Robots Part 4

- This isn't really a 'robots' class, but we will use the robots game as a base.
- Our goal is to import and use a sprite sheet.
- We will need to make a SpriteSheet class.
- We will need to make a Sprite class.
- ALL objects will be drawn by blitting the sprite. The sprite class will pull the correct image for animation.

General Overview

Class SpriteSheet

This class will be created with the name of an image file and the dimensions of the sprites inside. It will then go through and categorize sprites by number. Rows are flattened into one row by attaching subsequent rows to the first row. The SpriteSheet will then contain an array of images, the index to which is the sprite number. That is all the SpriteSheet has to do; to be able to return a cut (cropped-in) image based on a number.

Class Sprite

A Sprite will have an array of images. It's purpose is merely to hold and manage the images for animation. The sprite class will return the correct image because it is given an index number for the array of images. Therefore 'Sprite' will be used in a parent class which contains sprite-specific information. I.E. a 'mario' character class would be responsible for requesting the correct image based on the state of the character. This is going to have to be hardcoded, but the good thing is that once you define the state of the character you know there will be a certain number of images found on the sheet in order, so it will not be something overly onerous to hardcode.

Thing Classes

As mentioned above, we now need a 'thing' class to track sprite animation. We will need to make Class Stone, Class Wall, Class Robot, and Class Player. The locations of every object are now to be stored in arrays. This will allow faster map drawing and faster collision detection because we only need to worry about the arrays of objects and not every little square on the map.

In fact it may be possible to 'not use' a map at all, if everything is kept in arrays. The only trouble with that is that without a game map, proximity checks (such as while moving) become difficult.

The solution to this is to create a 'tile' class which holds everying in that tile. Thus, gameMap becomes a 2d array (list!!) of tiles, and there is no wasted space here.

Imperative versus OOP Programming

What we are really doing here, on another level, is to refactor the code in an 'object-oriented way'. It would certainly be possible to add a sprite sheet using imperative programming, but the complexity of the code will keep rising and rising. The real trick, thus, is to understand why we are using classes.

It's to collect relevant information. Not to provide methods.

So for example if we want to add robot hit points, in the old gameMap 'r'/'R' based model we need a separate 2d array to hold monster hit points, or we need a way to attach it to the gameMap – such as 'r7' for a robot with 7 hit points – but then only print the 'r'. This kind of programming is known as a 'kludge'. Eventually either codependent arrays will be used, or something akin to 'struct'. Therefore, the easiest way to start using classes is to treat them as structs and try to collect information only. This allows us to use one array and not codependent arrays and the code will feel very 'comfortable'. It is likely never necessary to move past this point in optimization unless there is a serious problem with the code or th logic of the code.

How to start

First write the SpriteSheet and Sprite classes and have them draw the primary image in drawGame().

Next add thing classes, which contain a character – and flags for 'seen' etc. so we don't need to do r/R (or just use r/R). Then start by adding a class to gameMap and not a character. Don't worry about a 'tile' class for now, but you could do that now, if required. Just add tiles to the map in setup and then add things to the tile's inventory. This also eases collision detection because we can first check if 'anything' is there vs. checking many things individually, or check only places which contain more than one object (instead of every space).

This, if done correctly, should only take one class (1.5h) with Roger.