

# CODING WITH CRYSTALS

## X6-001 and posix a Tutorial/Introduction

Habituation: \_\_\_\_\_

About navigating the workspace through the shell and coming to terms with the code through sublime. A simple guide to diving in and knowing what is what with coding examples to work out a basic framework. I do this as an exercise and an experiment while also trying to get something done – so, this is what I ended up with. Now, the goal is to not bother what is and isn't right or wrong with the code – but to set out on a simple goal. We want a 3D Engine that we can engage with through the controller and an overlay we can activate when pressing a button.

The first thing to understand, I guess, is the matter of what System Layer we are working with or want to be working with. So, provided you have got the basic X6 environment set up and running, you may open (`subl -n`) the 'folder' `crystals.gen/crystals.zero/public` – and parallel to that open `crystals.gen`. I then like to “swallow” the terminal with `mod+shift+minus` (cycle through them with `mod+minus`, and enlarge a window with `mod+shift+space`).

In one window you can find the 'public' folder with a single file in it; And in the other you can see the “general root” of the system. Looking at the single file, the `M_CORE`, you can scroll to the bottom and find the function `M_CORE::__MSYS_PRIME__`, and this is the first “regular” loop of the system. Or the only one. It is THE loop and integrating a 3D engine on that Level would be the first way to go about it. You might not want to change the one right there, but copy it to somewhere else. You can in the `crystals.gen` window find `syscom.CrystalsGL.cpp`, which basically shows what the sandbox nothing file uses. You can follow that up to `cmain.CrystalsGL.cpp` (don't get them confused) and see what you'd have to change in order to use your own `M-Core`. You only have to include the same files in the same order or otherwise meet their or your system's dependencies.

Within `__MSYS_PRIME__` you can find a label named `UNIT_LOOP__start`; And see a possibly somewhat suboptimal solution to a problem that maybe shouldn't exist – but herefrom a separate “thing” extends, the so called “`Meta`”, which you can think of as a system to get things to run inside of that loop without having to change it. And, or “but” withat “solution” come rules different to those that apply to that primary loop. Further then we can talk about “`things`” that use `the Meta` to get into that loop – which would be the third and for all I care final Level we could look at here. But this is also pretty much what we are working with. So. Now you might want to swallow this – or leave it where it is – but however open a new workspace and enjoy your wallpaper maybe. Got some music? I don't know how I ended up with a copy of 'Moon Safari' (Air), but uhm ... thanks to whomeverst the cause. ... hmm. ...

### micra Setup:

Here I've been running into problems setting up my home folder as a target because something in the scripting pipe didn't like to parse ~ as home. I use `/clx/bin` as target – and created a symlink (ln -s) of `sandbox.mic` to `/usr/bin/sandbox`. But using `/clx/bin/sandbox.mic` and then just 'up'ing through the history works just as well I suppose.

At the Moment: \_\_\_\_\_

We're here, in a way, where the bread is currently made. The **M-Core** and what's associated with it is a huge if or but – depending on whom you ask or what you want. As for me and what this is, we're OK with a little bit of bluntness. Not everything has to be bleeding edge. In turn we get is something that's maybe a little bit more comprehensive and easier to organize.

Well, on this new workspace open `crystals.gen/crystals.zero` – this to more specifically focus on `_CrystalsMeta.h`. If you care you can return to the previous workspace and locate `CrystalsMeta.h`, which, doesn't do much other than including the other.

Now – say we want to work on the **M-Core** itself; I have to say that using the **Meta** is the best way to do it. As it stands, the **M-Core** is part of a 20 Megabyte volume of RAM that is in and of itself designed to provide things such as access to input and graphics to some kind of “end profile”, herein codenamed “**Meta**”. So, to see what I mean, open *your “micra sources” folder* in a new sublime instance and open `sandbox.cpp`. Here, first replace the second line with the code you find in that file; And then replace the line that includes `cmain.CrystalsGL.cpp` with the includes from within it. Finally you need to update the include syntax (replace the “#” bracers with `<crystals.zero/#>`) - and that's step one to getting set up.

That is: The code your looking at does what it does. The main function is somewhere in there – and for motivational reasons we want to set ourselves a goal. So we do what all programming tutorials tell you to do at some point, which is to produce a very simple use-case. Say:

```
class MyAvatar
{
public:
    geo3D::vertex<float> position;
};
class Item
{
public:
    geo3D::vertex<float> position;
};
class MySandbox
{
public:
    ion_cue<Item> itemCue;
};
```

But of course it's not that simple. And a part of the trick is to know what each of those things is or ought to be – and that within the structure of the system or code. Code you may at this point have no idea or understanding of. And about that, let me tell you this: There's this posix upgrade ... which we'll have to wire in at some point. But before it really makes sense to do it, we have to first take care of some extras that'll be helpful then. So, for now ... get back to where you have `the meta open` – and get an idea of its structure.

