Revan's Expression Gate 2 Guide - Intermediate Level - Uncommonly Used Functionality

Things to Notice as you Read the Guide:

There are several symbols/colors I may use throughout this guide to help you find things, so here's a list of them all:

Term - A definition you should know

Other things I may make reference to:

The Expression Gate 2 Wiki - Every now and then I will link to a page on the wiki for you to reference. It's a good idea to bookmark this page, because even experienced E2 programmers such as myself will pull up the wiki mid game to find a function. To make opening it in game easier, I recommend that you either use the steam in-game browser or run Garrysmod in windowed mode (not full screen), a setting which can be altered from options.

Introduction

Welcome to part two of Revan's Guide to the E2 (Expression Gate 2). In reading this, it is my hope that you will better understand how and why the E2 works the way it does. I will make an attempt to keep my data here as accurate and current as possible, but considering that I no longer really play Garrysmod, I cannot guarantee that everything you see on this page will still function in the future, nor can I claim that the ways I show you how to do certain things here are the best ways to do them. The reasons for that include features being added later, my being unaware of the existence of a feature, or even just to teach you a concept, I may use a less appropriate way of achieving it for the sake of the example. Understand and respect those faults as you read.

Before continuing with reading, I advise that make certain that you understand all of the basics of E2 as I described them in my previous guide. If you have not read it already and you are familiar with the E2, I still advise that you look at my previous guide, if only just to skim it. I will assume in this guide that you understand each of the concepts I presented, and that you have made use of them in one form or another in Garrysmod. I assume that you did not just read the text and expect to be a master, but rather that you read, then tried to make your own code based on my explanations and then read some more. I expect that you've made some mistakes, and that you have been able to rectify them, either on your own, or by asking the community at http://www.wiremod.com. This being the case, you should be capable of understanding and using the concepts that I present to you in this guide, however, if you realize midway through that you are not, do not fear, reread my last guide, and focus on the basics for a while longer before you move on; I split the guide for a reason, you don't want to cover too much ground at once.

As a final word, understand that everything in this guide is rarely used. It was probably added to the E2 with a single purpose in mind, and isn't required or useful for the completion of most tasks.

Complex Numbers & Quaternions

Complex numbers are rather... simple to explain. If you know the term from mathematics, it really means the same thing. A complex number is just a domain (set of numbers) which allows you to keep track of imaginary values. If you don't know what an imaginary number is, an example would be the square root of negative '4'. It doesn't exist. Even though it doesn't exist however, we can write it mathematically as '2i' where 'i' is the square root of negative '1'. The E2 allows you to use this domain if you make a variable complex. You can make a number a complex number by typing:

```
comp(N)
```

Which would return the complex equivalent. The functions are rather similar for numbers and complex numbers as can be seen here. The operators for numbers should be the same for complex numbers.

Honestly, I doubt you'd ever use this functionality, but obviously someone needed it. If you had to, I think that it would somehow involve changing the number to a string using the complex number's 'toString' function.

Quaternions are angles and vectors that allow for complex numerical pitch, yaw, roll, and x, y, z values respectively. You can convert an angle or vector to a Quaternion by just putting it as a parameter to the function 'quat'. For instance:

```
quat(vec(0,0,0))
```

Creates a vector (0, 0, 0) and turns it into a quaternion. Alternatively, you can remove 'vec' and just put in the straight coordinates. Functions are listed here. The operators for vectors and angles should be the same for quaternions.

The For Loop

Loops are blocks of code that repeat themselves. If your E2 has an interval line, it's already looping by executing more than once, however, you can make it so that your code loops inside of each cycle of your code. Now, you may wonder, why would I want to repeat the same lines over and over and over again with a special command when I can just add a timer? Well, usually, you shouldn't, but there are cases where it can be very useful. The for loop in particular, is used to count. Since you first entered school you've been able to count out to yourself, 1, 2, 3, 4, and so on, well, your E2 can do the same.

The syntax of a for loop is as follows:

```
for(Var = Value, Till, CountBy) {}
```

What that means is, you will set the variable 'Var' to 'Value' at the start of the loop, and the loop will add 'CountBy' to 'Var' every time the loop runs until it reaches the value of 'Till'. If 'CountBy' is not specified, it will be assumed that it is '1'. Every time your loop runs, the code in between the two curley braces will be ran. Thus, this code:

```
for(Var=0, 10)
```

```
{
	#Code here
}
```

Will cause the 'Code here' block to execute '10' - '0' (10) times, where each time, 'Var' increases by '1'.

