Software Library version 1.0.3

Developers Manual

Contents

Chapter 1

Introduction

The OpenCTM file format is an open format for storing 3D triangle meshes. One of the main advantages over other similar file formats is its ability to losslessly compress the triangle geometry to a fraction of the corresponding raw data size.

This document describes how to use the OpenCTM API to load and save OpenCTM format files. It is mostly written for C/C++ users, but should be useful for other programming languages too, since the concepts and function calls are virtually identical regardless of programming language.

For a complete reference to the OpenCTM API, please use the Doxygen generated OpenCTM API Reference, which describes all API functions, types, constants etc.

Chapter 2

Concepts

2.1 The OpenCTM API

The OpenCTM API makes it easy to read and write OpenCTM format files. The API is implemented in the form of a software library that an application can be linked to in order to access the OpenCTM API.

The software library itself is written in standard, portable C language, but can be used from many other programming languages (writing language bindings for new languages should be fairly straight forward, since the API was written with cross-language portability in mind).

2.2 The triangle mesh

The triangle mesh, in OpenCTM terms, is managed in a format that is well suited for a modern 3D rendering pipeline, such as OpenGL.

At a glance, the OpenCTM mesh has the following properties:

- A vertex is a set of attributes that uniquely identify the vertex. This includes: vertex coordinate, normal, UV coordinate(s) and custom vertex attribute(s) (such as color, weight, etc).
- A triangle is described by three vertex indices.
- In the OpenCTM API, these mesh data are treated as arrays (an integer array for the triangle indices, and floating point arrays for the vertex data).
- All vertex data arrays in a mesh must have the same number of elements (for instance, there is exactly one normal associated with each vertex coordinate).

- All mesh data are optional, except for the triangle indices and the vertex coordinates. For instance, it is possible to leave out the normal information.

For an example of the mesh data structure see table 2.1 (vertex data) and table 2.2 (triangle data).

2.2.1 Triangle indices

Each triangle is described by three integers: one vertex index for each corner of the triangle). The triangle index array looks like this:

tri_0^0	tri_0^1	tri_0^2	tri_1^0	tri_1^1	tri_1^2	$\ldots$	tri_M^0	tri_M^1	tri_M^2

$\ldots$ where tri_k^j is the vertex index for the j:th corner of the k:th triangle.

2.2.2 Vertex coordinates

Each vertex coordinate is described by three floating point values: x, y and z. The vertex coordinate array looks like this:

x_0	y_0	z_0	x_1	y_1	z_1	$\ldots$	x_N	y_N	z_N

$\ldots$ where x_k, y_k and z_k are the x, y and z coordinates of the k:th vertex.

2.2.3 Normals

Each normal is described by three floating point values: x, y and z. The normal array looks like this:

x_0	y_0	z_0	x_1	y_1	z_1	$\ldots$	x_N	y_N	z_N

$\ldots$ where x_k, y_k and z_k are the x, y and z components of the k:th normal.

2.2.4 UV coordinates

A mesh may have several UV maps, where each UV map is described by:

- A UV coordinate array.
- A unique UV map name.
- A file name reference (optional).

Each UV coordinate is described by two floating point values: u and v. A UV coordinate array looks like this:

u_0	v_0	u_1	v_1	u_2	v_2	$\ldots$	u_N	v_N

$\ldots$ where u_k and v_k are the u and v components of the k:th UV coordinate.

2.2.5 Custom vertex attributes

A mesh may have several custom vertex attribute maps, where each attribute map is described by:

- A vertex attribute array.
- A unique attribute map name.

Each vertex attribute is described by four floating point values: a, b, c and d. An attribute array looks like this:

a_0	b_0	c_0	d_0	a_1	b_1	c_1	d_1	$\ldots$	a_N	b_N	c_N	d_N

$\ldots$ where a_k, b_k, c_k and d_k are the four attribute values of the k:th attribute.

2.3 The OpenCTM context

The OpenCTM API uses a *context* for almost all operations (function calls). The context is created and destroyed with the functions ctmNewContext() and ctmFreeContext(), respectively.

A program may instantiate any number of contexts, and all OpenCTM function calls are completely thread safe (multiple threads can use the OpenCTM API at the same time), as long as each context instance is handled by a single thread.

Each context is fully self contained and independent of other contexts.

There are two types of OpenCTM context: *import contexts* and *export contexts*. Import contexts are used for importing OpenCTM files, and export contexts are used for exporting OpenCTM files.

The context type is selected when creating the context.

