

# **VIXEN: The Art of 3D Programming**

**An Honors Thesis (HONRS 499)**

**By**

**John S. Iles**

**Thesis Advisor**

**Dr. Ralph Place**

A handwritten signature in black ink, appearing to read "Ralph L. Place", written in a cursive style.

**Ball State University**

**Muncie, Indiana**

**December 2001**

**December 16, 2001**

SpColl  
Thesis  
LD  
2489  
.Z4  
2001  
.I58

## ABSTRACT

VIXEN was conceived several years ago when I decided I wanted create 3D worlds. I wanted to do something really interesting with the techniques I had begun developing in the computer science program. So, in the summer of 1999 I purchased several texts on the subject and began researching 3D programming. I began my work in January of 2000 using OpenGL as the language to describe my 3D worlds. Before the month was out, I had developed VIXEN 1.0, a simple but effective demonstration of Windows programming, OpenGL, and interactivity.

After VIXEN 1.0, I began the design for a next-level project, then entitled VIXEN 2. As the semester break approached, I realized that VIXEN 2 would make an excellent Honors project. I enlisted Dr. Ralph Place as my advisor and hashed out an ambitious set of requirements. VIXEN 2 became VIXEN: The Art of 3D Programming, a set of three programs created to model 3D worlds.

The first of these programs is VIXEN Modeler. Modeler uses two windows working together to display and edit 3D models. Though primitive in nature, Modeler has proven to be quite effective if the user is patient and persistent. I have often boasted that I can create any 3D shape in Modeler if I had forever in which to do it.

VIXEN Texture Mapper is the second program in the set. Mapper uses a single window for its GUI, incorporating the 3D window as part of the interface. The user can load any 3D object created in Modeler and then "paint" the object with bitmap images, or textures. After a texture has been applied, the user saves the object in its final form, and it is ready to be displayed in the final program.

The last program is the graphics demonstration, or VIXEN 2\_1. Using Modeler, I created a bird model and its wings, and then loaded the finished object into the simulation. A set of square tiles was also defined to simulate the ocean, and an appropriate texture was applied.

Together, these three programs form the current backbone of my resume and portfolio as well as being my Honors project. VIXEN is proof of experience in 3D programming with Direct3D and OpenGL, object-oriented engineering, and Windows development.

## **ACKNOWLEDGEMENTS**

I would like to thank Dr. Ralph Place for volunteering his time to be my thesis advisor. Without him my requirements would have been far too broad, and I would not have been able to complete the project. As it is, there are a few “whiz-bang” graphics I planned into the project that did not make the final version.

## REFERENCES

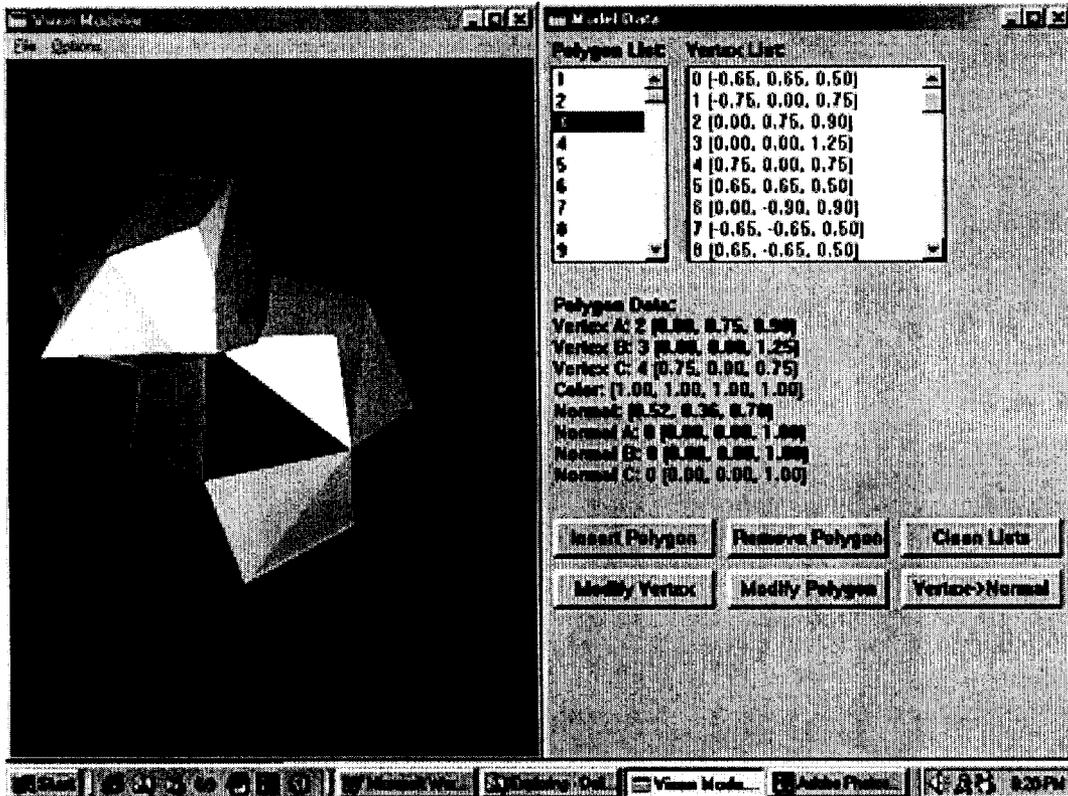
Ebert, David S. Texturing and Modeling 2<sup>nd</sup> Edition. San Diego: AP Professional, 1998.

Watt, Alan H. Advanced Animation and Rendering Techniques: Theory and Practice. New York: ACM Press, 1992.

Petzold, Charles. Programming Windows 95. Redmond: Microsoft Press, 1996.

Wright, Richard S., Jr. OpenGL Superbible, 2<sup>nd</sup> Edition. Waite Group, 1999.  
[www.gamasutra.com](http://www.gamasutra.com)

# VIXEN MODELER – USER’S MANUAL



VIXEN Modeler was created with the idea that I would be the sole user, so I designed it to be efficiently used by the person who wrote the program. The side-effect of such a development technique is that no one besides myself will have a very easy time using the software. In any event, a persistent user can create virtually any shape from this simple modeling utility.

## 1 Getting Started

### *System Requirements:*

Modeler was designed to run on Windows computers with support for OpenGL rendering. Modeler has trouble running on the latest versions of the Windows platform, especially Windows XP. It is highly recommended that the user update the drivers for their graphics hardware to obtain optimal performance.

### *Installing VIXEN Modeler:*

To install Modeler on a Windows computer:

1. Unzip the file "modeler\_20011128.zip" or whatever version number you have downloaded using WinZip.
2. Double-click the executable file "Modeler 2a" in the extraction folder.

### *Features:*

**List based data storage** enables the user to connect the polygons together and makes life easier when modifications to a vertex are necessary. In other words, changing the position or normal for any one vertex changes the data in all the polygons that use that vertex.