So, when do you use timers and when do you use a for loop? If you need to count sequentially, a for loop is generally a good idea, however, if counting is the only thing it does, use timers. It's also a bad idea to run loops that will loop through their code more than... about 100 times per second (not per cycle). If you need to count a large number of times, use some combination of timers and for loops to achieve this. For instance, need to count to 1000? Set an interval to '10' and then have a for loop that counts '3' or so times per cycle. Eventually, you'll reach 1000, and you won't lag anyone in the process. The reason for that, is that you put the different executions of your code spaced out over time. Now, the other downside of trying to run a ton of loops in the same cycle, is that eventually, the E2 will stop you. So, I both recommend and beg of you to use for loops wisely.

Just so you know - you can change the 'Var' variable inside of the curley braces.

Break & Continue

Break is a term which can be used inside of any loop. Break causes the current execution loop to stop and doesn't start the next one, effectively running whatever comes after the closing curley brace of you loop. Generally, break is used with if statements dependent on conditions which are not involving the value of the loop's variable itself. Don't use break to stop a loop at '50' every time if you're counting up to '100', instead, just make your loop count to '50'.

Continue is very similar to break. Continue causes the current execution of the loop, however it's different because it starts the next iteration of the loop to begin, and does not skip over to the end curley brace. Continue is a good way of 'skipping' a value of your loop.

The While Loop

The while loop is used to run a block of code as long as a condition is true. I can't think of why you'd use it in E2, but it's worth knowing if you ever persue another programming language. Here's the syntax:

```
while(Boolean) {}
```

Boolean is representative of any boolean expression, whcih we learned about in the last guide. Alternatively, you could put a numerical value in it's place, causing the loop to run so long as the value isn't '0'. So here's an example of code which (like a for loop would) counts to '10'.

```
X=1
while(X<11)
{
	X++
}
```

We set 'X' to a default value so that it exists for our condition, then we have our while loop, which says, yes, 'X' is less than '11', then it runs the block of code, which increases 'X' to '2', and then checks the condition again, we know that '2' is less than '11', so it runs again, and so on.

The code of a while loop will not run even once if the condition is not true when the line that says 'while' is ran. In other languages, there is a 'do..while' or 'repeat..until' loop that will always run at least once, which is good to know if you ever move to another language, but as of yet, there is no E2 equivilent that I am aware of.

Arrays & Tables

So at this point, you may be wondering, ok, that's all nice information, but why on Earth would we need it? Arrays. Arrays are structures that allow you to store multiple variables under one name. An example of an array would be {1,3,5}, which stores '1' in its 0 position, '3' in its 1 position and '5' in it's 2 position. Now, don't be confused, you won't see someone write out {1,3,5} in an E2, it's just how arrays are generally displayed. It assumes that the indexes of each of the numbers listed are in order starting at '0' usually (sometimes '1', varrying based on language), this is why I chose to call it the 0 position and not the 1st position. The numbers themselves, are the data stored in those slots. Bear in mind that arrays need not have their positions be in line, you could have items in positions 0, 3, and 10, however, there's no shorthand for writing that; you can also use negative positions.

A term that I may use later and you will definitely hear if you program in another language is an index. An index is what I have been referring to as a position. Indexes are the places in your array where the data is stored.

So, how do we make an array? Just type 'array()' and you'll create an empty array. After that we can add new data elements to like so:

```
X = array()
X[2, Number] = 5
```

So, what does that do? It takes '5' and puts it in the array's '2' position. One thing you may be wondering is why I had to type 'Number', it's an odd function that requires you to type our the type of the variable you're setting. The other odd thing here is that unlike most functions, this is typed with square brackets, not parenthesis, and has no colon, so make a note of that. If I later wanted to use that value, I would type:

```
X[2, Number]
```

Which returns '5' in this case.

There are a whole bunch of different functions for arrays that I won't go into here, but for now, understand that the above is the most commonly used array functionalities.

Now, I mentioned at the start of this section that arrays are used with loops, and I hope it has become clear why. You can use for loops to move through each index of an array to access the data inside like so:

```
for(I=1,10)
{
	X[I, Number]
}
```

While I don't actually use that value for anything, I'm sure you can see what it's doing, I'm getting the item at position '1', then '2', then '3', and so on by using the same piece of code. I do assume that you already have an array with data in it up to the 10 position when this runs, so if you try it and it doesn't work, that's why, you may want to use a for loop to set your data before you use one to read it to avoid errors.

Tables are essentially arrays that use strings for indexes. I won't spend much time on this because there is almost no reason to use one, here's an example of how to set one though:

```
X = table()
X["arr", Number] = 1
```

In this case, I set the index of "arr" to the 'Number' '1'.

GLON

GLON stands for Garrysmod Lua Object Notation. GLON is a set of functions which allow you to convert arrays and tables into string format, which makes them take up less space, and thus be more ideal for transferring large amounts of data. If you need to get an array or table to another E2 (at least I believe that's why this was added), then converting it to a string is the ideal way to do so. The functions to make the conversions, called encoding, and to convert them back, decoding, are listed here. I'm betting you can figure them out, I just wanted to explain what they did.