Idea of Structure: \_\_\_\_\_

So, if you're taking this as a coding tutorial and you're as lazy as can be – you should at the very least be able to tell the difference between `<>` and `“”` includes; And be able to make sense of how you should change them when copying include code to a different location.

Going from the bottom up, you'd first of all need the Init and Terminate functions – and without doing anything but what I thought was the minimum, the program should just quit. Something doesn't work right. But that way we're also not going to really get anywhere. So, instead, move on and look at the `_CONTROL_FRAME_.CPP.H`. Look at it, call it a mess, act as though it stinks, click into it, press `ctrl+a` followed by `ctrl+c`, then move to the sandbox window, open a new file and paste everything into there.

So, change the `_metalog` macro to `__LOG__`, is easier to remember and nicer on the eye, remove the part checking for the/a Zero Core and change the log identity to a line and colors of your choice. But really ... remove the parts pertaining to Lumberjack. Also get rid of that separator while you're there. Hint: `DEFAULT` is a nice pick for identity backgrounds. And now you need a name for the thing.

The main reason why you want it to be declared as Metacore is access to its features as provided through `M` and also access to the class itself. You can however also not do that, and write the relevant code in the second half. However, you can consider it your first exercise to produce a new and independent Meta using only these two files. I propose something like

```
class __CITADEL__ : public _METACORE_
{
public:
    _ScreenUnit_          SU;

bool INITIALIZE (void)
{
    if(! __REGISTER_XS_UNIT__ (& SU,"ScreenUnit",-10)) goto RF;
    if(! SU.MAIN_INIT        ()) goto RF;

    return true;
RF:   return false;
};
int SHUTDOWN (void)
{
    return 0;
};

};
```

Using a simple copy of `crystals.sys/METASYS/Metabar.cpp` as Screen Unit, except maybe only pass a single zero to the `Create__ScreenAccess` function.

Contemplation: \_\_\_\_\_

What we have at this point is a fairly basic SDL environment. If you have it all in the first file we created, you can very easily find the function that is being executed during that `M_CORE` loop we looked at earlier. Important when using multiple windows is that you have to activate a window before drawing to it. So, I assume you otherwise removed METABAR from the system – in that case you don't have to add anything. The "Screen Unit" does, in this example, entail the entire runtime we have basically "allocated" so far. We could then rename the Screen Unit into `_Sandbox_RT_Unit_` and get ourselves some glory this way. But first we want to get Input up and running – and for that I suppose you first save your progress, check that it compiles – close the windows and return to the first workspace.

So, what you're looking at now might feel somewhat outdated. Depending on what it is you have open. But – in the `crystals.gen` window you can now close all files and navigate to `CrystalsGL`. Here just take a brief look – and understand that I myself tend to forget where what is and how things link up sometimes. I suppose it could work on its own, like, you can take what's there and be fine. But we're looking for `SDLZero`.

**Open/Enter a shell**, use midnight commander (`mc`) to navigate to your downloads/imports and to your current source. If you want to take a closer look at what's happening, I suppose you take the shell to a new workspace and use `subl -n` to open a new window, while using `subl -a` to then add files to it.

So, from the `posix` package you'll first need `inputstates.cpp` from the `LazyInput` package – which goes to `crystals.gen/crystals/inputstates.cpp` – and in `crystals.gen/crystals.h` you'll have to add it like so:

```
//-----#
#include      "crystals/memjet.cpp"
#include      "crystals/fsysaccess.cpp"
#include      "crystals/inputstates.cpp" //<line 170
#include      "crystals/IONOSPHERE.cpp"
```

Well, you don't have to have to – but `LazyInput` won't work without it. It adds one class for buttons, one for axis – and the magic box for key updates. So, every time an input triggers, the corresponding button struct is added to a cue. Then at a designated point, for me usually just prior to new updates, the cue is updated such that 'trigger' and 'release' tags are removed. The `SAFETY` thing is basically unnecessary and can be removed. The `StateUpdate` function is inline and should work just fine otherwise.

