COLLADA: Geometry and Animation loading

Author: Celal Cansin Kayi

TheCansin.com

Dedicated to a very special Italian person who means a lot to me.

Contents

Introduction

I find it oddly humorous that something as malignant and useless as the “console war” is fought tooth and nail by eager fanboys; whilst a very important issue, the intermediate content format of choice for digital content creation applications; of which three of the most popular at the least are maintained in almost a monopoly by AutoDesk, is largely ignored.

This sad state of affairs leaves anyone new to games programming, anyone without a team with a decent amount of funding and anyone wanting to program anything that requires 3D mesh data more complex then a basic cube, **completely fucked**.

The main problem really is that there are several formats being pushed and one of which is being pushed by AutoDesk, who I would call the Apple or Microsoft in the realm of digital content creation applications.

What I mean is, of all the content formats being pushed .fbx has a very powerful and influential company behind it and that would be laudable if .fbx was a good format, **it isn’t**.

AutoDesk have definitely put a lot of effort into .fbx, in terms of the scheme of the data, the documentation, community support and samples provided for free.
This requires a mention as it is something they should be commended on.

However .fbx is still a closed format that is unreadable for humans and not parse able without their libraries as well as still having the inability to export Tangents and Bi-Tangents (at this current time of writing) from their own exporters or any that I know of.
Writing code to calculate such things isn’t too difficult but it’s just a very unnecessary chore that you’re forced to deal with when implementing .fbx support in any application outside of their own.

The problem COLLADA has is that it’s supported completely in open source ventures, whilst the community and their effort has been extraordinary, the documentation and beginner friendly implementations have been lacking, especially for DirectX.

Though this is the case with all the content formats really, even in the .fbx documentation there is almost **zero mention** of how to actually **read the data**, there is only a mention on how to make the data and once again nothing for DirectX outside of one open source project which is about as readable as Aramaic.

I can understand the reason for such nonexistent DirectX examples, due to its nature in the requirements of set up for a simple application. OpenGL is much

easier on the eyes for examples and I guess it can be adapted easily to DirectX for an experienced programmer.

I found however that learning programming for games almost always skips the issue of content formats; it is the extremely rare case of learning material for game programming that actually covers any format outside of the archaic .3ds or any format that will describe skeletal animation at all.

If someone wants to get into programming for games outside of XNA or any API that provides its own content pipeline, then they will hit a metaphorical titanium wall with death cannons, controlled by a SHODAN-style sentient artificial intelligence, targeting them as soon as they have reached learning about content pipeline creation.

If you are an experienced programmer, chances are you already either have a job and are working in a team of some sort or you at least have some colleagues to discuss with, both of which would enable you access to someone who has dealt with this issue before in some capacity, if not you would be in the same boat as me; which is to say lots and lots of hair pulling and frustration culminating in a murderous rage and broken spirit that finally ends in success after a surprising amount of time spent working on it.

I want to encourage people into games programming and the experience I have had learning it has been a nightmare, content pipeline comes at the worst imaginable time and completely broke a lot of my enthusiasm for programming as a beginner due to the confounding lack of support; the level of negligence in educating this topic is almost criminal(look at the various books boasting teaching you “how to make games” or teaching the learning of “Model Animation”, almost always using premade .x files or .obj with no skinned animation etc. and completely sidesteps the issue without mentioning it) and it is one of the most important topics that a beginner will face.

Learning how to set up an effective and easy to use content pipeline that supports everything your artists need is an essential concept for a games programmer, the mesh and animation content is by far the most important for a 3D game therefore COLLADA, being the most viable in delivering such content from a DCC application to your own, is integral. The only other option is learning how to manipulate the DCC application of your choice to craft a self made exporter, which has the same level of difficulty as training a magical leprechaun army and taking over the universe.

If you haven’t yet made up your mind on which format you want to support for intermediate content (that is, that you are sure you WANT an intermediate format but just aren’t sure which format is the better choice for you) I heartily recommend COLLADA. Why? A few simple reasons:

•Human-readable, thus allowing you to see if there are any errors in export from DCC package, or any errors on import into your own software and generally aids a lot in programming for it

•Easy to program for, it's basically just an XML

•Very tightly defined format, thus making sure your code is safe with almost everything you'll throw at it, .dae wise

That's not to say COLLADA doesn't have disadvantages though, the only real one that I have faced is that since almost everything to do with COLLADA is through open-source and all the "official" plugins for the popular DCC applications can't be counted on at all it leads to a sort of "wild west" with COLLADA exporters where each has a different style of exporting data.

Open-source is good, but a dedicated team that is actually paid will usually do a better job and a unified stance on data export needs to be properly specified. Khronos really needs to specify which exporters are the "official" and make sure that all other exporters made for other programs and future revisions follow their mold so you don't have to consistently update your own importer.

One example is how I have encountered two different types of exporters for COLLADA in terms of geometry, one that exports triangles as one big list and the other as each triangle in separate groups, each way has its own distinct advantages but programming a solution to cater for both is just a massive annoying chore and it's stuff like that which **heavily degrades the point of COLLADA**.

Enough said about the "content war", my aim for this document is to help all those who are trying to set up a content pipeline. I went through hell and back trying to get the skinned animation to work fully in DirectX as a beginner and the lack of education material on this topic drove me to insanity, I feel for all you who are programming without much help just as I was and still am. Thus this document is born, onwards to COLLADA and may we never look back!

Chapter 1: Before we get started

Prerequisites

I assume you should already know how to program in C++ and its standard runtime library; however I have made a lot of concessions in the code for a beginner. You should also know at the least the basics of HLSL/GLSL/CG and general rendering basics as well as an understanding of basic trigonometry, matrices, vectors and quaternions. You should also understand the 3D API of your choice, OpenGL or Direct3D. If you are programming for XNA or C# you can probably still follow along, though since XNA has animation built in I doubt your need for this.

Who this document is for

This document is for anyone who meets the prerequisites as detailed above who wants a thorough understanding of COLLADA, its DOM and skeletal animation in general. You can consider this like the unofficial documentation almost.

What this document does not cover

Anything aside from the geometry, skeletal animation and morphing animation will not be covered. To put it another way: materials, effects, physics etc. will **not** be covered, also any "special case" geometry will not be covered. However understand that this document will teach you enough about COLLADA and its DOM so that you will be able to get any more data from a COLLADA file easily.

Where to get the source code

In case you got here from somewhere else, go to: thecansin.com

A VERY important note

This document is written with DirectX in mind, as OpenGL has lots of COLLADA examples. However if you want to use this for OpenGL you can still follow along as there is nothing really API specific until the appendix, understand though that I do use the D3DX structures for matrices and other such things, but once again **all data loaded will be loaded straight from COLLADA and only converted to be DirectX friendly at a final stage that is removed from the reading of the COLLADA file**.

Also COLLADA is very well defined, but there is still no "official" exporter for the main DCC packages and as such results exported may vary with each, this text was written while using the COLLADAMaya exporter with Maya, it also was written for triangulated and indexed models only. However by the end of chapter 3 you will know enough about using the COLLADA DOM that this won't matter, as you'll find that it's a very versatile, easy to use interface that will allow you to get the data you want very easily.

One more thing, if you don't care about learning COLLADA and just simply want a C++ importer, just use the "COLLADADirectX" project's COLLADALoader class.

Chapter 2: A brief primer on COLLADA and XML

You can skip this chapter if you already know XML and COLLADA terminology etc. Just make sure to read the note on matrices.

Some basic XML

XML is fairly simple, take the following XML style example:

```
<Father>
        <Son>Barry</Son>
        <Son>Jeff</Son>
        <Daughter biological="no">Annie</Daughter>
</Father>
```

Terminology:

Tag/Element
Attribute type
<>/ Attribute data
Data

What you as a programmer need to understand from this is what's between the tags (<Father> is a tag) and what can precede such a tag. All tags must be closed with its corresponding </whatever> or a tag can close itself by having a "/" at the end of itself.

The information you should take from this is:

- <Father> is a **tag**, but in COLLADA when you see such things, they are referred to as "**elements**", every **tag** will have a closing **tag** </whatever the start was> and a **tag** can have other **tags** within it.
- <Son> is a **tag**, but it is also a **child** of <Father> as it is nested within its opening and closing **tags**. <Son> also has information between it's **tags**, this is often the case in a COLLADA document and is where the information you will want will usually be
- <Daughter> is a **tag**, and just like <Son> it too is a **child** of <Father>, you may notice that this time it has an extra bit of text in the tag, this is an **attribute** of that element
- A **tag** holds its information **between** the start and the end tag

At this point you should understand some basic terminology of tags, children, elements and attributes. This is mostly all you need to understand a .dae file.

How XML relates to COLLADA

COLLADA is really just an XML with a schema; a schema is just a document that outlines all the elements, attributes and format any such file that is under it will be in.

An important distinction

Before you start programming for COLLADA first you must understand the following example:

```
<library_geometries>
        <geometry id="cubeshape" name="cubeshape">
                <mesh>
                        <source id="cubeshape-positions" name="position">
                                <float_array count="4">12.0 5.0 2.0
3.1</float_array>
                        </source>
                </mesh>
        </geometry >
</library_geometries>
```

That may be somewhat confusing at the moment but you'll get used to this format quickly, especially because in any XML editor you'll be able to outline the tags better.

