by **Grimdoomer** » Apr 2015, Thu 23, 9:08 pm

You guys are only thinking about this from a binary file perspective, and not a memory perspective. Sure you can add a new bsp to the binary map file and everything will seem okay based on UAT and being able to view the bsp and etc. But you need to be thinking about this from a memory perspective.

Anyone remember the time old issue of using entity to build in a bsp that was larger than the existing bsp causing the map to black screen? Remember using Gemini to do bsp conversions because it was the mp map with the biggest bsp? There is a reason for all of this. Halo 2 only allows for one bsp to be loaded into memory at a time, and the bsp meta comes before all of the other tags in memory. So if you build in a bsp that is larger than the existing bsp in the map, you will end up overwriting the bsp meta in memory with other tags, or vice versa, causing your map to black screen.

Lets open up a clean ascension.map and anyone interested can follow along. The memory layout for ascension is as follows:

0x80061000 - Memory Header
This is the start of memory for tag data in RAM. This includes the tag index, the currently loaded bsp and lightmap tags, and all of the other tags listed in the tag index. This value is constant throughout the game. Peeking this address in memory will get you the 32 byte header of the tag index.

0x80061020 - Tag Index + 32 bytes
This address is the start of the tag index (ex IndexOffset + 32), it is also constant throughout the game. It's the first int in the tag index header. Navigate to 0x007C8301 in ascension.map and you will see this value. Note: Halo 2 is written in little endian so you will need to byteflip the data to get the same address I have posted here.

0x800AB000 - BSP/Lightmap meta (of the currently loaded bsp)
This address is the start of the bsp and lightmap meta block. This value can be found in the "Structure BSPs" tag_block in the scenario tag. The bsp and lightmap meta are in one block, so they both get loaded at the same time. Each bsp/lightmap block has a 16 byte header added to it which is as follows:

CODE: SELECT ALL
```
0 - BspBlockSize (size of the 16 byte header, bsp meta, and lightmap meta)
4 - BspMetaAddress (memory address of the bsp meta, ex BspOffset + 16)
8 - LightmapMetaAddress (memory address of the lightmap meta, ex BspOffset + 16 + BspMetaSize)
12 - BspGroupTag ("sbsp")
```

Halo 2 only allows for one bsp meta block to be loaded into memory at a time, and the bsp meta block comes before all other tags in memory. Anyone notice a problem with this? Remember trying to use entity to build in a bsp that was larger than the existing one and having your map freeze in game? Remember using Gemini to do bsp conversions because it's the mp map with the largest bsp?

Well here is the reason for both of those. Since the bsp meta block comes before the rest of the tags in memory, trying to replace it with a bsp that is larger will end up with you overwriting the bsp meta with another tags meta (starting with the matg tag) when the game tries to load the map. Loading the map into UAT and entity wont show you this, and you would never know! Your map would freeze and you would just chalk it up to shitty ol' entity.

So how do you fix this issue? Once the new bsp is in the map you have to recompute the memory address of all of the tags in the tag index. Then you recompute the secondary magic using the new memory address for the first tag in the tag index, the matg tag. Once you have your new magic value you can recompute EVERY, SINGLE, tag_block address in every non-bsp/lightmap tag in the tag index. Yup, every last single fucking one. Sorry guys who want to just shove data here and there and tweak a tag_block address, but that's not going to work with bsp! If you want to increase the size of the existing bsp or build in a larger bsp you best believe rebuilding the map is the easiest and, atm, the most reliable.

0x801AF000 - Meta Table (all other tags)
This address is really the memory address of the first tag in the tag index, the matg tag. It can be found in the tag index.

So now we can see that trying to increase the size of the bsp or building in a larger bsp will result in the map being broken at runtime, even though UAT or entity won't show you that. So how can you easily tell if your map is fucked up without having to load it in game? Simple, you use this tool I attached to the post. But first, lets do some algebra so you can understand what this tool actually computes, and how you can fail the arithmetic checks and still have a working map.

There are two equations we need to compute in order to tell if the index, bsp block, and meta table are all positioned in a way such that they do not overwrite each other. For the first we will using the memory header constant, 0x80061000, to check that the index does not overwrite the bsp block. For the second we will use the address of the first tag in the tag index, the matg tag, to check that the bsp does not overwrite any other tags.

We know that the Bsp comes after the index in memory, and our memory constant is the start of the index. So we know that:

CODE: SELECT ALL
```
MemoryConstant = BspBlockAddress - IndexSize - 32
```

If we subtract MemoryConstant from both sides, thus setting the equation equal to zero we get:

CODE: SELECT ALL
```
0 = BspBlockAddress - IndexSize - 32 - MemoryConstant
```

Viola! We can use this equation programmatically to check if the index overwrites the bsp meta block. If (BspBlockAddress - IndexSize - 32 - MemoryConstant) is not equal to zero then something is wrong.

Now to check if the bsp block overwrites other tags. We know that the address of the first tag in the meta table comes after the bsp block in memory. So we know that:

CODE: SELECT ALL
```
Tags[0].Address = BspBlockAddress + BspBlockSize
```

Just as we did with the first equation we will subtract Tags[0].Address from both sides, thus setting the equation equal to 0 we get:

CODE: SELECT ALL
```
0 = (BspBlockAddress + BspBlockSize) - Tags[0].Address
```

So now we have two equations we can use to check if memory will get corrupted when loading the map. But wait, we are only worried about **overwriting** memory, what happens if we build in a smaller bsp? Well this is where we can fail the arithmetic checks and still have a working map. Looking at the two equations:

CODE: SELECT ALL

```
0 = BspBlockAddress - IndexSize - 32 - MemoryConstant
0 = (BspBlockAddress + BspBlockSize) - Tags[0].Address
```

For the first equation if (BspBlockAddress - IndexSize - 32) is less than MemoryConstant will we get a negative result. Similarly with the second equation if (BspBlockAddress + BspBlockSize) is less than Tags[0].Address we will also get a negative result. But we wont overwrite anything in memory which is what we are concerned with! Hello Gemini bsp conversion solution!

So that is how the tags work in memory in a nutshell. At this point you should understand why we can't just make the bsp larger, why rebuilding using H2C&G for bsp conversions is way more reliable than using entity, and why we use to use Gemini to do bsp conversions.

Now that some of you understand this, it's time I repost an app I made that no one understood what the hell it did. It does everything I just explained, and runs the two equations we found for you. It will spit out two lines and tell you if the index check passed and if the bsp check passed. But keep in mind, that **the checks can fail when you have a negative result even though your map will still work!!** As long as the numbers it spits out are less than or equal to 0, your map should work.

ATTACHMENTS

MemoryChecker.rar
(4.09 KiB) Not downloaded yet

Don't snort the magic, we need it for the network.

Grimdoomer
Staff

Posts: 1806
Joined: Dec 2007, Sun 09, 2:09 pm