**Vertex shading** allows the user to approximate smooth surfaces by changing the way light bounces off of a surface.

### *Limitations:*

By using OpenGL as the rendering language, the user must specify the vertices of each polygon in counterclockwise order. In addition, the positive z-axis extends out of the monitor towards the user. These two very specific parameters create problems when \*.vxno objects are loaded into worlds that use Direct3D as the rendering language of choice.

## **2 Tutorial**

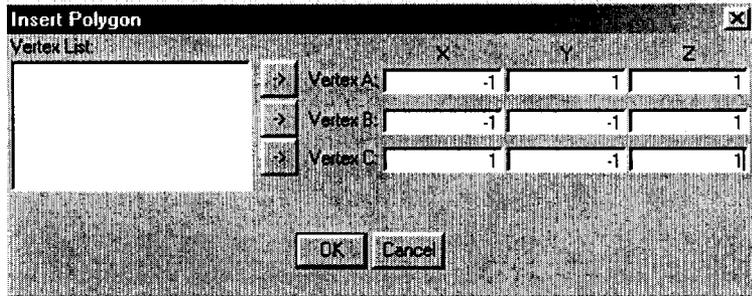
1. Open the VIXEN Modeler program.
2. This is the traditional blank slate. To begin designing your model, simply click on the Insert Polygon button.
3. The window you are presented with is the form for creating new polygons. Remember that you must specify the vertices so that they wind in a counterclockwise order. This is because the program is base on OpenGL and not Direct3D. The issue will be addressed when the software is ported.

- Once you have created a polygon, the vertices will appear in the list at the left. You may select those vertices and click on the arrow buttons to automatically fill the vertex boxes. For now though, we must specify each vertex manually. Use the following inputs:

Vertex A = (-1, 1, 1)

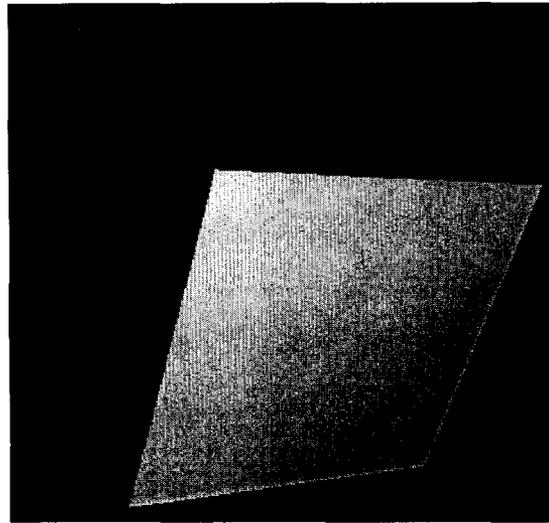
Vertex B = (-1, -1, 1)

Vertex C = (1, -1, 1)

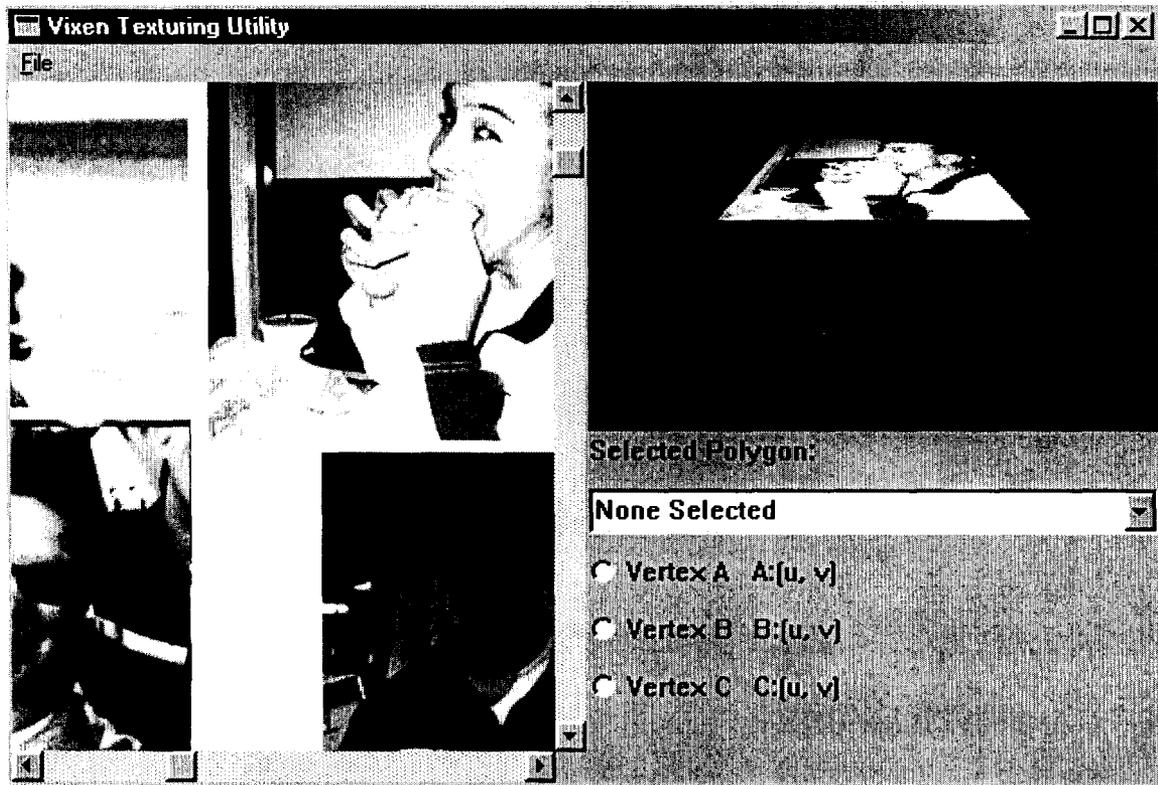


- After selecting OK, you will see your first triangle appear in the rendering window. You may select the new polygon by clicking on the number in the proper text box. By selecting the polygon, you can use the Remove Polygon button to delete the polygon. By selecting the Modify Polygon button you can adjust the vertex normals for the polygon individually.
- You can also select vertices. The Modify Vertex button allows you to change the coordinates for the vertex. Select Vertex 0 and then click the Modify Vertex button. Change the x value to a positive 1. Because I do not duplicate vertices in this program, editing the value for any one vertex changes all the polygons that use that vertex. The same goes for normals if you click the Vertex->Normal button.
- Because I save the vertices and normals in separate lists, you have to clean the residuals when you modify polygons, vertices, and normals. The Clean Lists button does just this, and you should use it to eliminate unused vertices and normals in your models.
- Finish creating a cube using the following vertices:

Upper front right (1, 1, 1)  
Upper back left (-1, 1, -1)  
Upper back right (1, 1, -1)  
Lower back left (-1, -1, -1)  
Lower back right (1, -1, -1)



# VIXEN TEXTURE MAPPER – USER’S MANUAL



VIXEN Texture Mapper allows a user to “paint” textures over the models they create using VIXEN Modeler. In the same vein as Modeler, Mapper uses an interface that takes a deal of patience to work with. However, patience is rewarded when the user completes a proper texture mapping. Mapper is also the tool that converts \*.vxno objects to the Direct3D compatible \*.vtp objects.

