Metal Force - Hitbox Script
RAM Search, Lua Scripting and Trace Log Tutorial

Scepheo

February 9, 2015

Contents

1 Introduction **2**
1.1 Tools 2
1.2 Required Skills 2

2 RAM Search - Player Data **3**
2.1 Health 3
2.2 X Position 4
2.3 Y Position 5
2.4 Camera Position 6

3 Scripting - Drawing on the Player **7**
3.1 Hello, world! 7
3.2 Drawing Health 8
3.3 Positioning - X 9
3.4 Positioning - Y 10

4 RAM Search - Enemy Data **12**
4.1 Enemy Health 12
4.2 Enemy Position 12
4.3 The Enemy Structure 13

5 Lua Scripting - Drawing Enemy Health **15**

6 Trace Log - Finding the Hit Box **16**

7 Lua Scripting - Drawing the Hit Box **16**

8 Conclusion **16**

1 Introduction

This will be a tutorial about finding memory addresses, writing Lua scripts and using the trace logger to debug game logic. I will explain these concepts by guiding you through the steps of making a Lua script to display enemy hitboxes, for the game Metal Force on the NES.

1.1 Tools

There will be two tools we'll be using for this tutorial, BizHawk and Notepad++. Although Notepad++ isn't necessary (any old text editor will do), it's what I use, and it has very handy syntax highlighting. This tutorial was written with BizHawk !!UNRELEASED!! in mind, but should apply to later versions too, although the names and exact locations of functions and menu items might differ. You can download BizHawk !!UNRELEASED!! from `!!NOWHERE!!`. Notepad++ is available from `http://notepad-plus-plus.org/`.

Note that there have been some fixes made to BizHawk !!UNRELEASED!! relating to RAM Search. As such, while most of the tutorial will work for earlier versions just fine, there are some cases where searches will have incorrect results. For this reason I do not recommend using this tutorial with version of BizHawk older than !!UNRELEASED!!.

1.2 Required Skills

For this tutorial, I will assume that you are familiar with the general workings of BizHawk, and know how to use frame advance and save states. I will also assume you have at least a vague familiarity with programming, and that in-depth explanation of the Lua syntax is not required, allowing me to focus on the logic and reasons behind the scripting, rather than the details of the language.

You will also need to understand the hexadecimal notation system. Although we won't be doing any advanced maths with it, it is the de-facto system for talking about low-level programs, which is what we'll be doing in this tutorial extensively.

2 RAM Search - Player Data

The first step will be to locate some standard memory addresses that will be useful to us. We're going to start with the player's health and position, and the camera position. Throughout these steps I will be making some assumptions about the data we're looking for, when I do, I will try my best to explain the rationale behind it.

2.1 Health

To get started, we're going into the first level. Near the top of the screen will be our health bar. As you can see, we start the game with 16 full bars of health. We're going to assume that these bars match our actual health. After all, there's no possibility to get more than 16 health, or to lose only part of a bar. As such, the first step will be to eliminate all addresses that are not 16.

To do this, pause the game (with the emulator, not just by pressing start) at the start of the first level, making sure you have full health. Next, open the RAM Search tool (Tools » RAM Search). First, we want to find all addresses that are equal to 16. So, at the top right, under Compare To/By, we select Specific Value and enter 16. As we want to find all values equal to 16, we leave the Comparison Operator set to Equal To. Next, hit the Search button.

Although the list has gotten a lot shorter, there are still a few addresses left. To narrow down our search, we're going to take a hit. At the start of the first level is a convenient turret, which we're going to allow to shoot us. So, unpause the game (or use frame advance), take a hit and pause again. If everything went right, we should just have taken 3 damage, putting our health at 13.

Now we're going to look for addresses equal to 13. So, in the box for Compare To/By » Specific Value, enter 13 and click the Search button again. At this point, you should have only a single address left, namely 031C. If not, try getting hit again and searching for that value, you should be able to do that by yourself now.

Double click the address, right click it and select Add to Ram Watch, or highlight it and press Ctrl+R. A new window should open called Ram Watch. You can also open this tool directly from Tools » RAM Watch. Here, you can double-click the address (Or Right click » Edit) to open an editing window. Give the address a useful name under Notes (say, "Health") and press OK. To be able to use this address in the future, use Files » Save to save your RAM Watch list to disk.

