Hello, and welcome to the *official* Cave Story ASM guide! Before we start, here are a few things to get you started.

First of all, for the most part in ASM, calculations and saved data are in ***HEXADECIMAL*** form. Explaining this is quite tedious, so instead of fully explaining it I'll just put some notes on it below.

- Numbers go 1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10
- Hexadecimal may be expressed as 0x(number) to avoid confusion between regular decimal and hex. For example, 0x40F350
- Every new digit is worth 16x the previous digit, e.g. 0x100 is 16 times as high as 0x10
- Sums are quite different from decimal, including such as 4 x 5 = 14.

That's all I'm going to talk about hex form. Sorry, but learning new number bases takes a while. If you still don't get the gist of it, you might want to learn about it more in
https://en.wikipedia.org/wiki/Hexadecimal.
Next thing you'll have to learn is the use of bits in hexadecimal. Hex number format is used because it runs faster. Why? Because in a computer, all numbers actually get expressed in 0s and 1s, using binary form.
If you don't know what that is stop immediately and read this.
https://en.wikipedia.org/wiki/**Binary**_number

This format, although fast for the computer to read, write and calculate, easily gets extremely tedious for a human to read, not to mention hard to write! So, the binary is converted into hexadecimal, which is equally easy for computers and humans to read. For example, the 'normal' number 100 is, in binary format, 1100100, and the computer splits it up into groups of powers of two, like this.
100 = 64 + 32 + 4
= 2 ^ 6 + 2 ^ 5 + 2 ^ 2

```
> 0110 0100
= 6    4
```

Why? Because in binary, four of each binary digit is equal to 1 hexadecimal digit, because the maximum 4 digit binary number is 1111, or F (largest 1 digit hexadecimal number).
So another example, 513
513 = 512 + 1
= 2 ^ 9 + 2 ^ 0

```
> 0010 0000 0001
= 2    0    1
```

There! Not so hard, is it?

Now, some things to do with actual assembly.

First of all, we're only going to use the CPU registers. We're not messing with memory or disk space yet. Registers are storage systems consisting of many latches/flip-flops that store bits of data.

So, I'll introduce you to your first and most basic register, AL.
AL is a byte sized register, and if you don't know what a byte is, you might want to check out https://en.wikipedia.org/wiki/Byte.

So a byte sized value is *eight* bits. This means it can store from 0000 0000 up to 1111 1111, in other words, from 0 to 0xFF, or in decimal form, up to 255.

So, the question is, how do we modify it?
Here are the most basic number commands.

MOV AL, (number) Sets AL to the target number (up to 0xFF).
ADD AL, (number) Adds target number to AL, if result is bigger than 255, only the last
8 bits are stored, so for an answer of C8 + 6E the answer is
0x136,
so it will store 0x36.
SUB AL, (number) Subtracts target number from AL. Likewise, if the answer is less
Than zero, the answer will be negative, so since in normal
unsigned
notation negative numbers cannot be stored, it will store the
answer
as 100 + (negative number). So for a sum of 43 - 7F, the answer is
-3C, which will be shown as 100 - 3C = C4.

So, how do we use it to do something?

I'll show you a few commands that do certain things when the answer of a previous sum is a certain state, but first you should download Ollydbg here, http://www.ollydbg.de/version2.html.
There is also a separate version of Ollydbg that I used Ollydbg to edit itself and fix some bugs in Ollydbg, and that's here
https://www.dropbox.com/s/21hwuaacrzg0zsd/Ollydbg_201_Patched.zip.
You can open CS now if you want, but don't change anything yet, because replacing necessary commands will break the game.

You'll see that the game is made up of 'addresses,' which are lines of code with a number on them. The number should be at the left side, and this tells you the position of the code is in terms of the executable.

You'll also see that some commands, particularly those that take large values, tend to take up a lot more numbers than others. This is due to the number that they have to work with being longer, so they take up more space. Right now you shouldn't worry about that yet.
There are two sum commands, INC and DEC, which we'll see later, which increase or decrease the target by 1. This is to save space, but don't worry about them yet either.
So, we've got our sums ready, so how are we going to use them?
First of all, I'm going to introduce a larger register.

AX

What is this? This is an extension of AL, storing up to 16 bits, or 2 bytes, or WORD size, which means up to 0xFFFF (65,535 in decimal).

The bytes/bits are organised like this.

```
/---------------AX-----------------\
/--AH-----------\/--------AL------\
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

As you can see, AX covers the whole thing, whereas AL only covers the right side. AH covers the left side, but we won't dive into that right now, because we can now use AX as a whole value!
Now, 0xFFFF is quite a large number, but not large enough for some precise measurements and calculations. So let's bring out the big guns and let loose the largest CPU register!

```
0000 0000 0000 0000
               \-----/ AL
          \-----/ AH
          \------------/ AX
\--------------------------/ EAX
```

So now you've got a 32 bit register, and when that whole number's filled, you're going to be able to store 2 ^ 32 - 1 as the maximum value; that results in FFFFFFFF in hex, 4,294,967,295 in dec.
This size is the typical value size used for 32 bit applications, which you can see why are called 32 bit.
One more thing is that changing the value of EAX changes AX and AL too, and vice versa.
So the following:

```
MOV AL, 50
MOV AX, 470
MOV EAX, 20000
```

^ This will set EAX to 20000, AX and AL to 0, because EAX covers all of the bits.
This example

```
MOV EAX, 20000
MOV AX, 470
MOV AL, 50
```

^This will set EAX to 20450, AX to 450 and AL to 50, because the smaller values are set after the big one.

So now, since we have such a large value, we can now do massive sums like the following:

```
MOV EAX, 5F8E1A87
SUB EAX, 68B3D1
ADD EAX, 74EE59
```

And if you'd written that in code (don't do that yet), in a blink of an eye your computer will calculate that for you to be 5F9A550F.
Also, note that AL is **byte sized**, AX is **word sized**, and EAX is **dword sized** (doubleword).

But how do the functions in CS work? You can't get a jumping Critter or a flying Bat through a few little sums?!

First of all, EAX is only one of the 8 main registers. Here is a list of all the CPU registers that are used normally;

- EAX
- EBX
- ECX
- EDX
- ESI
- EDI
- ESP   // Do not use as normal storage unless you know what you're doing.
- EBP   // Do not use as normal storage unless you know what you're doing.

There are others, but messing with them is almost a guaranteed crash, and is NOT recommended.
The reason why the last two are not allowed is because they control the thing called the 'stack,' which you'll learn about later.
The other registers work the same as EAX; they all are split into sections, e.g. EDX ⇒ DX ⇒ DL. However, keep in mind that ESI and EDI can only be split up into SI and DI and no further.

So now you have 6 different values to mess with! What's more, you can even add them together!

```
MOV EAX, 100
MOV EBX, 80
ADD EAX, EBX
```

This will cause it to add EBX to EAX (Since EBX is the second value of the sum), which makes the sum basically adds 80 to 100, getting 180. This 180 replaces the original EAX, like a normal ADD would. EBX **will stay** as 80.

Here’s another arithmetic sum, with examples.

IMUL Multiplies first number by second number, storing answer in the first. When multiplying a register by a given value, say the register twice.
So if you’re multiplying EDX by EDI, you’d do IMUL EDX, EDI.
If you’re multiplying ECX by 7, you’d do IMUL ECX, ECX, 7. The reason for this is that when multiplying registers by a given number (not another register), you also specify the destination of the answer. For IMUL A, B, C, it multiplies B and C, storing the answer in A. So IMUL EAX, EBX, 3 will multiply EBX by 3 and store in EAX.

Time to do stuff with the sums.
Remember the addresses we talked about? After each command, the computer reads and executes the next command directly below it, or, in other words, the next number after it with a command.
It turns out that you can jump across commands without having to execute the commands in between, and if you’re familiar with TSC, this command is similar to <EVE.

JMP (address)
This will cause the computer to start executing the command at the number specified. Here’s an example to prevent confusion.
Imagine you’re writing… Actually, open Cave Story in Ollydbg if you haven’t, and press CTRL+G.
A message box should appear, asking you for the address you want to go to. Type in 494000, and it should take you to an area with a bunch of random commands.
The reason I asked you to go to 494000 was because that part is full of useless stuff, so if we replace it (with more useless stuff) we won’t break the game.

So, let’s write this. Oh by the way, if you haven’t figured out by yourself already, to edit a command, select it by clicking and press [SPACE]. Or double click the actual command, it’s your choice. Now, click on the first command next to the number 494000, and open it. Deselect the option ‘Keep size,’ as that prevents you from doing most of the editing.
Type the following:

```
494000      MOV EAX, 400
494005      SUB EAX, 100
49400A      JMP 494011
49400C      ADD EAX, 200
494011      NOP
```

NOP (No OPeration) at the end is basically a command that does nothing, so don’t worry about that for now. What matters is what EAX will be when the computer gets to the NOP.
You’ll notice that Ollydbg autocorrected ‘JMP’ to ‘JMP SHORT.’ This is because JMP SHORT has a shorter range (rather obviously) than a normal JMP. However, JMP SHORT uses a lot less space (less than half the space) of JMP. Ollydbg is smart enough to know whether a jump is short enough to be a JMP SHORT, so leave it to Olly to decide.

So, what will EAX be? Let’s go through the steps taken.

1) EAX is set to 400.
2) 100 is subtracted from EAX, making it 300.
3) The computer jumps directly to 494011, *skipping the ADD EAX, 200*.

So, EAX will be 300 at the end, because the JMP skips the addition.
Furthermore, JMP can jump backwards, which can be used to create loops.
So far we’ve only been using JMP. But actually there are a variety of different jump commands, and those ones need to check for some properties first, and only jump if their prerequisite is met.
These conditional jumps are similar to the TSC command <FLJ, and here are a few:

JE Jumps to target only if the result of the last sum was 0, e.g. the following JE will be used:

```
MOV EDX, 100
SUB EDX, 100
JE (next)
```

But this one won’t be used:

```
MOV EDX, 101
SUB EDX, 100
JE (next)
```

JNE The opposite of JE. Jumps only if the result of the last sum was NOT 0. Again, this JNE will be used:

```
MOV EDX, 101
SUB EDX, 100
JNE (next)
```

But this one won’t:

```
MOV EDX, 101
SUB EDX, 101
JNE (next)
```

JA This jump will only be used if the result of the last sum is above 0. So this JA will run:

```
MOV EDX, 102
SUB EDX, 101
JA (next)
```

But not this one:

```
MOV EDX, 101
SUB EDX, 102
JA (next)
```

JB Jumps only if the result of the last sum was below zero. I shouldn’t need to give examples, since you should be able to tell how it works from the previous one.

JNB Jumps only if the result of the last sum was NOT below zero. At first glance, this seems to mean the same as JA, but it’s different in that it also jumps if the answer was 0.

JBE Jumps only if the result of the last sum was BELOW OR EQUAL TO 0.

Oh, by the way, all these jumps each have a SHORT version.
There are actually even more jumps, but those are currently useless for us, so we'll concentrate on the few ones we have.

Now that we have sums, jumps, and registers covered, now we'll move on to memory.
Every time a game such as CS is opened, it allocates itself some memory, and at the same time uses some of your computer's RAM. This is to allow the game to use more than just the 6 registers for variables.
Memory is used almost exactly like registers, except for some commands which are exclusive to either of them. Memory usually runs slower than registers due to the fact that most computers have a faster CPU than RAM. This means that it'll be better to work with multiple-step sums in registers rather than memory.
Unlike most higher-level programming languages, in assembly it is not actually possible to 'name' variables. They must be, and always must be referred to by either their address or a register containing their address inside square brackets. You can try labelling their addresses in a program like ollydbg, but that is only just a note to yourself, and cannot be referred to by the program.
To access memory, you put the address of the memory in square brackets, like this: [49E6CC].
[49E6CC] is a WORD sized variable, so

```
MOV WORD [49E6CC], 5
```

Will set it to 5.
So, here are a few important memory values in CS:

[49E6CC]	WORD sized, stores current health of Quote.
[49E6D0]	WORD sized, stores maximum health of Quote.
[49E654]	DWORD sized, stores Quote's X position in 1/8192 (dec) of a block. Yes! It's actually that precise!
[49E658]	DWORD sized, stores Quote's Y position in 1/8192 of a block.
[49E66C]	DWORD sized, stores Quote's X velocity that gets added to X position every frame.
[49E670]	DWORD sized, stores Quote's Y velocity, works like X velocity.
[49E6E8]	DWORD sized, stores Booster fuel in frames.

There are a lot more, but these are the ones that are most important. For a list of the majority of all useful variables in CS view:
https://cdn.discordapp.com/attachments/312733438153326593/312735614862622732/Assembly_Compendium.txt.

So if you do ADD WORD [49E6D0], 5, you'd increase the player's max health by 5.
If you did MOV DWORD [49E6E8], EAX, you'd set the booster fuel to EAX.
If you did ADD DWORD [49E654], 2000, you'd move the player to the right by 1 block.
So far, we've only been using **unsigned** numbers. This means that the bits in a number refer directly to the hexadecimal number they should be, which means we can only use natural numbers (positive integers) up to now.

Signed numbers are numbers that can be either positive or negative, but must remain integers. Most of the numbers used in CS are signed, and here's a basic summary of how signed numbers work.

- Any variable can either be signed or unsigned.
- Signed variables and unsigned variables use up the same amount of bytes.
- This results in the signed value having a lower maximum value.
- Instead of having the bits represent a positive number, any value with the first bit as a '1' will be a negative number. So in a byte sized value, 0110 0111 will be positive, but 1010 0110 will be negative.
- The way the negative numbers work is that they are actually the difference between the number and the number with all bits filled, so here's a more obvious representation using a nibble (half a byte, haha) of data.

Binary	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	A
1011	-5	B
1100	-4	C
1101	-3	D
1110	-2	E
1111	-1	F

Basically for signed numbers, half the numbers are negative, and since 1111 + 1 = 10000, it will overflow and the answer will be 0, because it won't be able to store any more digits! Since 1111 is -1, that also works since -1 + 1 = 0. However, adding 1 to the highest possible value gives the lowest possible value, in this case 1 + 7 = -8. This is one of the disadvantages of signed numbers, the other disadvantage being a reduced maximum number. For more info read https://en.wikipedia.org/wiki/Signed_number_representations.

So exactly why are most of CS' variables signed? Well firstly, although we already saw that making a value signed drastically reduces its maximum value, it allows for negative values. This means that in CS, we can have an object move in both directions without having to define another value for an opposite direction. Also, we were only using half a byte, which would only have a max of 0xF anyway, if we were to go and look at a DWORD (the majority of all variables in CS), the max would be 7FFFFFFF, and the min would be -80000000. This translates to 2,147,483,647 in dec and -2,147,483,648. These are more than enough to store almost any value you need. This is why usually signed notation is used instead.

Q: So how do you convert an unsigned number into a signed number?

A: You don't! The beauty about the way these numbers are placed is that it works both ways. Therefore, if you do something like MOV EAX, -80000000, it'd be the same thing as MOV EAX, 80000000, because depending on the type of operation you're doing, it still works as the same value!

Q: So how do we do operations in signed notation?
A: You don't! The operations in unsigned and signed are mostly all the same, because the numbers themselves are mostly the same! There are a few exceptions though, like division, which is slightly different for negative numbers. As for how to check the outcomes of such notation, here's how you do it.

Let's say we were checking if the player's current health was below 64 (0x40). We'd do

```
MOV AX, WORD [49E6CC]          (We copy the data to AX since we don't want to mess up
                                the player's health in the process)
SUB AX, 40                     (Subtracting to check if less than 0 or greater)
JB (Somewhere)                 (So we're going to jump to another command if the health
                                is below 0x40.
```

What if we're checking a potentially negative number? Like the player's X velocity (49E66C)? We're going to check if the player is moving at more than 2 blocks per second, **to the right side**.
2 bps translates to 16,384 times 1/8192, which means a change of 0x4000 (since 16,384 is 0x4000).
Now, since CS runs at 50 FPS, one frame would be 1/50 of a second, so we'd better divide it by 50 (dec).
0x4000 ÷ 0x32 (since 0x32 = dec 50)
= 147. Since 16,384 does not divide by 50 exactly, we're not going to get an exact answer. However, since it's per frame, and in 1/8192 of a block, it's going to be precise enough for our calculations.

So, we do:

```
MOV EAX, DWORD [49E66C]     (Saving the value so we don't modify it in the check)
SUB EAX, 147                (Subtracting 0x147 from it)
JG (Somewhere)              (If the player's X velocity is greater than 0x147, run some
                             other code).
```

WAIT A MINUTE!

Where did JG come from?

You'll notice that we've only worked with JE, JNE, JA, JB, JNB, JBE, and JMP.

Did I mention earlier that there'll be a lot more types of jumps?

Well here they are. These are for signed numbers.

JG	Jumps only if result of last sum is greater than 0, signed.
JL	Jumps only if result of last sum is less than 0, signed.
JGE	Jumps only if result of last sum is greater or equal to 0, signed.
JLE	Jumps only if result of last sum is less or equal to 0, signed.

There's actually even more jumps, but like before, to prevent confusion, we'll save them for later.

Now you know how to work with signed numbers.

By the way, JE and JNE still work with both signed and unsigned numbers, so don't worry about that. (Since the numbers are actually the same on both sides, one side is just treated as negative, so some values, like 0x-80000000 in DWORD size will be equal to 0x80000000).

Okay, we're about to take off into an actual CS function, but first I need to tell you about one more major part of ASM coding, and that is the **Stack**.

What is the stack?

Here's an example of what the stack might look like in a normal CS game.

