Creating a Region File System for a Voxel Game

Hey guys! I am really excited to be doing this dev blog! I thought I would kick it off with something I haven't seen online, a guide to making a file system for your voxel game! This system utilizes a region format in which chunks of voxels are grouped into region files. This file system is inspired by the Minecraft file system, so give this wiki page a read: http://minecraft.gamepedia.com/Region_file_format

All of my blogs in the future will be assuming at least a primitive knowledge of C++, so if anything confuses you don't hesitate to ask!

Section 1: Basic Concepts

Why region files?

Voxel worlds are typically organized into pages of data called chunks. In Seed Of Andromeda, I have chosen to use a chunk size of 32^3. In contrast, the minecraft chunk size is 16x16x256 [1].

When designing a saving and loading for your game, you need to be interested not only in the speed of your IO, but the resulting file size. Nobody is going to want to play your game if the save file takes up gigabytes upon gigabytes of disk space, and lags when saving data!

The naive method would be to save each chunk in its own file. We could save each file using the chunks coordinates to construct the file name. Something like s.X.Y.Z.dat could work. I went ahead and implemented this to see just how bad the performance was... It was about as bad as it can get. When saving 2475 chunks to disk, each in their own file, it took about 70 seconds to save them all without any compression. With run length encoding compression, it was brought down to about 30 seconds, which means on average each chunk took 12ms to save. That is unacceptable! Why is it so slow? It is slow because we have to open and close 2475 file handles, causing a lot of blocking as we wait for the OS to open the files.

So how can we speed this up? Since the obvious issue is that we are using too many files, we should try to pack many chunks into a single file. We could try to put all of the chunks in one file, but that would cause the file to get very large. As a file gets larger, some of the operations such as resizing a chunk will get much slower. Instead we will use a region file that contains chunks for a small region of the world. I have found that storing chunks in files of 16^3 = 4096 chunks works very well. Each one of these files is called a region file. With these region files, we can open one file handle to do operations on several chunks, thus eliminating the file handle bottleneck.

Getting the region file name

When we want to save a chunk, we need a string representing the filename of the region file. Since a region file will hold an area of 16x16x16 chunks, all of these chunks should map to the same file. Here is how it is done in Seed Of Andromeda:

X, Y, and Z are the coordinates of a chunk divided by 32.

```
string filename = "r." + to_string(X >> 4) + "." + to_string(Y >> 4) + "." +
to_string(Z >> 4) + ".soar";
```

so for instance if we have a chunk (2,0,3) it will have string r.0.0.0.soar. So will a chunk at (14,15,15). But a chunk at (17,15,15) will have string r.1.0.0.soar. >> is the right bit shift operator, and you could replace "X >> 4" with "X / 16" if you prefer. They are the same as long as X, Y, and Z are integers.

File Structure

In our region file, we must sequentially store 4096 chunks. The most compact way to store the chunks would be to store each chunk back to back with no empty space in between, however this poses a problem when chunks can have varying sizes. Lets say we have a large region file as illustrated by the table below.

Chunk #	0	1	2	3	4	5	...
Size (Bytes)	400	32	500	4092	16384	36	...

Offset (Bytes)	0	400	432	932	5028	21412	...

Observe that the offset of each chunk into the file is immediatly after the previous chunk. The problem with this structure is that any time we modify the size of a chunk, we must move ALL of the data after that chunk. For instance if we reduce the size of chunk 0 to 40, we must shift all of the data after chunk 0 to the left by 360 bytes. To do that we basically have to rewrite the file.

Can we prevent this? Yes! Instead of storing all chunk one after another, we store them using sector offsets. First we need a sector size. Minecraft uses 4096 bytes, so we will use that. Each chunk must be stored with an offset that is a multiple of our sector size. This means that there will be lots of unused space between our chunks, but the extra padding gives us some wiggle room when we update the chunks. Here is what our new region file looks like.

