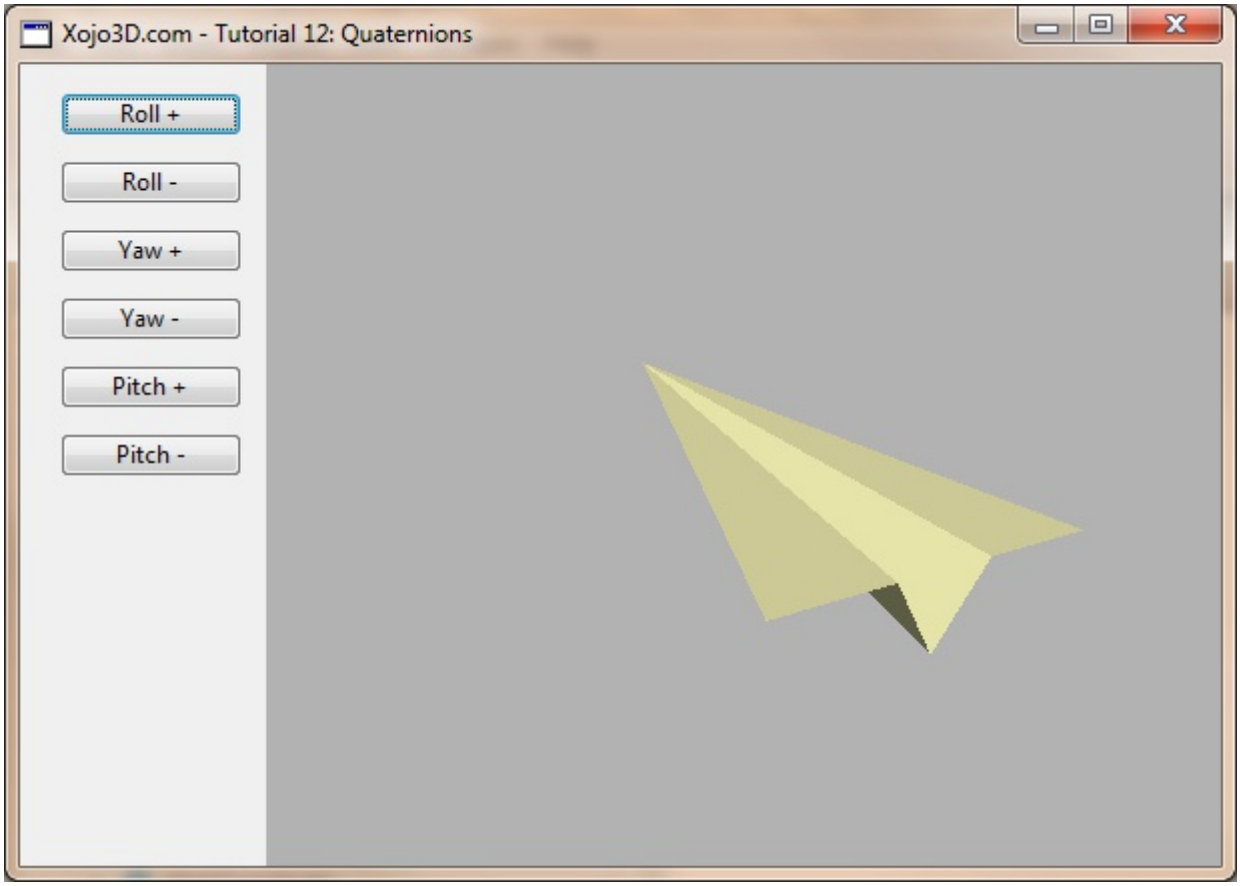
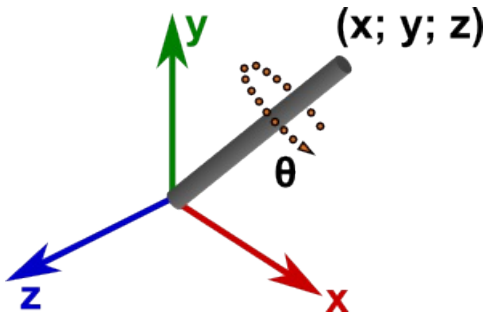
Tutorial 12: Quaternions

Gimbal lock is a common problem when applying successive rotations to a model in Euclidean space. In this tutorial we add rotation capabilities to your models, by making use of quaternions. Quaternions provide an elegant solution to avoid Gimbal locks.



