### Preface

I will explain how collision works and some techniques to use it in asm for both mario and custom objects. I will also go over how triangles are rendered and how to edit those with asm as well.

### Types of collision

There are two main types of collision. Static and dynamic. Static is basically your level geometry. It is loaded at the beginning on level load and never changes. Dynamic collision is usually object collision. Its collision that gets added to the pool of collision triangles every frame with the function 0x803839CC. As the name suggests, static is normally never changed and dynamic is changed all the time (usually by changing object position/rotation); we can use this to our advantage.

### Collision structure

Collision for both types is stored the same way in the ROM. You can find a document explaining the process here:

[Collision data explained](https://hack64.net/wiki/doku.php?id=sm64:collision_data)

The important thing to note here is the format: xx xx yy yy zz zz. Each position is a signed half and each vertex is six bytes. Triangles are 6 or 8 bytes depending on the collision type: [11 11] [22 22] [33 33] [bb bb]. 1 is vert one etc. Note that each triangle has its own parameter. This is something not realized in the current importer where the entire col type gets assigned the same bytes. You could theoretically have a winding path with each triangle having a unique wind direction, but there is no support for such a thing now. The same format is used for static and dynamic collision.

Collision is processed on loading and each triangle has its own struct in RAM. The struct can be found on the wiki “structs” page I linked earlier for the object struct tutorial. Another useful piece of info may be the plane equation:

Ax + By + Cz + D = 0

Here (a,b,c) are the plane normals, (x,y,z) are positions on the plane and D is the distance from the origin. Note that the distance is just from any point on the plane. Im sure sm64 just uses the first vert or something but I do not know for sure. Just know that it's the same relative location for all triangles.

This equation is how all collision is calculated. Various checks from an objects position times the normals and “negdot” (D in this eq) will grant the logic checks required to tell where an object and triangle are in relation to each other. The normals, negdot and more are all located in the col tri struct and are what we will be using if we need to do logic with a tri.

What is also not written is that there is a higher struct called surface nodes used to navigate the list of collision triangles. Collision triangles in RAM have no order that is of use to us; we cannot use a formula to load a specific triangle directly from the col tri struct list and be guaranteed it is the tri we want. We can do that with the surface node struct. Rather than loop through 15K triangles, we can reduce that to a few hundred. This is how the game calculates collision and helps reduce lag. This is only useful if you want to parse collision by yourself; there are many in game functions that already do that so you probably won't need to do it yourself but it may be useful knowledge for later.

### Using collision

First let's talk about in the ROM. Most importing needs are satisfied with your importer of choice, both with object models and level models. As mentioned before, the only thing the importers lack is a per triangle special parameter byte and certain col types (rom manager probably has them all if you manually type it in). There is no real elegant solution to adding unique special parameters per triangle other than manually typing it in a hex editor or just editing it inside the col tri struct during runtime. You can use several different materials and give them all different parameters but that is very tedious. Hopefully someone will make a method to apply a scalar field to collision triangles post import. There is not much else to do in the ROM that is not already standard for hacking.

Now during runtime, the easiest to access and probably most important collision triangles are the ones mario is on. You can access these from mario’s struct. He has a pointer to the wall, ceiling and floor that he is interacting with on any specific frame. A null pointer of course would mean no interaction. An easy way to hack a “custom” collision type in is to just execute code when mario is over a certain col type.

Ex. 1 conveyor belt collision

* LUI T0, 0X8034
* LW T1, 0XB1D8 (T0)//FLOOR PTR
* LH T2, 0X00 (T1) //col type
* ORI T9, R0, 0XD
* BNE T9, T2, END
* LWC1 F0, 0XB1AC (T0) //X POS
* ORI TI, R0, 0X10
* MTC1 T1, F2
* CVT.S.W F2, F2
* ADD.S F2, F2, F0
* SWC1 F2, 0XB1AC (T0)
* END:
* JR RA
* NOP

Nothing new here: we load 0x68 from the mario struct, compare it and if it is our desired value we increase mario’s x position every frame. You can do similar things for wall collisions and ceiling collisions. Keep in mind that the pointers here will be active even if mario is not standing. You can get floor/wall/ceiling values even in air so be sure to check. You can also get null pointers in wall or ceiling pointers, but not a floor null pointer. A floor null pointer causes an oob death and sometimes even crashes the game.

Ex. 2 changing col type

* LUI T0, 0X8034
* LH T3, 0XAFA0 (T0)
* ANDI T3, T3, 0X20 //L
* BEQ T3, R0, END
* LW T1, 0XB1D8 (T0)
* ORI T9, R0, 0X1
* SH T9, 0X0 (T1)
* END:
* JR RA
* NOP

This code will turn the collision under mario to lava when you press L. The collision will stay as lava until you reload the level as well. Be aware that you can edit any of the struct values like this. A good one to edit is the vertex positions and normals. This will fuck up your game real good, but if done correctly can allow for some dynamic collision. This is only useful if for some reason you want to edit collision that is not in an object so I won’t go over it. I do not recommend doing it unless you have a very specific idea in mind.

Unfortunately I do not have any great method for finding collision triangles in an arbitrary location. I have added the few functions I have tested myself to the function list online but there are several more I have not tested. What you'll have to do is look at the various resources online and test them out yourself. This will be the case for almost everything as only a small percentage of functions have proper documentation. If you can understand everything up to this point you should be fine.

