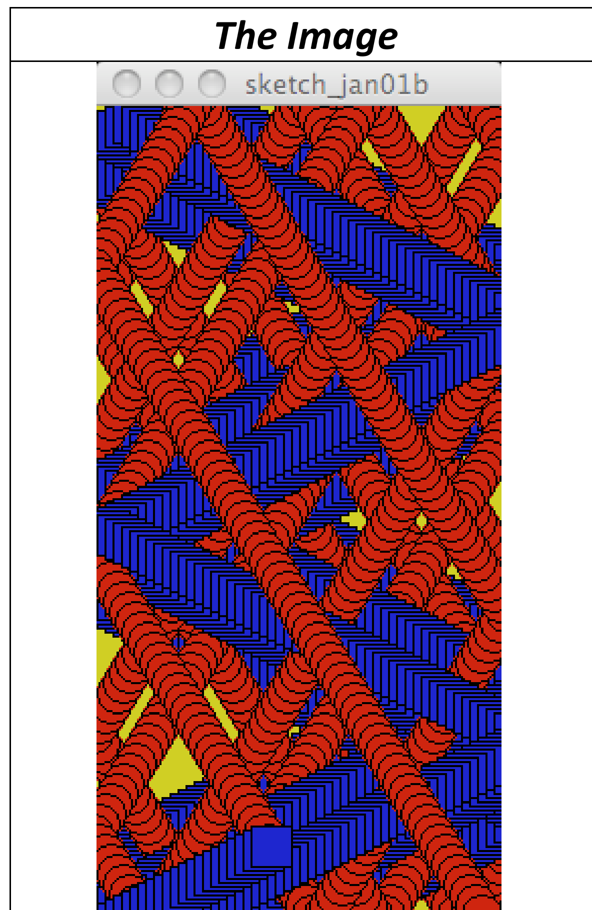
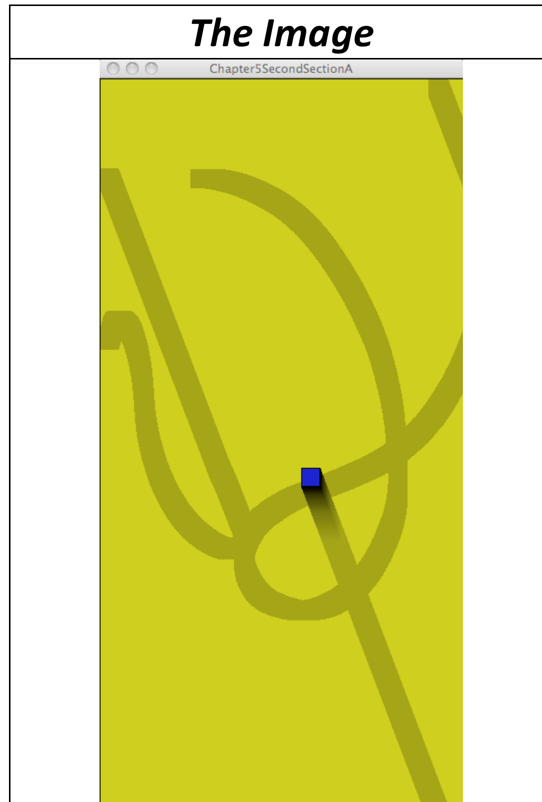
The Junior Woodchuck Manuel of Processing Programming for Android Devices

The Image



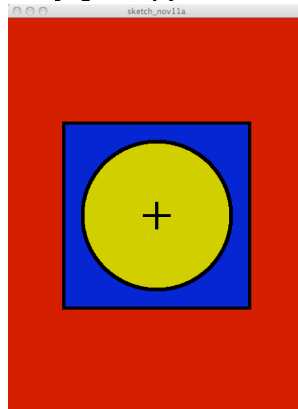
Copyright David Nassar and Jim Roberts, December 2011, Pittsburgh PA

Chapter 6
Almost Home . . .



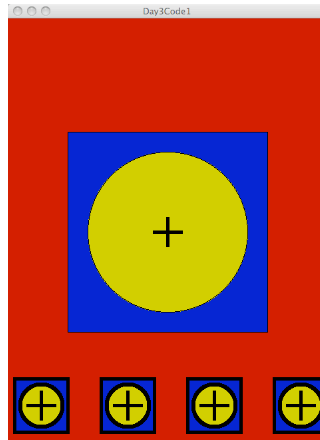
First, A Code Cleanup (and review)…

In Chapter 3 we introduced functions that we write and we use. The function we wrote was named figure( ) and it drew this:



Copyright David Nassar and Jim Roberts, December 2011, Pittsburgh PA

One reason we wrote the figure( ) function was so we could reuse it to draw as many figures as we wanted to without having to copy and past the figure code over and over again. Writing the figure( ) function let us draw this image:



and keep our draw( ) function very simple and short:

```
void draw( )
{
 // Big Figure
 drawFigure( );

 // Preparation for small figures
 w = 80;
 h = 80;
 y =  580;

 // Small Figure - fartherest left
 x = 50;
 drawFigure( );

  // Small Figure second from left
 x = 180;
drawFigure( );

   // Small Figure third from left
 x = 310;
drawFigure( );

   // Small Figure fourth from left
 x = 440;
drawFigure( );
}
```


Copyright David Nassar and Jim Roberts, December 2011, Pittsburgh PA

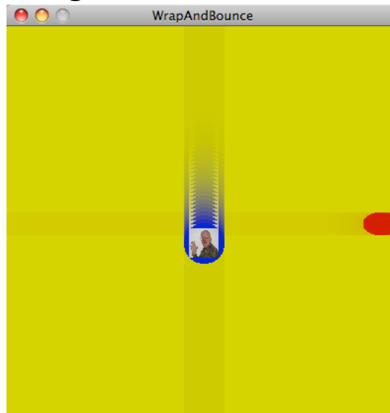
We followed this idea in the code we wrote last time the class met when we wrote this code:

```
//Wrap And Bounce
//Day 3 App Lab
//QVHS and MHS


float x, y, deltaX;
float px, py, deltaPY;
PImage p;


void setup( )
{
  size( 400, 400 );

  imageMode( CENTER );

  noStroke( );

  p = loadImage( "Jim.jpg");

  x = 0;
  y = height/2;
  px = width/2;
  py = 0;
  deltaX = 3;
  deltaPY = 4;
}

void draw( )
{
  prepareScreen( );

  drawBouncingFigure( );
  drawWrappingFigure( );

  changeBouncingFigure( );
  changeWrappingFigure( );
}

void prepareScreen( )
{
  fill( 200, 200, 0, 30 );
  rect(0, 0, width, height);
}

void drawBouncingFigure( )
{
  fill( 200, 0, 0 );
  ellipse( x, y, 33, 22 );
}
```

```
void drawWrappingFigure( )
{
   fill( 0, 0, 200 );
   ellipse( px, py, 40, 40 );
   image ( p, px, py, 30, 30 );
}

void changeBouncingFigure( )
{
  x = x + deltaX;
  if( x > width )  // too far right
  {
    //x = 0;
    //deltaX = deltaX + 2;

    deltaX = -deltaX;
  }

  if ( x < 0 )  // too far left
  {
    deltaX = -deltaX;
  }
}

void  changeWrappingFigure( )
{
  py = py + deltaPY;
   if ( py > height + 40 )
   {
      py = -40;
   }
}
```


Copyright David Nassar and Jim Roberts, December 2011, Pittsburgh PA

This code draws the following:



This is how we write and user our own functions. The code in the draw( ) function shown below tells Processing exactly what to do. However, these functions are not defined in Processing's API so we have to write the code that defines (tells Processing how to do) the behavior requested.

```
void draw( )
{
  prepareScreen( );

  drawBouncingFigure( );
  drawWrappingFigure( );

  changeBouncingFigure( );
  changeWrappingFigure( );
}
```

Here are the definitions for two of these functions:

```
void drawWrappingFigure( )
{
   fill( 0, 0, 200 );
   ellipse( px, py, 40, 40 );
   image ( p, px, py, 30, 30 );
}
```

```
void changeBouncingFigure( )
{
  x = x + deltaX;
  if( x > width )  // too far right
  {
    //x = 0;
    //deltaX = deltaX + 2;

    deltaX = -deltaX;
  }

  if ( x < 0 )  // too far left
  {
    deltaX = -deltaX;
  }
}
```

The syntax that is required is shown in blue. The code in black is up to us. In our code the parentheses are empty. If you take a "real" programming course, you will learn how and when to put stuff into the parentheses. We do not have time to study that part of programming.

Copyright David Nassar and Jim Roberts, December 2011, Pittsburgh PA

Compare the other function definitions on page 4 to these two definitions to see how they are the same and how they are different.

Why do we do this? There are several reasons. One is that it makes the code we want to write much easier to work with, debug, modify, and explain to others.