Then drop the other files into `crystals.gen/crystals.zero/_SYSTEM_/SDLZero` and add them to `SDLZero.PRIME.cpp` like so:

```
#include      "SDLZero/FancyGamepad.cpp"

#include      "SDLZero/LazyKeyboard.cpp"
#include      "SDLZero/LazyMouse.h"
#include      "SDLZero/LazyGamepad.cpp"

#include      "SDLZero/LazyInput.h"

//=====\\##\\-
class  SDL__ZERO
{      public:
```

- You do have to clean up the LazyMouse – but other than that, that’s part 1.
- For optimum performance be sure to leave struct sizes aligned to 8 bytes or use 8 byte alignment by default. Turning on 8 byte alignment by default can however drastically increase memory requirements that are difficult to tinker away.
- `__ms_ticks__` is `__sys_ticks__`. Maybe add it or change ... uhm ... . My bad, but ... uhm ...

Upgrades: \_\_\_\_\_

As for the **M\_CORE**, you can’t just patch it in, although you probably could and it should work just fine after adjusting all the things ... but as it stands I was myself a little under the weather and the codebase I was working on was a bit different, so we have to do this the normal way. Working it in manually and I suppose I’m again at fault for not working out a proper spinup.

But first you should add

```

{      public:
      LAZY_INPUT      LazyInput;

```

```

      bool      INITIALIZE      (void)

```

to your T Core Control (`crystals.zero/_SYSTEM/_ROOT/_T__.CPP`) and copy the corresponding input updates into the code. That would be `Lines 230-251` in `M_CORE.H`. And maybe rename Mface to something more ... sensible. Like `_meta_xs_`. You’ll have to run a test-compile to see if anything is broken.

If nothing explodes you can then replace the loop condition from counting to ten into checking for the runtime status. For the purposes of initializing this runtime status, I still propose a Mainloop init like so:

```

MAINLOOP__init:
      reg_Source = Mcore ;
      __M->runtime.STATUS      =      0x04;
      T__.t_CURR      =      __ms_ticks__ ;

MAINLOOP__start:
      reg__Dest =& __M->T_CONTROL;

```

along with the corresponding runtime struct and macro. [Breathe in ... breathe out ...](#)

If it doesn’t work ... try to return to the start – or the last functional state – and take it step by step. Sometimes that’s the secret. To crawl before you walk. To however be here where we can have a loop that ends on a manual input condition.

- > SERAPH is the key below the Escape key. Also known as Grave or whatever.
- > Use Seraph+Backspace to end

## Concussive Maintenance:

---

And yea. Now you've had a complete tour from the current back to front to the back again. And once more you may want to ponder the ifs and buts of working in the `M_CORE` directly versus now taking another route. But ... if it all works – you can then add the Delta Thread – based on the example. To do it the easy way:

First check out the includes and the runtime struct and remove/block `line 120` in the `M.DeltaThread.cpp.h` file. Then only add the init/shutdown functions to `M_CORE::INITIALIZE` and `M_CORE::SHUTDOWN` and if your machine supports multithreading you should see the thread starting and shutting down just fine.

And yea. Whether you add the code you want here or there ... what's the difference? Well – there certainly is one ... and we're exploiting it by taking things somewhere else entirely. Into ... yes ... yet another thread. We can't really do that in a single threaded environment. There comes then a point where this second thread would rejoin or otherwise sync up with the main thread – and at that point we'd have to substitutively do what we'd want those off-site processes to do for us – but so, first, let's look at the whole thing regardless:

```
M_CORE - Frame Start
{
  > Time Updates
  > Input Updates > Delta Thread ***
  > Unit Loop
  {
    Screen Unit :: main_exec
    {
      clear screen
      (***)
      update screen
    }
  }
}
M_CORE - ReLoop
```

To move on and draw a scene we can move around in, we need a projection matrix that takes care of the 3D projection math – and a transformation matrix that transforms the vertex coordinates we draw according to a position and orientation within that mathematical space that corresponds with our camera.

Lucky for us, the most difficult part about it is going to be to get some memory associated with that runtime that we have. Or in other words: Those classes we built initially, they'll now somehow have to get up there. A simple way would be to write something similar to all the other things. A class like so:

```
class _SANDBOX_
{
public:
  void (* _MyRuntime_) (void);
  memjet MEMORY;
  static bool INIT (void);
};
```

To add to your meta. Then create it using `malloc` or `__00mem__CREATE__` (`SYSTEM_CORE.cpp.h` `(15+16) > (181+182)`)... but since we don't really have the input working just yet, maybe we should look at that first. ... because ...

