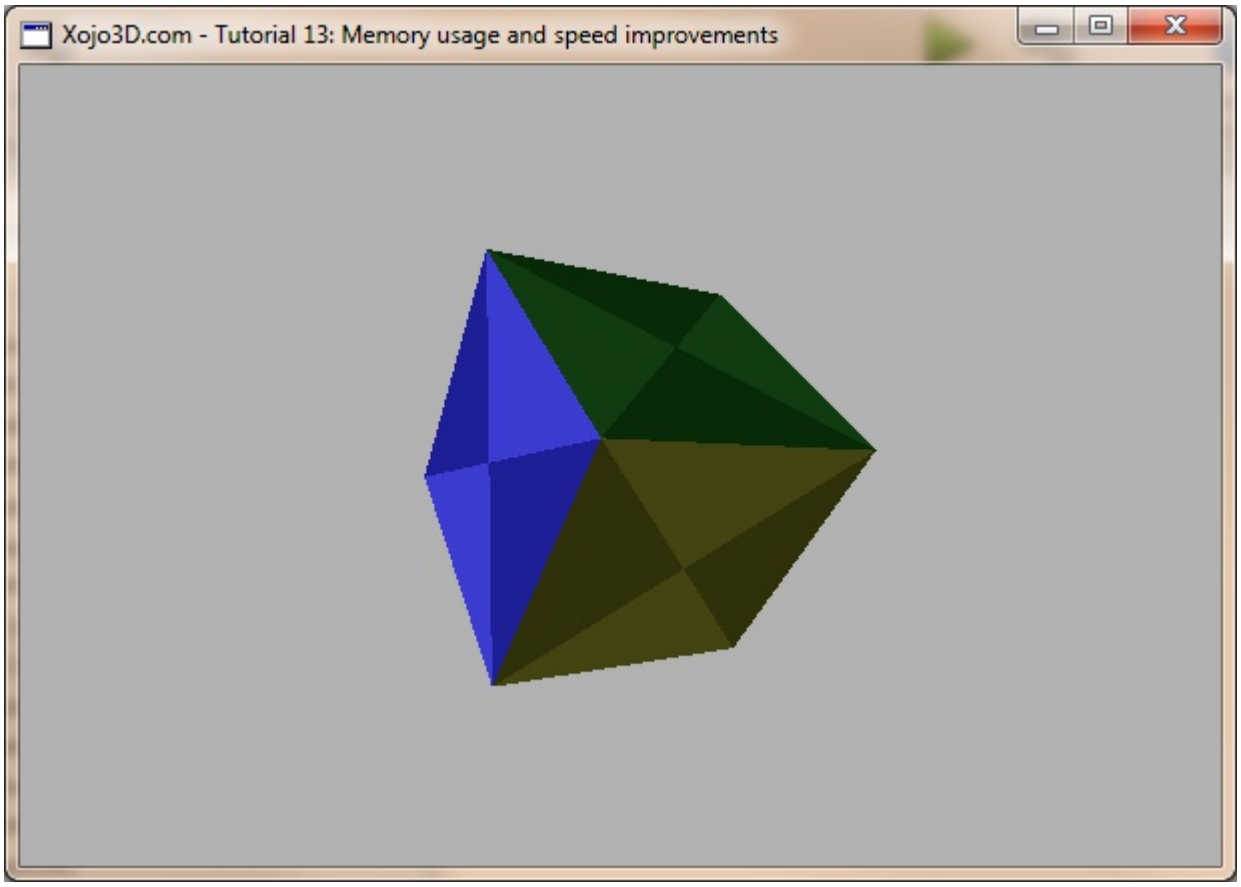
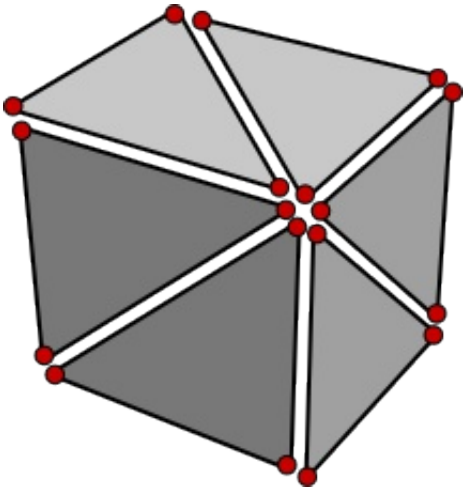
Tutorial 13: Memory usage and speed improvements

In this tutorial we revisit the 3D data structures. Our goal is to improve memory usage, and increase the rendering speed of models.

