The GOAT Source Official Documentation
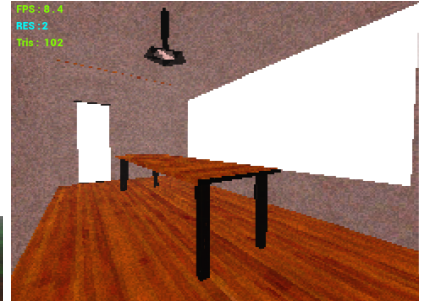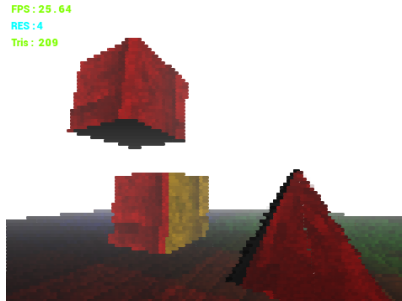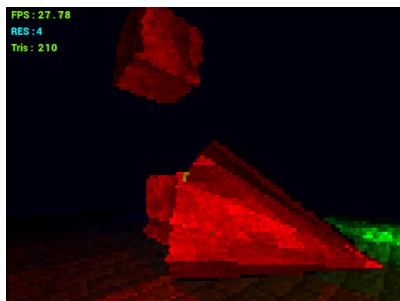
A programmer's guide to the insanity of the GOAT 3D Engine from Beginner to Advanced!

FPS:27.78
RES:4
Tris: 210

FPS:25.64
RES:4
Tris: 209

FPS:0.4
RES:2
Tris: 102

I hope you can learn from this!

Table of Contents

Introduction

If you're wondering what the GOAT Source 3D Engine is, it's a 3d engine; right there in the name! It's mainly scripting based (for now) and has many features out of the box.

The purpose of this document is to guide you through the "alien" code of the level format and scripting language, to importing obj models, and of course, making your first game!

Prerequisites:

Right now I haven't gone around making blender from scratch (in Scratch), so you'll just have to use a 3D modeling program that can at least export in the [.obj Wavefront File Format].

I suggest using blender, as it's free. You can either search up Blender or go through to the very bottom of this document and find the direct link to Blender's download page (if you're insane).

You'll need to be armed and ready with some sort of text editor. Just about everybody already has one, but if you can, use Visual Studio Code. It has syntax highlighting, and I have taken my time making a custom syntax highlighter for my custom language, as long as you rename the .txt extension as .val, and go to the bottom where the links are and download the Vector Assembly Programming Language.
It doesn't have Intellisense for the custom language, though. Also, I didn't want to make another syntax highlighter for the level data, so you'll just have to suck it up and see bleach white text for the level. And finally, you might want some sort of image editor for making textures. I suggest GIMP, as it's free and open source and all that nice shenanigans. But make sure your textures are really small, like 32x32, 64x64, 128x128, 32x64, 64x128, ect. If you're making an atlas, it's okay to have bigger sizes. Also, I suggest powers of two in size, as everybody loves them!

Purpose

Why did I make this engine? I dunno, it sounded fun. Why in scratch? Just because. Why did I make a custom scripting engine as well as a custom level script?

1. Easier Collaboration. There's more options to collaborate on a file than on scratch. Plus, I've used up most of the available space on everything else.

2. I like torturing people by making them learn a weirder version of assembly as well as trying to build a level from a file. Nah, I'm kidding. It's a learning experience. It also builds character and you can tell your friends that you learned assembly!

3. The editor lags and stuff when I use lots o' blocks for the level and trying to script objects using broadcasting and stuff is a lot.

4. I'll be able to then make a node-based visual scripting thing in the game engine editor for my 3D engine. It's also going to pay off when I can edit the level visually and have it compile to the files.

Why assembly for scripting? It's easier than having to build a proper language with an AST and traversing the tree one node at a time. Also, assembly's easier than you think.

Why don't you make normal scratch projects?
No. You can't make me. I like math, and it makes me think, which I like to do.

Terms and Abbreviations

~~Actually, I can't think of anything.~~
Vec3 : vector with 3 components, (x,y,z)
Vec2 : vector with 2 components, (x,y) (we might not use this much)
Val : vector assembly language
Why? : why.
Tris : triangles
Texture : an image applied to a model for it to look a certain way.
Rasterizer : an important part of rendering that draws the newly projected tris with textures and color.
pos x : position x
pos y : position y
pos z : position z
px : pos x
py : pos y
pz : pos z
rot x : rotation x
rot y : rotation y
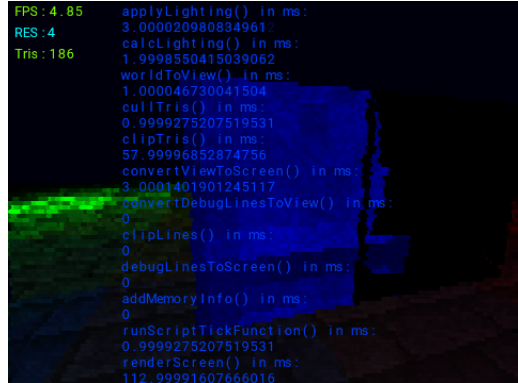rot z : rotation z
rx : rot x
ry : rot y
rz : rot z
sx : scale x
sy : scale y
sz : scale z

Brief Overview

Anyways, When you boot up the project (IN TURBOWARP),
You will start off with an example game. It's not really a game
as it is a demo, but anyways, you start off in the main level.

[Main level running on a chromebook]

If you want to open any debugging or window, you have to hold [`] and press any number key, [1,2,3,4,5,6,7,8,9,0]. Pressing [2] with [`] down gives you the entire frame stats and each rendering stage timed in milliseconds.

As you can see, the renderScreen() function takes up the most milliseconds. This is what renders each triangle by the rasterizer.

Somehow, some of the functions take up 0 milliseconds! The clipTris() function takes up 50 milliseconds, which makes sense, as around 18% of the ground is behind the player, and because of lighting limitations, the ground has to be a grid of vertices for there to be any lighting resolution.
But around 180 - 210 tris is a lot to even work on a chromebook!