What you need to understand from this is the difference between **child** and **descendant**. <library_geometries> has one **child** <geometry>; but its **descendants** are <geometry>, <mesh> and <source>. So suppose you had a way to read COLLADA files in C++ and it had two functions, getChildren() and getDescendants(), each returning a std::vector of pointers to each element in memory.
If you used getChildren you would get a pointer to the geometry node.
If you used getDescendants you would get a pointer to the geometry node, the mesh node and the source node.
Suppose you had two other functions, getChild(std::string name) and getDescendant(std::string name) that will find a child or a descendant by name of the element. Don't make the mistake of thinking that searching <library_geometries> **children** for a <mesh> element will work, if you instead search its **descendants** it will find the node.
This is an important distinction and will come in handy when you parse a .dae file.

A brief overview of COLLADA

COLLADA is very well defined in terms of the elements that will be within any .dae file. All the data is organized under specific "library" tags and everything references each other so for a given mesh you can get all the elements you want

easily and discard everything else just as easily. It allows you to read only what you need to read and makes everything connected, but not dependant on each other. For example, if you have an animated mesh but you just want to read the geometry then you can just do that very easily.
COLLADA organizes each of its data types in to <library_whatever> tags, the libraries we will be interested for this document will be:

- <library_geometries>
- <library_animations>
- <library_controllers>
- <library_visual_scenes>

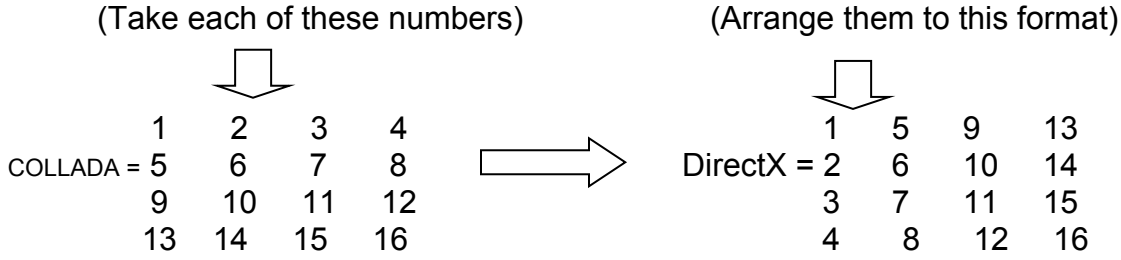
You'll notice that a COLLADA file also has an <asset> tag at the start and a <scene> tag at the end.
The <asset> tag will tell you how the file was exported, when it was exported, where to find the base file for the DCC package it was created in, its coordinate system and etc. You won't really need to worry about the data it has but you should make sure to set your own rules on how files should be exported and etc to make things consistent, especially coordinate wise.
The <scene> tag is there because, if you have ever programmed game physics etc. you'll understand, but basically in any game physics simulation you usually have two representations of a model: one that is for rendering, which can be thousands or millions of triangles and one that is for physics analysis which is usually a simple shape like a box or a cylinder etc or generally an n-sided polyhedra. You won't have to worry about this tag unless you want to support physics through COLLADA but it will basically tell you where to find which representation of the scene.

COLLADA also has data types; you'll notice throughout a typical .dae file that there are <float_array> elements and etc. The data types COLLADA has all pretty much correspond to C++, they may have different names though. For example, a <Name_array> is more like a string array.

I will provide a detailed explanation of each library_ as we use them rather then clogging this one space with everything. It's better to compound your knowledge of the libraries as you go with COLLADA.

Another VERY important note: COLLADA matrices

COLLADA was developed as a joint effort between Sony and khronos, khronos are the group that brought the OpenGL API and as such, COLLADA follows with OpenGL specifications. That is to say, COLLADA is a right handed coordinate system, but it uses column vectors in row-major matrices. So the conversion to DirectX would be as follows:



That's basically the conversion for the matrices, as for RHC->LHC, you can either inverse the sign on the z component of the world matrices, or you can inverse the sign of the vertex data. Alternatively, you can leave the signs as they are and use the RH variants of the view/projection in directx.

I would recommend you inverse the z scale of the world matrix for static meshes etc and for skinned meshes just inverse the z scale of the root joint.

Chapter 3: How to use the COLLADA DOM

Before we start, I assume you would have already compiled the DOM and linked it etc. I'm basically going to go through how to use the DOM for everything that we will need to get. The DOM is very easy to understand and use, you will quickly understand it, really this is the most important chapter because once you understand each topic here, **you will know everything you need to know for loading anything you want from any COLLADA file**. Also note that I do not bother to use the classes that extend the basic daeElement, there isn't really any use to them that the regular daeElement can't do.
I recommend you have a simple COLLADA file to follow along with just so you can see the data types I'm talking about.

How to open a .dae file

First off just include everything to do with the DOM so:

```
#include <dae.h>
#include <dom.h>
#include <dom/domCOLLADA.h>
#pragma comment(lib, "libcollada14dom21-d.lib")
#pragma comment(lib, "libxml2_a.lib")
#pragma comment(lib, "zlib.lib")
#pragma comment(lib, "wsock32.lib")
#pragma comment(lib, "pcre-d.lib")
#pragma comment(lib, "pcrecpp-d.lib")
#pragma comment(lib, "minizip-d.lib")
#pragma comment(lib, "libboost_filesystem.lib")
#pragma comment(lib, "libboost_system.lib")
```

To load a .dae is very simple, follow along the comments:

```
DAE dae;
daeElement* root = NULL;

int main(int argc, char** argv)
{
	if(argc < 2)
	{
		cout << "No input parameter!\n";
		return 1;
	}

	//Convert commandline to string
	string filename(argv[1]);

	//Load .dae to a root node element
	root = dae.open(filename);

	if(!root)
	{
		cout << "Document import failed.\n";
		return 1;
```

```
	}

	//You'll notice that I'm not dereferencing any pointers etc. the
	//DOM self manages memory so you don't have to worry about any of
	//that

	return 0;
}
```

That's it to load a .dae file into C++ and get its root node (the COLLADA element at the start of the .dae file). Very simple! **If you get any linker errors or etc try putting "msvcrt" in the ignored libraries in your project properties.**

How to get any element in the file

Assuming you have some piece of code like above, that you have a root node element, you can get any element in the .dae file from the root node by just getting a descendant by name.

```
daeElement* library_visual_scenes = NULL;
library_visual_scenes = root->getDescendant("library_visual_scenes");

if(!library_visual_scenes)
{
	cout << "Scene not found!\n";
	return 1;
}
```

If there is more then one of the specific type of node you want then it'll return the first it finds. In those cases you want to do things a bit differently, check the next point for that...

How to handle children and descendants

Let's say you have a daeElement that is one of the <mesh> nodes from a file. Typically a <mesh> has several <sources>, one <vertices> and a <triangles> element. If you want the children (**note what I said in the previous chapter the difference between child and descendant**) then you can do as follows:

```
//Get the children of the <mesh> element
daeTArray<daeElementRef> children = mesh->getChildren();
```

If you want an array of descendants though you're out of luck, you shouldn't ever need such an array though, with careful use of children you're better off. Also if you have multiple elements of the same type you're better off with getChildren(). Don't be scared by the daeElementRef either, it's basically the same thing as a daeElement*.

How to get the attributes of an element

Now let's say you have a <source> element from the mesh as before and you want to know what type of source it is by reading its attribute. You can get the attribute of any daeElement by doing like so:

```
//Get the attribute of an element
string id = source->getAttribute("id").data();
```

How to get the data in an element

Okay now suppose that you have a <float_array> element and you want to get the data between its tags, you can do so simply through the use of stringstreams:

```
//Get the float_array element
daeElement* floatArray = source->getChild("float_array");
int count = atoi(floatArray->getAttribute("count").data());

//This gets all the data for this tag in raw form
string rawData = floatArray->getCharData();

//Set up a stream for that raw data
stringstream stm(rawData);

//The Array we'll store the floats in
vector<float> floats;

//Read each float
for(unsigned int z = 0; z < count; z++)
{
	float t = 0;

	stm >> t;

	floats.push_back(t);
}
```

There are other ways to do this obviously, but stringstreams work well and are very simple as you can see.