Chunk #	0	1	2	3	4	5	...
Size (Bytes)	400	32	500	4092	16384	36	...
Offset (Bytes)	0	4096	8192	12288	16384	32768	...

Now, if we reduce the size of chunk 0, we dont have to move any of the other chunks! The only time we would have to rewrite data after a chunk is if we resized the chunk beyond a sector boundary, which is a fairly infrequent event. You will notice that the file size is quite a bit bigger in this case, but it is well worth it.

Can it be better to not use sectors?

My friend Scott Hooper pointed out that for some voxel games, you might want to ignore sectors and just store the data as close as possible. Some games, such as single player only games, may not need to do much IO at all, for instance if chunks are rarely updated. This would allow you to have an optimally compact file size. In that case you could:

1) Always rewrite (never any wasted space) 2) Always rewrite when appending, but when shrinking a chunk keep track of a "wasted byte count." When that count goes above perhaps 25% of the file size, rewrite it 3) Treat the file as a heap. Add a "used" bit and "length" indicator to your lookup table, and scan the table for an empty block whenever you need to re-write a chunk. (also, merge adjacent unused spaces)

In games where disk writes will be commonplace, such as on a server with dozens of players, you will likely want to use sectors to maximize performance.

Lookup Table

In order to keep track of where each chunk exists in the file, we need to create a lookup table. This table will go at the beginning of the file, and will devote 4 bytes to each chunk. 3 bytes will store the sector offset, and 1 byte will store the size in sectors for a chunk. Since we have 4096 chunks per file, the table will be 4*4096 = 16384 bytes in length. If the offset and size are both zero, that indicates that the chunk is not in the file yet.

If you need to store timestamps of the last access time for each chunk, you could add 4 bytes per chunk to the lookup table, or do what minecraft does and store a separate table [2]. For simplicity, we will not be using a timestamp table. So our region file format is now as follows:

Byte	0-16383	16384...
Description	Lookup Table	Chunks and padding

The lookup table will ALWAYS be 16384 bytes long. To determine where a chunks entry exists in the lookup table, we can use the following formula:

X, Y, and Z are the coordinates of a chunk divided by 32, since our chunks have dimensions 32^3.

```
offset = 4 * ((X % 16) + (Z % 16) * 16 + (Y % 16) * 256);
```

Where % is the modulus operator. After locating our position in the table, we can then determine the location of the actual chunk data.

Chunk Data

When reading or writing your chunks to the region file, you will look in the lookup table for the offset into the file where the chunk data is located. If the offset in the table is zero, then that means the chunk isn't in the file. In this case you need to put the chunk data at the end of the file and update the table accordingly.

Once you know where your data should be located, you will seek to that point in the file (fseek). Usually the size of the data is not an exact multiple of your sector size, so there will be some extra padding bytes after the data. You don't want to accidentally treat those pad bytes as actual chunk data. To keep this from happening, at the beginning of each chunk's data section, there should be a 4 byte quantity specifying the exact size in bytes of the chunk data after compression.

So when you are writing your data to it's respective slot in the file, you first compress your data and then determine the resulting size. You write out the size bytes to the file, and then write the rest of the compressed data. When reading in the file, you read the size bytes to determine how much of the data to process. Then you read and uncompress the chunk data.

Section 2: File Operations

Ok so now that we know how the region format is supposed to work, how do we ensure that we are utilizing it in the best way? How can we keep the number of IO system calls to a minimum? In this section I will assume that you are using C style file operations using cstdio (FILE *). It is likely that for some of the following things, such as buffering, the various libraries that you may use will do it automatically.

1. Minimize file switching

We want to avoid opening and closing files as much as we can. A good way to do this is to group all of our chunks based on their respective region file. We could use a key-val store such as an std::map [3]. The key would be the region string, and the val would be a queue of chunks that need to loaded or saved. We could use two of these maps, one for saving and one for loading. Our maps could be declared this way in C++:

```
map<string, queue< Chunk* > > loadList;
```

When we want to save or load files, we iterate through each region file in the map, and then loop through its respective queue saving the chunks one at a time. We only need to open and close each region file once, since we can keep it open as we loop through the queue.