Now if you look in the top-left corner of the screen, you can see that there's the FPS counter, displaying a glorifying 4.85 fps on a chromebook for a 3d engine in turbowarp with textures, a res option, which if you click on the number, you can type in a new number and after clicking away again, sets the new value for the res variable!
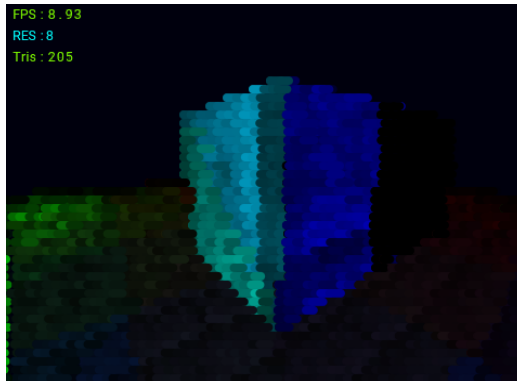
Boring Math Words and Stuff

Part I: essential gamedev knowledge

Okay, so to understand how any 3D coordinate system works, we must look back on how 2D coordinates work.

This is a coordinate: (0,0). Most notably, the origin, or center of everything.
I also plotted (5,5). As you can see (and hopefully remember), coordinates start at x, then y. That's why in algebra, you commonly see (x,y), because you have to usually solve for x and y. X comes first, then y.

					y					(5,5)
					(0,0)					x
	(x,y)									
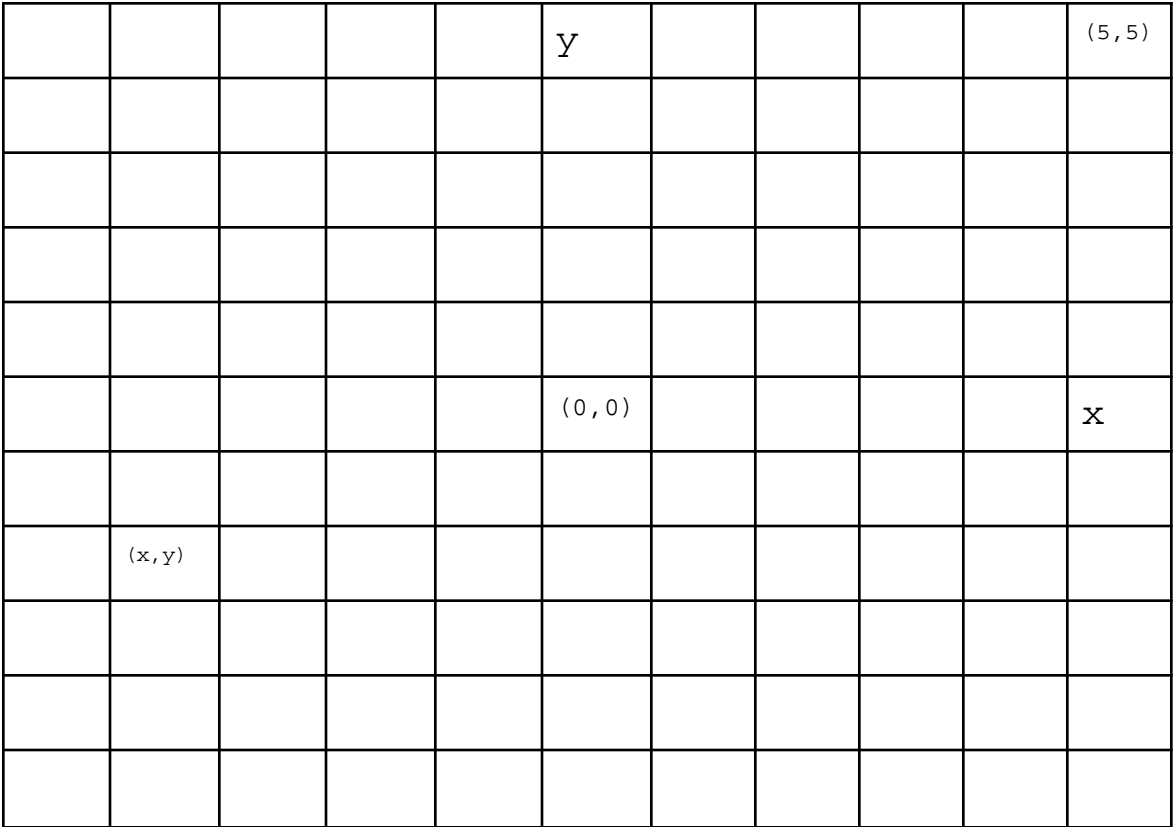
$|x,| = |-4|$

$|y| = |-2|$ Anyways, now you understand how coordinates work. Now imagine a z component! Dun dunnnnn, DUNN!!! (x,y,z) it's not that surprising that if y comes after x, then z comes after y. But now this lets us traverse the 3D space!

(taken from desmos 3d)
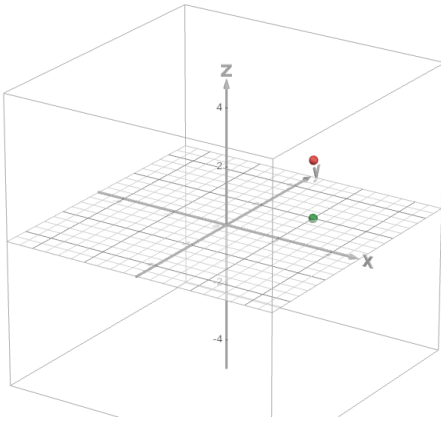
Now you might notice, z is up/down! Yea, it is kinda confusing, if you think 2D is in an orientation where y is up, but some 3D programs use z as up/down and y as depth.

Well, in this engine, imagine y is up/down, and z is depth. Makes more sense, right?

Here, the coordinates are: red = (2,2,2) and green = (2,2,0).

Now, for my engine, it will be:

Red' = (2,2,2), and green' = (2,0,2). This is basically one way to represent a 3d vector. They're just coordinates.

There is another way where you add vectors or something if they're notated with a -> at the top of the vector variable. Oh yeah, I forgot. In math, you can store entire vectors as a single variable, kinda.

