CS 1120A

Console Game Development

David Beirne
0705140

PS2X

Loading & Rendering X Files on The Playstation 2

30/03/2008

Contents

1 Abstract

This paper describes and evaluates the design and implementation of a DirectX .X File loader for the Playstation 2.

David Beirne
BSc(Hons) Computer Games Programming
University of Abertay Dundee

2 Introduction

PS2X is a DirectX .X File loader for the Sony Playstation 2 console. The aim of PS2X was to improve upon the mesh loading functionality of the PS2 Game Framework developed by Dr H S Fortuna [CS1120A Console Game Development Course Material,2008, Dr Henry Fortuna]. In addition to being a more widely used file format than the milkshape .ms3d, the X File implantation offers a number of extra features including a hierarchical transformation structure, automatically generated bounding volumes (sphere and boxes), multiple meshes in a single file, multiple materials, multiple textures and time-based key frame animation.

This report will introduce the X File format, explaining how an X File can be read both by a human and a machine. Next the report outlines how PS2X was implemented, the key classes and methods and their design rationale. The next section explains how to operate the PS2X program, including the necessary command line arguments and in-game controls. Finally a critical analysis discusses the limitations and successes of the current system.

To accompany this document, a full code listing and comment generated source documentation site is available at the following URL:

http://www.mygamedemos.com/projects/PS2X/doxy/

A copy of the source documentation is also available on the attached CD under /Framework/doxygen/html/index.html

3 Introducing the X File Format

The X File Format was introduced with the release of Microsoft DirectX 3.0 as a standard format for storing 3D model data, transformations, and animations within the DirectX API. Since its initial release, the format has gone through a number of versions with the current being 3.03 (introduced with DirectX 9.0c). While primarily used as a means of storing 3D model and animation data, the X File format itself is an architecture, and context-free file format capable of storing any data.

The X File format is template driven; a template defines how a data stream of the same type as the template is interpreted. A template has the following form:

```
template <template-name>
{
      <GUID>
      <member 1>;
      ...
      <member n>;
      [restrictions]
}
```

The template name can be any alphanumeric name which may include the underscore character but may not begin with a number. The GUID (Globally Unique Identifier) member is a value which uniquely

identifies the template surrounded by angled brackets, for example <3F2504E0-4F89-11D3-9A0C-0305E82C3301>.

Templates may be open, closed, or restricted. These restrictions determine which data types may appear in the immediate hierarchy of a data object defined by the template. An open template has no restrictions, a closed template rejects all data types, and a restricted template allows a named list of data types. Template members consist of a named data type followed by an optional name or an array of a named data type. Accepted primitive data types are:

Data Type	Size
WORD	16 bits
DWORD	32 bits
FLOAT	IEEE Float
DOUBLE	64 bits
CHAR	8 bits
UCHAR	8 bits
BYTE	8 bits
STRING	NULL-Terminated String

Additional data types, defined by other templates encountered earlier in the data stream, can also be referenced within a template definition. No forward references are allowed.

A data type may also be expressed in an array within a template definition. A standard array definition would consist of the following elements:

```
array <data-type> <name>[<size>];
```

The data-type of the array may be any of those listed in the table above or any previously defined template type. Size may be an integer or a reference to another template member whose value is then substituted. Arrays may be n-dimensional, where n is determined by the number of paired square brackets trailing the statement. For example:

```
array DWORD FixedHerd[24];
array DWORD Herd[nCows];
array FLOAT Matrix4x4[4][4];
```

