HOW 2 DRAW

At the moment, there are 2 functions: drawing every pixel, or every hollow circle.

There are other functions for guns GUI, the player, and the nodes.

We’re going to need much more complicated drawing functions – rockets, full circles, other polygons.

Therefore, it looks like there will need to be a draw function for every class that will have objects drawn.

Generally, this function will allow ALL objects to be put into a ‘draw list’ where the ‘draw’ function for each object will be called. This will be different for every object, of course.

EG: if object in ‘drawlist’ is a bullet: call bullet.draw. This function will need to do the ‘fade’ and ‘draw’ operations. Fade: gets the colour passed in, and makes it dimmer (closer to black). This is generic for every object. The draw function is simply pygame.draw.circle() or whatever. You can do that ez.

At the moment, the game isn’t drawing objects; it’s drawing the fade stuff as well. The object’s ‘draw’ is being put into the appropriate list. How about every object has a list of fadeObjects? For each object, for each thing in that list, do the fade thing, draw the fade thing? Seeing as nothing reacts to the fade stuff, they can be well-embedded.

What needs to be put in the list, and when?

|  |  |  |
| --- | --- | --- |
| **Object** | **Made when?** | **How to add to drawlist** |
| Player | Beginning | On scene creation |
| Nodes | Beginning | Scene creation |
| Bullets | Made by gun objects after pressing button | If bullet is made, pass flag back to the main loop. |
| Monsters | During waves, at intervals when old wave is clear | Pass list into monster creation function, fill it as each monster is made. Pass list back, fill the main list with it. |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

drawObjects are the base class of anything that is drawn. It contains a list of past drawn ‘sprites’. For each sprite, it fades the colour, determines whether or not to be removed from the ‘drawList’, and then draws the object at its position. Therefore, the position of the object is to be defined here, but manipulated further up, of course.

7-7-17

I’ve looked at pygame’s sprite stuff, you can define the shape and colour etc of an object once, and draw everything with one line. Every object has an update function, which can be used to change the rotation or colour. The ‘draw\_circle’ or draw\_rect’ is used, but the parameters can be changed.

The draw function apparently gets the object’s image and rect variables and blits it to the screen.

Each object can have its own group for fadey objects. These objects must also be sprites, with rects and images. The colour needs to be extracted from the image so that the fade function can be done.

I did it, but when the character moves, it leaves a very bright mark on the node, where the character was. Is this because a lot of sprites are on that spot, adding the alpha to its maximum? You’ll have to remove all fadeSprites on a spot if a new one is made on that spot. You only want the freshest one.

9/7/17

When a sprite is killed, all its fadey bits die as well, and therefore don’t fade away.

If there is a fader manager, with a universal list containing fadeSprites, it will never die and will always deal with fading any object in its list.

Every sprite object (an object that is drawn) must do the do\_fader function, which adds the current image and rect to the master list. The image and rect are part of the sprite class – fadeSprites.

Do we need the fadesprite class anymore?

1. Create bullet
2. In the bullet’s update function, do do\_fader()
3. Do\_fader() creates a copy of the sprite and adds it to the faderManager’s list
4. Each frame the fadermanager fades each of its list’s components, deleting if necessary

This seems very intensive. Hope it’s ok

6/9/17

The fading stuff is too intense – need to make it simpler.

The faderManager will have a list of, say 10 lists. Each list contains a set of fadeObjects that were made in an arbitrary frame. Every frame, the alpha of each object in the biglist is reduced by 10 or so. The freshest batch will be set to 240, the next 230, etc. This saves us calculating the alpha for each thing individually.

We can make some calculation to get the fade decrease factor, and the size of the biglist. For now, the biglist will have 5 sub-lists, and the difference in alpha will be 50, starting at 250 going to 0.

Option 1: At index 0, input all fadeObjects into biglist[index]. Increment index. Next frame, put everything into biglist[index]. For everything in biglist[index-1], set alpha to maxAlpha-50. Go back 1, decrement alpha by maxAlpha-100. Do that for the length of the biglist.

Option 2: set biglist[lsb] to biglist[lsb-1], biglist[lsb1-1] = biglist[lsb-2], etc. Put all new fadeObjects into biglist[0], for every object in other sub-lists decrement alpha by index\*50. Essentially, shift the lists over, just like a shift register. Not very happy about copying a whole list over multiple times a frame though, but this method is simple to get your head around.

Unfortunately, option 1 did not work so well. It works, but the slowdown is still significant. There are probably just too many objects to handle. How intense is the set\_alpha function?

Is it possible it’s because all the objects are sprites? They take up more memory, but do they hinder the framerate? I might have to rearrange things again to allow multiprocessing…

Try to do multiprocessing. For each of your sublists, make a new process.