For example:

$a = (18, 2, 54), \hat{a} = (0.316033, 0.0351147, 0.948098)$

Don't worry about the ^ on top, that just notation for normalizing the vector. Sounds complicated, right? Kinda, not really that complicated. It makes sure that the vector points to a direction inside a unit sphere or something. If it goes past that, it stays inside the sphere. If it's inside the sphere, it makes sure that the vector stays on the sphere, if you know what I mean. Anyways, don't worry about that, that's just me rambling on again.

What else? Oh yea, you'll need to know how to get the distance between two points! It's not that hard, most vector libraries have a vec3.dist(a,b) function which gives you a number of how far each vector is to each other. If you really wanna know the formula it's this:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} = return$$

Kinda complicated, right? It's just pythagoras theorem. You'll learn it someday.

Here are some pretty helpful things for vector math:
1. The cross product. I won't delve too much into how it works, but a good visual representation goes like this:
   a. take out your left hand and put out your thumb so it points up.
   b. put out your pointer finger to point forwards.
   c. put out your middle finger so it points to the right.

What you have seen is the left-hand rule for the visual representation for the cross product. If you rotate your wrist around, you can see that all your fingers also rotate the same to your wrist.

Now imagine that vector a is your pointer finger, and vector b is your middle finger. It should now be obvious that your thumb is the cross product of the vectors a and b, which is notated as $a \times b$.

2. The dot product. These both go hand in hand, except the dot product returns a number, indicating the angle between two vectors. When the vectors are pointing in opposition to each other, the value is -1. (yes, not 180, but -1). If they're in the same direction, the value is 1. Anyways, you can find a better explanation in the links below.

Oversimplification on Computer Architecture

Okay so here goes: Most computers have a CPU, RAM, a motherboard, and sometimes, a GPU. There's obviously a lot more, but for now we'll only need to focus on CPU and the RAM.

In the CPU there's a thing called the Arithmetic Logic Unit, or the ALU. Obviously, this does the math. CPUs themselves can't store many values, but they have these things called registers. registers hold single variable values, except that registers themselves have a predefined name.

For example, x86 processing chips have register names, typically stuff like EAX, EBX, ECX, EDX, ect. Some registers have a set purpose, like there's one called an "accumulator register". Don't ask me what that means, it might be an index for loops or something. Anyways, to be able to store more than just 8 variables or something, there's a chip specifically designed for storing memory to be accessed later.

This is called RAM. There's also hard drives and stuff, but you don't use that in programs except for reading and writing files. This basically has an array of many values that could be easily accessible. Instead of working with 8 variables, you can also put many variables into the RAM for later use.

Mainly for using the RAM, there's two ways to store such variables, the stack and the heap. The heap's very complicated and I won't explain it, but it's what stores resizable multi-sized variable arrays in a program, but is slower than the stack. The stack is easy to understand so here's how it works:

Imagine a stack of pancakes on a plate. A normal non-psychopath would grab the pancake on the top. And when the chef puts a new pancake they'd put it on the top. Pretty straightforward, right? Now imagine pancakes as variables, and grabbing them means getting the value of the stack, and putting a pancake on the stack means putting a variable on the stack. That's basically what a stack is. When a new value is added, there's a specific register in the CPU that moves so it points to the next empty element in the stack, which tells the program where to put the next value. I probably gave you lots of misinformation or I botched up the explanation on how a computer really works, but this is really all you need to know how assembly, human-readable machine code works. kinda.

Oversimplified Assembly Basics

```
section .text

    global _start

_start:
    mov rax, %10
    mov rbx, %21
    mov ebx,0
    mov eax,1
    int 0x80
```

     Insanity, right? I wrote this program which most likely gives me some sort of error. But this is a (bad) example of assembly! It does look like an alien language, (please don't go by the intimidation of assembly) but it's actually pretty simple.

     When the program starts, it goes to the part called section .text. Then it finds the part where it tells the program to start at _start. This is needed for some reason. Now before we try to know what mov is, or even int, here's the definitions for basic assembly operations:

     a, b (a is always first)

     mov : reg b = reg a

            or

     mov : reg b = (%)value

     add : reg b = reg a + reg b

     sub : reg b = reg a - reg b

     mul : reg b = reg a * reg b

     (lol I forgot what int does again)

     I hope this helps you in knowing what the operations do. Depending on what variant of assembly this is (yes assembly depends on the architecture of the computer and processor), you might also have the mov (short for move) to have the return register be on the right and the value on the left.

     Add is short for add, and like it suggests, you set the return register to the first register plus the same return register. sub is the same as add, except it subtracts instead of adds. Likewise, mul is the same, except it multiplies instead of subtract, or add.

Level Editing Crash Course

Start by creating a new text file. This will be the level data.

```
Level Startup
module LoadLevel
ports:
{
    "ExampleCube",
    [0, 0, -100]
    [0, 0, 0]
}
End Level


Level ExampleCube
Mesh Cube : Thing "the cube thingy"
0 0 0
0 0 0
2 2 2
tint : 1 1 1
texture : brick
uvscale : 1 1
collision : false
rayCollision : true
End Level
```

Okay, onto the actual meat, not only chewy fat. So, this is an example level file for a simple cube scene, positioned at (0, 0, 0). It's also rotated in the directions, (0, 0, 0) or forward. It's also scaled up by factors on the x, y, and z by (2, 2, 2) respectively. How did I get all that?

Do you see the three lines with three numbers separated by spaces after the line "Mesh Cube…blahblahblah"? This is called a transform. Most game engines, like Unity, or Unreal Engine have one. [Unreal Engine's Transform example]
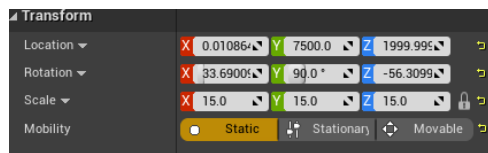
Though Unity's rotation has Quaternions, we're using euler angles, x, y, and z. no w, 'cause it's easy to learn and implement. It's not that hard to learn rotation, as you can think of it like Pitch, Yaw and Roll. If you know about airplane control, this should be second nature. If you're average however, I'll break it down for you:

Pitch : Nod up & down. That is rotation on the X axis.

