**Algorithms**

***What is an algorithm?***

When we have a problem to solve - an algorithm is the breakdown of the problem into a set of instruction that we can use to implement a solution to the problem.



The standard example used is a recipe, say for making a cake….

***Software Engineering Algorithms***

In the context of this particular course (programming) the process of solving a problem can be defined as the following steps:

Step1:

* Define the problem - as a series of steps (the algorithm)
* Get something working - code a solution (code - code - code)

Step 2:

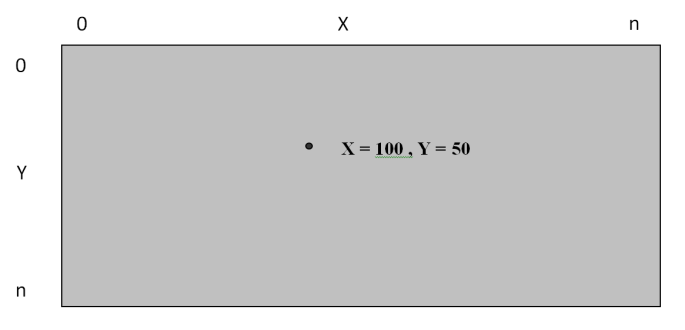
* Refine the solution - remove redundant code and optimize (recode)
* Make the solution re-useable across the application (put in a method)

Step 3:

* Make the solution portable (put in a class)

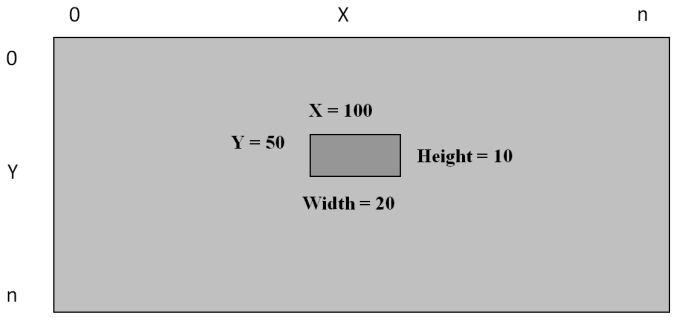
***Problem - Moving Images***

Let’s look at a basic example of this process - by moving some images on a window. First a little background information on the problem.

***2D coordinate systems***

In a 2D coordinate system we have a horizontal X-axis and a vertical Y-axis.

Points can be described as pairs of coordinates. **i.e. X = 100 , Y = 50**

******This system can translate into the internal dimensions of a Grid when we use an Image object.

We can access the image’s the X and Y coordinates using the first two parameters of the margin ***Margin="100,50,0,0"*** property. The parameters as the values represent Margin="***distance from left edge of the grid***, ***distance from top edge of the grid***,0,0"

Updating a Image’s Margin allows us to move the object to a new location - as the following code shows ***testImage.Margin = new Thickness(55, 110, 0, 0);***.

So now we have some background let’s solve some associated problems with moving object:

* Move using a key press
* Lock to the grid
* Move and Lock to the grid
* Follow an object
* Run away from an object
* Collide with an object
* Generate a random number

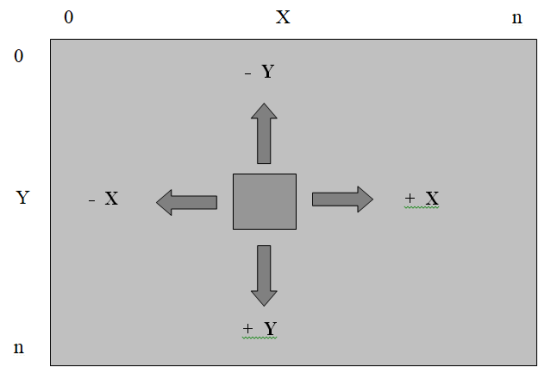
Note: these algorithms can be used to develop a basic flocking simulation.

As we have discussed previously we will develop solutions to these problems in a three stage process.

The following example projects can be used to develop your solutions:

* ***AlgorithmsStage1 -*** Build your basic solution in this project.
* ***AlgorithmsStage2 -*** Refine your solution in this project as a method.
* ***AlgorithmsStage3 -*** Extend your solution by adding to a class.

Note: There are also solution to the problems in the ***AlgorithmsStage1Solution, AlgorithmsStage2Solution*** and ***AlgorithmsStage3Solution*** projects.

***Problem -*** ***Move***

Moving object in a 2D coordinate system is achieved by incrementing and decrementing along the X-axis and Y-axis.