A simple Case: \_\_\_\_\_

In order to get the posix example to run, we need to basically “tell” the system what we want it to execute for us. One of the things you either removed from or added to the code in order to make it work pertains to the Delta Thread. Within the script function (lines 116 to 131) a mechanism has to be implemented; And in the M-Core the sync tag needs to be set. That is:  
adding

```
Mrun_.Delta.MSYNC__INPUT_UPDATE      ();    to M-Core  
and  {something else}                 to Delta Threads
```

where {something else} would be a

```
class m_CONTROL_GATE //M_CORE_SHELL.CPP.H line 20+  
{    public:  
  
    void      (*    _ControlUpdate_)      (void);  
    void      (*    _RuntimeUpdate_)      (void);  
    memjet    MEMORY;  
  
    static bool    _MAIN_INIT_            (void);  
  
};
```

where I highlighted in orange what would be the aforementioned “\_SANDBOX\_”. All we have to do in order to set it up, is to provide it to the **M-Core** – and then we have to change the **Delta Script** to call it. Our `Sandbox` class becomes the `_MAIN_CONTROL_UNIT_` - as there can only be one `_MAIN_INIT_` - and all we have to do there is to properly set up the memory.

```
if(!  _sandbox = (_MAIN_CONTROL_UNIT_*) __00mem__CREATE__ (MB__(250));  
_sandbox) goto RF;  
_M4META_ -> mControl = _sandbox;  
  
if(!  __REGISTER_XS_UNIT__ (&  SU,"ScreenUnit",-10)) goto RF;  
...  
  
_sandbox->MEMORY.setup(  _sandbox,MB__(250));  
reg__Option = _sandbox;  
if(!  _MAIN_CONTROL_UNIT_::MAIN_INIT_ ()) goto RF;
```

and

```
if(_sandbox) {  __00mem__DELETE__ (_sandbox); }
```

in our meta should do the trick. So, during that initialization `reg__Source` should still be pointing to the **M-Core** – which is why we can set it up. I use `_M4META_` as an alternative for `__M`, being sure that I have access to it in the meta. It’s a struggle – but ... for now I’m stuck with it. As for the orange line. Well, we can at this point still choose how we want the Delta thread to function. Using `reg__Source` will obscure M access from our m Control. Using `reg__Option` is ... certainly easier to use. We have to here however provide the access to the Main Init of the Sandbox so it can access itself. There it has to setup its own volume – by removing it from the Memory and shrinking it, before it can use the rest of it – but so ...

We can work with it:

---

Returning to sandbox.cpp we can add/change

```
#define _sandbox ((MySandbox*)reg__Option)
class MySandbox : public _MAIN_CONTROL_UNIT_
{
public:
    MyAvatar          avatar;
    ion_cue<Item>     itemCue;

    static void      my_control_update   (void);
    static void      my_runtime_update  (void);
};

bool _MAIN_CONTROL_UNIT_::_MAIN_INIT_ (void)
{
    _sandbox->MEMORY.mem__fromstart    (    sizeof(MySandbox));
    __sandbox->_ControlUpdate_         =    MySandbox::my_control_update;
    __sandbox->_MyRuntime_              =    MySandbox::my_runtime_update;
    return true;
};

void MySandbox::my_control_update      (void)
{
};

void MySandbox::my_runtime_update      (void)
{
};
```

and then all we need is

>M.DeltaThread.cpp.h:

```
__INPUT_READY = false;
reg__Option = __M->mControl;
((m_CONTROL_GATE*)reg__Option)->_ControlUpdate_ ();
goto _TAIL_SCRIPT_;
```

and >Your Meta

```
void _ScreenUnit::_exec_prime          (_ScreenUnit*_unit)
{
    reg__Meta = _M4META->_META;
    reg__Option = __meta->_sandbox;

    glClearColor(0.0f,0.5f,0.9f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    ((_MAIN_CONTROL_UNIT_*)reg__Option)->_MyRuntime_ ();

    _unit->_ScreenXS->UPDATE();
};
```

I only highlight reg\_\_Option because it is important, here, that both threads use the same register; And each has to set it individually. And there goes me not trying to confuse you. Well – anyhow, that’s the difficult part done ... I suppose. You can then move the glClear\* lines to the new runtime function and change the color to see it in action. Just so you know that something actually changed.