## 1 Getting Started

### *System Requirements:*

Mapper is designed to run on Windows enabled PCs. The user may note a marked decrease in performance as the speed of their computer increases. This is a bug, and the fix is in the works. Lastly, it is highly recommended that the user update the drivers for their graphics hardware to obtain optimal performance.

### *Installing VIXEN Texture Mapper:*

To install Modeler on a Windows computer:

1. Unzip the file "mapper\_20011128.zip" or whatever version number you have downloaded using WinZip.
2. Double-click the executable file "Texture Mapper 2\_0" in the extraction folder.

### *Features:*

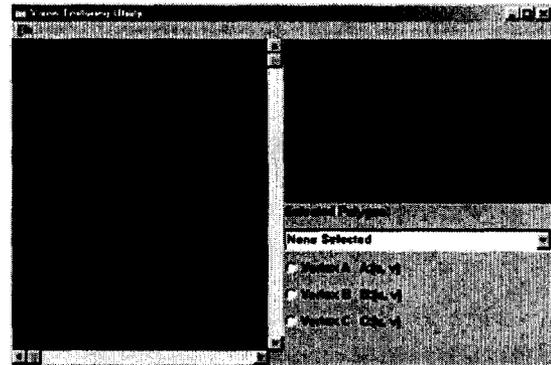
**Bitmap based textures** are easily adapted to Direct3D and Windows.

### *Limitations:*

Using Direct3D to draw the objects and map the textures creates certain complications given the fact that the objects were initially modeled in OpenGL. This means the application had to flip had to flip the z-axis and reverse the winding of the polygons before the objects would draw properly. There is also that speed bug on fast computers.

## **2 Tutorial**

1. Open the VIXEN Texture Mapper program.
2. This is the blank slate, select **File->Open VXNO Model** and select "cube.vxno" in the open file dialog.
3. You are now presented with a rotating cube.
4. Select **File->Open Bitmap** and select "smiley.bmp".
5. Choose *Polygon 0* from the combo box and select *Vertex A*.
6. Scroll the bitmap window horizontally to the right and vertically down until it scrolls no more and click in the lower right hand corner so that  $A:\{1.00, -0.00\}$  appears next to *Vertex A*.
7. Select *Vertex B*.



8. Scroll the bitmap window horizontally to the left until it scrolls no more and click in the lower left hand corner so that  $B:\{0.00, -0.00\}$  appears next to *Vertex B*.
9. Select *Vertex C*.
10. Scroll the bitmap window vertically up until it scrolls no more and click in the upper left hand corner so that  $C:\{0.00, -1.00\}$  appears next to *Vertex C*.
11. In the same fashion, select *Polygon 1* from the combo box.
12.  $A:\{1.00, -1.00\}$  -> click upper right corner of bitmap.  
 $B:\{1.00, -0.00\}$  -> click lower right corner of bitmap.  
 $C:\{0.00, -1.00\}$  -> click upper left corner of bitmap.



# DESIGN SPECIFICATIONS

The key to object-oriented software development is a thorough understanding of how classes of objects interact with each other. A well-defined set of classes will hide the specific implementation of each object from the rest of the program. Such abstraction lends itself well to modification, extension, and maintenance. I spent countless hours debugging the various applications, but I can not even fathom how many hours I would have spent had VIXEN been constructed from a procedural methodology rather than an object-oriented approach.

## 1 Environment

The selection of an appropriate development environment is critical to each software project. For VIXEN, I knew I would be using the C/C++ programming language because of its speed and reliability in addition to my familiarity with the syntax and intricacies of the language. My alternative would have been Visual Basic, which I ruled out because I wanted low-level access to the graphics commands. Lastly, I selected the Windows environment because it is the system most readily available to me.

I selected OpenGL as the initial rendering language for VIXEN with the first installment in the VIXEN project series back in January. The objects I created for that simple demo translated easily to VIXEN Modeler. However, when I began to design VIXEN Texture Mapper, I wanted to use Direct3D. As a part of Microsoft's DirectX software development kit, Direct3D is the most widely used rendering language for Windows. Unfortunately, it is also much more difficult to learn than OpenGL.

## 2 Evolution

VIXEN itself was originally much more ambitious than the final project description. For the reduction Dr. Place had me place on the effort, I am eternally grateful. We decided that one demo program would be sufficient, given my decision to develop my own modeling and texturing utilities. Known as Ocean Demo, the original specifications called for a pelican, fog, fractal water, and a coastline. The reality of Ocean Demo and the limitations of learning two rendering languages left the final demo with a sea gull, and animated water from a pre-generated texture rather than a fractal one. I've always read up the necessity of cutbacks in the post-production articles for video games, but this is the first time I've experienced it myself.

The switch from OpenGL to Direct3D between the development of Modeler and Mapper forced me to restructure the object hierarchies as well. As you will see from the class descriptions in the next section, these changes were significant and the two rendering frameworks barely resemble each other. A second and final overhaul of the rendering system was made on November 18<sup>th</sup> between the development of Mapper and Ocean Demo. This hierarchy lasted through the end of the development cycle, though it will have to be revisited before I can move on to create the next generation VIXEN applications.

### 3 VIXEN 1.0 Class List – OpenGL Language Base

#### **Renderer**

The Renderer object is an abstract type from which two specialized objects will be derived. These objects are OpenGL\_Renderer and Direct3D\_Renderer.

```
- hWnd          :HWND
- hDeviceContect :HDC
- hRenderingContext :HGLRC

+ create()
+ initialize( hWnd :HWND, hDC :HDC ) :bool
+ flushRenderer() :bool
+ setPixelFormat( windowWidth :int, windowHeight :int) :bool
```

#### **OpenGL\_Renderer**

The OpenGL\_Renderer inherits from the Renderer object. It uses the OpenGL API to render to the window. It is responsible for setting the proper rendering

### context and the pixel format.

```
- hWnd          :HWND
- hDeviceContect :HDC
- hRenderingContext :HGLRC
```

```
+ create()
+ initialize( hWnd :HWND, hDC :HDC ) :bool
+ flushRenderer() :bool
+ setPixelFormat( windowWidth :int, windowHeight :int) :bool
```

