

case KeyPress:
  key = event.xkey.keycode;
  printf("Key has been pressed with keycode: %d\n", key);
  if (key == 0x09) // Check for ESC key, then quit
    quit(display, win, gc);
  break;

case KeyRelease:
  key = event.xkey.keycode;
  printf("Key has been released with keycode: %d\n", key);
  break;

case ButtonPress:
  button = event.xbutton.button;
  XQueryPointer(display, win, &window_returned,
            &window_returned, &screenX, &screenY, &x, &y,
            &mask_return);
  printf("Button %d has been pressed at location (%d, %d)\n", button, x, y);
  break;

case ButtonRelease:
  button = event.xbutton.button;
  printf("Button %d has been released\n", button);
  break;

case MotionNotify:
  XQueryPointer(display, win, &window_returned,
            &window_returned, &screenX, &screenY, &x, &y,
            &mask_return);
  printf("Mouse moved to window pos (%d, %d) & screen pos (%d, %d)\n",
         x, y, screenX, screenY);
  break;

case ConfigureNotify:
  cEvent = event.xconfigure;

  // Need to check for window resizing, since this type of event
  // can be generated for other reasons too
  if ((cEvent.width != WINDOW_WIDTH) || (cEvent.height != WINDOW_HEIGHT)) {
    WINDOW_WIDTH = cEvent.width;
    WINDOW_HEIGHT = cEvent.height;
    printf("Window resized to be %d X %d\n", WINDOW_WIDTH, WINDOW_HEIGHT);
    redraw(display, win, gc);
  }
  break;
case ClientMessage:
  printf("Window closed\n");
  // We really should check here for other client message types,
  // but since the only protocol registered above is WM_DELETE_WINDOW,
  // it is safe to assume that we want the window closing event.
  quit(display, win, gc);
```

Redraw window each time it has been exposed.

Each key has a unique keycode. You will need to look them up or do trial and error.

Buttons can range from 1 to 9.

These variables are all set by function call.

Use this to get the mouse position with respect to the window and also the screen.

Detect window resize if width or height has been changed.

ClientMessage event is generated when window is closed, among other things.

```
    default:
      printf("Unknown Event\n");
    }
  }
}
```

Example:

Consider our bouncing ball example that we discussed earlier. How can we adjust the code so that we can grab the ball with the mouse and throw it around in the window? That is, if the user places the mouse cursor over the ball and clicks, then the ball stops moving and appears to be "stuck" to the mouse cursor until the mouse is released. Then when we let go of the mouse button, the ball should "fly off" in the direction that we threw it with a speed that varies according to how "hard" we threw it.

To do this, we should break the problem down into more manageable steps:

1. Add the ability to grab the ball and carry it around
2. Add the ability to throw the ball
3. Adjust the speed of the ball according to how "hard" we threw it.

To grab the ball, we would need to prevent it from moving (i.e., updating its location) while it is being held. Instead, we would set the ball's location to be the mouse location.

We can create a boolean to determine whether or not the ball is being held:

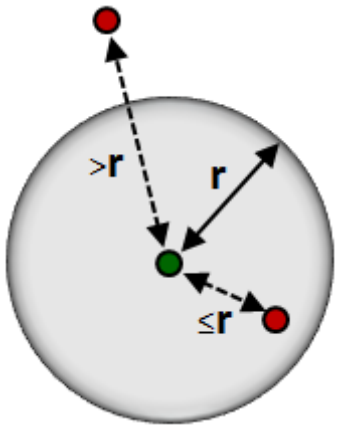
```
grabbed ← False
while(True)  {
    …
    if not grabbed then {
        x ← x + speed * cos(direction)
        y ← y + speed * sin(direction)
    }
    otherwise {
        x ← x position of mouse
        y ← y position of mouse
    }
    …
}
```

All that would be left to do is to set the **grabbed** variable accordingly. When the user presses the mouse button while on the ball, we should set it to **true** and when the user releases the mouse button, we should set it to **false**.

So, we need two event handlers:

```
case ButtonPressed:
    grabbed ← True

case ButtonReleased:
    grabbed ← False
```

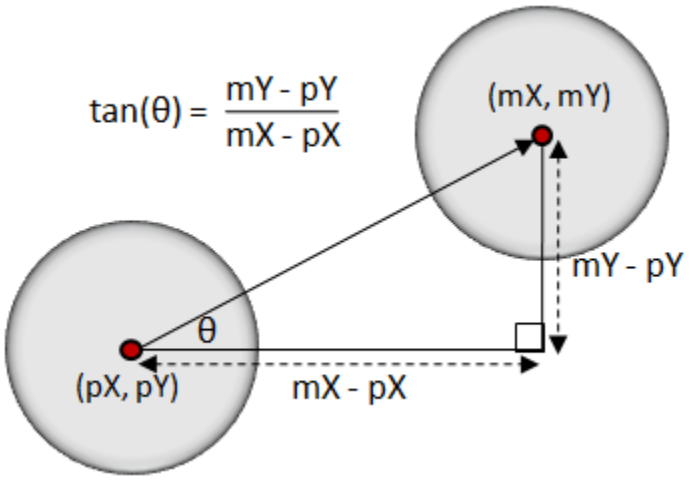
But this will ALWAYS "grab" the ball, even if the mouse cursor was not on it. How can we determine if the mouse cursor is over the ball? We can check to see if the mouse location is within (i.e., **≤**) the ball's radius.
We can compute the distance from the ball's center (i.e., **(x,y)**) to the location of the mouse **(mX, mY)**. If this distance is less than or equal to the ball's radius, then we can assume that the user has "grabbed" the ball.
Here is the adjusted code:




```
case ButtonPressed:
    d ← distance from (x, y) to (mX, mY)
    if (d <= radius) then
        grabbed ← True
}
```

When the user lets go of the ball, it will continue in the direction that it was in before it was grabbed. Now how do we adjust the code so that we are able to "throw" the ball in some particular direction?

Well, upon releasing the mouse, we will need to determine which direction the ball was being thrown in and then set the **direction** variable directly. We can determine the direction that the ball was thrown in by examining the current mouse location **(mX, mY)** with respect to the previous mouse location **(pX, pY)**.



The angle (i.e., **θ**) at which the ball should be thrown will be the **arctangent** of the differences in **x** and **y** coordinates as shown here.

However, in the case that we throw vertically, the difference in **x** coordinates will be zero and we are not allowed to divide by zero.

Fortunately, many computer languages have a function called **atan2(y, x)** which allows you to find the angle that a point makes with respect to the origin **(0,0)**. We can make use of this by assuming that **(pX,pY)** is the origin and translate **(mX,mY)** accordingly as follows: **atan2(mY-pY, mX-pX)**

So, upon a mouse release, we can do this:

```
case ButtonReleased:
    if grabbed then {
        direction ← atan2(mY-pY, mX-pX)
        grabbed ← False
    }
}
```

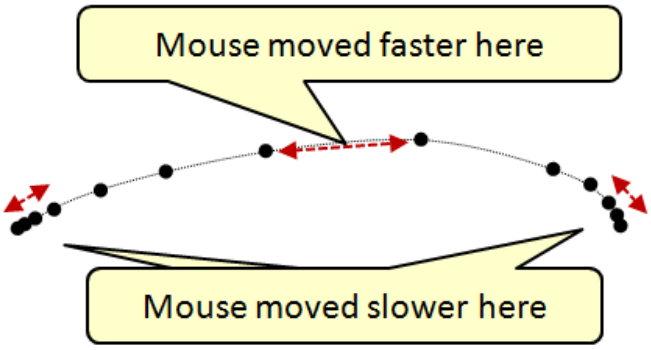
Notice that we only change the direction if we had already grabbed the ball.

One last feature of the program is to allow the ball to be thrown at various speeds. Likely, we want the ball to slow down as time goes on. We should add the following to the algorithm's main **while** loop:

```
speed ← speed – 0.1
if (speed < 0) then
    speed ← 0
```

Now to determine the speed at which the ball is thrown, we can take notice of how the mouse location varies according to the speed at which it is moved.

If the mouse is moved fast, the successive locations of the mouse will be further apart, while slow mouse movements will have successive locations that are relatively closer together.



So, as a simple strategy, the "*strength* of the throw" can be computed as a function of the distance between the current mouse location and the previous mouse location.

We can simply set the **speed** to this distance in the **ButtonReleased** handler as follows:

```
case ButtonReleased:
    if grabbed then {
        direction ← atan2(mY-pY, mX-pX)
        speed ← distance from (pX,pY) to (mX,mY)
        grabbed ← False
    }
}
```

This “should” work well. However, mouse motion events often occur quickly and it could be the case that the previous mouse location is very close to the current mouse location. That is, it could be off by just one **x** value or one **y** value. To get a better sense of the direction that the mouse is being moved in and the speed at which it is moving, we would need to look further back in history than just the previous mouse location. We could keep track of the previous **5** or so locations and then use the location “5 mouse motions ago” to compare.

```
prevCount ← 0
while (True) {
    pX[prevCount] ← mX
    pY[prevCount] ← mY
    prevCount ← (prevCount + 1) % 5
    …
    case ButtonReleased:
        if grabbed then {
            direction ← atan2(mY-pY[prevCount], mX-pX[prevCount])
            speed ← distance from (pX[prevCount],
                                   pY[prevCount]) to (mX,mY)
            grabbed ← False
        }
    }
}
```

Here is the updated code:

Code from **interactBall.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>
#include <unistd.h>
#include <math.h>
#include <sys/time.h>
#include <time.h>

// Constants defined in this program
#define PI            3.14159
#define DECELERATION  0.10    // Rate of deceleration
#define RADIUS        40      // Radius of the ball
#define FPS           100     // frames per second refresh rate

// Window attributes
int   WINDOW_WIDTH = 600;
int   WINDOW_HEIGHT = 600;
```

```
// Ball attributes
float  x = 300, y = 300;    // location of the ball on the window
char   grabbed = False;     // Indicates if ball is being currently held
float  direction;           // Direction of ball movement (radians)
float  speed = 4;           // Current ball speed

// Clean up and close the window
void quit(Display *display, Window win, GC gc) {
  XFreeGC(display, gc);
  XUnmapWindow(display, win);
  XDestroyWindow(display, win);
  XCloseDisplay(display);
  exit(0);
}

// Redraw everything
void redraw(Display *display, Window win, GC gc, int x, int y) {
  XSetForeground(display, gc, 0xFFFFFF);   // white background
  XFillRectangle(display, win, gc, 0, 0, WINDOW_WIDTH, WINDOW_HEIGHT);

  XSetForeground(display, gc, 0x000000); // black ball
  XFillArc(display, win, gc, x-RADIUS, y-RADIUS, 2*RADIUS, 2*RADIUS, 0, 360*64);
  XFlush(display);
}

// Get the current time in microseconds
long getTimeInMicroseconds(){
  struct timeval currentTime;
  gettimeofday(&currentTime, NULL);
  return currentTime.tv_sec * (int)1e6 + currentTime.tv_usec;
}

int main() {
  Display         *display;
  Window           win;
  GC               gc;
  XEvent           event;
  int              button;
  Window           window_returned;
  int              sx, sy, width, height, borderWidth, depth;
  float            d;
  unsigned int     mask_return;
  XConfigureEvent  cEvent;
  int              mouseX, mouseY;
  int              prevX[5], prevY[5], prevCount=0;  // 5 prev mouse locations

  // Opens connection to X server
  display = XOpenDisplay(NULL);

  // Create a simple window, set the title and get the graphics context then
  // make is visible and get ready to draw
  win = XCreateSimpleWindow(display,  RootWindow(display, 0), 0, 0,
                            WINDOW_WIDTH, WINDOW_HEIGHT, 0, 0x000000, 0xFFFFFF);

  // Indicate which events we want to handle
  XSelectInput(display, win,
               ButtonPressMask |      // When Mouse Button is Pressed
               ButtonReleaseMask |    // When Mouse Button is Released
               PointerMotionMask |    // When mouse is moved within window
```

```
            ExposureMask |          // When the window is exposed
            StructureNotifyMask     // When there is a change in window structure
            );

  XStoreName(display, win, "Throwable Ball");
  gc = XCreateGC(display, win, 0, NULL);
  XMapWindow(display, win);
  XFlush(display);