Theory

Any possible rotation of an object in three-dimensional space can be represented by an axis and a rotation around this axis. The axis is represented by a vector (x; y; z), and the angle of rotation around the axis by a scalar value θ. With these four values, x, y, z and θ, we can represent any possible rotation in 3D space.



Quaternions provide a way to "encode" the above axis-angle representation into four numbers, and apply the rotation around a selected point in space. The advantages of quaternions are numerous and include the following:

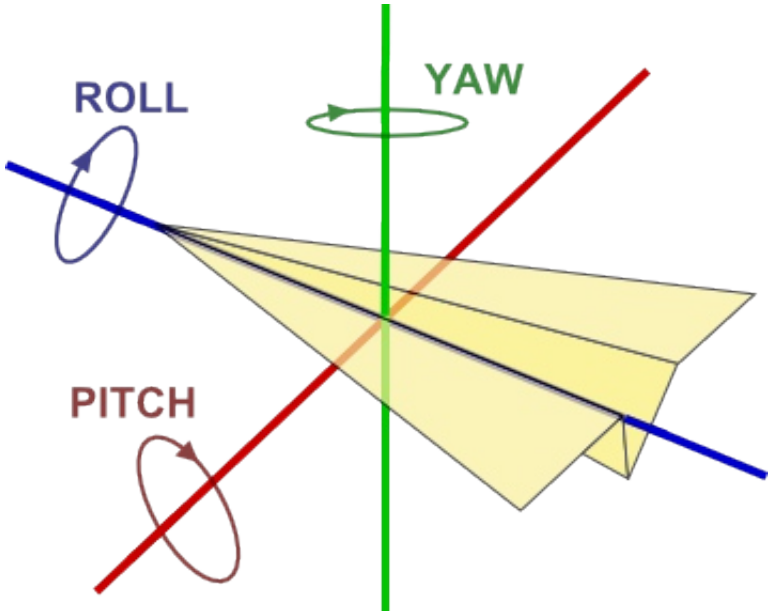
- Any possible rotation in 3D space can be represented easily with the four real values that make up a quaternion.
- It is easy to merge multiple successive rotations into a single rotation represented by a single quaternion, by multiplying quaternions.
- You don't have gimbal lock problems with quaternions.
- Once implemented properly, quaternions are very easy to work with.
- Fluent animations are easily composed by interpolating between quaternions.

The four values that make up a **quaternion** are better known as **x**, **y**, **z** and **w**. The following formulas are used to convert a Euclidean axis-angle pair, defined with vector (X, Y, Z) and angle θ respectively, into a quaternion:

$$\text{Quaternion X} = \text{Axis X} \times \sin(\theta / 2)$$
$$\text{Quaternion Y} = \text{Axis Y} \times \sin(\theta / 2)$$
$$\text{Quaternion Z} = \text{Axis Z} \times \sin(\theta / 2)$$
$$\text{Quaternion W} = \cos(\theta / 2)$$

An operation that is used often on quaternions, is **quaternion multiplication**. You effectively merge two successive rotations into a single rotation by multiplying two quaternions. The following formulas are used to calculate the product of two quaternions named Q_1 and Q_2:

$$\text{Result W} = Q_1W \times Q_2W - Q_1X \times Q_2X - Q_1Y \times Q_2Y - Q_1Z \times Q_2Z$$
$$\text{Result X} = Q_1W \times Q_2X + Q_1X \times Q_2W + Q_1Y \times Q_2Z - Q_1Z \times Q_2Y$$
$$\text{Result Y} = Q_1W \times Q_2Y - Q_1X \times Q_2Z + Q_1Y \times Q_2W + Q_1Z \times Q_2X$$
$$\text{Result Z} = Q_1W \times Q_2Z + Q_1X \times Q_2Y - Q_1Y \times Q_2X + Q_1Z \times Q_2W$$



We define three rotation actions when working with objects in three-dimensional space, **pitch**, **yaw** and **roll**, as shown below.

Pitch, yaw and roll rotations are now easily applied to existing quaternion by making use of quaternion multiplication.

Pitch rotate quaternion with θ degrees:

> If Q is the quaternion to rotate, and P the quaternion created from axis (1, 0, 0) and angle θ, then:
> **Pitch rotated Q = Q × P**

Yaw rotate quaternion with θ degrees:

> If Q is the quaternion to rotate, and Y the quaternion created from axis (0, 1, 0) and angle θ, then:
> **Yaw rotated Q = Q × Y**

Roll rotate quaternion with θ degrees:

> If Q is the quaternion to rotate, and R the quaternion created from axis (0, 0, 1) and angle θ, then:
> **Roll rotated Q = Q × R**

The multiplication technique shown above can be used to rotate a quaternion around any axis in 3D space, when more advanced rotations are needed than simple pitch, yaw and roll rotations.

