Module: **Multimedia and Interactivity**
Year: 2
Lecturer: Maxwell Robertson
Email: maxwell.robertson@cumbria.ac.uk
Session: 13 - Imaging Lingo

CUMBRIA Institute of the Arts

© Copyright 2006 (All Rights Reserved.)

The lingo code here is licensed for use by Maxwell Robertson and is based on copyrighted work for RT/1 ©1986-2003 and interactive installations ©1986-2003. No unauthorized duplication is allowed.

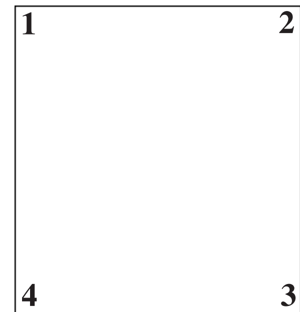
Up until now we have worked the graphics in Director in a fairly straight forward way. This meant working with `the rectangle`, `the width`, `the height`, `the locH` and `the locV` of a particular graphic. One option that has not been explored is `the quad`. This is similar to a rectangle. A rectangle is defined by numbers for the left, top, right and bottom edges. A Quad is also made up of four items, each item is a point and defines each corner. If you have a graphic on the screen in **channel** 1, then in the **message** window type:

```
put sprite(1).quad
```

you will see something like this appear:

```
[point(50.0000, 0.0000), point(270.0000, 0.0000),
     point(270.0000, 240.0000), point(50.0000, 240.0000)]
```

These point define the corners starting at the top / left and working clockwise around the graphic. Each of these points can be queried in turn by accessing it specifically. So query `point 3` you would access it as follows:




```
put sprite(1).quad[3]
```

this will return:

```
point(270.0000, 240.0000)
```

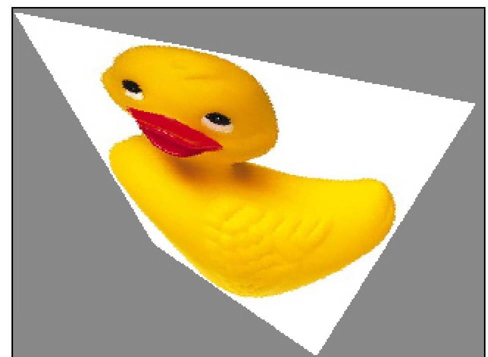
Each point is made up of a horizontal and vertical position. Each part of these points can be accessed by using either `the locH` or `the locV` as follows;

```
put sprite(1).quad[3].locH
```

this will return:

```
270.0000
```

Using this capability you can programmatically alter the shape of a graphic, so that it is irregular. This following demonstration will show you how to do that.



Brampton Road Carlisle Cumbria England UK CA3 9AY Telephone +44(0)1228 400300 Facsimile +44(0)1228 514491
E.mail recept@cumbria.ac.uk Internet http://www.cumbria.ac.uk

1/ Create a new movie and import a graphic. For this example, it will assume the stage is 320*240 and the graphic is 220*240. So you can resize the graphic to match the later size using “`Modify->Transform Bitmap`”.

2/ Change the **stage** colour to grey and then center the image on the **stage** in **channel** 1.

3/ Create a **movie script** with a `startMovie` handler as follows:

```
on startMovie
     global gQuad
     sprite(1).puppet = true
     --store the quad to be able to reset the graphic
     gQuad = sprite(1).quad
end startMovie
```

We will then create a `stopMovie` handler as follows:

```
on stopMovie
     global gQuad
     sprite(1).quad = gQuad
     updateStage
     sprite(1).puppet = false
end stopMovie
```

When the **movie** stops playing, this handler will be called and `the quad` will reset to its starting position.

4/ On a **score frame script** create a **script** that loops on the **frame**:

```
on exitFrame
     go the frame
end
```

5/ Select the graphic in the **score** and assign a new **behavior**:

```
on mouseDown me
     Squished me
end
```

(***Note***: We have changed it to a `mouseDown` not a `mouseUp`.)

We just need to create command called `Squished`.

Open the **movie script** that contains the `startMovie` and `stopMovie` handlers and create a new handler called `Squished` as follows:

```
on Squished me
     --figure out which sprite is clicked
     curSprite = me.spriteNum
     --get the quad for the sprite
     checkQuad = sprite(curSprite).quad
     --set a variable to contain which point is clicked
     checkPt = 0
     --step through the four points
     repeat with x =  1 to 4
          --get the point
          apt = sprite(curSprite).quad[x]
          --see if mouse is within 5 pixels of the corner
          if (((apt.locH - 5) < the mouseH) and \
          ((apt.locH + 5) > the mouseH)) and \
          (((apt.locV - 5) < the mouseV) and \
          ((apt.locV + 5) > the mouseV)) then
               --if it is store the point
               checkPt = x
               --exit the repeat loop
               exit repeat
          end if
     end repeat
     --if no point is selected then exit the command
     if checkPt = 0 then
          exit
     end if
     --update the quad to match mouse location
     repeat while the mouseDown = true
          --change the corner location we have clicked on
          checkQuad[checkPt].locH = the mouseH
          checkQuad[checkPt].locV = the mouseV
          --assign the quad back to the sprite
          sprite(curSprite).quad = checkQuad
          updateStage
     end repeat
end Squished
```

(***Note***: The comments should explain the code.)