2.2 X Position

Our next step will be finding the player location. Again, we will start by making a few assumptions. Up until now we haven't touched the Size and Display settings in the bottom right. However, a single byte, as used for health, can only hold 256 possible values. As the level is a lot wider than 256 pixels, the x position of the player cannot be stored in just a byte. As such, select 2 Byte as our size, and hit New in the tool bar to restart our search.

Contrary to before, with health, we don't know the precise value for our x position. What we do know, however, is that we can increase and decrease it at will. So, get to a position where you can freely move left and right and we can start our search.

The first step is to get rid of all values that change without you moving. To do this, we start of by pressing the little arrow next to New (the tool tip should read "Copy Value to Previous"). What this does is remember the current value as the previous value, so we can compare to it later. Next, advance a few frames without moving at all. We want to compare to our previous value, so set Compare To/By to Previous Value. Comparison Operator should be set to Equal To (after all, it should still be the same as when we clicked the arrow) and you can do a search.

A few addresses will vanish, but not nearly enough to get anywhere. Most 2D games use a coordinate system with (0, 0) in the top-left, with x increasing as you go right and y increasing as you go down. Assuming Metal Force is the same, we move a little to the right. We still want to compare to the previous value, but this time we want to find values that are larger, so select Greater Than as our Comparison Operator, and press Search.

As you will see, we'll already have a lot less address. Move a bit more to the right and perform another search to reduce our list even further. We can also move to the left and use Less Than to filter addresses that decrease. Repeat these steps a few times until you have only one address left. (If you have about 10 left, move far enough to make the camera go along - these are positions relative to the camera.) If everything went right, only 002E should be left in your list.

Again, we will add our address to RAM Watch, name it (e.g. "X Position") and save our RAM Watch list again. We will also be doing something new: poking an address to test its correctness. To do so, right click your new-found x position address and select Poke. Enter a value somewhat higher or lower than what it was, hit Poke and close the window. Advance a frame (or let the game run) to see your changes take effect: the player moves to the new position you just entered.

2.3 Y Position

Having found the player's x position, it's time to look for the y position. There's two ways we can go about this: we can look for it in much the same way we did for the x position, or we can assume that the addresses will be close together, and work from there. We're going to start with the former, then show how the latter would work later. I recommend doing this at the start of stage 1, for reasons that will be explained later.

Again, we set the size to 2 Byte (don't forget to hit New). Next, we're going to make an assumption about the direction of the y axis: does it go up (higher on screen means higher y value) or down (higher on screen means lower y value). As I stated before, most games have (0, 0) in the top-left, so we're going with the latter.

First, copy the current value to previous (the little arrow next to New. Then, using frame advance jump and pause the game when you're a bit off the ground. Following our assumption, y should now be less than it was before. Again, we compare to previous value using Less Than. Let the character continue on his upward trajectory a bit more and repeat the process once or twice more. When he starts going down, make sure he passes the point where you searched last time and do another search using Greater Than.

The alternative method is to guess the location in memory. Still using RAM Search, set size to 2 Byte and start a new search. Rather than actually using the search function, scroll down to (or use Search » Go to Address...) the address for x: 002E. Moving up and down and jumping a bit while looking at the addresses surrounding it should reveal fairly quickly that the next address, 0030, goes up and down with your position.

Regardless of which method you used, you can always poke the address to see if it really corresponds to your y position. Assuming this is the case (i.e. you found 0030), add it to your RAM Watch again. Normally, we would be done for now, but there's one peculiar thing about your position: assuming you're at the start of level one, continue until you go down. You'll notice that your y position goes all the way up to 65535 (or slightly less) and then suddenly goes to zero and continues from there.

This is due to the way numbers are stored. 2 bytes are 8 bits, and as such allow for $2^16 = 65536$ possible values. For unsigned numbers, which is what we have been using until now, this is from 0 up to and including 65535. However, if you want negative numbers, only the last 15 bits are used for the number, with the first bit indicating whether it is positive or negative. They start at 0, going up to (but *not* including) $2^15 = 32768$. After that, the negative flag (bit) is set, and we continue *from the bottom*. This means the next number is -32768, and it continues up to (and including) -1.

This means that the unsigned number 65536 has the same bits as the signed number -1. As such, it seems that our position is signed (i.e. can be negative), and we need to adjust our RAM Watch accordingly. With your RAM Watch open, edit (double click or Right click ⟩ Edit) your X Position, set Display Type to Signed and press OK. Do the same for Y Position, and you should now see a smooth transition from negative to positive numbers.

