Client-Side QuakeC

Posted on September 30th, 2012 | Last modified on February 5th, 2023

So what do I need to actually DO this stuff?

Well Lets hope you at a minimum know the basics of how to compile QuakeC! If not though don't worry its super easy. All you need is one of the modern day QuakeC compilers out there such as FTEQCC orFRIKQCC. (I recommend FTEQCCGUI for new windows users who want to get into the thick of it as quickly as possible).

Once you have one of these you'll need CSQC's definitions. This is usually one or two .qc files which will hold a list of all the numbered builtins for the engine as well as the various field and global variables CSQC supports. An additional file holds useful constants and various constants used by the definitions and builtins in a seperate file for organizational purposes, but if you wish to merge it into the definitions file no one is going to stop ya!

Open each of these links and copy each of the page's contents to their own respective .qc files. We'll be adding each of these to our .src file to finally compile them all into the csprogs.dat and use this as a base for future explanation of stuff. (Also its a great reference)

CSQC definitions file

CSQC constants file

CSQC builtins file

CSQC required functions file

The engine has some expectations of functions to exist for the bare minimum of CSQC to run (above the CSQC required functions file should be the following functions in easy to copy/paste form). There aren't many and they're all very self explanatory (at least superficially) so I'm just going to list them out here with some simple explanations:

NOTE: some of these functions such as CSQC_Parse_Print and CSQC_Parse_CenterPrint can be omitted and CSQC will still run, however are implemented here for completion sake and knowledge for another way to take advantage of CSQC

```
  When the client receives a print message (such as bprint, sprint from SVQC) it passe
  This behavior simply takes the content of the string into the "string msg" argument

void CSQC_Parse_Print (string msg)
{
        print(msg);
}
```

```
 Same sort of print hook as the previous function, however this one is specifically fo
 from SVQC

void CSQC_Parse_CenterPrint (string msg)
{

        centerprint(msg);
}

  This is the general hook function CSQC uses when it receives an update or is told to
  entity to perform actions on it such as parsing the data stream and assigning values
  this entity is brand new (as in just spawned). You can then set initialising values
```

```
void CSQC_Ent_Update (float isnew)
{

}
```

```
  Another shared entity hook function, this time for when an entity is removed on the
  this function to do any sort of special effects or unique client side control of ren
  Like the previous function the "self" reference is to the entity to be removed in qu
```

```
void CSQC_Ent_Remove ()
{
        remove(self);
};
```

```
  This function is called when CSQC is first initialised. You'd do all kinds of stuff
  sort of start up concepts for your game you want. (NOTE: CSQC at the time of writing

void CSQC_Init ()
{

}
```

```
  This function is called when CSQC is unloaded. This may be from disconnecting from a
  drop into here (careful, if for some unfortunate reason your game crashes you can't

void CSQC_Shutdown ()
{

}
```

```
  A hook for CSQC to parse console commands. the argument "string str" is the console

void CSQC_ConsoleCommand (string str)
{

}
```

```
  A nifty little function that can be used to poll key strokes. It has 3 arguments he
  (2) when it is mouse input data. When the mouse data is passed, param and param2 ar
```

```
  The second argument "float param" is the key that was pressed and can be looked up
  don't worry!). The third is the ascii equivalent of the key pressed, CITATION NEEDE
  can return true if you would not like to have the engine continually send this key

float CSQC_InputEvent (float event, float param, float param2)
{
    local string key, keybind;

        key = KeyMap(param);
        keybind = getkeybind(param);

        if (!event) // key pressed down
        {

                if (keybind == "+forward")
                {
                   print("pressing key bound to move forward!\n");
                }

                if (key == "m")
                {
                  print("pressing the M key!\n");
                }
        }

        if (event == 2) // mouse movement!
        {
             print("X: ", ftos(param), " ");
             print("Y: ", ftos(param2), "\n");
        }

        return false;
}

  Okay now in the previous function I had used a function called "KeyMap" that certai
  a string. So here is that function, (It was stolen somewhere from daemon, I hope he
```

```
string KeyMap(float keynum)
{
```

```
    local string chrlist, key;

    if(keynum == 32)
        key = " ";
    else
    if(keynum >= 39 && keynum <= 61)
    {
        chrlist = "'  *+,-./0123456789 ; =";
        key = substring(chrlist, keynum - 39, 1);
    }
    else
    if(keynum <= -39 && keynum >= -61)
    {
        chrlist = "\"    <_>?)!@#$%^&*( : +";
        key = substring(chrlist, (-keynum) - 39, 1);
    }
    else
    if(keynum >= 91 && keynum <= 93)
    {
        chrlist = "[\\]";
        key = substring(chrlist, keynum - 91, 1);
    }
    else
    if(keynum <= -91 && keynum >= -93)
    {
        chrlist = "{|}";
        key = substring(chrlist, (-keynum) - 91, 1);
    }
    else
    if(keynum >= 97 && keynum <= 122)
    {
        chrlist = "abcdefghijklmnopqrstuvwxyz";
        key = substring(chrlist, keynum - 97, 1);
    }

    return key;
}
```

```
    Another hook here! This time to digest when the server sends a stuffcmd to our cli
    implementation here is actually using one of the csqc builtins to ignore stuffcmd'

void CSQC_Parse_StuffCmd (string msg)
{
        if(isdemo())
                return;

        localcmd(msg);
}
```

```
    This here is our main render loop called every frame. You'll want to use this to s
    and after all that potentially draw any HUD or menu stuff. It receives 3 arguments
    the last one is true when the menu is open, false when not.
```

```
   You can use CSQC to render multiple view ports, resize them, and some other fun stu
   working with here! So be patient :) Also to note, the MASK_* bitflags there are CSQ
   rendering when the player has chase_active for chase camera on.). An entity must ha
   field variable of the entity. More on this later, its simple but deserves a good ex

void CSQC_UpdateView (float width, float height, float menushown)
{
   // make a function to update any local resolution stuff and call it here.. We'd do

        clearscene();           // CSQC builtin to clear the scene of all entities / r

   // update our view location for our camera such as a chase camera function here.. I

        setproperty(VF_ORIGIN, pmove_org); //pmove_org is a CSQC definition (global ve
        setproperty(VF_CL_VIEWANGLES, input_angles); // input_angles is a CSQC definit

        setproperty(VF_DRAWWORLD, 1);           // we want to draw our world!
        setproperty(VF_DRAWCROSSHAIR, 1);               // we want to draw our crossh
        addentities(MASK_NORMAL | MASK_ENGINE | MASK_ENGINEVIEWMODELS);
        renderscene();          // render that puppy to our screen,

  // render hud using builtins like drawstring, drawpic, and any of that fun stuff her
}
```

Okay so I get these functions, and I got these .qc files now, but how do I make it JUST GO?

Good question! Well lets recap what you SHOULD have right now in pretty bullet pointed list form:

csqc_defs.qc, csqc_builtins.qc, csqc_constants.qc, main.qc all in some folder, ready to be lobotomized by your favorite text editor [(Notepad++ is really cool for such stuff and has tabs)]
a compiler such as FTEQCC or FRIKQCC (link to downloads on previous page)
a breadth of knowledge on the most basic barebones functions we need to get your CSQC project off the ground and have it simply run in the engine
a good understanding of what CSQC as a virtual machine is, why its a great addition to SVQC, and that it can do a lot of stuff (but it'd hurt everyone's brain to be told it all at once)
the integrity to soldier on when the going gets tough and an awareness sites such as [Inside3d] offer community support as well as the darkplaces irc channel on the AnyNet server #darkplaces

Now, Im assuming everything went ok if you're reading this far, if not and you're frustrated hit up that last bullet point to get assistance, but ya, we're moving on now, we've got places to be! So thusly lets move on to actually getting all them codey bits to compile into a csprogs.dat file and hopefully if you have enough know-how on the basics of QuakeC this should all be a downhill coast from here!