How to get an element by URI reference

At the beginning of this chapter I said I don't bother with the classes that extend the daeElement, but for URI reference you must and is the only place where I do. If in your .dae you have something like:

```
<instance_controller url="#CubeShape-skin">
```

There should be a <controller> somewhere else in the file, it'll be under <library_controllers> but if you want the element it's referring to in that <instance_controller>, assuming you have some daeElement* that this node is under, then you can do like so:

```
daeElement* controller = NULL;
domInstance_controller* instance_controller = NULL;

//Get the <instance_controller> that's under whatever node you have
instance_controller =
   (domInstance_controller*)node-
>getDescendant("instance_controller");

//If the instance_controller exists
if(instance_controller)
{
     //This will get the actual element that the
          //instance_controller is referring to
     controller = cont->getUrl().getElement();
}
```

How to get an element by ID reference

Say you have an element with an attribute named source that has something like #something as its data, it's referencing some other element in the file and you can find it as follows:

```
//Get the id
string source = node->getAttribute("source").data();

//Erase the "#" at the start
source.erase(0, 1);

//Get the element from the document
daeElement* sourceNode =
           dae.getDatabase()->idLookup(gID, node->getDocument());
```

That pretty much does it for the DOM, you should by now know everything you need to know to load any element in a COLLADA file and get its data, attributes and children. You can see how simple the DOM really is and it allows you to very simply access everything. The next few chapters are pretty much a breeze as everything in COLLADA is stored logically.

Chapter 4: Importing the data needed for static geometry

What is needed to render static geometry?

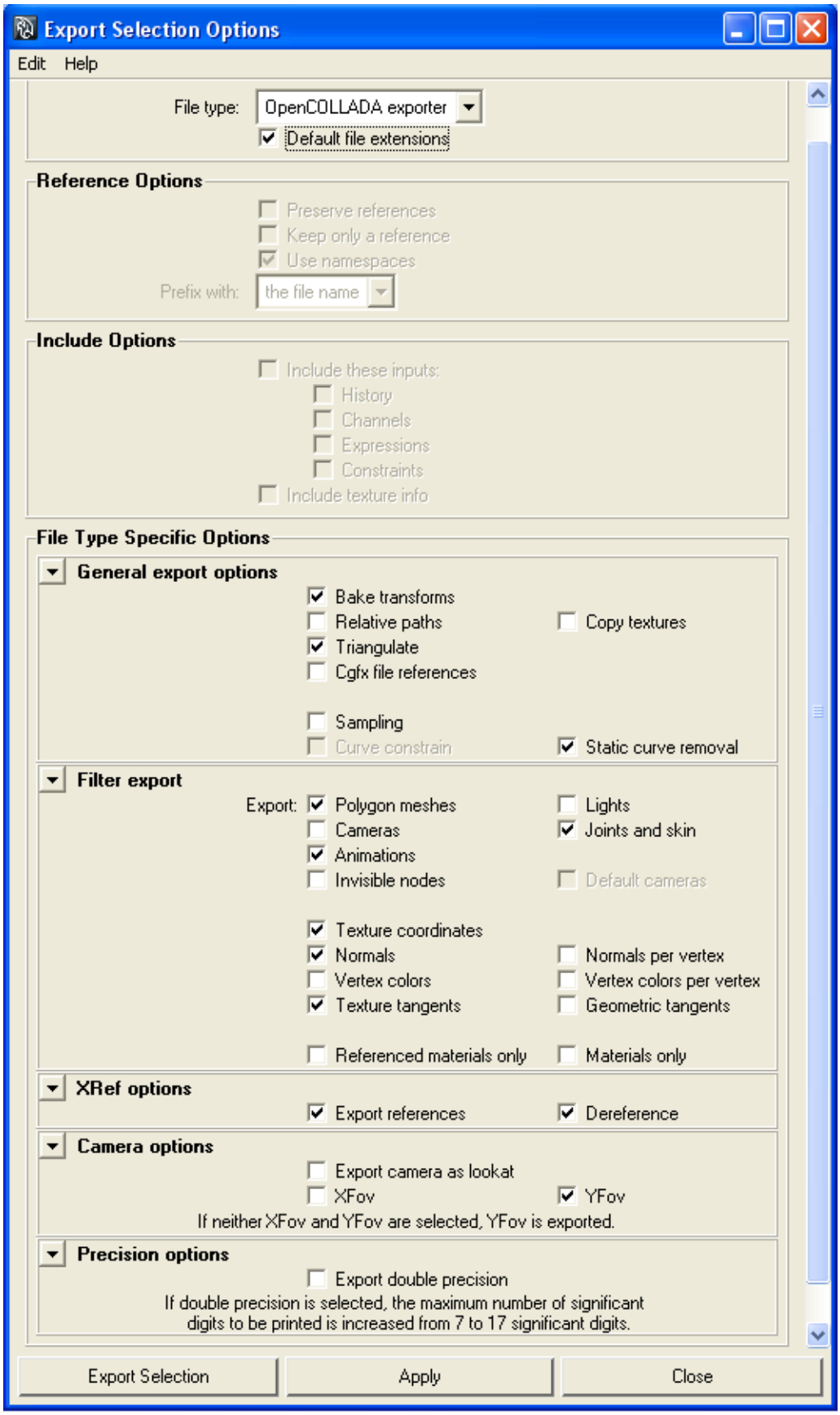
This is an important question to ask before going to import any data, what data do you need for you to render static geometry? Assuming you're trying to get something working for any modern game, you need:

- Position for each vertex
- Normal for each vertex
- UV for each vertex
- Tangent for each vertex
- Bi-Tangent for each vertex
- 16 or 32 bit Integer index array
- World matrix

The COLLADA exporters available right now will all produce results for un-indexed meshes, don't fret though cause you can download a tool "COLLADA Refinery" that will make an indexed mesh out of any .dae file, **read Appendix(ii) for more information**.

Okay so you know what data you need to render a mesh now, where would you find this data in a COLLADA file? In what format will the data be in?

Before I answer this question one simple note, this chapter will detail for you, very specifically how to load any static polygonal meshes exported with the following options in the "OpenCOLLADA2010" which I've only tested on Maya2010, however from Chapter 3 and a little bit of time looking at COLLADA files yourself you should understand how to get this data.
These are the options I use when I export static meshes:


Export Selection Options
Edit
Help
File type:
OpenCOLLADA exporter
Default file extensions
Reference Options
Preserve references
Keep only a reference
Use namespaces
Prefix with:
the file name
Include Options
Include these inputs:
History
Channels
Expressions
Constraints
Include texture info
File Type Specific Options
General export options
Bake transforms
Relative paths
Copy textures
Triangulate
Cgfx file references
Sampling
Curve constrain
Static curve removal
Filter export
Export:
Polygon meshes
Lights
Cameras
Joints and skin
Animations
Invisible nodes
Default cameras
Texture coordinates
Normals
Normals per vertex
Vertex colors
Vertex colors per vertex
Texture tangents
Geometric tangents
Referenced materials only
Materials only
XRef options
Export references
Dereference
Camera options
Export camera as lookat
XFov
YFov
If neither XFov and YFov are selected, YFov is exported.
Precision options
Export double precision
If double precision is selected, the maximum number of significant
digits to be printed is increased from 7 to 17 significant digits.
Export Selection
Apply
Close

Okay! We know what data we need, we are sure that every static mesh will be exported with the same options, you can write code to verify the options but really it's a waste of time, we will assume for the sake of this chapter that the mesh we'll be loading is an "average case scenario". You can download the .dae file for this chapter, there should be a link somewhere on the site.

So let me break it down right here, importing a COLLADA document starts with looking through the `<library_visual_scenes>` element. This element will have a child `<visual_scene id="VisualSceneNode" name="Scene">` element, this will contain a high level overview of the objects that make up the scene. If you have just a list of static meshes then it will just have a list of `<node id="mesh" name="mesh">` elements.

Each node element will be a container for the geometry and the world matrix for that geometry. Think of those node elements as like a "Mesh" class in rendering, it will contain a reference to the Vertex Buffer and Index Buffer as well as the World matrix for rendering them the right way.

- The World matrix for the mesh is found as the `<matrix sid="transform">` element under the node.

- You can find the Vertex and Index Buffer information as a reference to another element in the file, **note that this is only for static geometry**, it will be a `<instance_geometry url="#meshShape">` element under the node, it will be referencing a `<geometry id="meshShape" name="meshShape">` element found under the library_geometries. Through the technique you read about in chapter 3 you will be able to easily find the element it's referring to.

Knowing what to expect for a node in the visual_scenes, now we turn our attention to the actual vertex information. All the vertex information of a COLLADA file, for a static mesh, is found under library_geometries. Each mesh is referenced by a node in the visual_scene by an instance_geometry, that element will point you to it's geometry element. This geometry element will have a `<mesh>` tag, the reason it doesn't outright have the children of `<mesh>` is to compensate for the `<extra>` element that is optionally there as well as different types of geometry you may encounter, that's topic uncovered here though.

A `<mesh>` element will be as follows:

- A bunch of `<source>` elements, these will be the individual sources that make up the mesh, for example if you have a mesh that is defined by positions and UV's, then you will have two sources, a position source and a UV source. It will contain the data referenced by the index buffer, so basically this data will be used for the vertex buffer.
- A `<vertices>` element, you can pretty much ignore this

- A <triangles> element, this will have a few <input> elements, detailing how to read the index array that is the element <p>, the information there will be used as the index buffer basically.

Now that you know the World matrix is stored with the node that references the Vertex and Index Buffer, as well as knowing how the index and vertex buffer is stored, though it may be a bit confusing still, I will show you the way to read all the information you need for a list of static polygonal meshes into memory.