In hindsight:

---

Well, if you got it to work – congratulations. It wasn't difficult, but it wasn't for free either. Well, what is difficult? Anyway, you can be proud of yourself. Especially if you got here beforehand already. But yes, this is where you would be at – had you followed along and overcome the challenges. The next part then ... is really simple. Really. So ... to your sandbox add:

```
class MyAvatar
{
public:
    FancyGamepad          gamepad;
    geo3D::vertex<float>  position;

class MySandbox : public _MAIN_CONTROL_UNIT_
{
public:
    MyAvatar              avatar;
    ion_cue<Item>         itemCue;

    geo3D::matrix<float> mat__Projection;
    geo3D::matrix<float> mat__Transform;
    geo3D::matrix<float> mat__Offset;
    geo3D::matrix<float> mat__Rotation[2];
    float                rotation[2];

    bool                 MatrixReady;
    dega                 dRES[7];

bool _MAIN_CONTROL_UNIT::_MAIN_INIT_ (void)
{
    ...

    //Fancy Gamepad
    __sandbox->avatar.gamepad.memInit__Registers (& __sandbox->MEMORY,50);
    __Tx->LazyInput.gamepad.MakeFancy           (& __sandbox->avatar.gamepad);
    __sandbox->avatar.gamepad.InitIntegrity      ();
    /*
        Here I create 50 "registers" – which are cells arranged as a wheel. Whenever a new input
        occurs – or in case of the motion stick: when the registered direction (8 directional) changes
        – a new cell is "filled out" and the wheel is incremented. Basically you'd only need them if
        you wanted to keep track of such things – like for Street Fighter type combos. Maybe. Maybe
        other things.
        Make Fancy then maps the lazy gamepad onto the fancy one, or the other way around. In essence
        the "fancy buttons" are linked with the lazy ones – so it reads them out as a second pass.
        Integrity is just to finalize the setup. It could be hidden away. As a part of make fancy. I'm
        not sure if the order is strict.
    */

    __sandbox->mat__Projection.Projection3D      (9.0f/16.0f,__PI__*0.3f,1000.0f);
    /*
        This and the following is mostly math as found in the geo3D package part of crystals. We
        here pass aspect ratio (A/V), Field of View and a depth. Depth is only relevant if you do
        anything involving the Depth Buffer (which would basically be always) as it determines the
        resolution of the depth – as to set a draw distance beyond which things aren't "in scene"
        anymore.

        Here we also right away load the created Projection Matrix into OpenGL.
    */

    glMatrixMode (GL_PROJECTION);
    glLoadMatrixf (__sandbox->mat__Projection.q);
    glMatrixMode (GL_MODELVIEW);
    /*
        Then we set up the matrices we'll use/need during runtime.
        First the rotation matrices. It is important that we setup their Identity, because the
        runtime function we'll use only sets up the values it changes. The same goes for Offset. The
        "Matrix Ready" bit – uhm. I guess "we don't like to do this" – to say, add a "halt" condition
    */
}
```

to anywhere in the loop to get synchronized with another thread. We'll get to it later. And finally we just set up some initial rotation values for the camera.

```

*/
__sandbox->mat__Rotation[0].Identity      ();
__sandbox->mat__Rotation[1].Identity      ();
__sandbox->mat__Offset.Identity          ();
__sandbox->MatrixReady                   =    false;

__sandbox->rotation[0]                   =    __PI__ * 0.3f;
__sandbox->rotation[1]                   =    __PI__ * 0.125f;

return true;
RF:   return false;
};

void   MySandbox::my_control_update      (void)
{
__sandbox->avatar.gamepad.UPDATE        ();
/*
simple enough.
*/
__sandbox->rotation[0]      +=    (__sandbox->avatar.gamepad.LOOK[0]/1200.0f) * T__.t_HydVec;
__sandbox->rotation[1]      +=    (__sandbox->avatar.gamepad.LOOK[1]/1200.0f) * T__.t_HydVec;
/*
As written, the axis are ranged scaled to 1200. So, perhaps that should change. So, here we
normalize the values to 1 and multiply it by hydrons. This should give us 1 radian of rotation
per hydron. [0] ought to be the a axis, so, rotation along the up-down axis. [1] is v – that
is, rotation around the right-left axis. So, that's the looking up and down part. A is the
direction.
*/
__sandbox->mat__Rotation[0].setRotation__A    (__sandbox->rotation[0]);
__sandbox->mat__Rotation[1].setRotation__V    (__sandbox->rotation[1]);
/*
I always get confused about plus and minus here. Now, technically we have to update the
position, but if you want to do this right, you'll need a little something extra. See below.
*/
__sandbox->mat__Offset.m[3][0]    =-    __sandbox->avatar.position.q[0];
__sandbox->mat__Offset.m[3][1]    =-    __sandbox->avatar.position.q[1];
__sandbox->mat__Offset.m[3][2]    =-    __sandbox->avatar.position.q[2];
/*
As with all things too mathy ... but ... first we multiply the rotations together
*/
geo3D::matrix<float> matX;
matX.Multiply                    (&    __sandbox->mat__Rotation[0],
&    __sandbox->mat__Rotation[1]);
__sandbox->mat__Transform.Multiply (&    __sandbox->mat__Offset,&matX);
/*
Then apply them to the offsets – and ultimately add 10 for good measure. Uhm, well. No, this
adds 10 to the 'g' transformation, essentially moving the result 10 "units" further into the
distance. All ... relative where 1 is the closest anything can get to the screen before
disappearing into mathematical obscurity.
*/
__sandbox->mat__Transform.m[3][2]    +=    10.0f;
__sandbox->MatrixReady                =    true;
};

void   MySandbox::my_runtime_update      (void)
{
glClearColor(1.0f,1.0f,1.0f,1.0f);
glClear(GL_COLOR_BUFFER_BIT);

//   RELOOP:
//   if(!__sandbox->MatrixReady) goto RELOOP;
//   __sandbox->MatrixReady = false;
/*
We can do without the reloop. Maybe the input function updates the matrix while or after it is
being used to update the scene ... but then it'll be right the next frame. It's fine.
*/
glLoadMatrixf(__sandbox->mat__Transform.q);
/*