You should be able to play the **movie** now and click on the image corners and drag them around the screen.

> ***Warning***: Manipulating the **quads** physically changes the graphic, so you must remember to change them back to the original, if that is required.

Undocumented in Director 8 is a set of commands that come under the label of imaging lingo. These commands allow you to manipulate the actual graphics that are stored in the **casts**. The commands that are available are as follows:

```
the disableImagingTransformation
image(width, height, bitDepth, <alphaDepth>, <paletteMember>)
member(whichMember).image
(the stage).image
window("whichWindow").image
(image).depth
(image).height
(image).rect
(image).width
(image).useAlpha
(image).copyPixels(sourceImage, destRect_or_quad, sourceRect,
<paramList>)
```

`<paramList>` can contain any or all these items:

```
#color, #bgColor, #ink, #blendLevel, #dither, #useFastQuads,
#maskImage, #maskOffset
```

```
(image).createMask()
(image).createMatte(<alphaThreshold>)
(image).crop(cropRectangle)
(image).draw(left, top, right, bottom, color, <parameters>)
```

`<parameters>` can contain any or all these items:

```
#shapeType (Options: #oval, #rect, #roundRect, or #line.
Default is #line.), #lineSize, #color
```

```
(image).duplicate()
(image).extractAlpha()
(image).fill(left, top, right, bottom, color, <parameters>)
```

`<parameters>` can contain any or all these items:

```
#shapeType (Options: #oval, #rect, #roundRect, or #line.
Default is #line.), #lineSize, #color, #bgColor
```

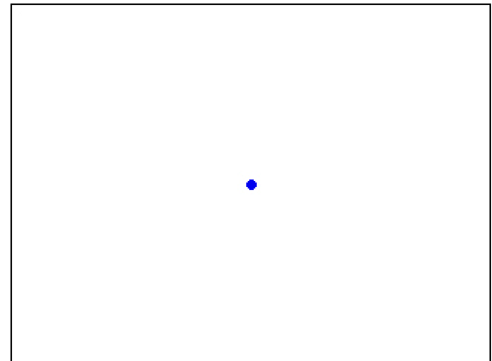
```
(image).getPixel(x,y)
(image).setAlpha(alphaImage)
(image).setPixel(x,y,color)
(image).trimWhiteSpace()
```

(***Note***: The commands are all documented in Director 8.5 and MX. If you need more information about them. Several of the more obvious ones will be illustrated here)

These commands allow you to create and manipulate graphics in many different ways. This example will show you how to distort and image using some simple mathematics.

Firstly we are going to create an animation where an object moves across the screen in a curved path.

1/ Create a small graphic and place it on the screen in **channel** 1.



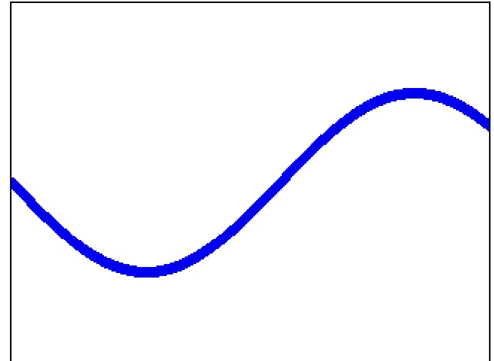
2/ Create a **movie script** with a `startMovie` handler as follows:

```
on startMovie
     global gLocH
     sprite(1).puppet = true
     gLocH = 0
end startMovie
```

3/ On a **score frame script** create a **script** that as follows:

```
on exitFrame me
     global gLocH
     gLocH = gLocH + 1
     --make sure that object resets when it
     --goes off the screen
     if gLocH > 320 then
          gLocH = 0
     end if
     curveHeight = 60
     curveFrequency = 1
     --set the h location of the sprite
     sprite(1).locH = gLocH
     --figure out v location of the sprite
     --use h location and frequency, then covert to radians
     curveValue = (gLocH * curveFrequency ) / 57.1919
     --multiple the sin value (1 to -1) by the
     --curveHeight we want
     --the add 120 to move it half way down the
     --320 * 240 stage
     sprite(1).locV = 120 + (curveHeight * sin(curveValue))
     --draw the image
     updateStage
     go the frame
end
```