### VxnObject

The VxnObject class describes a container object for collections of polygons and vertices. It uses a storage system that references vertices from a list to prevent duplication and make it easier on the user and the programmer to modify the polygons in the interface.

```
- polygonList      :VxnList<POLYGON>
- vertexList      :vector<COORDINATE>
- normalList      :vector<COORDINATE>
- nextIndex       :int
- szVertexList    :int
- selectedPolygon :int
```

```
+ create()
+ create( filename :char* )
+ render() :void
+ flat_render() :void
+ load( filename :char* ) :int
+ save( filename :char* ) :int
+ close() :void
+ getSelection() :int
+ getPolygon( index :int, &polygon :POLYGON ) :int
+ getVertex( index :int, &vertex :COORDINATE ) :int
+ getNormal( index :int, &normal :COORDINATE ) :int
+ insertPolygon( polygon :POLYGON ) :int
+ insertVertex( vertex :COORDINATE ) :int
+ insertNormal( normal :COORDINATE ) :int
+ removePolygon( index :int ) :int
+ updatePolygon( index :int, polygon :POLYGON ) :int
+ updateVertex( index :int, vertex :COORDINATE ) :int
+ matchVertexToNormal( vertexIndex :int, normalIndex :int ) :int
+ displayPolygon( hPolygonData :HWND, index :int) :int
+ selectPolygon( index :int ) : void
+ loadListbox( hPolygonList :HWND, hVertexList :HWND ) :void
+ loadVertexListbox( hVertexList :HWND ) :void
+ loadNormalListbox( hNormalList :HWND ) :void
```

### POLYGON

The POLYGON object defines a public structure to contain relevant information for properly drawing polygons.

```
+ aVertex :int
+ bVertex :int
+ cVertex :int
+ aNormal :int
+ bNormal :int
+ cNormal :int
+ normal :COORDINATE
```

```
+ color :COLOR
```

### **COORDINATE**

The COORDINATE object creates a structure that outlines sets of 3D points.

```
+ x :SCALAR  
+ y :SCALAR  
+ z :SCALAR
```

### **COLOR**

The COLOR object defines a color using four scalar values to describe red, green, blue, and an alpha value.

```
+ red :SCALAR  
+ green :SCALAR  
+ blue :SCALAR  
+ alpha :SCALAR
```

### **VxnListNode**

The VxnListNode object provides a structured base unit for the VxnList object. They do not exist outside of the VxnList class. In addition, this object is templated, which means the data type T can be any object.

```
+ identifier :int  
+ data :T  
+ next :VxnListNode<T>*  
+ create( id :int, d :T )
```

### **VxnList**

The VxnList object stores a list of VxnListNode objects using a templated data type T. The list includes an iterator object.

```
- first :VxnListNode<T>*  
- iterator :VxnListNode<T>*  
- search( identifier :int ) :VxnListNode<T>*  
+ create()  
+ flush() :void  
+ search( identifier :int, &returnData :T ) :int  
+ insert( identifier :int, data :T ) :int  
+ remove( identifier :int) :int  
+ update( identifier :int, data :T ) :int  
+ iteratorReset() :int  
+ iteratorGetNext() :int  
+ iteratorGetCurrent( &data :T ) :int  
+ iteratorGetIndex() :int  
+ getSize() :int
```

### **Polygon Functions**

These functions are defined globally as a module rather than a class because they operate independently and in conjunction with the other objects.

```
+ calculateNormalVector( &polygon :POLYGON, aVertex :COORDINATE,  
                        bVertex :COORDINATE, cVertex :COORDINATE  
                        ) :void  
+ normalizeVector( &normal :COORDINATE ) :void
```

### Control Functions

As with the polygon functions, these functions are global and operate outside the parameters of the class hierarchy.

```
+ setEditBox_float( hWnd :HWND, idEditBox :int, number :float )
                    :void
+ getEditBox_float( hWnd :HWND, idEditBox :int ) :float
+ setLabel_float( hWnd :HWND, idLabel :int, number :float) :void
```

### Primary Window

These functions are the core operating procedures for VIXEN Modeler. In particular, the window procedure is the control mechanism for the rendering window.

```
+ OpenGL_wndProc( hWnd :HWND, message :UINT,
                 wParam :WPARAM, lParam :LPARAM ) :LRESULT CALLBACK
+ OpenGL_wndSetup( hInstance :HINSTANCE, &windowClass :WNDCLASSEX
                  menuName :char* ) :void
+ OpenGL_wndCreate( hInstance :HINSTANCE, style :DWORD,
                   xPosition :int, yPosition :int,
                   xSize :int, ySize :int,
                   hParentWindow :HWND ) :HWND
+ renderScene_OpenGL( pVxnObject :VxnObject* ) :void
+ renderFlat_OpenGL( pVxnObject :VxnObject* ) :void
+ resetGlobals() :void
```

### Data Window

These functions operate the data window of the VIXEN Modeler. It is responsible for the myriad of child window controls and sets up the dialog procedures.

```
+ data_wndProc( hWnd :HWND, message :UINT,
               wParam :WPARAM, lParam :LPARAM ) :LRESULT CALLBACK
+ data_wndSetup( hInstance :HINSTANCE, &windowClass :WNDCLASSEX
                menuName :char* ) :void
+ data_wndCreate( hInstance :HINSTANCE, style :DWORD,
                 xPosition :int, yPosition :int,
                 xSize :int, ySize :int,
                 hParentWindow :HWND ) :HWND
```

### Insert Polygon Dialog Window

These functions operate the dialog window that allows the user to input a new polygon into the scene.

```
+ insertPolygon_dialogProcedure( hWnd :HWND, message :UINT,
                                wParam :WPARAM, lParam :LPARAM
                                ) :BOOL CALLBACK
+ insertPolygon( hWnd :HWND ) :void
```

### Modify Normal Dialog Window

These functions operate the dialog window that modifies the normal vector for one vertex. The change fluctuates to all polygons touching that vertex.

```
+ modifyNormal_dialogProcedure( hWnd :HWND, message :UINT,
                                wParam :WPARAM, lParam :LPARAM
                                ) :BOOL CALLBACK
+ modifyNormal( hWnd :HWND, vertexIndex :int ) :void
```

### Modify Polygon Dialog Window

These functions operate the dialog box that enables the user to change the vertex normals on a polygon by polygon basis.

```
+ modifyPolygon_dialogProcedure( hWnd :HWND, message :UINT,
                                wParam :WPARAM, lParam :LPARAM
                                ) :BOOL CALLBACK
+ modifyPolygon( hWnd :HWND, index :int ) :void
```

### Modify Vertex Dialog Window

These functions operate the dialog box that allows the user to change the position of any vertex. The changes fluctuate throughout the scene, so all polygons using the vertex will be affected.

```
+ modifyVertex_dialogProcedure( hWnd :HWND, message :UINT,
                                wParam :WPARAM, lParam :LPARAM
                                ) :BOOL CALLBACK
+ modifyVertex( hWnd :HWND, index :int ) :void
```