So we've got our files, take all those files and put them inside a folder called "csqc", and now this is the part where I biasely push my own folder and file structure on you because its how I've most enjoyed working with multiple VM's at once. Put that "csqc" folder inside your source folder that holds any of your other .qc files that most likely exist for your normal QuakeC editing (SVQC stuff). Granted this isn't symmetrical, a smart person would properly seperate SVQC, CSQC, and MenuQC(we won't touch on that here) in 3 seperate folders entirely. However with the unique communication between SVQC and CSQC for shared entities you may find your work easier having CSQC a concept that is layered onto SVQC, and not entirely seperate as this may cause headaches with adding shared files later. So thusly, your structure may look something like this:

```
id1/ (our base directory we're making our game / mod out of)
id1/Source/ (all your SVQC .qc files here like combat.qc, etc)
id1/Source/csqc/ (all your CSQC .qc files here, such as well, all that stuff I previou
```

now put your qc compiler inside of your "Source" folder. We're going to now make our .src file for our csqc and target specific that specific .src file to compile our csprogs.dat We're almost there!
Now, granted you named all the .qc files as was previously mentioned and are following this set directory structure we shall now put together a .src ready to be targetted by our compiler. Copy this into a file

```
../csprogs.dat

csqc/csqc_defs.qc // definitions
csqc/csqc_constants.qc // constants
csqc/csqc_builtins.qc // builtins
csqc/main.qc // required functions
```

now save that file into your "Source" folder as 'csprogs.src'.

Now, theres a few ways we can go about compling our csprogs.dat. If we're using the fteqccgui compiler you can actually boot it up, select that .src file, click compile, and it'll spit out the csprogs.dat And we're done! If you are using the commandline compiler (I can only speak for fteqcc from experience but Im sure frikqcc is similar) you can simply target the srcfile as such:

fteqcc.exe -srcfile csprogs.src

If all is well the compiler should be pleased to give you the csprogs.dat, and should have put it inside of your base directory (id1 in our example directory structure here).

Now you should be able to run your game, and see your player at midget heights (we set our camera to our player's origin pmove_org, you can add a vertical height to offset this).

But more importantly, you survived the most difficult part, just getting up and running! Now lets get on to the less tedious and more fun stuff, such as drawing a HUD!

Finally, visuals!

As previously mentioned and now must feel drilled into your brain, CSQC as a VM offers far

more influence in the text and picture drawing department than could ever be offered by abusing print functions. So lets start off slow and use some existing builtins (that can be found in csqc_builtins.qc) to draw some text.

Okay, so where do I put my stuff to draw on the screen?

We'll be putting our drawing logic right into the render loop (for learning purposes, smart people encapsulate things to their own respective functions). The specific function is: void CSQC_UpdateView ()

So open up main.qc which should now be residing in your "csqc" folder and scroll down to that function and at the end of it. We'll be add but 1 single simple line to draw text and about our player's origin and a few lines of local variables initialised to common values when printing to help you see exactly whats going on.

So first, our virgin function:

```
void CSQC_UpdateView (float width, float height, float menushown)
{
   // make a function to update any local resolution stuff and call it here.. We'd do

        clearscene();           // CSQC builtin to clear the scene of all entities / r

   // update our view location for our camera such as a chase camera function here.. I

        setproperty(VF_ORIGIN, pmove_org); //pmove_org is a CSQC definition (global ve
        setproperty(VF_CL_VIEWANGLES, input_angles); // input_angles is a CSQC definit

        setproperty(VF_DRAWWORLD, 1);           // we want to draw our world!
        setproperty(VF_DRAWCROSSHAIR, 1);               // we want to draw our crossh
        addentities(MASK_NORMAL | MASK_ENGINE | MASK_ENGINEVIEWMODELS);
        renderscene();          // render that puppy to our screen,

  // render hud using builtins like drawstring, drawpic, and any of that fun stuff her
}
```