```
19FF7C      0048CAE8        <ESP
19FF80      00000060
19FF84      74C338F4
19FF88      00200000
19FF8C      74C338D0
19FF90      78D9488F
19FF94      0019DFFC
19FF98      77685DE3        <EBP
```

You'll see that the 'Stack' is actually a list of DWORD memory variables. So what's so special about that?

It gives us a faster way of adding values onto the stack. So, here are a few commands that work on the stack.

PUSH Adds a value onto the top of the stack. In the case with the previous stack example, it'd add a new address, 19FF78 (since the stack goes downwards in value to prevent it growing into in-game values), and set it to what you PUSHed. You can PUSH numbers or variables, so PUSH 50 will add 50 to the stack. When PUSHing registers or variables, it copies it from the value, without affecting the value itself.

POP Takes the top value from the stack, saves it to target value or register. Also Removes value from stack, so if we did POP EDX, it would remove the '48CAE8' and set EDX to it. The top of the stack is now 19FF80.

You'll also see the values I marked, ESP and EBP. These are the 'forbidden' registers I talked about before, because they mark the top and bottom of the stack, respectively. When you add a number to the stack, you subtract 4 from ESP at the same time, and when removing a value, you add 4 to it. EBP remains unchanged, because it usually doesn't matter, unless you've PUSHed/POPped so many values that the stack pointers go outside the stack segment, in which case trying to add another value will trigger a buffer overrun and crash the game to prevent it messing with other stuff outside the game, e.g. other applications.
These 2 registers control the stack, which means changing them will affect the stack.

So why is the stack a thing?
One reason, as I've already said, is because removing and adding values to the stack is more efficient than using memory locations.
The other reason is that it allows the use of CALL and RETN.

Now, let me explain that a bit.
CALL is basically a JMP, but it PUSHes the value of the command after it before jumping.
Like this.

```
494000          CALL 40F350
494005          NOP
```

The CALL will PUSH 494005, and then jump to 40F350. What exactly does this do? It allows us to use RETN, which is like this.

```
40F371 (End of 40F350 command)   RETN
```

^ What does RETN do? It POPs the last number PUSHed, then JMPs to that address. So in other words, it looks a bit like this:

```
POP EAX
JMP EAX
```

Except it doesn't change EAX in the process.

So the example above will successfully return back to 494005, assuming that the stack is not messed up during the last function.
So, now you might wonder; if CALL and RETN basically just jump in and out of a command, why don't we just use JMPs?
The reason is (apart from RETN using less space), so that a command can be CALLed from multiple different other commands, and used by the other commands.

So why don't we have a designated value for CALLs and RETNs, which CALL would set and RETN would use to return back to after the CALL?
The reason for that is that a simple value for a CALL will only allow for a single CALL, and not multiple CALLs within CALLs.
For example, in CS, 4820AC is CALLed by 483749, which is CALLed by 4837E9, which is CALLed by 487701, which is CALLed by 48437E, which is CALLed by 48170A, which is CALLed by 40F350, which is CALLed by 4029B0, which is CALLed by 403740, which is CALLed by 410400, which is CALLed by 40F5F0, which is CALLed by 412420, which is CALLed by 481D27. So, that's another reason why we need a stack.

The third reason for the stack to exist is that we can store numbers easily with a PUSH, and restore them with POP.

Now, just one more thing before we get you started on a CS function! (Gosh, so many interruptions)
I'll just tell you what all of these commands do.

4820AC	Clearing/setting flags
483749	Initialise exception locking number
4837E9	Locking windows exception
487701	Setting memory for random numbers, including windows API exceptions
48437E	Getting Windows Application Exceptions, adding them to random numbers
48170A	Getting random number
40F350	Random Number Generator (for number in designated range)
4029B0	Check for whether a bullet can break a starblock
403740	Bullet tile collision algorithm
410400	Main Game Loop
40F5F0	Updating game window, maintaining game loop, etc
412420	Creating game window, starting game
481D27	Entry point of executable (startup)

Feel a little small now? Your computer runs all of these functions, and many more functions (Since this is only **one** branch of functions), **every frame** (50 times a second). And of course it also runs much more stuff every frame, including every other game you're playing on your computer, many of which use a lot more functions than CS, and of course, your operating system too! Next time you accuse your computer of being slow, consider the millions and billions of operations it's been doing!

That said, let's take a look at an actual CS function. We'll go through it command by command. Open Ollydbg again (Unless you've never closed it since the start), use CTRL+G to go to 0x416AC0. This is the function CALLed when <MOV is used in TSC. Here goes nothing…

```
416AC0      PUSH EBP
416AC1      MOV EBP, ESP
416AC…
```

Wait a moment, what? ESP and EBP were supposed to be the forbidden registers???!
This is because EBP is not actually used in assembly, it is just used as a buffer to prevent the stack overwriting variables from other areas. It is also used for the computer to track the stack easier through stack functions.

So PUSH EBP will save EBP onto the stack, which means that we can set EBP and ESP equal, since we've saved EBP. Let's get back to our function.

```
416AC0      PUSH EBP
416AC1      MOV EBP, ESP
416AC3      MOV EAX, DWORD PTR SS: [EBP+8]
```

Wait wait wait… What is [EBP+8]?
Now we get onto the fun stuff.
Okay, there's something that I should've told you earlier, and that is to run this function (<MOV), you need to PUSH the X value and Y value you want to move the player to, before CALLing 416AC0. These values are used by the <MOV command to set the player's position. The Y value is PUSHed first, then the X value.

So by setting EBP equal to ESP, we get the EBP we saved inside the address **at** EBP (Since when you PUSH a number, it is at the address at ESP, and now because EBP = ESP, we have the original EBP saved in the new (changed) EBP! What this means is that the value PUSHed by the CALL (Whatever command uses <MOV) is at the address before EBP, or, in other words, [EBP+4] (since the stack always goes lower in value for higher placed values).

This means that the number PUSHed directly **before** the CALL (The X value) will be in [EBP+8], so the formidable command MOV EAX, DWORD PTR SS: [EBP+8] is actually just copying the X value given into EAX! Which also means that the number before that (Y value) will be [EBP+C]!
Moving on...

Actually, one more thing. You should also have noticed this:
MOV EAX, DWORD PTR **SS**: [EBP+8]

SS is a segment representing the stack, so it tells the computer to read from the stack. There are multiple segments, but you don't need to worry about typing the correct one, since Olly autocorrects it for you.

```
416AC0    PUSH EBP                            Setting up stack
416AC1    MOV EBP, ESP
416AC3    MOV EAX, DWORD PTR SS: [EBP+8]      Getting X value
416AC6    MOV DWORD PTR DS: [49E654], EAX     Aha, here's a new segment - DS!
                                              also [49E654] is player X position.
416ACB    MOV ECX, DWORD PTR SS: [EBP+C]      Getting Y value
416ACE    MOV DWORD PTR DS: [49E658], ECX     Same concept as before
416AD4    MOV EDX,DWORD PTR DS: [49E654]      Getting (new) player X value
416ADA    MOV DWORD PTR DS: [49E65C], EDX     Setting camera X position
416AE0    MOV EAX,DWORD PTR DS: [49E658]      Getting (new) player Y value
416AE5    MOV DWORD PTR DS: [49E660], EAX     Setting camera Y position
416AEA    MOV DWORD PTR DS: [49E664], 0       Setting X camera movement to 0
416AF4    MOV DWORD PTR DS: [49E668], 0       Setting Y camera movement to 0
416AFE    MOV DWORD PTR DS: [49E66C], 0       Setting Quote X velocity to 0
416B08    MOV DWORD PTR DS: [49E670], 0       Setting Quote Y velocity to 0
416B12    MOVZX ECX, BYTE PTR DS: [49E638]    Oh noes…
```

What is MOVZX? And I thought you couldn't use ECX with BYTE sized variables??? (If you don't remember, you had to match the size, so you would have to set ECX to 0 and then use CL to store it)...
It turns out that MOVZX allows it to set ECX equivalent to a smaller sized number, with the rest of ECX becoming 0. There is a different function, MOVSX, which is like MOVZX, but it is signed, which means that negative numbers stay negative. Don't confuse them, and remember that MOVZX is only for unsigned, MOVSX for signed. So, since
(again referencing
https://cdn.discordapp.com/attachments/312733438153326593/312735614862622732/Assembly_Compendium.txt)
We have [49E638] is the player's flags, such as walking, hidden, looking at something, etc.

```
416B19    AND ECX, FFFFFFFE
```

Oh no, not another one?
Actually, I should've told you about this before. AND is one of the logic operations, and this particular one works like this:
Let's say you're ANDing 0x147 with 0x159.
The computer will convert them to:
0000 000**1** 0**1**00 011**1** and
0000 000**1** 0**1**01 100**1**.
AND only takes the bits if ***both*** values have it, so it'll go like this.
0000 000**1** 0**1**00 000**1**.

Since it only takes the bits in common as 1. In AND operation:
0 + 0 = 0
0 + 1 = 0
1 + 0 = 0
1 + 1 = 1
^ So 0x147 AND 0x159 = 141.
Before we move on let’s take a look at some of the other bitwise operations.

OR	Takes bits from either of the values
XOR	Takes bits from either of the values but **not** both.

So 0x147 OR 0x159 will work like this:
0000 000**1** 0**1**00 0**111**
0000 000**1** 0**1**0**1** **1**00**1**
0000 000**1** 0**1**0**1** **1111** = 0x15F.

0x147 XOR 0x159 would be:
0000 0001 0100 0**11**1
0000 0001 010**1** **1**001
0000 0000 0001 1110 = 0x18.

Understood? No? Take a further look at https://en.wikipedia.org/wiki/Logic_gate.

Since flag 0x1 in [49E638] is used to determine whether the player is looking/inspecting an item, ANDing ECX with 0xFFFFFFFE is basically getting rid of that 1. Here’s an example:

[49E638]	0000 0000 0000 0000 0000 0000 0**11**0 **11**01
AND	1111 1111 1111 1111 1111 1111 1**11**1 **11**10
=	0000 0000 0000 0000 0000 0000 0**11**0 **11**00

You can see why it gets rid of only the last bit of the variable, as only the last bit of 0xFFFFFFFE is 0 (It’s 1 less than everything being set). So what this command does is it stops the player from ‘inspecting’ an item.

Onwards we go.

416B1C	MOV BYTE PTR DS: [49E638], CL	Setting the current playerflags to the Modified one without ‘inspecting’
416B22	CALL 420FA0	Move Whimsical Star to Quote’s new position (more on this later)
416B27	POP EBP	Restore value of EBP
416B28	RETN	Return to CALLer of function

There! We’ve managed to understand **one** of the thousands of functions in CS.
So, let’s go on and learn about a completely different function, the ‘AI’ that controls the entities.

Before we get started, I am obliged to teach you the basics of entities and the like.
Cave Story's NPC data is stored in an **Array of Records**, located at 0x4A6220, each record 0xAC bytes long, and there can be up to 0x200 records. For more information on Arrays of Records, view https://en.wikipedia.org/wiki/Record_(computer_science)

Each entity has its own record, with a bunch of variables designated to it, and some of those include X and Y Positions, Entity Display Boxes, Entity Hitboxes, Entity Type and most importantly, the one value that determines whether the entity is alive or not. The entity's function will only be run when the entity's alive (obviously when you think about it).
So, instead of diving headfirst into NPC modification, with catastrophic consequences like in BLink's guide (sorry BLink), we're going to go take a look at the entity AI function first.

Go back to Olly (unless you've closed it again) and go to address 46FA00.

```
0046FA00      PUSH EBP                                      We've been through saving the
stack
0046FA01      MOV EBP, ESP                                  already.
0046FA03      SUB ESP, 8                                    Oh no….
```

Okay, we've got a problem. Don't panic! Think about it. Why are we subtracting 8 from ESP? If we've saved EBP onto the stack, and then edit EBP, then edit ESP as well, doesn't that mean we've messed up the stack entirely?

Not quite.
When we saved EBP, we modified EBP so that EBP was in [EBP]. We also made sure that ESP equalled EBP. This means that modifying ESP is okay, since the original ESP is saved as EBP! This means we've saved both values, and we can get them back although we've modified them! However, we can only modify one of them, because we need the other to save the values.
In this case, EBP is the value that is used to save both the original EBP and ESP.
This means we can modify ESP as we like, but not EBP.
Now, why SUB ESP, 8? Normally we'd use

```
PUSH 0
PUSH 0
```

^ And this would actually subtract 8 from ESP, so SUB ESP, 8 essentially does the same thing. However, the main reason we're doing this is that it frees up 2 DWORD sized variables from the stack for use in the function.

```
0046FA06      MOV DWORD PTR SS: [LOCAL.1], 0
```

Okay, so now what? What is [LOCAL.1]? That's not actually in the code. This is just Olly showing you that this is the first one of the 2 values we created from SUB ESP, 8. Press space or double click on the address and it should show [EBP-4]. Because the original ESP is in EBP, [EBP-4] is now the number PUSHed after EBP, and in this case it'll be the first of our 'random' numbers.

```
0046FA0D      JMP SHORT 0046FA18
```

Okay, we're already jumping and skipping part of the code. You'll see later.

```
0046FA0F      / MOV EAX,DWORD PTR SS:[LOCAL.1]
0046FA12      | ADD EAX,1
0046FA15      | MOV DWORD PTR SS:[LOCAL.1],EAX
```

^ These 3 commands add 1 to our local variable, which had been set to 0.
If Pixel's compiler had been efficient, it would've done INC DWORD PTR SS: [LOCAL.1].
These 3 commands are skipped first, and you'll see why later.

```
0046FA18      | CMP DWORD PTR SS:[LOCAL.1],200
```

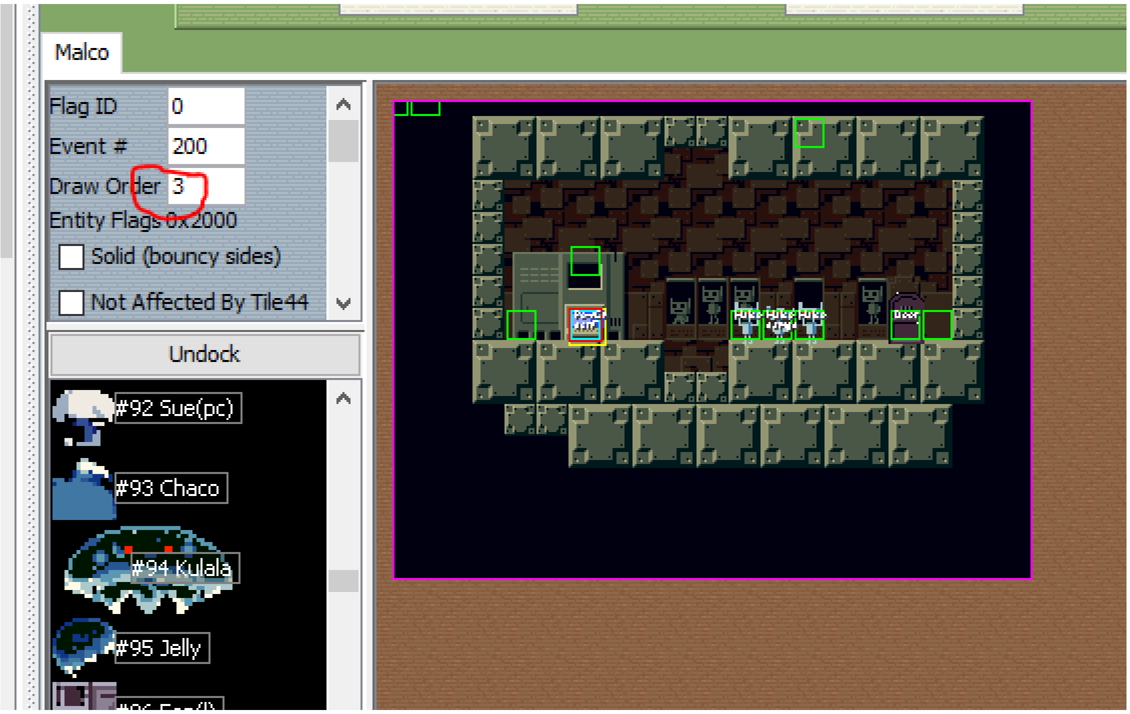
Okay… What's CMP? CMP is like SUB, but it doesn't change the value. Then what's the point? Although it doesn't change the value, it still counts as a sum, which means you can use conditional jumps on it! There is an 'equivalent' command for AND, and that is TEST.

