**How Does De\_Flux Fucking Work?**

A brief introduction by Jensen.

**Motivation:**

De\_Flux is a work-in-progress. If you want to be able to work on this project with me (Jensen) then you will need to have the following things:

* A proficiency in C. Also some exposure to the SDL graphics libraries.
* A proficiency at programming in general
* The ability to think logically and creatively.
* **An understanding of the core elements of De\_Flux**

I will assume you already have attained the first three items on this list. This short document is supposed to be a guide through attaining that last one.

Understanding the Core Elements of De\_Flux

[**1.** How the Tile Sets and Item Sets Work 2](#_Toc361151780)

[**2.** The Data Structure of Maps 6](#_Toc361151781)

[oneTile 6](#_Toc361151782)

[oneItem 6](#_Toc361151783)

[mapPortal 6](#_Toc361151784)

[map 6](#_Toc361151785)

[**3.** How Maps are Printed to the Screen Using SDL 7](#_Toc361151786)

# How the Tile Sets and Item Sets Work

The tile\_set and item\_set images are very important. They both work in the same way, so I will explain their functionality using the tile\_set as my example.

The tile\_set looks like this:



I have set up a system by which you can easily print these tiles to the screen using functions. For example, if you want to print the green portal in the upper left hand corner, you would simply use the command:

// applies the green portal texture to the grid at grid-coordinate (x=0,y=0)

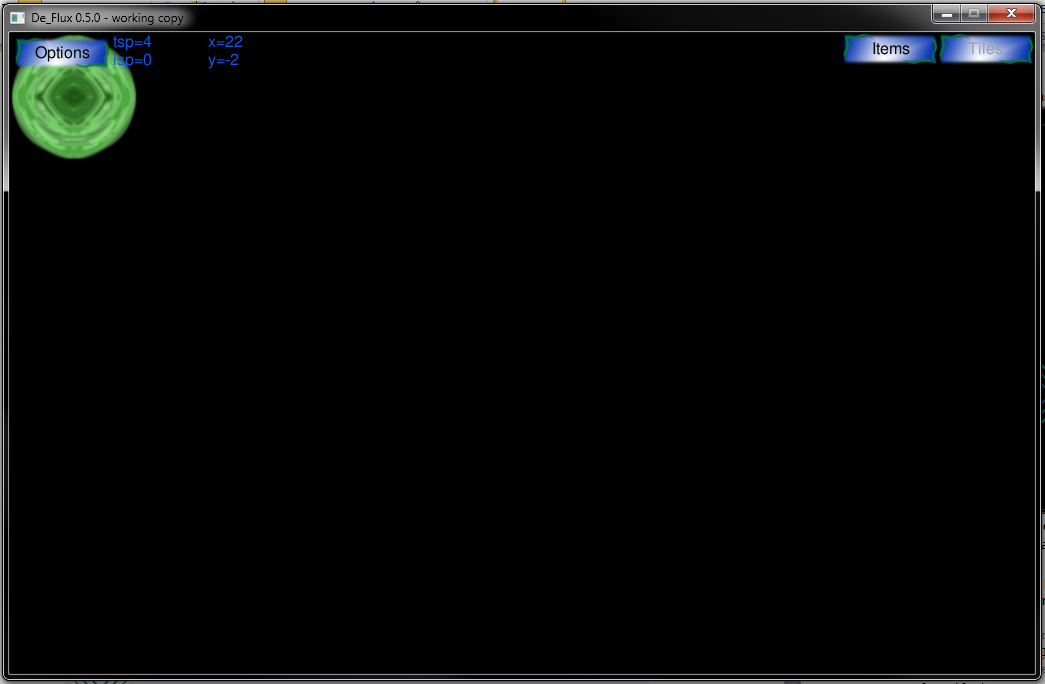
apply\_tile(t\_portal\_green, 0, 0);

// updates the screen

SDL\_Flip(screen);

*It is important to remember to flip the screen if you want your change to show up.* ***apply\_tile()*** *does not automatically show the changes.*

That code would produce this:



*Notice there is a green portal in the top left corner of the screen at coord (0,0)*

Now if you wanted to print a stone to the screen on the right of the green portal, you will need to run this command:

//applies the green portal to the screen at grid-coordinate (1,0)

apply\_tile(t\_portal\_green, 1, 0);