2. Cache the lookup table

As we loop through our queue of chunks, there is no need to read our table entry every time. Instead, we create a buffer.

```
char tablebuffer[16384];
```

When we first open the region file, we read the entire table into this buffer.

```
fread(tableBuffer, 1, 16384, file);
```

It is much faster to lookup the table entry for a chunk using the memory buffer than it is to look it up in the file, since memory speed is way faster than hard disk speed. We can even take it a step further by storing the table buffers for several region files at a time! This is known as caching; sacrificing memory for speed.

3. Buffered output

EDIT: No need to buffer your output so long as your IO functions handle buffering! fread and fwrite do this for you in c++!

4. Compress your data!

When you are saving and loading your chunks, overwhelmingly the bottleneck will be disk IO. For this reason, we want to minimize the amount of data that we read and write to the file. A great way to do this is to spend a little extra CPU time compressing the data. Not only will this speed up saving and loading, but it will greatly reduce the size of your chunk on disk!

Run Length Encoding (RLE)

The first thing that we should do is compress our chunk data with run length encoding. For simplicity let's assume that our chunk data consists of 32768 bytes representing block IDs. (Your actual data is probably much larger). Often in voxel games you will have runs of blocks that are all the same. For instance, high above the ground, you might have only air blocks. So if we had 10 air blocks in a row, we would have to write the following bytes to disk:

```
0000000000
```

With run length encoding, we instead store the number of blocks in the run followed by the block data itself. So the ten byte output above is reduced to just two bytes!

```
10 0
```

Depending on the complexity of our chunk data, if we apply RLE before we write to file, we should greatly reduce the final size!

zlib Compression

Run length encoding is great, but we can take it a step further! After using RLE on our data, we can use zlib to compress the data even further! Here is where you can grab zlib: http://www.zlib.net/

Using both of these techniques, I was able to compress 450 MB of data down to about 12.1 MB. With a sector size of 512 bytes, the file size went all the way down to 1.85 MB. That's a vast improvement!

Dont compress the lookup table. You should only compress the chunk data.

5. Multithreaded IO

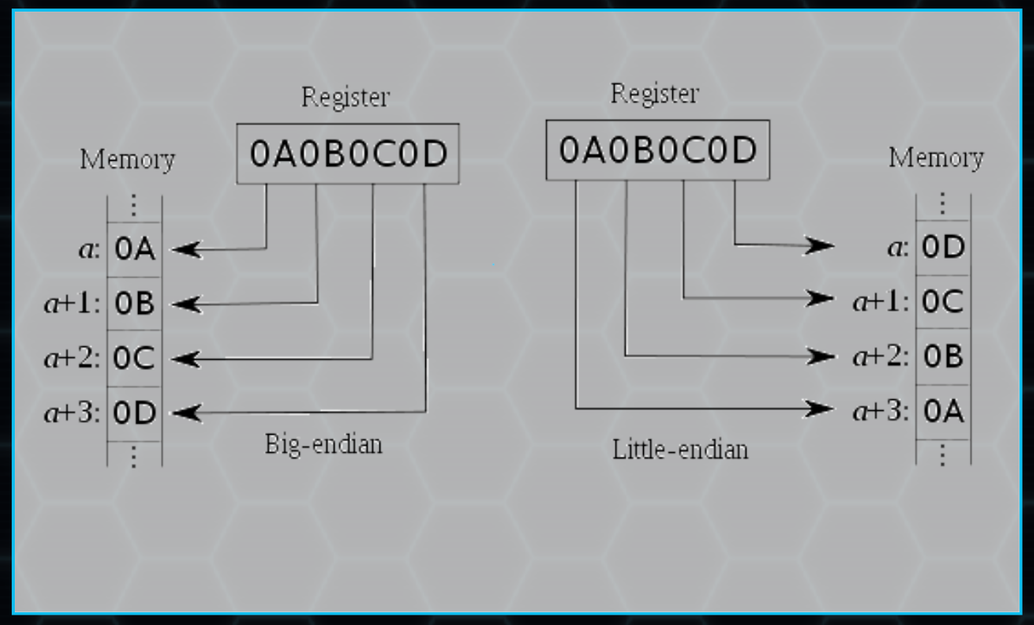
In order to get the best performance, we should run our file IO in a separate thread from the main thread. This will keep the main game from blocking when it makes read calls. You will simply run the thread in a loop that checks to see if the loadList or saveList are not empty. If they are empty, it will block on a conditional variable and wait for the main thread to signal it when expanding the load or save list. Be sure to protect your load and save lists with a mutex lock, as well as any other shared data! Here is some documentation on the C++ 11 thread library: http://en.cppreference.com/w/cpp/thread

