Scott Bengs

Voxel optimizations for Dwarf Fortress like worlds

This paper is about optimizations when it comes to worlds that are similar to Dwarf Fortress. Examples include Minecraft, 7 Days to Die(the original versions), Cube World, and many more. First the paper will go into detail about Dwarf Fortress and the 3D visualizer that uses its data and then the details on what algorithms and their implementations will follow. Feel free to skip around.

A quick note is that I'll be comparing mostly Minecraft and Dwarf Fortress which have similar systems. When talking about algorithms I'll be focused on my implementation. A lot of it can be applied elsewhere and I'll try to point that out as we cover the material. Please do look into all of these techniques more. The sources listed and many more have excellent pictures and videos to better illustrate what is going on and how.

I won't be pasting code here since the paper is already getting very long. Instead I'll link in each relevant section to the files directly on GitHub. No account is needed to view them.

Table of contents

1. Dwarf Fortress
2. Opal Prospect
3. What are voxels and why use them
4. Model optimizations
5. Run length encoding
6. Huffman Encoding
7. Zlib's deflate
8. Comparisons
9. Conclusion

Dwarf Fortress

First it helps to understand a little about the game and how its world works. It is a voxel game where you don't directly control dwarves. You oversee and place designations to have material mined, things constructed, farm plots to be made, and so on. It's a very complex game and this paper would be too long to go into detail about the mechanics. It uses ASCII graphics which many people find difficult to use. Instead, it will focus on the world and how it's stored.

The main feature is that the world is made up of voxels and is almost 100% modifiable. There are some materials that can't be mined or destroyed by any normal means. Other than that it's completely changeable. This is a common theme most voxel games employ. They have a world where you can change it as you see fit.

Dwarf Fortress breaks the world up into many small pieces. It goes much further than Minecraft does. Minecraft only uses chunks. They are 16x256x16 sized volumes. As you'll soon see, powers of 2 are commonly used. This allows the developer to use left and right shifts instead of full multiplication. Shifts allow one to multiply or divide by a power of two only. It's much faster than using normal multiplication or division. It comes at the cost of only using numbers such as 2, 4, 8, and so on as previously noted.

DF uses 16x16 areas of tiles as its smallest unit. These are called map blocks. They contain 256 tiles in them. Those fit into embark tiles which are 3x3 areas of map blocks. They are 48x48 tiles in size. You'll notice I have not mentioned a height. This is because it is entirely dependent on the specific part of the map and whether or not there is a mountain there, among other things. All of these blocks are layered on top of each other to give height. Embark tiles are contained in region blocks which are contained in world tiles. A standard embark is a 4x4. This refers to 4x4 embark tiles and thus 192x192 tiles per layer. An average height of 150 layers comes out to 5,529,600 tiles

in a fortress map. This can go higher or lower if the embark size is changed.

Now that you have some idea of how tiles and blocks work, and a good baseline size to work with, we can move onto DFHack and what it does. It is simply a mod for the game that allows access to all of its memory. Since it's a singleplayer game there is no way to hurt other players. I have never been a fan of the name but we are stuck with it.

The choice of using a Lua script instead of a C++ plugin was because a plugin requires getting code approved and added to DFHack. This is possible but much more difficult. It also means I don't have control over when newer versions are released since it's an open source project. Using Lua also allows one script to be used over many versions until major game changes occur. The script is also forced to save as a text file which has the benefit of not caring about endianness. If you use a binary format you have to deal with that issue.

DFHack also supports Ruby and can be used to help mod, fix bugs, make quality of life changes, and so much more. I use it for the reveal function to expose hidden tiles for testing and to get access to the map data.

Opal Prospect

This is the 3D visualizer that allows a player to save a snapshot of their map and view it in 3D later. It uses DFHack and a Lua script to extract the world information and save it in a run length encoded string. This is used by the visualizer to construct the world in 3D. I'll give a quick rundown on how the world space is defined and how the grid works. These are the important bits to understand if you want to delve deeper.

It uses a left handed coordinate system meaning that right is to the positive x, positive y is up, and positive z is away from the screen. With no rotation the camera faces north and the coordinate system follows the previous sentence.

The grid starts at the bottom left backside. Back referring to the most negative z coordinate. If you are at origin and the grid is created in front of you, the start is the bottom left block you can instantly see. Below is a diagram looking from above to help understand how the indices work.

