Skip to Content Skip to Navigation

CSQC guide for idiots

From Quake Wiki

Download the latest version by clicking here.

The following is based on the version from 2017-07-28 by Spike.

Contents

- 1 Purpose of this text
- 2 Motivation for CSQC
- 3 Setting up a basic CSQC mod and compiling
- 4 Our first Code (the 3d game view)
- 5 Our first hud (2d stuff and stats)
- 6 User Input (GUIs)
- 7 CSQC-only entities
- 8 CSQC Entity Networking
- 9 Prediction
- 10 Advanced Skeletal Animation (FTE_CSQC_SKELETONOBJECTS)
- 11 Ragdoll (FTE_QC_RAGDOLL / FTE_QC_RAGDOLL_WIP)

Purpose of this text

This little document is a little primer on Client Side QuakeC (CSQC) for people that have no understanding of it.

The reader is expected to have an understanding of Server Side QuakeC (SSQC) basics (entities, globals, functions, prototypes, fields, etc).

As I am the primary author of FTEQW+FTEQCC, I am biased towards it and its feature set. I will attempt to include sidenotes where my instructions might not match other engines (namely: DarkPlaces), but the code examples given are intended to run in either engine.

Motivation for CSQC

Quake's architecture is that of client and server. Even in a single player game, your engine is running both a client and server simultaneously (for example, demos are simply a capture of all the server->client packets).

This architecture brings with it the ability for new clients to connect mid-game, without having to have run the game right from the start of the map.

And this is the distinction between SSQC and CSQC.

Traditional QC code is compiled into a single progs.dat file which runs purely on the server. It controls where, when, and how things move from one place to another, and the server will automatically 'replicate' a subset of the current state of the game to each connected client each network frame, and that is that subset which the client draws (for terminology, Quake3 refers to these gamestate copies as 'snapshots', which is a good way to think of it).

SSQC, running on the server, has absolutely no control over the client beyond what it can send it. Primarily this includes snapshots, stuffcmds, stats, and 'temporary entities'. But worse than this, SSQC is commonly running on a remote machine, with latency, packetloss, and other networking limitations.

CSQC provides a way around these issues because:

1. CSQC has full control over the screen. It can draw the game where it wants. It can draw a hud how it wants.
2. CSQC is running locally. It has no latency issues at all. Mouse movement is present directly within the frame that follows, rather than needing to bounce off the server.
3. CSQC can perform its own player prediction, allowing mods to freely provide their own player physics without breaking client expectations.
4. CSQC can interact with input devices like keyboard and mice, providing rich user interfaces.

CSQC's down sides:

1. CSQC is separate and distinct from SSQC. The two modules share no data (but may still share significant parts of code). You will need to learn how to get data from ssqc to csqc and vice versa.
2. As a large "after-gpl release of Quake" feature, different engines tend to have their own interpretation of the CSQC spec. As a result, cross-engine compatibility may be hard, but is not impossible.
3. CSQC's versatility can be a curse. The engine tries to avoid doing too many things automatically, most noticeably in regards to the screen - a CSQC mod with functions but no code results in nothing being drawn to the screen. Mods need a small bootstrap chunk of code to retain feature compatibility.
4. CSQC attempts to clean up the API slightly, which can result in certain differences between SSQC and CSQC.
5. Compared to SSQC, CSQC is often written in the context of a single client. It has a greater focus on callbacks, globals, and builtins to retrieve data from snapshots and other hiding places within the engine.
6. CSQC needs a certain amount of cheat protection. Thus there are limitations in place to ensure that the csprogs matches one available on the server (the original versions had other additional limitations!).
7. CSQC is running on the client, and is typically autodownloaded. The client must thus also place certain additional restrictions on console commands and cvars, just as it would via stuffcmds.

In essence, CSQC sits somewhere between the middle of the client's renderer, networking, and input modules. Basically, the use input is filtered via CSQC before going to the client/networking. Snapshots are still parsed by the engine's networking, but the renderer will not draw them without the CSQC saying to do so.

Setting up a basic CSQC mod and compiling

CSQC is compiled just like SSQC. Both start from a .src file, but typically a different one.

While you can use a progs.src inside a different directory, I will write this with the expectation that your CSQC's .src file can be found as eg: c:\quake\mymod\src\csprogs.src

Where SSQC outputs to a 'progs.dat' (or 'qwprogs.dat'), CSQC outputs to a 'csprogs.dat' file. This is forms the first line of your csprogs.src file. Typically with a ..\ prefix so that the qcc writes it to the right directory...

The second line of your csprogs.src should be to specify some sort of 'defs.qc' equivelent for CSQC. Sadly, CSQC has different system+field+builtin defs from SSQC and thus needs a different file.

With FTEQW, you can get the engine to generate such a file for you by issuing the command 'pr_dumpplatform -O csdefs -Tcs' at FTEQW's console (be warned that it includes FTEQCC extensions which will not work in other qccs). You'll want to copy this file to your src directory (instead of it being in your home dir where the qcc can't see it - the given command will name the system path that you'll need to copy it from). If you're exclusively targetting DP, you may wish to get DP-specific defs from other sources, but be warned that certain builtins may have different names, and you will need to work around that. For the things documented here, using an FTEQW-generated csdefs.qc should work okay with current DP versions.

And the third line of your csprogs.src file should name one of your own code files (which you will need to write).