```
0046FA1F      | JGE 0046FAA6                            See?
0046FA25      | MOV ECX,DWORD PTR SS:[LOCAL.1]          Getting local variable 1
0046FA28      | IMUL ECX,ECX,0AC                        Multiplying by AC
0046FA2E      | MOVZX EDX,BYTE PTR DS:[ECX+4A6220]
```

Okay, what are we doing here? We're actually getting the NPC's variables, even though we don't know which entity it is! This is why we have the first CMP, comparing the local variable to 0x200, is used - Because that's the entity limit! If you're on Ollydbg, you'll notice that the code is surrounded by a bracket. This indicates a **loop** inside the function, and this is obvious now, since we're counting all the entities one by one! If you don't know what a loop in coding is, see https://en.wikipedia.org/wiki/For_loop.
You can also make loops in TSC with <FLJ and <EVE, if you didn't know.
For future reference, loops in code will from now on be indicated by the /|\ brackets.

Now, so, what is [ECX+4A6220]?

We already know that it is counting up, from the adding of 1 to the local variable at the start.
So we know that [LOCAL.1] is a variable from 0 to 1FF. (Since if it is 200 or over then it jumps away with JGE). This means that [LOCAL.1] is actually the entity number of the NPC we're checking ATM.



Flag ID 0
Event # 200
Draw Order 3
Entity Flags 0x2000
□ Solid (bouncy sides)
□ Not Affected By Tile44

^ The entity number is the Draw Order of the NPC we're checking. This is the only way we can be sure of checking all the entities. It's also the way they're ordered, how convenient!
So, we've got IMUL ECX,ECX,0AC, multiplying the draw order by AC. This is due to the same variable in each entity being 0xAC bytes apart, since each entity has 0x2B DWORD variables.
This number is then added to 4A6220, because that is where the numbers start to count from.
I'll now give you what each of them mean. Let's assume we're looking at an entity with draw order 3 (Malco from the picture).
We multiply 3 by AC, add 4A6220, getting 4A6424. That is where Malco's variables will be. Now let **X** be 4A6264.
This 'X' is the **pointer** that the entity reads from. It determines all variables for all entities in CS.

[X] is the value to determine whether the entity is alive. If the value is equal to 80, it's alive.

[X+4] is the value that is used to check whether the entity is colliding with a wall. Bit 1 is for left, bit 2 for up, bit 3 for right, and bit 4 for down. So if [X+4] is 0000 0000 0000 0000 0000 0000 0000 0110 in binary, that means that Malco is colliding with the ceiling and a wall on the right side.

[X+8] is the value that determines the X position of the entity. This, like [49E654] for Quote, is measured in 1/8192 of a block.

[X+C] is the Y value of the entity.

[X+10] does nothing by itself, but in entity codes in CS it's used to calculate the X velocity for the entity.

[X+14] is the same as [X+10], but for Y velocity.

[X+18] is also useless by itself, but is used for many purposes, one being X acceleration.

[X+1C] is once again useless, occasionally being used for Y acceleration.

[X+20] is used to tell Curly AI its own X position, allowing Curly to shoot it. Not important right Now.

[X+24] is used to tell Curly its Y position, blah blah blah…

[X+28] is used to tell the entity's type. So, for example, entity type 0x40 would be a regular critter.

[X+30] is the entity's event number on the map editor. You can also see this in the BL screenshot in the previous page.

[X+34] is the entity's spritesheet it reads from. This is used when the game renders the entity, Obviously. Examples include:

0x0 - Title image

0x1 - '2004 Studio Pixel'

0x2 - Current Map Tileset

You can see more on this in BL's npc.tbl editor.

[X+38] is the sound played by the entity when it is hit by a bullet. Also accessible in BL npc.tbl.

[X+3C]is the sound played by the entity when it dies. From npc.tbl, blah blah blah

[X+40] is the current health of the NPC. It dies when the value reaches 0 or below.

[X+44] is the amount of EXP/hearts/missiles the NPC drops when dead. Hearts and missiles can only be 2, 6 or 1, 3 respectively.

[X+48] is the size of the explosion created by the NPC's death. This also indirectly affects the NPC's maximum size. (Larger death explosion, larger maximum size)

[X+4C]is the NPC's direction. In BL, this can be modified by setting flag 0x1000 'Spawn with alt direction,' which sets the direction to 2. It can also be modified by <CNP, <ANP, and <SNP.

[X+50] is the actual flags of the entity. For example, flag 1 would be 'Solid,' flag 2 would be 'Ignore Tile 44,' etc. Accessible via BL; also as previously mentioned, flag 0x1000 sets the NPC's current direction to 2.

[X+54] is the NPC's left display box frame edge. Used to set the spritebox size. (No wonder you Weren't able to properly change the NPC's sprite size in BL's npc.tbl editor).

[X+58] is the NPC's top display box frame edge.

[X+5C]is the right side.

[X+60] is the bottom side.

[X+64] is another free variable, used for counters.

[X+68] is once again a free variable.

[X+6C] ANOTHER FREE VARIABLE

[X+70] OMG ANOTHER ONE

[X+74] is another one…

[X+78] Another free variable, and I promise you this'll be the last one…

[X+7C]is the entity's hitbox left frame edge. Unlike the sprites, hitboxes are accessible in npc.tbl.

[X+80] is the top hitbox edge.

[X+84] is the right side.
[X+88] is the bottom side.
[X+9C]is the variable to determine whether the NPC is touching a bullet.
[X+A0]is the variable to determine the damage taken by an NPC, used for damage counters.
[X+A4]is the damage the NPC will deal to Quote on contact, another variable in npc.tbl.
[X+A8]is the value set by some entities being spawned by other entities. This value is used to let the entity know how it was spawned, and in some case, which entity it was spawned by. For example, the Fuzz Core spawning the Fuzzes, who have their [X+A8] set to the Fuzz Core's [X]! This means that Fuzzes can access any of their Fuzz Core's variables too, an example of Pixel's cleverer techniques. This is required for the Fuzz enemy to rotate around a moving object (Fuzz Core).

Right. Now, where were we? Oh, we were back in the NPC calling function! I'll give you a moment to restore your mind (And possibly reread the stuff just before we went through variables) and also, FYI, there is a similar table for bullets as well! We won't go through the bullet variables right now, though.

Right. Back to work.

```
0046FA35      | AND EDX,00000080
```

^ This checks whether the NPC's [X] is 80, by ANDing it with 80. This also means that the entity is alive if the [X] value is 81 or FF, but not 7F or 100.

```
0046FA3B      | JE SHORT 0046FAA1
```

^ If the entity is dead, don't run the entity's AI.

```
0046FA3D      | MOV EAX,DWORD PTR SS:[LOCAL.1]              - Getting [X] again
0046FA40      | IMUL EAX,EAX,0AC                            -
0046FA46      | MOV ECX,DWORD PTR DS:[EAX+4A6248]
```

If [ID*AC+4A6220] is [X], then [ID*AC+4A6248] would be [X+28]. Make sense? If it doesn't take a closer look. 4A6248 = 4A6220 + 28. Get it now?

```
0046FA4C      | MOV DWORD PTR SS:[LOCAL.2],ECX
```

^ Saving the entity's [X+28] (entity type) to the second variable we created at the start of the code!

```
0046FA4F      | MOV EDX,DWORD PTR SS:[LOCAL.1]              - Getting [X] again
0046FA52      | IMUL EDX,EDX,0AC                            -
0046FA58      | ADD EDX,OFFSET 004A6220                     - Oh wait...
```

^ What have we done? We've actually made [EDX] equal to [X]!

```
0046FA5E      | PUSH EDX                                    PUSHing onto the stack!
0046FA5F      | MOV EAX,DWORD PTR SS:[LOCAL.2]              Grabbing entity type
0046FA62      | CALL NEAR DWORD PTR DS:[EAX*4+498548]       Oh noes…
```

Okay, here's something you might not've seen before.

You can CALL the value from a variable. If a variable contains an address that can be CALLed, the CALL will work.
Now, we're actually CALLing a value of four times the entity type plus 498548. If you go to 498548, you'd see a table of values, and the bytes are stored in reverse for easier access to the value in different sizes. If you don't follow this, don't worry. Just keep in mind that 498548 is the list of entity 'AI's that can be CALLed.

So it actually **CALLs the entity's AI, PUSHing the [X]** for the entity first!

```
0046FA69    | ADD ESP,4
```

Let's see why we are adding 4 to ESP. Did you notice that subtracting 8 values from ESP creates 2 new values? Well, adding 4 to it unconditionally removes 1 value. It's just like a POP, but without modifying anything other than ESP and the stack. Why do we do this? To remove the numbers PUSHed before the CALL. In this case that'll be the [X] we gave the NPC's AI to work with.

```
0046FA6C    | MOV ECX,DWORD PTR SS:[LOCAL.1]          Getting [X] again...
0046FA6F    | IMUL ECX,ECX,0AC
0046FA75    | MOVZX EDX,BYTE PTR DS:[ECX+4A62BC]      [ECX+4A62BC] is [X+9C]
0046FA7C    | TEST EDX,EDX                            Checking whether NPC is
0046FA7E    | JE SHORT 0046FAA1                       touching a bullet
0046FA80    | MOV EAX,DWORD PTR SS:[LOCAL.1]          If the NPC is, decrease the
0046FA83    | IMUL EAX,EAX,0AC                        frames that it is touching for
0046FA89    | MOV CL,BYTE PTR DS:[EAX+4A62BC]         by 1 (Since it runs once per
0046FA8F    | SUB CL,1                                frame)
0046FA92    | MOV EDX,DWORD PTR SS:[LOCAL.1]
0046FA95    | IMUL EDX,EDX,0AC
0046FA9B    | MOV BYTE PTR DS:[EDX+4A62BC],CL         Replacing the original value
0046FAA1    \ JMP 0046FA0F                            Jumping back to entity check
```

So we're JMPing backwards here, which means a loop. This loop goes on to check the values for every entity. Then, after checking the last possible entity, it JGEs to 46FAA6, which is the command below.

```
0046FAA6    MOV ESP,EBP
```

Remember our MOV EBP, ESP at the start, saving both EBP and ESP using EBP? Well, now we're restoring ESP using the EBP.

```
0046FAA8    POP EBP
```

If the original (before the command) EBP was in [EBP], and we now set ESP equal to EBP, now the original EBP will be in the new [ESP]. Therefore, POP EBP restores EBP, completing the function, also restoring ESP to what it was immediately after the command was CALLed!

```
0046FAA9    RETN
```

Now we get to return back to the function that CALLed this - the main game loop.

Here’s something interesting about the way the entity AIs are made.
Do you remember how the entity’s AI is CALLed from a table, and each AI has a corresponding entity type, so each type also has its own AI?
And also that [X] for the entity is PUSHed just before the CALL?
Let’s see why that is, and let’s open up one of the entity AIs.
An easy one first, the Null entity (entity 0).
The first CALL, if you go to 498548, you’ll see the first few commands look like this:

```
00498548    3065 42              XOR BYTE PTR SS:[EBP+42],AH
0049854B    00B0 654200F0        ADD BYTE PTR DS:[EAX+F0004265],DH
```

What the hell is going on? Take a look at not the commands, but the numbers in the middle. The commands shown are what Olly thinks they should be. But they’re not supposed to be commands. Notice how the first 4 groups of 2 numbers are ‘30’ ‘65’ ‘42’ ‘00’.
You might say that they look random, but what if you reverse the order, like the computer does when it reads that sequence?
⇒ 00 42 65 30!
What’s that? The first CALL for the first entity type, entity 0!
There, you’ve figured out the location of the entity 0’s AI!
Let’s go there. (Using CTRL+G)

```
00426530    PUSH EBP                            Initiating command, saving stack
00426531    MOV EBP,ESP
00426533    SUB ESP,10                          Creating 4 variables now!
00426536    MOV DWORD PTR SS:[LOCAL.4],0        Setting all 4 variables
0042653D    MOV DWORD PTR SS:[LOCAL.3],0
00426544    MOV DWORD PTR SS:[LOCAL.2],10
0042654B    MOV DWORD PTR SS:[LOCAL.1],10
00426552    MOV EAX,DWORD PTR SS:[ARG.1]
```

What’s [ARG.1]? It’s like another variable, but if you click on it in Olly you’ll notice it says [EBP+8]. Sound familiar? It should be. [EBP+8] is the value PUSHed before the CALL. Again, sound familiar? Also should be. [EBP+8] is the address of [X] we PUSHed before! So, [EAX] is now [X].

```
00426555    CMP DWORD PTR DS:[EAX+74],0
```

If [EAX] is [X], then [EAX+74] is [X+74]. If you go back and check, that’d make it one of the free variables. These variables are specific for every entity.

```
00426559    JNE SHORT 00426580                  If not 0, jump forwards
0042655B    MOV ECX,DWORD PTR SS:[ARG.1]        Getting [X]
0042655E    MOV DWORD PTR DS:[ECX+74],1         Setting to 1 if not 1
00426565    MOV EDX,DWORD PTR SS:[ARG.1]        Getting [X]
00426568    CMP DWORD PTR DS:[EDX+4C],2         Checking if [X+4C] is 2
```

^ Just a reminder if you’d forgotten, [X+4C] is the entity’s direction.

```
0042656C    JNE SHORT 00426580
0042656E    MOV EAX,DWORD PTR SS:[ARG.1]                 Grabbing [X]
00426571    MOV ECX,DWORD PTR DS:[EAX+0C]                [X+C] is Y position
00426574    ADD ECX,2000
```

^ Why are we getting the Y position? If the entity's direction is 2, then the entity spawns 0x2000 units, or 8192/8192 of a block, or 1 block below!

```
0042657A    MOV EDX,DWORD PTR SS:[ARG.1]
0042657D    MOV DWORD PTR DS:[EDX+0C],ECX                Resetting [X+C]
00426580    MOV EAX,DWORD PTR SS:[ARG.1]
00426583    ADD EAX,54
```

^ Wait a moment, why are we adding 54 to X, not [X]? Because it saves trouble writing [X+54] and stuff later (and confusion as well). So now [EAX] is [X+54], [EAX+4] is [X+58] etc.

