Using Coordinates to Move a Graphic

Before you begin, make sure that you understand how the coordinate system works on a Form:

https://sites.google.com/gotvdsb.ca/aldworth/home/ics-2o/intro-to-programming/coordinate-system-in-vb

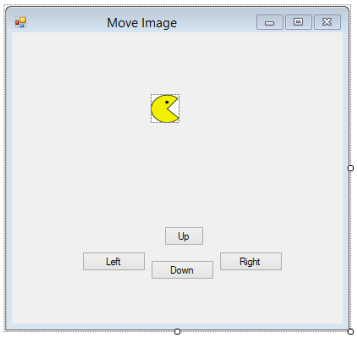
Part 1 - Create your Form and user interface

We will start by creating a new project named Move Image. Create the following interface:

frmMoveImage **imgPacMan**

btnLeft **btnRight**

btnUp **btnDown**



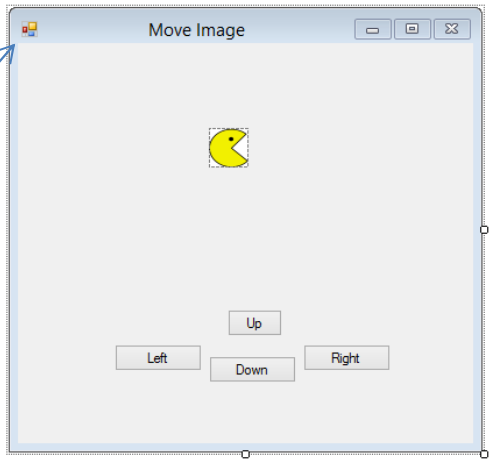
Create an integer variable called **intSpeed** at the top of your program, and set it to 5 in the Form Load event.

Add the four Pac-Man images provided in the Images folder (or any images of your choice) into the Project Resources. These will represent your moving character in different directions. Set the image of imgPacMan to one of the Pac-Man pictures from resources and set its **Sizemode** property to **Stretched**.

Part 2 - Moving the Character and the Coordinate System

PictureBox controls (and many other controls) have a **Location** property that includes an X and Y coordinate. We can set a new Location value to move a PictureBox.

Because the upper-left portion of an application window tends to be fixed, this is where the coordinate (0, 0) is. The x-coordinates increase as you move right, and the y-coordinates increase as you move down.



Add the following line of code to the **btnRight** Click event:

We can set the Location property of our PictureBox to a new location by making a **new Point**

We do not want to change the y-coordinate.

```
imgPacMan.Location = New Point(imgPacMan.Location.X + intSpeed, imgPacMan.Location.Y)
```

imgPacMan.Location.X will give us the x-coordinate of the upper left corner of **imgPacMan**.

Here we add **intSpeed** which has a value of 5 to the current x-coordinate of our image.

Run your program and test the btnRight.

Moving in All Directions

Add code to the appropriate Click events for the remaining directional Buttons (btnLeft, btnUp and btnDown) on your Form.

Move Left*:* instead of adding **intSpeed** to the x-coordinate (as we did to move right), subtract it to move left.

Moving Up: to move our PictureBox up, we must subtract **intSpeed** from the y-coordinate (recall that y-values increase as you move down).

```
imgPacMan.Location = New Point(imgPacMan.Location.X, imgPacMan.Location.Y – intSpeed)
```

Moving Down: try this on your own using a similar idea as in moving left.

Run your Program. What happens when PacMan reaches the edge of your Form?

Things to try:

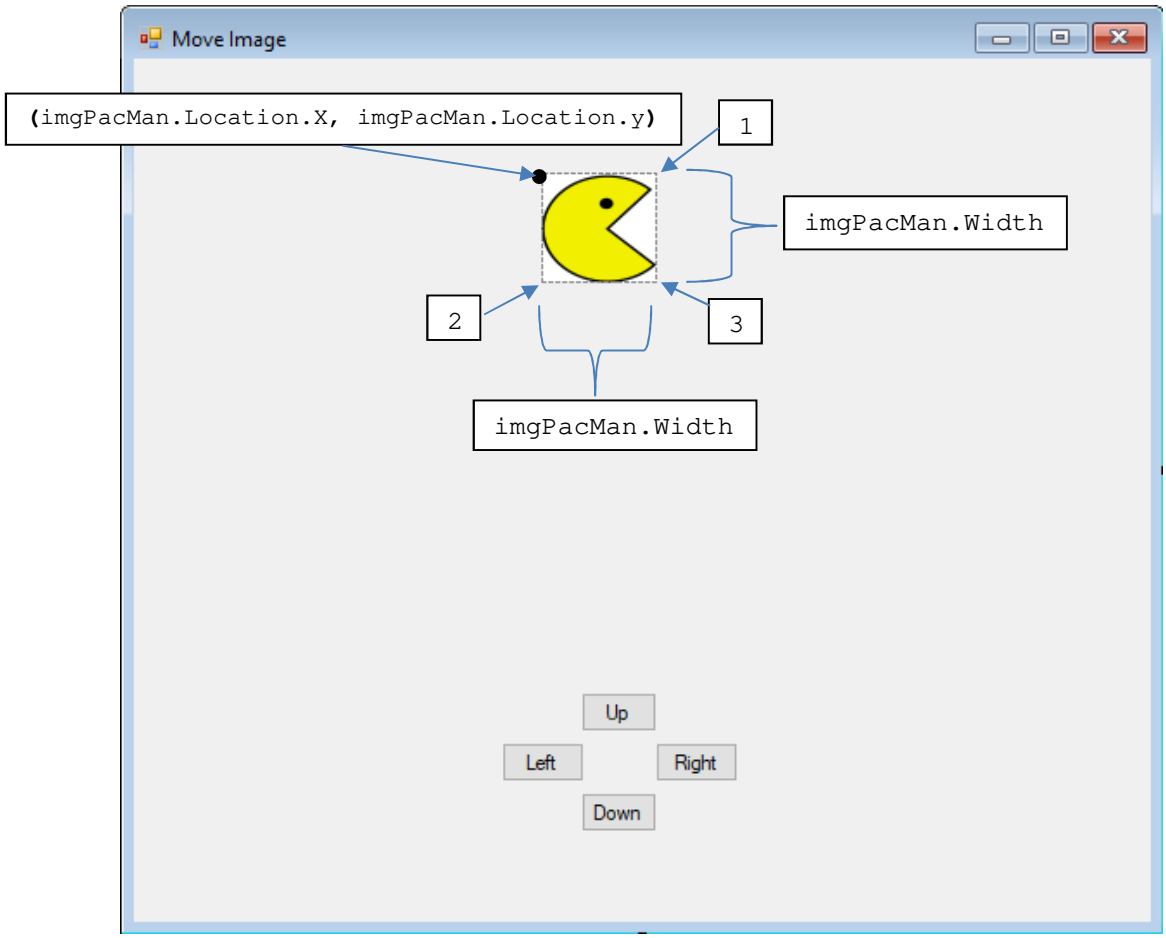
- Experiment with the speed at which your character moves.
- We have four different images of Pac Man in the program resources. Change the **Image** property of imgPacMan to display a picture of Pac Man facing in the direction that he is travelling.
- Add two more Buttons called btnGrow and btnShrink that will change the **Size** property of your PictureBox. You decide how much.
  Hints:
  - `imgPacMan.Width` and `imgPacMan.Length` will give you the current width and height.
  - Use '**`new Size(width, height)`**' in a similar way to the '**`new Point(x, y)`**' to change the size of a PictureBox: `imgPacMan.Size = new Size(_____, _____)`
  - If your PictureBox is not a perfect square you will want to use a multiplier, or a ratio of the sides in order to get it to grow proportionally.
- Learn how to use **KeyDown** events so that the user can use the arrow or (W,A,S,D) keys to move Pac Man instead of Button Click events.
- Use two variables for speed. One for up and down and another for left and right.
- There are **Left** and **Top** properties of PictureBoxes that allows us to **set** the distance of the left side and top of the PictureBox from the edge of the Form (there are also READ ONLY properties for **Right** and **Bottom**). Comment out (don't delete) the code you used to move imgPacMan left, right, up and down (using an apostrophe: ') and replace it with the code below:
  - **Move Left:** imgPacMan.Left -= intSpeed *This allows you to modify the x coordinate*
  - **Move Right:** imgPacMan.Left += intSpeed *You cannot set the **Right** property directly*
  - **Move Up:** imgPacMan.Top -= intSpeed *This allows you to modify the y-coordinate*
  - **Move Down:** imgPacMan.Top += intSpeed *You cannot set **Bottom** property directly*

Hey, that's easier than using the Location property! Why would we not just use Top and Left all the time?