Simple 3x2x3 example. Left side is below the right side. Starts at 0 for the first layer, second layer starts at 9.

```
6 7 8       15 16 17
3 4 5       12 13 14
0 1 2       9 10 11
```

Camera here looking straight ahead

Opal's RLE strings are based on layers. Each single string stores either a material or shape. Layers are the width and length which are the x and z axis. These are layered starting from the bottom of the world and working their way up until it hits the end. The strings are read left to right and correspond with the indices starting at 0 and going positive as it's read. There will be more detail in the RLE encoding section but that will get you the basics of how the visualizer treats them.

That's a quick overview of the important parts of the visualizer. Next we'll cover voxels and then get into algorithms.

What are voxels and why use them

Voxels describe volumes in the same way that pixels describe areas. Vo from volume and xel from pixel is one way to think of the word. Most voxel systems use one standard shape such as a box that is placed in a grid. This allows all or most of the volume to be described in a tightly packed and well defined system. This lends arrays as a good data structure to store them. This is the most efficient data structure for contiguous items because you can save only the items and nothing else.

They are back to back in memory which is also very cache friendly.(Kapoulkine)

As already mentioned voxels main use is to allow for fully changeable worlds. You can find other systems that allow building placement and maybe some very limited terrain modification. There is always a limit. Voxels are mostly limited by hard drive space and memory capacity. Minecraft for example uses 1 cubic meter blocks that can be dug up and replaced or left empty. While it has a very blocky look it offers a large variety. And as Minecraft shows you are not limited to a single shape.

Dwarf fortress has many types of shapes from full tiles to floors, walls, buildings, workstations, mechanisms, doors, and more. This makes it more difficult to store due to the number of possiblities. Another big concern is that all of these shapes can be made of almost any material. Minecraft has each block as a unique type. In a Minecraft like system you can more easily encode all the possible blocks. Simply save the block id. For DF you need to save both the material and the shape. Or create an id for each possible combination. You'll need a lot of numbers.

As shown there are many ways to implement a voxel system. While I mostly describe how video games use them there are two other fields who use them as well. Geology and medical imaging. Next we'll look at optimizations for models used to display the world.

Model optimizations

https://github.com/swbengs/OpalProspect/blob/senior_sem_setup/Opal%20Prospect/DwarfFortress/NaturalTerrainModelBuilder.cpp

The most important part of this is understanding that during rasterization, the process of putting 3D data onto a 2D screen, a flat surface can be described as a single shape or many. In our case it will be lots of rectangles lined up in long stretches. Instead of defining 1 per tile, we can combine all similar materials in a row into a larger rectangle. This loses no visual quality. The only cost is that techniques such as light mapping are harder to do since you now have 1 rectangle where there was 2 or more.(0 FPS)

I implemented a simple version that works only horizontally. The other and better version is called greedy meshing. You'll hear a term of face merging which simply is merging flat surfaces, sometimes called faces.(Gedge)

The basic idea is to break the world up into 2D layers so that you only need 1 method to do the merging. I used an array passed into this method. The method that calls the merge method has two loops. The outer loop controls where the layer starts, and the inner loops grabs all tile indices that are passed to it. Then the merge method just has to handle finding runs of the same shape and material. Offset the merged tiles x, y, and z so the middle is in the middle of all the merged blocks.

After it's done you have a fraction of the space used. My first tests of it took ¼ of the space. That's a compression ratio of 75%. Best of all the user can't tell the difference unless they have a way to view the mesh's triangle components directly. I don't use fancy lighting techniques so I pay no cost to this. An important trade off is also the time it takes to process the world and merge it. If you transform it, said merging has to be redone. With a system like Minecraft and using chunks you can reduce the amount of the world that needs to be merged. Still an important note.

Greedy meshing would move diagonally and combine as large of an area at a time as possible. Then go through an array of the non merged blocks and repeat until the entire surface has been merged as much as possible(Fogleman).