So, inside c:\quake\mymod\src\csprogs.src we have these lines:

```
../csprogs.dat  //the name of the file to
csdefs.qc       //system defs to match the engine, including various extensions
view.qc         //name it whatever you want. This is where you will be writing code.
```

To make things easy (assuming you're using FTEQCC), you can add the following line to your defs.qc

```
#pragma sourcefile csprogs.src
```

This line will cause fteqcc to automatically start compiling your csprogs.dat after it compiles your progs.dat. Its one of those convenience features.

Note that this will not affect your actual progs.dat file (beyond compiling it).

Alternatively you can rename your progs.src to ssprogs.src, and make a new progs.src that contains two pragmas that name your two src files, then you can freely choose just SSQC, just CSQC, or both.

Or you can instead use separate batch files and the -srcfile argument to distinguish between which src file to use.

Create an empty view.qc file and try compiling. You should get a csprogs.dat from it (side note: if you used DP-specific defs instead, you'll get errors about functions not being defined, you'll need to make stubs for these at some point, typically they'll need to return false).

Our first Code (the 3d game view)

Open up your empty view.qc file and add this function:

```
void(float width, float height, float menushown) CSQC_UpdateView =
{
        drawfill('0 0 0', [width, height, 0], '1 1 0', 1, 0);
        drawstring('0 0 0', "Hello, World", '8 8 0', '0 0 1', 1, 0);
};
```

If you save+compile+run, you will now find that your fledgeling mod has had its first effect... The game view has disappeared, to be replaced by a yellow screen! oh noes! By adding that function, the engine now knows you want to take explicit control of the screen, and it is no longer doing that automatically for you. So we need to at least get back to what we had. This will need some explaination. If you're just aiming for a HUD and nothing else, you can grab the example code at the end of this section and otherwise skip this section.

Contrary to 2d drawing, 3d drawing in CSQC is based around building up an entire scene then telling the renderer to do its stuff in one go. This model is somewhat derived from the way Quake3 works, and allows the engine to provide its own shadow, rtlight, etc effects within the gameview as a whole. 3d rendering is thus based upon the concept of an active scene or view (with its set of view properties), as well as a list of the entities present in that scene.

Due to certain expectations with input, the scene is NOT normally cleared automatically by the engine. Even if it was, certain default properties change over frames anyway (like the player position). Thus the First thing we need to do is to clear the scene with:

```
clearscene();
```

Well, okay, that function name was a little obvious. Remember I said that the scene had various 'view properties'? Well, clearscene just set them to their default values for you. However, the default is for the engine to not draw a hud at all. So if only as an example of how to reconfigure scene properties, you'll need the following line too:

```
setviewprop(VF_DRAWENGINESBAR, 1);
```

Lets throw in a crosshair too:

```
setviewprop(VF_DRAWCROSSHAIR, 1);
```

For reference:

```
//setviewprop(VF_MIN, '0 0 0');
//setviewprop(VF_SIZE, [width, height, 0]);
```

Will allow you to move the 3d view around on the screen. The min+size values should be in virtual positions, but DP currently uses physical pixels (sidenote: alternative coordinate systems SUCK!).

Right, so now our scene includes a crosshair, a status bar (aka: hud). The view position and angles are in their default positions (ie: angle pulled from client state, origin pulled from prediction or lerped from snapshots).

Its still not being drawn though. We've only said where it should be. We've not actually said to draw it.

So lets do that for the luls:

```
renderscene();
```

Urr, yeah, another obvious name there from Mr Imaginative.

If you compile and run, what do we get? Try it!

You didn't try it did you. You just read the next line without bothering... Well, if you HAD run it, you would have seen that we now get a fullscreen view of the game, with an sbar and the world. We can run around as normal... But there's something missing, and that, my friend, is entities.

renderscene(); will not automatically hunt for entities for you. The easiest way to achieve this is with the following line:

```
addentities((intermission?0:MASK_VIEWMODEL)|MASK_ENGINE);
```

Of course, you'll want to ensure that the call is between clearscene and renderscene, or they'll just be wiped at the start of the next frame.

This builtin reaches into the networking snapshot code and pulls out all the entities that match the masks. MASK_ENGINE refers to the snapshot entities. MASK_VIEWMODEL refers to the entity that is normally generated from your ssqc player's .weaponmodel and .weaponfame fields (which you generally don't want to appear during intermission).

If you have spawned some entities in CSQC, you can call addentity(yourent) to add a copy of it the scene also. This is useful for one-only entities like custom view models being predicted in CSQC.

But for CSQC entities, the easiest way to add them to the scene is to set your entity's .drawmask to something evil like MASK_ENGINE, and then *addentities* will automatically add it to the scene for you. More on this later.

One thing to note about the following code is that *getviewprop* also exists, if you wish to read a default like VF_ORIGIN. For instance, you can make the view bob randomly by reading its current position, adding sin(time*period)*magnitude to the z coord.

So, to compact that into something small for the lazy people who skipped most of this section. I'll add a few other no-op values commented out as the value given is the default(ish) already.

```
void(float width, float height, float menushown) CSQC_UpdateView =
{
        clearscene();                                   //wipe entity lists. reset view properties to their defaults.
        setviewprop(VF_DRAWENGINESBAR, 1);              //draw a status bar (or hud or whatever the engine normally does) around the screen.
        setviewprop(VF_DRAWCROSSHAIR, 1);               //draw a crosshair in the middle of the screen.
        //setviewprop(VF_ORIGIN, '0 0 0');              //view position of the scene (after view_ofs effects).
        //setviewprop(VF_ANGLES, '0 0 0');              //override the view angles. input will work as normal. other players will see your player as normal.
        //setviewprop(VF_DRAWWORLD, 1);                 //whether the world entity should be drawn. set to 0 if you want a completely empty scene.
        //setviewprop(VF_MIN, '0 0 0');                 //top-left coord (x,y) of the scene viewport in virtual pixels (or annoying physical pixels in dp).
        //setviewprop(VF_SIZE, [width, height, 0]);     //virtual size (width,height) of the scene viewport in virtual pixels (or annoying physical pixels in
        //setviewprop(VF_AFOV, cvar("fov"));            //note: fov_x and fov_y control individual axis. afov is general
        //setviewprop(VF_PERSPECTIVE, 1);               //1 means like quake and other 3d games. 0 means isometric.
        addentities((intermission?0:MASK_VIEWMODEL)|MASK_ENGINE);       //add various entities to the scene's lists.
        renderscene();                                  //draw the scene to the screen using the various properties.
};
```