```
00426586    MOV ECX,DWORD PTR SS:[LOCAL.4]     Grabbing 4th variable (0x0)
00426589    MOV DWORD PTR DS:[EAX],ECX         Setting position of left spritebox to 0
0042658B    MOV EDX,DWORD PTR SS:[LOCAL.3]     Grabbing 3rd variable (0x0)
0042658E    MOV DWORD PTR DS:[EAX+4],EDX       Setting position of top spritebox to 0
00426591    MOV ECX,DWORD PTR SS:[LOCAL.2]     Grabbing 2nd variable (0x10)
00426594    MOV DWORD PTR DS:[EAX+8],ECX       Setting position of right side to 10
00426597    MOV EDX,DWORD PTR SS:[LOCAL.1]     Grabbing 1st variable (0x10)
0042659A    MOV DWORD PTR DS:[EAX+0C],EDX      Setting position of bottom side to 10
0042659D    MOV ESP,EBP                        Restoring ESP
0042659F    POP EBP                            Restoring EBP and original stack
004265A0    RETN                               Returns to 46FA69
```

There are a few things to look at here. Firstly, that entity 0 does have a sprite, and that its sprite is just invisible in the game. Secondly, you might be wondering, why set the local variables when the only thing you need is to set the invisible spriteboxes? Well technically, for this entity, you can simply replace everything with RETN and it'd still work as normal. However, the technique for spriteboxes, although wastes coding space, **saves confusion** when copying and determining multiple moving sprites. Unless you're an expert you might want to start off with this method.

Another thing to note is that it keeps grabbing [EBP+8] when we don't actually need it (Since we only need to MOV it once and it's saved). The reason for this is that the code was compiled from C++, which didn't know when we actually needed to reset it (Whether the register was modified and whether the stored [X] was edited).

Now, time to edit an entity in the actual game! Let's make entity 0xE3, or 'Kanpachi's bucket' spawn an EXP crystal every frame!

I'll save you the trouble of reading through 498548 backwards by telling you the location of the AI.

Go to 452D10 and read through the code step by step, like I did before. You'll see that it acts just like entity 0, except this time it's got an actual visible sprite, at locations 20~30 and D0~E0. If you open up the spritesheet 'NPCGuest' and convert the numbers to decimal, the sprite should be:
32 ~ 48, 208 ~ 224.
Find and look at the bucket sprite, and take a look (in MS paint or PDN) at the corners of the sprite. See any resemblance? These are the actual positions of the sprite.

Now, let's cut out some of that junk from the thing with a few optimisation tricks.
Because we only have one sprite, we don't need any local variables. Because we don't need local variables, we don't actually need to start with a stack entry!
When the AI for 'bucket' is CALLed, the last number PUSHed will be the RETN address, which for entities is 46FA69. The second last number is the [X] for the entity. So, we can actually do this.

```
POP ESI
POP EDI
```

Now [EDI] is [X] and [ESI] is the RETN address!
Now we move on quickly with the sprites.

```
PUSH 20
POP DWORD [EDI+58]
PUSH 30
POP DWORD [EDI+60]
SUB EAX, EAX
MOV AL, D0
MOV DWORD [EDI+54], EAX
ADD AL, 10
MOV DWORD [EDI+5C], EAX
```

That's the sprites done. By the way, SUB EAX, EAX is the same as MOV EAX, 0, but it **somehow** uses less space, because of how machine code works! If you know your logic gates, XOR EAX, EAX works the same way as well.
Why are we PUSHing the numbers and POPping them into the variables instead of setting them in the first place? Because that also uses less space! MOV DWORD [EDI+58], 20 uses more space than PUSH 20 / POP DWORD [EDI+58].
In fact, MOVing a BYTE into a DWORD is the best way to waste space! Even setting EAX to something and MOVing it into the DWORD, or modifying individual bytes of EAX (AL) is shorter in code!

Now, we go to the spawn entity function! For this, I should give you this link, https://cdn.discordapp.com/attachments/312733438153326593/426964925358014464/Noxids_Function_List.txt, made by Noxid and a few others. This should help a bit. From this link, the spawn entity function is 46EFD0. It is a CALL, but you have to specify 8 numbers via PUSHing first, to determine the entity's variables.

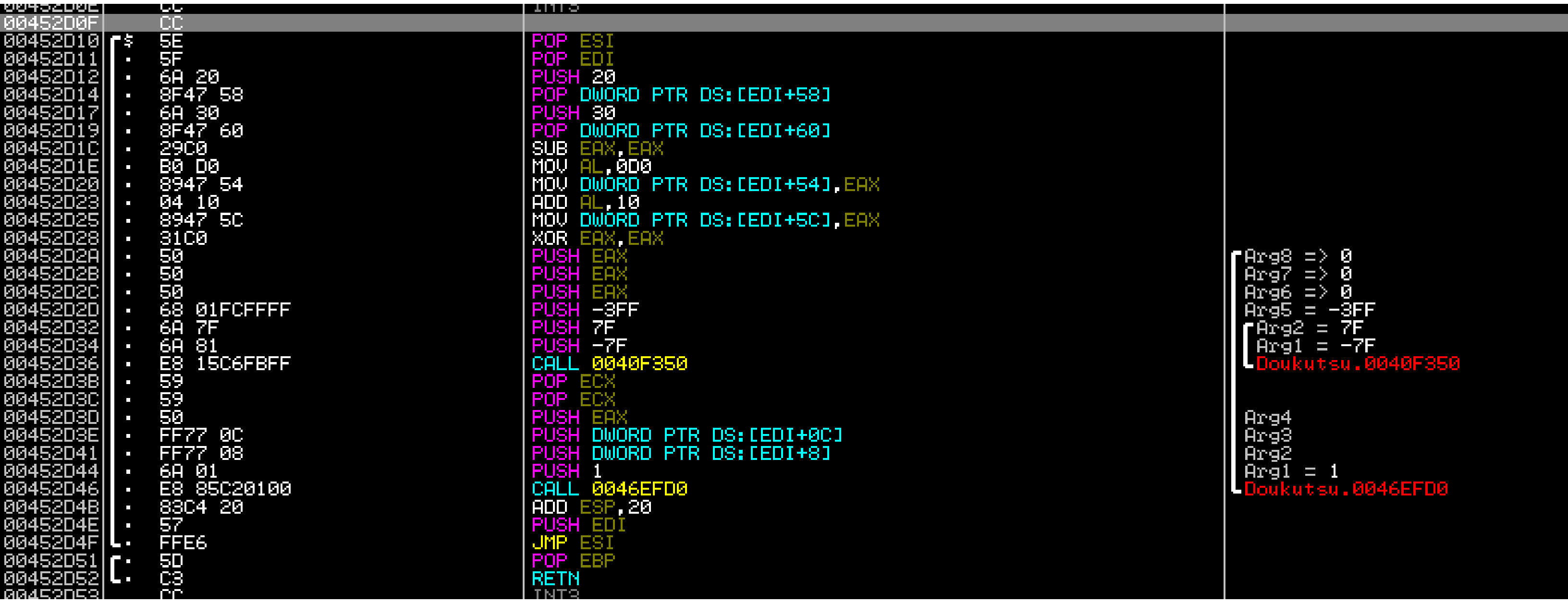
It works like this:
PUSH (Number to start with, basically maximum amount of this type of entity you can have on the map, is subtracted from 0x200, so the lower the number, the higher the limit. Use 0 if you’re unsure.)
PUSH ([X+A8] of the entity. This is used to give the entity an idea of something to track. Some entities crash the game if they can’t figure out how they were spawned, either from division by 0 or trying to access address 0.)
PUSH ([X+4C] direction of the entity.)
PUSH ([X+14] Y velocity of the entity. Immobile entities usually do not use this value, so you cannot make moving buckets or doors with this.)
PUSH ([X+10] X velocity.)
PUSH ([X+C] Y position, again in 1/8192 of a block.)
PUSH ([X+8] X position, again in blah blah blah.)
PUSH ([X+28] Entity type. The most important of all values PUSHed - what the entity is!)

So 46EFD0 is basically <SNP but with a few extra things.
So we’re to spawn one EXP crystal every frame, from inside the bucket. We should also make it look like it’s coming out of the bucket. To do this, you’ll have to understand the previous list of PUSHes before the CALL.
Let’s also make the direction randomised, like as if the entity is spewing EXP randomly. For this we’ll need function 40F350, the RNG function.
Let’s make the spawn limit the maximum, 0x200, or 512 in dec. Write this after the previous code.

```
XOR EAX, EAX                We’re actually resetting EAX to 0 and PUSHing that,
PUSH EAX                    because PUSH EAX uses half the space of PUSH 0.
PUSH EAX
PUSH EAX
PUSH -3FF                   PUSH Y Velocity in 1/8192 blocks per frame
PUSH 7F                     Now we’re accessing the RNG during a PUSH sequence,
PUSH -7F                    but don’t panic! The RNG functions like this: You PUSH 2
CALL 40F350                 values, and it finds a random number between, saved in
                            EAX. So in this case we want an X velocity of between 7F
                            and -7F.
POP ECX                     We also want to get rid of the PUSHed numbers after, so
POP ECX                     that we don’t mess up our spawn function.
PUSH EAX                    Now we use our random number.
PUSH DWORD [EDI+C]          [EDI+C] is [X+C], Y position of bucket.
PUSH DWORD [EDI+8]          [X+8], X position of bucket.
PUSH 1                      1 will be the entity’s type, the EXP entity.
CALL 46EFD0                 Finally we create the entity.
ADD ESP, 20                 We’ve PUSHed 8 numbers for the CALL, so now we get rid
                            of them by ADDing to ESP.
```

PUSH EDI	Remember how we took [X] off the stack? Well, now we'd better return it.
JMP ESI	Remember how we took the return address as well, and saved it into ESI? Now we can use that as a RETN.

Now, after entering the commands, press CTRL+A and Olly should analyse it as correct code.



```
00452D0E     CC              INT3
00452D0F     CC
00452D10  $  5E              POP ESI
00452D11  .  5F              POP EDI
00452D12  .  6A 20           PUSH 20
00452D14  .  8F47 58         POP DWORD PTR DS:[EDI+58]
00452D17  .  6A 30           PUSH 30
00452D19  .  8F47 60         POP DWORD PTR DS:[EDI+60]
00452D1C  .  29C0            SUB EAX,EAX
00452D1E  .  B0 D0           MOV AL,0D0
00452D20  .  8947 54         MOV DWORD PTR DS:[EDI+54],EAX
00452D23  .  04 10           ADD AL,10
00452D25  .  8947 5C         MOV DWORD PTR DS:[EDI+5C],EAX
00452D28  .  31C0            XOR EAX,EAX
00452D2A  .  50              PUSH EAX                        Arg8 => 0
00452D2B  .  50              PUSH EAX                        Arg7 => 0
00452D2C  .  50              PUSH EAX                        Arg6 => 0
00452D2D  .  68 01FCFFFF     PUSH -3FF                       Arg5 = -3FF
00452D32  .  6A 7F           PUSH 7F                          Arg2 = 7F
00452D34  .  6A 81           PUSH -7F                         Arg1 = -7F
00452D36  .  E8 15C6FBFF     CALL 0040F350                    Doukutsu.0040F350
00452D3B  .  59              POP ECX
00452D3C  .  59              POP ECX
00452D3D  .  50              PUSH EAX                        Arg4
00452D3E  .  FF77 0C         PUSH DWORD PTR DS:[EDI+0C]      Arg3
00452D41  .  FF77 08         PUSH DWORD PTR DS:[EDI+8]       Arg2
00452D44  .  6A 01           PUSH 1                          Arg1 = 1
00452D46  .  E8 85C20100     CALL 0046EFD0                   Doukutsu.0046EFD0
00452D4B  .  83C4 20         ADD ESP,20
00452D4E  .  57              PUSH EDI
00452D4F  .  FFE6            JMP ESI
00452D51  :  5D              POP EBP
00452D52  .  C3              RETN
00452D53     CC              INT3
```

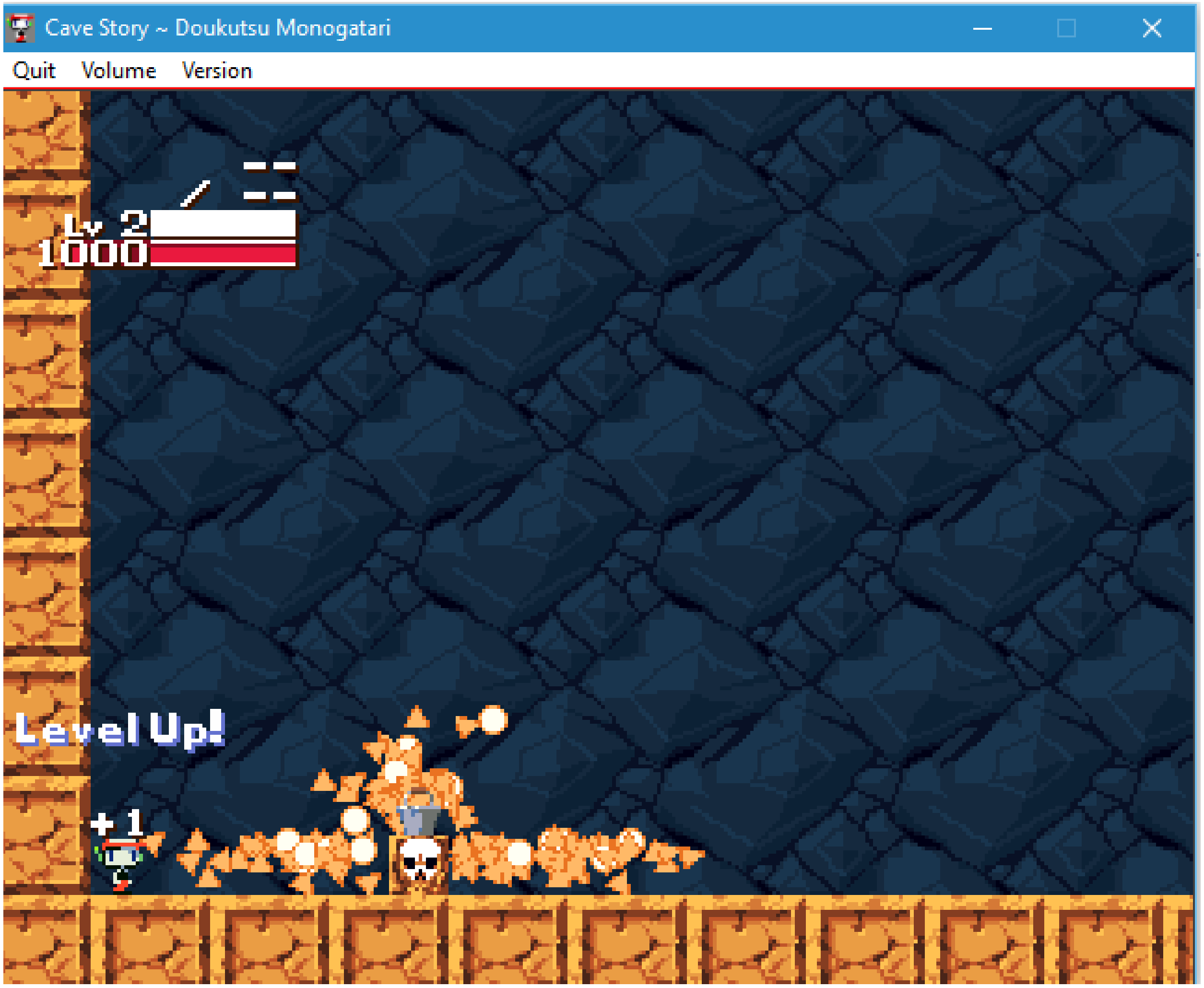
^ Don't mind the colours, just look at the brackets to the left of your code. If they don't show up, or the first one is split up into 2 or more groups, then that indicates you've typed something wrong. Don't worry, it's your first time editing whole chunks of code. Instead of forcing you to redo it, I'm going to let you off by letting you copypaste it in. Select the numbers below and copy (CTRL+C).

```
5E 5F 6A 20 8F 47 58 6A 30 8F 47 60 29 C0 B0 D0
89 47 54 04 10 89 47 5C 31 C0 50 50 50 68 01 FC
FF FF 6A 7F 6A 81 E8 15 C6 FB FF 59 59 50 FF 77
0C FF 77 08 6A 01 E8 85 C2 01 00 83 C4 20 57 FF
E6
```

Now go to Olly, select the entire code from 452D10 to 452D52 and right click. There should be an option for ‘Binary Paste.’ Use that to paste the code in. Then right click the whole area again, select ‘Copy to executable,’ and then a smaller, thinner window should pop up. Close that by pressing the ‘X’ on the top right corner, and Olly will ask to save it.
Do this only if entering the code didn’t work for you.

IMPORTANT: If this is your mod you’re editing, make sure to keep a backup in case something fails. Olly does backup, but they are easily lost by repeated hacks, since olly only keeps 1 backup.

Save it, use a level editor to place a bucket in a map that you can enter, open the exe, and enjoy!



There! You now have an EXP fountain!

By the way, there is an individual EXP spawning function, and that works by function 46F2B0. This function is required to spawn larger amounts of EXP, so if we were to spawn a 20 EXP triangle in one go, instead of the above 46EFD0 function we'd do:

```
PUSH 20
PUSH DWORD [EDI+C]
PUSH DWORD [EDI+8]
CALL 46F2B0
ADD ESP, C
```

...but I wanted you to learn the entity spawn function, and also it looks cooler if the EXP is spraying out of the bucket rather than from underneath the bucket.

Now, for more complicated NPCs, like critters, we need to actually move the NPC around a lot. Go to 433C00 (regular critter AI) and read through. Everything there should already be explained, except maybe this one.