For an object to be able to move we need to consider the following things:

* Find the location of the object
* To move an object right increment the X-axis
* To move an object left decrement the X-axis
* To move an object down increment the Y-axis
* To move an object up decrement the Y-axis
* Reset the location of the object

The following code expresses this algorithm

|  |
| --- |
| Add to the ***MainPage.xaml.cs*** file in the ***AlgorithmsStage1*** project |
| #region Move using a key press  leftMargin = testImage1.Margin.Left;  topMargin = testImage1.Margin.Top;  rightMargin = testImage1.Margin.Right;  bottomMargin = testImage1.Margin.Bottom;  if (flagA == true) leftMargin = leftMargin - 5.00;  if (flagD == true) leftMargin = leftMargin + 5.0;  if (flagW == true) topMargin = topMargin - 5.00;  if (flagS == true) topMargin = topMargin + 5.00;  testImage1.Margin = new Thickness(leftMargin, topMargin, rightMargin, bottomMargin);  #endregion |

Now let’s refine this into a method

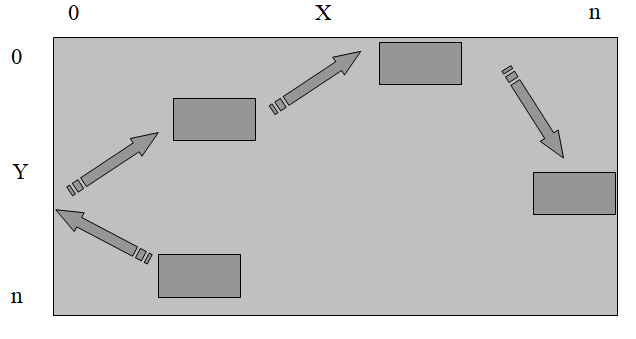
|  |
| --- |
| Add to the ***MainPage.xaml.cs*** file in the ***AlgorithmsStage2*** project |
| #region Move using a key press  Move(testImage1, flagA, flagD, flagW, flagS, 5.00);  #endregion  #region Move using a key press  /// <summary>  /// Move an Image  /// </summary>  public void Move(Image anImage, bool left, bool right, bool top, bool bottom, double speed)  {  double leftMargin = anImage.Margin.Left;  double topMargin = anImage.Margin.Top;  double rightMargin = anImage.Margin.Right;  double bottomMargin = anImage.Margin.Bottom;  if (left == true) leftMargin = leftMargin - speed;  if (right == true) leftMargin = leftMargin + speed;  if (top == true) topMargin = topMargin - speed;  if (bottom == true) topMargin = topMargin + speed;  anImage.Margin = new Thickness(leftMargin, topMargin, rightMargin, bottomMargin);  }  #endregion |

Finally let’s refine this into a class

|  |
| --- |
| Add to the ***MainPage.xaml.cs*** file in the ***AlgorithmsStage3*** project |
| #region Move using a key press  Utils.Move(testImage1, flagA, flagD, flagW, flagS, 5.00);  #endregion |

|  |
| --- |
| Add to the ***Utils.cs*** file in the ***AlgorithmsStage3*** project |
| #region Move using a key press  /// <summary>  /// Move an Image  /// </summary>  public static void Move(Image anImage, bool left, bool right, bool top, bool bottom, double speed)  {  double leftMargin = anImage.Margin.Left;  double topMargin = anImage.Margin.Top;  double rightMargin = anImage.Margin.Right;  double bottomMargin = anImage.Margin.Bottom;  if (left == true) leftMargin = leftMargin - speed;  if (right == true) leftMargin = leftMargin + speed;  if (top == true) topMargin = topMargin - speed;  if (bottom == true) topMargin = topMargin + speed;  anImage.Margin = new Thickness(leftMargin, topMargin, rightMargin, bottomMargin);  }  #endregion |

Now comment your code….

***Problem -*** ***Lock to the grid***

Next let’s look at locking the Image to the grid’s boundaries.

To stop a rectangular object leaving a domain we need to ensure the following:

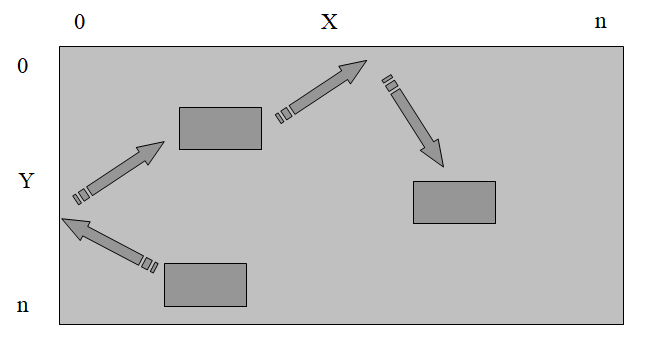
* Find the location of the object
* The Right edge of the rectangular object does not pass the Right side of the domain
* The Left edge of the rectangular object does not pass the Left side of the domain
* The Bottom edge of the rectangular object does not pass the Bottom side of the domain
* The Top edge of the rectangular object does not pass the Top side of the domain
* Reset the location of the object

|  |
| --- |
| Some example code for the solution |
| // if the object has passed the right edge of the grid stop it  if ((leftMargin + testImage1.Width) > TestGrid.Width) leftMargin = TestGrid.Width – testImage1.Width; |

Add code to the following projects:

* ***AlgorithmsStage1 -*** Build your basic solution in this project.
* ***AlgorithmsStage2 -*** Refine your solution in this project as a method.
* ***AlgorithmsStage3 -*** Extend your solution by adding to a class.