A detailed walkthrough of the code for loading static geometry

In this part of the chapter, I'll walk you through the source code provided for this chapter; I recommend you read along in the project file supplied. The application that is provided is a command line COLLADA loader, you run the program with a filename for the argument and it will load all the static meshes in the file.

The meat of the application is basically in the COLLADALoader class. I've made it as simple as possible to understand, so you can probably find ways to improve it's speed.

First I'll give you an overview of the class:

```
class COLLADALoader
{
private:
	//Dae file
	DAE dae;

	//Root node
	daeElement* root;

	//<library_visual_scenes> node
	daeElement* library_visual_scenes;

	//<library_geometries> node
	daeElement* library_geometries;

public:
	//Constructor
	COLLADALoader()
	{
		root = NULL;
		library_visual_scenes = NULL;
		library_geometries = NULL;
	}

	//Load all the meshes from a file
	vector<Mesh*> Load(string filename);
```

```
private:
	//Process a <library_visual_scenes> node
	void processVisualScenes(vector<Mesh*> &meshes);

	//Process a <geometry> node for each mesh
	void processGeometries(vector<Mesh*> &meshes);

	//Process a <source> node
	void processSource(Mesh* mesh, daeElement* source);

	//Process a <triangles> node
	void processTriangles(Mesh* mesh, daeElement* triangles);

	//Read a <matrix> node, better to have this as a generalized
	//function, will read into OpenGL style, conversion to DirectX
	//later...
	D3DXMATRIX processMatrix(daeElement* node);
};
```

You will notice that the more important nodes of a .dae file each have a corresponding function to them in this class, but more importantly that I have generalized matrix reading to one function. I **highly** recommend you generalize matrix reading and that you don't convert any data until you have loaded everything. This is just a personal preference though, I prefer conversion functions outside of importing, it's slower but allows you flexibility in conversion and easier to read code.

You will also notice that while the daeElement objects are pointers, the root DAE object is not. Let the DAE object manage itself, theres no reason to make it a pointer really as you won't be passing it and it's fairly effective at managing itself.

The "**`vector<Mesh*> Load(string filename);`**" function is what you'll use to load everything(duh..) and it will return an array of Mesh*. The Mesh class will hold the nodes that refer to the data in the COLLADA file as well as it's loaded data in the program.

```
class Mesh
{
private:
	//<geometry> element to get mesh data from
	daeElement* geometry;

	//Component Vertex Data, to be compiled into Vertices later...
	vector<D3DXVECTOR3> Positions;
	vector<D3DXVECTOR2> UVs;
	vector<D3DXVECTOR3> Normals;
	vector<D3DXVECTOR3> Tangents;
	vector<D3DXVECTOR3> BiTangents;

	//Set it so COLLADALoader can access privates
	friend class COLLADALoader;
```

```
	//Combine the component vertex data to Vertices array
	void combineComponents();

public:
	//Name of this mesh
	string Name;

	//World transform
	D3DXMATRIX World;

	//Combined Vertex Data, ready for Vertex Buffer
	vector<Vertex> Vertices;

	//Index data, ready for Index Buffer
	vector<unsigned int> Indices;

	//Constructor
	Mesh(string Name, D3DXMATRIX World)
	{
		//Set Name
		this->Name = Name;

		//Set World
		this->World = World;

		//Initialize Component Vertex Data arrays
		Positions = vector<D3DXVECTOR3>();
		UVs = vector<D3DXVECTOR2>();
		Normals = vector<D3DXVECTOR3>();
		Tangents = vector<D3DXVECTOR3>();
		BiTangents = vector<D3DXVECTOR3>();

		//Initialize Combined Vertex Data array
		Vertices = vector<Vertex>();

		//Initialize Index Data Array
		Indices = vector<unsigned int>();

		//Initialize COLLADA pointers to NULL
		geometry = NULL;
	}
};
```

All the DirectX friendly data is public and if you're only interested in using this class and not writing your own, then just look to that public section for all the data you need to render a static mesh.
The Vertex class is just a simple holder:

```
class Vertex
{
public:
	//Components of a Vertex
	D3DXVECTOR3 Position;
	D3DXVECTOR2 UV;
	D3DXVECTOR3 Normal;
```

```
	D3DXVECTOR3 Tangent;
	D3DXVECTOR3 BiTangent;

	//Constructor
	Vertex(D3DXVECTOR3 Position, D3DXVECTOR2 UV, D3DXVECTOR3 Normal,
			D3DXVECTOR3 Tangent, D3DXVECTOR3 BiTangent)
	{
		this->Position = Position;
		this->UV = UV;
		this->Normal = Normal;
		this->Tangent = Tangent;
		this->BiTangent = BiTangent;
	}
};
```

Now that you know how we are gonna store the data from the COLLADA file, its just a matter of getting the data.

So let's see what the Load function looks like...

```
//Load all the meshes from a file
vector<Mesh*> Load(string filename)
{
	//Output array
	vector<Mesh*> meshes = vector<Mesh*>();

	//Open the file and get the root node
	root = dae.open(filename);

	//Check if import succeded
	if(!root)
	{
		cout << "Document import failed. \n";
		return meshes;
	}

	//Get the library_visual_scenes
	library_visual_scenes =
			root->getDescendant("library_visual_scenes");

	//Check if there is a <library_visual_scenes>
	if(!library_visual_scenes)
	{
		cout << "<library_visual_scenes> not found.\n";
		return meshes;
	}

	//Get the library_geometries
	library_geometries = root->getDescendant("library_geometries");

	//Check if there is a <library_geometries>
	if(!library_geometries)
	{
		cout << "<library_geometries> not found.\n";
		return meshes;
```

```
	}

	//Process <library_visual_scenes>
	processVisualScenes(meshes);

	//Process the <geometry> node for each mesh
	processGeometries(meshes);

	//Compile vertex components into one buffer
	for(unsigned int i = 0; i < meshes.size(); i++)
			meshes[i]->combineComponents();

	//Close the .dae file
	dae.close(filename);

	//Return list of meshes
	return meshes;
}
```

That's quite a bit to take in at first but let me give you a simplistic overview:

1. Open the file, the open function will also return the root node of the entire file, that is the `<COLLADA>` node at the beginning.
2. Using this root node, get the C++ representation of the `<library_visual_scenes>` and the `<library_geometries>` node.
3. Process every child node of `<library_visual_scenes>` to get each static mesh in the scene and its corresponding `<geometry>` node, this is done in the function processVisualScenes().
4. Process each mesh's `<geometry>` node to find it's component data and index buffer, this is done in the function processGeometries()
5. Combine the component data for each mesh into a DirectX friendly data format.
6. Close the file and return the array of static meshes.

Like I said before, we start by looking through the `<library_visual_scenes>` of the file, we will check each node if it fits the criteria for a static mesh, if it does we'll get the World matrix and a reference to the `<geometry>` node for that mesh. The criteria is just that the node not be of type "JOINT", has an `<instance_geometry>` node and that the node returns a valid geometry node pointer.

Let's look at the processVisualScenes() function:

```
//Process a <library_visual_scenes> node
void processVisualScenes(vector<Mesh*> &meshes)
{
	//Get the <visual_scene> node
	daeElement* visual_scene =
		library_visual_scenes->getDescendant("visual_scene");
```

```
	//Get all the <node>'s for the <visual_scene>
	daeTArray<daeElementRef> nodes = visual_scene->getChildren();

	//For each <node>...
	for(unsigned int i = 0; i < nodes.getCount(); i++)
	{
		//Get the ID, the SID, the name and the type, if they exist
		string Name = nodes[i]->getAttribute("name").data();
		string Type = nodes[i]->getAttribute("type").data();

		//Skip JOINT node's, only meshes
		if(Type == "JOINT") continue;

		//Get the <instance_geometry> node that corresponds to this
		//<node>
		domInstance_geometry* instance_geometry = NULL;
		instance_geometry =
(domInstance_geometry*)nodes[i]->getDescendant("instance_geometry");

		//If there is no <instance_geometry>, this isn't a static
			//mesh and we will skip it.
		if(!instance_geometry) continue;

		//Get the <geometry> node that is referenced by the
			//<instance_geometry>
		daeElement* geometry =
			instance_geometry->getUrl().getElement();

		//If the referenced node was not found, skip this node
		if(!geometry) continue;

		//Now create a new mesh, set it's <geometry> node and get
			//it's World transform.
		meshes.push_back(
new Mesh(Name, processMatrix(nodes[i]->getDescendant("matrix"))));
		meshes.back()->geometry = geometry;
	}
}
```

The code looks very messy in this document but read along in the project file and it will seem much clearer.
Now let me give a simplistic overview of this function:

1. Get the `<visual_scene>` node from the library.
2. Get all the children of the `<visual_scene>`, these will all be `<node>` elements.
3. For each `<node>`:
    1. Get it's Name and Type
    2. Skip any node of Type "JOINT"
    3. Get the `<instance_geometry>` child and the `<geometry>` node it points to that is located in the `<library_geometries>` elsewhere in the file, use the getUrl() function to find it for you.