Our first hud (2d stuff and stats)

Unlike 3d stuff which uses some huge complex scene concept, 2d stuff is immediate only. All 2d drawing calls are immediately sent to the renderer in the exact order they are given, and can thus overlap without issues. Later calls will always overwrite whatever is already on the screen at the position specified.

First of all, add a call at the bottom of your CSQC_UpdateView function to some Hud_Draw function:

```
Hud_Draw([width, height]);
```

And then include the following code just before your CSQC_UpdateView function (or in a different file ideally, just be sure to prototype etc like you would in ssqc):

```
float stitems, stitems2, stweapon;
void Hud_Draw(vector scrsz)
{
        vector pos = [(scrsz_x-320)/2, pos_y = scrsz_y - 24, 0];        //calculate the top-left of the sbar, assuming it is 320 units wide and placed in the
        //get some commonly refered to values
        stitems = getstatbits(STAT_ITEMS, 0, 23);                       //this is the player's self.items value (STAT_ITEMS is generated specially by the ser
        stitems2 = getstatbits(STAT_ITEMS, 23, 9);                      //this is the player's self.items2 value if it exists, otherwise it is the serverflag
        stweapon = getstatf(STAT_ACTIVEWEAPON);                         //this is the player's self.weapon value.

        drawpic(pos, "sbar", '320 24 0', '1 1 1', 0.5, 0);              //draw the sbar background image slightly transparently

        Hud_DrawLargeValue(pos+'24 0 0', getstatf(STAT_ARMOR), 25);
        Hud_DrawLargeValue(pos+'136 0 0', getstatf(STAT_HEALTH), 25);
        Hud_DrawLargeValue(pos+'248 0 0', getstatf(STAT_AMMO), 10);
};
```

And here's the helper function I used which wasn't defined above.

```
void(vector pos, float value, float threshhold) Hud_DrawLargeValue =
{
        float c;
        float len;
        string s;
        if (value < 0)
                value = 0;      //hrm
        if (value>999)
                value = 999;
        s = ftos(floor(value));
        len = strlen(s);
        pos_x += 24 * (3-len);
        if (value <= threshhold)
        {       //use alternate (red) numbers

                while(len>0)
                {
                        len-=1;
                        c = str2chr(s, len);
                        drawpic(pos+len * '24 0 0', sprintf("anum_%g", c-'0'), '24 24 0', '1 1 1', 1, 0);
                }
        }
        else
        {       //use normal numbers

                while(len>0)
                {
                        len-=1;
                        c = str2chr(s, len);
                        drawpic(pos+len * '24 0 0', sprintf("num_%g", c-'0'), '24 24 0', '1 1 1', 1, 0);
                }
        }
};
```

Okay, try running that.

Now you get your basic sbar appearing, with health and armor values, that goes red if the values are low.

Note that I'm not going to give any more HUD code. Go write your own! Here's one I wrote earlier.

There's a few generic builtins there which you may have never seen before. sprintf: generates a formatted temp-string (%g, %f are replaced with a float argument, %s uses a string argument, %v uses a vector argument, most other things can use %i for debugging), str2chr(x,y): reads the character at position Y from string X. These are both available in ssqc as parts of various extensions and are really quite useful.

Here are the builtins for 2d stuff:

```
vector(string picname) drawgetimagesize = #318;                                                     //retrieves the size of a picture.
float(vector position, float character, vector size, vector rgb, float alpha, optional float drawflag) drawcharacter = #320;//draws a single character.
float(vector position, string text, vector size, vector rgb, float alpha, optional float drawflag) drawrawstring = #321;//draws a string without stopping fo
float(vector position, string text, vector size, vector rgb, float alpha, float drawflag) drawstring = #326;           //supports colour codes
float(vector position, string pic, vector size, vector rgb, float alpha, optional float drawflag) drawpic = #322;      //draw a picture on the screen. gfx/
float(vector position, vector size, vector rgb, float alpha, optional float drawflag) drawfill = #323;                 //draws a simple box
float(string text, float usecolours, optional vector fontsize) stringwidth = #327;                                     //gets the virtual width of the stri
void(vector pos, vector sz, string pic, vector srcpos, vector srcsz, vector rgb, float alpha, optional float drawflag) drawsubpic = #328;//draws a subregion
```

All of those builtins can be used to draw various 2d stuff. Using different fonts is an extra extension that I'm too lazy to explain here.

But there's also *getstatf* and *getstatbits* that need some explaining. To do so, I'll need to explain the network protocol a little.

Quake's networking protocol is more than just entity snapshots. It also includes a concept of stats. In NQ, different stats are transfered to the client with various limitations, while in QW they're generally more generic. Generic aproaches are good, so all stats are exposed via the same sort of interface similar to the one that the engine uses.

