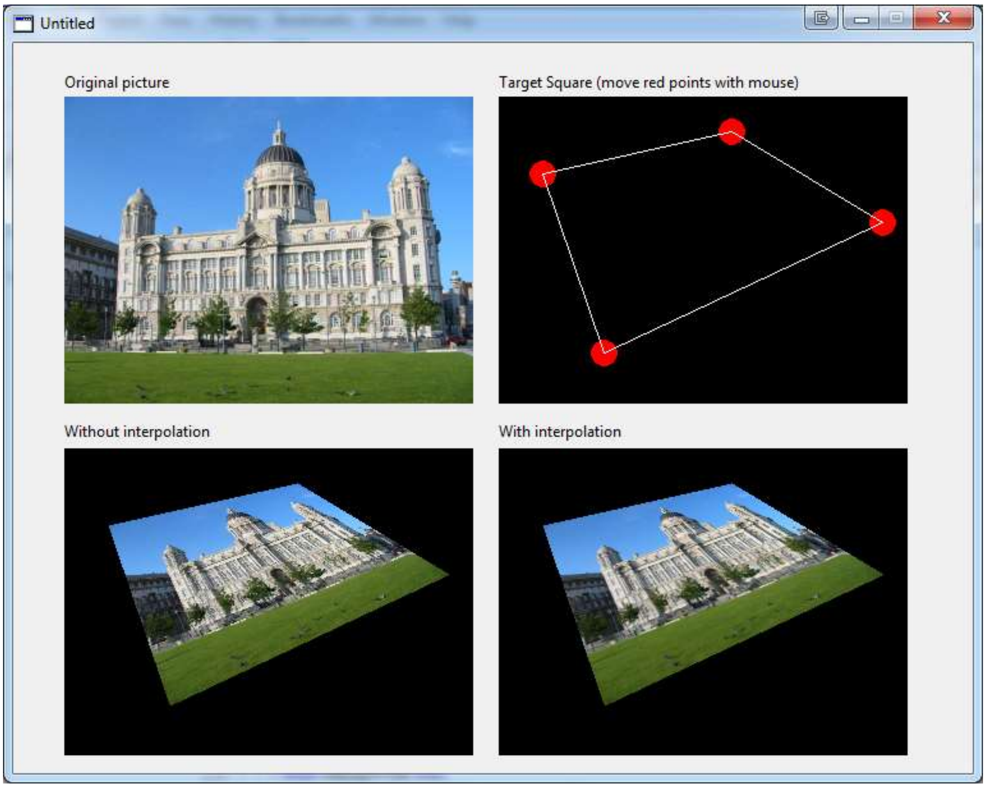
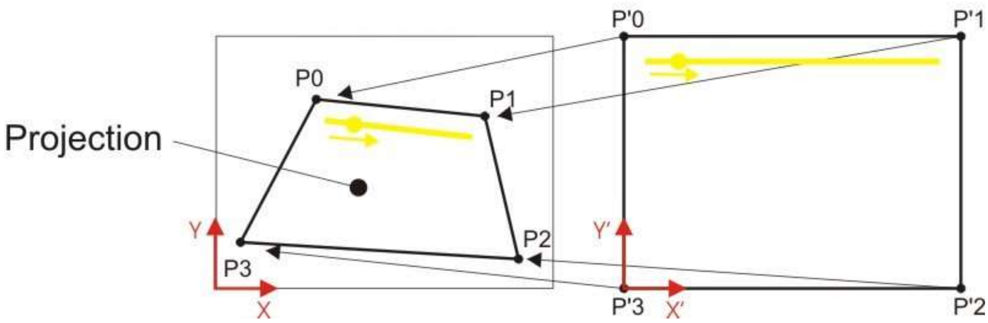
Realbasic: Canvas Tutorial Lesson 10-Perspective



Last week I had a question on how to do perspective on images in RealBasic. As the answer is not that easy, I decided to make a blog post on this.

First, let's look at what actually happens when a rectangle is put into perspective. We have to map the four corners of the picture to four points in the '3D' space as shown in this illustration:



This gives us a framework to map every other point in the picture to its respective point in the 3D world.