4. Add a Mesh to the array, give it the name of the node and read the <matrix> child for this node to get the World matrix

That about sums up the processing of the <library_visual_scenes>, so far so good I'm hoping on your side of things.

So at this point in the code, you have an array of Mesh*, each has a Name, a World matrix and a reference to find the geometry data. Now to get the component vertex data and the index buffer.

We get this data by processing the <geometry> node that is stored in each Mesh*, this is done in the processGeometries() function, which will subsequently also process some <source> nodes and a <triangles> node.

```
//Process a <geometry> node for each mesh
void processGeometries(vector<Mesh*> &meshes)
{
	//Foreach mesh...
	for(unsigned int i = 0; i < meshes.size(); i++)
	{
		//Get the <mesh> node
		daeElement* mesh =
			meshes[i]->geometry->getDescendant("mesh");

		//Get the <source> nodes
		daeTArray<daeElementRef> sources = mesh->getChildren();

		//Get the <triangles> node (yes it will be in the sources
			//array above if you wish to find it that way)
		daeElement* triangles = mesh->getDescendant("triangles");

		//Process each <source> child
		for(unsigned int z = 0; z < sources.getCount(); z++)
				processSource(meshes[i], sources[z]);

		//Process the <triangles> child
		processTriangles(meshes[i], triangles);
	}
}
```

So this function simply finds the <mesh> child of the <geometry> node for the mesh, then processes each of its <source> nodes and finally its <triangles> node. The <source> nodes will give the Positions, UV's, Normal's, Tangent's and BiTangent's, the <triangles> will give the index buffer.

The processSource() function has a lot of redundancy so rather then show the entire function in this document I'll show the general form by showing how to get the Position's component.

```
//Get Positions
if(source->getAttribute("name").find("position") != string::npos)
{
	//Get the <float_array> node
	daeElement* float_array = source->getChild("float_array");

	//Get the number of raw float's contained
	unsigned int count =
			atoi(float_array->getAttribute("count").data());

	//Get the raw string representation
	string positionArray = float_array->getCharData();

	//Set up a stringstream to read from the raw array
	stringstream stm(positionArray);

	//Read each float, in groups of three
	for(unsigned int i = 0; i < (count / 3); i++)
	{
		float x, y, z;

		stm >> x;
		stm >> y;
		stm >> z;

		//Push this back as another Position component
		mesh->Positions.push_back(D3DXVECTOR3(x, y, z));
	}

	return;
}
```

So for every other source it's pretty much the same, for UV's it would obviously read in groups of two instead. A simplistic overview of the general form of the processSource() function is as follows:

1. What type of source is it? This depends on which type of component name is found in the total name attribute of the `<source>`, in this case it's a positions source as it found "position" somewhere in the node's name attribute.
2. Get the `<float_array>` child of this source, this node essentially holds the raw data.
3. Read the "count" attribute of the `<float_array>` node to determine how many float's are in the `<float_array>`.
4. Get a string representation of the `<float_array>` node's data.
5. Set up a stringstream with that string representation, this will allow you to read each float individually easily.
6. Read x, y and z of each position and add it to the Positions array for the mesh.

Get used to the stringstream and using it as it's very handy. I'm sure there are a ton of other ways to do all this but I found this works easily, both to understand and program at the same time.

So this function will basically operate on the 5 `<source>` nodes you should receive. From here it's a matter of getting the index buffer, note once again that the COLLADA exporters don't export indexed primitives natively so you have to use a conditioner to make it so before loading in this application.

The processTriangles() function is pretty much just the processSource() function only for unsigned integers, I'm assuming your index buffer will be 32bit but if you want 16 then just make it unsigned shorts, it should work pretty much the same.

```
//Process a <triangles> node
void processTriangles(Mesh* mesh, daeElement* triangles)
{
	//Get the <p> node
	daeElement* p = triangles->getDescendant("p");

	//Get the number of faces, multiply by 3 to get number of indices
	unsigned int count =
		atoi(triangles->getAttribute("count").data()) * 3;

	//Get the raw string representation
	string pArray = p->getCharData();

	//Set up a stringstream to read from the raw array
	stringstream stm(pArray);

	//Read each unsigned int
	for(unsigned int i = 0; i < count; i++)
	{
		unsigned int p = 0;

		stm >> p;

		mesh->Indices.push_back(p);
	}
}
```

You can see it really is pretty much the same as processSource() so it should be easy to understand.

At this point now we just combine the vertex components into one singular DirectX friendly array, for each mesh you call it's "`combineComponents()`" function which is as follows, minus the lousy formatting look in the pdf(damn you character limit!):

```
//Combine the component vertex data to Vertices array
void combineComponents()
{
      for(unsigned int i = 0; i < Positions.size(); i++)
      {
            Vertices.push_back(Vertex(Positions[i],
                                      UVs[i],
                                    Normals[i],
                                   Tangents[i],
                                 BiTangents[i]));
      }
}
```

So at this point, the .dae file is closed and you will reach in the main() function, the following comment:

```
//---------------------------------------------------------------------
-
//At this point, all static meshes will have been loaded from the
//collada document and stored in "vector<Mesh*> meshes"
//---------------------------------------------------------------------
-
```

That's all there is to loading all the static meshes from a COLLADA document. Easy right?

Chapter 5: Importing the data needed for skeletal animation

This chapter follows sequentially from the code of the previous chapter so if you skipped ahead, I thoroughly recommend you go back and read it.

What is needed for skeletal animation?

Just like in the previous chapter, the best thing to do first is to clearly identify the data we want for skeletal animation. Take everything needed for a static mesh and add the following to the list:

- Bind Pose matrix for each joint
- Inverse Bind Pose matrix for each joint
- Bind Shape matrix
- Bone Indices for each vertex
- Bone Weights for each vertex
- A matrix for a joint for each key frame of animation as well as the time at which the key frame is set.

These may seem very easy to get, for the most part they are, but the process is a lot more convoluted than with static meshes.

So we know what data we want, where would this data be in a .dae file?

- Bind Pose will be in the `<matrix>` child node for the joint's `<node>` in the `<visual_scenes>`
- Inverse Bind Pose is given as a `<source>` under the `<skin>` node that you will find for a mesh's `<controller>`
- Bind Shape matrix is given under the `<skin>` as well, it's under the node `<bind_shape_matrix>`
- Bone Weights are given as an out of order list of weights that will be put in order when reading the Bone Indices, under a `<v>` tag in `<vertex_weights>` for the `<skin>` node, similar to reading a static mesh index buffer, so that's both of them out of the way.
- Time and Matrix for each key frame for a specific joint is given as an `<animation>` node under `<library_animations>`, the id attribute of the `<animation>` node should contain the joint's name and "matrix".

The skinning equation for COLLADA is as follows:

$$\text{Position} = \text{VertexPosition} \times \left(\text{Bind Shape} \times \Sigma(\text{MatrixPallette}[n] \times \text{Weight}[n])\right) \quad \text{where } 0 \le n < 4$$

MatrixPallette[n] = inverseBindPose[n] x worldTransforms[n]

worldTransforms[n] = BindPose[n] x worldTransforms[parent]

That looks more complicated than it really is, in the appendix when I show how to render everything in detail it will become far clearer, if you already understand skinning then it shouldn't be anything new really.

One major difference in the reading of a skinned mesh in comparison to a static mesh in COLLADA is that when you find the mesh's node representation in the <visual_scenes> it will not have an <instance_geometry> child, it will instead have an <instance_controller> child and finding the geometry for it requires a bit of extra work. But it's an important note because it's an easy mistake to make.

Another note that I think deserves a mention, you may be thinking that if you have the bind pose for a joint, then you can get the inverse bind pose by just inversing it using a D3DXMatrixInverse() function etc, I have found that this is not the case unfortunately, though I haven't tested this too thoroughly. You shouldn't ever really need to make your own inverse bind pose anyway but I thought it was still an important note.

Everything is exported using the same options as the previous chapter.

Now that we know what data we need, where the data is and a general form of the data, lets get to programming!