Run length encoding

For both encoding and decoding view the code here
https://github.com/swbengs/OpalProspect/blob/senior_sem_setup/Opal%20Prospect/DwarfFortress/NaturalTerrainFileLoader.cpp

https://github.com/swbengs/OpalProspect/blob/senior_sem_setup/Opal%20Prospect/lua/opal.lua

The basic idea of RLE strings is simply that given something like an array of data, a contiguous set of information, we can write that out as a pattern of runs. Anytime two side by side items are the same thing we can write it as a number and letter combo instead. So let's say in a text file we see the same letter 3 times in a row like the letter a. We could instead write it as 3a and when we decompress it we would replace 3a with aaa. This can be used for text and binary. It can be used for files or for in memory data.(Pigeon)

If you saw the poster at MSUM's Spring 2020 student academic conference you might remember the following strings.

2bl1ba2bq
5w

The first one is a material RLE and the second is shape. It's forms a 5x1x1 voxel for this example. An important note is that to figure out its dimensions we need to have the x, y, and z defined. It could also be interpreted as a 1x5x1 or 1x1x5 if we don't specify the dimensions. This is done at the top of the file. We also need a table for the materials so we know what those are. For our example that won't matter. Shapes are already predefined with what letters are what.

One important note of my implementation is that some data can be lost. This only occurs for blocks that have not been exposed. My visualizer was designed to only let others see what has been exposed. This prevents things like the locations of different caverns from being exposed by accident. This does improve compression which is why I'll compare played on maps and maps that have their entire volume exposed via DFHack's reveal command.

As mentioned earlier I use two letters for materials because there are more than 256 possible materials. There are many variations of RLE. Static and dynamic/variable are the more popular versions. Static means the codes are a preset length and don't change. Variable length means they can change in length as needed. 256 is important because a single byte can only store that many different possibilities. Basic text is ASCII and stored as a single byte per character. This will also be important in Huffman encoding below.

And lastly you can encode as one, two, three and so on dimensional strings. I use a single dimension array that describes 3 dimensions. The string matches the same style except that each layer is a separate string to help in debugging encoding and decoding issues. It could achieve higher compression ratios if the whole world was one string but then it becomes difficult to find out where errors occur. So instead of a single 3 dimensional string, I have lots of 2 dimensional ones.

Huffman Encoding

https://github.com/swbengs/OpalProspect/blob/senior_sem_setup/Opal%20Prospect/Huffman.cpp

Huffman encoding works by taking character codes that are usually a whole byte, 8 bits, and using less to encode them. We'll only discuss ASCII. The easiest way to implement this is a binary tree. There's a unique path to get to each leaf that is a character code. More frequent characters get the smallest bit codes. Then you save these codes as binary and use bit operators to save and extract them.(Murray)

Once you have the hashmap it's easy to compress or decompress. Simply iterate over the entire set of data and when you find that character it goes into the hashmap and you get the bit code back out. Write that off to a new location that stores the final result and you are done. Just reverse the hashmap so the bitcode is first and the actual character is second and repeat the process to decode.

Zlib's deflate

Deflate runs off mainly LZ77 and Huffman. It's a very well known and commonly used algorithm that is open source. It gives good compression ratios while also maintaining good speed for compression and decompression. The article tied to this paragraph goes into excellent detail about the many factors to consider when picking and algorithm and also to explain their new one. (Collet, Turner)

It's so easy to use with the default settings this is all it takes after you download and build the Zlib library. Static or dynamic binding doesn't matter for our use. Here I using their simplified methods to compress. You can call deflate directly with many different options if you wish but that takes a lot more code.

https://github.com/swbengs/OpalProspect/blob/senior_sem_setup/Opal%20Prospect/Opal%20Prospect.cpp

```
void zlib_deflate(std::string filename)
{
    FILE* infile = nullptr;
    fopen_s(&infile, filename.c_str(), "rb");
    gzFile outfile = gzopen("zlib_comp.txt", "wb");

    char inbuffer[128];
    unsigned int num_read = 0;
    while ((num_read = fread(inbuffer, 1, sizeof(inbuffer), infile)) > 0)
    {
        gzwrite(outfile, inbuffer, num_read);
    }

    fclose(infile);
    gzclose(outfile);
}
```

This takes in a file and will create a new file called zlib_comp.txt that is the compressed version of whatever you gave it. There isn't much to go into since other articles can explain the code and math behind why deflate works so well.

Next up is the comparison of mix and matching the algorithms to see how small we can store the world in.