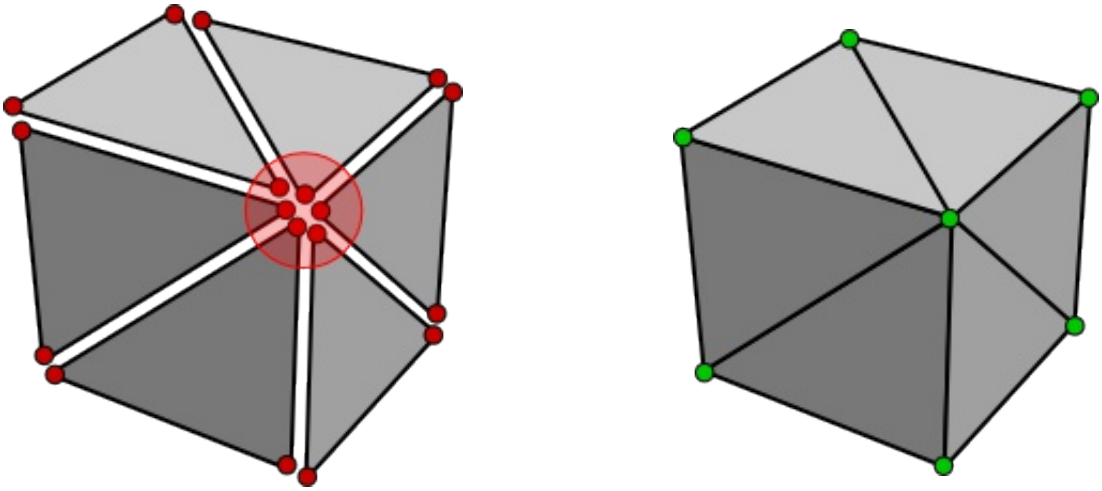


Theory

When we consider the cube model below, it is easy to determine that a basic cube with six faces (front, left, top, right, back and bottom), results in a total of 12 triangular polygons. Each polygon requires three vertices that amounts to a total of 12*3=36 vertices. Each vertex consists of three double values (x, y and z). The total memory required to store a vertex is, therefore, 8*3=24 bytes. The amount of memory required to store the vertex data of a simple cube is, therefore, 36*24=864 bytes. You can imagine that with more detailed models, memory usage can quickly become a concern.

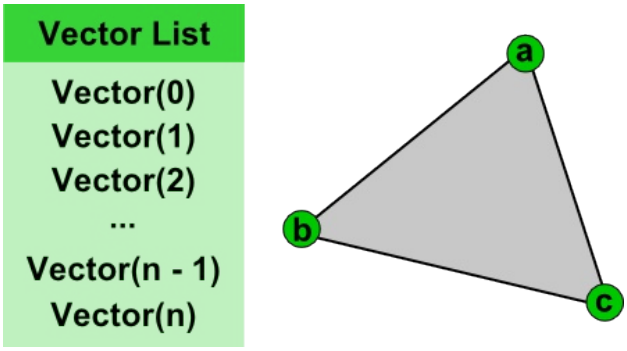


Let's have another look at the cube model. Some vertices occupies the same position in space and are really only duplicates of each other. If we use a single vertex instance on each corner of the cube and shared it between polygons, rather than having six separate vertices, then the total number of vertices required to represent the cube is reduced to only eight vertices (see the green vertices). Memory usage is now reduced to 192 bytes, a considerable saving of 77.78%.



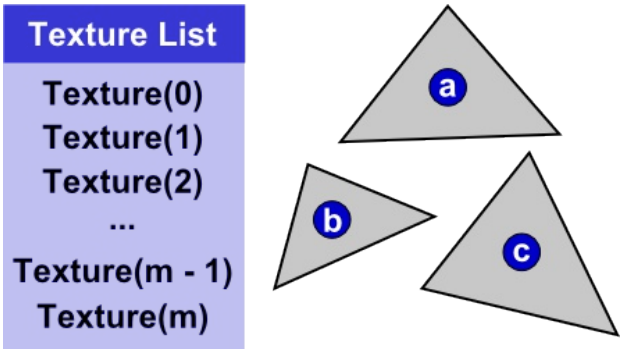
If you remember from the previous tutorials, each polygon stores its own list of vertices. To achieve the above explained reduction in memory usage, we need to change the architecture to store all the vertices used by the model in a **global list**. Our polygon then simply stores an index that points to a vertex in this global list, instead of storing the vertex itself. We now only have to store a vertex once, but can use it as many times as needed by using index pointers.

In the following diagram, a, b and c are Integer values that point to vectors stored in a global vector array. These vertex indexes can be any value between 0 and n. All the polygons of a model use the same vector array to eliminate duplicate vectors, and thereby improving memory usage.



More than one polygon will usually use the same texture during texture mapping, so it makes sense to store a texture instance once in a global list, rather than storing a separate instance of the texture in each polygon. A texture can then be accessed in this global list by an index that is stored in the polygon. With this approach it is easy for multiple polygons to access the same texture.

