Parser Project
3DM Praser for Precomp-cpp

Documentation

Jerguš Lysý **11.2. 2018**

Table of Contents

1 Introduction

3DM file is an open source 3DM model format and native file format for Rhinoceros. It contains a 3D model which includes surface, points and curve information. These files allow CAD, CAM CAE, and computer graphics software to accurately save and exchange 3D geometry using both NURBS and polygon mesh representation.

An official SDK to work with 3DM files is openNURBS[1]. This SDK consists of C++ source code for a library that read and write 3DM files. The proposed and implemented parser is based on this library and certain aspects of files are read out of from the source code of openNURBS. However the parser simplifies the work with 3DM.

Precomp-cpp is an open source project started by Christian Schneider and open for any contribution. The project tries to reduce the size of certain file formats even more than it would be by directly compressing them with a certain compression method. The parser, as proposed and implemented, was specifically made for precomp-cpp project in order to help to reduce size of 3DM even more than the precomp-cpp can do so currently in intense mode.

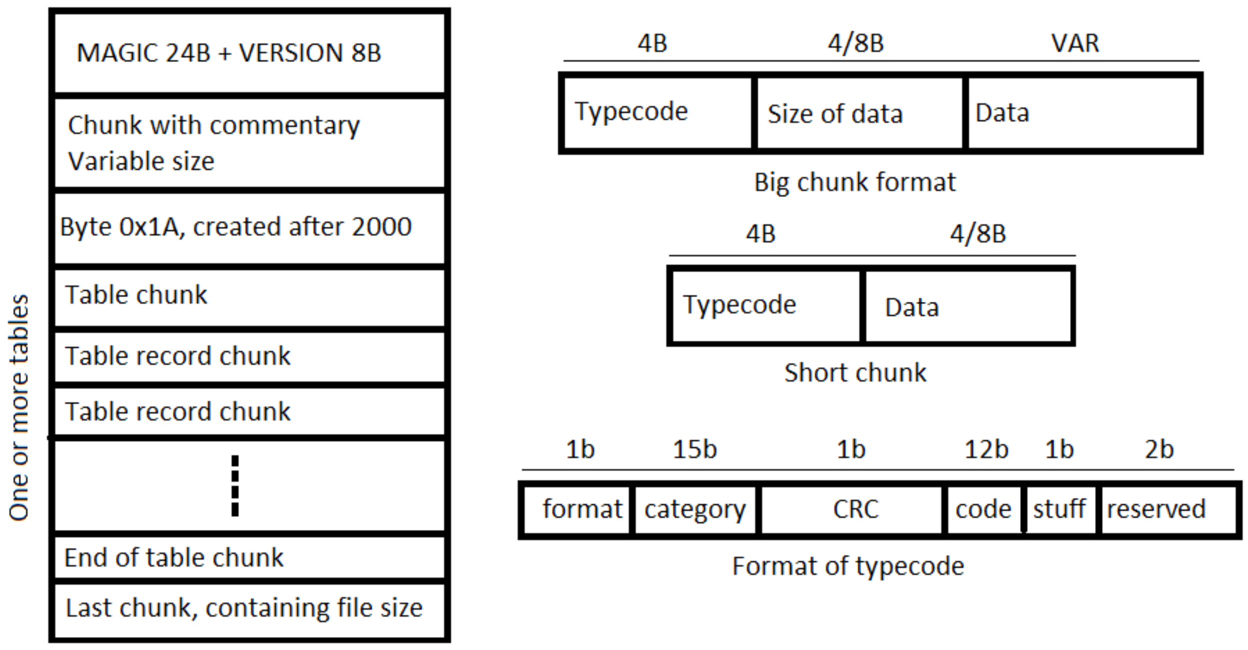
1 https://www.rhino3d.com/opennurbs

2 .3dm file format

Structure of the format can be divided to three main parts:

1. Beginning of the file consisting magic constant, along with version, commentary block that describes how and when the file was created.
2. The body of the file are called chunks (in fact the commentary block is already a chunk) that carry information about the 3D model. These chunks are usually tables which contain one or more chunks containing data related to these tales. Such data can be be a bitmap, various shapes used in model, compressed preview image, and many more.
3. The very last block of the file is last chunk that is specified by carrying the size of the file.