Another thing that we did last time. We have used some variables in the code that we own.
Here are the variables:

```
float x, y, deltaX;
float px, py, deltaPY;
PImage p;
```

Here is how we initialize them in our setup( ) function:

```
void setup( )
{
  size( 400, 400 );

  imageMode( CENTER );

  noStroke( );

  p = loadImage( "Jim.jpg");

  x = 0;
  y = height/2;
  px = width/2;
  py = 0;
  deltaX = 3;
  deltaPY = 4;
}
```

The function calls like im`ageMode(CENTER)`***and*** `loadImage("Jim.jpg")` ***are explained in the API. Look them up if you are not sure what they are doing.***

You can look at the other function definitions to see how we used these variables.

Copyright David Nassar and Jim Roberts, December 2011, Pittsburgh PA

Why do we do this? You just read this above but here it is again: It makes the code we want to write much easier to work with, debug, modify, and explain to others. It also lets our program changes it behavior as it is running.

We also used some variables owned by Processing. Two of these "many more" are* width *and* height*. These variables are initially 100 and 100. But when we use the* size( ) *function, they have different values. Processing's variable* width *uses the first argument as its value and its variable* height *uses the second argument for its value. You can see how we used these variables in the code below:

```
//Wrap And Bounce
//Day 3 App Lab
//QVHS and MHS
float x, y, deltaX;
float px, py, deltaPY;
PImage p;

void setup( )
{
  size( 400, 400 );
  imageMode( CENTER );
  noStroke( );
  p = loadImage( "Jim.jpg");
  x = 0;
  y = height/2;
  px = width/2;
  py = 0;
  deltaX = 3;
  deltaPY = 4;
}

void draw( )
{
  prepareScreen( );

  drawBouncingFigure( );
  drawWrappingFigure( );

  changeBouncingFigure( );
  changeWrappingFigure( );
}

void prepareScreen( )
{
  fill( 200, 200, 0, 30 );
  rect(0, 0, width, height);
}
```

```
void drawBouncingFigure( )
{
  fill( 200, 0, 0 );
  ellipse( x, y, 33, 22 );
}

void drawWrappingFigure( )
{
   fill( 0, 0, 200 );
   ellipse( px, py, 40, 40 );
   image ( p, px, py, 30, 30 );
}

void changeBouncingFigure( )
{
  x = x + deltaX;
  if( x > width )  // too far right
  {
    //x = 0;
    //deltaX = deltaX + 2;

    deltaX = -deltaX;
  }

  if ( x < 0 )  // too far left
  {
    deltaX = -deltaX;
  }
}

void  changeWrappingFigure( )
{
  py = py + deltaPY;
   if ( py > height + 40 )
   {
      py = -40;
   }
}
```


Copyright David Nassar and Jim Roberts, December 2011, Pittsburgh PA

If we use Processing's variables like this, then if we change the size of the window, we do not have to change anything else in our code.

We have just spent 10 pages blathering about something that programmers call "style".

You do not have to write functions. You can put all of your code in setup( ) *and* draw( ).

You do not have to use variables. You can use numbers everywhere.

This decision in this class is yours.

BUT if you want to be able to quickly fine mistakes, or quickly modify your program, or reuse to do something related but different, or explain it to someone else, this style will make you life as a programmer much easier.

And other programmers that might look at your code will have much greater respect for your work .

Now.. on to some really new stuff. . .

Copyright David Nassar and Jim Roberts, December 2011, Pittsburgh PA

Some Really New Stuff…

We have written code that wraps a moving figure and code that bounces a moving figure. Let's take the code that wraps a moving figure and add user control to the movement.

The way we alter the speed and direction of the rect is to alter the values of the variables deltaX for the ellipse and deltaPY for the picture. It is possible to do this while the program is running. We saw that last time. Since we can do that, we can put in some code that will let the user "drive" the image on the screen.

This question brings us to a new kind of function. Let's review – we have seen three kinds of functions:

- ***Processing writes them and we used them: rect( ), fill( ), . . .***
- ***We write them and Processing uses them: setup( ), draw( ).***
- ***We write them and we use them: figure( ), moveRectFigure( ), . . .***

Let us introduce you to a fourth kind of figure:

- ***We write it and Processing will use it when the "user does something".***

Anytime a "user does something" with or to the computer, it is called an "event". The typical events happen when the user presses a key or moves the mouse. There are many different events. We are going to use one that works with the computer, in the Android emulator, and your Android device.

The event we are going to use happens when the user drags the mouse (dragging means that the mouse button is down) or moves their finger on the Android screen. Here is how we do it.

If we write a function named

```
mouseDragged( )
{

}
```


Copyright David Nassar and Jim Roberts, December 2011, Pittsburgh PA

then whenever the user:

- ***presses the mouse button and moves the mouse on the computer,***
- ***presses the mouse button and moves the mouse in the emulator, or***
- ***puts and moves their finger on the Android device screen***

Processing will use this function.