While the DirectX API defines a number of standard templates, including Mesh, Frame, Material, & Animation among many others, the X File format is not restricted to, or limited by, this set of templates. As an open format, it is possible to create and register custom any valid new templates. Below is an example of how a Vector and MeshFace template are defined, and how these are used to compose the definition of the Mesh template:

```
// Vector template states that a vector holds 3 floats (x,y,z). Template is closed
so cannot contain anything else.
template Vector
{
    // Template GUID
    < 3D82AB5E-62DA-11cf-AB39-0020AF71E433 >
    //Template Members
    float x;
    float y;
    float z;
}

// MeshFace template states that a MeshFace holds the DWORD nFaceVertexIndices
containing the number of face indices of in the mesh face, and an array of DWORDS of
size nFaceVertexIndices containing the face indices of this mesh face. Template is
closed so cannot contain anything else.
template MeshFace
{
    // Template GUID
    < 3D82AB5F-62DA-11cf-AB39-0020AF71E433 >
    // Template primitive data Memeber
    DWORD nFaceVertexIndices;
    // Template Array Member
    array DWORD faceVertexIndices[nFaceVertexIndices];
}

//The Mesh template is a open template containing at least a DWORD nVertices
containing the number of vertices in the mesh, an array of type Vector (described by
the Vector template), a DWORD contining the number of faces in the mesh and an array
of type MeshFace (defined by the MeshFace template) of size nFaces. Template is open
so can contain any other template.
template Mesh
{
    // Template GUID
    <3D82AB44-62DA-11cf-AB39-0020AF71E433>
    // Template primitive data member
    DWORD nVertices;
    // Template Array of Vector Type (reference to earlier definition of type Vector)
    array Vector vertices[nVertices];
    // Template primitive data member.
    DWORD nFaces;
    // Template Array of type MeshFace (reference to earlier definition of MeshFace)
    array MeshFace faces[nFaces];
    // Open restriction (can contain 0 or more other templates)
    [ ... ]
}
```

Data objects contain the actual data with a corresponding template type which identifies how the data should be read. A Data Object, in this sense can be considered an instance of its respective template. A data object is defined by first its template name, then an optional instance name, followed by a pair of braces encapsulating the objects data:

```
// Data Object of type Mesh named CubeMesh.
Mesh CubeMesh
{
```

```
    8;                                      // nVertices = 8
    1.000000;1.000000;-1.000000;,           // vertices[0] = Vector{1.0;1.0;-1.0}
    -1.000000;1.000000;-1.000000;,          // vertices[1] = Vector{-1.0;1.0;1.0}
    -1.000000;1.000000;1.000000;,           // etc...
    1.000000;1.000000;1.000000;,
    1.000000;-1.000000;-1.000000;,
    -1.000000;-1.000000;-1.000000;,
    -1.000000;-1.000000;1.000000;,
    1.000000;-1.000000;1.000000;;           // ';;' indicates the end of a containter

    12;                                     // nFaces = 12
    3;0,1,2;,                               // face 0 has 3 vertices
    3;0,2,3;,                               // etc...
    3;0,4,5;,
    3;0,5,1;,
    3;1,5,6;,
    3;1,6,2;,
    3;2,6,7;,
    3;2,7,3;,
    3;3,7,4;,
    3;3,4,0;,
    3;4,7,6;,
    3;4,6,5;;                               // ';;' indicates the end of a containter
}
```

Data objects defined by open templates may be nested either inline or by reference to describe a hierarchy:

```
// Data Object of type Mesh named CubeMesh.
Mesh CubeMesh
{
    ....                            // mesh data here (as above)

    //Mesh contains an instance of the Material template
    Material Box_Material
    {
          ... // Material Data

          // Material Contains an instance of Texture File Name template
          TextureFileName Box_Texture
          {
                "diffuse.bmp";   // Texture File Name data
          }

          // Material Contains an EffectInstance Template Instance
          EffectIntance
          {
                "reflection.fx";        // EffectInstance data

                // EffectInstance contains EffectParamsDWORD instance (named Time)

                EffectParamDWord Time
                {
                      100;        //EffectParamsDWord data
                }
          }
```

```
    }
}
```

In the example above, the Material, TextureFileName, EffectInstance, and EffectParamDWord Templates are assumed to be defined.

An X File may be ASCII text or binary encoded, as defined in its Header template. In its ASCII form, an X file is easily human readable and editable. The X file is also a long established and widely used 3d model and animation format within the DirectX game development community, but, due to the lack of helper functions in other API's it is less common. At this time, there are quite a few X File loaders for OpenGL but currently no public X File loader for the Playstation 2 (although it is likely one has been written within a game development company).

4 Implementation

4.1 Structure

In order to maintain as much platform independent code as possible, the program is split into two distinct parts: a platform independent X File loader and a (PS2) platform specific renderer.

4.1.1 X File Loader

The Aim of the X File loader is to open the file, to parse through it and assign all supported data to the relevant data structures. There is an equivalent data structure for each of the supported templates defined in "XFile_Defines.h" which also defines an ID value for each supported template and contains an array of stXDefines which is a struct matching a defined template ID to the string that defines that template within the X File. For example the id X_Mesh is linked to the string "Mesh".