The way to do such a mapping is using a technique called *BackwardQuadrilateralTransformation*. Could be the name of something out of Star Trek :-)

But this sounds more difficult than it is. The idea of the algorithm is based on homogeneous transformation and its math is described by Paul Heckbert in his
paper. Here is a link for the ones who like to read more on this: http://graphics.cs.cmu.edu/courses/15-463/2008_fall/Papers/proj.pdf

Ready to enter the Matrix? Let's dive into the code!

First we'll need a class ABPoint to hold a vector:

```
x as integer
y as integer

Sub Constructor(x as integer, y as integer)
  me.x = x
  me.y = y
End Sub
```

Me make a module mPerspective that will hold the code to convert a picture from 2D to 3D space.

The main function is ABBackwardQuadrilateralTransformation. As parameters it takes the source picture, a table containing the four destination corners in the '3D' space, if we want interpolation and what the backcolor of the new picture should be.

The four destination points have to be added in a clockwise order. So P0 -> P1 -> P2 -> P3.
The interpolation parameter can be used so the transformation is more smooth, but it also means it takes more time to do the conversion.

```
Function ABBackwardQuadrilateralTransformation(srcPic as picture,
destinationQuadrilateral() as ABPoint, useInterpolation  as boolean, FillBackColor as
Color) As picture
```

I'll go a little more over some parts of this function. The full function can be found in the project at the end of this article.

Getting the bounds of the rectangle. What it simply does is getting the 4 most ubound points within a group of points. In our case we only have four of them but this function could find them even if you have a lot of points.

```
  ...
  'get bounding rectangle of the quadrilateral
  GetBoundingRectangle destinationQuadrilateral, minXY, maxXY

  dim startX as integer = minXY.X
  dim startY as integer = minXY.Y
  dim stopX as integer = maxXY.X
  dim stopY as integer = maxXY.Y
  ...
```

Here is the function:

```
Private Sub GetBoundingRectangle(cloud() as ABPoint, byref minXY as ABPoint, byref
maxXY as ABPoint)
  dim minX as integer = 10e6
  dim maxX as integer = -10e6
  dim minY as integer = 10e6
  dim maxY as integer = -10e6

  dim i as integer
  for i = 0 to UBound(cloud)
    if cloud(i).x &lt; minX then minX = cloud(i).x
```

```
    if cloud(i).x &gt; maxX then maxX = cloud(i).x
    if cloud(i).y &lt; minY then minY = cloud(i).y
    if cloud(i).y &gt; maxY then maxY = cloud(i).y
  next

  minXY = new ABPoint(minX, minY)
  maxXY = new ABPoint(maxX, maxY)
End Sub
```

Next, we'll need to calculate the transformation matrix. This can be done with the MapQuadToQuad() function and here is where the magic happens. You'll notice there are two functions named MapQuadToQuad but one of them is just a help function for the other one.

The MapQuadToQuad() function will make our matrix given two rectangles. We also need some help functions to multiply two 3x3 matrixes, to calculate the adjugate of a 3x3 matrix and one to calculate the determinant of a 2x2 matrix.

If this sounds like gibberish to you, I'll suggest you google around and read some math tutorials. Don't worry, It's all very basic.

```
  ...
  'calculate tranformation matrix
  dim srcRect(3) as ABPoint
  srcRect(0) = new ABPoint(0,0)
  srcRect(1) = new ABPoint(srcWidth -1 ,0)
  srcRect(2) = new ABPoint(srcWidth - 1, srcHeight - 1)
  srcRect(3) = new ABPoint(0, srcHeight - 1)
  dim matrix(2,2) as Double = MapQuadToQuad(destinationQuadrilateral, srcRect)
  ...
```