Now comment your code….

***Problem -*** ***Move and Lock to the Grid***

Next let’s combine a couple of things by looking at an algorithm to make an object bounce around the grid.

In this case we will need to move it and check to see if it has hit the side of the grid, if so we will reverse its direction on the axis.

To achieve this we need to ensure the following:

* Find the location of the object
* Move the object left if required
* Move the object right if required
* Move the object up if required
* Move the object down if required
* When the Left edge of the rectangular object hits the Left side of the domain
  + Stop it
  + Move right
* When the Right edge of the rectangular object hits the Right side of the domain
  + Stop it
  + Move left
* When the Top edge of the rectangular object hits the Top side of the domain
  + Stop it
  + Move down
* When the Bottom edge of the rectangular object hits the Bottom side of the domain
  + Stop it
  + Move up
* Reset the location of the object

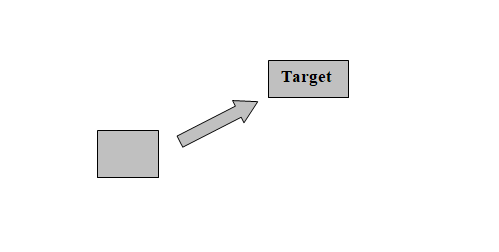
|  |
| --- |
| Some example code for the solution |
| // set the move direction left  if (testFlag1X == true) leftMargin = leftMargin - 2;  // if the object has passed the right edge of the grid stop it  if ((leftMargin + testImage1.Width) > TestGrid.Width)  {  leftMargin = TestGrid.Width - testImage1.Width;  testFlag1X = true;  } |

Add code to the following projects:

* ***AlgorithmsStage1 -*** Build your basic solution in this project.
* ***AlgorithmsStage2 -*** Refine your solution in this project as a method.
* ***AlgorithmsStage3 -*** Extend your solution by adding to a class.

Now comment your code….

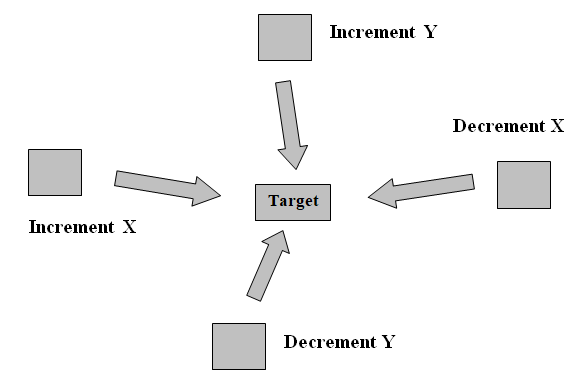
***Problem - Follow an Object***



One of the main features of flocking is a follow algorithm.

Here the object needs to move to a target object.

To achieve this we need to ensure the following:

* Find the location of the object
* When the object is left of the target move right
* When the object is right of the target move left
* When the object is above of the target move down
* When the object is below of the target move up
* Reset the location of the object

|  |
| --- |
| Add to the ***MainPage.xaml*** file of the projects ***AlgorithmsStage1***, ***AlgorithmsStage2*** and ***AlgorithmsStage3*** |
| <Image x:Name="testImage2" HorizontalAlignment="Left" VerticalAlignment="Top" Height="40"  Margin="50,50,0,0" Width="40" Source="Picture1.png" Stretch="Fill" />    <Image x:Name="testImage3" HorizontalAlignment="Left" VerticalAlignment="Top" Height="40"  Margin="100,350,0,0" Width="40" Source="Picture1.png" Stretch="Fill" /> |

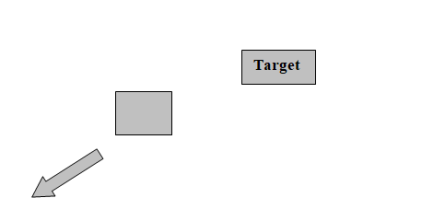
|  |
| --- |
| Try moving two images |
| Follow(testImage2, testImage1, 1.50);  Follow(testImage3, testImage1, 1.25); |

Add code to the following projects:

* ***AlgorithmsStage1 -*** Build your basic solution in this project.
* ***AlgorithmsStage2 -*** Refine your solution in this project as a method.
* ***AlgorithmsStage3 -*** Extend your solution by adding to a class.