Index	0	1	2	3	4	...	N
Vertex	v_0	v_1	v_2	v_3	v_4	...	v_N
Normal	n_0	n_1	n_2	n_3	n_4	...	n_N
UVCoord1	$t1_0$	$t1_1$	$t1_2$	$t1_3$	$t1_4$	...	$t1_N$
UVCoord2	$t2_0$	$t2_1$	$t2_2$	$t2_3$	$t2_4$	...	$t2_N$
Attrib1	$a1_0$	$a1_1$	$a1_2$	$a1_3$	$a1_4$	...	$a1_N$
Attrib2	$a2_0$	$a2_1$	$a2_2$	$a2_3$	$a2_4$	...	$a2_N$

Table 2.1: Mesh vertex data structure in OpenCTM, for a mesh with normals, two UV coordinates per vertex, and two custom attributes per vertex.

Triangle	tri_0	tri_1	tri_2	tri_3	tri_4	...	tri_M

Table 2.2: Mesh triangle data structure in OpenCTM, where tri_k is a tuple of three vertex indices. For instance, $tri_0 = (0, 1, 2)$, $tri_1 = (0, 2, 3)$, $tri_2 = (3, 5, 4)$, ...

Chapter 3

Compression Methods

The OpenCTM file format supports a few different compression methods, each with its own advantages and disadvantages. The API makes it possible to select which method to use when creating OpenCTM files (the default method is MG1).

3.1 RAW

The RAW compression method is not really a compression method, since it only stores the data in a raw, uncompressed form. The result is a file with the same size and data format as the in-memory mesh data structure.

The RAW method is mostly useful for testing purposes, but can be preferred in certain situations, for instance when file writing speeds and a small memory footprint is more important than minimizing file sizes.

Another situation where the RAW method can be useful is when you need an easily parsable binary file format. Usually the OpenCTM API can be used in almost any application, but in some environments, such as certain script languages or data inspecion tools, it can be handy to have access to the raw data.

3.2 MG1

The MG1 compression method effectively reduces the size of the mesh data by re-coding the connectivity information of the mesh into an easily compressible format. The data is then compressed using LZMA.

The floating point data, such as vertex coordinates and normals, is fully preserved in the MG1 method, by simply applying lossless LZMA compression to it.

Under typical condititions, the connectivity information is compressed to about two bytes per triangle (17% of the original size), and vertex data is compressed to

about 75% of the original size.

While creating MG1 files can be a relatively slow process (compared to the RAW method, for instance) the reading speed is usually very high, thanks to the fast LZMA decoder and the uncomplicated data format.

3.3 MG2

The MG2 compression method offers the highest level of compression among the different OpenCTM methods. It uses the same method for compressing connectivity information as the MG1 method, but does a better job at compressing vertex data.

Vertex data is converted to a fixed point representation, which allows for efficient, lossless, prediction based data compression algorithms.

In short, the MG2 method divides the mesh into small sub-spaces, sorts the data geometrically, and applies delta-prediction to the data, which effectively lowers the data entropy. The re-coded vertex data is then compressed with LZMA.

When using the OpenCTM API for creating MG2 files you can trade mesh resolution for compression ratio, and the API provides several functions for controlling the resolution of different vertex attributes independently. Therefor it is usually important to know the resolution requirements for your specific application when using the MG2 method.

In some applications, such as games, movies and art, it is important that the 3D model is not visually degraded by compression. In such applications you will typically tune your resolution settings using trial and error, until you find a setting that does not alter the model visually.

In other applications, such as CAD/CAM, 3D scanning, calibration, etc, reasonable resolution settings can usually be derived from the limitations of the process in which the model is used. For instance, there is usually no need for nanometer precision in the design of an airplane wing, and there is little use of micrometer resolution in a manufacturing process that can not reproduce features smaller than 0.15 mm.

As a side effect of the fact that MG2 produces smaller files than the MG1 method does, loading files is usually faster with the MG2 method than with the MG1 method. Saving files with the MG2 method is about as fast as with the MG1 method.

Chapter 4

Basic Usage

4.1 Prerequisites

To use the OpenCTM API, you need to include the OpenCTM include file, like this:

```
#include <openctm.h>
```

You also need to link with the OpenCTM import library. For instance, in MS Visual Studio you can add ”openctm.lib” to your Additional Dependencies field in the Linker section. For gcc/g++ or similar compilers, you will typically add -lopenctm to the list of compiler options, for instance:

```
> g++ -o foo foo.cpp -lopenctm
```