```
433CC4: JMP NEAR DWORD PTR DS: [EDX*4+433FA5]
```

Now, if you read what is before this command, EDX contains [EBP-64], which in turn contains [EAX+74], the [X+74] ‘free variable’ of the entity. For critters (and in fact most other NPCs), [X+74] is used to determine the critter’s **state**, e.g. falling, waiting, jumping, landing. So, in this case, falling is 0, waiting is 1, jumping is 2, and landing is 3. This is multiplied by 4 and added to 433FA5, so if you go to 433FA5 you should see a few unusual commands, i.e. DD 433CCB. This is actually pre-stored variable, which is not modified in-game, that tells the critter’s AI where to jump to from each of the critter’s states.

Another thing is the commands CMP and TEST. We mentioned earlier that CMP is a sum, SUB, and that it doesn’t change values. TEST is a logical operation, AND, that doesn’t change values. Both can be used with jump checks. But in the critter code you’ll find this:

```
00433DB3:     TEST ECX, ECX
00433DB5:     JE SHORT 433DD5
```

Okay, so if we’re ANDing something with itself, obviously we’re going to get the same answer, but what does this have to do with the JE after?

Remember that JE only jumps if the result of the last sum was 0. So, basically, TESTing something with itself just **checks whether it is 0.** CMP ECX, 0, SUB ECX, 0, or ADD ECX, 0 would work too, but that uses more space. Ways to check if (for example) EAX = 0 are:

```
TEST EAX, EAX,
AND EAX, EAX,
OR EAX, EAX,
DEC EAX / INC EAX,
INC EAX / DEC EAX.
```

All of these can be used to check if EAX is 0, and all of them leave EAX unchanged. Works for any other register.

Now, moving on…

```
0x433E8F:     PUSH 1
0x433E91:     PUSH 1E
0x0433E93:    CALL 420640
0x0433E98:    ADD ESP, 8
```

Aha! Another one of those CALLs! This one, 420640, is the one for playing a sound effect. If you take a look at the game’s sound effects with a program such as SeaTone (or the <SOU in TSC script editor in BL), you’ll see that sound 0x1E, or 30 in dec, is the critter’s hopping sound!

But what about the ‘1’ PUSHed first? This tells the sound engine how it is played. Usually we use 1, because there are 2 channels for sounds, PUSHing 0 uses channel 1, while PUSHing -1 uses channel 2. Because sometimes a channel is off, PUSHing 1 is better since it plays the same sound on both channels, preventing a sound not being played properly.

Read through the rest of the code, and you should get a decent idea of how critters work now.

In CS, bullets work very similarly as NPCs, in the way that they also get CALLed by a universal bullet command. Like NPCs, bullets also have their [X], however the variables used are a little different. The values are listed as follows:

X = [EBP+8]

[X] is the value to determine whether the bullet is colliding with a block. In bits, identical to the one for NPCs.

[X+4] is the type of bullet it is. The bullet types are listed below:

0 = Null

1 = Snake level 1

2 = Snake level 2

3 = Snake level 3

4 = Polar Star level 1

5 & 6 = Polar Star level 2 & 3

7 ~ 9 = Fireball level 1 ~ 3

A ~ C = Machine Gun level 1 ~ 3

D ~ F = Missile Launcher level 1 ~ 3

10 ~ 12 = Explosion 1 level 1 ~ 3

13 ~ 15 = Bubbler level 1 ~ 3

16 = ‘Bubbler level 4’ (Popped level 3 bubbles)

17 = ‘Blade level 4’ (Slashing level 3)

18 = Falling Spike NPC damage

19 ~ 1B = Blade level 1 ~ 3

1C ~ 1E = Super Missile Launcher level 1 ~ 3

1F ~ 21 = Explosion 2 level 1~3

22 ~ 24 = Nemesis level 1 ~ 3

25 ~ 27 = Spur level 1 ~ 3

28 ~ 2A = Spur Trail level 1 ~ 3

2B = Curly Nemesis level 1

2C = Instant kill all shootable entities on screen

2D = Whimsical Star trail

[X+8] is the flags of the bullet, e.g. Ignore Solid, Ignore Invincible Entity, etc

[X+C] is whether the bullet is ‘Alive.’ If not 80, then it dies.

[X+10] is the X position of bullet in 1/8192 of a block.

[X+14] is the Y position blah blah blah…

[X+18] is a free variable; Pixel uses this to store X velocity of bullet.

[X+1C] is another free variable, used to store Y velocity of bullet.

[X+20] is a free variable, not really used in-game.
[X+24] is another unused free variable.
[X+28] is another free variable, used to determine whether the bullet is setup already.
[X+2C] is yet another free variable.
[X+30] is another free variable, occasionally used for timers.
[X+34] is another free variable, usually used to calculate states.
[X+38] is the direction of the bullet.
[X+3C]is the left side of bullet sprite.
[X+40] is the top side.
[X+44] is the right side.
[X+48] is the bottom side.
[X+4C] is a free variable, used to calculate the 'life' of the bullet.
[X+50] is a free variable, not used very often.
[X+54] is the maximum life of bullet. Doesn't do anything unless you kill the bullet when its life runs
out.
[X+58] is the damage of the bullet.
[X+5C]is the amount of hits the bullet has left. Allows a bullet to hit the same entity multiple times,
or multiple entities, e.g. level 2 Blade can hit 1 entity dealing 18 damage or 3 entities, dealing 6 each. Bullet automatically dies when [X+5C] runs out (to 0).

So, let's go on to modifying a bullet… How about making the level 1 Polar Star shoot
ONE OF EVERY BULLET IN THE GAME?

The 'AI' that controls the Polar Star bullet starts at 4047B0, so now you can pretend you knew that already and search for it in Olly.
Now, if the level 1 Polar Star bullet is ID 4, that means that the bullet is level 1 if and only if the ID is 4. We can use that to check by subtracting 4 and seeing if the answer is 0.
Unless you want to go optimising Pixel's code and make it shorter, we're going to use some free space. First, make a check at the start of the code.

```
004047B0    PUSH EBP
004047B1    MOV EBP, ESP
004047B3    SUB ESP, 74
004047B6    MOV ECX, DWORD PTR SS: [EBP+8]
004047B9    INC DWORD PTR DS: [ECX+4C]
004047BC    CMP DWORD PTR DS: [ECX+4], 4
004047C0    JE 0048B900
004047C6    NOP
004047C7    NOP
004047C8    NOP
004047C9    NOP
004047CA    NOP
004047CB    MOV EDX, DWORD PTR DS: [ECX+4C]
```

Ignore the rest of the code. What's important is that we've made the code jump to 48B900, free space, if the ID is 4 (if the level is 1). If you've forgotten, INC is just like ADD except it can only add 1, and it uses slightly less space.

Now go to 48B900 and write the following:

```
0048B900    AND DWORD PTR SS:[EBP-4],00000000
0048B904    / CMP DWORD PTR SS:[EBP-4],27
0048B908    | JA SHORT 0048B92C
0048B90A    | MOV ECX,DWORD PTR SS:[EBP-4]
0048B90D    | INC DWORD PTR SS:[EBP-4]
0048B910    | CMP ECX,4
0048B913    | JE SHORT 0048B904
0048B915    | MOV EDX,DWORD PTR SS:[EBP+8]
0048B918    | PUSH DWORD PTR DS:[EDX+38]
0048B91B    | PUSH DWORD PTR DS:[EDX+14]
0048B91E    | PUSH DWORD PTR DS:[EDX+10]
0048B921    | PUSH ECX
0048B922    | CALL 00403F80
0048B927    | ADD ESP,10
0048B92A    \ JMP SHORT 0048B904
0048B92C    MOV EDX,DWORD PTR SS:[EBP+8]
0048B92F    AND DWORD PTR DS:[EDX+0C],00000000
0048B933    LEAVE
0048B934    RETN
```

Okay, a few things to note here.

- Yes, we are making a loop to spawn one of every entity. It'd be **extremely** tedious to individually spawn every bullet. The loop's counter is [EBP-4].
- CALL 403F80 is the bullet spawn function, with PUSHes as follows:
  PUSH (Direction)
  PUSH (Y position in 1/8192 blocks)
  PUSH (X position in 1/8192 blocks)
  PUSH (ID)
  CALL 403F80
- We had to CMP ECX, 4 at first, because we can't let the bullet spawn itself, otherwise it'd keep spawning itself forever!
- ANDing something with 0 is just like MOVing 0 into it (Can you see why?) and uses less space!
- LEAVE is a substitute for these 2 commands (that uses less space):
  MOV ESP, EBP
  POP EBP
- This code could be further optimised, but I didn't want to leave you confused.

Now, right click on any of the commands, click 'Select All,' then right click again, copy to executable, close the window popup and save!
Enjoy!

So now you should have the basics of NPC and bullet editing. So, I'm going to tell you about two other tools that are great at CS ASM modding: Doukutsu Assembler, and PEONS. Both of these applications were made by CSTSF members. DouA was made by Carrotlord, and PEONS was made by gamemanj, who later renamed himself 20kdc. If you use either of them, you should probably consider crediting them.

Here are the download links.
DouA:
https://www.dropbox.com/s/yi2i0t4hnvybxpc/Doukutsu%20Assembler%201.31.zip?dl=0
PEONS:
https://www.dropbox.com/s/mdmlzw7wy7ftc3b/PEONS.rar?dl=0

To use DouA, view http://www.cavestory.org/forums/threads/doukutsu-assembler.2658/, when it was created.

As for PEONS, gamemanj/20kdc did not make a guide, so I'll briefly explain it here:

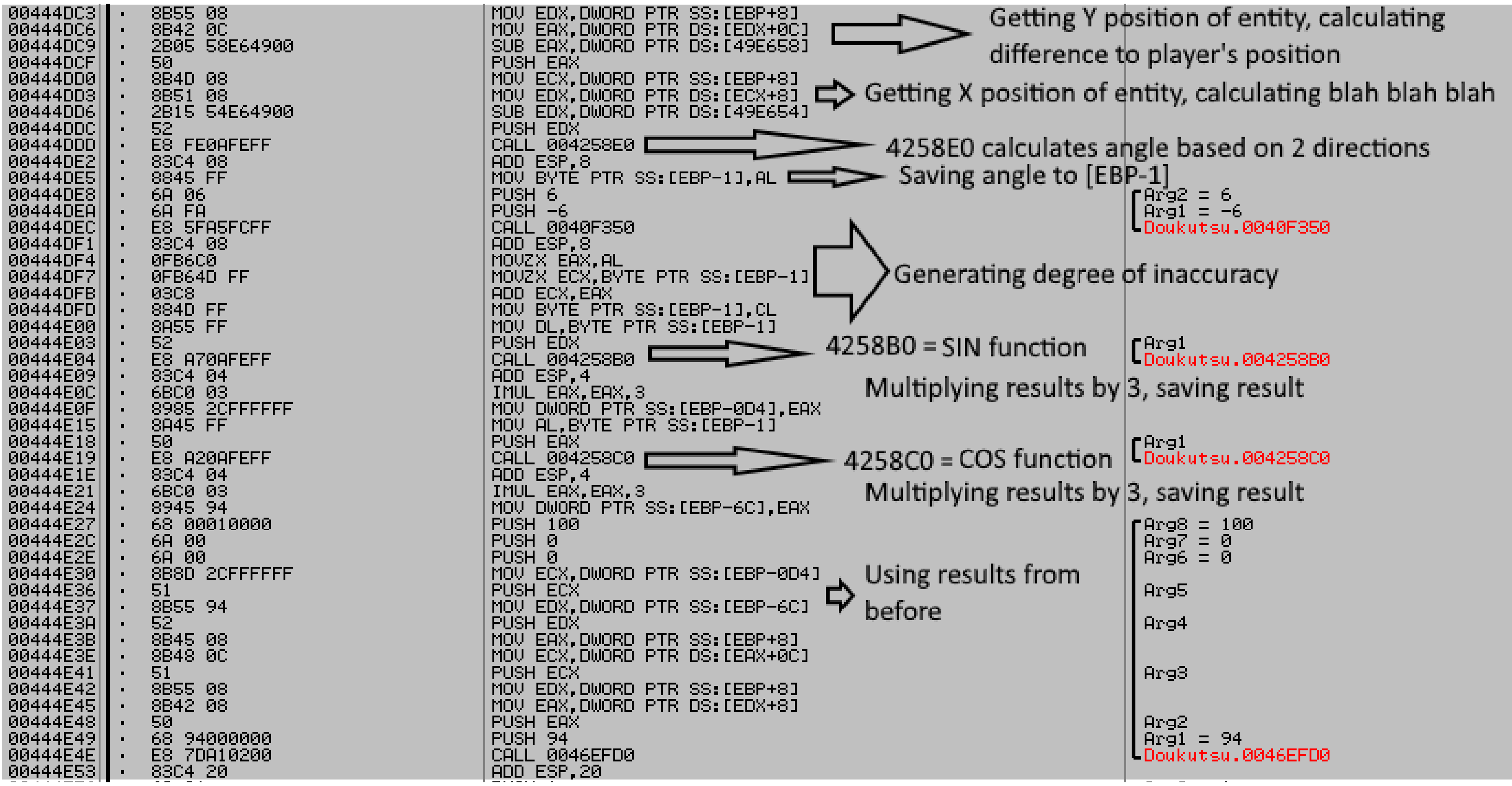
- PEONS is a CS modding tool that can add assembly space between segments without messing up other data.
- It can be used in conjunction with many other things, like NICE-compatible stuff.
- It gives a variety of possibilities, for example increasing the entity limit and bullet limit (https://www.youtube.com/watch?v=NF9jYHB0Xbs and https://www.youtube.com/watch?v=aiFfA2PHhPA)
- It also allows the creation of new variables or new game mechanics through the free space.

So, if you open it up, you'll see a window that pops up. Don't touch anything yet, especially 'Delete Segment.'
Let's say you wanted to add 400,000 bytes to your executable. Click on 'Linearize,' then, after confirming the linearize, click on 'Automatic.'
Then, in the box underneath the one with '.gsa2,' replace the '10000' with '400000' or however much data you want to add. You can add as much data as you like, but your friends won't like your mod very much if it's five hundred yottabytes!

As for Doukutsu Assembler, use that anytime you want, for big, messy functions like large AIs. When using it, make sure you're either writing into free space, or writing into a large function, because you don't want to be going too big and overwriting necessary functions!

For smaller edits, and complicated commands such as floating point, trigonometry, and the like (We'll get through that later), it's better to use Ollydbg, mainly because Olly can use all ASM commands, not just basic sums and data editing.

Now that's that. I mentioned trigonometry earlier, would you like to start on that?
Okay, let's get into the basics of this, and get an example of how this works in the original CS. One example would be the Labyrinth purple critters that can shoot the purple blobs at you. So how exactly do they use trig to calculate how to shoot at the player?
Don't worry if you haven't learnt trig at school yet, you can still copy this function when you need it.