Answer: When we change the **Top** or **Left** property of a PictureBox, it is moved immediately on our Form. Once we start getting into collision detection, we may to want to know if our character is going to collide with something before we actually move it. This will save your program from drawing a character inside a wall for a split second. We will use **Top** and **Left** (and **Bottom** and **Right**) when we can.

Part 3 - How do I Keep my Character from Running off my Form?

It is important to first understand how we can get the x and y vaules of the top, bottom, left and right sides of our PictureBox. The x and y coordinates of the **Location** property of our PictureBox (and all controls) are its upper left corner. Each control also has a **Width** and **Height** property. We can do some simple math to figure out the x value of the right side of our PictureBox, and the y value of the bottom of our PictureBox.



X Value of the <u>right</u> side of our PictureBox: add the width to the x-value of the Location property.

`imgPacMan.Location.X + imgPacMan.Width` **OR** `imgPacMan.Right`

Y-Value of the <u>bottom</u> of our PictureBox: add the height to the y-value of the Location property

`imgPacMan.Location.Y + imgPacMan.Height` **OR** `imgPacMan.Bottom`

What are the coordinates of each remaining corner (2 possibilities for each)?

1:

2:

3:

Make sure we are moving to a place on the Form.

Each time we want to change the Location property of our imgPacMan, we should test to see if its future Location will cause any part of it to be outside of our Form before we actually move it. We do not want to move it off our Form only to move it back again. This will cause jittery movement at the bounds of our Form.

Left Side

Making sure that our PictureBox does not move off the left side of our Form is relatively straightforward. We just need to make sure that after we subtract intSpeed from its current x value, that it is not less than zero (the left edge of the Form).

Add the following code to your **btnLeft** click event:

We will use this variable to tell us whether we are able to apply a normal movement by intSpeed later.

```
Dim bolSafeMove = True
```

Future x coordinate.

If the move will put imgPacMan partially off the Form, set it right against the left edge instead by making its x-value 0.

```
If imgPacMan.Left - intSpeed < 0 Then
       imgPacMan.Left = 0
       bolSafeMove = False
End If
If bolSafeMove Then
       imgPacMan.Left -= intSpeed
End If
```

This tells us that we were not able to move by intSpeed.

This is done if we didn't detect a movement off the Form earlier.

Right Side

Making sure we do not go off the right side of the Form is a bit more complicated because we can only change the Left property of imgPacMan. To find the x-value that we want to set imgPacMan.Left to we subtract imgPacMan's width from the edge of the Form.

Add the following code to the **btnRight** click event:

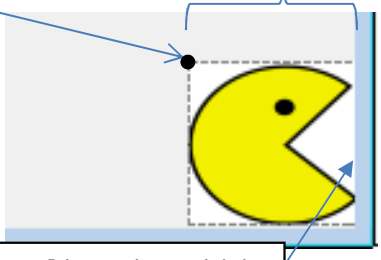
imgPacMan.Right (read only) will give us the x-value of the right side of imgPacMan.

Me.ClientSize.Width is the width of the Form (also the x-value of the right edge of your Form).

```
Dim bolSafeMove = True
If imgPacMan.Right + intSpeed > Me.ClientSize.Width Then
   imgPacMan.Left = Me.ClientSize.Width - imgPacMan.Width
   bolSafeMove = False
End If
```

If part of our Picturebox will go off the edge of the Form when we move it, we can place it at the edge instead.

```
If bolSafeMove Then
       imgPacMan.Left += intSpeed
End If
```



Things to try:

- What effect does changing **intSpeed** have on your program?
- How could you to allow the user the ability to change the speed of imgPacMan themselves (directly or indirectly)? Try it.
- Use a similar idea as left and right to keep imgPacMan from leaving the **top** or **bottom** of your Form (you will be using the **Top** and **Bottom** properties of imgPacMan along with the **Height** property).
- Put in a background image. Your program will be more efficient if you make the BackgroundImage property of the Form contain the image instead of using a PictureBox. Use image editing software to make the size of the background image be the same as your Form's ClientSize before importing it into your program. You do not want to force your program to re-size images each time the Form is updated.
- Remove (or comment out) the checks that prevent imgPacMan from leaving the confines of the Form. When the player moves imgPacMan off the edge of the Form, have imgPacMan re-enter the Form from the opposite side (or a random side).
- Use a String variable to keep track of the direction Pac-Man is facing. Add an If Statement to each direction Button so that the Image property of imgPacMan only changes when PacMan changes direction. This will save your program from re-loading an image into imgPacMan each time a Button is clicked and only do it when the picture needs to be changed to face a new direction.