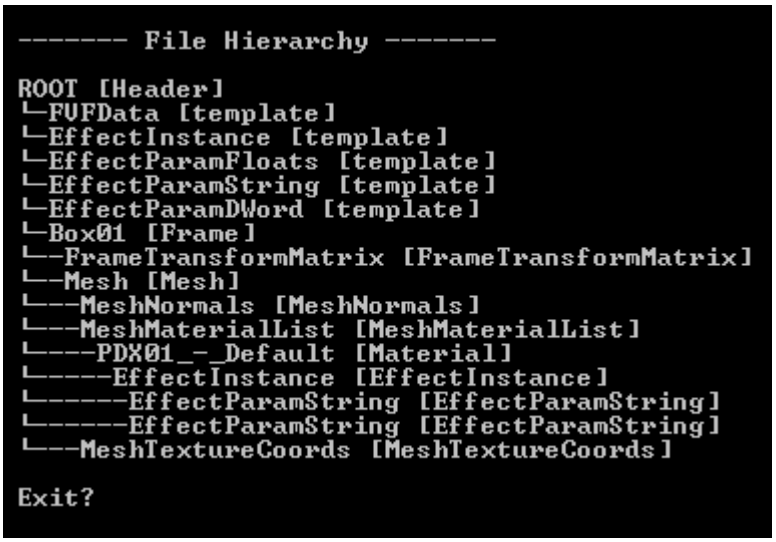
The X File loader loops through the X File and every time it encounters one of these template identifying strings, it creates a new stTemplate instance and pushes it onto a 'parents' stack. An stTemplate is a generic container for any template instance (data object) described by the X File. The stTemplate contains the data objects name, its stXDefines type identifier, and a void pointer to the actual data structure containing the template instances data. Using the stXDefines type, it is possible to cast the data back into its native format.

The processBlocks function of the X File parser switches on the type of template instance discovered and calls a corresponding parse function to process all data within the block (between a set of braces). For example if the current template type is X_ANIMATION, processAnimation is called which processes all animation data within the current block and returns it in an Animation struct into the pData pointer of the current stTemplate. Because data objects (template instances) can be nested, the current templates parent is considered to be

the top of the 'parents' stack which is pushed every time a new template is encountered (at an opening brace) and popped once a block ends (on encountering a closing brace).

The stTemplate struct also contains a pointer to a parent stTemplate (essentially forming a linked list of data objects) and a vector of pointers to all children of that template (which avoids the need to search for children or siblings in the hierarchy). When a template instance has no parent within the X File, the parent value is assigned to a common root stTemplate creating an n-dimensional tree hierarchy of the data within the X File.

The X File loader is separate to the renderer so it also contains functions to display its data hierarchy as a text output, see below:



In the text output above, data objects are displayed by name and template type (in square brackets), along with their hierarchical relationship represented by a hyphen indicating their 'depth' within the hierarchy. It can be seen that, for example, the templates FVFData,EffectInstance, EffectParamFloats,EffectPramString and EffectParamDWord have been defined at the start of the file and as such are children of the header 'ROOT' data object . The file then contains an instance of type Frame called 'Box01' which contains an instance of FrameTransformMatrix, which contains an instance of Mesh which, in turn, contains an instance of MeshNormals, MeshTextureCoords and MeshMaterialList. The MeshMaterialList contains an instance of Material (named PDX01_-_Defailt) which contains an EffectIntance containing two string parameter instances of type EffectParamString.

It should be noted that in the above example, the X File is, strictly-speaking, a malformed definition as it references templates (Frame, Mesh, MeshNormals, MeshMaterialList, Material and MeshTextureCoords) undefined within the file. However, since these templates have been defined as supported types in "XFile_Defines.h", the parser has picked them up anyway. Had the X-File encountered a totally undefined template type, it would assign the default X_UNSUPPORTED type to the template instance and skip over the block.

Aside from the entire file hierarchy, the X File parser also constructs a separate frame hierarchy as a linked list with a common root (rootFrame) which is used for transforming the meshes of an X File both through animation and external means. The X File also contains flat lists (in this case vectors) of pointers to all frames, meshes, materials, effect instances, texture file names, and animation sets encountered during the parse of the file. This makes accessing these specific data structures easier in the Renderer where, for example, each texture file name is used to specify the file names of textures to be loaded into a PS2 specific CTexture instance.

4.1.2 X File Renderer

The X File Renderer uses an X File loader to load an X File into a hierarchical structure, then; using the data in the hierarchy loads external texture resources, constructs subset meshes, updates, animates and renders the X File on the target platform.