Here is are the functions:

```
Private Function MapQuadToQuad(input() as ABPoint, output() as ABPoint) As double(,)
  Dim squareToInput(2,2) as Double = MapQuadToQuad(input)
  Dim squareToOutput(2,2) as Double = MapQuadToQuad(output)

  Return MultiplyMatrix(squareToOutput, AdjugateMatrix(squareToInput))
End Function

Private Function MapQuadToQuad(Quad() as ABPoint) As double(,)
  dim sq(2,2) as double
  dim px, py as Double

  dim TOLERANCE as double = 1e-13

  px = quad(0).X - quad(1).X + quad(2).X - quad(3).X
  py = quad(0).Y - quad(1).Y + quad(2).Y - quad(3).Y

  if ( ( px &lt; TOLERANCE ) And ( px &gt; -TOLERANCE ) And ( py &lt; TOLERANCE ) And (
py &gt; -TOLERANCE ) ) then
    sq(0, 0) = quad(1).X - quad(0).X
    sq(0, 1) = quad(2).X - quad(1).X
    sq(0, 2) = quad(0).X

    sq(1, 0) = quad(1).Y - quad(0).Y
    sq(1, 1) = quad(2).Y - quad(1).Y
    sq(1, 2) = quad(0).Y

    sq(2, 0) = 0.0
    sq(2, 1) = 0.0
```

```
    sq(2, 2) = 1.0
  else

    dim dx1, dx2, dy1, dy2, del as Double

    dx1 = quad(1).X - quad(2).X
    dx2 = quad(3).X - quad(2).X
    dy1 = quad(1).Y - quad(2).Y
    dy2 = quad(3).Y - quad(2).Y

    del = Det2( dx1, dx2, dy1, dy2 )

    if ( del = 0 ) then
      return sq
    end if

    sq(2, 0) = Det2( px, dx2, py, dy2 ) / del
    sq(2, 1) = Det2( dx1, px, dy1, py ) / del
    sq(2, 2) = 1.0

    sq(0, 0) = quad(1).X - quad(0).X + sq(2, 0) * quad(1).X
    sq(0, 1) = quad(3).X - quad(0).X + sq(2, 1) * quad(3).X
    sq(0, 2) = quad(0).X

    sq(1, 0) = quad(1).Y - quad(0).Y + sq(2, 0) * quad(1).Y
    sq(1, 1) = quad(3).Y - quad(0).Y + sq(2, 1) * quad(3).Y
    sq(1, 2) = quad(0).Y
  end if

  return sq
End Function

Private Function MultiplyMatrix(a(,) as double, b(,) as double) As double(,)
  ' Multiply two 3x3 matrices
  dim c (2,2) as Double

  c(0, 0) = a(0, 0) * b(0, 0) + a(0, 1) * b(1, 0) + a(0, 2) * b(2, 0)
  c(0, 1) = a(0, 0) * b(0, 1) + a(0, 1) * b(1, 1) + a(0, 2) * b(2, 1)
  c(0, 2) = a(0, 0) * b(0, 2) + a(0, 1) * b(1, 2) + a(0, 2) * b(2, 2)
  c(1, 0) = a(1, 0) * b(0, 0) + a(1, 1) * b(1, 0) + a(1, 2) * b(2, 0)
  c(1, 1) = a(1, 0) * b(0, 1) + a(1, 1) * b(1, 1) + a(1, 2) * b(2, 1)
  c(1, 2) = a(1, 0) * b(0, 2) + a(1, 1) * b(1, 2) + a(1, 2) * b(2, 2)
  c(2, 0) = a(2, 0) * b(0, 0) + a(2, 1) * b(1, 0) + a(2, 2) * b(2, 0)
  c(2, 1) = a(2, 0) * b(0, 1) + a(2, 1) * b(1, 1) + a(2, 2) * b(2, 1)
  c(2, 2) = a(2, 0) * b(0, 2) + a(2, 1) * b(1, 2) + a(2, 2) * b(2, 2)

  return c
End Function

Private Function AdjugateMatrix(a(,) as double) As double(,)
  ' Calculates adjugate 3x3 matrix
  dim b(2,2) as double
  b(0, 0) = Det2( a(1, 1), a(1, 2), a(2, 1), a(2, 2) )
  b(1, 0) = Det2( a(1, 2), a(1, 0), a(2, 2), a(2, 0) )
  b(2, 0) = Det2( a(1, 0), a(1, 1), a(2, 0), a(2, 1) )
  b(0, 1) = Det2( a(2, 1), a(2, 2), a(0, 1), a(0, 2) )
  b(1, 1) = Det2( a(2, 2), a(2, 0), a(0, 2), a(0, 0) )
  b(2, 1) = Det2( a(2, 0), a(2, 1), a(0, 0), a(0, 1) )
  b(0, 2) = Det2( a(0, 1), a(0, 2), a(1, 1), a(1, 2) )
  b(1, 2) = Det2( a(0, 2), a(0, 0), a(1, 2), a(1, 0) )
  b(2, 2) = Det2( a(0, 0), a(0, 1), a(1, 0), a(1, 1) )

  return b
End Function

Private Function Det2(a as double, b as double, c as double, d as double) As double
  ' Calculates determinant of a 2x2 matrix
```

```
  return ( a * d - b * c )
End Function
```

Now we are ready to continue with our main function ABBackwardQuadrilateralTransformation() where we will manipulate the picture.
I worked out the two systems: with and without interpolation.