4.2 Loading OpenCTM files

Below is a minimal example of how to load an OpenCTM file with the OpenCTM API, in just a few lines of code:

```
CTMcontext context;
CTMuint vertCount, triCount, * indices;
CTMfloat * vertices;

// Create a new importer context
context = ctmNewContext(CTM_IMPORT);

// Load the OpenCTM file
ctmLoad(context, "mymesh.ctm");
if(ctmGetError(context) == CTM_NONE)
{
  // Access the mesh data
```

```
  vertCount = ctmGetInteger(context, CTM_VERTEX_COUNT);
  vertices = ctmGetFloatArray(context, CTM_VERTICES);
  triCount = ctmGetInteger(context, CTM_TRIANGLE_COUNT);
  indices = ctmGetIntegerArray(context, CTM_INDICES);

  // Deal with the mesh (e.g. transcode it to our
  // internal representation)
  // ...
}

// Free the context
ctmFreeContext(context);
```

4.3 Creating OpenCTM files

Below is a minimal example of how to save an OpenCTM file with the OpenCTM API, in just a few lines of code:

```
void MySaveFile(CTMuint aVertCount, CTMuint aTriCount,
  CTMfloat * aVertices, CTMuint * aIndices,
  const char * aFileName)
{
  CTMcontext context;

  // Create a new exporter context
  context = ctmNewContext(CTM_EXPORT);

  // Define our mesh representation to OpenCTM
  ctmDefineMesh(context, aVertices, aVertCount, aIndices,
      aTriCount, NULL);

  // Save the OpenCTM file
  ctmSave(context, aFileName);

  // Free the context
  ctmFreeContext(context);
}
```

Chapter 5

Controlling Compression

When creating OpenCTM files, one of the most important things to control with the API is the compression method.

5.1 Selecting the compression method

You can select which compression method to use with the ctmCompressionMethod() function. The different options are:

Name	Description
CTM_METHOD_RAW	Use the RAW compression method.
CTM_METHOD_MG1	Use the MG1 compression method (default).
CTM_METHOD_MG2	Use the MG2 compression method.

For instance, to select the MG2 compression method for a given OpenCTM context, use:

```
ctmCompressionMethod(context, CTM_METHOD_MG2);
```

5.2 Selecting the compression level

You can select which LZMA compression level to use with the ctmCompressionLevel() function. The compression level can be in the range 0-9, where 0 is the fastest compression, and 9 is the best compression. The compression level also affects the amount of memory that is used during compression (anywhere from a few megabytes to several hundred megabytes).

```
ctmCompressionMethod(context, 4);
```

The default compression level is 1.

5.3 Selecting fixed point precision

When the MG2 compression method is used, further compression control is provided through the API that deals with the fixed point precision for different vertex attributes. The different attribute precisions that can be controlled are:

Attribute	**API function**
Vertex coordinate	ctmVertexPrecision() / ctmVertexPrecisionRel()
Normal	ctmNormalPrecision()
UV coordinates	ctmUVCoordPrecision()
Custom attributes	ctmAttribPrecision()

Reasonable default values for the fixed point precisions are selected by the API unless the corresponding API functions are called. However, the API does not know the requirements for the mesh, which is why it is always a good idea to specify the fixed point precision that is most relevant for your specific mesh.

5.3.1 Vertex coordinate precision

The vertex coordinate precision can be controlled in two ways:

- Absolute precision - ctmVertexPrecision().
- Relative precision - ctmVertexPrecisionRel().

You typically specify the absolute precision when you know the properties of the mesh and what is going to be used for (for instance, if it is a product of a measurment procecss, or if it will be used in a manufacturing process). For example, if the vertex coordinate unit is meters, and the precision is specified as 0.001, the fixed point precision will be 1 mm:

```
ctmVertexPrecision(context, 0.001);
```

When you do not know much about the mesh, it can be useful to specify the relative precision. The ctmVertexPrecisionRel() function will analyze the mesh to find a useful base measure, which is multiplied by a scaling factor that is given as an argument to the function.

The relative precision function uses the average triangle edge length as the base measure. So for example, if you specify 0.01 as the relative precision, the precision will be 1% of the average triangle edge length, which is usually a good figure for meshes that will be used in visualization applications:

```
ctmVertexPrecisionRel(context, 0.01);
```

It should be noted that unlike the ctmVertexPrecision() function, the ctmVertexPrecisionRel() function requires that the mesh has been specified before calling the function.

The default vertex coordinate precision is $2^{-10} \approx 0.00098$.

5.3.2 Normal precision

In the MG2 compression method, each vertex normal is represented in spherical coordinates (the coordinate system is aligned to the average normal of all triangles that connect to the vertex).

The precision controls both the angular resolution and the radial resolution (magnitude). For instance, 0.01 means that the circle is divided into 100 steps, and the normal magnitude is rounded to 2 decimals:

```
ctmNormalPrecision(context, 0.01);
```

The default normal precision is $2^{-8} \approx 0.0039$.