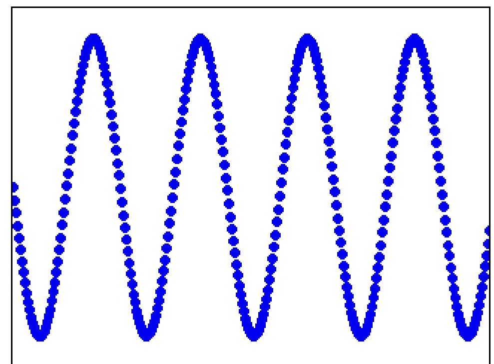
If all is correct, then when you play the movie, you should see the object move on a path similar to the one shown here.



Changing the `curveHeight` will result in a steeper or shallower curve.

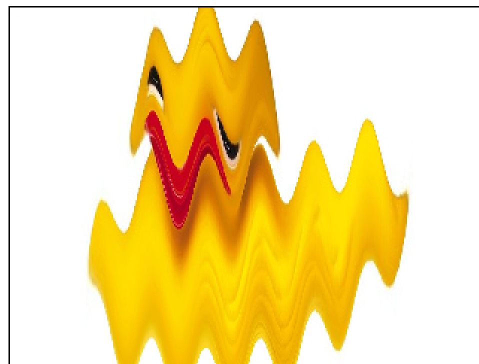
Changing the `curveFrequency` will result in more or less troughs and peaks.

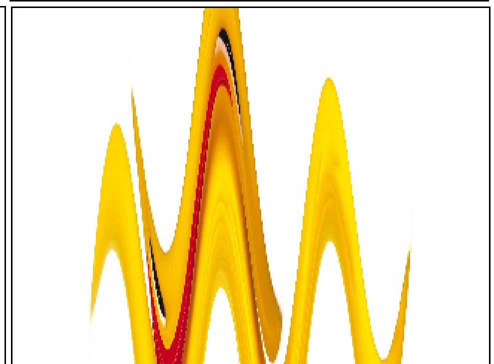
This example used **100** as the `curveHeight` and **5** as the `curveFrequency`.



Bearing this in mind what would happen if you applied this kind of path to an image. If you sliced an image up into 1 pixel wide slivers and then place the centre of the sliver on the path, you will have results like this. (***Note***: Duck is the starting image.)

The image on the left has a height of 20 and a frequency of 10. The image on the right has a height of 60 and a frequency of 5.





This effect is done
using imaging lingo commands. The following example shows you how.

1/ Create a new **movie**, the **stage** is 320 * 240.

2/ Place the graphic on the **stage**. The duck image is 220 * 240, so that it fits on the **stage**.

3/ In the cast call the Duck image “DuckMess”.

4/ Duplicate the Duck cast and call it “Ducky”.

5/ Create a **movie script** with a `startMovie` handler as follows:

```
on startMovie
     sprite(1).puppet = true
end startMovie
```

Imaging lingo modifies the **cast** graphics like the **Quad** command. As a result you need to reset the image back to normal after you are finished or else the changed image will be saved with the file.

6/ We create a `stopMovie` handler that will reset the graphic:

```
on stopMovie
     --copy the good duck image
     myImage = duplicate(member(“ducky”).image)
     --replace the bad image with the good copy
     member(“duckMess”).image = duplicate(myImage)
     --set the container to empty to release the memory
     myImage = empty
end stopMovie
```

This introduces you to the first imaging **lingo** command: `duplicate`. This makes a copy of an item. Here we are using “`.image`”, so we a copying the image to the container called `myImage`. The commands after that should be self explanatory.

7/ On a **score frame script** create a **script** that as follows:

```
on exitFrame me
     newHeight = 20
     SinCurveImage newHeight
     go the frame
end
```

`newHeight` is going to be used to define the height of the curve we are going to use. `SinCurveImage` is the command we are going to create to actually manipulate the image. `newHeight` is then passed in as a variable.