Yaw : Shake left & right. That is rotation on the y axis.
Roll : Tilt your head around, like you're confused. That is
the z axis. What about the scale? What is that? So imagine a
cube that encases over every object. The scale on the X is
Width. Imagine as you increase that, the mesh deforms in that
axis to fit to the cube. The scale on the x and y axis, then, is
height and depth respectively. If you still don't know what I
mean, then imagine you're stretching an object along the x, y
and z axis. That's the scale.

There's also extra data for specific types. For example,
the mesh has properties: tint : r g b, texture : textureName,
and uvscale : u, v. The properties depend on the type. Also be
careful of order, that's important!

Here's a table:

Mesh:	Pointlight:	Box collider & trigger:	[When updates come out, I'll be sure to add more types.]
tint : r g b Texture : textureName uvscale : u v collision : (bool) rayCollision : (bool)	tint : r g b intensity : i radius : r attenuation : a falloff : f	rayCollision : (bool enabled?)	

The Humble Thing

Okay, I forgot to mention: What is a "Thing"? If you've used or even seen unity or unreal or godot, you've probably seen terms like unity's GameObjects, unreal's Actors, or Godot's nodes or something like that. In this engine however, it's a Thing. Basically, a name for an object. You ALWAYS have to append : Thing "insert custom name here".

The Thing is what stores the transform in 3D space:
px py pz
rx ry rz
sx sy sz

That determines the object's state in 3D. There's nothing else to say about it so moving on.

The Simple Mesh

Okay so here's the level data again:

```
Level Startup
module LoadLevel
ports:
{
    "ExampleCube",
    [0, 0, -250]
    [0, 0, 0]
}
End Level

Level ExampleCube
Mesh Cube : Thing "the cube thingy"
0 0 0
0 0 0
2 2 2
tint : 1 1 1
texture : brick
uvscale : 1 1
collision : false
rayCollision : true
End Level
```

If you haven't copied this to try it yourself, I suggest doing so. Anyways, if you've noticed the identifier, "Mesh", you can also see that the name "Cube" is after that. That's the actual mesh name that references it in the …/content/source file. If you don't know where to find it, don't worry.

I forgot that you have to press [`] + [6] to toggle the window to the three files: …/content/source, …/content/levels, and …/content/scripts. Yea, that's essential to actually being able to create anything. Oh well.

Once you find it, download all three files and open it with your text editor of your choice. If you have Visual Studio, and my .val Syntax Highlighting extension installed, be sure to rename the scripts' file extension to .val. There are many example scripts, levels and meshes already pre-included with it, such as the cube we mentioned earlier. Anyways, the mesh is simply a 3d model that you can make in an external program, such as blender. If you dig around a bit, you can find other meshes in the file. Like there's also a sphere, cube, ground, ect.

How to import a mesh

Okay, so now we have some basic knowledge about:
1. What a transform is
2. How to open editor windows
3. What a mesh is
4. How 3d coordinates and vectors work with some basic vector knowledge, and more!

So now I will tell you how to import a mesh. In your 3D modeling software of your choosing, make sure to export your model as .obj Wavefront File Format. Then in the options, make sure -Z of forward, and Y is up. If you don't have .obj as an option, either find a way to export it as .obj, or use Blender.

Then, in the engine, press [`] + [8] and load up the .obj mesh file, and the .mtl material file. I hope you haven't lost either files, 'cause they're both important. Then, press enter. It will then import the obj file and then return a message, saying imported.

One thing that you need to know before exporting the obj from your 3D modeling program is that you HAVE TO MAKE SURE IT'S TRIANGULATED. In blender you can add the triangulation modifier, and in other programs, I dunno. I've never used other 3D modeling programs.

Anyways, then you have to right click on the mesh file and click import. Then, you have to make sure that it shows all files, and then click on the mesh file. Then, do the same for the material list, but for the material file, and then, stop the project. This will ensure nothing bad happens while the game is running. Then, press enter.

You will be asked the name and scale of the mesh. I suggest a scale of 50 or 100 on all axes, as the player moves a bit fast. If the import was successful, you should see that the lists have been discarded and a message in place of the mesh list saying "imported". There ya go! You've imported a mesh!

Colliders and Triggers

Once upon a time, or something like that, there was 2D graphics. This was upon the oldest of times, all the way back sometime ago there were 2D collisions. Back then, they only really used square triggers and collisions.

So let's start with the collider. What does it do? Well, by default, the engine has a player camera built-in, which means that you automatically have a first-person player controller when you boot up the game! So instead of having to code it, you can worry about other things.

This means that when you place a collider in the level, when the player collides with the collider, it keeps the player from moving through the collisions, like how real life works (It's been awhile since I've seen real life though).

The official collider types are: Box, Sphere, and Capsule. I might add Cylinder next. Rotation doesn't work, 'cause that adds unnecessary complexity.

So what is a trigger then? A trigger is the same as a collider but in the sense that it's used for triggering events when the player's inside it. A trigger is basically an invisible section that does something, like run some piece of code once triggered by either the player or something else.

Actually, speaking of code…

Scripting in Vector Assembly

Purpose

Okay, so what's the point in a game engine that can't run code you just programmed? So I decided to let the user suffer in a less annoying way: I made a custom language based on "human-readable" machine code. The more annoying way is having scratch's editor lag and making the user figure it out themselves, only to not have their project save 'because scratch has a 5MB limit on the code file size. All you have to do for "convenience" is to learn my own personal hell of a language!

Changes I

Here's a basic example of a program that prints the value of a register that increments by 1 each frame:

```
Script Test:

    /* This is a test. It adds 1 to a register and prints it to the console. */
    /*help*/
    fn Init:
        mov 0 lr1
        mov 0 lr2
    End Init

    fn Tick:
        add 1 lr1
        print lr1 lr2
    End Tick

End Test
```

(Wow the copy and paste also had highlighting, text color, ect that looks like it came straight out of Visual Studio Code!)

If you don't have syntax highlighting, either download and install the custom syntax highlighter I made on github (links at the bottom) into the extensions of vscode, or check if the extension name is .val, and if not, change it.