Part 4 - Collision Detection

Important information:

In order to detect collisions, we are going to use a function called IntersectsWith(). This function will take two rectangles, and tell you if they intersect. The rectangles that IntersectsWith() compares consist of a single coordinate (upper left) and a width and height. There is a property of PictureBoxes (along with most other controls) called Bounds that is such a rectangle.

We can get the bounding rectangle of imgPacMan using this line of code:

```
imgPacMan.Bounds
```

How can we have our character eat a cookie?

Create a PictureBox and name it **imgCookie**. Find the image of a cookie, and add it to the resources and then put that image into **imgCookie**.

Each time we move our character, we need to check to see if we collided with our cookie. Do this with the following code in the **btnRight** click event (after you have already moved imgPacMan):

```
'Checks for collision with cookie
If imgPacMan.Bounds.IntersectsWith(imgCookie.Bounds) Then
       imgCookie.Visible = False
End If
```

This will return **True** if the rectangles that make up the boundaries of the two PictureBoxes touch.

Try **Contains()**, instead of **IntersectsWith()** for this check. What effect does this change have?

How can I have an obstacle block my character?

Create a Picturebox and name it **imgRock**. Find the image of a rock, and add it to the resources and then put that image into **imgRock**.

Each time we want to move our character, we need to check to see if we will collide with an obstacle before we actually move. If we detect a future collision, we will place the edges of our two PictureBoxes boxes next to one another instead of overlapping.

To do this, we will make a virtual Rectangle that will be in the location that we intend on moving imgPacMan to. We will call this virtual Rectangle **tempRect**.

To make a virtual Rectangle, we need an **(x, y) coordinate** of the top left, and a **width** and **height**.

Moving Right

Add the following lines of code to the beginning of the **btnRight** click event:

```
Dim tempRect As Rectangle
tempRect = New Rectangle(imgPacMan.Location.X + intSpeed, imgPacMan.Location.Y, imgPacMan.Width, imgPacMan.Height)
```

Apply the move to the x-coordinate to move imgPacMan to the right.

All other properties of imgPacMan should remain the same.

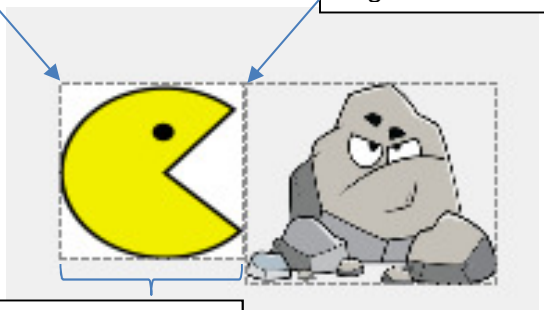
Now that we have a Rectangle that can represent where imgPacMan will be after moving him, we can use this Rectangle to check for collisions.

We need to add an **If** statement to btnRight. We can put this check just after we check to see if imgPacMan moves off the edge of our Form.

Add this to the **btnRight** click event:

```
'Checks for collision with obstacle
If tempRect.IntersectsWith(imgRock.Bounds) Then
        imgPacMan.Left = imgRock.Location.X - imgPacMan.Width
        bolSafeMove = False
End If
```

We want the right side of imgPacMan to rest against the left side of the obstacle. That means that imgPacMan's x-value should be exactly its width away from imgRock's x-value.



imgRock.Location.X

imgPacMan.Width

Moving Left

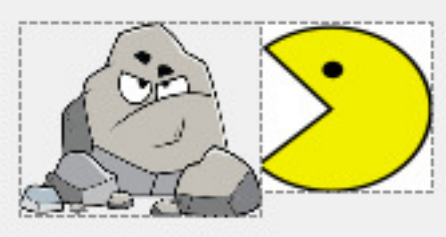
We essentially do the same thing, except if we detect a collision, we will set imgPacMans left edge against the right side of imgRock.