Section 3: Additional Tidbits

1. Endianness

When computers store multi-byte quantities, such as ints, they will usually utilize big or little-endian. Give this article a read: http://en.wikipedia.org/wiki/Endianness

If we have a 1 byte quantity, such as 00101001, it will be the same no matter what the endianness of the machine. However, lets say we have a 32 bit integer. It will consist of 4 bytes. Though each byte is the same on every machine, the order of the bytes is different. Let's call our bytes A B C D, where A is the most significant byte, and D is the least significant. On a big endian machine, they will be ordered ABCD in memory. However, on a little endian machine, they will be DCBA. This image explains it well:



If we write our data to file as integers on a big-endian machine, then if we read it on another machine that is little-endian, the data will be incorrect. However there is a simple way to ensure that we have no endian problems. We always treat our data as big-endian, and we only store our data in byte buffers (unsigned char). For instance, if we have a buffer of 32 bit unsigned integers:

This is wrong:

```
unsigned int buffer[4096];
fwrite(buffer, 4, 4096, file);
```

Instead, we write like this!

```
unsigned char buffer[16384];
fwrite(buffer, 1, 16384, file);
```

Since endianness does not change the order of bits in a byte, if we write and store all our quantities as bytes, then we will have no issue! You might ask, "well now my buffer is bytes, how do I use it to represent integers?", well that is simple. If we want to extract a 32 bit unsigned integer from our buffer at a particular offset, we use the following function:

```
//Extracts a big-endian integer from an array of bytes
inline unsigned int ExtractInt(unsigned char *buffer, unsigned int offset)
{
unsigned int byte0, byte1, byte2, byte3;
byte0 = (unsigned int)buffer[offset] << 24; //most significant byte
byte1 = (unsigned int)buffer[offset+1] << 16;
byte2 = (unsigned int)buffer[offset+2] << 8;
byte3 = (unsigned int)buffer[offset+3]; //least significant byte
return byte0 | byte1 | byte2 | byte3; //combine all the bytes into a 4 byte quantity
}
```

If we want to set an int using 4 bytes, we simply reverse the above process.

If the preceding code confuses you, it utilizes two bitwise operations | (or), and << (left bit shift). Read up on bitwise operations here: http://en.wikipedia.org/wiki/Bitwise_operation