Anyways, you can see some noticeable changes:

1. A new script starts with Script [name] and ends with End [name]. This is different from the level or source data as it doesn't end in name after the "End" keyword.
2. there's two functions: Init and Tick, which you don't specify the whole _start: thing, but it's kinda similar, just better.
3. Instead of eax, ebx, ect. there's lr1, lr2, ect.
4. Things slightly make more sense.
5. The console print function is actually labeled "print" like in python.
6. numbers don't start with % this time.

So how does this make things any better? Well, this is more suited towards 3D, and game engines, so you can see that like unreal or unity, you put the stuff that runs at the start when the level is loaded at the Init function, and the stuff that you want to run each frame is in the Tick function.

Changes II

Okay, but what does this do? Like, nothing makes sense.
So, right now, if we paste these lines into the level (before the End Level tag),
module Test
ports:
{
}
And when we boot up the level, nothing happens. Then, if you didn't know, press [`]+[3] to open the actual dang console. You will then see a bunch of values each frame, that all are bigger than the last by 1.

Depending on whether you placed the module right after the mesh data or before any meshes after the Level ExampleCube tag, the script will either have a thing be it's "parent" or not. We will come back to this later, but for now, let's see just what this script does.

Here's what we actually need to see:

```
fn Init:
    mov 0 lr1
    mov 0 lr2
End Init

fn Tick:
    add 1 lr1
    print lr1 lr2
End Tick
```

Let's start with the Init function. Notice how the register that gets the actual return value during the move, add, subtract, multiply, and in this variation, divide (or div for the operator name), is on the right, not the left. This is important to remember as it does get slightly confusing at times.

When the script gets initialized, it sets the value of register 1 to 0. By default, registers have it set to "loveisdead". It also sets register 2 to 0.

Next, in the Tick function, when the script runs in the main game loop, it adds 1 to the value of register 1, then stores it. Finally, it outputs the value of register 1 as text to the console, with a hue of the value inside of register 2, which is 0, giving it a red color.

Syscall/Operand Table

Before you can begin programming in Vector Assembly, you'll need to know the basic functions and operators that are used to do things. Here's a table of all the functions, operators, and types that you'll need to know:

mov a b	sets the value of b to the value of a. A can be a register name or a literal value. multi-word strings not supported. You can use ports for strings.
add a b	adds the value of a and b and stores it inside register b. a can also be a literal number as well as a register name.
sub a b	similar to add, except that it's subtraction, not addition.
mul a b	similar to sub, except that it's multiplication, not subtraction.
div a b	A new operand not seen in most forms of computer assembly, as it was seen as a slow operation. Similar to mul, except that it's division, not multiplication.
vmov V_a V_b	All operands and functions that start with the letter v are reserved only for the vector3 registers. These are for doing math with vec3 components (x, y, z). This sets the values of vector register b to the values of vector register a. Oh, and I forgot, V_a V_b can also be object references for getting and setting transform elements of the object. Like vmov thing:transform:position V_b. This sets vector register b's value to the position of the object the current script is attached to. For more information, look at notes below the External Links and References for the Notes.

vadd v_a v_b	Similar to add except it's for vector registers and it also adds all components of vector register a with vector register b and thus stores it in vector register b.
vsub v_a v_b	Similar to vadd, except it's subtraction, not addition.
vmul v_a v_b	Similar to vsub, except it's multiplication, not subtraction.
vdiv v_a v_b	Similar to vmul, except it's division, not multiplication.
vcross v_a v_b	So if you remember, we talked a bit about the cross product, but that's what this does. It finds the cross product of vector register values a and b and then stores it in vector register b. If you don't know the cross or dot product, you can find links to cross and dot at the bottom.
vdot v_a v_b, c	The dot instead returns a number, so we'll have to introduce another parameter at the end that gets the return of vector register values a dot b and stores it in the normal register c. I've been notating the vector param as v_x and you've probably known.
vrot v_a v_b	This takes in the first vector register's value and rotates it around the vector register 2, using x, y, and z as pitch, yaw, and roll. Then, as always, it stores the result in vector register b.
vsplit v_a b c d	This is a way for you to convert vector register values into three separate registers. It takes the x of a and puts it in the register b. It then does the same but for y and uses the register c, and finally, the z goes into register d.
vmerge a b c v_d	This is a way for you to convert three separate register values into a

	single vector register. It sets the components, x, y, and z of vector register d as the values from registers a, b, and c respectively.
vecx v_a b	This gets the value from the x component in vector register a and stores it into register b.
vecy v_a b	This gets the value from the y component in vector register a and stores it into register b.
vecz v_a b	This gets the value from the z component in vector register a and stores it into register b.
vray v_a v_b c d e f v_g v_h	Oh boy, this one's hard! Takes in a ray position as vector register a, a ray direction as a vector register b, a max ray distance as c, and returns to registers, d, e, f, V_g, and V_h. Register d holds whether there was a hit at all. Register e will hold the name of the object it hit. Register f will hold the hit distance from the ray origin. Vector register V_g will hold the world position of the hit event, and finally V_h holds the hit normal.
Level load a	So this command actually has two keywords, and 1 register, a. As you can see, and probably guess, it loads a level with the name being the value of register a.
@[Custom label name here]/[script name here]	This is a custom label. You will use this when you start adding branches, jump commands, delays, ect. An example of this is: @HelpMe/Rigidbody. The label name is HelpMe, and the script that the code is in is called Rigidbody.
Delay a @b	We will notate label references with an @ before the label name. There's also no /[script name] after the label reference. In practice, there's also no @ before the label reference; I only notated label references as that. Also it can be a register, but most of the time you'll use the actual label