Comparisons

Unlike my poster, for these comparisons I will use the full map data. This is done with the “reveal hell” command that exposes the entire map by marking all hidden tiles as not hidden. To undo this you call “unreveal”. Next run the Lua script with opal “filename_to_save_to_here”. Rinse and repeat with different biome and different heights to get a better sample.

We'll be comparing the in memory size, RLE only file, and Zlib and Huffman run on the RLE only file.

Once at least a few maps are done we need to take the RLE encoded string files and put them into both Huffman and Deflate methods to see if they shrink even further. After running a few through and keeping track of the sizes you should get a table much like this.

size in bytes	size vs memory	compression ratio
16,588,800 in memory		
2,494,562 rle only	15.04%	84.96%
1,249,715 rle+huffman	7.53%	92.47%
757,362 rle+zlib	4.57%	95.43%

So the breakdown is that in memory it takes a minimum of 3 bytes per tile. 2 for material and 1 for shape. So with an average size of 192x150x192 and 3 bytes each you get 192*192*150*3 and should come up with that top number. Next is the RLE string file. No other compression is done on it yet. That cuts the size down to about 15% of normal on average. Next we take that and compress it with only huffman and get another 7.5% drop in total file size which is good. Finally we use zlib's deflate on the same file, the 2,494,562 byte RLE only one. That brings it down to under 5% of the in memory size. For reference the in memory is about 16 mega bytes in size. The deflate file ends up at about 3/4 of a mega byte.

Conclusion

As can be easily seen with how easy it is to use the default simple methods and deflate, you're better off just using the time tested algorithm. It's fun to try out new ones and implement them yourself but in the real world other people have done it better. But it's still good to see that even simple and naive versions of RLE and Huffman can compete with the best.

The next best part is that as even newer compression algorithms are created the results could get even better. It would be fun to try out the one listed in Yann Collet and Chip Turner's article which is supposed to have even higher compression ratios on data they've tested. And who knows what else you could come up with by mixing other types of compression. There's always the option to accept some loss of data if disk space is that important as well.

Bibliography

0 FPS. “Meshing in a Minecraft Game” 0 FPS.
https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/ (2/3/2020)

Arseny Kapoulkine. “Voxel terrain: storage" Zeux.
https://zeux.io/2017/03/27/voxel-terrain-storage/ (2/3/2020)

Dhanesh Budhrani. “How data compression works: exploring LZ77” Towards Data Science.
https://towardsdatascience.com/how-data-compression-works-exploring-lz77-3a2c2e06c097
(2/10/2020)

Jason Gedge. “Greedy Voxel Meshing - Jason Gedge” Gedge.
https://www.gedge.ca/dev/2014/08/17/greedy-voxel-meshing (2/3/2020)

Lakshmi K, Robert Theivadas J, Markkandan S. “Variable-to-Variable Run Length Encoding Technique for Testing Low Power VLSI Circuits” Journal of Electrical & Electronic Systems
https://www.omicsonline.org/open-access/variabletovariable-run-length-encoding-technique-for-testing-low-power-vlsi-circuits-2332-0796-1000300-108457.html (2/3/2020)

Michael Fogleman. “Voxel Rendering Techniques” Medium.
https://medium.com/@fogleman/voxel-rendering-techniques-fa8d869457ca (2/3/2020)

Murray, James D. , Van Ryper, William . “Chapter 9. Data Compression” File Format Info.
https://www.fileformat.info/mirror/egff/ch09_01.htm (2/10/2020)

Murray, James D. , Van Ryper, William . “CCITT (Huffman) Encoding” File Format Info.
https://www.fileformat.info/mirror/egff/ch09_05.htm (2/10/2020)

Steven Pigeon. “Compressing Voxel Worlds” Word Press.
https://hbfs.wordpress.com/2011/03/22/compressing-voxel-worlds/ (2/3/2020)

Steven Pigeon. “Ad Hoc Compression Methods: RLE” Word Press.
https://hbfs.wordpress.com/2009/04/14/ad-hoc-compression-methods-rle/ (2/3/2020)

The DFHack Team. “DFHack Plugins Lua API” DFHack.
https://dfhack.readthedocs.io/en/stable/docs/Plugins.html#lua-api (2/3/2020)

Yann Collet, Chip Turner. “Smaller and faster data compression with Zstandard” Facebook Engineer.
https://engineering.fb.com/core-data/smaller-and-faster-data-compression-with-zstandard/
(2/10/2020)