Basically, all the various player-specific fields form the various bits of your HUD are various 'stats' (technical term rather than general term).

The first 32 stats are reserved by the engine for one reason or another, either for future expansion or because its just part of the protocol that has always existed (they are typically some representation of various player fields and are safe to read - later ones may depend upon server extensions). Stats between 32 and 127 inclusive are 'free' stats reserved for mod-specific purposes by QC.

- getstatf reads the numeric stat into a float.
- getstati reads the numeric stat into an integer (as most qc code doesn't support ints, you would normally think it useless).
- getstats reads a string stat, returning a tempstring.
- getstatbits is a bit more awkward, and is provided only for legacy compatibility with STAT_ITEMS. This needs more explaining... STAT_ITEMS, as you will see in the Hud_Draw function I gave you, is read twice. Why read it twice, you might ask... Because STAT_ITEMS is an integer bitfield over the network. Worse - the upper bits of this integer (which cannot normally be used in floats due to precision) is packed with info on runes currently held! So, in order to read both parts of this stat, you need a special builtin which can decode the various integer bits into a float which qc can actually use. The positions and bitcounts should generally be useable for any mod, although you might need to handle runes being a little shifted if your mod does not make use of items2.

So that's how you read a stat, but you'll still need to know how to create your own stats for your awesome hud, and that requires some ssqc (which is where the info is).

The main way to add a stat is with this builtin:

```
void(float num, float type, .__variant fld) clientstat = #232;
```

Which is typically invoked from worldspawn using something like:

```
clientstat(STAT_MYSUPERSTAT, EV_FLOAT, mysuperstat);
```

where mysuperstat was defined in defs.qc or whereever as:

```
.float mysuperstat;
```

And, as noted above, is read with getstatf(STAT_MYSUPERSTAT)

You'll want to share the STAT_FOO defines between ssqc+csqc.

.int foo; clientstat(STAT_FOO, EV_INTEGER, foo); int myfoo = getstati(STAT_FOO); works too, if your qcc+engine support integers.

.string foobar; clientstat(STAT_FOOBAR, EV_STRING, foobar); string myfoobar = getstats(STAT_FOOBAR); works as well. Hurrah.

.entity enemy; clientstat(STAT_ENEMYNUM, EV_INTEGER, foo); float myenemynum = getstatf(STAT_ENEMYNUM); works too, if your qcc+engine support integers.

However, vectors are not normally supported as stats. You will need to separate the various componants (ie: myvector_x) and send+read as 3 separate floats.

Entities are sent only as numbers _in the SSQC_. The numbers don't always match up, thus findfloat(world, entnum, getstatf(STAT_ENTNUM)) generally needs to be used if its to refer to some csqc entity. Note that it may still evaluate to world, as entities may not be in the pvs or may not even be visible to csqc. Stats and snapshots may all have different latency and arrive at different times due to packetloss or whatever. As noted above, those custom STAT_FOO values should normally be between 32 and 127 for futureproofing. You can generally go higher too, but its not guarenteed.

Sidenote: DP compat with stats is more awkward. Unlike FTEQW, DP still has exclusively integer stats. numeric stats are specifically integer stats thus float stats are thus rounded towards 0. strings on the other hand are limited to 15 chars, and actually consume 4 consecutive stats. If you actually need floating point precision with DP, you can tell the engine that the stat is actually an integer, using ev_integer and getstati. Doing so will avoid truncation and has resulted in some enterprising dp users renaming getstati to getstatf just to confuse everyone, so be careful of that if you're using dp-specific defs...

User Input (GUIs)

User input (keyboard and mice and stuff) can be intercepted using the CSQC_InputEvent function.

This function is a generic entry point for multiple different sorts of events. The type of event is passed in evtype. The later arguments have different meanings depending on the event type.

Event handling is basically implemented via interception. That is, if you want to do something with the event you can do so, and you can then 'cancel' the event in the engine's eyes by returning TRUE.

So, if you did something and now want the engine to ignore it, return TRUE. If you want the engine to handle the event for you, return FALSE.

- Note that not all events can be meaningfully canceled, and other events should not be canceled.

For example, if the user presses a key, the key has been pressed, the unicode meaning of that key combination is still determined (normally by the operating system), and you will still receive a release event when the user releases the key, but by cancelling the event, you can get the engine to avoid doing any key binding logic. Of course, key UP events should generally never be canceled as a general rule. The reasoning for this is that if you are cancelling up events, you must be very careful in tracking down events and the owners of those down events, because if you cancel a down event and NOT an up event, you can end up with the engine's key binding system thinking a key is still pressed and thus with the player running forwards endlessly.

So:

Key down events:

- scanx: this is the quake KEY_ code value associated with the physical hardware button. May be set to 0. You'll always get the same code for each key regardless of whether shift etc are pressed also.
- chary: this is the unicode code point of the key. May be set to 0. You'll get different values from the same key depending on whether shift etc is pressed.
- devid: for mouse buttons, this is a mouse-device/touch-event id (see mouse absolute events for details). for keyboard buttons, this is a keyboard device id.
- cancel to avoid bindings (like disabling weapon switches or +forward etc).
- Either scanx or chary may be set to 0, but not both at the same time.