### \*.vxno File Dialog Window

These functions create open and save window dialogs for the \*.vxno filetype.

```
+ initializeVxnFile( hWnd :HWND, &ofnVxnFile :OPENFILENAME ) :void
+ vxnOpenDialog( hWnd :HWND, &ofnVxnFile :OPENFILENAME,
                 fileName :PSTR, titleName :PSTR ) :BOOL
+ vxnSaveDialog( hWnd :HWND, &ofnVxnFile :OPENFILENAME,
                 fileName :PSTR, titleName :PSTR ) :BOOL
```

## 4 VIXEN 2\_1 Class List - Direct3D Language Base

### D3D\_Renderer

The D3D\_Renderer object creates and maintains the Direct3D objects used to render to the window. The object takes a scene graph object as a parameter to its calls, so multiple scene graphs can be used with just one renderer object.

```
- pDirectDraw :LPDIRECTDRAW7
- pddsPrimarySurface :LPDIRECTDRAWSURFACE7
- pddsBackBuffer :LPDIRECTDRAWSURFACE7
- pddsZBuffer :LPDIRECTDRAWSURFACE7
- pDirect3D :LPDIRECT3D7
- pDirect3DDevice :LPDIRECT3DDEVICE7
- screenRectangle :RECT
- resetData() :HRESULT
- initializeDirect3D( hWnd :HWND, pDeviceGUID :const GUID*,
                    &ddsSurfaceDescription :DDSURFACEDESC2 ) :HRESULT
- restoreSurfaces() :HRESULT
+ create()
+ initialize( hWnd :HWND, pDriverGUID :GUID*,
             pDeviceGUID :GUID* ) :HRESULT
+ initialize( pSceneGraph :D3D_SceneGraph* ) :HRESULT
+ initializeSceneObject( pSceneObject :D3D_SceneObject* ) :HRESULT
+ initializeTextureObject( pTextureObject :D3D_TextureObject* ) :HRESULT
+ windowResize( hWnd :HWND, pDeviceGUID :const GUID* ) :HRESULT
+ cleanupDirect3D( pSceneGraph :D3D_SceneGraph* ) :HRESULT
```

```

+ cleanupSceneObject( pSceneObject :D3D_SceneObject* ) :HRESULT
+ cleanupTextureObject( pTextureObject :D3D_TextureObject* ) :HRESULT
+ render( pSceneGraph :D3D_SceneGraph* ) :HRESULT
+ frameMove( pSceneGraph :D3D_SceneGraph*,
             pTimerData :TIMERDATA* ) :HRESULT
+ onMove( x :int, y :int ) :HRESULT
+ showFrame() :HRESULT

```

### D3D\_SceneGraph

The D3D\_SceneGraph object organizes scene objects and texture objects into some comprehensible order.

```

- first :D3D_SceneNode*
- pCamera :D3D_Camera*
- renderingHeap :RenderingHeap

+ create()
+ setCamera( pCam :D3D_Camera* ) :HRESULT
+ createSceneNode( groupIdIdentifier :INT, nodeIdIdentifier :INT,
                  objectIdentifier :INT,
                  pSceneObject :D3D_SceneObject*,
                  pTextureObject :D3D_TextureObject*,
                  pAnimateNode :D3D_TransformNode ) :INT
+ createGroupNode( groupIdIdentifier :INT,
                  pAnimateNode :D3D_TransformNode* ) :INT
+ removeGroupNode( groupIdIdentifier :INT ) :HRESULT
+ removeSceneNode( nodeIdIdentifier :INT ) :HRESULT
+ removeSceneObject( objectIdentifier :INT ) :HRESULT
+ flush() :HRESULT
+ initializeDirect3D( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT
+ cleanupDirect3D( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT
+ render( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT
+ frameMove( pDirect3DDevice :LPDIRECT3DDEVICE7
            pTimerData :TIMERDATA* ) :HRESULT

```

### D3D\_Camera

The D3D\_Camera object positions, rotates, and generally controls the camera objects in a scene.

```

- position :D3DVECTOR
- lookVector :D3DVECTOR
- upVector :D3DVECTOR
- rightVector :D3DVECTOR
- translationMatrix :D3DMATRIX
- pitchMatrix :D3DMATRIX
- yawMatrix :D3DMATRIX
- rollMatrix :D3DMATRIX

- regenerateBaseVectors() :VOID
+ create()
+ translate( x :FLOAT, y :FLOAT, z :FLOAT ) :VOID
+ yaw( theta :FLOAT ) :VOID
+ roll( theta :FLOAT ) :VOID
+ pitch( theta :FLOAT ) :VOID
+ initializeDirect3D( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT
+ cleanupDirect3D( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT
+ render( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT
+ getPosition() :D3DVECTOR

```

### D3D\_SceneNode

The D3D\_SceneNode object is the base class from which the real power of the scene graph is created.

|  |                          |
|--|--------------------------|
| - pAnimateNode :D3D_TransformNode*   | - worldMatrix :D3DMATRIX |
| + nodeIdentifier :INT  | + next :D3D_SceneNode*   |
| + create()   |                          |
| + getPosition() :D3DVECTOR   |                          |
| + setWorldMatrix( matrix :D3DMATRIX ) :VOID  |                          |
| + initializeDirect3D( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT                  |                          |
| + cleanupDirect3D( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT                     |                          |
| + render( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT                              |                          |
| + frameMove( pDirect3DDevice :LPDIRECT3DDEVICE7<br>pTimerData :TIMERDATA* ) :HRESULT |                          |

### VTP\_SceneNode :public D3D\_SceneNode

The VTP\_SceneNode object extends the functions of its parent to associate scene objects, texture objects, with the animation nodes.

|   |  |
|---|--|
| - pTextureObject :D3D_TextureObject*  |  |
| - pSceneObject :D3D_SceneObject*  |  |
| + objectIdentifier :INT   |  |
| + create()  |  |
| + create( nodeIdentifier :INT, objectIdentifier :INT,<br>pSceneObject :D3D_SceneObject*,<br>pTextureObject :D3D_TextureObject*,<br>pAnimateNode :D3D_TransformNode ) :INT |  |

### GROUP\_SceneNode :public D3D\_SceneNode

The GROUP\_SceneNode object extends the functions of its parent to group related models together for animation purposes in the model.

|   |                        |
|---|------------------------|
| - first :D3D_SceneNode*   | + groupIdentifier :INT |
| + create()  |                        |
| + create( nodeIdentifier :INT, objectIdentifier :INT,<br>pSceneObject :D3D_SceneObject*,<br>pTextureObject :D3D_TextureObject*,<br>pAnimateNode :D3D_TransformNode ) :INT |                        |
| + removeSceneNode( nodeIdentifier :INT ) :HRESULT   |                        |
| + removeSceneObject( objectIdentifier :INT ) :HRESULT   |                        |
| + flush() :HRESULT  |                        |
| + zOrder( &renderingHeap :RenderingHeap, position :D3DVECTOR ) :VOID  |                        |