Tutorial Steps

1. Open Xojo.
2. In the Project Chooser select Desktop.
3. Enter "Tutorial012" as the Application Name, and click OK.
4. Save your project.
5. Configure the following controls:

Control	Name	Left	Top	Caption	DoubleBuffer	Maximize Button
Window	SurfaceWindow	-	-	-	-	ON
OpenGLSurface	Surface	123	0	-	ON	-
Generic Button	RollPlusButton	20	14	Roll +	-	-
Generic Button	RollMinusButton	20	48	Roll -	-	-
Generic Button	YawPlusButton	20	82	Yaw +	-	-
Generic Button	YawMinusButton	20	116	Yaw -	-	-
Generic Button	PitchPlusButton	20	150	Pitch +	-	-
Generic Button	PitchMinusButton	20	184	Pitch -	-	-

6. Position and size Surface to fill the whole part of the window not occupied by the buttons, and set its Locking to left, top, bottom and right.

7. Add the following code to the *SurfaceWindow.Paint* event handler:

```
Surface.Render
```

8. Import the X3Core module, created in the previous tutorial.
   You can download the module from http://www.xojo3d.com/tutorials/tut012/x3core.zip.

9. Add the following code to the *Surface.Open* event handler:

```
X3_Initialize

X3_EnableLight OpenGL.GL_LIGHT0, new X3Core.X3Light(0, 0, 1)
```

10. Add the following code to the *Surface.Resized* event handler:

```
X3_SetPerspective Surface
```

11. Add the following constants to *X3Core*:

Name	Value	Type
X3_180OverPi	57.295779513082321	Number
X3_PiOver180	0.017453292519943	Number

12. Add a new class named "X3Quaternion" to module *X3Core*.
13. Add the following properties to *X3Quaternion*:

Name	Type
X	Double
Y	Double
Z	Double
W	Double

14. Add the following method to *X3Quaternion*.

```
Sub Constructor()
  W = 1
  X = 0
  Y = 0
  Z = 0
End Sub
```

15. Add the following method to *X3Quaternion*.

```
Sub Constructor(initW As Double, initX As Double, initY As Double,
initZ As Double)
  W = initW
  X = initX
  Y = initY
  Z = initZ
End Sub
```

16. Add the following method to *X3Quaternion*.

```
Sub Normalize()
  Dim m As Double
  m = Sqrt(x^2 + y^2 + z^2 + w^2)
  if m > 0 then
    w = w / m
    x = x / m
    y = y / m
    z = z / m
  end if
End Sub
```

17. Add the following method to *X3Quaternion*.

```
Sub Multiply(q As X3Quaternion)
  Dim resultX As Double
  Dim resultY As Double
  Dim resultZ As Double
  Dim resultW As Double

  resultW = (w * q.w) - (x * q.x) - (y * q.y) - (z * q.z)
  resultX = (w * q.x) + (x * q.w) + (y * q.z) - (z * q.y)
  resultY = (w * q.y) - (x * q.z) +(y * q.w) + (z * q.x)
  resultZ = (w * q.z) +(x * q.y) - (y * q.x) +(z * q.w)

  x = resultX
  y = resultY
  z = resultZ
  w = resultW
End Sub
```

18. Add the following method to *X3Quaternion*.

```
Sub FromEulerRotation(x As Double, y As Double, z As Double,
                      angle As Double)
  Dim halfAngle As Double
  Dim sinAng As Double

  halfAngle = (angle * X3_PiOver180) / 2
  sinAng = sin(halfAngle)

  Me.X = (x * sinAng)
  Me.Y = (y * sinAng)
  Me.Z = (z * sinAng)
  Me.W = cos(halfAngle)
End Sub
```

19. Add the following method to *X3Quaternion*.

```
Sub Pitch(angle As Double)
  Dim tmpQuat As new X3Quaternion
  tmpQuat.FromEulerRotation(1, 0, 0, angle)
  Multiply tmpQuat
  Normalize
End Sub
```

20. Add the following method to *X3Quaternion*.

```
Sub Yaw(angle As Double)
  Dim tmpQuat As new X3Quaternion
  tmpQuat.FromEulerRotation(0, 1, 0, angle)
  Multiply tmpQuat
  Normalize
End Sub
```