```
00444DC3  .  8B55 08          MOV EDX,DWORD PTR SS:[EBP+8]
00444DC6  .  8B42 0C          MOV EAX,DWORD PTR DS:[EDX+0C]
00444DC9  .  2B05 58E64900    SUB EAX,DWORD PTR DS:[49E658]
00444DCF  .  50               PUSH EAX
00444DD0  .  8B4D 08          MOV ECX,DWORD PTR SS:[EBP+8]
00444DD3  .  8B51 08          MOV EDX,DWORD PTR DS:[ECX+8]
00444DD6  .  2B15 54E64900    SUB EDX,DWORD PTR DS:[49E654]
00444DDC  .  52               PUSH EDX
00444DDD  .  E8 FE0AFEFF      CALL 004258E0
00444DE2  .  83C4 08          ADD ESP,8
00444DE5  .  8845 FF          MOV BYTE PTR SS:[EBP-1],AL
00444DE8  .  6A 06            PUSH 6                                  Arg2 = 6
00444DEA  .  6A FA            PUSH -6                                 Arg1 = -6
00444DEC  .  E8 5FA5FCFF      CALL 0040F350                           Doukutsu.0040F350
00444DF1  .  83C4 08          ADD ESP,8
00444DF4  .  0FB6C0           MOVZX EAX,AL
00444DF7  .  0FB64D FF        MOVZX ECX,BYTE PTR SS:[EBP-1]
00444DFB  .  03C8             ADD ECX,EAX
00444DFD  .  884D FF          MOV BYTE PTR SS:[EBP-1],CL
00444E00  .  8A55 FF          MOV DL,BYTE PTR SS:[EBP-1]
00444E03  .  52               PUSH EDX                                Arg1
00444E04  .  E8 A70AFEFF      CALL 004258B0                           Doukutsu.004258B0
00444E09  .  83C4 04          ADD ESP,4
00444E0C  .  6BC0 03          IMUL EAX,EAX,3
00444E0F  .  8985 2CFFFFFF    MOV DWORD PTR SS:[EBP-0D4],EAX
00444E15  .  8A45 FF          MOV AL,BYTE PTR SS:[EBP-1]
00444E18  .  50               PUSH EAX                                Arg1
00444E19  .  E8 A20AFEFF      CALL 004258C0                           Doukutsu.004258C0
00444E1E  .  83C4 04          ADD ESP,4
00444E21  .  6BC0 03          IMUL EAX,EAX,3
00444E24  .  8945 94          MOV DWORD PTR SS:[EBP-6C],EAX
00444E27  .  68 00010000      PUSH 100                                Arg8 = 100
00444E2C  .  6A 00            PUSH 0                                  Arg7 = 0
00444E2E  .  6A 00            PUSH 0                                  Arg6 = 0
00444E30  .  8B8D 2CFFFFFF    MOV ECX,DWORD PTR SS:[EBP-0D4]
00444E36  .  51               PUSH ECX                                Arg5
00444E37  .  8B55 94          MOV EDX,DWORD PTR SS:[EBP-6C]
00444E3A  .  52               PUSH EDX                                Arg4
00444E3B  .  8B45 08          MOV EAX,DWORD PTR SS:[EBP+8]
00444E3E  .  8B48 0C          MOV ECX,DWORD PTR DS:[EAX+0C]
00444E41  .  51               PUSH ECX                                Arg3
00444E42  .  8B55 08          MOV EDX,DWORD PTR SS:[EBP+8]
00444E45  .  8B42 08          MOV EAX,DWORD PTR DS:[EDX+8]
00444E48  .  50               PUSH EAX                                Arg2
00444E49  .  68 94000000      PUSH 94                                 Arg1 = 94
00444E4E  .  E8 7DA10200      CALL 0046EFD0                           Doukutsu.0046EFD0
00444E53  .  83C4 20          ADD ESP,20
```

Getting Y position of entity, calculating difference to player's position

Getting X position of entity, calculating blah blah blah

4258E0 calculates angle based on 2 directions

Saving angle to [EBP-1]

Generating degree of inaccuracy

4258B0 = SIN function

Multiplying results by 3, saving result

4258C0 = COS function

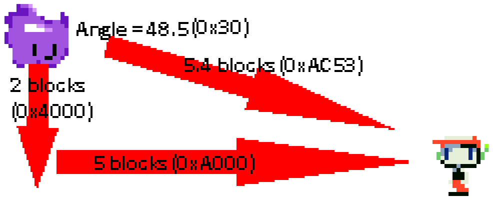
Multiplying results by 3, saving result

Using results from before

Angles in CS are calculated to the nearest 1/256 of a circle, rather than degrees or radians. This is to make it easier for the computer to calculate.

4258B0 and 4258C0 read from a table of values pre-calculated by a trig function using the FPU (We'll look at that later as well). The results of these calculations give a velocity of 512/8192 = 0.0625 (dec) blocks per frame, or 3.125 blocks per second. The critters multiply that by 3 (Both horizontal and vertical), giving approximately 9.375 blocks per second for their projectile. They summon entity 0x94, or in decimal form, 148. This is the critter's projectile, obviously.

Here's an example of a calculation that might occur.



In this case, the X velocity and Y velocity will be calculated, the Y velocity being 0xDA, and the X velocity being 0x157.

Of course, the overall velocity it is going in is 0x200 (Since it's calculated in a circle).

The critter will then multiply each of these velocities by 3, giving an X velocity of 0x405 and a Y velocity of 0x28E.

We can actually optimise the critter’s trig functions made by Pixel into something more like this:

```
MOV EDI, DWORD PTR SS: [EBP+8]         (Set pointer)
MOV EAX, DWORD PTR DS: [EDI+0C]        (Get Y value)
SUB EAX, DWORD PTR DS: [49E658]        (Find difference to Player’s Y value)
PUSH EAX                               (Resulting side of triangle)
MOV EDX, DWORD PTR DS: [EDI+8]         (Get X value)
SUB EDX, DWORD PTR DS: [49E654]        (Find difference to Player’s X value)
PUSH EDX                               (Resulting side of triangle)
CALL 004258E0                          (Get angle to Player)
PUSH EAX                               (Using the angle calculated)
CALL 004258B0                          (Sine Function)
XCHG EAX, EBX                          (Putting value into EBX to save it)
CALL 004258C0                          (Cosine Function)
MOV ESP, EBP                           (Reset Stack)
```

^ This will store the Y velocity in EBX and X velocity in EAX. You can then multiply these however you like, for example in the case of the critter by 3 each.

Let’s make a function that fires an NPC with velocity directed at the player.
Go to 0x0494000 and write:

```
PUSH EBP
MOV EBP, ESP
MOV EDI, DWORD PTR SS: [EBP+8]
MOV EAX, DWORD PTR DS: [EDI+0C]
SUB EAX, DWORD PTR DS: [49E658]
PUSH EAX
MOV EDX, DWORD PTR DS: [EDI+8]
SUB EDX, DWORD PTR DS: [49E654]
PUSH EDX
CALL 4258E0
MOV ESP, EBP
PUSH EAX
CALL 004258B0
XCHG EAX, EBX
CALL 004258C0
SHL EAX, 1
SHL EBX, 1
MOV ESP, EBP
SUB ESI, ESI
PUSH ESI
PUSH ESI
PUSH ESI
PUSH EBX
PUSH EAX
PUSH DWORD [EDI+C]
PUSH DWORD [EDI+8]
PUSH DWORD [EBP+C]
CALL 46EFD0
LEAVE
RETN
```

What have we made? An assembly function that actually fires an NPC at the player.
To run it, first PUSH the NPC ID that you want to fire at the player. Not all NPCs can be fired; examples that can be fired are the Labyrinth Critters’ projectiles, Skullhead bones, Fish Missiles, Core energy blasts, etc.
Then PUSH the DWORD stored in [EBP+8] of the entity firing the bullet. This function will allow you to fire an entity out of another entity, but will NOT allow you to fire one out of nothing. For now, you can do this to fire an enemy bullet.

As for the command SHL, that basically means Multiply X by 2 ^ Y. Like this:
SHL EAX, 1
⇒ EAX = EAX * 2 ^ 1 = EAX * 2
SHL EDX, 3
⇒ EDX = EDX * 2 ^ 3 = EDX * 8.

Another way to look at these functions (the way they're supposed to be looked at) is through the bits (again)...
Let's say EAX = 15F.
SHL EAX, 7
0000 0000 0000 0000 0000 0001 0101 1111
0000 0000 0000 0000 1010 1111 1000 0000 ⇐ 7 places!
= 0xAF80.

There is another command, SHR, that is the opposite of SHL. It divides numbers by powers of 2, rounding DOWN.
SHR EAX, 7
0000 0000 0000 0000 0000 0001 0101 1111
0000 0000 0000 0000 0000 0000 0000 0010 ⇒ 7 places!
= 0x2.

There are also 4 other shifts, and they work like this:

- SAL, which is exactly the same as SHL, so they are completely interchangeable. (There should really only be 1 of those commands, haha)
- SAR, which is like SHR, but for signed numbers. So for EAX = 0x-15F, it'd be like this:
  1111 1111 1111 1111 1111 1110 1010 0001
  1111 1111 1111 1111 1111 1111 1111 1101 ⇒ 7 places, but signed,
  ^ SAR recognises that the number is negative, and continues it by adding 1s on the left instead of 0s.
- ROL, which is like SHL but it replaces the bits that fall off the end back onto the front. So, imagine EAX = 0x58F013EA,
  ROL EAX, 3
  0101 1000 1111 0000 0001 0011 1110 1010
  1100 0111 1000 0000 1001 1111 0101 0010 ⇐ 3 places, but keeping all bits
- ROR, which is like ROL, but sends the bits in the opposite direction.

Let’s make the Debug Cat fire bullets at the player.

```
0x04517F0
PUSH EBP
MOV EBP, ESP
MOV ESI, DWORD [EBP+8]              ;Set Pointer, also allows us to PUSH this
XOR EAX, EAX                        ;Setting EAX to 100 using less space
MOV AL, -1
INC EAX
MOV DWORD [ESI+54], EAX
ADD AL, 10
MOV DWORD [ESI+5C], EAX
MOV BYTE [ESI+58], C0
MOV BYTE [ESI+60], D8
MOV ECX, DWORD [ESI+4C]             ;Using this way allows to check for 0
JECXZ :End of Code                   using less space
INC DWORD [ESI+6C]
CMP DWORD [ESI+6C], 7F              ;0x7F = 127 (dec)
JL :End of Code
AND DWORD [ESI+6C], 0
PUSH 94
PUSH ESI                            ;We had [EBP+8] (X) in ESI, remember?
CALL 494000                         ;The function we made earlier
:End of Code (Not a function, just to tell you where the jumps go)
LEAVE
RETN
```

The Debug Cat will now fire a Labyrinth Critter projectile at the player every 127 (dec) frames, but only if the direction is set to not 0, or Flag 0x1000 is set.

Okay, time to introduce you to another conditional jump.
JECXZ - Jumps if, and only if ECX = 0.
Using JECXZ, we can check whether a value is 0 by MOVing it to ECX, since JECXZ does not need a sum to activate!

JECXZ is one of its kind, as there is no JEAXZ, JEDXZ, or anything like that.

Another new useful command: - PUSHes the following registers:
EAX
ECX
EDX
EBX
ESP (original)
EBP
ESI
EDI
This is a rather useful command for making local variables, as it takes only 1 byte of space. It also can be used to save certain registers.
POPAD - POPs the following registers:
EDI
ESI
EBP
(Does not POP ESP)
EBX
EDX
ECX
EAX
The reason it does not POP ESP is because doing that will cause ESP to be incremented after the rest of the POPs.

More new commands:

REP - A prefix for a command, that will repeat the instruction after, and subtracting 1 from ECX until it is 0, and then it will stop. It subtracts 1 from ECX before using the command.

So this

```
PUSH 4
POP ECX
REP PUSH 5
```

^This will PUSH 5 5 times, setting ECX to 0 at the same time.

STOS DWORD PTR [EDI] - Stores EAX to the value at EDI, then adds 4 to EDI

^ This can be used with REP which is a useful way to do a command more than once.

REP repeats the command after unless ECX is 0, where it will stop repeating it.

So the following

```
PUSH EBP
MOV EBP, ESP
MOV ECX, 5600
MOV EDI, 4A6220
XOR EAX, EAX
REP STOS DWORD [EDI]
POP EBP
RETN
```

^ This will delete all entities, because it will set all variables for all entities to 0. It repeats 5600 times, storing 0 to all values in the NPCs' variables.

```
CMPS DWORD [ESI], DWORD [EDI]
```

Compares [EDI] to [ESI]. Increases both ESI and EDI by 4.

```
MOVS DWORD [ESI], DWORD [EDI]
```

Copies [EDI] to [ESI]. Also increases ESI and EDI by 4.

```
SCAS DWORD [EDI]
```

Compares [EDI] to EAX. Increases EDI by 4.

CMPS, MOVS, and STOS also have word/byte forms.

Also, don't use REP NOP to set ECX to 0; SUB ECX, ECX uses the same space and runs MUCH faster. Oh, and by the way, tables also apply for bullets and effects:

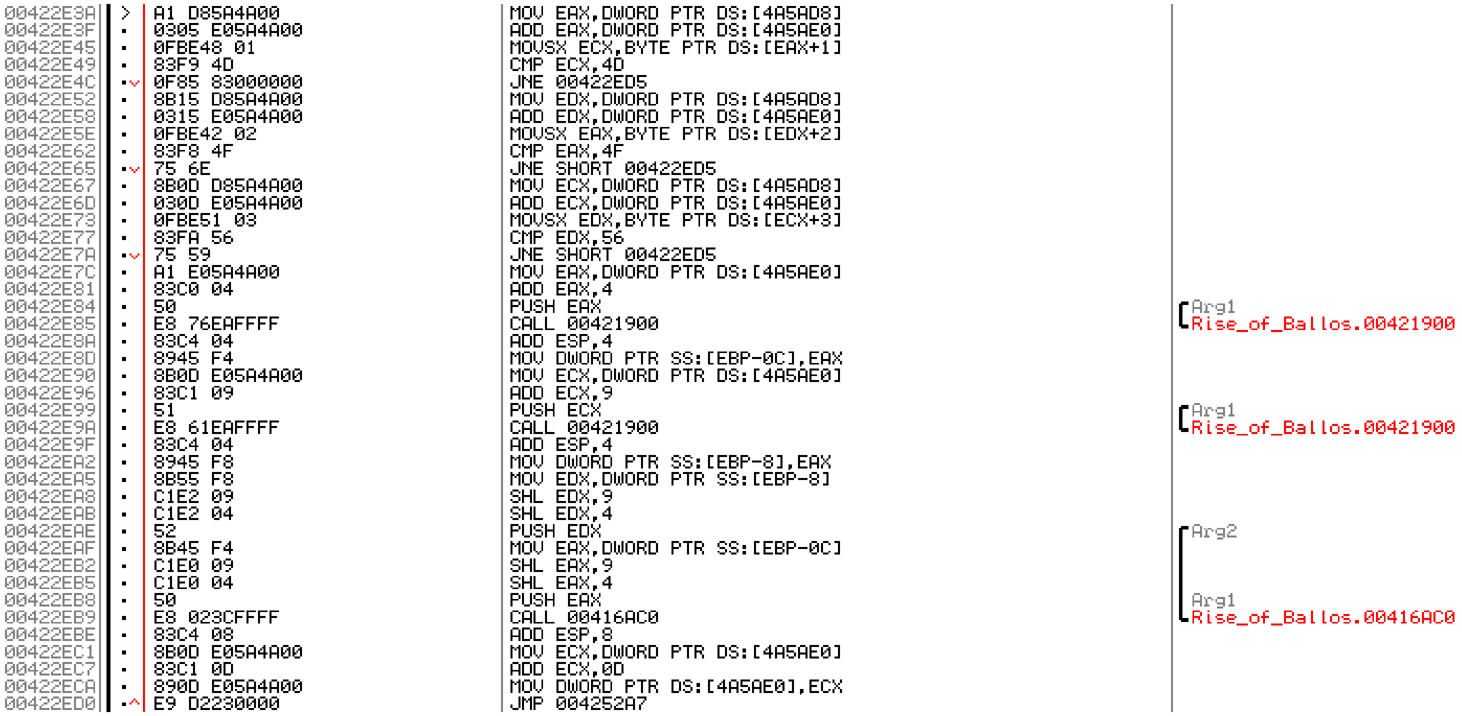
```
PUSH EBP
MOV EBP, ESP
MOV ECX, 800
MOV EDI, 499C98
XOR EAX, EAX
REP STOS DWORD [EDI]
POP EBP
RETN
```

^This will delete all bullets.

TSC commands
Okay, we've had a look at 2 TSC commands, <MOV and <SOU. Now we're going to actually make our own TSC command.
Instead of making one from scratch, we're going to edit a useless one.
When Pixel wrote the game, for some reason he decided to make 2 <FAC commands. We're going to replace one of them to make our command.

Here's how TSC commands are structured:
[4A5AE0], position of script parser (Which TSC script part it's reading)
[4A5AD8], position of current script (Which TSC script it's reading)
CALL 421900 = Getting 4 decimal digits from a TSC file and converting to hexadecimal number
JMP 4252A7 / JMP 4225CB / JMP 42556C = End TSC command

Let's look at a simple TSC command, <MOV.
Didn't we already look at <MOV? Well, the one we looked at (416AC0 if you've forgotten) is the function CALLed by <MOV. Same for <SOU (420640). Sorry I lied, now we're going to actually look into the TSC stuff.



