Setting up a simple Station Terminal

Check here »www.frictionalgames.com/forum/thread-31645.html«

To the people who actually have been waiting for this page's completion, I AM SO SORRY. I only recently got an actual internet connection so expect things to speed up in terms of page completion. Again, sorry for the wait, but the page has been expanded enough for you to add a text box! Check the 'Terminal still isnt shiny' section for details, and I'll add more later.

What is in this tutorial, and what should I know/do first?

(Note: This guide is pretty big, but be patient and you will be able to set up your own terminal!) After much demand, here is a guide on how to set up a basic terminal in SOMA.This guide will cover these concepts:

- Setting up your script/map for a terminal
- Basic terminal logic: how the game reads what you code, and where to code it
- How to use some of the fast functions for GUI included in SOMA, such as a main menu, back button, adding dialog, displaying an image, etc.

The guide will not cover more advanced topics like password input, video cameras, locking doors, etc. Those will be in a **later guide**, so if you're searching for an advanced topic like one of the ones I've listed then be patient until a guide is made for them. Keep in mind, there are few limits to what you can pull off with a terminal; I'll be teaching you how to quickly set up the normal terminals seen throughout SOMA, but you can also setup GUI for things like the laptop at the start of the game, and, as far as I know, much more advanced and interactive GUI than the basic stuff if you have the patience for it.

I also recommend you know the following things first:

- How to set up codelite! I really have to stress this one. It will make your life easier if you use it properly.
- Make sure you've set up your mod. Check here: wiki.frictionalgames.com/hpl3/community/hpl3_getting_started
- How to use language files in SOMA. Not necessary, but I recommend you have an english file set up (or whatever language you prefer.) There isn't a guide for this yet, but there will be sooner or later. Reading the guide for Amnesia for this might help out, as it uses the same format.
- Basic understanding of C, and how to read error messages. Seriously, knowing what a simple error message means will help you debug, and knowing the different variable types in C and the function types (like what is a void, a bool, or an int function?) If you know this stuff, than you'll also have an easier time learning new code on your own. You could probably just google 'learning basic C' to find some quick tutorials.

After you've done ALLLLL this crap, or the things you think are necessary, then follow me to the basic setup.

Just let me add my terminal already!

Alright. So to start off, make sure you have a map set up with the terminal you want. Name the terminal something you'll remember thats code friendly. (IE coolTerminal, cool_terminal, etc) If you select the terminal, you should see something like this in it's entity tab:

If not, then find another terminal to use that has this menu. (I imagine there are ways to create any entity into a terminal with the model editor, but I have not looked yet.) Make sure your terminal is once again properly named, and then in terminal window find the boxes entitled 'OnGuiFunction', 'EnterCallback', and 'LeaveCallback'. These are the callbacks that we will primarily use in our code to run the terminal. I recommend clicking the generate button beneath each, but if you prefer you can name each callback yourself. Also make sure to check the 'allowInteraction' box if the player can interact with the terminal. Otherwise they can only look at it. Once the boxes have names, generated or otherwise, save your map and open up your script. Your terminal on the map side of things should be done!

Script Setup

This tutorial assumes you have a basic .hps file set up with your map already. Scroll down to the bottom of the script and you may see a section like this:

```
        /////////////////////////////////////////
    // ==============
    // TERMINALS
    // ==============
    //{//////////////////////////////////////

        //-------------------------------------------------------

        /////////////////////////////////////////
        // Terminal *Name Of Terminal*
        //{//////////////////////////////////////

        //-------------------------------------------------------

        /*Put any variables that are only used Terminal here.*/

        //-------------------------------------------------------

        /*Put any functions that are only used Terminal here.*/

        //-------------------------------------------------------

        //} END Terminal *Name Of Terminal*

        //-------------------------------------------------------
    //} END TERMINALS
```

This is the template setup. I do not recommend you delete this from any script! It is good reference and a good starting point for your first terminal in any map. Also, check the part of your scipt where the #include's are; add the lines:

```
#include "helper_imgui_station.hps"
#include "helper_imgui_station_app_audioplayback.hps"
#include "helper_imgui_station_app_photoviewer.hps"
```

These will include some functions that can quickly setup SOMA's terminal's GUI, which you will need if you directly follow this terminal's setup. Now, going back to the template for your terminal, edit it to look something like this (you dont have to copy the *comments, but do read them over as they contain information): /*

```
         /////////////////////////////////////////
     // ==============
     // TERMINALS
     // ==============
     //{////////////////////////////////////////

        //--------------------------------------------------------
        /////////////////////////////////////////
        // Terminal room_terminal
        //{//////////////////////////////////////
        //--------------------------------------------------------
        void room_terminal_OnGui(const tString&in asEntityName, float
asTimeStep)
        {
            //variables
            int lActiveApp = -1; //these variables are only necessary if
your terminal will have a menu with multiple sub-programs. This variable
indicates the active app on the terminal.
            int lBackApp  = -1; //This variable indicates what app the back
button will bring you to.
            bool bDrawBackButton = true; //This variable is a true or false
that determines if the terminal will draw the back button.
            //you can of course create more variables as necessary, but
these are the basic ones for a terminal with multiple menus.

            //------------------------------------------

            //this is where the main code will be written.

        }

        //------------------------------------------

        void room_terminal_GuiEnter(const tString&in asEntityName)
        {
            //This code runs whenever the player enters the terminal.
        }

        //----------------------------------------------

        void room_terminal_GuiLeave(const tString&in asEntityName)
        {
            //This code runs whenever the player leaves the terminal.
        }

        //--------------------------------------------------------
        //} END Terminal room_terminal
        //--------------------------------------------------------

    //} END TERMINALS
```

(Do not directly copy this code as it was indented with spaces instead of Tab)

Make sure you replace my lines as necessary. Your 'Terminal' and 'End Terminal' comments should have your terminal's name instead, and the callbacks should be what you named them in the editor. Reference which ones are which by reviewing the image of the terminal tab above (in the 'just let me add my terminal already' section). Also decide if your terminal will have a menu with multiple programs. If not, don't bother writing in the variables above the functions section.