### D3D\_SceneObject

The D3D\_SceneObject object acts as a storage unit for the vertex-by-vertex, polygon-by-polygon representation of a model.

|  |                        |
|--|------------------------|
| - vertexList :D3DVERTEX*   | - sizeVertexList :LONG |
| - material :D3DMATERIAL7   |                        |
| + create()   |                        |
| + loadFile( fileName :char* ) :BOOL  |                        |
| + initializeDirect3D( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT  |                        |
| + cleanupDirect3D( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT   |                        |
| + render( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT  |                        |
| + frameMove( pDirect3DDevice :LPDIRECT3DDEVICE7<br>pTimerData :TIMERDATA* ) :HRESULT   |                        |
| + setMaterial( ambientRed :FLOAT, ambientGreen :FLOAT,<br>ambientBlue :FLOAT, diffuseRed :FLOAT,<br>diffuseGreen :FLOAT, diffuseBlue :FLOAT ) :HRESULT |                        |

**VTP\_SceneObject :public D3D\_SceneObject**

The VTP\_SceneObject acts just like its parent class except the load function has been overwritten to load models using the \*.vtp file extension.

+ create()

**VXNO\_SceneObject :public D3D\_SceneObject**

The VXNO\_SceneObject acts just like its parent class except the load function has been overwritten to load models using the \*.vxno file extension.

+ create()

**TMAP\_SceneObject :public VXNO\_SceneObject**

The TMAP\_SceneObject object allows the program to access the data inside the model for display in the program.

- regularMaterial :D3DMATERIAL7                      - selectedMaterial :D3DMATERIAL7  
- selectedPolygon :INT

+ create()  
+ saveFile( fileName :char\* ) :BOOL  
+ loadVtpFile( fileName :char\* ) :BOOL  
+ fillComboBox( hComboBox :HWND ) :HRESULT  
+ selectPolygon( hWindow :HWND, polygonIndex :int ) :HRESULT  
+ setTextureCoordinates( pTextureCoordinates :TCOORDINATES\* ) :HRESULT

**WATER2\_SceneObject :public D3D\_SceneObject**

The WATER2\_SceneObject acts just like its parent class except the load function has been overwritten to create a series of WATER2\_Tile objects used for terrain in Ocean Demo.

- waterTile\_1 :WATER2\_Tile                      - numberOfRows :int                      - numberOfColumns :int  
+ create()

**WATER2\_Tile**

The WATER2\_Tile object defines a square of some default size. Texture coordinates are assigned to each of the four vertices, and these coordinates are moved at some default speed by the frameMove method.

- uOne :FLOAT                      - vOne :FLOAT                      - uTwo :FLOAT                      - vTwo :FLOAT  
- uThree :FLOAT                      - vThree :FLOAT                      - uFour :FLOAT                      - vFour :FLOAT  
- xLeft :FLOAT                      - xRight :FLOAT                      - zTop :FLOAT                      - zBottom :FLOAT  
- one :D3DVECTOR                      - two :D3DVECTOR  
- three :D3DVECTOR                      - four :D3DVECTOR  
- vertexList :D3DVERTEX[]

+ create()  
+ render( pDirect3DDevice :LPDIRECT3DDEVICE7,  
          xCorner :FLOAT, zCorner :FLOAT ) :HRESULT  
+ frameMove( pDirect3DDevice :LPDIRECT3DDEVICE7  
              pTimerData :TIMERDATA\* ) :HRESULT

**D3D\_TextureObject**

The D3D\_TextureObject object loads and stores image data that can be mapped to DirectX surfaces.

- pBitmapFileHeader :PBITMAPFILEHEADER                      - pBitmapInfo :PBITMAPINFO  
- pddsTexture :LPDIRECTDRAWSURFACE7                      - pBitmapImage :BYTE\*

```

+ create()
+ loadFile( fileName :char* ) :BOOL
+ initializeDirect3D( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT
+ cleanupDirect3D( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT
+ render( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT
+ frameMove( pDirect3DDevice :LPDIRECT3DDEVICE7
             pTimerData :TIMERDATA* ) :HRESULT

```

### **BMP\_TextureObject :public D3D\_TextureObject**

The BMP\_TextureObject object loads \*.bmp files onto the DirectX surface.

```

+ create()

```

### **TMAP\_TextureObject :public BMP\_TextureObject**

The TMAP\_TextureObject object allows the program to draw the loaded texture to a window as if it were a regular bitmap.

```

- xCorner :LONG                                     - yCorner :LONG

```

```

+ create()
+ paint( hDeviceContext :hDC ) :void
+ convertCoordinates( &u :float, &v :float ) :void
+ getDimensions( &bitmapWidth :LONG, &bitmapHeight :LONG ) :void
+ setXCorner( &x :LONG, clientWidth :LONG ) :void
+ setYCorner( &y :LONG, clientHeight :LONG ) :void

```

### **D3D\_TransformNode**

The D3D\_TransformNode object is the standard positional and rotational descriptor for scene nodes. Derivative classes allow for animation.

```

- position          :D3DVECTOR                    - thetaVector :D3DVECTOR
- translationMatrix :D3DMATRIX                    - pitchMatrix :D3DMATRIX
- yawMatrix         :D3DMATRIX                    - rollMatrix  :D3DMATRIX

```

```

+ create()
+ translate( x :FLOAT, y :FLOAT, z :FLOAT ) :VOID
+ yaw( theta :FLOAT ) :VOID
+ roll( theta :FLOAT ) :VOID
+ pitch( theta :FLOAT ) :VOID
+ initializeDirect3D( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT
+ cleanupDirect3D( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT
+ render( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT
+ getPosition() :D3DVECTOR
+ getModelMatrix() :D3DMATRIX
+ moveTo( pos :D3DVECTOR ) :VOID
+ moveTo( x :FLOAT, y :FLOAT, z :FLOAT ) :VOID
+ moveAlong( x :FLOAT, y :FLOAT, z :FLOAT )
+ apply( pDirect3DDevice :LPDIRECT3DDEVICE7 ) :HRESULT
+ frameMove( pDirect3DDevice :LPDIRECT3DDEVICE7
             pTimerData :TIMERDATA* ) :HRESULT

```

### **BIRD\_AnimateNode :public D3D\_TransformNode**

The BIRD\_AnimateNode object describes a figure eight flight path for a model.

```

- phase          :INT                               - angularVelocity :FLOAT
- thetaDirection :FLOAT                           - totalTheta      :FLOAT
- circle_radius  :D3DVECTOR                       - circle_origin   :D3DVECTOR
- circle1_radius :D3DVECTOR                       - circle1_origin  :D3DVECTOR
- circle2_radius :D3DVECTOR                       - circle2_origin  :D3DVECTOR