A detailed walkthrough of the code for loading skeletal animation

As with the previous chapter I will now thoroughly walk you through the code provided for this chapter, so again I advise that you read along with the source code provided in visual studio. The program for this chapter is pretty much the same as the previous only now it will load every skinned mesh and the animation for the mesh.
Anyways, as before our journey begins at an overview of the new COLLADALoader class.

```
class COLLADALoader
{
private:
	//Dae file
	DAE dae;

	//Root node
	daeElement* root;

	//<library_visual_scenes> node
	daeElement* library_visual_scenes;

	//<library_animations> node
	daeElement* library_animations;

public:
```

```
        //Constructor
        COLLADALoader()


        //Load all the meshes from a file
        MeshManager* Load(string filename)
private:
        //Process a <library_visual_scenes> node
        void processVisualScenes(MeshManager* Meshes)

        //Process each Skinned Mesh's <controller> and build a skeleton
        //for them
        void processSkinnedMeshes(MeshManager* Meshes)

        //Build a Skeleton from a SkinnedMesh's rootJoint <node>
        void buildSkeleton(SkinnedMesh* Mesh)

        //Process a Joint <node> and it's children recursively
        void processJoint(SkinnedMesh* Mesh, daeElement* joint)

        //Process a <controller> node
        void processController(SkinnedMesh* Mesh)

        //Process an element that contains raw inverse bind pose data
        void processBindPoseArray(vector<Joint> &Joints, daeElement*

        source)

        //Process an element that contains raw weights data
        vector<float> processWeightsArray(daeElement* source)

        //Process a <library_animations> node
        void processAnimations(MeshManager* Meshes)

        //Process an <animation> node
        void processAnimation(Joint* joint, int jointIndex, daeElement*

        animation)

        //Process a <geometry> node for each Static Mesh
        void processGeometries(MeshManager* Meshes)

        //Process a <source> node for Static Mesh
        void processSource(Mesh* mesh, daeElement* source)

        //Process a <source> node for Skinned Mesh
        void processSource(SkinnedMesh* mesh, daeElement* source)

        //Process a <triangles> node for Static Mesh
        void processTriangles(Mesh* mesh, daeElement* triangles)

        //Process a <triangles> node for Skinned Mesh
        void processTriangles(SkinnedMesh* mesh, daeElement* triangles)
```

```
	//Read a <matrix> node, better to have this as a generalized
	//function, will read into OpenGL style, conversion to DirectX
	//later...
	D3DXMATRIX processMatrix(daeElement* matrix)
};
```

First thing you'll notice is that now the Load() function returns a MeshManager* as opposed to a vector<Mesh*>. You will also notice a SkinnedMesh class being referenced throughout.
One of the requirements for this document was to be profficient in C++, however I understand that this document will most likely be read more by beginners to average in programming as well. With that in mind I made a very beginner friendly choice which is to **not extend the Mesh class with SkinnedMesh**, meaning they are **two distinct objects** and SkinnedMesh does not derive from Mesh and can't be converted to Mesh (unless you write a custom conversion operator).
I found that extending classes in code can be confusing to some and this document isn't about learning C++ it's about COLLADA so I opted for this path, thus you will see some redundancy in the code at the cost of readability. I think the trade off is fine but I recommend that in your implementation you just extend the Mesh class.

With the overview of the class in check, let's take a look at what the Load() function returns, MeshManager.

```
class MeshManager
{
public:
	//Array of static meshes
	vector<Mesh*> StaticMeshes;

	//Array of skinned meshes
	vector<SkinnedMesh*> SkinnedMeshes;

	//Constructor
	MeshManager()
	//Destructor
	~MeshManager()
};
```

MeshManager just holds two std::vector<>'s which contain all the types of meshes that can be loaded.

We have already seen the Mesh class, let's take a look at the SkinnedMesh class, when I said that things get a bit convoluted this chapter you'll start to see what I mean in looking at this class.

```
class SkinnedMesh
{
```

```
private:
	//<controller>
	daeElement* controller;

	//<geometry>
	daeElement* geometry;
	//Root <node> for skeleton
	daeElement* rootJoint;

	//Component Vertex Data, to be compiled into Vertices later...
	vector<D3DXVECTOR3> Positions;
	vector<D3DXVECTOR2> UVs;
	vector<D3DXVECTOR3> Normals;
	vector<D3DXVECTOR3> Tangents;
	vector<D3DXVECTOR3> BiTangents;
	vector<Index> boneIndices;
	vector<Weight> Weights;

	//Set it so COLLADALoader can access privates
	friend class COLLADALoader;

	//Combine the component vertex data to Vertices array
	void combineComponents()
	//Make Animation list
	void combineJointAnimations()
public:
	//Name
	string Name;

	//Root Transform Matrix
	D3DXMATRIX RootTransform;

	//BindShape Matrix
	D3DXMATRIX BindShape;

	//Joints
	vector<Joint> Joints;

	//Time sorted Animation Key list
	vector<AnimationKey> Animations;

	//Combined Vertex Data, ready for Vertex Buffer
	vector<SkinnedVertex> Vertices;

	//Index data, ready for Index Buffer
	vector<unsigned int> Indices;

	//Constructor
	SkinnedMesh(string Name, D3DXMATRIX RootTransform)
};
```

You will notice five new classes at play here:

1. SkinnedVertex: Same as Vertex as used by Mesh only with two added components per vertex, Indices and Weights.

```
class SkinnedVertex
{
public:
	//Components of a Vertex
	D3DXVECTOR3 Position;
	D3DXVECTOR2 UV;
	D3DXVECTOR3 Normal;
	D3DXVECTOR3 Tangent;
	D3DXVECTOR3 BiTangent;
	Index Indices;
	Weight Weights;

	//Constructor
	SkinnedVertex(D3DXVECTOR3 Position,
		      D3DXVECTOR2 UV,
		      D3DXVECTOR3 Normal,
		      D3DXVECTOR3 Tangent,
		      D3DXVECTOR3 BiTangent,
		      Index       Indices,
		      Weight      Weights)
	{
		this->Position = Position;
		this->UV = UV;
		this->Normal = Normal;
		this->Tangent = Tangent;
		this->BiTangent = BiTangent;
		this->Indices = Indices;
		this->Weights = Weights;
	}
};
```

2. Index: Just a class to hold a fixed short array. The reason for using short's instead of int's is because you will never run into a mesh that has over 60000 joints. Not in the next ten years at least. I doubt even the next twenty.

```
class Index
{
public:
	//Indices
	short Indices[4];

	//Constructor
	Index()
	{
		for(int i = 0; i < 4; i++)
		{
			Indices[i] = 0;
		}
	}
};
```

3. Weight: Same as above only with float instead of short.

4. AnimationKey: Each AnimationKey will hold a Matrix, a Bone it references and a Time.

```
class AnimationKey
{
public:
	//Time this key is set
	float Time;

	//Matrix for this key
	D3DXMATRIX Matrix;

	//Bone this key affects
	int Bone;

	//Base Constructor
	AnimationKey()
	//Detailed Constructor
	AnimationKey(float Time, D3DXMATRIX Matrix, int Bone)
};
```

5. Joint: This will hold all the information for each Joint. Basically a Name, a BindPose, an InverseBindPose, an index to it's parent in the array it's kept and a pointer reference to it's parent.
You will notice that the joints for a skinned mesh are being held in an array rather than a linked list in the SkinnedMesh class. I have found that an array style is easier to iterate through the entire skeleton for animation as well as easier to set on the GPU for rendering, a linked list type would be beneficial in terms of structure though, it all depends on your style so it's not a big issue really.

```
class Joint
{
private:
	//SID
	string SID;

	//Node for this in the COLLADA file
	daeElement* Node;

	//Animation Keys for this Joint
	vector<AnimationKey> Animations;
	//COLLADALoader access rights
	friend class COLLADALoader;

	//SkinnedMesh access rights
	friend class SkinnedMesh;

public:
	//Name
	string Name;
```

```
	//Bind_Matrix
	D3DXMATRIX bind_matrix;

	//Inv_Bind_Matrix
	D3DXMATRIX inv_bind_matrix;

	//Parent Index
	int parentIndex;

	//Parent Joint*
	Joint* parentJoint;

	//Base Constructor
	Joint()
	//Detailed Constructor
	Joint(string Name, string SID, daeElement* Node)
};
```

The reason each joint has its own std::vector<AnimationKey> is because when reading animation from COLLADA it is always given as the animations for a specific joint each.
In the SkinnedMesh class you will notice a combineJointAnimations() function, what that will do is add all the Joints animations together in the SkinnedMesh to one big array and bubble sort them by least-most time, the reason for one big array has to do with the animation system you will implement in the Appendix. Once again things have been done with the beginner in mind so it may seem like a hassle now but it will simplify things greatly later, just remember that this code is for you to learn and has been crafted in that sense rather than in the sense of commercial ready and crazy fast.

Now that we have covered the background data structures and the overview of the new version of COLLADALoader, let's take look at the new Load() function:

```
//Load all the meshes from a file
MeshManager* Load(string filename)
{
	//Output object
	MeshManager* Meshes = new MeshManager();


	//Open the file and get the root node
	root = dae.open(filename);

	//Check if import succeded
	if(!root)
	{
		cout << "Document import failed. \n";
		return Meshes;
	}

	//Get the <library_visual_scenes>
	library_visual_scenes =
```

```
                  root->getDescendant("library_visual_scenes");

      //Check if there is a <library_visual_scenes>
      if(!library_visual_scenes)
      {
            cout << "<library_visual_scenes> not found.\n";
            return Meshes;
      }

      //Get the <library_animations>
      library_animations = root->getDescendant("library_animations");

      //Check if there is a <library_animations>
      if(!library_animations)
                        cout << "<library_animations> not found.\n";

      //Process <library_visual_scenes>
      processVisualScenes(Meshes);

      //processSkinnedMeshes
      processSkinnedMeshes(Meshes);

      //Process library_animations if it exists
      if(library_animations) processAnimations(Meshes);

      //Process the <geometry> node for each Static Mesh
      processGeometries(Meshes);

      //Compile vertex components into one buffer for each Skinned Mesh
      //and also compile Animation Keyframes list
      for(unsigned int i = 0; i < Meshes->SkinnedMeshes.size(); i++)
      {
            Meshes->SkinnedMeshes[i]->combineComponents();
            Meshes->SkinnedMeshes[i]->combineJointAnimations();
      }

      //Compile vertex components into one buffer for each Static Mesh
      for(unsigned int i = 0; i < Meshes->StaticMeshes.size(); i++)
                              Meshes->StaticMeshes[i]-
>combineComponents();

      //Close the .dae file
      dae.close(filename);

      //Return MeshManager
      return Meshes;
}
```

Okay nothing too complex just yet, last time I broke the function down meticulously due to it being so alien to a beginners eyes, but at this point I don't think you need a breakdown of this function as it's pretty much the same. Get the nodes of interest for each mesh and then operate on those nodes of interest to get the data, then transform the data to something a little more DirectX friendly, that's about the gist.