2.4 Camera Position

Although we now have the position of the player in the level, this doesn't tell us where he is on-screen. As such, if we want to draw anything on top of him (which we want to), we'll need to be able to account for the camera's position.

I'm not going to describe the exact steps required to find these, as I very much recommend finding these yourself as practice. However, there is one pitfall I'm willing to help you with. If you don't want any hints, stop reading now. I mean it. All right then: the y coordinate for the camera goes the other way round. Up is higher values, and down is lower values.

Assuming you have gotten everything right, you should now have a RAM Watch file containing player health and position and camera position. I recommend trying to find some other values on your own now, like speed or lives, just to get more familiar with the process. If, however, you are keen to start scripting, feel free to move on to the next section.

3 Scripting - Drawing on the Player

We will now begin to write our script. As it stands, we already have all the addresses we need to draw on the player's location (player and camera position), and we have something to draw on top of him (his health). So, all that is left for us to do is write the script.

If you have trouble with Lua in general, or with the functions used to interact with the emulator, see `http://www.lua.org/manual/5.1/manual.html` or `http://tasvideos.org/Bizhawk/LuaFunctions.html`. I highly recommend visiting those pages whenever you do not understand something, after all, figuring out how do things is the leading principle behind both TASing and programming.

3.1 Hello, world!

To start off, we'll need to create the file that will become our script. To do this, go to Tools » Lua Console. In the newly opened window, we can select Script » New Script to create our script. Give it some name ("Player HP.lua", for example) and BizHawk will open the script. In what program it opens it depends on your system settings. If it does not open in the program you want, either change your settings or browse to the file and open it from there with the program of your choice.

You will see a file containing the following text:

```
while true do
    emu.frameadvance();
end
```

Basically, the "while true do ... end" part is an infinite loop, and `emu.frameadvance()` waits for the emulator to advance a frame. As such, any logic within this loop will be executed each frame. Right now, though, your script isn't activated yet (as indicated by the "(0 active, 0 pause)" message). To do this, select it by clicking on it and the check mark (or Script » Toggle). Nothing will happen, but that is because our script does not do anything yet.

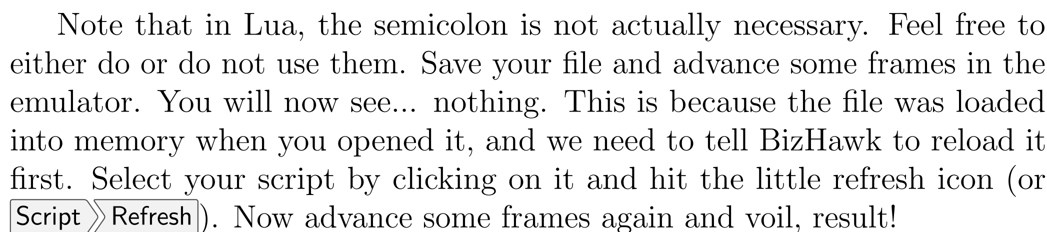
To start off, we're going to make the standard "Hello, world!" script. To draw text on-screen, BizHawk exposes a function called `gui.drawText(x, y, text)`. In general, any time you want to draw something you will be using the `gui` library. As such, make some space between the start of the loop and `emu.frameadvance()`, and put the function there.

For x and y (the position of the text), we will choose 150 and 100, respectively – 150 pixels from the left, 100 from the top. For the text, enter

"Hello, world!", *with* the quotes. These tell Lua that what is in between is one thing: a string. You should now have something like this:

```lua
while true do
    gui.drawText(150, 100, "Hello, world!");
    emu.frameadvance();
end
```

Note that in Lua, the semicolon is not actually necessary. Feel free to either do or do not use them. Save your file and advance some frames in the emulator. You will now see... nothing. This is because the file was loaded into memory when you opened it, and we need to tell BizHawk to reload it first. Select your script by clicking on it and hit the little refresh icon (or Script > Refresh). Now advance some frames again and voil, result!



Should your script at any point return an error (and it probably will), you will have to re-activate it, the same way you did before. As long as it's not selected, you can tell whether a script is active by the light blue background it is given.

3.2 Drawing Health

While we do have a working script now, it doesn't really do anything useful yet. So now we're going to make it draw our health. First, we'll need to get our actual health value from memory. To do this, we will be using the `memory` library. Or, to be more precise, `memory.read_u8(address)`. The reason we pick "u8" is because this reads an unsigned byte: the "u" stands for unsigned, the "8" indicates that it reads 8 bits, or a byte. And as you may recall from the previous section, our health is stored in memory as an unsigned byte.