```



### Primary Window

These functions are the core operating procedures for VIXEN Mapper and Ocean Demo. In particular, the window procedure is the control mechanism for the rendering window.

```
+ wndProc( hWnd :HWND, message :UINT,
           wParam :WPARAM, lParam :LPARAM ) :LRESULT CALLBACK
+ wndSetup( hInstance :HINSTANCE, &windowClass :WNDCLASSEX
            menuName :char* ) :void
+ wndCreate( hInstance :HINSTANCE, style :DWORD,
             xPositon :int, yPositon :int,
             xSize :int, ySize :int,
             hWndParent :HWND ) :HWND
```

### Texture Window

These functions draw and operate the scrollbars of the texture child window.

```
+ texture_wndProc( hWnd :HWND, message :UINT,
                  wParam :WPARAM, lParam :LPARAM ) :LRESULT CALLBACK
+ texture_wndSetup( hInstance :HINSTANCE, &windowClass :WNDCLASSEX
                  menuName :char* ) :void
+ texture_wndCreate( hInstance :HINSTANCE, style :DWORD,
                   xPositon :int, yPositon :int,
                   xSize :int, ySize :int,
                   hWndParent :HWND ) :HWND
```

### D3DIM Window

These functions render the scene to the Direct3D child window.

```
+ d3dim_wndProc( hWnd :HWND, message :UINT,
                wParam :WPARAM, lParam :LPARAM ) :LRESULT CALLBACK
+ d3dim_wndSetup( hInstance :HINSTANCE, &windowClass :WNDCLASSEX
                menuName :char* ) :void
+ d3dim_wndCreate( hInstance :HINSTANCE, style :DWORD,
                  xPositon :int, yPositon :int,
                  xSize :int, ySize :int,
                  hWndParent :HWND ) :HWND
```

### \*.vtp File Dialog Window

These functions create open and save window dialogs for the \*.vtp filetype.

```
+ initializeVxnFile( hWnd :HWND, &ofnVtnFile :OPENFILENAME ) :void
+ openVtpFile( hWnd :HWND, fileName :PSTR, titleName :PSTR ) :BOOL
+ saveVtpFile( hWnd :HWND, fileName :PSTR, titleName :PSTR ) :BOOL
+ vtnOpenDialog( hWnd :HWND, &ofnVtnFile :OPENFILENAME,
                fileName :PSTR, titleName :PSTR ) :BOOL
+ vtnSaveDialog( hWnd :HWND, &ofnVtnFile :OPENFILENAME,
                fileName :PSTR, titleName :PSTR ) :BOOL
```

### \*.bmp File Dialog Window

These functions create open and save window dialogs for the \*.bmp filetype.

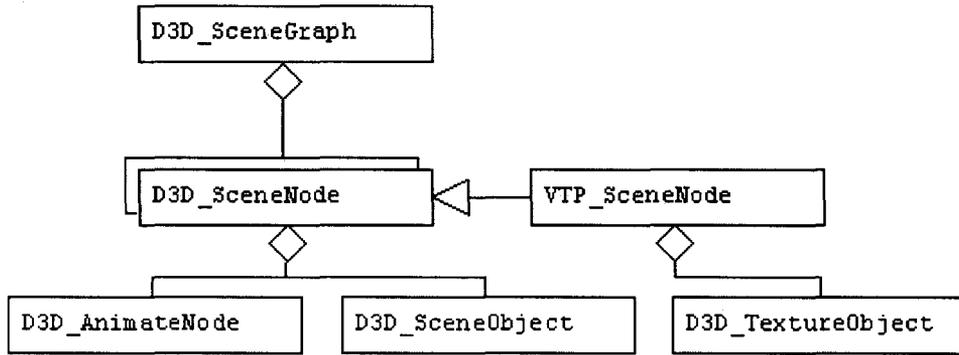
```
+ initializeBmpFile( hWnd :HWND, &ofnBmpFile :OPENFILENAME ) :void
+ openBmpFile( hWnd :HWND, fileName :PSTR, titleName :PSTR ) :BOOL
+ bmpOpenDialog( hWnd :HWND, &ofnBmpFile :OPENFILENAME,
                fileName :PSTR, titleName :PSTR ) :BOOL
```

## D3D Math Functions

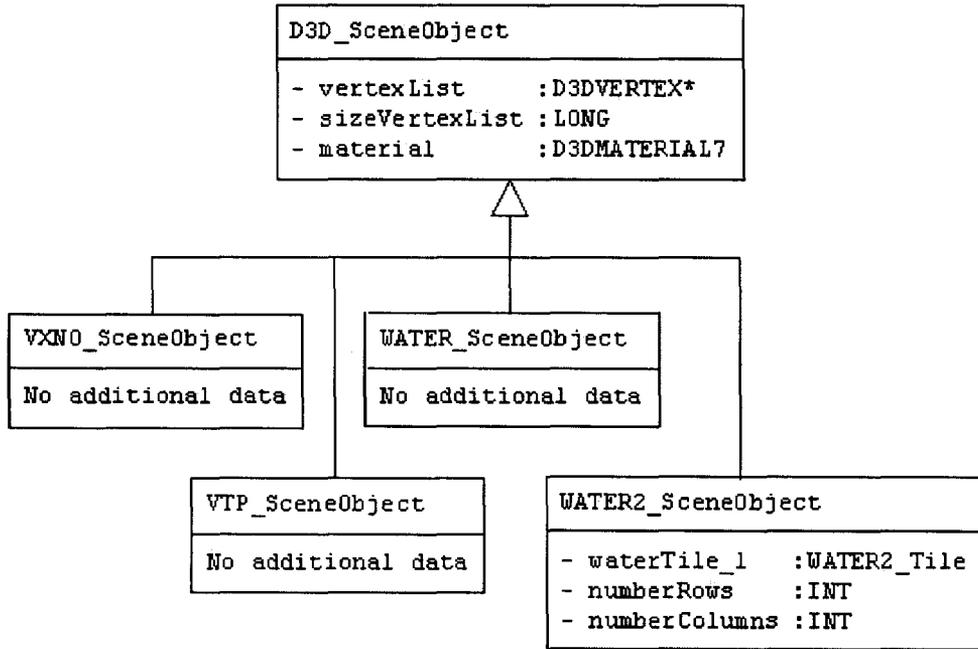
The D3D Math functions provide various linear algebra mathematics to aid in problem solving.

```
+ loadIdentityMatrix( &matrix :D3DMATRIX ) :void
+ loadProjectionMatrix( &matrix      :D3DMATRIX,
                       fieldOfView :FLOAT,
                       aspectRatio  :FLOAT,
                       nearPlane    :FLOAT,
                       farPlane     :FLOAT ) :HRESULT
+ xRotateMatrix( &matrix :D3DMATRIX, theta :FLOAT ) :void
+ yRotateMatrix( &matrix :D3DMATRIX, theta :FLOAT ) :void
+ zRotateMatrix( &matrix :D3DMATRIX, theta :FLOAT ) :void
+ translateMatrix( &matrix :D3DMATRIX,
                  x :FLOAT, y :FLOAT, z :FLOAT) :void
+ scaleMatrix( &matrix :D3DMATRIX,
               x :FLOAT, y :FLOAT, z :FLOAT) :void
+ multiplyMatrices( a :D3DMATRIX, b :D3DMATRIX ) :D3DMATRIX
+ normalizeVector( &a :D3DVECTOR ) :void
+ crossProduct( a :D3DVECTOR, b :D3DVECTOR ) :D3DVECTOR
+ distanceBetween( a :D3DVECTOR, b :D3DVECTOR ) :FLOAT
+ translateVector( D3DVECTOR &vector, FLOAT x, FLOAT y, FLOAT z) :VOID
+ translateVector( D3DVECTOR &vector, D3DVECTOR translationVector) :VOID
+ xRotateVector( &vector :D3DVECTOR, theta :FLOAT) :VOID
+ yRotateVector( &vector :D3DVECTOR, theta :FLOAT) :VOID
+ zRotateVector( &vector :D3DVECTOR, theta :FLOAT) :VOID
+ multiplyVectorMatrix( vector :D3DVECTOR,
                       matrix :D3DMATRIX) :D3DVECTOR
```