21. Add the following method to *X3Quaternion*.

```
Sub Roll(angle As Double)
  Dim tmpQuat As new X3Quaternion
  tmpQuat.FromEulerRotation(0, 0, 1, angle)
  Multiply tmpQuat
  Normalize
End Sub
```

22. Add the following method to *X3Vector*.

```
Sub Normalize()
  Dim m As Double
  m = Sqrt(x^2 + y^2 + z^2)
  if m > 0 then
    x = x / m
    y = y / m
    z = z / m
  end if
End Sub
```

23. Add the following property to *X3Model*:

Name	Type
Rotation	X3Quaternion

24. Add the following method to *X3Model*.

```
Sub Constructor()
  Rotation = new X3Quaternion()
End Sub
```

25. Add the following method to module *X3Core*:

```
Sub X3_SetRotation(rotation As X3Quaternion)
  Dim angle As Double
  Dim axis As new X3Vector(rotation.x, rotation.y, rotation.z)

  axis.Normalize
  angle = ACos(rotation.w) * 2 * X3_180OverPi

  OpenGL.glRotated angle, axis.x, axis.y, axis.z
End Sub
```

26. Add the following method to module *X3Core*:

```
Sub X3_RotateWithXY(q As X3Quaternion, xAngle As Double,
     yAngle As Double)
  Dim result As new X3Quaternion
  Dim tmpQuat As new X3Quaternion

  if xAngle <> 0 then
    tmpQuat.FromEulerRotation(1, 0, 0, xAngle)
    result.Multiply(tmpQuat)
  end if

  if yAngle <> 0 then
    tmpQuat.FromEulerRotation(0, 1, 0, yAngle)
    result.Multiply(tmpQuat)
  end if

  result.Multiply(q)
  result.Normalize

  q.W = result.W
  q.X = result.X
  q.Y = result.Y
  q.Z = result.Z
End Sub
```

27. Add the following method to *X3Polygon*:

```
Sub CalculateNormal()
  Dim v1X As Double
  Dim v1Y As Double
  Dim v1Z As Double
  Dim v2X As Double
  Dim v2Y As Double
  Dim v2Z As Double
  Dim cpX As Double
  Dim cpY As Double
  Dim cpZ As Double
  Dim m As Double

  v1X = Vertex(1).X - Vertex(0).X
  v1Y = Vertex(1).Y - Vertex(0).Y
  v1Z = Vertex(1).Z - Vertex(0).Z

// continue on next page
```

```
// continued from previous page
  v2X = Vertex(2).X - Vertex(1).X
  v2Y = Vertex(2).Y - Vertex(1).Y
  v2Z = Vertex(2).Z - Vertex(1).Z

  cpX = v1Y * v2Z - v1Z * v2Y
  cpY = v1Z * v2X - v1X * v2Z
  cpZ = v1X * v2Y - v1Y * v2X

  m = Sqrt(cpX^2 + cpY^2 + cpZ^2)

  Normal.X = cpX / m
  Normal.Y = cpY / m
  Normal.Z = cpZ / m
End Sub
```

28. Import the X3Test module into your project.

    You can download the module from http://www.xojo3d.com/tutorials/tut012/x3test.zip.

29. Add the following properties to *SurfaceWindow*:

Name	Type
Model	X3Core.X3Model
MousePrevX	Integer
MousePrevY	Integer

30. Add the following code to the *SurfaceWindow.Open* event handler:

```
Self.MouseCursor = System.Cursors.StandardPointer

Model = X3Test_Paperplane()
Model.Rotation.Pitch(20)
```

31. Add the following code to the *Surface.Render* event handler:

```
OpenGL.glClearColor(1, 1, 1, 1)
OpenGL.glClear(OpenGL.GL_COLOR_BUFFER_BIT +
OpenGL.GL_DEPTH_BUFFER_BIT)

OpenGL.glPushMatrix

OpenGL.glTranslatef 0, 0, -5

X3_RenderModel Model

OpenGL.glPopMatrix
```

32. Add the following code to the *Surface.MouseDown* event handler:

```
MousePrevX = x
MousePrevY = y

return true
```

33. Add the following code to the *Surface.MouseDrag* event handler:

```
X3_RotateWithXY Model.Rotation, (y - MousePrevY), (x - MousePrevX)

Surface.Render

MousePrevX = x
MousePrevY = y
```