```

So we only need to load the transformation matrix and can then move on to draw a scene:

```
*/
glBegin(GL_LINES);
glColor3ub(0x00,0x00,0x00);

    glVertex3f    (0.0f, -10.0f, 0.0f);
    glVertex3f    (0.0f, 10.0f, 0.0f);

    glVertex3f    (0.0f, -10.0f, 2.5f);
    glVertex3f    (0.0f, 10.0f, 2.5f);

    glVertex3f    (0.0f, -10.0f, -2.5f);
    glVertex3f    (0.0f, 10.0f, -2.5f);

    glVertex3f    (0.0f, 0.0f, -10.0f);
    glVertex3f    (0.0f, 0.0f, 10.0f);

    glVertex3f    (0.0f, 2.5f, -10.0f);
    glVertex3f    (0.0f, 2.5f, 10.0f);

    glVertex3f    (0.0f, -2.5f, -10.0f);
    glVertex3f    (0.0f, -2.5f, 10.0f);

glEnd();
/*
And that is that.
*/
};
```

As for updating the position, here's a way:

```
float vec_v[2];
float vec_g[2];
float mov_v  =-    (__sandbox->avatar.gamepad.AXIS__MOVE._AXIS[0]->VOLUME / 1200.0f) * T___.t_HydVec;
float mov_g  =-    (__sandbox->avatar.gamepad.AXIS__MOVE._AXIS[1]->VOLUME / 1200.0f) * T___.t_HydVec;

vec_v[0]     =    mov_v * __sandbox->mat__Rotation[0].m[1][1];
vec_v[1]     =    mov_v * __sandbox->mat__Rotation[0].m[2][1];

vec_g[0]     =    mov_g * __sandbox->mat__Rotation[0].m[1][2];
vec_g[1]     =    mov_g * __sandbox->mat__Rotation[0].m[2][2];

__sandbox->avatar.position.q[1]    +=    (vec_v[0]+vec_g[0]);
__sandbox->avatar.position.q[2]    +=    (vec_v[1]+vec_g[1]);
```

After all this I'm getting weird issues with the rotation – like, the controller still sending a signal - ... sigh. Whatever. Maybe also do add

```
glPointSize(5);
glBegin(GL_POINTS);

    glColor3ub(0x00,0x7F,0x00);
    glVertex3f    (
        __sandbox->avatar.position.q[0],
        __sandbox->avatar.position.q[1],
        __sandbox->avatar.position.q[2]
    );

glEnd();
```

And the rest is up to your imagination. So, on one side the point was that technically all one has to do is use OpenGL – and the same goes for using textures. Which is why there isn't much in terms of 2D drawing just yet. Because I'll be using OpenGL for that too. And for that we'll first have to write ourselves a shader – and there's lot's of fun things we can do with that. So – is this "it"? It depends. But for what's next ... from my end ... this is sort of what it's all about.