- Some windowing systems support scanx and chary both being set in a single call (like scanx==KEY_1 and chary=='!'), or may invoke the function twice (ie: scanx==KEY_1 and chary==0, and scanx==0 and chary=='!').
- This is significant when it comes to dead keys and unicode input in certain locales.
- Thus if you are typing, you should use ONLY chary for actual typing, with scanx for control keys like the cursor keys+delete, and otherwise support bindings via the scanx value and ignore chary.

key up events:

- scanx: as in key down events.
- chary: as in key down events.
- devid: as in key down events.
- It is not guarenteed that you will receive unicode up events as the windowing system may not support it, but you will receive scancode up events.
- cancelling avoids cancelling -forward type bindings which should normally always be safe for duplicates, thus cancelling up events should generally be avoided.

mouse motion events:

- scanx: how many pixels across the screen the mouse has moved (scaled in terms of virtual pixels, so may be fractional).
- chary: how many pixels down the screen the mouse has moved (scaled in terms of virtual pixels, so may be fractional).
- devid: a mouse device or touch event id. see mouse absolute events for details.
- If running a UI, you would want to add the two values to the previous mouse cursor position before returning TRUE.
- cancel to avoid the engine changing view angles.

mouse absolute events:

- scanx: the absolute X position on the screen in terms of virtual pixels.
- chary: the absolute X position on the screen in terms of virtual pixels.
- devid: a mouse device or touch event id.
- The device id for mice and touch events can be somewhat complex.
- For mice, you are likely to see all mice appear as id 0, simply because the operating system merges all mouse devices to simulate a single one which is all the engine can see.
- On windows, you tend to need an engine that supports raw input in order to get distinguishable mouse devices.
- When used in a mouse-driven user interface, a mouse devid is a simple mouse device id that is constant for every event that the mouse generates.
- For touch events, things are more complicated. Touch ids have a limited life span in order to facilitate multitouch screens.
- A multitouch sequence thus follows these lines:
    - mouse absolute
    - key down (scanx=key_mouse1)
    - mouse absolute etc
    - key up (scanx=key_mouse1)
- After which, the touch id can be recycled as a different touch event.
- (Side note: to test multitouch in fteqw+vista+, use these console commands: in_simulatemultitouch 1; in_rawinput 1; in_restart; note that you will need multiple mice plugged in, one for each 'finger')
- A mouse device may use either mouse motion or mouse absolute events depending on whether the mouse is grabbed for use exclusively with quake or not.
- It is not uncommon for engines to automatically switch to absolute positions when the console is down.
- cancel to avoid changing view angles if the engine has some sort of touchscreen mode enabled, otherwise does nothing.

accelerometer events:

- scanx: x acceleration
- chary: y acceleration (-9.8 (gravity) if the screen is held vertically I believe)
- devid: z acceleration (-9.8 (gravity) if the screen is flat facing upwards)

focus events:

- scanx: now has mouse focus. true=focused. false=unfocused. -1=unchanged
- chary: now has keyboard focus. true=focused. false=unfocused. -1=unchanged
- devid: the mouse or keyboard for which focus changed.
- for the sake of simplicity, you can just ignore any events with a devid other than 0.
- note that it is the engine's choice whether the qc receives input or not. if the console or the menus are active, those may exclusively receive input instead.

So for a windowing system with a mouse cursor and a simple text field, we can write this simple code:

```
string userinputtext;
vector mousecursor;
float(float evtype, float scanx, float chary, float devid) CSQC_InputEvent =
{
        switch(evtype)
        {
        case IE_KEYDOWN:
                string newt = userinputtext;
                if (scanx == KEY_BACKSPACE)
                        newt = substring(newt, 0, -2);  //-1 = end of string. -2=one char before the end of the string
                else if (scanx == KEY_ENTER)
                {
                        //send it to the server in the form of a say command. just to show some useful(ish) interaction with the server.
                        localcmd(strcat("cmd say ", userinputtext, "\n"));
                        newt = "";
                }
                else if (chary)
                        newt = strcat(newt, chr2str(chary)); //add the char to the end.
                else if (scanx == KEY_MOUSE2)
                        newt = ""; //clear it on right click.
                //other scancodes not recognised, but don't add redundant 0s on the end.
                //temp strings are not valid between calls.
                if (newt != userinputtext)
                {
                        strunzone(userinputtext);
                        userinputtext = strzone(newt);
                }
                break;
        case IE_KEYUP:
                return FALSE;   //don't break things if the menu was enabled mid-game while other keys are potentially still held, or some other way.
        case IE_MOUSEDELTA:
                //note: we can cope with multiple separate mouse devices here.
                mousecursor_x += scanx;
                mousecursor_y += chary;
                return TRUE; //don't change view angles
        case IE_MOUSEABS:
                if (devid != 0) //no stuttering please.
                        return FALSE;
                mousecursor_x = scanx;
                mousecursor_y = chary;
                return TRUE; //don't change view angles
        default:
                break;
        }
        return FALSE;
};
//NOTE: this conflicts with previous examples! skip the drawfill and add the rest after your renderscene call or something.

void(float width, float height, float menushown) CSQC_UpdateView =
{
        drawfill('0 0 0', [width, height, 0], '1 1 0', 1, 0);   //clear the screen
        drawstring('0 0 0', userinputtext, '8 8 0', '0 0 1', 1, 0);     //draw their input text
        float sc = (sin(time*8)+1.5)*8
        drawcharacter(mousecursor - '0.5 0.5 0'*sc, '+', sc*'1 1 0', '0 0 1', 1, 0);    //draw a lame cursor with pulsing size (to distinguish it from any p
};
```

Of course, as you go further into windowing systems, you'll need to make some sort of heirachy or screen regions or some other logic so that things can be clicked on. Explaining how to write UI wigits is somewhat outside of the scope of this tutorial, so look elsewhere for such things.