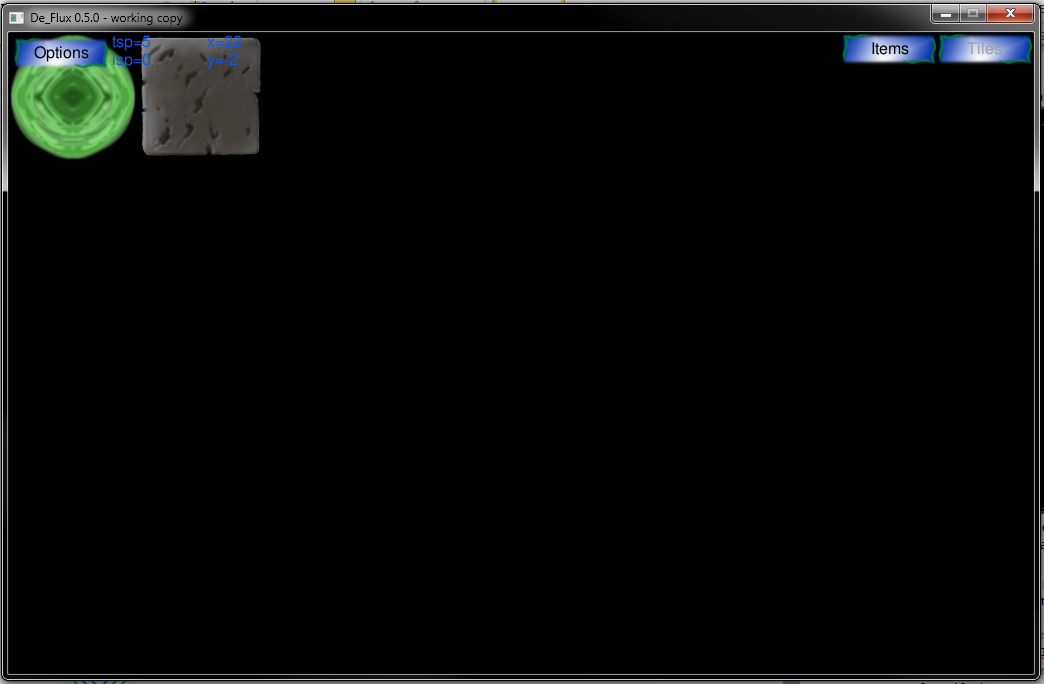
//applies the stone to the screen at grid-coordinate (0,0)

apply\_tile(t\_stone, 1, 0);

// updates the screen

SDL\_Flip(screen);

That would produce an image like this:



*Notice that in addition to the portal, there is now a stone to the right of it at coord (1,0)*

That is how you print tiles to the screen.

Another thing you will need to understand is how De\_Flux addresses the tile textures. In other words, we need to know how De\_Flux cuts the portal and stone images from the tile\_set and pastes them to the screen.

To address a tile, you will use a 4-bit hexadecimal number.

Don’t be frightened. All that means is that De\_Flux uses a number like **“0xabcd”** to access the tiles. Hexadecimal numbers are very easy to use once you get accustomed to them.

Lets break down what this hex number is:

**“0x” –** This is the prefix. This identifies the number as hexadecimal. It this isn’t here, the number ceases to address the correct tile texture. De\_Flux may print the wrong tile if you set this wrong, but more likely, it will simply not print anything.

**“ab” –** These are the first two hexadecimal bits. They tell De\_Flux how far TO THE RIGHT it has to go to from the upper left tile to find the appropriate tile. For example, if these two hex-bits were set to “03” De\_Flux would go to the column with green portal and the closed chest in it (see tile\_set).

**“cd” –** These are the last two hexadecimal bits. They tell De\_Flux how far DOWN to go from the upper left tile. If these were set to “01” it would go down to the row with the “SKIP” tile at its beginning (see tile\_set).

This image graphically shows what each address is of each tile:



Now would be a good time to look at the “tile\_and\_item\_definitions.h” header file and take a peek at the definitions. You should find that the names of the tiles correspond to the hexadecimal addresses shown here.

Let’s pretend that you made a new texture for a bush. Just a simple green bush. You can place it anywhere on the tile\_set that you want. There are only a few things you need to do to add your bush to the existing tile\_set:

1. Make sure you don’t place it over any existing tiles! That would be a horrible idea.
2. Make sure that your texture fits completely in a 128x128 area.
3. Align your texture on the 128x128 cell boundaries. i.e. make sure your tile doesn’t overlap with other tiles. It needs to fit inside it’s own 128x128 area on the 128x128-pixel grid of the tile\_set.

Well, that should be enough information about the tile\_set. You now understand the basics of that system enough to be able to work with it on a basic level. You can always learn more if you want to work on the system itself!

# The Data Structure of Maps

Let’s first find where the map structure is declared. Find it in “globals\_and\_headers.h”. You will see four structures:

1. oneTile
2. oneItem
3. mapPortal
4. map

Get a good look at all four of these. You should read through the comments for these structures completely. Seriously. Don’t skip that step. Nevertheless, I have written up an explanation of what each structure does:

## oneTile

This is a structure that describes a tile. We can set all the essential information: its x-location, its y-location, and what type of tile it is.