In the following diagram, a, b and c are *Integer* values that point to textures stored in a global texture array. These texture indexes can be any value between 0 and m. All the polygons of a model use the same texture array to eliminate duplicate texture instances, and thereby improving memory usage.



There are plenty of ways to increase the rendering speed of your 3D scenes, when you make clever use of the features provided by OpenGL. One feature of OpenGL that saves CPU processing time, and thereby increases your frame rate, is display lists. A display list is effectively an object that is created and stored in OpenGL's memory. This display list can then be redrawn on demand, without using CPU processing power.

It is highly recommended that you make use of display lists to increase the rendering speed of your scenes.

Tutorial Steps

1. Open Xojo.
2. Save your project.
3. Configure the following controls:

Control	Name	DoubleBuffer	Left	Top	Maximize Button
Window	SurfaceWindow	-	-	-	ON
OpenGLSurface	Surface	ON	0	0	-

4. Position and size Surface to fill the whole window, and set its Locking to left, top, bottom and right.

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

```
Surface.Render
```

6. Import the X3Core module, created in the previous tutorial.
   You can download the module from http://www.xojo3d.com/tutorials/tut013/x3core.zip.
7. Add the following code to the *Surface.Open* event handler:

```
X3_Initialize

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

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

```
X3_SetPerspective Surface
```

9. Remove the property named Vertex() from *X3Polygon*.
10. Remove the property named Texture from *X3Polygon*.
11. Add the following properties to *X3Polygon*:

Name	Type	Default
VIndex()	Integer	
TIndex	Integer	-1
ParentModel	X3Model	

12. Convert *X3Polygon.ParentModel* into a computed property.
13. Change the type of *X3Polygon.mParentModel* be of type *WeakRef*.
14. Replace the code in the *X3Polygon.ParentModel* SET method with the following:

```
mParentModel = New WeakRef(value)
```

15. Replace the code in the *X3Polygon.ParentModel* GET method with the following:

```
return X3Model(mParentModel.Value)
```

16. Add the following properties to *X3Model*:

Name	Type
Texture()	X3Texture
Vertex()	X3Vector
OGLName	Integer

17. Add the following method to *X3Model*:

```
Sub AppendPolygon(poly As X3Core.X3Polygon)
  poly.ParentModel = me
  Polygon.Append poly
End Sub
```

18. Replace the code in *X3Core.X3_RenderModel* with the following code:

```
Dim i, j As Integer
Dim poly As X3Core.X3Polygon

OpenGL.glPushMatrix

X3_SetRotation(model.Rotation)

if (model.OGLName > 0) then

  OpenGL.glCallList model.OGLName

else

  model.OGLName = OpenGL.glGenLists(1)
  OpenGL.glNewList model.OGLName, OpenGL.GL_COMPILE

  OpenGL.glBegin OpenGL.GL_TRIANGLES

  for i = 0 to model.Polygon.Ubound

// continue on next page
```

```
// continued from previous page

    poly = model.Polygon(i)
    OpenGL.glNormal3d poly.Normal.X, poly.Normal.Y, poly.Normal.Z

    if (poly.TIndex >= 0) and (poly.UVMap.Ubound >=
          poly.VIndex.Ubound) then

      OpenGL.glColor4d(1, 1, 1, 1)
      OpenGL.glBindTexture(OpenGL.GL_TEXTURE_2D,
          model.Texture(poly.TIndex).OGLName)

      for j = 0 to poly.VIndex.Ubound
        OpenGL.glTexCoord2d poly.UVMap(j).U, poly.UVMap(j).V
        OpenGL.glVertex3d model.Vertex(poly.VIndex(j)).X,
          model.Vertex(poly.VIndex(j)).Y,
          model.Vertex(poly.VIndex(j)).Z
      next j

    else

      if poly.FillColor <> nil then
        OpenGL.glColor4d(poly.FillColor.Red, poly.FillColor.Green,
          poly.FillColor.Blue, poly.FillColor.Alpha)
      else
        OpenGL.glColor4d(1, 1, 1, 1)
      end if
      for j = 0 to 2
        OpenGL.glVertex3d model.Vertex(poly.VIndex(j)).X,
          model.Vertex(poly.VIndex(j)).Y,
          model.Vertex(poly.VIndex(j)).Z
      next j

    end if

  next i

  OpenGL.glEnd
  OpenGL.glEndList
  OpenGL.glCallList model.OGLName

end if

OpenGL.glPopMatrix
```

19. Make the following changes to the code in the *X3Polygon.CalculateNormal* method:

```
' replace

' 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

' with

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

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

20. Import the X3Test module into your project.

You can download the module from http://www.xojo3d.com/tutorials/tut013/x3test.zip.