If you wish to use a 2d mouse cursor to interact with 3d positions, you can do something like:

```
vector() CursorToWorldCoord =
{
        vector wnear = unproject([mousecursor_x, mousecursor_y, 0]);    //determine the world coordinate for the mouse cursor upon the near clip plane
        vector wfar = unproject([mousecursor_x, mousecursor_y, 100000]);//determine the world coordinate for the mouse cursor upon the far clip plane, with a
        traceline(wnear, wfar, TRUE, world);
        return trace_endpos;
};
```

Similarly:

```
vector(vector worldpos) WorldToScreen =
{
        vector twodee = project(worldpos);
        twodee_z = 0;   //discard all depth info (note that depth is typically scaled between 0 and 1, so avoid)
        return twodee;
};
```

CSQC-only entities

I'm going to introduce you to a new entrypoint just for the luls:

```
void(float apiver, string enginename, float enginever) CSQC_Init =
{       //note: the three arguments are engine-defined, and can be used to to detect if the engine version you're running in has bugs. Just call error("plea
        local entity foo = spawn();
        precache_model("progs/player.mdl");     //or something else
        setmodel(foo, "progs/player.mdl");      //same as ssqc, see, its easy!
        setorigin(foo, '0 0 0');                //you'll need to fix the origin to ensure you don't put it in a wall...
        foo.drawmask = MASK_ENGINE;             //cause the entity to be added to the scene automatically via the addentities(MASK_ENGINE) builtin.
};
```

So yeah, when the csprogs is loaded, the CSQC_Init function is automatically called by the engine. Inside we spawn an entity, just like we would in ssqc. Give it a position, just like you would in ssqc, etc.

In fact, the only difference so far is the extra drawmask line. This line is so that the core csqc can handle subviews gracefully instead of all entities being added to the scene unconditionally.

So if we try running it, our CSQC_UpdateView (at least ones that actually draw a view) automatically pick up our new entity, but we've not set a think function nor are we trying to animate it.

CSQC entities do not interpolate or autoanimate, nor does the engine interpolate them. What's worse, is that think functions with their predefined interval can result in video frames skipping with jerky animations! oh noes!

So lets add an extra csqc-specific line to where you spawned your entity.

```
foo.predraw = AnimateSomeLameModel;
```

Nice simple function field assignment there...

And we need to define that function. Here's one I wrote earlier:

```
float() AnimateSomeLameModel =
{
        #define FIRSTFRAME 0
        #define NUMFRAMES 6
        self.lerpfrac -= frametime * 10;        //animate at 10fps
        while(self.lerpfrac < 0) //protects against sudden low framerates.
        {
                self.frame2 = self.frame; //if we're at 0, frame2 became redundant anyway
                self.frame += 1;        //move on to the next frame
                if (self.frame >= FIRSTFRAME+NUMFRAMES) //this should be relative to the current animation that should be shown
                        self.frame = FIRSTFRAME;        //restart it.
                self.lerpfrac += 1; //go to the start of the frame
        }

        makevectors(getviewprop(VF_ANGLES));    //set v_forward etc.
        vector vieworg = getviewprop(VF_ORIGIN);//read the current view origin
        setorigin(self, vieworg + v_forward*128);//reposition the entity to 128 units infront of the view so it doesn't get lost in the wall, so we know its
        return FALSE;
};
```

Yes, its completely lame, but it demonstrates the basics of animation, which is what is important, right?

Anyway, the three fields 'frame', 'frame2', and 'lerpfrac' work together to provide animation blending. A lerpfrac of 1 means that only frame2 is used, while a lerpfrac of 0 means that only frame is used. A lerpfrac of 0.5 means that both are used with the result being 50% between them. If you don't do the lerpfrac thing and instead just set frame inside some think function, the animation will NOT be smooth due to the lack of auto interpolation.

With the code above, I've made lerpfrac act as a stepping timer that keeps counting down from 1 to 0. when it drops below 0, a new frame is chosen and the old one is archived off in frame2 and lerpfrac is reset to about 1. This means that the frame choice after a frame switch is still mostly the old frame, blending into the new frame over time.

Be warned that frametime is in local time rather than tied to server time, and thus the animation is purely clientside with no relation to server timings and is thus unsyncronised. If you want a more robust solution, you would need to syncronise based upon the time difference between time and some starttime value when the new action became valid.

Sidenote: To animate within a framegroup, you will need to update self.frame1time and self.frame2time each frame. This is the absolute time into the animation. If you increment the value by something like: frametime*distance_traveled/animated_distance_traveled, it should be possible to achieve foot syncronisation with different movement speed settings.

CSQC Entity Networking

CSQC entity networking is works with callbacks, and includes tolerance for packetloss and latency. Basically, the SSQC sets various SendFlags on the entity to say which parts of the entity have changed, and the server calls the SendEntity callback in order to build an update mssage at some point in the future. The callback is told the SendFlags that need to be sent.

The redundantcy and vauge wording is where packetloss tolerance comes in. Basically, the server is responsible for tracking the flags in each message, and any which never arrived on the client must be resent. This can take a successful round-trip before dropped packets are noticed, hence the 'in the future', and why the callback is told the fields which got dropped.

The network message itself is framed only by the entity number. The callback is responsible for stating exactly what sort of entity it is, and what is actually inside the message. This is typically achieved with a leading entitytype id byte.