Notice that last comment about render hud builtins and fun stuff here? I really meant that. So now lets print our player origin to screen! Copy the following right below the comment mentioning where to put hud stuff.

```
        local float text_alpha;
        local string player_org_str;
        local vector text_location, text_size, text_color;

        player_org_str = strcat("Player Origin: ", vtos(pmove_org)); // merge our play

        text_location_x = 320; // text position 320 canvas units to the right
        text_location_y = 200; // text position 200 canvas units vertically downward

        text_size = '12 12 0'; // 12x12 font size
        text_color = '1 1 1'; // RGB color, 1 1 1 is white, 0 0 0 is black, 1 0 0 woul

        text_alpha = 1.0; // alpha of our text, 1 is opaque, 0.5 semi transparent, 0 t
```

```
        drawstring (text_location, player_org_str, text_size, text_color, text_alpha,
```

Now of course there is no requirement to make so many local variables to draw stuff, however I believe it shows some common values and whats going on with our drawing. Now if you compile this you should see your white opaque semi-small text mentioning Player Origin followed by a 3 component vector that updates with your player's origin when you move.

Notice how the text_location from left to right is an increase, and top to bottom is an increase. Experiment as you see fit with various values and things, next we'll get into printing some other data your csqc spec supporting engine of choice so nicely provided as addstats!

First off, what IS an addstat?

Think of an addstat as simply one of a handbag worth of ways for the SVQC to communicate data to the CSQC we have. Now of course reading that list of important CSQC functions you'll see you can communicate from SVQC to CSQC already in a number of ways. You could technically send data by print functions, stuffcmd's, shared entities (although this certainly is a later to be explained concept). But nothing really mentions addstats for functions, thats because addstats are unique, and again, a bias for how one programmer feels CSQC should be used, I shall now impart what little I know of addstats on a technically level however this info should not impede your workflow.

Addstats are in some form or fashion an indexed value (can be an integer, float, or string that is a field variable of a client) that SVQC communicates down to CSQC whenever their value changes. CSQC can choose to read this data by passing the index SVQC assigns to these values to getstat* builtin functions. Voila! Thats all you need to know! So now before we get into implementation, lets get a bit into form and when to use them.

Okay, so when should I actually use an addstat?

Addstats are things that first off are field variables to clients usually. So things the client obviously needs to know about, and most likely are updated often. Something like a player's health or ammo already exist for their respective fieldvar's as addstat's that SVQC will continually send down to CSQC that it can operate on. However beyond the engine defined addstats of course you can create your own fieldvars and use those. So when to use addstats in short list form:

field variable the client needs to know about

the client needs to know about it often enough it deserves it's own dedicated "channel"

Marvelous, now how do I do all this?

Slow your roll there! First before we make a NEW addstat lets take a look into csqc_constants.qc and look at the already predefined addstat's the engine has so nicely provided us.

You'll see a few lines that look something like this:

```
const float STAT_HEALTH         = 0;
const float STAT_WEAPONMODEL    = 2;
const float STAT_AMMO           = 3;
const float STAT_ARMOR          = 4;
const float STAT_WEAPONFRAME    = 5;
const float STAT_SHELLS         = 6;
const float STAT_NAILS          = 7;
const float STAT_ROCKETS        = 8;
const float STAT_CELLS          = 9;
const float STAT_ACTIVEWEAPON   = 10;
const float STAT_ITEMS          = 15;
```

these are the hard coded indexes we can pass to the engine to return data using the getstat* builtins. So now lets put our knowledge into action and print our health to our HUD! Open back up main.qc and scroll down to your "CSQC_UpdateView" function.

At the bottom there we had made a bunch of local variables spelling out what we were going to be drawing followed by ultimatetey a call to drawtext with said local variables printing our player's origin.

We're going to be modifying this to now instead of printing our player's origin printing our player's health.

