CMPSC 311 Assignment 4: 3D Model File Format

Out: March 30, 2015 Due: April 17, 2015

Purpose

This assignment is designed to give you further practice in understanding and creating abstractions, as well as implementing some simple binary file I/O. Given the specification for a 3D model file format, you will create a basic library which allows the user to read models from a file, modify them, print them on a TipTap, and save them to a new file.

Description

In this assignment, you will create a layer on top of the device driver you implemented in Assignment 3. To be clear: **you must start this assignment using the code that *you* submitted for Assignment 3**. The new layer will be a basic library to work with 3D models stored in binary files. This library will allow for reading a model from a file, writing a model to a file, and adding chunks to a model. In addition, it will interface with the driver from the previous assignment to print a model on an attached TipTap.

File format

The 3DM format is a fictitious 3D model file format which stores two main pieces of information:

- The title of the model
- The material in the model and where it goes (organized into "chunks")

Each 3DM file begins with a 17-byte header at offset 0 in the file, and its structure is as follows (offset and size are measured in *bytes*, not bits):

Offset	**Size**	**Description**
0	4	Magic number `0x03113d6d` to identify file format
4	4	File offset where chunk list begins
8	4	File offset where title begins
12	4	Number of chunks
16	1	Length of title

All integers in a 3DM file are unsigned and in *big-endian* byte order. You will need to read the `man` pages for `ntohs` and `ntohl` to learn to convert values using appropriate functions, where "network byte order" means big-endian. (Attempting to do byte-order conversion by hand is not portable!)

The title is simply a string describing the model. This can contain any characters except the null character. The file format does not itself require a null terminator for the title string, so you will need to be careful to ensure your C string is properly terminated.

Chunk data

The model represented by a 3DM file is composed of a list of chunks. Each chunk represents a rectangular "chunk" of one material, located at a particular X, Y, and Z coordinate, with a given width, height, and depth. A chunk is represented in the following 13-byte format (again, offset and size are in *bytes*, not bits):

Offset	**Size**	**Description**
0	2	Left X coordinate
2	2	Top Y coordinate
4	2	Lowest Z coordinate
6	2	Width (X dimension)
8	2	Height (Y dimension)
10	2	Depth (Z dimension)
12	1	Material ID

The first chunk is located at the offset given in the 3DM file header. Each subsequent chunk immediately follows the previous one in the file, so that every 13 bytes will be the start of a new chunk until the number specified by the file header is reached. If two or more chunks overlap with each other, then the material in the overlapping locations comes from whichever chunk is *last* in the list. (This means that, as you scan through the list of chunks to construct a layer to print, you can just overwrite the material from earlier chunks.)

Library API (your code)

Your code will provide an API of several functions which can be used by an application to load, save, and manipulate 3DM models as well as print them on an attached TipTap. These functions can, when needed, call the functions provided by your TipTap driver.

- `model_init(md, title)`: Initializes all of the fields in the given model. The title should be set to the given value, and the model should have 0 dimensions and no chunks. If the `title` parameter is too long to fit in the allotted space, it should be truncated to fit. This function should return 0 for success and nonzero if anything fails.
- `model_add_chunk(md, ck)`: Adds a new chunk to an existing model. You will need to allocate or reallocate memory to extend the array of chunks in the model, add this chunk to the array, and update the dimensions of the model if necessary. This function should return 0 for success and nonzero if anything fails.
- `model_destroy(md)`: Releases any resources which are allocated by the given model.
- `model_load(md, filename)`: Fills a model struct with data from the given file. You may assume the model has just been initialized with `model_init`. This function should open the model file using low-level I/O, read the file header, title, and chunk data from their respective offsets, and fill out the fields in the model struct. You will need to allocate and initialize any memory needed by the struct (or call an existing function which does this). After the data has been loaded, the file should be closed.

  Each field must be checked for validity. The magic number should be correct, and you should not encounter an early end-of-file before you have read the entire header, title, or list of chunks. If anything fails or any field is not valid, this function should destroy the model, close the file, and return nonzero. Otherwise, return 0 for success.

- `model_print(md, tt, x, y)`: Prints the model starting at coordinates (`x`, `y`, 0) on the given TipTap. This function should position the nozzle to begin printing, then print each successive layer starting with Z=0. You will need to set up the `materials` array by iterating over the chunks and setting the appropriate elements to the chunk's material if it crosses the current Z layer. Once this array is set up, you can call your driver to print the layer.

  This function should assume that the TipTap is already initialized and should not destroy it when it is finished. It should return 0 for success and nonzero if anything fails.

Testing

The provided verifier program, `a4verify`, will run a number of tests on each function in your implementation, using the provided models in the `models` directory, and output information about which tests have failed. The verifier also tests for correct handling of error conditions, so you may also see error messages in the output.