### Visuals

Each frame all the triangles are passed to a buffer with textures applied and with various perspective corrections. Then from the buffer, a different component of the N64, the RCP, (equiv to a graphics chip) does the calculations to rasterize them. The RCP ***cannot*** be programmed. I repeat, the RCP ***cannot*** be programmed. This means once the information is put on the buffer it is out of our hands (outside of frame buffer editing but many emus/plugins don't support this). How an N64 game puts the triangles and textures to the buffer is different for each game, but in general, graphics microcode is the standard method.

Microcode is just another way of saying that we have an instruction set specific to controlling the graphics which is different from general asm. This is basically a layer between asm and direct hardware control. Super Mario 64 uses Fast3D, one of the first graphics microcodes developed for the N64. You can use different microcodes and there are people who have successfully converted over but it will be some time before that is publicly ready for people to add custom levels to (early-mid 2020 maybe). You also shouldn't worry too much about this asm or Fast3D knowledge being obsolete because even as the community moves to newer technology, this will be a strong foundation for how the game operates and will most likely be compatible with future hacking tools. Also newer microcodes are just extensions of Fast3D so if you understand this you will understand the newer microcodes.

Besides microcode, there is also the data microcode has to process. This is basically vertices and textures. Vertices are 0x10 bytes each and have a format like this:

XX XX YY YY ZZ ZZ 00 00 UU UU VV VV RR GG BB AA

* (x,y,z) are for position
* (u,v) Vert and horiz texture coords
* (r,g,b,a) are either vertex colors or normals used for shading

What's important to note is that all coordinates are signed halfs. This is important when you want to increment them for a texture scroll or something else. Also note that you can have normal shading *or* vertex colors not both.

Textures are stored differently depending on the format. You have rgba16/32 which simply gives a wide range of colors at the expense of taking more space per pixel. CI textures which use a color palette to save space. And IA/I which are intensity textures that are basically grayscale and grayscale with alpha. ROM manager should support all of these so importing is as simple as selecting the options. For CI textures make sure you are in indexed color mode (in img editor) and for IA make sure you are in grayscale. You can use a 64x64 CI4 texture and a 128x64 IA4 texture. The number after them refers to the resolution of colors. 4 means 4 bits of resolution aka 16 colors. I am not sure how ROM manager imports these if you go over the number of colors you define so be careful. These formats are great to combine with vertex colors to get a similar color but much higher resolution or for general textures that do not use many colors.

### Fast3D (Display Lists)

I will go over some of the important cmds in Fast3D (Display Lists) that are also easy to understand. Display lists or DL are the commonly referred to name for Fast3D microcodes so I will refer to it with that name. Here is a list of cmds for reference.

[Fast3D display lists](https://hack64.net/wiki/doku.php?id=super_mario_64:fast3d_display_list_commands)

DLs need to pass all sorts of instructions to the RDP and are generally hard to completely understand even for experts. As someone just learning you shouldn't worry about high level things and just start with the basics. The best cmd to focus on in the beginning is the FD (set img) cmd:

FD [xx] 00 00 [bb bb bb bb]

Here xx is the texture type and b’s are the segmented address of your texture. Segmented referring to a bank + offset or just the bank address. You can change the texture on all the triangles rendered following this cmd by changing this single address. For example if you wanted to change the M on mario's head to a HUD star you could change the 0xFD cmd to this:

0xFD 00 00 00 02 00 48 00

Here our star icon’s texture address (0x4800 in seg 0x2) gets put in the 0xFD cmd changing the referenced texture. This alone isn't enough to change it completely though. This is because graphics are complex and there are a lot of parameters we have to pass other than just texture address so that everything displays properly. We also need to change texture tile size, texture memory loaded, blend settings, and texture tile properties. Most of the time you will not have to change all of these but be aware that there is a lot going on behind the scenes that can easily stop working if you introduce something new. It may not always crash but it will probably not display correctly unless you change all the associated cmds. It will come down to experimentation and reading documentation.

Some other cmds you should worry about while you are just learning is the 0x06, 0xF2 and 0xFB cmds. The 0x06 cmd is a jump cmd while the other two control texture coordinates, environment color respectively. Texture coordinates just refer to how the texture gets tiled. Usually it is defaulted to 0 and 32. Meaning that the texture starts at 0 and ends at 32. You can however go to 16 to 48 meaning all your textures will be offset by half. This will be the goto way to scroll a large amount of textures with little lag.

0xFB is used to color triangles. 0xFB is used usually over a grey or white texture. It takes an RGBA parameter as their argument so editing this during runtime will allow for cool effects. For example you can darken flames by changing the 0xFB cmd or make flames loop through a set of colors.

0x06 cmds will be very important for making variants of models with different textures or 0xFB cmds. If you wanted to make a green flame, you could copy/paste the entire thing and put new cmds or you could do it in 3 lines with an 0x06 cmd. It will also be instrumental in making real time DL generation/manipulation.

### Conclusion

This is not a simple topic and there is a lot of information to cover. I will go over examples in the next tutorial as this was just meant as a primer for the topic. Read the references I linked and you can probably do some cool things with just this little info by itself. Next time will be mostly focused on display list manipulation during runtime.