5.3.3 UV coordinate precision

UV coordinate precision is specified on a per UV map basis, and gives the absolute precision in UV coordinate space.

The effects of different precisions depend on many different things. For instance if the UV map is used for mapping a 2D texture onto the triangle mesh, the resolution of the texture can influence the required UV coordinate precision (e.g. a 4096x4096 texture may require better precision than a 256x256 texture). The resolution of the mesh may also affect the required UV coordinate precision.

To specify a resolution of 0.001 for the UV map $uvMap$, use:

```
ctmUVCoordPrecision(context, uvMap, 0.001);
```

The default UV coordinate precision is $2^{-12} \approx 0.00024$.

5.3.4 Custom attribute precision

As with UV coordinates, the precision for custom vertex attributes are specified on a per attribute basis.

The precision of a custom attribute depends entirely on the type of attribute. For instance, standard color attributes typically do not require more then eigh bits per component, which means that $1/256$ is a good precision setting (if the value range is $[0, 1]$):

```
ctmAttribPrecision(context, attribMap, 1.0/256.0);
```

For integer values, the precision 1.0 is a good choice.

The default vertex attribute precision is $2^{-8} \approx 0.0039$.

Chapter 6

Error Handling

An error can occur when calling any of the OpenCTM API functions. To check for errors, call the ctmGetError() function, which returns a positive error code if something went wrong, or zero (CTM_NONE) if no error has occured.

See 6.1 for a list of possible error codes.

The last error code that indicates a failure is stored per OpenCTM context until the ctmGetError() function is called. Calling the function will reset the error state.

It is also possible to convert an error code to an error string, using the ctmErrorString() function, which takes an error code as its argument, and returns a constant C string (pointer to a null terminated UTF-8 format character string).

Code	Description
CTM_NONE (zero)	No error has occured (everything is OK).
CTM_INVALID_CONTEXT	The OpenCTM context was invalid (e.g. NULL).
CTM_INVALID_ARGUMENT	A function argument was invalid.
CTM_INVALID_OPERATION	The operation is not allowed.
CTM_INVALID_MESH	The mesh was invalid (e.g. no vertices).
CTM_OUT_OF_MEMORY	Not enough memory to proceed.
CTM_FILE_ERROR	File I/O error.
CTM_BAD_FORMAT	File format error (e.g. unrecognized format or corrupted file).
CTM_LZMA_ERROR	An error occured within the LZMA library.
CTM_INTERNAL_ERROR	An internal error occured (indicates a bug).
CTM_UNSUPPORTED_FORMAT_VERSION	Unsupported file format version.

Table 6.1: OpenCTM error codes.

Chapter 7

C++ Extensions

To take better advantage of some of the C++ language features, such as exception handling, a few C++ wrapper classes are availbale through the standard API when compiling a C++ program. As usual, just include ”openctm.h”, and you will have access to two C++ classes: CTMimporer and CTMexporter.

The main differences between the C++ classes and the standard API are:

- The C++ classes call ctmNewContext() and ctmFreeContext() in their constructors and destructors respectively, which makes it easier to use the C++ dynamic scope mechanisms (such as exception handling).
- Whenever an OpenCTM error occurs, an exception is thrown. Hence, there is no method corresponding to the ctmGetError() function.

7.1 The CTMimporter class

Here is an example of how to use the CTMimporter class in C++:

```
try
{
  // Create a new OpenCTM importer object
  CTMimporter ctm;

  // Load the OpenCTM file
  ctm.Load("mymesh.ctm");

  // Access the mesh data
  CTMuint vertCount = ctm.GetInteger(CTM_VERTEX_COUNT);
  CTMfloat * vertices = ctm.GetFloatArray(CTM_VERTICES);
  CTMuint triCount = ctm.GetInteger(CTM_TRIANGLE_COUNT);
  CTMuint * indices = ctm.GetIntegerArray(CTM_INDICES);
```

```
  // Deal with the mesh (e.g. transcode it to our
  // internal representation)
  // ...
}
catch(exception &e)
{
  cout << "Error: " << e.what() << endl;
}
```

7.2 The CTMexporter class

Here is an example of how to use the CTMexporter class in C++:

```
void MySaveFile(CTMuint aVertCount, CTMuint aTriCount,
  CTMfloat * aVertices, CTMuint * aIndices,
  const char * aFileName)
{
  try
  {
    // Create a new OpenCTM exporter object
    CTMexporter ctm;

    // Define our mesh representation to OpenCTM
    ctm.DefineMesh(aVertices, aVertCount, aIndices,
        aTriCount, NULL);

    // Save the OpenCTM file
    ctm.Save(aFileName);
  }
  catch(exception &e)
  {
    cout << "Error: " << e.what() << endl;
  }
}
```