If you need to inspect the data in a particular model file to aid in debugging, you can do this using the `hexdump` utility, e.g.:

```
$ hexdump -C models/model2.3dm
```

If you cannot determine the source of the problem from the output of the verifier and inspecting any relevant models, you will need to use a debugger to locate the issue. In this assignment, the verifier is a binary executable that doesn't take extra arguments, so you will run GDB as follows:

```
$ gdb a4verify
(gdb) ... set up any breakpoints if needed ...
(gdb) run
```

You are also encouraged to install Valgrind and use it to check your code for memory errors:

```
$ sudo apt-get install valgrind
(enter your password)
$ valgrind ./a4verify
```

(Note: the verifier from Assignment 3 is still available as `a3verify`, so if you need to change your driver code you can use this verifier to test it.)

Procedure

1. Log into your virtual machine and open a terminal emulator.

2. Download the starter code and extract the tarball as you did in previous assignments:

   `http://www.cse.psu.edu/~djp284/cmpsc311-s15/docs/assign4.tar.gz`

   This will create a directory `assign4` containing these files:

   - `model.c`: Library implementation. All of your code goes here. *This is the only file you need to edit.*

- `model.h`: Header file containing prototypes for the functions you will be implementing and structs to store model and chunk state. You should not change this file.
- `libtiptap.a`: TipTap implementation, provided as a static library. (This is the 64-bit version; on a 32-bit machine, edit the Makefile to include `libtiptap32.a` instead.)
- `verify.c`: Program which runs unit tests on the model format implementation.
- `Makefile`: Makefile for the entire project.
- `models/`: Directory of sample 3DM-format models.

3. Copy your `driver.c` and `driver.h` from Assignment 3 into the `assign4` directory.
4. Build the starter code with `make` and ensure that you are able to run the unit tests.
5. Comment all of your code as you write it to explain what the code is doing.
6. Begin implementing the four functions from the Library API in `model.c`, following along with the unit tests as they guide you through the implementation.

Submission

1. Run `make clean` to remove any build products (binaries, object files, etc.) from the directory.
2. Create a tarball containing the `assign4` directory, complete with the source code you have written. The tarball should be named `LASTNAME-uid1234-assign4.tar.gz`, where `LASTNAME` is your last name in all capital letters and `uid1234` is your PSU email ID (the one you would use to log into ANGEL). For example, if the instructor were submitting this assignment, he would run:

```
$ tar -cvzf POHLY-djp284-assign4.tar.gz assign4/
```

3. Verify that the tarball contains all of your source code files by viewing the file list:

```
$ tar -tvzf LASTNAME-uid1234-assign4.tar.gz
```

4. Before sending the tarball, test it in a clean temporary directory using the following commands:

```
$ mkdir /tmp/assign4test
$ cp LASTNAME-uid1234-assign4.tar.gz /tmp/assign4test
$ cd /tmp/assign4test
$ tar -xvzf LASTNAME-uid1234-assign4.tar.gz
$ cd assign4
$ make
$ ./verify
```

5. Send the tarball in an email with the subject line “311 submission” to both the instructor (`djpohly@cse.psu.edu`) and the TA for your section (listed on the course website). Put your own email in the Cc: line so you can be sure the email was sent properly and contains everything. **YOU ARE RESPONSIBLE FOR DOING THIS CORRECTLY.** Lost emails, emails not sent to both the instructor and TA, bad tarballs, and missing files are the student’s responsibility and will be treated as late or missing. There will be no exceptions.

   This email should be sent by 11:59pm of the due date of the assignment; late submissions will be accepted up to **TWO** days late, with a penalty of 10% for each day. This late period is your extension; no further extensions will be granted.

Bonus Points

Implement the one remaining function, `model_save`. This function creates a new 3DM file with the given filename and writes the data from the given model struct into it using low-level I/O. If the file already exists, it should be truncated and overwritten. The file should be closed when the function finishes. This function should return 0 for success and nonzero if anything fails.

Important reminder

As with all assignments in this class, you are not permitted to:

- Copy any content from the Internet.
- Discuss or share ideas, code, configuration, text, or anything else related to the assignment with others in the class.
- Solicit help from anyone inside or outside the class other than the instructor, TAs, or assistants explicitly permitted by the instructor.
- Allow your own ideas and code to fall into the hands of others. You are responsible for protecting your code.

If you are unsure of what is permitted, you should ask the instructor *before* doing it.

You may be asked by the instructor or TA to explain portions of your code. If you cannot explain how or why your own submission works, it will be considered plagiarized. Consulting online sources is acceptable, but under no circumstances should *anything* be copied. Failure to abide by this requirement will result in dismissal from the class as described in the course syllabus.

Above all: when in doubt, be honest. If you do for any reason get significant help online, in person, etc., or use code from a source such as previous projects or other courses' lecture notes, please *document it in your submission* before we have to ask you about it. This will go a long way!