  // Indicate that we'd like to be able to gracefully handle window closing
  Atom WM_DELETE_WINDOW = XInternAtom(display, "WM_DELETE_WINDOW", False);
  XSetWMProtocols(display, win, &WM_DELETE_WINDOW, 1);

  // Set the direction to be random
  srand(time(NULL));
  direction = (rand()/(double)(RAND_MAX))*2*PI;

  // Go into infinite loop, updating the animation at FPS rate
  unsigned long lastRepaint = getTimeInMicroseconds(); // time in microseconds

  while(1) {
    // Keep updating the mouse location. We remember the last 5 locations so that
    // we can get a sense of what direction the ball has been thrown as well as
    // how fast it was thrown.   We will be comparing the current mouse reading
    // with one 5 moves ago.
    prevX[prevCount] = mouseX;
    prevY[prevCount] = mouseY;
    prevCount = (prevCount + 1) % 5;  // Update for the next time around
    XQueryPointer(display, win, &window_returned,&window_returned,
                  &sx, &sy, &mouseX, &mouseY, &mask_return);

    // Handle any pending events, and then we'll deal with redrawing
    if (XPending(display) > 0) {
      XNextEvent(display, &event);
      switch(event.type) {
      case Expose:
        redraw(display, win, gc, x, y);
        break;

      case ButtonPress:
        // Check if the mouse was clicked within the ball's radius
        d = sqrt((mouseX - x)*(mouseX - x) + (mouseY - y)*(mouseY - y));
        if (d < RADIUS)
          grabbed = True;
        break;

      case ButtonRelease:
        if (grabbed == True) {
          // Compare the difference between current mouse location and the one 5
          // mouse motions ago.  Use this to compute the new direction and speed.
          int px = prevX[prevCount];
          int py = prevY[prevCount];
          direction = atan2(mouseY - py, mouseX - px);
          speed = (int)(sqrt((mouseX - px)*(mouseX - px) +
                             (mouseY - py)*(mouseY - py)))/2;
          if (speed > 50) // Limit to something reasonable
            speed = 50;
        }
        grabbed = False;  // Let go of the ball if we were holding it
        break;
```

```
      case MotionNotify:
        if (grabbed == True)  // Refresh screen if we are carrying ball around
          redraw(display, win, gc, x, y);
        break;

      case ConfigureNotify:
        cEvent = event.xconfigure;
        // Need to check for window resizing, since this type of event can be
        // generated for other reasons too
        if ((cEvent.width != WINDOW_WIDTH) || (cEvent.height != WINDOW_HEIGHT)) {
          WINDOW_WIDTH = cEvent.width;
          WINDOW_HEIGHT = cEvent.height;
          redraw(display, win, gc, x, y);
        }
        break;
      case ClientMessage:
        quit(display, win, gc);
      }
    }

    // Get the time in microseconds and store it
    unsigned long end = getTimeInMicroseconds();

    // If it has been long enough, animate and redraw everything
    if (end - lastRepaint > 1000000/FPS) {
      // Draw the ball
      redraw(display, win, gc, x, y);

      // Move the ball forward
      if (grabbed == False) {
        x = x + (int)(speed*cos(direction));
        y = y + (int)(speed*sin(direction));
      }
      else {
        x = mouseX;
        y = mouseY;
      }

      // Slow the ball down
      speed = speed - DECELERATION;
      if (speed < 0)
        speed = 0;

      // Check if the ball collides with borders and adjust accordingly
      if (x >= (WINDOW_WIDTH-RADIUS)) {
        direction = PI - direction;
        x = WINDOW_WIDTH-RADIUS;
      }
      else if (x <= RADIUS) {
        direction = PI - direction;
        x = RADIUS;
      }
      if (y >= (WINDOW_HEIGHT-RADIUS)) {
        y = WINDOW_HEIGHT-RADIUS;
        direction *= -1;
      }
      else if (y <= RADIUS) {
        y = RADIUS;
        direction *= -1;
```

```
      }
      lastRepaint = getTimeInMicroseconds(); // Remember last repaint
    }

    // IMPORTANT: sleep for a bit to let other processes work
    if (XPending(display) == 0) {
      usleep (1000000 / FPS - (end - lastRepaint));
    }
  }
}
```