```
00422E3A  > A1 D85A4A00     MOV EAX,DWORD PTR DS:[4A5AD8]
00422E3F  . 0305 E05A4A00   ADD EAX,DWORD PTR DS:[4A5AE0]
00422E45  . 0FBE48 01       MOVSX ECX,BYTE PTR DS:[EAX+1]
00422E49  . 83F9 4D         CMP ECX,4D
00422E4C  .v 0F85 83000000  JNE 00422ED5
00422E52  . 8B15 D85A4A00   MOV EDX,DWORD PTR DS:[4A5AD8]
00422E58  . 0315 E05A4A00   ADD EDX,DWORD PTR DS:[4A5AE0]
00422E5E  . 0FBE42 02       MOVSX EAX,BYTE PTR DS:[EDX+2]
00422E62  . 83F8 4F         CMP EAX,4F
00422E65  .v 75 6E          JNE SHORT 00422ED5
00422E67  . 8B0D D85A4A00   MOV ECX,DWORD PTR DS:[4A5AD8]
00422E6D  . 030D E05A4A00   ADD ECX,DWORD PTR DS:[4A5AE0]
00422E73  . 0FBE51 03       MOVSX EDX,BYTE PTR DS:[ECX+3]
00422E77  . 83FA 56         CMP EDX,56
00422E7A  .v 75 59          JNE SHORT 00422ED5
00422E7C  . A1 E05A4A00     MOV EAX,DWORD PTR DS:[4A5AE0]
00422E81  . 83C0 04         ADD EAX,4
00422E84  . 50              PUSH EAX                            [Arg1
00422E85  . E8 76EAFFFF     CALL 00421900                       [Rise_of_Ballos.00421900
00422E8A  . 83C4 04         ADD ESP,4
00422E8D  . 8945 F4         MOV DWORD PTR SS:[EBP-0C],EAX
00422E90  . 8B0D E05A4A00   MOV ECX,DWORD PTR DS:[4A5AE0]
00422E96  . 83C1 09         ADD ECX,9
00422E99  . 51              PUSH ECX                            [Arg1
00422E9A  . E8 61EAFFFF     CALL 00421900                       [Rise_of_Ballos.00421900
00422E9F  . 83C4 04         ADD ESP,4
00422EA2  . 8945 F8         MOV DWORD PTR SS:[EBP-8],EAX
00422EA5  . 8B55 F8         MOV EDX,DWORD PTR SS:[EBP-8]
00422EA8  . C1E2 09         SHL EDX,9
00422EAB  . C1E2 04         SHL EDX,4
00422EAE  . 52              PUSH EDX                            [Arg2
00422EAF  . 8B45 F4         MOV EAX,DWORD PTR SS:[EBP-0C]
00422EB2  . C1E0 09         SHL EAX,9
00422EB5  . C1E0 04         SHL EAX,4
00422EB8  . 50              PUSH EAX                            |Arg1
00422EB9  . E8 023CFFFF     CALL 00416AC0                       [Rise_of_Ballos.00416AC0
00422EBE  . 83C4 08         ADD ESP,8
00422EC1  . 8B0D E05A4A00   MOV ECX,DWORD PTR DS:[4A5AE0]
00422EC7  . 83C1 0D         ADD ECX,0D
00422ECA  . 890D E05A4A00   MOV DWORD PTR DS:[4A5AE0],ECX
00422ED0  .^ E9 D2230000    JMP 004252A7
```

The first thing you should notice is the immediate adding of the script position and current script. We then get a check of the hex number 4D - Why is that?

Go to a website like http://www.asciitohex.com/, and find what 4D, 4F, and 56 are. Yes, they are actually the characters M O V! Now, we see that the code jumps out of the command if either the characters M, O, or V don't match. Ollydbg's CTRL+E Binary Edit function also can see ascii/unicode values.

Now, you should see that it jumps to 422ED5 (next command) if the TSC command isn't <MOV. This is so that it checks for all TSC commands.

So, if the function is <MOV, we get the following:

Set [EBP-C] to given X value

Set [EBP-8] to given Y value

Multiply both X and Y values by 8,192 (dec) ***(with inefficient code)*** (SHL EAX, 9 / SHL EAX, 4)

PUSH new Y value in 8,192ths of a pixel

PUSH new X value in 8,192ths of a pixel

CALL 416AC0

Reset stack (ADD ESP, 8)

Adds 0xD to TSC parser

End TSC function

Now, if you go back, you'll see that 0x0416AC0 is the function to move the player, among other things such as stopping the player from inspecting an item, and moving the camera.

Now, you can write new TSC commands like that, in the format of the one detailed above. However, the way with checking letters one at a time is inefficient; instead it is easier to check them all at once. So, let's now make a new TSC command, <SNB, that works like <SNP but spawns a bullet instead of an entity!

It'll work like this; <SNBWWWW:XXXX:YYYY:ZZZZ, where W is the bullet type, X is the X coordinate (in blocks), Y is the Y coordinate, and Z is the direction.

Go to 0x424EAF and you'll see a command that checks for <FAC. Because the script checks for <FAC already before, this second check is useless. We're going to replace that check with our new <SNB command.

```
00424EAF    MOV EAX,DWORD PTR DS:[4A5AD8]      Getting current script state
00424EB4    ADD EAX,DWORD PTR DS:[4A5AE0]      Adding to script position
00424EBA    CMP DWORD PTR DS:[EAX], 424E533C   We're checking all of <SNB at once
00424EC0    JNE SHORT 00424F33                 (Backwards)
00424EC2    MOV EAX,DWORD PTR DS:[4A5AE0]      Getting current script position
00424EC7    LEA EBX, [EAX+4]                   New command, explained below
00424ECA    PUSH EBX
00424ECB    CALL 00421900                      Getting first variable
00424ED0    MOV DWORD PTR SS:[EBP-24], EAX     Saving to [EBP-24]
00424ED3    ADD EBX, 5                         Adding 5 to check, to skip the colon
00424ED6    PUSH EBX                           (since it's XXXX:YYYY)
00424ED7    CALL 00421900
00424EDC    MOV DWORD PTR SS:[EBP-8], EAX      Saving second var. to [EBP-8]
00424EDF    ADD EBX, 5
00424EE2    PUSH EBX
00424EE3    CALL 00421900
00424EE8    MOV DWORD PTR SS:[EBP-0C], EAX     Third var. = [EBP-C]
00424EEB    ADD EBX, 5
00424EEE    PUSH EBX
00424EEF    CALL 00421900
00424EF4    PUSH EAX                           Directly using 4th var. to save space
00424EF5    PUSH DWORD PTR SS:[EBP-0C]        PUSHing 3rd var.
00424EF8    PUSH DWORD PTR SS:[EBP-8]          PUSHing 2nd var.
00424EFB    PUSH DWORD PTR SS:[EBP-24]         PUSHing 1st var.
00424EFE    CALL 00403F80                              Spawn bullet
00424F03    ADD ESP, 20                        Reverting all PUSHes before
00424F06    ADD EBX, 5                         Adding to parser position
00424F09    MOV DWORD PTR DS:[4A5AE0], EBX     Moving parser forwards
00424F0F    JMP 004225CB                       End TSC command
```

Alright, new command, LEA. LEA is a combination of ADD, SHL, and MOV. It calculates the sum in brackets and sets target register to it.
So LEA EBX, [EAX+4]
Will ADD 4 to EAX, then make EBX equal to that.
EAX is unchanged during and after the operation.

LEA can also be used to do multiplication in powers of 2.
So you can do the following:
LEA ECX, [EAX*4], and LEA ECX, [EAX*3] (Which converts to [EAX*2+EAX])
But not this:
LEA ECX, [EAX*7]

LEA is also useful to calculate addresses.
You can combine multiplications with additions too:
LEA EAX, [EBX*8+EBX+4B]
^ This is the same as this:
IMUL EAX, EBX, 9
ADD EAX, 4B
Note: LEA also does not change flags, so you cannot use JA, JE, JLE etc on a LEA sum.

Now, for divisions (other than SAR, SHR and ROR).
Divisions are unique (and annoying to do) in the way that they cannot be used with any random old register. The basic CPU division can only divide EDX:EAX by something, and always stores answer in EAX, remainder in EDX.
And by EDX:EAX, I mean by pairing EDX with EAX, so, if EDX = 9A and EAX = 4B3F018, EDX:EAX will be 9A04B3F018. This allows you to divide with larger numbers.
Let’s say we’re dividing 0x100 by 0x7. We’d do:

```
SUB EAX, EAX            Setting EAX to 0
INC AH                  Increasing AH to 1, making EAX 100
SUB EDX, EDX            Setting EDX to 0 using less space than MOV
PUSH 7                  Setting ECX to 7 using less space than MOV
POP ECX
IDIV ECX
```

^ This will store answer (0x24) in EAX and remainder (0x4) in EDX, leaving ECX as 7.

So now we can divide by numbers other than powers of 2!
Isn’t that great!
Also another useful command:
CDQ - Sign-extends EAX to EDX:EAX.

Basically this sets EDX equal to 0 if EAX is positive, or -1 if EAX is negative. EAX is unchanged.
Useful to use this before a division, since it uses less space than SUB EDX, EDX.

FPU operations

If you’ve been messing around with Ollydbg, you might have seen something or some references to something called the ‘FPU.’ So what exactly does this thing do?
The FPU, or **Floating Point Unit**, is (nowadays) part of the CPU in computers, that calculates **floating point**, or **decimal** numbers.
To make it easy for you, Ollydbg even displays the FPU’s registers in base 10!
So how do we use it to calculate sums?
Remember functions 0x4258B0 and 0x4258C0? Remember what they did?
Actually, you probably don’t know what exactly they do. To whoever uses the function, they are approximate sine and cosine function. But in the code, it reads from a table of values calculated using the Floating Point Unit at function 0x4257F0 to approximate (rounded down) values.

Now, a bit more about the FPU.
It’s made up of 8 values, or ‘registers,’ that make up its own stack separate from the CPU’s one. The bottom of the FPU stack is **st7** and the top is **st0**. Each register on the FPU is 80 bits long in data; however the system of storage on the FPU is very different from the hexadecimal system used for the CPU. Most FPU operations **only affect the top** (st0) of the FPU stack.
Unlike the CPU, the FPU also allows certain ‘illegal’ sums, such as 1337 ÷ 0, which, instead of crashing, will simply store the largest value possible onto the target, typically ‘INF.’
Also, since the FPU stack is very limited to only 8 values, instead of crashing, upon stack overflow, the FPU will send an exception error, then pass the exception automatically.
That said, it wouldn’t be a good idea to randomly overflow the FPU stack, as it tends to save the overflowed data as an ‘indefinite’ number, or NAN.

Let’s explore some FPU commands.
FILD - Loads a hex value into the FPU stack into the form of a decimal number. Basically PUSH but for FPU.

FISTP - Translates top of FPU stack into hex integer, then basically POPs it into target memory location. The stored value can then be accessed by the CPU.
Will automatically round off to the nearest integer when storing; NAN becomes 0

FLD - Loads a decimal notation value from memory location directly into the FPU stack. If you want to see an example of a decimal notation value in CS, go to 0x48CA14. That’s an example of a decimal float value, 256.0 in dec, which can be used by the FPU.

FSTP - ‘POPs’ a decimal value directly into a memory location. If you just want a value saved, this is better than FIST because it can store digits after the decimal point.
Warning: The decimal notations in ASM are very different from the hex notations; don’t even think of accessing the value with the CPU without loading it as a proper hex value (FISTP)

FLD1 - Faster way that consumes less space to directly PUSH 1 onto the FPU stack.

FLDZ - Faster way that consumes less space to directly PUSH 0 onto the FPU stack.

FADD - Adds to the top of FPU stack the target memory location operand specified, in decimal notation.

FSUB - Subtracts from the top of FPU stack the target memory location operand specified, in decimal notation.

FMUL - Multiplies the top of FPU stack by the target memory location operand specified, in decimal notation.

FDIV - Divides top of FPU stack by the target memory location operand specified, in decimal notation. If you divide by 0, the max possible value (usually infinity) will be stored.
FIADD, FISUB, FIMUL, FIDIV I don’t need to explain because these are just the above with the source being in hex form. FADD, FSUB, FMUL and FDIV can be used on another FPU register.

FSQRT - Calculates the square root of st0 and stores it in st0. Rooting a negative number will result in NAN.

FLDPI - Faster way that consumes less space to directly PUSH π onto the FPU stack.

FCOM* - CMPs st0 to target decimal value. **Will need some checks.**

FICOM* - CMPs st0 to target hex value. **Will need some checks.**

FSIN - Calculates the sine of st0 and stores it in st0. Angles in radians.

FCOS - Calculates the cosine of st0 and stores it in st0. Angles in radians.

FABS - Calculates the absolute (positive) value of st0

FFREE - Unloads target register; does NOT set it to 0

FRNDINT - Rounds st0 to nearest integer

FSTSW - Saves FPU flags into target register, if used as FSTSW AX followed by SAHF, it copies FPU flags into the CPU flags.

Now, how would we use a FPU function?
Here’s an example.
We’re making a function that calculates the square root of the number PUSHed, then rounds it off and sends it to EAX. So pretend we’ve PUSHed a number already, then CALLed this

```
PUSH EBP                          ;Setting up stack, blah blah blah
MOV EBP, ESP
FILD DWORD [EBP+8]                ;Copying our number to st(0)
FABS                              ;We don’t want negative numbers; we can’t
                                   save NAN into CPU registers
FSQRT                             ;Calculates square root of st0
FRNDINT                           ;Rounds off first (More accurate)
PUSH ESP                          ;Creating CPU variable to save result
FISTP DWORD [ESP]                 ;Saves back into hex form
POP EAX                           ;Loads back into EAX
LEAVE                             ;Destroy stack frame
RETN                              ;Returns
```

This command will now calculate the square root of a number to the nearest whole number, and return it in EAX, so, for example, if we’d PUSHed A29, EAX would be set to 33, because in decimal notation, √2601 = 51.
The FPU is great - The numbers can be easily calculated by humans as we are used to base 10, it can store decimal point float values, it can have all sorts of trig functions and even square roots. So, why wouldn’t you use the FPU for all your calculations?
First of all, the FPU runs quite a bit slower than the CPU. It has many more bits to calculate, so obviously it’d be more efficient to use the CPU rather than the FPU for the same calculation. Also, the FPU calculations typically take more space than the CPU ones, which you probably don’t want.

*If you’re using FCOM or FICOM, you must use FSTSW AX followed by SAHF immediately after. Then you can use conditional jumps; JL, JLE, JG, and JGE are NOT allowed. Using those can cause inaccurate results. JA, JNB, JB and JBE are allowed though. And obviously, JE, JNE, JPE, JPO and the others are allowed. Infinity or negative infinity CAN be compared.
Another thing is you have to use memory for functions, you can’t FILD EAX or something like that.

Just to give you a break, here’s another very (not) useful command.
FNOP - Does absolutely nothing, basically 2 NOPs in 1 instruction.

One more note:
If you’re CALLing a function, and that function uses ECX, for example, doesn’t that modify ECX during the function, so it’d mess up any value in ECX before the CALL?

The answer is yes. However, most functions in most programs, including CS, do not use the registers EBX, ESI, and EDI, so you can still use those registers for storage (Used in functions above).

So, how do you prevent yourself making a function that messes up registers and then calculating them?
If you want a CALL to use the registers but preserve all the registers, there is a way to do that, using the stack, and the command we explored back in page 39.