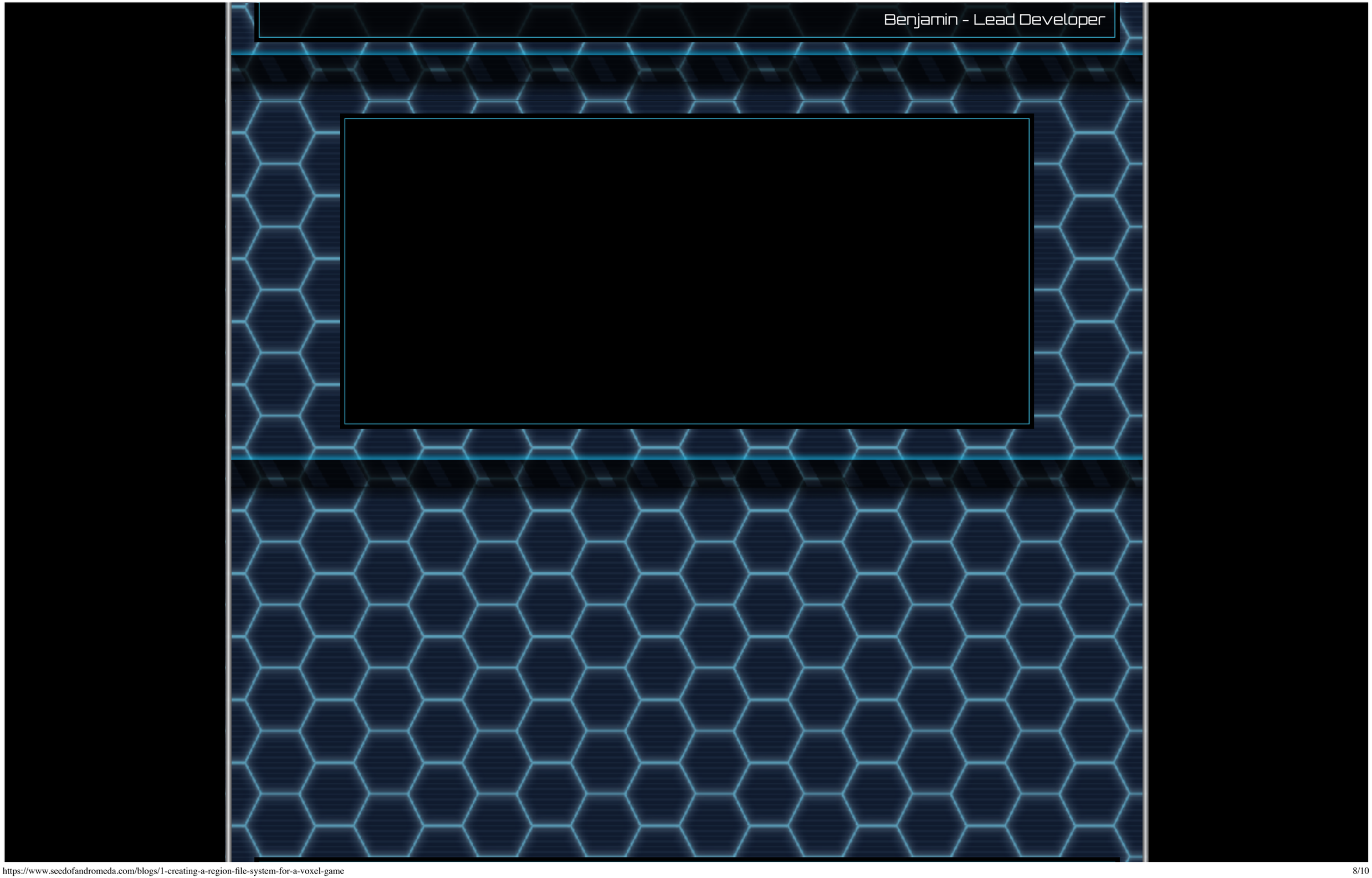
2. Memory-Mapping

Depending on your needs, memory mapping your region files might be benificial! It is not needed in Seed of Andromeda, but if you would like to read up on it, look here: http://en.wikipedia.org/wiki/Memory-mapped_file

End Result

After using all of these methods, Seed Of Andromeda can save 2475 chunks in 1-2 seconds, and can load those chunks in 0.6-1 seconds! That is a vast improvement over the 30 second save time when we were using a single file per chunk.

So thats it for my first blog post! It ended up taking quite a while. If you have any questions, comments, or suggestions regarding the content of this post, use the comments section below!


Benjamin - Lead Developer