You'll notice that you don't need library_animations, sometimes you may just want to export a skinned mesh with no animation and that's okay so theres no need for the library to explicitly be there.

This time when processing the visual_scenes for skinned meshes you have to do a lot more searching and generally it's far more convoluted.

The processVisualScenes () function has now grown to a point where it is just far too convoluted for it to be just pasted here so I'll give an overview of it's procedure instead:

1. Get the <visual_scene> node
2. Get the children of the <visual_scene>
3. Foreach child <node> of <visual_scene>
    1. Get it's name and type attributes, if it's of type "JOINT" then skip it.
    2. Find out if the <node> has an <instance_controller> or not, if it doesn't then it's a static mesh and you can just use the standard <instance_geometry> method.
    3. The <node> has an <instance_controller> so now let's get the <controller> being referenced, if it doesn't exist in the file then skip this <node> altogether.
    4. From the <controller> node found before, get it's descendant <skin> element.
    5. Get the "source" attribute of <skin> and erase the "#" at the beginning, now search the .dae for the geometry being referenced. If no such geometry was found to match then skip this <node>.
    6. Now to find the root of the skeleton for this skinned mesh, try and stay with me on this cause it's a bit confusing. Get all the <skeleton> children of the <instance_controller> for this <node>. Now search all the <node>'s under <visual_scenes> and see if any of them (not counting their children) match any one of the <skeleton> children we are interested in. If so then that is the root joint's <node> for the skeleton. You will note that if a mesh has a **root joint that is a child of something else, then the <node> will be skipped**.
    7. Now that you have the elements of interest, push_back a new SkinnedMesh and set it's COLLADA references, also set the root joint for it.

You can see even in a simplistic overview form it's quite convoluted but at it's core it's still fairly simple, it used to be much easier back when the old COLLADA exporter would only put the root joint as the sole <skeleton> child for the <instance_controller>.

Alright at this point you now have a list of static and skinned meshes from the COLLADA file and references to their data. For a static mesh the path ahead is to just get the Vertex and Index buffer as per last chapter. A skinned mesh on the other hand requires a lot more work before it can get to that stage.

So let's get all the extraneous data for a SkinnedMesh then, let's take a look at processSkinnedMeshes()

```
//Process each Skinned Mesh's <controller> and also build a skeleton for
them
void processSkinnedMeshes(MeshManager* Meshes)
{
	//Foreach Skinned Mesh
	for(unsigned int i = 0; i < Meshes->SkinnedMeshes.size(); i++)
	{
		//Build it's skeleton
		buildSkeleton(Meshes->SkinnedMeshes[i]);

		//Process it's <controller>
		processController(Meshes->SkinnedMeshes[i]);
	}
}
```

Naturally this function will build a skeleton from the root joint's <node> and then process all the information desired from the <controller> node for a skinned mesh.

Let's take a look at how to build a skeleton, in this case a Joint array rather than a LinkedList, the reason as detailed previously. Building a skeleton in this case is basically two functions rather than one, the first part is just getting all the joints in the skeleton and the second part is setting the parent references etc.

Getting all the joints is done through a recursive function called at first with the root joint:

```
//Process a Joint <node> and it's children recursively
void processJoint(SkinnedMesh* Mesh, daeElement* joint)
{
	//Get the children of this joint
	daeTArray<daeElementRef> Children = joint->getChildren();

	//For each child
	for(unsigned int i = 0; i < Children.getCount(); i++)
	{
		//Get the Type of element this Child node is
		string elementName = Children[i]->getElementName();

		//If this element isn't of type node, skip it
		if(elementName.find("node") == string::npos) continue;

		//Get this Joint's Name and SID
		string Name = Children[i]->getAttribute("name").data();
```

```
            string SID = Children[i]->getAttribute("sid").data();

            //Add a Joint to this Mesh's Joint array
            Mesh->Joints.push_back(Joint(Name, SID, Children[i]));

            //Get the bind_matrix for this Joint
            Mesh->Joints.back().bind_matrix =
                        processMatrix(Children[i]-
>getDescendant("matrix"));

            //Process this Joint's children
            processJoint(Mesh, Children[i]);
      }
}
```

Simple enough recursive function, I separate the parenting part from this function due to my paranoia with reading order, there may be some .dae that is exported that has a joint referencing a parent that is yet to be read even though it's impossible I'll be prepared if it happens! I guess it's stupid on retrospection, I might as well guard my Y2K food stockpile with a shotgun while I'm at it.

Anyway, this function is called from the actual buildSkeleton() function, which the second part of takes care of the parent references for each joint.

```
//Build a Skeleton from a SkinnedMesh's rootJoint <node>
void buildSkeleton(SkinnedMesh* Mesh)
{
      processJoint(Mesh, Mesh->rootJoint);

      //Find each Joint's parent
      for(unsigned int i = 1; i < Mesh->Joints.size(); i++)
      {
            //Get the parent node's name in the COLLADA file
            string parentName =
      Mesh->Joints[i].Node->getParent()->getAttribute("name").data();

            //Check the array of Joints to see if any match the
                  //parentName
            for(unsigned int z = 0; z < Mesh->Joints.size(); z++)
            {
                  //Get this joint node's name in the COLLADA file
                  string jointName =
                  Mesh->Joints[z].Node->getAttribute("name").data();

                  //Compare names
                  if(parentName == jointName)
                  {
                        //Set the parentIndex to z
                        Mesh->Joints[i].parentIndex = z;

                        //Set the parentJoint reference to the address
                              //of the parentJoint
                        Mesh->Joints[i].parentJoint = &Mesh->Joints[z];
```

```
                        //Go to the next Joint in the list
                        break;
                    }
                }
            }
}
```

The parenting procedure is simple enough, find the joint in the joints array that has a name that matches the COLLADA representation's parent <node> name.

Now that we have a skeleton for this SkinnedMesh, it's time to get the influence that the skeleton has on each vertex as well as each joint in the skeleton's inverse bind pose.

This is all done in the processController() function. This function is comprised of many parts so here is an overview of the function:

1. Get the <skin> node
2. Get the <bind_shape_matrix> descendant of the <skin> and get it's matrix.
3. Get the <source> node child that contains the inverse bind poses, then get each inverse bind pose for each joint.
4. Get the <source> node child that contains the list of unordered weights, then get all it's data into a std::vector<float>
5. Get the <vertex_weights> node child and get its <vcount> and <v> children.
6. Each integer in <vcount> gives the number of influences for a vertex. Then for the number of influences get the bone and the weight influence of that bone on the vertex from <v>. Get the weight influence by using the integer after the bone specifier to index from the weights array.

You'll notice that unlike reading matrices like before with a simple generalized function, that doesn't work this time around due to the storage of the matrices being in a <float_array>. It's a good thing we're keeping conversion to the end right, otherwise any small change in conversion will require a lot of extra redundant work. There are definitely ways to still generalize matrix reading, but I found them to somewhat detract from the readability of the code so left them out in this educational iteration.
At this point you now have all the joint's and per position bone influences, so now we are onto processAnimations(), if they exist of course.

```
//Process a <library_animations> node
void processAnimations(MeshManager* Meshes)
{
            //Get each <animation>
            daeTArray<daeElementRef> animations = library_animations->getChildren();

            //Foreach <animation> node
            for(unsigned int i = 0; i < animations.getCount(); i++)
```

```
            {
                  //Get it's ID
                  string ID = animations[i]->getAttribute("id").data();

                  //Check if any of the SkinnedMeshes has any Joint that is affected
by this node
                  for(unsigned int z = 0; z < Meshes->SkinnedMeshes.size(); z++)
                  {
                        for(unsigned int x = 0; x < Meshes->SkinnedMeshes[z]-
>Joints.size(); x++)
                        {
                              //Check if you can find this joint's name in the ID and
if this                             //<animation> node is for a matrix
                              if((ID.find(Meshes->SkinnedMeshes[z]->Joints[x].Name)
                                    != string::npos)
                                    && (ID.find("matrix") != string::npos))
                              {
                                    //Process this <animation> node and store it in
this Joint's                              //Animation array
                                    processAnimation(&Meshes->SkinnedMeshes[z]-
>Joints[x],
                                                      x, animations[i]);
                              }
                        }
                  }
            }
}
```

Damn that looks so messy in .pdf form! Essentially what this function does is simple, it will iterate over every <animation> node and check if it's ID attribute contains the name of any joint in any of the SkinnedMeshes, if it does and the ID also contains "matrix" then we process the <animation> node. I guess this is a filtering function then really.