Now comment your code….

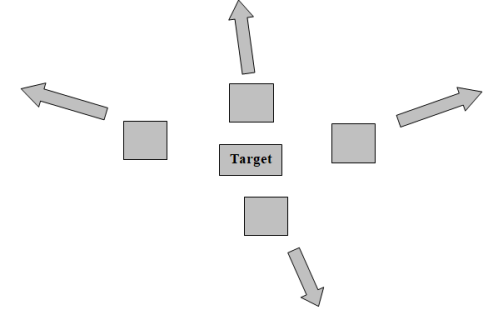
***Problem - Runaway from an Object***



Another main feature of flocking is a runaway algorithm.

Here the object needs to move away from a target object.

To achieve this we need to ensure the following:

* Find the location of the object
* When the object is left of the target move further left
* When the object is right of the target move further right
* When the object is above of the target move further up
* When the object is below of the target move further down
* Reset the location of the object

|  |
| --- |
| Add to the ***MainPage.xaml*** file of the projects ***AlgorithmsStage1***, ***AlgorithmsStage2*** and ***AlgorithmsStage3*** |
| <Image x:Name="testImage4" HorizontalAlignment="Left" VerticalAlignment="Top" Height="40"  Margin="400,200,0,0" Width="40" Source="Picture1.png" Stretch="Fill" /> |

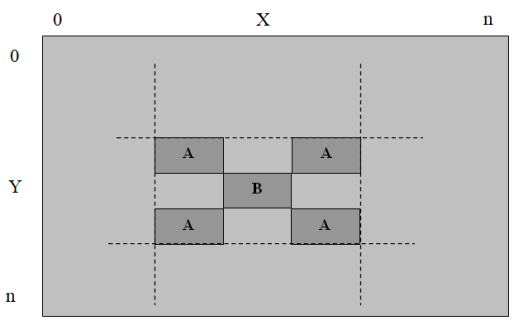
|  |
| --- |
| Try locking the image to the ggrid |
| Runaway(testImage4, testImage1, 0.50);  Lock\_To\_Grid(testImage4, TestGrid); |

Add code to the following projects:

* ***AlgorithmsStage1 -*** Build your basic solution in this project.
* ***AlgorithmsStage2 -*** Refine your solution in this project as a method.
* ***AlgorithmsStage3 -*** Extend your solution by adding to a class.

Now comment your code….

***Problem - Collide with an Object***



Collision between two rectangular objects is the next algorithm we will look at.

We use this type of collision detection to recognize objects hitting each other.

To recognize a collision between two rectangular objects (Object A and Object B) we need to ensure that if any part of Object A collides with any part of Object B we have a hit.

To achieve this we need to ensure the following:

* If the image left plus the image width is greater than the target image left
* And the image left is less than the target image left plus the target image width
* And the image top plus the image height is greater than the target image top
* And the image top is less than the target image top plus the target image height
* Then we have a collision

Add code to the following projects:

* ***AlgorithmsStage1 -*** Build your basic solution in this project.
* ***AlgorithmsStage2 -*** Refine your solution in this project as a method.
* ***AlgorithmsStage3 -*** Extend your solution by adding to a class.

Now comment your code….

***Generate a Random Number***

We now have a set of methods that can be used to model flocking behavior.

The only element missing is the addition of some randomness into our system.

Built into the C# library is a pseudo-random number generator – an algorithm that produces a sequence of numbers that meet a basic statistical requirement for randomness.

Setting this up is a 3 phase process…

First we set up a random number object.

***Random number;***

Then we instantiate the random number generator by using the current time as a seed.

***number = new Random(DateTime.Now.Millisecond);***

Then we grab the next number generated in the sequence by calling the Next method.

***int test = number.Next(1,21);***

This example would return a number between 1 and 20.

***Extension Exercises***

1. Modify your methods to make them more specific to your requirements.
2. Extend the following method to allow the object to follow based on just the position of the top / left corner.

public static void Follow(Image anImage, Image targetImage, double speed)

1. Extend the following method to allow - on a collision the objects move apart and reset their size and opacity (eat).

public static void Collide(Image anImage, Image targetImage, double topSpeed)

The following is required:

* Checks for a collision
* Pushes the object away from the target
* Increases the object size
* Decreases the object opacity
* Decreases the target size
* Increases the target opacity
* Locks the max / min size of the objects
* Locks the max / min opacity of the objects

1. Extend the *Runaway from an Object*algorithm to only work based on the object size – if it is too big it will not run away.