So lets back it up to what that function should look like now if everything was left as it was from the previous page:

```
        local float text_alpha;
        local string player_org_str;
        local vector text_location, text_size, text_color;

        player_org_str = strcat("Player Origin: ", vtos(pmove_org)); // merge our play

        text_location_x = 320; // text position 320 canvas units to the right
        text_location_y = 200; // text position 200 canvas units vertically downward

        text_size = '12 12 0'; // 12x12 font size
        text_color = '1 1 1'; // RGB color, 1 1 1 is white, 0 0 0 is black, 1 0 0 woul

        text_alpha = 1.0; // alpha of our text, 1 is opaque, 0.5 semi transparent, 0 t

        drawstring (text_location, player_org_str, text_size, text_color, text_alpha,
```

Okay cool, lets change that player_org_str local variable name to something more concept agnostic and concatenate a textual identifer to our STAT_HEALTH addstat. If all goes well you should see the value of your health be displayed where your player's origin was.

```
        local float text_alpha;
```

```
        local float stat_data;
        local string data_str;
        local vector text_location, text_size, text_color;

        stat_data = getstati(STAT_HEALTH); // we pass our hard coded index to getstat

        data_str = strcat("Player Health: ", ftos(stat_data)); // merge our stat data

        text_location_x = 320; // text position 320 canvas units to the right
        text_location_y = 200; // text position 200 canvas units vertically downward

        text_size = '12 12 0'; // 12x12 font size
        text_color = '1 1 1'; // RGB color, 1 1 1 is white, 0 0 0 is black, 1 0 0 wou

        text_alpha = 1.0; // alpha of our text, 1 is opaque, 0.5 semi transparent, 0

        drawstring (text_location, data_str, text_size, text_color, text_alpha, 0); /
```

Now when you change the “health” field variable on the client on the server that text should update with the correct number as the engine behind the scenes feeds that data down from the SVQC to the CSQC. This is networked meaning its all online ready!

Engine defined addstats, check. Custom addstats for my game?

Okay so adding your own addstats is quite simple, but its going to involve a bit of your own direction on when to use them as said before. Lets list what we’re about to do to get you on your way:

Create a field variable for our client to use in our SVQC

Muse over sharing a file for the constant we’ll be defining to index our addstat

discuss how the engine expects different datatypes to be sent and recap on the getstat

builtins to match them

print our custom addstat client side

So first, lets create a field variable on our SVQC. Now nowhere in this series has there been any structure or expected project scope to your SVQC, the hope is CSQC is piggy backing on existing QuakeC knowledge, so I wont hold your hand on all the bare bones qc things for defining a field variable, well, maybe a bit :)

First off lets open up wherever you keep your SVQC definitions. Most people have a defs.qc of sorts, this is where it stores field variables like origin, angles, movetype, etc. Now scroll to the very bottom of it and add this line:

```
.string my_str_addstat; // our string addstat!
```

Intense, okay now part 2 of my 4 bullet pointed list, the musing.

Now previously the directory structure as you well know relies on csqc essentially piggy backing as a folder inside the same folder as your SVQC files. Now it may not be readily

apparent right now and this technique may not suit your personal needs (however this was done for speed explanation not perfection). We're going to want to share some data between SVQC and CSQC. They got a lot in common, and for them to communicate they need some common data. So how about we make a file that shares data they can both use?

This idea would apply in this case as our addstat's hardcoded index must be set in our SVQC for our custom one, but also the CSQC must know it to get the builtin data. Now of course you can just define the constant for each VM to read, and theres nothing wrong with that for now. But to automate the knowledge between these two VM's we may want to simply share some of this data between them. Now this would be done by including a .qc file for both VM's inside of their respective .src's. Commonly you'll see a folder named "shared" alongside a client and server folder with such shared files inside. Things you'd share would be as such:

Constants (such as addstat indexes, helper constants for PI)

Helper Functions (VM agnostic helper functions, potentially for logging or something both VM's need to be able to calculate)

Shared Entities (entities that are shared may take advantage of preprocessor functions to separate what gets compiled into what VM, but having one central location of a shared entity can add clarity)

Now that we're done on that, you can consider doing this, possibly making a folder named "shared" and putting it in your equivalent "Source" folder alongside your "csqc" folder and putting a .qc file in it with our constant addstat index we're about to make. But again, you could just define it in 2 places, its up to you. And we wont touch much on this again until we hop over to shared entities.