34. Make the following changes to the *X3Core.X3_RenderModel* method:

```
' add the following two lines directly after the variable
declarations
' just before OpenGL.glBegin OpenGL.GL_TRIANGLES

OpenGL.glPushMatrix

X3_SetRotation(model.Rotation)

' add the following line to the end of the method
' just after OpenGL.glEnd

OpenGL.glPopMatrix
```

35. Add the following code to the *RollPlusButton.Action* event handler:

```
Model.Rotation.Roll(10)
Surface.Render
```

36. Add the following code to the *RollMinusButton.Action* event handler:

```
Model.Rotation.Roll(-10)
Surface.Render
```

37. Add the following code to the *YawPlusButton.Action* event handler:

```
Model.Rotation.Yaw(10)
Surface.Render
```

38. Add the following code to the *YawMinusButton.Action* event handler:

```
Model.Rotation.Yaw(-10)
Surface.Render
```

39. Add the following code to the *PitchPlusButton.Action* event handler:

```
Model.Rotation.Pitch(10)
Surface.Render
```

40. Add the following code to the *PitchMinusButton.Action* event handler:

```
Model.Rotation.Pitch(-10)
Surface.Render
```

41. Save and run your project. Drag the paperplane with your mouse to rotate it.

Analysis

X3Quaternion.Normalize:

```
Sub Normalize()
  Dim m As Double
  m = Sqrt(x^2 + y^2 + z^2 + w^2)
  if m > 0 then
    w = w / m
    x = x / m
    y = y / m
    z = z / m
  end if
End Sub
```

To create a **unit quaternion** (a quaternion with a length of 1), you need to normalize the quaternion. The Normalize() method does this by dividing each component of the quaternion with the magnitude of the quaternion.

X3Quaternion.Multiply:

```
Sub Multiply(q As X3Quaternion)
  Dim resultX As Double
  Dim resultY As Double
  Dim resultZ As Double
  Dim resultW As Double

  resultW = (w * q.w) - (x * q.x) - (y * q.y) - (z * q.z)
  resultX = (w * q.x) + (x * q.w) + (y * q.z) - (z * q.y)
  resultY = (w * q.y) - (x * q.z) +(y * q.w) + (z * q.x)
  resultZ = (w * q.z) +(x * q.y) - (y * q.x) +(z * q.w)

  x = resultX
  y = resultY
  z = resultZ
  w = resultW
End Sub
```

Quaternion multiplication is invaluable when it comes to applying successive rotations to a model.

The explanation of the mathematics used to multiply two quaternions is beyond the scope of this tutorial, but it is important to note that quaternion multiplication is non-commutative. This means that if Q and R are two quaternions, then Q×R will yield a different answer than R×Q. The order in which you multiply quaternions is, therefore, very important.

X3Quaternion.FromEulerRotation:

```
Sub FromEulerRotation(x As Double, y As Double, z As Double, angle As
Double)
  Dim halfAngle As Double
  Dim sinAng As Double

  halfAngle = (angle * X3_PiOver180) / 2
  sinAng = sin(halfAngle)

  Me.X = (x * sinAng)
  Me.Y = (y * sinAng)
  Me.Z = (z * sinAng)
  Me.W = cos(halfAngle)
End Sub
```

FromEulerRotation is a helper method that is used to transform a Eulaer axis-angle pair into a four-dimensional quaternion. Note that the angle is given in degrees. The explanation of the mathematics used to transform a Euler axis-angle pair into a quaternion is beyond the scope of this tutorial.

X3Quaternion.Pitch:

```
Sub Pitch(angle As Double)
  Dim tmpQuat As new X3Quaternion
  tmpQuat.FromEulerRotation(1, 0, 0, angle)
  Multiply tmpQuat
  Normalize
End Sub
```

The Pitch method, of the X3Quaternion class, applies a pitch rotation to an existing quaternion. First, we create a new pitch rotation quaternion, from the unit x-axis and a rotation applied around this axis. Then, we simply multiply the new rotation quaternion with the existing quaternion. Finally, we normalize the result to ensure that we end up with a unit quaternion.

X3Quaternion.Yaw:

```
Sub Yaw(angle As Double)
  Dim tmpQuat As new X3Quaternion
  tmpQuat.FromEulerRotation(0, 1, 0, angle)
  Multiply tmpQuat
  Normalize
End Sub
```

The Yaw method, of the X3Quaternion class, applies a yaw rotation to an existing quaternion. First, we create a new yaw rotation quaternion, from the unit y-axis and a rotation applied around this axis. Then, we simply multiply the new rotation quaternion with the existing quaternion. Finally, we normalize the result to ensure that we end up with a unit quaternion.