Now the basic code for a simple terminal is setup, and we can adjust it for our needs in the next section. Don't bother testing your map, as in its current state the terminal won't do anything.

My terminal still isn't shiny! What do?

It doesn't do anything fun yet, but your terminal *is* running. Now you just have to decide what it runs. Most of this stuff will be dealt with in the void terminal_onGui function. This part of the tutorial is about setting up a single window stylized like the regular SOMA terminals which will display text and an image. I'll provide the code and then explain all of it with detailed comments and insight. For this part of the tutorial, we WILL ONLY be looking at the onGui function. We'll keep adding bits of code on as we go along so you can gain a better understanding of how the individual lines function. Start by adding the following to your variables section:

```
cImGuiWindowData windowData = StationGui_CreateDefaultWindowData();
//Creates default window data which is stored in a class
cImGuiTextFrameData textFrame = ImGui_GetDefaultTextFrame(); //Creates
default textbox data and is stored in a class
textFrame.mFont.SetFile("sansation_large_bold.fnt"); //sets the font
textFrame.mFontAlign = eFontAlign_Left; //Aligns the text to the left side
of the text box. You can also align it to the center or the right side of
the text box.
textFrame.mbUseBackgroundGfx = false; //Default is true. If true the
background behind the frame will be filled in with a solid color and it will
```

```
look like it was made in MS paint.
textFrame.mFont.mvSize = ImGui_NrmSizeKeepRatio(0.04); //Sets the font size
to 4% of the screen size.
```

This creates an ImGuiWindowData class object, as noted by the c, named windowData, with the default window parameters. These parameters include things such as font coloring and sizing, window coloring, and padding (I suspect these are the window borders?), as well as some GFX properties that require further investigation. The following line creates a class called textFrame that we will be using for our textbox. Everything beneath that line tweaks the text frame's properties. Then, add the following lines to the onGui's main code section:

```
ImGui_SetTransCategory("");
StationGui_DoWindowStart("WindowTitle", windowData, ImGui_NrmPos(0.05, 0.05,
2), ImGui_NrmSize(0.9, 0.9), false);
ImGui_DoTextFrameExt("This is a text box.", , , , textFrame,
ImGui_NrmPos(0.025f, 0.025f, 2), ImGui_NrmSizeGroup(0.95, 0.95));
StationGui_DoWindowEnd();
```

The first line, ImGui_SetTransCategory sets the category used in your language file. If you use “ ” as the input which I have done, it will directly take string input for your functions. The next line actually initiates our window; it sets up its title, location, and sizing, while using our windowData class as a base. The ImGui_NrmPos(cVector3f()) and ImGui_NrmSize(cVector2f()) functions I will explain momentarily. The following line , ImGui_DoTextFrameExt, draws in a text box within our window. If your language category was set to “ ”, then whatever you write in quotations will be the text displayed in the text box. If you set the language category to an actual category within your mod's lang file, then write the entry name within the quotes instead. The 3 0's I still need to look into, but the following textFrame value is the class we created earlier for our textbox. The ImGui_NrmPos and the ImGui_NrmSizeGroup I will now note on.

As mentioned above, there are some values inputted into the program named ImGui_NrmPos, ImGui_NrmSize, etc. NrmPos is the item's postion on the screen using a 3d vector input. Simply put, the first value, in a range of 0-1, is how offset the object's x coordinate is, with 0.5 being centered, 0 being at the exact left side, and 1 being at the very right edge. The second value within the is the object's y offset, with 0 at the top, 0.5 centered, and 1.0 at the bottom of the display. The third value is the layer; object with a value of 1 here draw in first, and object with a value of 2 second, and so on. (Keeping in mind that objects draw on top of each other, so drawing in second means the image will appear on top of whatever drew in first.) ImGui_NrmSize is the object's scaling relative to the screen, with the first value being how large it's x value will be and the second it's y. If you use NrmPosGroup or NrmSizeGroup, than the objects positioning or scale will be relative to whatever it is drawn to; so, if drawn within a window, it's size is based off of it's parent window instead of the overall screen.

If you don't understand this, don't worry; ask a more experienced coder on the forum to help explain, and for now just copy this code. There's only so much I can get across with a wall of text.

The last line is equally important to all the others, especially if you are using multiple windows. I haven't tested this yet, but I suspect the game will crash if you do not include this last line.

From:
https://wiki.frictionalgames.com/ - **Frictional Game Wiki**

Permanent link:
https://wiki.frictionalgames.com/hpl3/community/scripting/terminal_basics

Last update: **2015/11/10 07:15**