Basically what it does is map every pixel from the source rectangle to the target rectangle using the matrix we just created. When we use interpolation, we'll use the pixels around our pixel to calculate a new color that is the mix of all those colors. This smooths the picture a little.

```
  dim x,y as integer

  dim  factor, srcX, srcY as Double
  dim tgtPic as Picture
  tgtPic = NewPicture(srcWidth, srcHeight, 32)
  tgtPic.Graphics.ForeColor = FillBackColor
  tgtPic.Graphics.FillRect 0,0, srcWidth, srcHeight

  dim srcRGB, tgtRGB as RGBSurface
  srcRGB = srcPic.RGBSurface
  tgtRGB = tgtPic.RGBSurface

  if useInterpolation then
    Dim srcWidthM1 as integer = srcWidth - 1
    Dim srcHeightM1 as Integer = srcHeight - 1

    'coordinates of source points
    dim dx1, dy1, dx2, dy2 as Double
    dim sx1, sy1, sx2, sy2 as Integer

    ' temporary pixels
    dim p1,p2,p3, p4 as Color
    dim r, g , b as integer

    ' for each row
    for y = startY to stopY
      'for each pixel
      for x = startX to stopX
        factor = matrix(2, 0) * x + matrix(2, 1) * y + matrix(2, 2)
        srcX = ( matrix(0, 0) * x + matrix(0, 1) * y + matrix(0, 2) ) / factor
        srcY = ( matrix(1, 0) * x + matrix(1, 1) * y + matrix(1, 2) ) / factor
        if srcX >= 0 and srcY >= 0 and srcX< srcWidth and srcY < srcHeight then
          sx1 = srcX
          if sx1 = srcWidthM1 then
            sx2 = sx1
          else
            sx2 = sx1 + 1
          end if
          dx1 = srcX - sx1
          dx2 = 1.0 - dx1

          sy1 = srcY
          if sy1 = srcHeightM1 then
            sy2 = sy1
          else
            sy2 = sy1 + 1
          end if
          dy1 = srcY - sy1
          dy2 = 1.0 - dy1

          ' copy the pixel from the source to the target using interpolation of 4
points
```

```
          p1 = srcRGB.Pixel(sx1, sy1)
          p2 = srcRGB.Pixel(sx2, sy1)
          p3 = srcRGB.Pixel(sx1, sy2)
          p4 = srcRGB.Pixel(sx2, sy2)

          r = dy2 * ( dx2 * ( p1.red ) + dx1 * ( p2.red ) ) + dy1 * ( dx2 * ( p3.red )
+ dx1 * ( p4.red ) )
          g = dy2 * ( dx2 * ( p1.green ) + dx1 * ( p2.green ) ) + dy1 * ( dx2 * (
p3.green ) + dx1 * ( p4.green ) )
          b = dy2 * ( dx2 * ( p1.blue ) + dx1 * ( p2.blue ) ) + dy1 * ( dx2 * ( p3.blue
) + dx1 * ( p4.blue ) )
          tgtRGB.Pixel(x,y) = RGB(r,g,b)
        end if
      next
    next
  else
    ' for each row
    for y = startY to stopY
      'for each pixel
      for x = startX to stopX
        factor = matrix(2, 0) * x + matrix(2, 1) * y + matrix(2, 2)
        srcX = ( matrix(0, 0) * x + matrix(0, 1) * y + matrix(0, 2) ) / factor
        srcY = ( matrix(1, 0) * x + matrix(1, 1) * y + matrix(1, 2) ) / factor
        if srcX >= 0 and srcY >= 0 and srcX< srcWidth and srcY < srcHeight then
          ' copy the pixel from the source to the target
          tgtRGB.Pixel(x,y) = srcRGB.Pixel(srcX, srcY)
        end if
      next
    next
  end if

  Return tgtPic
```

And we're done! In the project you can download I added a system so you can easily change the four '3D' points and see the result of the transformation.

Until next time!