```
PUSHAD
PUSH EBP
MOV EBP, ESP
……….                Here is whatever the function is.
LEAVE
POPAD
RETN
```

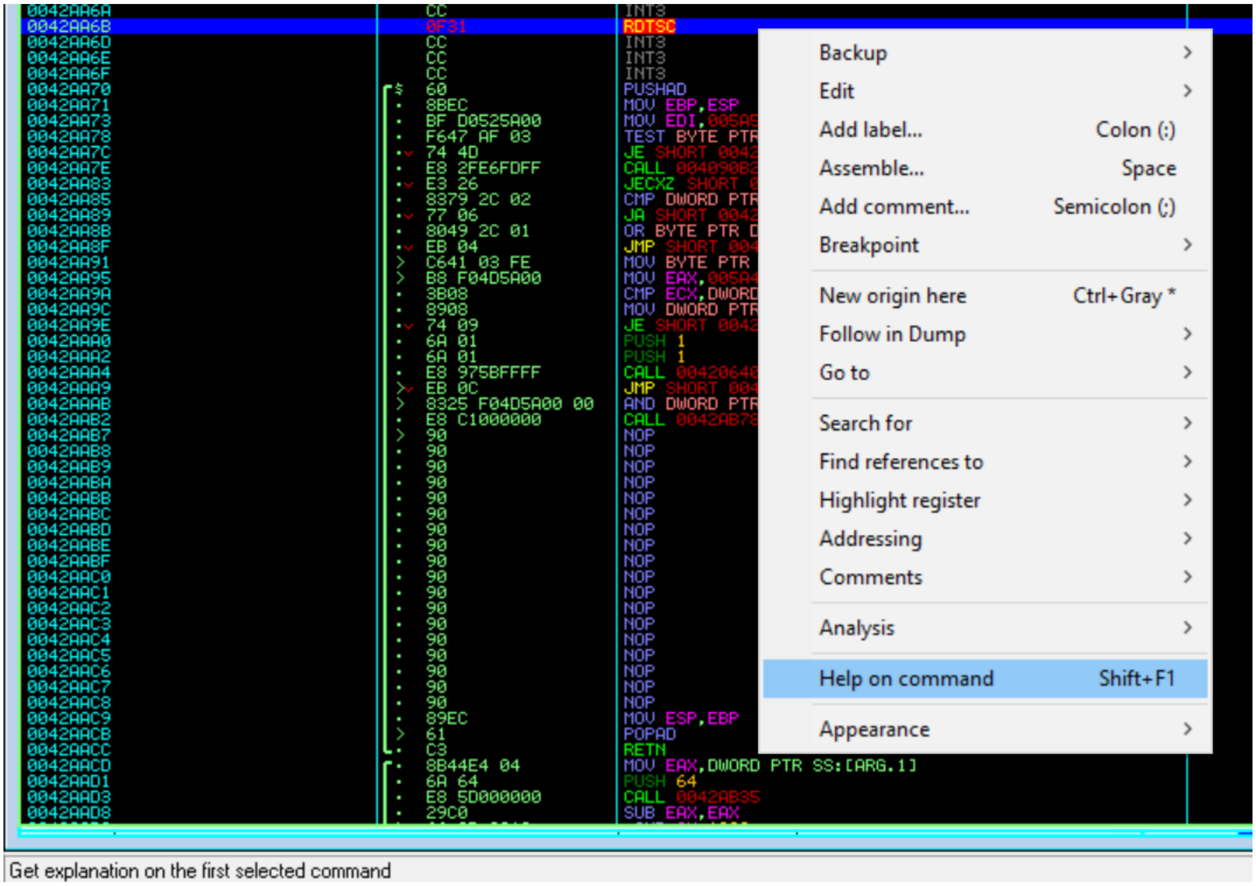
So we’re saving all registers except ESP, which is saved through the use of LEAVE. Therefore all registers remain unchanged.

However, since we changed ESP by using PUSHAD before PUSH EBP, the usual values PUSHed before the CALL are also incremented. Because each register is 4 bytes and PUSHAD uses 8 registers, we subtracted 8 * 4 = 0x20 from ESP before the functions.

Therefore, the usual [EBP+8] is now [EBP+28], the [EBP+C] is now [EBP+2C], [EBP+10] is now [EBP+30], etc.

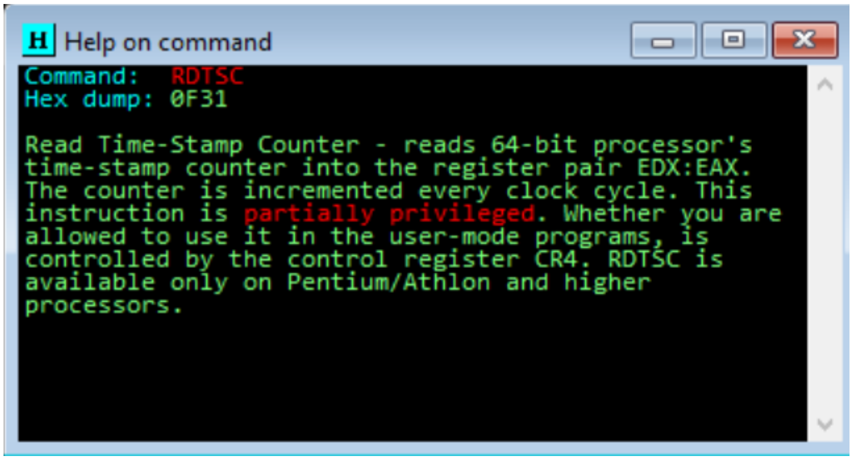
A separate bit of help

Ollydbg supports its own help support for certain assembly commands. If you ever come across a command not explained in this guide (I'd say that's unlikely), or if you've forgotten one of the commands but can't be bothered trying to find it, ollydbg has got your back! To activate command help, simply type the command in, right click it, and click "Help on command." It should give you a fairly accurate description of the selected command. The system seems to be lacking the ability to look up SSE and MMX commands, but as assembly hackers we don't really use those registers (They're mostly for the higher level languages for long data storage and transportation). So anyway, you should be fine with the help tool provided by Ollydbg.



```
0042AA6A      CC                INT3
0042AA6B      0F31              RDTSC
0042AA6D      CC                INT3
0042AA6E      CC                INT3
0042AA6F      CC                INT3
0042AA70   $  60                PUSHAD
0042AA71   .  8BEC              MOV EBP,ESP
0042AA73   .  BF D0525A00       MOV EDI,005A5
0042AA78   .  F647 AF 03        TEST BYTE PTR
0042AA7C   .v 74 4D             JE SHORT 0042
0042AA7E   .  E8 2FE6FDFF       CALL 004090B2
0042AA83   .v E3 26             JECXZ SHORT 0
0042AA85   .  8379 2C 02        CMP DWORD PTR
0042AA89   .v 77 06             JA SHORT 0042
0042AA8B   .  8049 2C 01        OR BYTE PTR D
0042AA8F   .v EB 04             JMP SHORT 004
0042AA91   >  C641 03 FE        MOV BYTE PTR
0042AA95   >  B8 F04D5A00       MOV EAX,005A4
0042AA9A   .  3B08              CMP ECX,DWORD
0042AA9C   .  8908              MOV DWORD PTR
0042AA9E   .v 74 09             JE SHORT 0042
0042AAA0   .  6A 01             PUSH 1
0042AAA2   .  6A 01             PUSH 1
0042AAA4   .  E8 975BFFFF       CALL 00420640
0042AAA9   >v EB 0C             JMP SHORT 004
0042AAAB   >  8325 F04D5A00 00  AND DWORD PTR
0042AAB2   .  E8 C1000000       CALL 0042AB78
0042AAB7   >  90                NOP
0042AAB8   .  90                NOP
0042AAB9   .  90                NOP
0042AABA   .  90                NOP
0042AABB   .  90                NOP
0042AABC   .  90                NOP
0042AABD   .  90                NOP
0042AABE   .  90                NOP
0042AABF   .  90                NOP
0042AAC0   .  90                NOP
0042AAC1   .  90                NOP
0042AAC2   .  90                NOP
0042AAC3   .  90                NOP
0042AAC4   .  90                NOP
0042AAC5   .  90                NOP
0042AAC6   .  90                NOP
0042AAC7   .  90                NOP
0042AAC8   .  90                NOP
0042AAC9   .  89EC              MOV ESP,EBP
0042AACB   >  61                POPAD
0042AACC   .  C3                RETN
0042AACD   .  8B44E4 04         MOV EAX,DWORD PTR SS:[ARG.1]
0042AAD1   .  6A 64             PUSH 64
0042AAD3   .  E8 5D000000       CALL 0042AB35
0042AAD8   .  29C0              SUB EAX,EAX
```

Get explanation on the first selected command



```
Command:  RDTSC
Hex dump: 0F31

Read Time-Stamp Counter - reads 64-bit processor's
time-stamp counter into the register pair EDX:EAX.
The counter is incremented every clock cycle. This
instruction is partially privileged. Whether you are
allowed to use it in the user-mode programs, is
controlled by the control register CR4. RDTSC is
available only on Pentium/Athlon and higher
processors.
```

More on the FPU

When loading a floating point number, it is important to know the type of floating point number you would like to use. The FPU supports the loading and storing of 3 different floating point formats:
DWORD - Single (https://en.wikipedia.org/wiki/Single-precision_floating-point_format)
QWORD - Double (https://en.wikipedia.org/wiki/Double-precision_floating-point_format)
TBYTE - Extended (https://en.wikipedia.org/wiki/Extended_precision)

Since you probably aren't that familiar with the FPU yet, let's just use DWORD for now, since that's the same amount of space a regular int32 variable takes up.

Using the FPU for distance calculation

Ever heard of Pythagoras? No?
(Read https://en.wikipedia.org/wiki/Pythagorean_theorem if you haven't)

So, in a right-angled triangle, the length of the hypotenuse (longest side) squared is equal to the sum of the lengths of the other two sides squared. So **a**^2 + **b**^2 = **c**^2.

How are we going to use it to calculate distance? We're going to take the X distance as **a**, the Y distance as **b**, and the absolute distance as **c**, which we will be calculating. Let's assume you're in the code of an NPC, with the pointer set as [EDI], and you want to calculate the absolute distance from Quote. We'll be setting EAX to 49E654 to save some space.

```
MOV EAX,49E654          #Using EAX as a pointer to [X,Y] player position
FILD DWORD [EDI+8]      #[EDI+8] = NPC's X position
FISUB DWORD [EAX]       #[EAX] = Quote's X position
FMUL ST0, ST0           #Raising distance to the power of two
FILD DWORD [EDI+C]      #NPC Y pos
FISUB DWORD [EAX+4]     #Quote Y pos
FMUL ST0, ST0           #Once again squaring the answer
FADDP ST1, ST0          #Adding the two answers together
FSQRT                   #Taking square root of sum
PUSH ESP                #Allocating 4-byte variable on the stack
FISTP DWORD [ESP]       #Storing into new 4-byte variable
POP EAX                 #Collect result into EAX
```

As usual, distance is in 1/8192ths of a block. Now you can make things like a circular hitbox that damages the player if they're within 1 block of the NPC, by CMPing EAX to 0x2000 (The distance of 1 block), and then PUSHing an amount of damage followed by CALL 419910.

NPC table scanning

Which NPC table am I talking about? Well, obviously not the *npc.tbl* file within CS' data folder. You can edit that yourself without problem.

We're looking at the memory table, or more specifically, the *array of records* that stores each NPC's variables during the game. For example, the [**X**+C] of an NPC stores its Y position. But where is the variable actually located?

It turns out that we can find each and every NPC in game, at any time.

Let's assume for whatever reason we want to make Quote instantly destroy any shootable NPC within 1.5 blocks. Because this concerns Quote and not a specific entity, this function must be placed somewhere in the main game loop.

So, first of all, we'll have to do a linear search through the table. This is done by having loop iterations that start at the start of the table, at 4A6220. Sound familiar? (We looked through the calling function way back on page 20).

```
MOV EDI, 49E654                   #Setting EDI to player location pointer
PUSH 0                            #Creating variable for loop counter
:loopstart
/ POP EAX                         #Getting pointer into EAX for optimized comparison
| CMP AH, 2                       #(AH >= 2); (EAX >= 200)
| JNB :end                        #If EAX >= 200, we've checked all possible NPCs
| IMUL ECX, EAX, 2B               #Each NPC has 0x2B amount of variables
| LEA ESI, [ECX*4+4A6220]         #Each NPC variable is 4 bytes long, table starts at 4A6220
| INC EAX                         #Increment counter
| PUSH EAX                        #Place counter back on stack
| TEST BYTE [ESI], 80             #Don't destroy NPC if it's already dead
| JE :loopstart
| TEST BYTE [ESI+50], 20          #Don't destroy NPC if it's not shootable
| JE :loopstart
| FILD DWORD [ESI+8]              #Same function as before to calculate absolute distance
| FISUB DWORD [EDI]                between player and NPC
| FMUL ST0, ST0
| FILD DWORD [ESI+C]
| FISUB DWORD [EDI+4]
| FMUL ST0, ST0
| FADDP ST1, ST0
| FSQRT
| PUSH ESP
| FISTP DWORD [ESP]
| POP EAX
| CMP EAX, 3000                   #Don't destroy NPC if distance is greater than 1.5 blocks
| JG :loopstart
| PUSH ESI                        #Destroy NPC by calling the "Kill" function
| CALL 471B80
| POP ESI
\ JMP :loopstart                  #Check next NPC on the list
:end                              #Finished
```

If you're able to put this somewhere near the end of the main game loop (410400), when you play the game, Quote will instantly kill any enemy within 1.5 blocks of him, even most projectiles! Probably not very practicable because of how overpowered that mechanic is, but at this point you should be good enough at ASM to turn it into something useful, such as having it being activated by a certain flag, and destroy NPCs with 1 health or lower as a kind of shield mechanic for example.

Homing bullets

An enemy projectile targeting the player was easy enough wasn't it? Just use 4258E0, 4258B0 and 4258C0 and you're done!
But what if you want the player to shoot a bullet that follows enemies? We're going to need a table similar to what we did on the last page, except this time we're only going to read coordinates off the table, we won't be destroying any NPCs directly.
We'll be making the bullet move towards the closest NPC within an 8-block radius that is shootable, and if its angle is within 45 degrees of the bullet's current vector.

```
MOV EDI, [EBP+8]                #SetPointer
PUSH DWORD [EDI+1C]             #Using bullet's X and Y velocities to calculate current
PUSH DWORD [EDI+18]             #angle of flight
CALL 4258E0
XOR AL, 80                      #Reversing angle so that we get the angle towards
PUSH EAX                        #Placing angle on stack
PUSH 10000                      #Placing 10000 (8-blocks) on stack
PUSH 0                          #Placing loop counter on stack
:loopstart
/ POP EAX
| CMP AH, 2                     #0x200 NPC limit as usual
| JNB :end
| IMUL ECX, EAX, 2B
| LEA ESI, [ECX*4+4A6220]
| INC EAX
| PUSH EAX
| TEST BYTE [ESI], 80           #Don't target dead or unkillable NPCs as usual
| JE :loopstart
| TEST BYTE [ESI+50], 20
| JE :loopstart
| FILD DWORD [EDI+10]           #Calculating distance between bullet and NPC
| FISUB DWORD [ESI+8]
| FMUL ST0, ST0
| FILD DWORD [EDI+14]
| FISUB DWORD [ESI+C]
| FMUL ST0, ST0
| FADDP ST1, ST0
| FSQRT
| PUSH ESP
| FISTP DWORD [ESP]
| POP EAX
| CMP EAX, [ESP+4]              #If there's already a closer valid NPC found, ignore this
| JNB :loopstart                 one
| MOV [ESP+C], EAX             #Otherwise, temporarily save the distance
| MOV ECX, [EDI+10]
| MOV EDX, [EDI+14]
| SUB ECX, [ESI+8]
| SUB EDX, [ESI+C]
| PUSH EDX
| PUSH ECX
| CALL 4258E0                  #Angle of elevation to NPC
| POP ECX
| POP EDX
| MOVZX ECX, AL                #Making two copies of angle, one sign-extended due to the
| MOVSX EDX, AL                #way angles work, a full circle may cause some angles to
| SUB ECX, [ESP+8]             #not be found
| JG :positive1
| NEG ECX
| :positive1
| CMP ECX, 20                  #0x20 = ⅛ of 0x100 which is ⅛ of 360 degrees, = 45 deg.
| JLE :inrange
| SUB EDX, [ESP+8]
| JG :positive2
| NEG EDX
| :positive2
| CMP EDX, 20                  #Checking if any were missed because of int overflow
| JG :startloop
| :inrange
| MOV [ESP+10], EAX            #If angle and distance valid, save newfound entity over any
| MOV EAX, [ESP+C]             #existing found ones
| MOV [ESP+8], EAX
\ JMP :startloop
:end                           #Note: ESP increases by 4 since loop counter disappears
CMP DWORD [ESP+4], 10000       #If distance still = 8 blocks, no valid entities found
JE :nonefound
MOV ECX, [ESP+C]               #Otherwise, use the found angle to calculate a vector
PUSH ECX                       #towards found target
CALL 4258B0
XCHG EAX, EBX
CALL 4258C0
```

If no valid entities are found, the code will redirect to :nonefound, which should be placed after you use the variables. EAX is the X component and EBX the Y component of the vector to the entity found, if any.

That’s all for now!
If you have any questions, you can find me on discord at the CS Modding Community at https://discord.gg/xRsWpz6.
Happy hacking!

16th February Smudge invaded and once again returns to place her flag >:3 **places flag and conquers this doc** | 🙃