In SSQC:

```
float(entity to, float sendflags) MySendEntity =
{
        WriteByte(MSG_ENTITY, 5);               //write some entity type header
        WriteByte(MSG_ENTITY, sendflags);       //tell the csqc what is actually present.
        if (sendflags & 1)
        {
                WriteCoord(MSG_ENTITY, self.origin_x);
                WriteCoord(MSG_ENTITY, self.origin_y);
                WriteCoord(MSG_ENTITY, self.origin_z);
        }
        if (sendflags & 2)
                WriteByte(MSG_ENTITY, self.frame);

        if (sendflags & 128)
                WriteByte(MSG_ENTITY, self.modelindex); //sending a modelindex is smaller than sending an entire string.

        return TRUE;    //handled. If you return FALSE here, the entity will be considered invisible to the player.
};
```

In your spawn function:

```
        self.SendEntity = MySendEntity;
```

And when the origin changes:

```
        self.SendFlags |= 1;
```

And when the frame changes:

```
        self.SendFlags |= 2;
```

Note how the SendEntity function refers to sendflag 128, but its not set anywhere. Well, 'new' entities always have sendflags set to 0xfffff or something just large enough to avoid overflowing floats. Thus entities which have just been spawned or otherwise have just entered a player's pvs will always have sendflag bit 128 set, and thus the modelindex will be sent for such new entities without needing to explicitly state it.

From the CSQC side, the engine notices the entity update and then calls CSQC_Ent_Update. If the entity is new, it calls CSQC_Ent_Update beforehand. The CSQC must then read whatever header the ssqc wrote, determine what sort of entity it is, and determine what the meaning of the various fields are.

Once the SSQC removes the entity or it just becomes invisible, the server will send a remove event, which the client will handle by calling CSQC_Ent_Remove. The CSQC is expected to then call remove(self) before returning.

```
void(float isnew) CSQC_Ent_Update =
{
        float enttype = readbyte():
        float flags;
        switch (enttype)
        {
        case 5:
                flags = readbyte();
                if (flags & 1)
                {
                        self.origin_x = readcoord();
                        self.origin_y = readcoord();
                        self.origin_z = readcoord();
                        setorigin(self, self.origin);   //for correctness.
                }
                if (flags & 2)
                        self.frame = readbyte();
                if (flags & 128)
                {
                        setmodelindex(self, readbyte());
                        self.drawmask = MASK_ENGINE;
                }
                break;
        default:
                error("Unknown entity type! oh noes! panic!");
        }
};
void() CSQC_Ent_Remove =
{
        remove(self);
};
```

As you add new types of entities, you'll need to add more cases. Ideally you'd just move much of that code into a function call.

Also, you might want to combine the enttype byte and the flags byte, at least for entities which don't have many flags.

If you keep your entity networking fairly generic, you can combine various projectiles into common classes, and doing so shouldn't really need more than 16 different classes, thus giving four flags for potentially less overhead. But that's optimisation talk and thus perhaps out of place here. Anyway, you get the idea.

The above example is very simplistic, of course, as it only sets origin, frame, and model.

There's a few things to look out for. As described in the previous section, there is no automatic interpolation or animation lerping, so you'll need to combine that with the predraw stuff. To interpolate positions, you'll have to keep track of the update time, old update time, new origin and old origin. lerping is then:

```
local float frac = (time - self.lastupdate) / (self.lastupdate - self.prevupdate);
frac = bound(0, frac, 1);
self.origin = self.prevorigin + (self.lastorigin-self.prevorigin) * frac;
```

You can of course make assumptions along the lines of an entity updating 10 times a second on the dot, which may or may not result in more robust interpolation. When interpolating angles, be sure to take care with wrapping from 360 down to 0.

Prediction

In theory, prediction is easy. Simply keep an input frame log, track which input frame was the last one applied to movement data, and applay any unacknowledged input frames to the last state received using the same logic that the server will use. In practise, prediction is not reliable. It is a guess. A guess which is not particuarly compatible with traditional NQ player entity physics frames with their lack of syncronisation or even acknowledgements.

Generally this means that you must implement ALL player physics code yourself, using tracebox. Alternatively you may be able to get away with using the engine's runstandardplayerphysics function instead, modifying the various input_* values slightly before calling it.

From the SSQC side, things are fairly simple. The server calls:

```
void() SV_RunClientCommand =
{
        input_angles_y += 180;          //a fun mindfuck.
        runstandardplayerphysics(self); //and apply the input frame to the current entity state.
}:
```

From the CSQC side, when you receive an entity state the servercommandframe global will say the input frame number that was last acknowledged by the server. Thus any player entity from the server will have this input frame already applied. While the clientcommandframe global is the input frame that is currently being generated. Thus if 'self' has its fields set to the most recently received state, you can apply all pending input frames with:

```
for (if = servercommandframe+1; if <= clientcommandframe; if+=1)
{
        if (!getinputstate(if))
                continue;               //that input frame is too old.
        input_angles_y += 180;          //a fun mindfuck.
        runstandardplayerphysics(self); //and apply the input frame to the current entity state.
};
```

The input frame that matches clientcommandframe is updated each frame, until it is finally sent at which point the global is bumped. Input frames will not be remembered forever, but will otherwise be static until they are forgotten.

You can then set the VF_ORIGIN view property to match the predicted position. You will also likely want to smooth out steps, and perhaps add error correction too.

Advanced Skeletal Animation (FTE_CSQC_SKELETONOBJECTS)

When animating players and such, you may desire more than just frames.

If you're using a skeletal model format (like IQM), then more complete bone control becomes available to you.