To store our health we're going to need a variable. While Lua allows for variable declarations without any keywords (`x = 3;`), this creates a global variable that might interfere with any other scripts we are running. As such, we will declare that it is local to our script: `local hp = memory.read_u8(0x031C);`. Why the "0x" before our health address? This indicates that the following number is in hexadecimal representation, rather than a normal number. After all, "10" is a correct number in both hexadecimal and decimal, so we need to make it clear what we mean.

The next step is to draw our health on screen. So, take our function from the previous step and remove the "Hello, world!" string, replacing it with our variable for health (`hp`). All in all, our script should now look something like this:

```lua
while true do
    local hp = memory.read_u8(0x031C);
```

```lua
    gui.drawText(150, 100, hp);
    emu.frameadvance();
end
```

If you reload your script now and continue playing a bit, you will now see your health displayed where "Hello, world!" was before.

Right now, however, all that is shown is a number, which isn't very clear. As such, we would like the text "HP: " to be put in front of it. In Lua, passing a number (`hp`) where it expects a string (in `gui.drawText`) will cause the number to be automatically converted to a string. As such, we can just use the string concatenation operator `..` to add our text:

```lua
while true do
    local hp = memory.read_u8(0x031C);
    gui.drawText(150, 100, "HP: " .. hp);
    emu.frameadvance();
end
```

3.3 Positioning - X

The next step will be to position our text on the player. We're going to start of simple, and simply try to align our text with the player on the x axis. There is one more concept that will need explaining before this, which is endianness. Values that consist of multiple bytes can, of course, be stored in different orders. The endianness is this order. There are two options: a value is either little-endian (least significant byte first) or big-endian (most significant byte first). The values we have found for camera and player position were little-endian. This is the most common endianness, and therefore BizHawk's default search setting (which is why we did not need to change it).

We start of by reading our player's x position. For this we use the function `memory.read_s16_le(address)`. This time, the "s" means signed, the "16" means 16 bits (2 bytes) and the "le" stands for little-endian. As we have already found the addresses we need (which we can see in our RAM Watch), all that is left for us to do is store this value in a variable. So, before we draw our health, add the following line: `local player_x = memory.read_s16_le(0x2E);`. We can now use this value as the x position to draw our text. So in the function `gui.drawText`, replace the number 150 with `player_x`.

If you reload your script now, you will see that the health value now moves along with the player. It doesn't *quite* work correctly though: as long as the camera remains to the far right, everything is in order, but the moment the camera starts moving, the value slides off the screen. To fix this, we

will need to account for the camera position. So we read this into a variable too: `local camera_x = memory.read_s16_le(0x34);`. To correct for the camera position we need to subtract it from the player position. After all, the difference between these two values is the amount the player is "into" the screen. Adding a few blank lines to keep the code readable, this gives us the following script:

```
1  while true do
2      local hp = memory.read_u8(0x031C);
3
4      local player_x = memory.read_s16_le(0x2E);
5      local camera_x = memory.read_s16_le(0x34);
6
7      gui.drawText(player_x - camera_x, 100, "HP: " .. hp);
8
9      emu.frameadvance();
10 end
```

Saving and reloading our script now will yield the desired result: a health value that neatly moves along with the player – or at least along the x axis.

3.4 Positioning - Y

If we simply repeat the same process for the camera's and the player's y position, we will notice that things don't quite work right. Unless the player is at the very bottom of the level, the health value is drawn in completely the wrong location, if on the screen at all.

Studying the y position addresses reveals why: as we have asserted before, the player's y value goes down as he goes up on the screen. The camera's y value, however, does the exact opposite. So to correctly account for the camera's y position, we need to invert it: instead of subtracting it from the player's y position, we will add it. Storing the values from the calculations into variables to keep the lines from getting too long, this results in roughly the following script:

```
1  while true do
2      local hp = memory.read_u8(0x031C);
3
4      local player_x = memory.read_s16_le(0x2E);
5      local camera_x = memory.read_s16_le(0x34);
6
7      local player_y = memory.read_s16_le(0x30);
8      local camera_y = memory.read_s16_le(0x36);
```

```
9
10      local draw_x = player_x - camera_x;
11      local draw_y = player_y + camera_y;
12
13      gui.drawText(draw_x, draw_y, "HP: " .. hp);
14
15      emu.frameadvance();
16  end
```

4 RAM Search - Enemy Data

Of course, if we want to draw data relating to the enemies (i.e. their hitboxes), we're going to have to find their data in memory first. To do this, we're going to start with health and work from there. Again, the start of level one is a good place to do this: there is a single enemy (the turret) that takes multiple hits, allowing for an easy search.