	name. Anyways, as you can guess, this delays execution and while it's delaying, it jumps to the label as @b's reference. Then it continues to go down the execution below it. Until it delays again. Just like ue4's delay node. This is in beta testing, so I'm not sure if it works properly. I also suggest making the @b label jump to the next delay underneath if you have one.
killAll	Just a simple command. It stops the program, by terminating the program.
Math sin a b	Calculates the sine of the register value of a and returns to register b.
Math cos a b	Calculates the cosine of the register value of a and returns to register b.
Math tan a b	Calculates the tangent of the register value of a and returns to register b.
Math abs a b	Calculates the absolute of the register value of a and returns to register b.
Math floor a b	Calculates the floor of the register value of a and returns to register b.
Math ceil a b	Calculates the ceiling of the register value of a and returns to register b.
Math asin a b	Calculates the arcsine of the register value of a and returns to register b.
Math acos a b	Calculates the arccosine of the register value of a and returns to register b.
Math atan a b	Calculates the arctangent of the register value of a and returns to register b.
Math ln a b	Calculates the natural logarithm of the register value of a and returns to register b.
Math log a b	Calculates the logarithm of the register value of a and returns to register b.
Math epow a b	Calculates the power of the register value of a to constant e and returns to register b. Formula: e ^ a.
Math tnpow a b	Calculates the power of the register value of a to 10 and returns to register b. Formula: 10 ^ a. I probably didn't test any of the extra math functions, though. I'm sure they work.
Cmp a b	Compares the value of register a the value of register b. The carry flags are:

	a > b a < b a = b a >= b a <= b a != b
Vdist v_a v_b c	This gets the distance between vector register value a and vector register value b, then returns to register c.
Vlen v_a c	This gets the length of vector register value a, then returns to register c.
Cast a b	This gets the index of the object name of register a and returns to register b. If the object wasn't found, it prints an error to the console, and immediately stops.
Mod a b	I forgot to include this somehow, but basically, it gets the remainder of b / a and returns to register b. If you don't know what that means, it just wraps the number around the second number, so it doesn't go over the second number.
Print a b	Prints the text from the value of register a with the hue of the value of register b to the console.
And a b	If both register values of a and b are booleans (0 or 1, true or false), then if performs the and operation of a AND b and then returns to register b.
Nand a b	If both register values of a and b are booleans (0 or 1, true or false), then if performs the and operation of NOT (a AND b) and returns to register b.
Not a b	If register a is a boolean, it finds the opposite of a and returns to register b.
Or a b	If both register values of a and b are booleans (0 or 1, true or false), then if performs the and operation of a OR b and returns to register b.
Xor a b	If both register values of a and b are booleans (0 or 1, true or false), then if performs the and operation of a XOR b and returns to register b. XOR is short for exclusive or. Basically, one or the other, but not both. OR is one or the other, or both.
Push a	Remember the stack I talked about before? Well this is when we can use it. This simple instruction adds the value of a

	or as a register, the value of the register of a to the top of the stack.
Pop a	This instead gets the top element of the stack and puts it into the register of a. If a's not a register, then the compiler complains.
Stare a	Unlike pop, it actually only gets the value of the last element WITHOUT removing the last value.
Call a	Unlike jmp, this calls every script to run a function with the name being the value of a, or if a's a register, the register value of a. If the scripts have the function name, that is.
Jmp @a	Speaking of the jmp command, it seems like I forgot to put it in before. So what this does is that it skips to the label of a. Yes, you can make the program jump around to segments of code with labels.
Be @a	If you know what a branch condition is, then it makes this easier. Basically, to have this work properly, you have to do a comparison first to get the carry flags to even do the branch. Then, all this does is check the carry flag where the two registers were checked if they're equal to each other or not, and if they are, it jumps to the label of a.
Bgt @a	similar to the branch equal to instruction above, but this time it checks if register a was greater than register b and jumps accordingly. (a > b)
Blt @a	Yes, this makes a BLT sandwich in realtime! Jokes aside, this is similar to the two instructions above, but this time it checks if register a was less than register b and jumps accordingly. (a < b)
Bge @a	Same as the others, but this time it checks if register a was greater than OR equal to register b. (a >= b)
Ble @a	Same as the others, but this time it checks if register a was less than OR equal to register b. (a <= b)
Bne @a	Same as the others, but this time it checks if register a was NOT equal to register b. (a != b)
Bbl a @b	I dunno why I added this in, but all this does is check if register a (as a bool) is

	true and if so, jump to the label of b.
Sfx play2d a b c	Plays a regular sound effect with register a being the name, b being the volume (0 - 100) and c being the pitch effect.
Sfx play3d a b c v_d	Plays a sound source in 3d space, with spatialization and sound falloff distance stuff! Plays the sound effect of register a, the RELATIVE volume (usually 0 - 2500?) c being the pitch effect, and vector register d being the location of the sound source.
------------------Round	2: UI---------------------
UI clear	Simple enough, all it does is clears all text, and sprites that have been loaded previously in other gameplays. I suggest creating a script that does only this in the Init function and put that in the startup level or something.
UI Text a b c d e f g h i j k l	OOF. Long like the vray operator, but we'll get through this. The purpose of this is to display some text, with a bunch of options for visual appeal. Here, I'll just show it to you: Register a : text Register b : x pos Register c : y pos Register d : size (0 - 1, not 0 - 100) Register e : direction Register f : hue Register g : brightness(0 - 100; 0 is black) Register h : pixelate Register i : ghost Register j : x spacing Register k : y spacing Register l : unique ID name Ya got that?
UI Sprite a b c d e f g h i j k l m	The purpose of this is to display a sprite, with a bunch of options for visual appeal. Here, I'll just show it to you: Register a : x pos Register b : y pos Register c : size (0 - 1, not 0 - 100) Register d : direction Register e : hue Register f : brightness(0 - 100; 0 is black) Register g : pixelate Register h : ghost Register i : mosaic Register j : whirl Register k : fisheye Register l : costume name Register m : unique ID name Ya got that?

Programming Basics Tutorial

Okay, so all this might be incoherent, as I have made changes to the engine while making this documentation, so please forgive the weirdness going on here. But let's start with the …/contents/scripts file. As you know, pressing [`] + [6] will show you everything under the hood. Now, if you want to, you can look at the default scripts, but unfortunately, I had to run from the SCRATCH FILE SIZE POLICE, so I had to condense it so scratch's happy with the measly 5MB of pure engine code.

So instead, let's start anew. First, make a new folder for all your source code and level/source data. This will be the whole game content. Then you'll want to create a new Text Document, called "Scripts" or something. Make sure that you can see the extension name at the end. If you can't, find a way to do so, by using google. Then, change the extension name to .val.

This will now make vscode, or if you somehow got my extension in your text editor of choice, be able to recognize the file as vector assembly scripts. Then, make another text document but for levels. I didn't make syntax highlighting for level data, so it can be .txt instead.
I will be using vscode as my preferred text editor, so I will be saying some vscode stuff; be aware.