X3Quaternion.Roll:

```
Sub Roll(angle As Double)
  Dim tmpQuat As new X3Quaternion
  tmpQuat.FromEulerRotation(0, 1, 0, angle)
  Multiply tmpQuat
  Normalize
End Sub
```

The Roll method, of the X3Quaternion class, applies a roll rotation to an existing quaternion. First, we create a new roll rotation quaternion, from the unit z-axis and a rotation applied around this axis. Then, we simply multiply the new rotation quaternion with the existing quaternion. Finally, we normalize the result to ensure that we end up with a unit quaternion.

X3Vector.Normalize:

```
Sub Normalize()
  Dim m As Double
  m = Sqrt(x^2 + y^2 + z^2)
  if m > 0 then
    x = x / m
    y = y / m
    z = z / m
  end if
End Sub
```

To create a **unit vector** (a vector with a length of 1), you need to normalize the vector. The Normalize() method does this by dividing each component of the vector with the magnitude of the vector.

X3Core.X3_SetRotation:

```
Sub X3_SetRotation(rotation As X3Quaternion)
  Dim angle As Double
  Dim axis As new X3Vector(rotation.x, rotation.y, rotation.z)

  axis.Normalize
  angle = ACos(rotation.w) * 2 * X3_180OverPi

  OpenGL.glRotated angle, axis.x, axis.y, axis.z
End Sub
```

X3_SetRotation is a helper function, to apply the rotation defined by a quaternion, in the OpenGL environment.

X3Core.X3_RotateWithXY:

```
Sub X3_RotateWithXY(q As X3Quaternion, xAngle As Double, yAngle As
Double)
  Dim result As new X3Quaternion
  Dim tmpQuat As new X3Quaternion

  if xAngle <> 0 then
    tmpQuat.FromEulerRotation(1, 0, 0, xAngle)
    result.Multiply(tmpQuat)
  end if

  if yAngle <> 0 then
    tmpQuat.FromEulerRotation(0, 1, 0, yAngle)
    result.Multiply(tmpQuat)
  end if

  result.Multiply(q)
  result.Normalize

  q.W = result.W
  q.X = result.X
  q.Y = result.Y
  q.Z = result.Z
End Sub
```

Sometimes you might need to rotate a model when you only have the X and Y values as input, e.g. in a 3D editor when the user uses the mouse cursor to rotate a model. X3_RotateWithXY is a helper function, to rotate a quaternion when you only have the X and Y values available.

Sometimes you might need to rotate a model when you only have the X and Y values as input, e.g. in a 3D editor when the user uses the mouse cursor to rotate a model. X3_RotateWithXY is a helper function, to rotate a quaternion when you only have the X and Y values available.

X3Polygon.CalculateNormal:

```
Sub CalculateNormal()
  Dim v1X As Double
  Dim v1Y As Double
  Dim v1Z As Double
  Dim v2X As Double
  Dim v2Y As Double
  Dim v2Z As Double
  Dim cpX As Double
  Dim cpY As Double
  Dim cpZ As Double
  Dim m As Double

  v1X = Vertex(1).X - Vertex(0).X
  v1Y = Vertex(1).Y - Vertex(0).Y
  v1Z = Vertex(1).Z - Vertex(0).Z

  v2X = Vertex(2).X - Vertex(1).X
  v2Y = Vertex(2).Y - Vertex(1).Y
  v2Z = Vertex(2).Z - Vertex(1).Z

  cpX = v1Y * v2Z - v1Z * v2Y
  cpY = v1Z * v2X - v1X * v2Z
  cpZ = v1X * v2Y - v1Y * v2X

  m = Sqrt(cpX^2 + cpY^2 + cpZ^2)

  Normal.X = cpX / m
  Normal.Y = cpY / m
  Normal.Z = cpZ / m
End Sub
```

When you instantiate a new polygon with vertices, the CalculateNormal can be used to calculate the normal of this polygon using the vertices. It can also be called when the vertex data change, to re-calculate the new normal.

First, the two vectors that form two edges of the triangular polygon is determined, v1 and v2 respectively. Then, the cross product of these two vectors are calculated. The cross product is the vector that is perpendicular to the surface formed by v1 and v2, better known as the normal. Finally, we normalize the vector and store the results.