Create our temporary rectangle:

```
Dim tempRect As Rectangle
tempRect = New Rectangle(imgPacMan.Location.X - intSpeed, imgPacMan.Location.Y, imgPacMan.Width, imgPacMan.Height)
```

We now can add our test for a collision:

```
If tempRect.intersectsWith(imgRock.Bounds) Then
      imgPacMan.Left = imgRock.Right
      bolSafeMove = False
End If
```



Things to try

- When your character eats the cookie, instead of making it invisible, change the image to crumbs
- Add multiple 'edible' objects
    - If you want different effects from a collision with different objects, you will need separate If statements for each edible
- Add a 'Portal' that will 'transport' PacMan to a different Location on your Form. Try to use the Contains() method for collision detection instead of IntersectsWith()
- Add a second rock (don't add more though, there is an easier way)
- Change the players speed/size/image/behavior in some way when an object is consumed

Part 5 – Multiple Collisions

The above methods work for detecting collisions with a small number of objects but it does not scale well. For example, if you had ten walls, would you want to type out ten If statements? What about fifty walls?

The solution to that is to put items that have similar behaviors into groups called Lists. For example, PacMan should interact with all impassible objects in the same way, so we can group all of our rocks together. He likely would interact with multiple cookies in the same way.

Creating a List of Rocks

Add the following line of code to the top of your program:

```
Dim Rocks As List(Of PictureBox)
```

We are going to add all of the **PictureBoxes** that are rock obstacles to this list.

Make Rocks and Add them to List

Create 2 more PictureBoxes, name them **imgRock2** and **imgRock3**, and set their image property to a rock. You can save time by copying and pasting the first rock PictureBox you made. Just make sure you change the Name property.

To add the three rocks to your List, add the following code to your Form Load event.

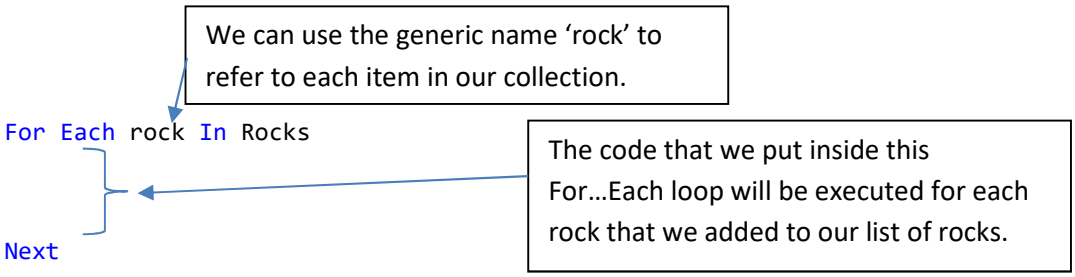
```
Rocks.Add(imgRock)
Rocks.Add(imgRock2)
Rocks.Add(imgRock3)
```

We now can iterate through our collection Rocks to detect collisions.

Use a Loop to Iterate through Collection

Since we want the effect of bumping into a wall to be the same no matter what wall we hit, we can use pretty much the same code as before. All we need to do it all a loop.

Step 1 - Add the following code just before your collision check with imgRock.



Step 2 - Cut and paste the entire If statement check that contained our collision detection with the rock into this For..Each loop:

```
For Each rock In Rocks
       If tempRect.IntersectsWith(imgRock.Bounds) Then
              imgPacMan.Left = imgRock.Location.X - imgPacMan.Width
              bolSafeMove = False 'Indicates that a normal move by intSpeed should not be done
       End If
Next
```

Step 3 – Change each instance of **imgRock** to **rock** in the code:

```
For Each rock In Rocks
       If tempRect.IntersectsWith(rock.Bounds) Then
              imgPacMan.Left = rock.Location.X - imgPacMan.Width
              bolSafeMove = False 'Indicates that a normal move by intSpeed should not be done
       End If
Next
```

This needs to be done because each time our loop moves onto the next item in our collection, it will be stored in a generic variable named 'rock' so that we can refer to it without using its Name property.

Add a For…Each loop to the remaining directional movements.

If you haven't done so yet, you may want to try the **Keyboard Events** tutorial before you start on this mini project, and then have the movement based on events from the keyboard (such as arrow keys, or W,A,S,D) instead of Button clicks.

Mini Project

Create a mini maze game with your own characters and graphics. Add some interesting features such as:

- Have a finish line unlocked by a key that must be eaten
- Present some type of victory screen upon completion
- Have walls that change location to block or unblock sections when certain power-ups are consumed
- Have hazards that cause the player to 'die'
- Add a transporter/portal
- Add a health mechanic
- Any other interesting ideas