The processAnimation() function, will process an individual <animation> node. An <animation> node will be comprised of a <source> for input(times at which each of the keyframes occurs), a <source> for output(values for each keyframe, in this case a matrix) and a bunch of superflous information unless the interpolation type is something other then linear.

processAnimation works as follows:

1. Get the children of the <animation> node
2. Initialize two vectors, one to hold times and the other to hold D3DXMATRIX's.
3. Foreach <source> child
    - If it's ID attribute contains "input" then read all of it's float's as individual times.
    - If it's ID attribute contains "output" then read all of it's float's in groups of 16 as individual matrices.
4. Now just order the data into the selected joint's AnimationKey array.

At this point you will have loaded all the animations for every joint, all the bone weights and bone indices for each vertex as well as inverse bind poses, bind poses, bind shape matrices and root transform matrices. The only thing now is to get the vertex and index buffer from the <geometry> node, this is **exactly** the same as getting it for a static mesh so theres no point repeating it here.

At the point where you reach:

```
//-----------------------------------------------------------------------
-
// At this point, all static AND skinned meshes will have been
loaded // from the collada document stored in "MeshManager* meshes",
// Doowutchyalike
//-----------------------------------------------------------------------
-
```

That's skeletal meshes covered, convoluted but still somewhat simple. If you're a fan of 90's hiphop you'll get the stupid reference. Onto morphing meshes!

Chapter 6: Importing the data needed for morphing animation

Unfortunately for us, although COLLADA supports morphing animation, the COLLADA exporter doesn't seem to be able to export them correctly. Therefore whilst the code written here is perfect for loading morphing animation, until the exporter is fixed there's no real point to it. I will make up to you all for this shortcoming by making a tutorial on morphing using XNA. Visit my blog for more details.

What is needed for morphing animation?

After covering skeletal meshes you might be a bit burned out, morphing meshes are fairly easy by comparison. Essentially they are just static meshes so it's fairly straightforward.
So to answer the topic, what's needed for morphing animation is:

- A base mesh to which all targets will inherit any extraneous data, for example if you want a skinned morph target you would make the base mesh just the skinned and use it's weights for the targets.
- A set of target meshes to morph to based on some input weight.
- To animate you need a set of key's detailed the weight of the target morphing per certain time.

That's all that's needed for morphing animation, so like I said before it's pretty much just loading static meshes, so how exactly do we get the morphing mesh data? How is it represented in a .dae file?

- A Base mesh will be in the <visual_scene> as a node with an <instance_controller> which points to a <controller> with a single <morph> child.
- Target meshes will be under <visual_scene> as just regular meshes, you have to identify them by the description in the <morph> node under the <controller> for the base mesh.
- Animations can be found as individual nodes for each target mesh. You'll see animation for each target as being Base.TargetName. Usually the animation key's are exported as BEZIER, this doesn't matter though, you

can either ignore the tangents and linearly interpolate or just program a Bezier interpolation system.

Generally the biggest annoyance with loading morphing meshes is the identification of the targets.

A detailed walkthrough of the code for loading morphing animation

To start off with, let's look at the key of loading morphing meshes, the processVisualScenes() function, once again I will go through it's procedure instead of just pasting it's source.

1. Set up an array of booleans to cover every node in the visual scenes which will be used to set/check whether the node should be processed.
2. Find a morph target node, if any and check which nodes in the visual scene are connected to it as a target, for every node involved with the morph target set them to not be processed as standard mesh/skinned meshes.
3. Process skinned and static meshes as per normal.

The reason for careful processing selection is because we want the target meshes in the morphing mesh container rather than the static mesh container. After the processing of the visual scenes we will have geometry nodes of interest for processing in the processGeometries() function. The animations will also be gotten through the processAnimations() function.

That was a basic overview of the processVisualScenes() function as it is now, let me now walk you through the procedure for determining which nodes belong to the morph mesh:

For each node in visual_scene

a) Check if it has an <instance_controller> and check if the <controller> it point's to has a <morph> child. If not then skip this node.
b) Check the <source> node children of <morph> for one with an id containing "targets", if none is found then skip this morphing mesh as it has no targets.
c) From the <source> node containing the target names from before, extract each string contained(using processStringSource()).
d) Get the <geometry> node for the base mesh by reading the "source" attribute of the <morph> node. Remove the "#" at the beginning of it's name and search for the node to find it, if it isn't found then skip this morphing mesh.
e) Set the process boolean for this node as false, it's clear it is a valid morphing mesh, also you can now allocate memory for it with "new" at this point.
f) Now check every node in visual_scene if it's <geometry> node's name corresponds to any names specified in the <source> from part c), if so

then it is a target mesh and shouldn't be processed, it can also be added to the internal mesh targets array for the morphing mesh.

So at this point you now have all the geometry nodes for every mesh in the morphing mesh, the base and the targets. Anytime you want the geometry you can extract it using the exact same functions as the static meshes(see the processGeometries() in the source code).

All that's now needed is the animations, this operation is so simple there is no point even detailing it, you can just check the processAnimations() functions. I think at this point you are self-sufficient in dealing with COLLADA as we have covered absolutely everything at this point, all the rest in loading of morphing meshes is just derivative of static meshes.

Now that we have all the COLLADA data, you can either go off in OpenGL or you can read the appendix to see how to convert the data for use in DirectX!

Conclusion

I hope as you are reading this that you are now sitting there with a very good understanding of COLLADA, even though you may be tempted to just use the importer and forget about it, it's important that you understand the basic operations of loading data from a COLLADA file and how the data is generally presented.

The games industry is constantly evolving and therefore COLLADA must to evolve to meet their demands, thus changes to the exporting system and the data representations are inevitable. With a solid understanding of COLLADA you can easily adapt your own importer to any changes.

I think at this point also I must address the "content war" again. At this point you would pretty much understand all the good things about COLLADA, I advise you to try FBX though just to make up your own mind on which content format is the better choice. Maybe sometime in the future I may give a similar tutorial on FBX, though I doubt it.

Thank you for taking the time to read this document, I hope it served you well. Continue visiting my website for more tutorials! =)

Appendix (i): Using the data collected from COLLADA in DirectX

Check out COLLADADirectX folder in the file accompanying this text for the source code.

This will not be a full tutorial on how to set up vertex buffers or other basic operations for DirectX. The main focus is really in converting the geometry and matrices to be DirectX compatible. If you want to know about those simple operations just check the source code as it's all fairly simple to find.

The conversion of all the meshes is done before any DirectX resources are made. The conversion code is in the MeshManager class.

To make a static mesh DirectX compatible:

1. You must convert the World matrix for the mesh from COLLADA layout to DirectX layout
2. You must then multiply that converted matrix by a scaling matrix with a negative z.
3. The indices of the mesh must be converted to be in the correct layout. To do this you must do as follows:

```
for(unsigned int z = 0; z < StaticMeshes[i]->Indices.size(); (z += 3))
{
	unsigned int v0 = StaticMeshes[i]->Indices[z];
	unsigned int v1 = StaticMeshes[i]->Indices[z+1];
	unsigned int v2 = StaticMeshes[i]->Indices[z+2];

	StaticMeshes[i]->Indices[z] = v0;
	StaticMeshes[i]->Indices[z+1] = v2;
	StaticMeshes[i]->Indices[z+2] = v1;
}
```

To make a skinned mesh DirectX compatible:

1. Convert the RootTransform for the SkinnedMesh and multiply it by negative z scaling matrix
2. Convert every animation matrix from COLLADA format to DirectX format, DO NOT APPLY NEGATIVE Z SCALING MATRIX. Each animation matrix is a relative matrix from its previous joint, thus if you apply a negative z at the root joint of the skeleton, every joint that descends from it will automatically be fixed.
3. Convert each Joints bind and inverse bind matrices from COLLADA to DirectX, once again do not apply scaling matrix.
4. Convert the bind shape matrix for the mesh from COLLADA to DirectX, also without scaling matrix.
5. Convert the index buffer for the skinned mesh the same way as with a static mesh.

To make a morphing mesh DirectX compatible just convert each of its meshes the same as static meshes, as that's what they are.

For details on the shaders just check the .fx files included.

Appendix (ii): Using COLLADA refinery to make indexed meshes

Whenever you export a mesh from Maya etc. it will be an un-indexed primitive and thus criminally inefficient as well as unsupported. To convert all meshes in a file to being indexed do as follows in COLLADA refinery:

1. Create an "Input" node
2. Create an "Output" node
3. Create a "DeIndexer" node
4. Connect the "Input" to the "DeIndexer"
5. Connect the "DeIndexer" to the "Output"
6. Double-Click the "Input" node and find the .dae file you want to convert
7. Double-Click the "Output" node and specify where to output the converted scene, you can pick the exact same file as the input.
8. Click convert and that should be it!