21. Add the following properties to *SurfaceWindow*:

Name	Type
Model	X3Core.X3Model
MousePrevX	Integer
MousePrevY	Integer

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

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

Model = X3Test_Cube6()
```

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

```
OpenGL.glClearColor(0.7, 0.7, 0.7, 1)
OpenGL.glClear(OpenGL.GL_COLOR_BUFFER_BIT +
     OpenGL.GL_DEPTH_BUFFER_BIT)

// continue on next page
```

```
// continued from previous page

OpenGL.glPushMatrix

OpenGL.glTranslatef 0, 0, -5

X3_RenderModel Model

OpenGL.glPopMatrix
```

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

```
MousePrevX = x
MousePrevY = y

return true
```

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

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

Surface.Render

MousePrevX = x
MousePrevY = y
```

26. Save and run your project. Drag the cube with your mouse to rotate it.

Analysis

The first major change we made was to remove the X3Vector array property and X3Texture property from the X3Polygon class. These properties are replaced with an integer array named VIndex(), and an integer property named TIndex. These new properties are index pointers into a new X3Vector array and X3Tecture array located in the X3Model class.

To indicate that a polygon should only be filled with a color, and not be texture mapped with a texture, we simply set the TIndex property of the polygon equal to any value less than 0.

X3Polygon.ParentModel:

```
Get
  return X3Model(mParentModel.Value)
End Get

Set
  mParentModel = New WeakRef(value)
End Set
```

Since the actual vertex and texture data of the polygon is now stored in the X3Model class, we need a way to access that data. The computed ParentModel property serves that purpose.

The problem is that we now have a reference that points to the parent model, and the parent model stores a reference back to the polygon, creating a circular reference. Circular references cause problems when objects are unloaded from memory.

To circumvent memory problems the mParentModel property is of type WeakRef. Using a WeakRef object breaks the circular reference and ensures the correct unloading of polygon objects from memory. The set and get methods respectively cast X3Model objects to and from WeakRef instances.

X3Model.AppendPolygon:

```
Sub AppendPolygon(poly As X3Core.X3Polygon)
  poly.ParentModel = me
  Polygon.Append poly
End Sub
```

AppendPolygon is a helper method to easily add new polygons to an existing model. It sets the ParentModel property of the polygon correctly and adds the polygon to the existing array of polygons.

X3Core.X3_RenderModel:

```
Dim i, j As Integer
Dim poly As X3Core.X3Polygon

OpenGL.glPushMatrix

X3_SetRotation(model.Rotation)

if (model.OGLName > 0) then

  OpenGL.glCallList model.OGLName

else

  model.OGLName = OpenGL.glGenLists(1)
  OpenGL.glNewList model.OGLName, OpenGL.GL_COMPILE

  OpenGL.glBegin OpenGL.GL_TRIANGLES

  for i = 0 to model.Polygon.Ubound

// continue on next page
```

```
      poly = model.Polygon(i)
      OpenGL.glNormal3d poly.Normal.X, poly.Normal.Y, poly.Normal.Z

      if (poly.TIndex >= 0) and (poly.UVMap.Ubound >=
            poly.VIndex.Ubound) then

        OpenGL.glColor4d(1, 1, 1, 1)
        OpenGL.glBindTexture(OpenGL.GL_TEXTURE_2D,
            model.Texture(poly.TIndex).OGLName)

        for j = 0 to poly.VIndex.Ubound
          OpenGL.glTexCoord2d poly.UVMap(j).U, poly.UVMap(j).V
          OpenGL.glVertex3d model.Vertex(poly.VIndex(j)).X,
            model.Vertex(poly.VIndex(j)).Y,
            model.Vertex(poly.VIndex(j)).Z
        next j

        OpenGL.glBindTexture(OpenGL.GL_TEXTURE_2D, 0)

      else

        if poly.FillColor <> nil then
          OpenGL.glColor4d(poly.FillColor.Red, poly.FillColor.Green,
            poly.FillColor.Blue, poly.FillColor.Alpha)
        else
          OpenGL.glColor4d(1, 1, 1, 1)
        end if
        for j = 0 to 2
          OpenGL.glVertex3d model.Vertex(poly.VIndex(j)).X,
            model.Vertex(poly.VIndex(j)).Y,
            model.Vertex(poly.VIndex(j)).Z
        next j

      end if

    next i

    OpenGL.glEnd
    OpenGL.glEndList
    OpenGL.glCallList model.OGLName

  end if

  OpenGL.glPopMatrix
```

In the new render method, notice how the arrays model.Texture and model.Vertex are now used with the indexes poly.TIndex and poly.VIndex respectively. Since all the vertex and texture data are now stored in the model, we need to retrieve the respective instances from the arrays in the model.

X3Polygon.CalculateNormal:

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

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

Similar to the render method, the CalculateNormal method had to be changed to access vertex information in the parent model.