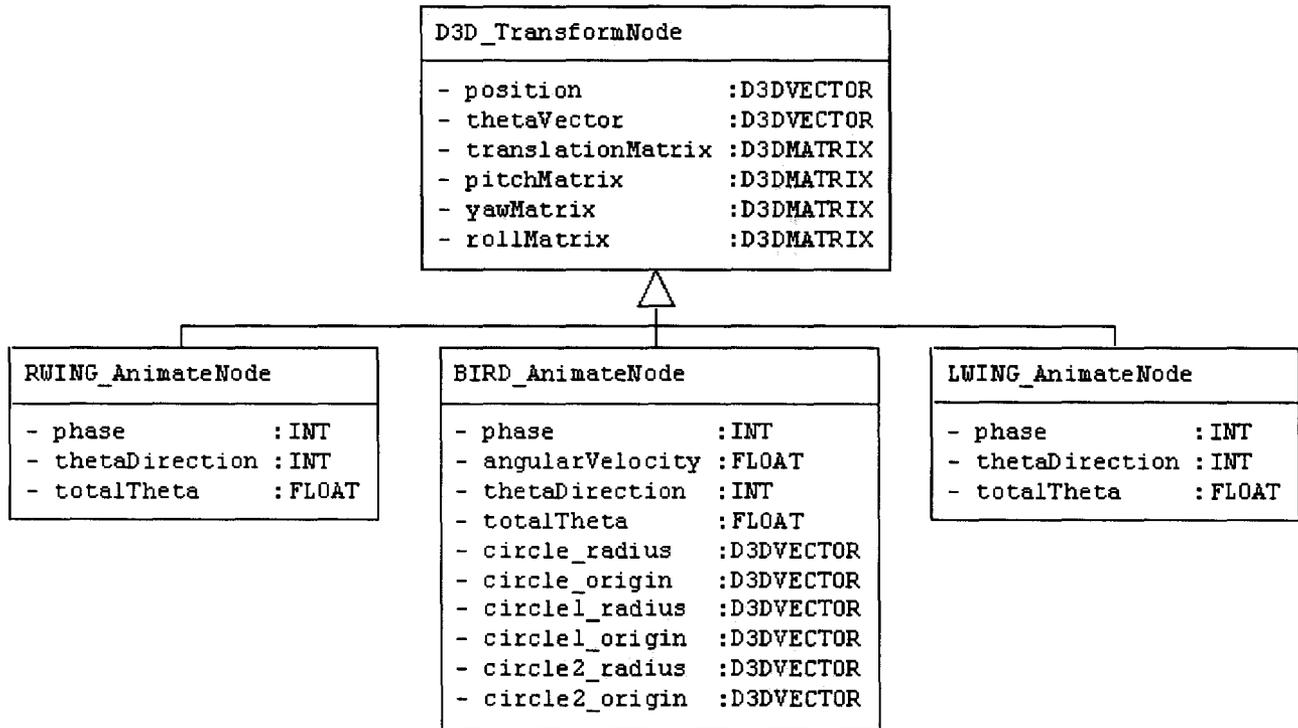
- RELATIONSHIP DIAGRAM -  
- SCENE GRAPH -



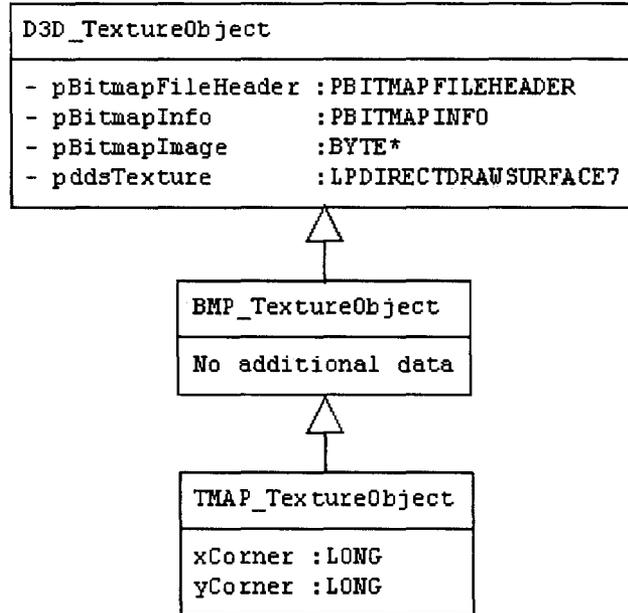
- INHERITANCE DIAGRAM -  
- SCENE OBJECT -



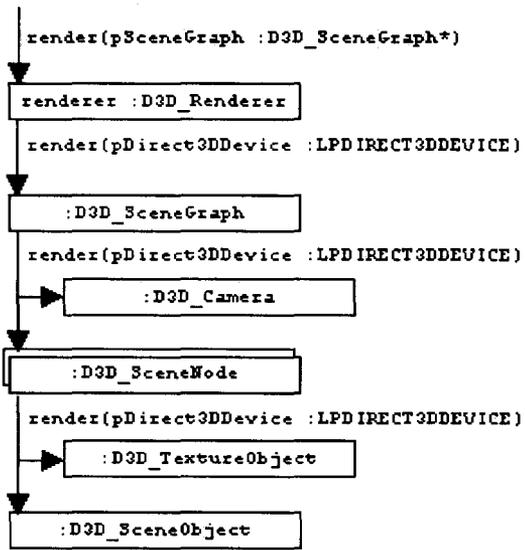
- INHERITANCE DIAGRAM -  
- TRANSFORM NODE -



- INHERITANCE DIAGRAM -  
- TEXTURE OBJECT -



- INTERACTION DIAGRAM -  
- RENDERING PROCESS -



- INTERACTION DIAGRAM -

- ANIMATION PROCESS -