The X File Renderer structure follows the commonly used initialise, load, update, render, clean up data flow where initialise and load are called prior to frame-time to initialise all memory and load in any external data. The render function is called every time the object should be drawn to the screen, and update is called whenever the objects transformation changes (either via an external transformation or through animation). The cleanup function then clears any memory used by the object on deletion.

4.1.2.1 Initialise

The initialise method is invoked during the objects construction and on each call to load via the reset function (which cleans up then re-initialises the object). Initialise sets all internal attributes of the X File Renderer to default values.

4.1.2.2 Load

The load method is used to load a specified X File into the internal X File Loader instance of the X File Renderer. The load function is then responsible for using the data within the internal X File Loader to perform any (PS2) platform specific operations. In the current implementation, the load function takes the following steps:

4.1.2.2.1 Load Textures

Texture data is not stored within an X File; instead the textureFileName template specifies an external file where the texture data can be loaded from. As mentioned earlier, the X File Loader maintains a flat-list of all textureFileName instances encountered as it parses the file. The loadTextures function of the XFile Renderer is responsible for taking these file names and loading in the specified texture file data.

In order to maintain a clear connection between each texture and its textureFileName definition in the X File Loader, the mappedTexture struct (defined in xFile_Render.h) contains a pointer to a CTexture instance along with a string containing the texture file name as specified in the textureFileName struct which acts as a unique identifier for the texture within the scope of the X File Renderer instance.

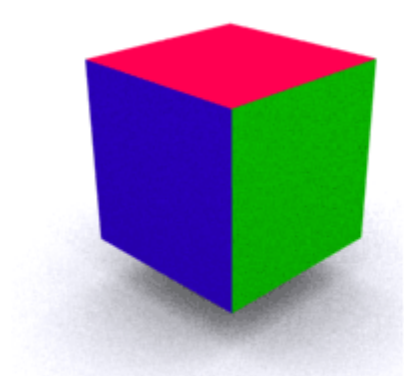
The loadTextures method creates an array of mappedTextures (stored in 'textures') of equal size to the number of textureFileName instances found by the X File Loader. The loadTextures method then iterates through each textureFileName and creates a mapped texture instance containing the file name of the current texture and its data in a CTexture instance. If a CTexture fails to load, a NULL pointer is stored.

The texture loading capabilities of the X File Renderer are currently limited by the CTexture class provided by the PS2 Game Framework by Dr. Henry S Fortuna [CS1120A Console Game Development Course Material, 2008, Dr Henry Fortuna] as defined in texture.h and implemented in texture.cpp. The current limitations require textures to be of bitmap (.bmp) format with a width and height of a power of 2 with 1, 4, 8, or 24 bits per pixel. If a texture is of a different format, it will fail to load – an error message will be output to the CLI and a NULL pointer will be stored. The X File will still render but the failed texture will not be visible.

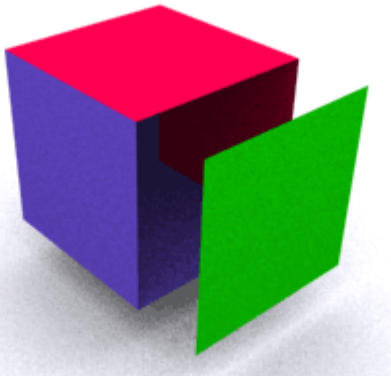
4.1.2.2.2 Create Subsets

A subset is a set of primitives (triangles) in a mesh which share the same material. For performance reasons, it is necessary to pre-sort the triangles of a mesh (or better yet all meshes) into subsets to minimise the number of material, texture, and render-state changes per-frame when drawing the mesh. As subsets are drawn separately, they can be considered separate meshes.

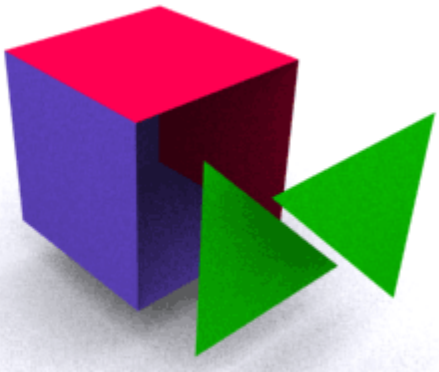
There is one SubsetMesh for every Mesh loaded in by the X File Loader. Each SubsetMesh has one Subset for every Material used in the mesh, each Subset contains a list of the triangles that belong to that Subset.