It must be named exactly as show and the parentheses must be empty.

Inside the braces we can do anything that we want to do.

What we want to do is change the speed of the movement of the rect.

Question, "How do we do that?"

Copyright David Nassar and Jim Roberts, December 2011, Pittsburgh PA

Here is one way we might solve this:

- **user moves mouse right – horizontal speed increases**
- **user moves mouse left – horizontal speed decreases**

This might work. You may have a better idea but your teacher is writing this so it will have to do…sigh…

Another Question – "How do we know which way the mouse is moving?" When we have a problem like this and we are not sure how to do it or we have never solved a problem like this, a good way to start is to list all of the data we have (variables) and see what might be helpful.

Let's see… we have our variables in the code:

```
float x, y, deltaX;
float px, py, deltaPY;
PImage p;
```

Which variable or variables do we need to change so we can speed up and slow down the picture (not the ellipse)? Think about it.

Copyright David Nassar and Jim Roberts, December 2011, Pittsburgh PA

Did you really think about it?

```
deltaPY
```

This is the variable that controls the speed of the picture. If we change it, we change the speed of the picture.

Now which Processing variables can we use to help us alter our variables?

width, height, mouseX, mouseY

Again, think about it.

We are working with the vertical movement only – you will work on the horizontal movement during the next week.

It seems reasonable to check the vertical movement of the mouse. So, we will use mouseY.

But we only know where the mouse is now (**mouseX and mouseY*****). In order to know which way the user moved it, we need to know where it was. If we know where it was, we can compare where it was to where it is and figure out which way it is moving.***

Well, where was it?
It turns out that Processing "remembers" where the mouse was in the previous frame. Processing stores that information in two other variables that we can use:

pmouseX and pmouseY

which stands for "Previous Mouse X" and "Previous Mouse Y".

We know - you are probably thinking, "Is there anything Processing does not know?"

Great, now a third question – "How do we use these variables?"

We are working with the vertical movement. The vertical movements should be very similar.

Copyright David Nassar and Jim Roberts, December 2011, Pittsburgh PA

If the user is moving the mousedown then the y coordinate of the mouse would be getting larger from the past frame to the current frame. The top edge is position zero and the bottom edge is position `height` *or in this code, 400*

```
void setup( )
{
    size( 400, 800 );
    . . .
```

so when the user is moving the mouse to the down, mouseY will get bigger. We need to ask the question, "Is the value of mouseY is getting bigger?"

Remember that the way we ask questions in Processing is to use the `if( )` *so we can do that with some code like this:*

```
  if ( mouseY > pmouseY)
   {

   }
```

This says in English "If the value of the variable, mouseY is bigger than the value of the variable, pmouseY" which is what we want to know.

If the answer is true, then we want to increase the value of the variable `deltaPY`

How much we increase it is up each of us. We can increase it by a small amount like 0.1 or by a larger amount like 1, or by a huge amount like 10.

We have to try different values and see what we like. Ideally this value will be a variable so we can find and change it easily as we tinker and tune our code. Let's assume we have a variable named `changeAmount` *and it is set to a small value like 0.1.*

If we do, we can write the code like this:

```
if ( mouseY > pmouseY)
   {
      deltaPY = deltaPY + changeAmount;
   }
```


Copyright David Nassar and Jim Roberts, December 2011, Pittsburgh PA

The new mouseDragged( ) *would look like this:*

```
void mouseDragged ( )
{
  if ( mouseY > pmouseY)
  {
     deltaPY = deltaPY + changeAmount;
  }

}
```

If the movement upward , we can subtract changeAmount *from* deltaPY.

This same pattern can be used for the horizontal movement of the mouse.

Remember that you need to add horizontal movement. You will need a deltaPX variable and you will need add code to the

```
changeWrappingFigure( )
```

function to create horizontal movement. We leave them for you to write.

On the next page is your teacher's version of this code. He has left a long trace on the window so you can see how the movement works.

Copyright David Nassar and Jim Roberts, December 2011, Pittsburgh PA

In the Next Exciting Chapter…

We are going to try to put this code together with the functions that bounce the circle off the edges of the window and use the resulting code for the basis for a game. You will try to drive the rect around the screen and collide with the bouncing circle. If you do collide, you will "dramatize" the collision and increase your score. You will display your score on the screen and, possibly how long you have been playing the game.

We will have to figure out how to detect the collision and how to display text and time the game. We can do that!

So, go back over this and the previous chapters and try to understand the code. If you do not, write your questions down and bring them to class.

if you have time, look up the distance( ), text( ), and millis( ) functions in the API.

Good programming. . .

Copyright David Nassar and Jim Roberts, December 2011, Pittsburgh PA