Okay so whatever you chose it'll basically be either the same line in 2 places (one for each VM at the bottom of their constant or definitions file) or in one place (in their shared file).

We need to make a hardcoded index so we can grab data. The one we used previously was STAT_HEALTH which is hardedcoded to the value "0".
so lets make one on the first available slot after the engine defined addstats (the lowest index it can start at is 32):

```
const float STAT_MYSTR = 32;
```

now make sure that constant as said is available to CSQC to read from as well as SVQC. Now in SVQC go to where your worldspawn() function is (you DO spawn worldspawn right?)

We're going to want to initialize our addstat connection for the server to send to CSQC there right when that is spawned. The function prototype for adding a stat (incase you dont have dpextensions.qc) is:

```
void(float index, float type, .void field) SV_AddStat = #232;
```

Now we can inside of our worldspawn function put this:

```
AddStat (STAT_MYSTR, 1, my_str_addstat); // our hardcoded index, a number defini
```

So that middle one, a number defining the datatype we are. There are 3 possible datatypes to send, which are the following:

```
//          1: string (4 stats carrying a total of 16 charactures)
//          2: float (one stat, float converted to an integer for transportation)
//          8: integer (one stat, not converted to an int, so this can be used to tra
```

Okay now 1 is string, 2 is float, 8 is integer. But whats all this 4 stats or 1 stat? Well you have to make sure when you make your next addstat it offsets forward that many elements for the hardcoded index. So for example our STAT_MYSTR is 32, it is a string, and thus as it says above takes 4 stats to send. Thus our next AddStat's index would need to start no lower than 36 (shifted 4 elements forward). For floats and integers simply increment one stat element as it says.

Okay now we're almost done, but our "my_str_addstat" field variable on our client has nothing remarkable stored to print! So go wherever you initialise your player and hardcode a value to his/her "my_str_addstat" field:

```
self.my_str_addstat = "custom w00!"; // we can only be 16 char's long!
```

Now, the tough part is definitely done, as we're done with the SVQC and sending data, we now are on the part where CSQC should be receiving this data and can now read it back using those getstat* functions and have it print out!

So lets quickly take a gander at those getstat* function builtins:

```
float(float stnum) getstatf = #330;
float(float stnum) getstati = #331;
string(float firststnum) getstats = #332;
```

Now remember: getstatf is if you're sending a float, getstati if you sent an integer, and getstats if you sent a string. In our case we sent a string. Thusly we should be using getstats. So now open back up your main.qc in your "csqc" folder and scroll down to the "CSQC_UpdateView" function

we had previously a section of the sort:

```
stat_data = getstati(STAT_HEALTH); // we pass our hard coded index to getstat

data_str = strcat("Player Health: ", ftos(stat_data)); // merge our stat data
```

We're going to now simply change those 2 lines to instead use our STAT_HEALTH engine addstat to our custom one we made:

```
        stat_data = getstats(STAT_MYSTR ); // we pass our hard coded index to getstat

        data_str = strcat("Custom addstat: ", stat_data); // merge our stat data with
```

granted your CSQC appropriately has that constant index somewhere defined you should be able to compile BOTH SVQC and CSQC (do not forget you modified both) and now see your custom string with whatever data you filled into it. Now if you change that string in SVQC at any time you can see that data. Its great for short hand names, images to load, monster classnames, or various other things. Of course integers and floats are both also very important, and granted you follow the rules of assigning and reading the 3 possible data types, addstats are very quick and easy to implement!