Single SubsetMesh with 3 Materials (therefore 3 Subsets)



Single (Green) Subset



Individual triangles of the Subset

Once the Subset structure is populated, the data for that subset is added to the Playstation 2 Static DMA in the method AddSubsetToDMA(). AddSubsetToDMA adds each vertex, texture coordinate and normal of each triangle in the subset to the static

memory in 4K chunks and returns the index of the first vertex in static memory. If Normals or Texture Coordinates are not defined by the vertex format of the subsets triangles, default values are set and the user is warned via text output to the CLI, this enables non-standard meshes to be visualized on position of vertices alone and increases compatibility with different X Files.

Once the AddSubsetToDMA() method has looped through all meshes loaded into the X File Loader and created SubsetMeshes, containing Subsets which list each triangle within the subset, its material, the index of its texture in the textures array (or -1 if the material has no texture), and its index in static memory, the method returns.

4.1.2.2.3 Set Animation

The X File Renderer contains a Boolean value 'isAnimated' and an AnimationSet pointer 'setAnimationSet' which points to the currently set AnimationSet of the X File Renderer. If one or more AnimationSet data objects (template instances) have been found by the X File Loader, setAnimationSet is assigned a pointer to the first AnimationSet of the file and isAnimated is set to true. By default setAnimationSet is NULL and isAnimated is false.

If the X File contains animation data, SetAnimation can be called at any time to change which animation the X File Renderer is playing.

4.1.2.2.4 Perform Initial Update

In order for animations and bounding boxes to be assigned correctly after the X File Renderer is initialised, it is necessary to perform an initial call to the Update function (described in the next section). This will set all frame transformations to those found initially in the file (prior to any animations taking place) allowing animations / transformations to be applied with the base transformation in the correct position.

4.1.2.3 Render

The render method is called every time the X File is drawn to the screen. Render takes the current time as a parameter which enables it to call updateAnimation() with the current time if a file is animated. The render method calls the following methods:

4.1.2.3.1 Update Animation

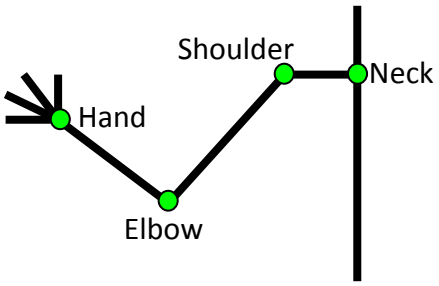
The updateAnimation method is called by Render if the X File is animated and an animation is set. The updateAnimation method iterates through all animation data in the currently set animation (setAnimationSet). If the animation is bound to a bone (frame transformation), then the method determines which is the key frame with the closest time to that input to the method as a parameter. Next a linear interpolation based on time between the current keyframe and the next keyframe closest to the current time which results in the transformation at the current time. The resulting transformation is then applied to the frameMatrix of the frameTransformation to which the bone is

mapped. On the next call to Update, the frameMatrix is applied to the frame resulting in the final transformation.

4.1.2.3.2 Update

The Update method is used to apply an update to the matrix transformations of the mesh hierarchy. The matrices need only be updated when a mesh is moved by an external transformation or animation has been applied to the mesh.. Update is called automatically by the Render function when an external transformation has been evoked and the "changed" Boolean attribute of the XFile Renderer class has been set to true (via the setGlobalPose member function). setGlobalPose() and Render can be called multiple times per-frame in order to draw instances of the same X File at different transformations.

Update is a recursive function which takes, as a parameter, a pointer to a Frame instance. Generally Update is called with a pointer to the root frame stored in the X File Loader but it is possible to call it with another frame in order to update the transformation on only a part of the X Files frame transformation hierarchy. Update takes the parent frames transformation matrix and multiplies it with that of all of the frames children, then calls itself to apply the transformation to all of the children of the children etc, effectively applying the transformation of all parents to all children through the frame hierarchy. For example a transformation to the upper arm bone of a character will be applied to the forearm; the transformed forearm transformation will be applied to the hand, the hand to each of the fingers etc:



Bone	Transformation
Neck =	Neck
Shoulder =	Shoulder x Neck
Elbow =	Elbow x Shoulder x Neck
Hand =	Hand x Elbow x Shoulder x Neck

4.1.2.3.3 Draw Mesh

The DrawMesh method is responsible for uploading the light directions and colours, setting the world and world-view-projection matrix and then calling the vertex data from static memory, creating the final GS packet for each subset and sending it to VU1 to be rendered. The render method loops through all subset meshes rendering each subset.