The general structure of 3DM file and its parts is showed on the following picture.



Chunk starts with a typecode. This typecode contains information about the chunk such as the format, type of chunk (table or record), whether CRC is calculated and more. Depending on the format, there are two types of chunks: short and big. The short chunk contains typecode and the 4 or 8 bytes long data. This size depends on when the file was created that is a version of the file. The big chunk, on the other hand, contains an actual data size that follows, instead of the data itself. Some of the big chunk also contain CRC that is attached to the end of the data. This happen in case when the CRC flag in typecode is set.

The version of the file and its creation date are important from the parsing prespective. Based on it, it can be distinguished, whether the size of chunk data or data itself are 4 or 8 bytes long. Moreover, the size of attached CRC is 2 bytes in old files, while it is 4 bytes long in newer ones.

3 Parser and implementation

The implemented parser is written in C and directly uses some of the source files from openNURBS SDK. Its parsing capabilities are, however, much weaker than those of openNURBS. Here are the major differences:

- The parser does not check each possible version of files and it does not check the date of creation of the file at all. The files are recognized as “old” (version 4 and everything older) and “new” (everything newer than version 4). The parser must recognize at least this since the process of parsing would fail in reading out the correct number of bytes for storing sizes or CRC.
- The parser cannot really perform deep inspection or dump each chunk of the file. The parser only needs to find certain data to dump and validate the input. If this cannot be done then the parser fails and returns an error.
- The parser cannot properly parse old files containing legacy typecodes. These files contained chunks that were not tables freely in the file. The structure of such files is not same as was described above and therefore there are not considered. Moreover, some of the older writers that were used added goo chunks freely in files or added their own header. None of these files can be parsed correctly and will return an error.
- The parser was made only to one pupose, not to be a dynamic reader/writer. This purpose is to find and dump compressed data and it was made and adjusted to work with precomp-cpp. Therefore much functionality, such as openNURBS provides, was left out to give the precomp-cpp a way to work with files even if they were corrupted or are not completely valid.

The main source code of the parser is file *3dm.c* and its header file *3dm.h*. The files starting with opennurbs are taked directly from the openNURBS. There are three such files *opennurbs_crc.h*, *opennurbs_crc.c* and *opennurbs_3dm.h*. The former two are used to calculate CRC on data. Two functions are implemented ON_CRC16 and ON_CRC32. These two calculate CRC of size 2 or 4 bytes. They are modified versions of CRC calculation function from zlib library. The later file is used for constants. It contains all the typecodes, even the legacy ones, and describes structure of some of the record types – typecode for a certain table has its own derived typecode for its record. Parser exposes function *parse_tdm_file* to a user[2]. This function takes a file descriptor as an argument and performs parsing on a file. The output is an information about located compressed data inside file and its dump. If the function fails then an error can be printed by function *print_tdm_error*. Error can happen because of one these reason:

- an invalid input was given to a funcion (TDM_ERROR_FUNC_INPUT),
- a file is invalid that is parser failed to correctly parse it (TDM_ERROR_INVALID_FILE),

2 This is a function to be used by user. It is not what precomp-cpp needs and wants to use. This function was also used for testing purposes of the parser.

- a typecode is invalid - mostly this is because a file was corrupted and so parser expected a valid typecode while given some incorrect value in a certain stage of parser (TDM_ERROR_INVALID_TCODE),
- there was not enough memory for allocation – this can happen when size of big chunk is too large[3] (TDM_ERROR_OUT_OF_MEMORY),
- invalid CRC – this is the case when compressed data inside file contain CRC that does not match with the calculated one (TDM_ERROR_INVALID_CRC),
- a file version is not correct – this is mostly when version could not be properly parsed the value of version is different as expected (TDM_INVALID_FILE_VER),

The mentioned function of the parser performs the following steps:

1. Get the file size. This will be used at the end of parsing when the size of the file will be read from the last chunk.
2. Parse header using *parse_header* function. This static function checks the magic of file, the correct alignment of the version and version itself, which is also returned as an output.
3. First chunk is read as a big chunk (TCODE_COMMENTBLOCK). The reading itself is done by first peeking to the typecode and if it matches then skipping it.
4. The loop for reading tables follows. This loop is not endless and will quit after 100 steps. This number can be different and it is only to prevent a program from running forever. First, the parser peek at the typecode in file. This typecode is then match with typecodes for property table, bitmap table and last chunk. If none of them matched then check whether the typecode represents a table, if so skip it, if not return an error.
5. If the typecode is for property table, then run the function *parse_property_table*. This functions parses and skips all the records in this table, but the one representing compressed preview image. When such a chunk carrying this image is located then parse this chunk and finally dump the compressed data (the compression is inside anonymous chunk, which needs to be there).
6. The case for typecode representing bitmap table is implemented, however the implementation for the parser itself is not. This table can contain bitmaps which can be compressed too.
7. If the last chunk is read, then read this chunk and check whether it data (representing the size of file) matches the an actual size of file that was saved at the beginning of the program.

3 Check the security issues of the parser at the end of this document.

4 Precomp-cpp project

The purpose of the project is to compress files containing zlib or deflate streams even more effectively than they could be otherwise. Therefore, precomp-cpp works effectively with such files containing those compressed streams. First, a file is read and if a compressed stream is found then decompression can take place. This can be done by knowing an actual compressed stream or just by brute-forcing the size. The decompressed stream is then stored instead of the compressed stream and the file is outputed with .pcf format. This file can be either larger or same size (depending on the compression used inside the file). After this, an entire file can be recompressed again with *lzma2* or *bZip2* which likely gives even less size of file than was the original one (with compressed stream inside).

Precomp-cpp can also process files that it cannot recognize. For that it has intense (testing magic for a zlib stream) or brute-force (brute force detection of zlib) modes that search for zlib streams. Then it tries to guess the best size of such a stream in order to give the best ration for compressed/decompressed size. However, such searching of zlib streams is a time consuming process as the pure stream contains only three possible magic values that are only two bytes long (eg. 78 DA for best compression). Therefore it is convenient to parse the data and find the compressed streams inside without doing a brute-force.

Parser, as was proposed, searches for such zlib stream(s). Such a stream is contained inside a chunk that contains preview image (TCODE_PROPERTIES_COMPRESSED_PREVIEWIMAGE). This chunk is a primary goal for the parser to find. If it cannot be found then no change is done in file and precomp-cpp can compress the file without any other pre-compression.

5 Security issues and testing of parser

The parser was tested throughout the development as well as after that. The following methods and tools were used:

- Static methods – compiler -Wextra and -Wall switch, cppcheck, source code continuosly checking by the author.
- Dynamic methods – valgrind (with memory corruption checking), gdb
- Blackbox (fuzzi) – radamsa

There are some issues that were also directly found by the static or dynamic checkers. The first notable issue is that the file descriptor is almost never closed when the parser fails. This was found by cppcheck as well as valgrind (memory after unsuccessful run). The other issue is the size that can be read. The parser basically read chunks and stores them in memory. This is proper in case of short chunks, however big chunks specifies the size of data and an input can be easily crafted to have as much size as possible. In best case this causes *malloc* to fail. In worse case this can cause a slight DoS by working with a large portion of memory. There are two cases when big chunks are read. First is when reading out compressed preview image. The data is allocated and stored in this chunk. Whener a function is about to return an error, the data is freed. The second case is reading last chunk. When this chunk is read a flag variable *do_quit* is set to 1. After the main loop, this variable is tested and chunk is freed, if not then the function returns an error. If the reading big chunk fails the data is not allocated and error is returned.

The parser was tested with some (valid) sample 3dm files. These are located in folder Samples and a script that automate testing of all them is *test_samples.py*. After running the test all of the files were parser properly and no error appeared (which was the desired state). Next test, performed using bash script *test_radamsa.sh*, is running fuzzer Radamsa on parser. The script randomly chooses a file from Sample, put it as input for Radamsa, creates an output, fuzzed file, and then run parser on this input. This is done until the program crashes or user decides to stop it. After running it for and hour, no malformed input caused a crash of program.