Right now, the game needs to start somewhere! So, make a startup level. This is mandatory and is the essential level that the game runs at the start for instructions of which level to go to. You can also attach initializer modules there, too.

To start, write Level Startup and then below, write End Level in the Levels file. Right now, this can't load anything, or do anything for that matter.

So, let's code a level loader! Now, in the scripts file, we'll need to do a few things. To make the script, we'll need the level name, and the player's starting position and rotation.

I'm not sure if I told you this, but to be able to set the values of registers from modules in the level file, you can use ports coming in from the module that calls the script.

Here's an example that sets port 01 to 8 and port 02 82:

```
module Thingy
ports:
{
    8,
    82
}
```

Really simple, right? The Thingy is just a placeholder for the script name, and inside the brackets, each value is assigned to a port, split by lines. The first port is always the top, and the order goes down each line.

Great! Let's do that! So start by making a new script, by typing Script LoadLevel, and below, type End LoadLevel. Scripts always end with the name that starts the section, not the type, like how levels end with End Level, instead of End [level name]. It can be a bit confusing at first, but that's just how it is.

Currently, all we did was make a new file with nothing in it, so let's use the init function to run the level loading code when the game starts!

Start by typing fn Init inside the script and end script lines, and then below the fn Init type end Init. This declares code to be run inside the init function only once when the green flag is clicked (yes, that's all the init function is).

Currently, we should have:

```
Script LoadLevel
    fn Init:

    End Init
End LoadLevel
```

Which is pretty pathetic, so let's add the actual meat of the stuff to be run. Inside the Init function, we'll have to do

some things to get the values to load the level. Let's break this down:

1. we'll need to move port 01 as the level name
2. we'll need to move port 02 as the vector for camera position
3. we'll finally need to move port 03 as the vector for camera rotation.

So, let's get started! To get port 01's data and set it to the level name, we'll have to remember the register that will have the data. I will be using register 1, and incrementing as we get further. To move port 01 into register 1, we start by typing inside the init function: mov port01 lr1. Remember, if you saw at the operand table, you will know that the instruction mov is just shorthand for move, with the first register setting the second register's value to the first.

Then, we will need the camera's position and rotation. This is also pretty simple, but now we will have to get a bit more advanced. We will be using the star of the show: vector registers! Most of the simple arithmetic operations also have an alternative where there's a v at the start, and it does its operation, but on all three components. (Look at the operand table first, 'cause I won't keep explaining the same things later on!)

So start by typing below: vmov vport02 lvr2. This will assign lvr2 to port02, but as a vector. Yes, ports have types, (but can easily be abused and exploited or if used incorrectly, cause data leaks into other ports when using the mov and vmov operations.) Then, to get the rotation, we use port 03 and vector register 3 as the rotation (obviously). So underneath, type vmov vport03 lvr3. So far you should have this:

```
Script LoadLevel
   fn Init:
      mov port01 lr1
      vmov vport02 lvr2
      vmov vport03 lvr3
   End Init
End LoadLevel
```

Now we're getting somewhere! We now have everything we need to perform the most important instructions: level load and setting the player's position & rotation. Finally, underneath

the mov/vmov operations, type: vmov lvr1
camera:transform:position. This sets the position of the
transform of the camera(which is the player) to the value of
lvr1. Then, we have to type something similar, but for rotation,
which we use lvr2 and camera:transform:rotation, which together,
is: vmov lvr2 camera:transform:rotation.

Now for the most important of important instructions: level
load. If you've seen the operand table, you can figure it out
for yourself, but if you're lazy or just want to see this work,
then type: level load lr1, and you're done! The script loads the
level, and spawns the player at specified coordinates and view
angles. The script should look like this:

```
Script LoadLevel
   fn Init:
       mov port01 lr1
       vmov vport02 lvr1
       vmov vport03 lvr2
       vmov lvr1 camera:transform:position
       vmov lvr2 camera:transform:rotation
       level load lr1
   End Init
End LoadLevel
```

Right now, we will need to make a new level that's going to
be where the player spawns in. To see if this works, we might
also need a mesh in the level. I will call my level HelpMe, and
use the blender suzanne monkey. You can use any mesh or level
name in exchange for the monkey and HelpMe respectively.
Start by going into the level file. You should have
Level Startup and End Level already typed. We will start by
making a new level for the level loader to load the new level.
As I've said, I will name it HelpMe, but it can be whatever you
want. So underneath the startup level, below the end marker, I
will type Level HelpMe and below that, I will make sure to type
End Level.
Then I will add a mesh to demonstrate the level loader
actually works. As I've said, I will add in the suzanne monkey.
I forgot what the actual meshes were in there by default and
their names, so I will make a list here:

1. Triangle
2. Cube
3. Plane

4. Suzanne
5. Cylinder
6. Sphere
7. Cone
8. Torus
9. Pillar
10. Ground
11. House_Scene
12. Metal_Door

Obviously, House_Scene, Pillar, and Metal_Door was for the demo that you saw in the beginning, but you can use it however you please. So for me, I will type:

```
Mesh Suzanne: Thing "Monke"
0 0 0
0 0 0
2 2 2
tint : 1 1 1
texture : brick
uvscale: 1 1
collision: false
rayCollision: false
```

Which looks like a lot, but it's really simple. I have shown you something similar, and how you can set the transform with the three rows of three numbers underneath the mesh identifier at the top. But, here's how the top works: you start with the thing type, a mesh, a light, a collider/trigger, ect. Then you have some specifier after it. For the mesh it's the mesh name you're referencing, for the light it's the light type (there's only one; it's the pointlight), for the collider/trigger, it's the shape. Then you've gotta make sure that there's a colon after with the trademark Thing with the UNIQUE name in double quotes. I called it "Monke".

After all that, there's some additional information you need to fill out. This is specific to the object, but as I've shown you before with the table, the mesh has a color tint, texture name reference, a uvscale, and whether collision and raycasting collision is enabled.