4.1.2.4 Clean Up

This method is called by the X File Render's destructor and clears up any memory used by the X File Renderer.

4.2 Operation

4.2.1 Running the Program

The program runs from /Framework/main

By default, the program will load tiny.x but the program can be configured using the using the following command line arguments.

4.2.2 Command Line Arguments

X Files are loaded via the first command line argument:

Number	Name	Type	Usage	Example
1	File Name	string	Specifies which X File is opened (case sensitive).	./main animation.x
2	Verbose Output	bool	1 for debug text output to the CLI or 0 for no debug text.	./main animation2.x 1
3	Draw Bounding Boxes	bool	1 to draw bounding boxes on all meshes or 0 to not draw bounds.	./main animation2.x 0 1

4.2.2.1 X Files available

PS2X comes with a number of X Files in the programs directory, below is a list of the X Files available:

File Name	Description
animation.x	A metal pendulum swinging over a ground plane.
animation2.x	Falling crates and balls physics animation.
animation3.x	Braking teapot and physics crates animation.
animation4.x	Dancing blue skeleton model.
bird.x	Walking bird animation.
Box.x	Static crate mesh.
Sphere.x	Static sphere mesh.
tiny.x	Tiny skinned character model from DirectX SDK.
Track.x	Race Track Mesh (no textures).

4.2.3 In Game Operation

The in-game controls allow the camera to be moved with the left analogue stick and rotated with the right. The up and down d-pad directions increase and decrease the speed of the animation playing in the current X File Renderer and the Cross button resets the speed to its default value. The triangle face button takes a screenshot and saves it to a .tga image file in the programs directory. The Start and Back keys pressed to together exit the program.

5 Critical Analysis

5.1 Limitations

The list below summarises the limitations of the current solution:

5.1.1 Text Only

The X File loader can only load ASCII encoded X Files which fit the current (0303txt) text specification. A future implementation could include binary support.

5.1.2 No Skinned Mesh

In its current state, the X File loader does not load skinning data. The reason for this is that PS2X was implemented in the standard version of the PS2 Game Framework which uses a standard Vector Unit program. To implement skinning, it would be necessary to write a VU program to apply the sum of all connected bone weights to each vertex.

5.1.3 Material Properties

Material properties such as diffuse, ambient, emissive and specula power are loaded but have no effect on the final rendered output. This is because the vector unit program has not been altered to take the values into account.

5.2 Successes

5.2.1 Completeness

Aside from the 3 limitations listed above, the X File loader is complete and able to load any X File in the 0303txt format successfully. The X File loader also has a lot of debug output when the verbose command line argument is initialized to true, which would help in fixing any problems encountered in extending functionality.

5.2.2 Design

The program is very clearly designed with a definitive split between the platform independent loader and the PS2 specific renderer. Due to the design, implementing further functionality (such as skinning or binary file loading) should be relatively straight-forward.

5.2.3 Documentation & Commenting

The PS2X project is very heavily commented in the doxygen style allowing source code documentation and class diagrams to be generated automatically. The high level of code documentation, along with this explanatory report, should also allow future programmers to be able to understand the X File loader if it is included into the PS2 Framework.

6 Conclusion

Through this project, I have learned how to successfully implement a fairly complex system in what was originally an unfamiliar environment. I have learned a lot more about the low-level operations of the Playstation 2 and consoles in general. The ability to follow the rendering pipeline and memory allocation at such a low level has certainly proved beneficial to my overall understanding of game programming and the way everything fits together in the bigger picture – more specifically I have found that a greater understanding of the underlying hardware can lead to much more efficient and effective software.

7 References

1. Advanced Animation with DirectX, 2003, Jim Adams, Premier Press, ISBN 1-59200-037-1

2. Direct-X File Format,1999, Paul Bourke, http://local.wasp.uwa.edu.au/~pbourke/dataformats/directx/

3. Direct X 3D File Format,2005, Ben Kenwright, http://www.xbdev.net/3dformats/x/xfileformat.php

4. Loading & Displaying .X Files without DirectX, 2001, Paul Coppens, http://www.gamedev.net/reference/programming/features/xfilepc/

5. CS1120A Console Game Development Course Material,2008, Dr Henry Fortuna, http://www.hsfortuna.pwp.blueyonder.co.uk/

6. X Format File Reference, MSDN, Microsoft, http://msdn.microsoft.com/en-us/library/bb173014.aspx