8/ We are now going to create the `SinCurveImage` command, which will be added to the movie script containing the `startMovie` and `stopMovie` scripts:

```
on SinCurveImage curveHeight
     curveFrequency = 5
     --copy the duck image that we are going to mess up
     --this is so we have the correct size
     myImage = duplicate(member("ducky").image)
     --fill the copy with white
     myImage.fill(rect(0, 0, 220, 240), rgb(255, 255, 255))
     --image is 220 pixels wide, so we are going to have
     --to process 220 lines, since they will be 1 pixel wide
     repeat with x = 1 to 220
          --create a rectangle that is 1 pixel wide and
          --240 pixels tall
          srcRect = Rect(x, 0, x + 1, 240)
          --we now need to figure out how far the rectangle
          --is going to move, to line up on the curve
          hDist = 0
          --figure out the angle
          vDist = sin((x * curveFrequency ) / 57.1919)
          --multiple it by the height
          vDist = vDist * curveHeight
          --we need to offset the distance by the half
          --the height so that the graphics stay in centre
          vDist = vDist + integer((curveHeight / 2) * -1)
          --use the two values to offset the rectangle
          destRect = srcRect.offset(hDist, vDist)
          --use the src and dest rects to move the graphics
myImage.copyPixels(member("ducky").image, srcRect, destRect)
     end repeat
     --copy the new image to the cast,
     --so that we can see it
     member("DuckyMess").image = myImage
     --empty the container, to free up the memory
     myImage = empty
end SinCurveImage
```

When you play the movie you should now see your image be distorted. When you stop the movie, it should reset back to normal as well.

Commands that are illustrated here are:

`fill` - this fills an area with a colour. Here we fill a rectangle the size of the image with white.

`offset` - This is not a imaging command. However it takes a rectangle and moves it horizontally and vertically a set number of pixels.

`copyPixels` - This copies a graphic or part of it from one location to another. This can be in the same image or as here, between different images. It uses source and destination rectangles also. Here they are both 1 pixel wide.

We can now make this whole process interactive by allowing the movement of the mouse to control the height of the curve. Edit the exitFrame script and change:

```
on exitFrame me
     newHeight = 20
     SinCurveImage newHeight
     go the frame
end
```

to

```
on exitFrame me
     newHeight = the mouseH / 6
     SinCurveImage newHeight
     go the frame
end
```

When you play the movie now, you should see the object change as you move the mouse.

Thefollow code is not explained here but affects the image in the horizontal and vertical at the same time. To make it work you need to duplicate the “Ducky” cast member once more and rename it “DuckyBU”.

The `exitFrame` code is as follows:

```
on exitFrame me
     newHeight = the mouseH / 6
     newWidth = the mouseV / 6
     SinCosCurveImage newHeight, newWidth
     go the frame
end
```

We track both the horizontal and vertical and pass the values in as variables to a new command called SinCosCurveImage. This command contains the following code:

```
on SinCosCurveImage curveHeight, curveWidth
     curveFrequency = 5
```

```
        myImage = duplicate(member("ducky").image)
        myImage.fill(rect(0, 0, 220, 240), rgb(255, 255, 255))
        repeat with x = 1 to 220
             --create a rectangle that is 1 pixel wide and
             --240 pixels tall
             srcRect = Rect(x, 0, x + 1, 240)
             --we now need to figure out how far the rectangle
             --is going to move, to line up on the curve
             hDist = 0
             --figure out the angle
             vDist = sin((x * curveFrequency ) / 57.1919)
             --multiple it by the height
             vDist = vDist * curveHeight
             --we need to offset the distance by the half
             --the height so that the graphics stay in centre
             vDist = vDist + integer((curveHeight / 2) * -1)
             --use the two values to offset the rectangle
             destRect = srcRect.offset(hDist, vDist)
             --use the src and dest rects to move the graphics
     myImage.copyPixels(member("ducky").image, srcRect, destRect)
        end repeat
        --copy the image to a new cast member called "DuckyBU"
        member("DuckyBU").image = duplicate(myImage)
        --clear the current container to white
        myImage.fill(rect(0, 0, 220, 240), rgb(255, 255, 255))
        --now we are going to move the image  horizontally
        repeat with x = 1 to 240
             srcRect = Rect(0, x, 220, x + 1)
             vDist = 0
             hDist = cos((x * curveFrequency) / 57.1919)
             --multiple it by the height
             hDist = hDist * curveWidth
             --we need to offset the distance by the half
             --the height so that the graphics stay in centre
             hDist = hDist + integer((curveWidth / 2) * -1)
             --use the two values to offset the rectangle
             destRect = srcRect.offset(hDist, vDist)
             --use the src and dest rects to move the graphics
myImage.copyPixels(member("DuckyBU").image, srcRect, destRect)
        end repeat
        --copy the new image to the the cast,
        --so that we can see it
        member("DuckyMess").image = myImage
        --empty the container, to free up the memory
        myImage = empty
     end SinCosCurveImage
```

The `stopMovie` script needs to be modified as follows:

```
on stopMovie
     myImage = duplicate(member("Ducky").image)
     member("DuckyBU").image = duplicate(myImage)
     member("DuckyMess").image = duplicate(myImage)
     myImage = empty
end stopMovie
```

When you play the **movie** now, you should find that you can distort the image both horizontally and vertically, depending how you move the mouse.

End of **Session #13**.