8/9/17

I need to split up each of the subLists into a process. Each process loops for ever, constantly doing stuff.

OR:

Each time a ‘fade’ needs to happen, send it off to one of the new processors until it’s done. Don’t try splitting stuff up into groups, just do one thing at a time. Sounds like it could get dangerous, dealing with graphics asynchronously…

Every frame, *just after* a draw has happened, start sending the fadelist to the processor. That’ll make a queue, one for each item in the fadelist.

In English: put each fadeObject into a queue, so that the array of multiprocess things can take a fadeobject and do the fading. Hopefully, the whole lot will complete before the next frame. If it doesn’t, the method needs to be scrapped.

The problem is that we need to pass the biglist to all the new processes, and return the new alpha values. We can’t just make the function do drawing – it needs to modify the variables, so we might as well keep the drawing to the drawing section.

If the function takes the fadeObject as an input, it can return a fadeObject with the modified information. In this case, every frame we’re essentially remaking the drawList, rather than modifying it.

Also, maybe the slowdown is due to the trying to draw so many things onto the screen, as opposed to the calculations. Maybe the new processes should be drawing, instead of calculating…

26/09/17

I got rid of faderstuff today. To replace it, instead of there being a screen.fil(black) then drawing everything, the screen is set to a very transparent black. All the objects on the screen appear to be fading like. Hooray!

Unfortunately, there still appears to be a lot of slowdown when there are about 100 bad guys. Especially when the game tries to make them, a spike defo appears. This might be due to the movement and all that as opposed to drawing now.

7/9/17

I want to make the game ‘zoom’, depending on the level. This will make all sprites bigger/smaller, and change the number of nodes available to move between.

I could be smart about it and try to determine the best size/node space ratio. I might have to re-think the way the game determines ‘resoChange’.

I’m going to have to overhaul the whole resoChange stuff. It is no longer needed, I’ve figured out how to modify the screen size by dragging- this means the ***new resolution will have to be recalculated based on the new width and height of the screen***. If we know the ratio we want (15:9) we can find the ‘floor’ of the current screen size, and add blank areas at the sides of the screen.

There may be another solution; keep the current resolution behaviour, but have something else that changes the screen size. You can have low resolution and big screen, high res and small screen, anything. Deal with them separately. To do this, blit everything at the current reso to a surface, then stretch the surface to the screen size defined by the VIDEORESIZE we looked at. I guess this is kinda like how gamemaker deals with it, with its screens and ports.

If this is how it’ll work, screen height and width need to be completely separate from resoChange.

9/10/17

I had to mess around with what goes in the definitions module. The module should now only contain static variables that will not change, that all other modules will probably need to know, like colours.

However, there are some things that will change, like screen height and width, and the new variable objectSize – this variable tells the game, when it makes a level, how big all the objects should be in relation to the resolution. Even if at max reso, the objects can appear big (for a small level), or small (for a big level). During a level, the area can change size – when a boss appears. This variable must be at the top level, passed down to pretty much everything. That blows.

I think if definitions imports mainscript, and mainscript hold manipulatable variables, it’ll be ok. Brb

Ok as it turns out you can modify stuff just fine. Hooray!

HOWEVER

There is a new problem. We draw stuff to the absolute position on the screen. We need to draw it to the absolute position on the scale of the resolution, then stretch to the screen dimensions.

So:

1. Define the resolution. Let’s say 320x240, for now.
2. Objects on the screen will be placed and will move according to this.
3. Every frame, stuff is drawn to this surface. It is stretched up to the window\_size, and blitted to the screen.

But there’s a problem with this – the size and speed of the objects must be relative to the resolution. EG: at the lowest resolution, the nodes might only be 10 pixels apart, but higher resos have nodes 30 or so apart. So there needs to be a base resolution, which is multiplied up to meet the desired resolution. This multiplier also affects the speeds and sizes of the objects. Isn’t that what we already have?

We’ll need a multiplier for the x and y directions individually. If I make the screen taller, the objects should only be stretched vertically.

11/9/17

Really did work on the resolution and screen size stuff. ATM, there are 2 important surfaces: the screen, and the ‘main\_suurface’. The screen is the actual displayed thing, as in it uses the WIDTH and HEIGHT variables. Main\_surface uses RESOWIDTH and RESOHEIGHT. Every frame, the main\_surface is scaled either up or down to match the screen size. If the resolution is lower (RESO<WIDTH), there is stretching, otherwise it looks much crisper, but ofc is more intensive.

When actually positioning in the level, the game does so to the main\_surface. Therefore, all **sizes and movements must be scaled by the resolution multiplier/factor.** This should also include speeds, as it’s covering a larger area in the same amount of time.

It really doesn’t like scaling down. You need to implement some system for relating screen size to a resolution. We might have to try to looking into OpenCL, or solutions that don’t include scaling…