## oneItem

This is a structure that describes an item. We can set the x-location, the y-location, and what item it is.

The oneItem and oneTile structures are brain-dead-simple. mapPortal and map are where the real meat of this section is laid out.

## mapPortal

This is a map portal. It holds all the information needed to teleport from one map to another. We can set the coordinates of the entrance on the current map, the coordinates of the exit on the next map, and the name of the next map.

For example, let’s take a look at what would happen if we made a portal on our currentMap like this:

//declare the portal

currentMap myPort;

//set old coordinates

myPort.oldCoor[0] = -3; //x

myPort.oldCoor[1] = 10; //y

//set new coordinates

myPort.newCoor[0] = 20; //

myPort.newCoor[1] = -9; //

//set next map name

strcpy(myPort.nextMapName, “adventureMap2”);

This would initialize a portal that starts on at coordinates x=-3, y=10 and teleports you to the map named “adventureMap2” at coordinates x=20, y=-9.

Note: “maps\adventureMap2.map” must exist or else you will get an error when you try to use the portal in-game.

## map

The first thing to know is this: when you are playing on a map or editing it in the map maker, you are editing data in the currentMap map. When the user is making a map with the map maker, De\_Flux is working with and manipulating data in that currentMap array.

1. Each map will have a name.The name is just a string (an array of characters).
2. Maps can have tiles on them. The tiles are stored in the currentMap.tiles[] array. There can be anywhere from 0 to MAX\_TILES\_PER\_MAP on any given map. You can find the definition of MAX\_TILES\_PER\_MAP in “globals\_and\_headers.h”. De\_Flux keeps track of how many tiles a map has with the tsp variable. It stands for “tile stack pointer” because the tile array works like a stack. tsp tells you how many tiles you have placed down. Initially, where there are no tiles, the tsp is set to zero. But after put a tile into currentMap.tiles[0], you will need to increment the tsp with currentMap.tsp++. Because of the nature of the tsp, you can add a tile to the map by indexing into the tiles array with it like so: currentMap.tiles[currentMap.tsp]. Once you set the tile at currentMap.tile[currentMap.tsp], you can then increment the tsp with currentMap.tsp++. That will ensure that the next time you try to add a tile to the currentMap.tiles[] array, it will be adding it to a new space (and not overwriting your last tile addition! :S).
3. items work the same way as tiles. Instead of using the tsp, you use the isp (item stack pointer). The same process is applicable to add items to the map’s item array. Put an item into currentMap.items[currentMap.isp], then use currentMap.isp++.
4. Maps can have up to a set number of portals. They can have anywhere from 0 to MAX\_PORTALS number of portals. The portal structures are arranged in an array. The pspvariable (portal stack pointer) is the variable that tells you how many portals you have used. It is initially 0. After you create the first portal on a map (which is stored in the currentMap.portals[0] structure), the psp is incremented to 1. Because of this nature, it is very useful to use the psp to index into the portals[] array. I.e. you can modify the portal at currentMap.portals[currentMap.psp] and then simply increment with currentMap.psp++.

That is enough information about maps to give you a good base of information on them. Now you know the gist of how maps work in De\_Flux.

# How Maps are Printed to the Screen Using SDL

Rejoice! This is a short section. That is because it relies on what you have read in the previous two sections.

Maps are printed to the screen using the print\_map(struct map \*datmap) function. This function accepts a pointer to a map.

You can find this function in “graphics\_functions.h”

These are the tasks that print\_map() does:

* Figures out how large the window is.
* Rifles through the tiles array to find tiles to print. It stops looking for tiles once it has reached tsp or MAX\_TILES\_PER\_MAP. It will only print tiles if they are within the viewing window, as there is no need to render textures we cannot see.
* Rifles through the items array to find item to print. It stops looking for items to print once it has reached isp or MAX\_ITEMS\_PER\_MAP. It will only print items if they are within the viewing window. It is pointless to render textures that we cannot see.
* Print the spawn tile on top of all other tiles.

That is all it does.

This function is (at the moment) not called by either the map\_maker\_environment() or new\_game() functions. Both actually call a function entitled, “update\_map()” which operates a few small features on the side, and then calls print\_screen() itself. It can blank the screen and display the player. You can find declaration of this function on the “graphics\_functions.h” header.

Well, that is it. That is the core material of De\_Flux. There are many, *many* more details to the program than what was presented here, but this is the basic idea. I hope this helps you understand the program!

If you have any questions or comments, send them to me at JensenR30@Gmail.com