Oh, and to have the default first-person player controller, you need this instruction above the mesh data:

```
      Settings Camera DefaultCamera.

      For me, the whole thing looks like this:
Level Startup
End Level
Level HelpMe
Settings Camera DefaultCamera
Mesh Suzanne: Thing "Monke"
0 0 0
0 0 0
2 2 2
tint : 1 1 1
texture : brick
uvscale: 1 1
collision: false
rayCollision: false
End Level
```

Now for the finale: loading HelpMe. We first need to have a module with the LoadLevel attachment and the ports in the RIGHT ORDER. Yes, there's an order. Port01 is the top line, and all the ports increasing in number go down the list. Start by typing: Module LoadLevel. Then, we get to the ports.

We will first type ports: (the colon is included). Then underneath that, type the opening curly bracket and underneath that, the closing curly bracket. Inside curly brackets which hold all the port data, the first inside it will be the level name as it came first in the scripts.

Strings are always enclosed in double quotes, like a real programming language. So for me, as my level name was helpMe, I will type "HelpMe". Underneath, we will get the position and rotation. Vector inputs are represented with [x, y, z]. Same for rotation; they're vectors. Oh yea, and while vectors have any number of dimensions, these vectors are actually just 3d vectors, but like Unreal Engine, they're just called a vector.

Since I have set the position of my mesh to [0, 0, 0], I will push the player back by 200 units or something, so I'm not inside it. So the position is [0, 0, -200]. Then for rotation, I just made the player look forward in the direction of the mesh, which is [0, 0, 0]. So all together, the module will look something like this:

```
Module LoadLevel
ports:
{
    "HelpMe"
    [0, 0, -200]
    [0, 0, 0]
}
```

    Now it's time to test the whole thing! In the engine, press [`]+[6] and import the scripts and level data, making sure that it shows all files. Then press the green flag! …oh wait, you can't see anything, as there's no lighting at all.

    If you want, you can mess around with lighting, but doing so is a very fidgety process, especially without a visual game editor. So instead, we'll just turn off the lighting!

    To do so, there's this simple instruction that goes underneath the default camera instruction: Settings LightingEnabled (0/1).

    Yes, the boolean after is a 0 or 1 this time. Actually, it's always like that. Anyways, to turn off lighting, type: Settings Lighting Enabled 0. Now, when you run the game, you should see the mesh you picked rendering in front of the camera!

    Here's what I see (with a res of 1):

    As you can see, we are seeing the back-side of suzanne! I can go and set the rotation to [0, 180, 0], but I'm too lazy and there's not too much time to be fiddling around. I hope this short introductory tutorial into how to make your first games in this engine was helpful, and as always, who knows what will happen next?

External Links And Resources

.val Vector Assembly Language Github:
  https://github.com/rinostar1998/Vector-Assembly-Language

(Hello from the future! I have come with good news! I recently released the extension into the marketplace, so you can download it without having to touch vscode's source files! Just search for "Vector Assembly Language" and install the one made by Rinostar.)

Blender.org:
  blender.org/download
Visual Studio Code:
  https://code.visualstudio.com
GIMP:
  https://www.gimp.org/downloads
Texture Haven:
  https://polyhaven.com/textures
CC0 Textures:
  https://ambientcg.com
Open Game Art Textures:
  https://opengameart.org/textures
"Boring" Maths:
  Cross Product
  Dot Product
"Fun" Programming:
  How to write "Hello World" in assembly!
The Hat 3D Editor (for fun):
https://scratch.mit.edu/projects/708802886/

Oh yea, and some more reference tables/Notes:

Notes:

Window Shortcuts

[`] + [1]	toggles the main display.
[`] + [2]	toggles the performance stats.
[`] + [3]	toggles the console display.
[`] + [4]	toggles the memory stats.
[`] + [5]	toggles the camera transform2.
[`] + [6]	toggles the source files.
[`] + [7]	toggles the view uv display.
[`] + [8]	toggles the obj importer.
[`] + [9]	toggles the editor settings.
[`] + [0]	enables texture preview in idle

[=] cycles through the debug views. Wireframe can sometimes be laggier than regular raster, somehow.
[/] + [-] is used to terminate the program and clear all temporary buffers.

When the texture preview is on, you'll have to stop the project. Then, press [.] + [;] or [,] + [;] to cycle through the loaded textures.

More Notes:

The stack this time is global, not local.

vmov obj references:
 - vmov camera:transform:position V_b sets vector register b to position in transform of the main camera.

 - vmov v_a camera:transform:position sets the main camera's position in the transform to the value of vector register a.

 - vmov camera:transform:rotation V_b sets vector register b to rotation in transform of the main camera.

- `vmov v`$_a$` camera:transform:rotation` sets the main camera's rotation in the transform to the value of vector register a.

Thing References:
- `vmov thing:transform:position V`$_b$ sets vector register b to position in transform of the thing the script is attached to.

- `vmov v`$_a$` thing:transform:position` sets the script attachment's thing position in the transform to the value of vector register a.

- `vmov thing:transform:rotation V`$_b$ sets vector register b to rotation in transform of the thing the script is attached to.

- `vmov v`$_a$` thing:transform:rotation` sets the script attachment's thing rotation in the transform to the value of vector register a.

- `vmov thing:transform:scale V`$_b$ sets vector register b to scale in transform of the thing the script is attached to.

- `vmov v`$_a$` thing:transform:scale` sets the script attachment's thing scale in the transform to the value of vector register a.

Reg Ptr References:
- `vmov a:obj:transform:position V`$_b$ Sets vector register b to the position of the transform of the CASTED thing as a ptr for register b. (The index of the target thing)

- `vmov v`$_a$` b:obj:transform:position` Sets the position of the transform of the CASTED thing ptr for register b to the value of vector register a.

- `vmov a:obj:transform:rotation V`$_b$
- Sets vector register b to the rotation of the transform of the CASTED thing as a ptr for register b.

- `vmov v`$_a$` b:obj:transform:rotation` Sets the rotation of the transform of the CASTED thing ptr for register b to the value of vector register a.

- `vmov v`$_a$` b:obj:transform:scale` Sets the scale of the transform of the CASTED thing ptr for register b to the value of vector register a.

- `vmov a:obj:transform:scale V`$_b$
  - Sets vector register b to the scale of the transform of the CASTED thing as a ptr for register b.