When your entity is first spawned after you set the model, add some code like:

```
        self.skeletonindex = skel_create(self.modelindex);
```

Each frame, update your .frame, .frame2, .lerpfrac, .frame1time, .frame2time accordingly and then call:

```
        skel_build(self.skeletonindex, self, self.modelindex, retainfrac, firstbone, lastbone, addfrac);
```

That call needs some explaining. Especially the last four arguments. retainfrac is what fraction of the pose info to retain. At the start of each frame you should generally pass 0. Later calls will generally pass 1.

addfrac is what fraction of the animation data to add. Note that this also serves as a sort of scaler. All skel_build calls for each bone in a single skeletal object should add up to 1 to avoid extra scaling.

Using these two values you can blend multiple animations together, for instance allowing running animations to fade into a standing animation, or to blend both sideways and forwards running animations together at rates depending on direction+speed in order to achieve foot-sync.

firstbone and lastbone provide a way for you to affect only a specific region of the model. Beware that this requires that your skeleton has bones that are carefully ordered.

If firstbone is 0, it'll be corrected to 1. Bone 0 is not otherwise valid, as 0 refers to the entity position itself.

If lastbone is 0, it'll be replaced with the total bone count.

If ordered correctly, you can split the model by legs+torso by a bone at the base of the spine. With that achieved, you can find the spine bone with skel_find_bone(self.skeletonindex, "spine1") or so (depends what your model calls it), and then build the two parts of your animation with:

```
float() playerpredraw =
{
        //interpolate origin, and calc displacement
        //note: you probably want to replace this with prediction if you're working on the local player's entity.
        float frac = (time - self.lastupdate) / (self.lastupdate - self.prevupdate);
        frac = bound(0, frac, 1);
        vector newpos = self.prevorigin + (self.lastorigin-self.prevorigin) * frac;
        vector moved = newpos - self.origin;    //this is how much we've moved since the last render frame, note that it implies frametime.
        self.origin = newpos;
        //determine animation weights
        makevectors([0, self.angles_y, 0]);     //set v_forward,v_right vectors so we can do dot products to see how much of which direction the entity move
        self.forwardweight += fabs(v_forward*moved) * 32;       //scaler should rise to 1 after 0.1 secs when moving at 320qu
        self.sideweight += fabs(v_right*moved) * 32;    //scaler should rise to 1 after 0.1 secs when moving at 320qu
        //clamp
        float sc = self.forwardweight + self.sideweight;
        if (sc > 1)
                sc = 1/sc;      //greater than 1 would add negative legs...
        else if (moved_x || moved_y)
                sc = 1;         //don't drop any movement weights if we're moving
```

```
	else
		sc = max(0, sc - frametime * 10);	//drop down to 0 over 0.1 secs if we stopped moving.
	self.forwardweight *= sc;
	self.sideweight *= sc;
	self.lerpfrac = 0; //just in case...
	//add forward animation (retain frac = 0 to reset it)
	self.frame = $forwardanimation;
	self.frame1time = (self.forwardtime += frametime * (v_forward*moved));
	skel_build(self.skeletonindex, self, self.modelindex, 0, 0, spinebone, self.forwardweight);
	//add side animation (retain frac = 1 to not overwrite forward)
	self.frame = $sideanimation;
	self.frame1time = (self.sidetime += frametime * (v_right*moved));
	skel_build(self.skeletonindex, self, self.modelindex, 1, 0, spinebone, self.sideweight);
	//add idle animation (retain frac = 1 to not overwrite directions)
	self.frame = $standanimation;
	self.frame1time = (self.sidetime += frametime);
	skel_build(self.skeletonindex, self, self.modelindex, 1, 0, spinebone, 1-(self.forwardweight+self.sideweight));
	//add torso animation (retain frac = 0 to reset torso parts. note that this doesn't affect legs due to the bone ranges)
	self.frame = $torsoanimation;
	self.frame1time = time - torsostarttime;
	skel_build(self.skeletonindex, self, self.modelindex, 0, spinebone, 0, 1);
	return FALSE;
}
```

and in your spawn code:

```
	setmodel(self, "progs/skelplayer.iqm");
	self.skeletonindex = skel_create(self.modelindex);
	self.predraw = playerpredraw;
	self.drawmask = MASK_ENGINE;
```

and when its killed:

```
	skel_delete(self.skeletonindex);
```

Then when your entity is added to the scene (via addentities), the predraw function is called to interpolate your entity's origin and update its skeltal object.

You'll have to ensure your parse code updates lastupdate and prevupdate timestamps, along with lastorigin and prevorigin, that it updates the angles.

$forwardanimation etc frame macros will need to be set to the correct framegroups within the model (note that this code requires framegroups, and would become more complex if you wished to animate between descrete frames for each animation, but that doing so is possible by using lerpfrac and frame2 instead of frame1time).

The code does not interpolate angles, which is something you'll probably want to add, as well as add a concept of actions so that you can blend between idle+shoot animations as appropriate.

Ragdoll (FTE_QC_RAGDOLL / FTE_QC_RAGDOLL_WIP)

Ragdoll support is dependant upon ODE, which requires cvar("physics_ode_enable") to be set in order to work properly.

Once you have a skeletal object, you can add in ragdoll by calling skel_ragupdate(self, "doll foo.doll", 0); at the end of your predraw function (which will cause the ragdoll to update and move based upon an animated model, and then call skel_ragupdate(self, "animate 0", 0); once your entity dies and goes limp (typically, this is done part-way through the death animation, so the ragdoll inherits a certain amount of velocity from said animation).

The .doll file describes the bodies and joints within the model. If a body is set to animate, it will use the bone data your code specifies. If a body is not set to animate, it will be ragdolled even when the player is still alive. This can be used for things like ponytails, cloth, etc.