4.1 Enemy Health

At the start of level one, there is a single enemy, the turret. Making sure to save a state before we do anything, our first step is figuring out how much health the enemy has. Attacking it until it dies reveals that it takes two hits to kill it: a reasonable assumption then being that it has, in fact, two health.

Loading our state, we start of by searching for any memory addresses containing the value 2. To simplify the process, we're going be using the automatic search function. So when you have set the Compare To/By and Operator correctly, click on the magnifying glass next to the Search button (or Options » Auto-Search). This will automatically perform a search at the end of each frame. The health value of the enemy should not change unless we attack it, so just stand around and dodge his projectiles for a bit while the automatic search function reduces the list to only about 15-20 remaining addresses.

Now, toggle the automatic search off again and attack the turret. After you've hit him, search for addresses that are 1. There should be only one address remaining: 03B4. Poking this address to higher values reveals that it is indeed correct, as the enemy will take more hits accordingly. Setting it to 0 will not work as expected, as the check for whether it should be destroyed only happens when your projectile hits the enemy.

4.2 Enemy Position

Now that we have found the health, we have an indication of where in memory the data for the enemy is stored. There is one more thing we need to know before we can effectively draw anything on an enemy: its position.

There's is one thing we can safely assume: the enemy's position will, like that of the player, be stored in signed 2-byte integers. So, open the RAM Search window, set it to 2 Byte signed and start a new search. If we stand slightly to the left of the turret, we know that our x position will be less than that of the enemy. Using this knowledge, we copy our own x position (which we can see in RAM Watch) into the box for Specific Value, and search for

values greater than it. We can do roughly the same thing by standing on the right, and searching for values less than our x position. Repeating this process a few times (making sure to adjust our comparison value every time we switch sides) we should quickly find... nothing.

You may have noticed that the addresses in the list increase by two each step. To a certain extent this makes sense, as each value is two bytes large. However, there is nothing preventing a value from starting at some address in between. To fix this issue, check the Settings > Check Mis-aligned option. Restarting your search now will also look for all other addresses, and repeating the steps from before will now result in finding the address 03AB.

Much like the player's y location, we have two options for the enemy's y position: either we look for it the same way as we did for x, or we simply assume it is next to x. Either way should quickly yield 03AD. Now that we have found the addresses we need for one enemy, the next step should be obvious: finding it for every enemy.

4.3 The Enemy Structure

Although we currently have the data for a single enemy, we'd like our script to display the health for all enemies on the screen. This means we're going to have to find two more things: a value that indicates whether the data for an enemy is used (i.e. the enemy is active) and the memory layout of *all* the enemies on screen.

We're going to start with the value that indicates enemy activity, which we'll call "status" from now on. Again, we're going to start by making some assumptions. This time we're going to assume that A) the status is stored in a byte (as it does not need to store much data) and B) the status will be 0 if the enemy isn't active, and some other value when it is. So, while the enemy is alive, repeatedly search for values greater than zero. Then, kill the enemy, and when it (and its explosions) are good and well gone, search for values equal to zero.

If your list is still fairly long, load a state from before the enemy is killed and repeat the process a bit, until you only have a few addresses left. You'll probably have found the enemies health address (after all, it fulfils the criteria we set) and one other value that is close by: 03A7. For now, we'll just hope that is the correct value.

The next step will be to figure out how each enemy is stored in memory. In general, this sort of data is stored in sequential structures: all the data belonging to one enemy, followed by the data for the next enemy and so on. This means that our next step will be to find the size of such a structure. To do this, look for the health values of other enemies. Go do this now. It's

good practice.

You'll find the address 03B4 a few more times, but you'll probably also find 03C4, 03D4 and maybe even 03E4 and 03F4. It's fairly easy to see the pattern: each address is 16 (10 in hexadecimal) further than the previous one. This means that each set of data is 16 bytes in size. Now that we have this data, we're ready to start writing our script.

5 Lua Scripting - Drawing Enemy Health

6 Trace Log - Finding the Hit Box

7 Lua Scripting - Drawing the Hit Box

8 Conclusion