Getting Started on GOMC4

1.0 INTRODUCTION

Dev2.py looks a bit scary at first but that's largely due to me throwing things around and learning how to use PyGame. I'll go over some basics and walk through what's happening step by step. There's quite a bit to work on so whatever you want to play with just let me know. This is to have fun and learn so if you're not really enjoying it we can switch to something else. If you don't recall the basic Python syntax, or what an if-else statemet or while loop, etc is I would do a quick refresher using an online tutorial just to get the basic concepts. You should know what variables, functions, and flow control (while, for, if, etc) are. Let me know if you have any questions at any point.

2.0 PYTHON

Best place to start would be getting you all of the necessary files and build tools. This is where Python is really nice as a language since it doesn't really have build tools, just the run time environment. You can download Python and launch it (I'm currently running Python 2.7, select windows...msi). It'll provide you with a command prompt where you can either run python commands or call python scripts (like dev2.py). In addition you'll need PyGame which is the external library I'm using to handle all of the graphics. You can get it here:

http://www.pygame.org/download.shtml

Whatever I have pushed to the master branch on GitHub should always be able to run, so if you pull it, open up a windows command prompt (search 'cmd' in your windows search bar) and navigate to the download directory (you can also shift click in the directory and select open command window here if you don't know how to navigate directories from the command line) you can run the command "python dev2.py" to execute it.

3.0 GIT AND VERSION CONTROL

To actually get the dev files from Github, you have three options: (1) GitBash command line tool, (2) TortoiseGit windows plugin, (3) GitGUI. Git bash (1) is what I use but may be a bit ugly to pick up. I think (2) would be easiest but I'm also not a huge fan of the GUI. The second one will let you right click on a directory and have the Git tools show up in the windows right click menu. You can initialize directories, pull, push, merge, etc from there. I'd recommend going over the GitHub tutorials really quickly just to get the concepts in place and learn how to use the toolset really quickly. I'll talk a bit about Git bash commands below because I'm more familiar with those and they should translate pretty closely to what commands you'll use in whatever option you use above.

3.1 Git Concepts

Git is an example of a version control tool. Think of it as a dropbox or google drive where files are stored online on a server, but here the history is preserved. People can pull files off of this online storage to their local machines and modify them. If they want to push their changes back to the online server for everyone to see and copy over to their own local instances, they can do that easily. If a mistake was made, you can revert back to previous states. Additionally you have things called branches. The Master branch is the default, top level branch. This is where you start off. You "branch" off of the master line, which means you take a copy of the drive and make a new one alongside it in a sense. Whatever changes you make and push online to this separate branch (you can have multiple) will only be reflected in this side branch, not in Master. An example would be having a Master and a Dev (development) branch. Master will always be buildable and stable, whereas you can push and test new features to the Dev branch. Once the changes you made are stable and tested, you can merge the branches. This updates the Master branch with the new contents from the other line (Dev in this case). It can be a bit of a weird concept to get a grasp of a first but it's very powerful and helpful.

I will make a Dev branch specifically for you so you can change whatever you want and push things to the new branch without causing damage so don't by shy with updating. Let's say there's a branch called 'ave-dev'. If I were using Git Bash (again the other tools will have similar names, you just don't need to type them out), I would call these commands to get started.

+ Navigate to new (empty) directory
+ 'git init'
+ 'git remote add origin master https://github/kgilbert1993/GOMC4.git'
    // or whatever the link is for the repo you want
+ 'git pull origin [BRANCH NAME]'
    - For example: git pull origin master
    - or: git pull origin ave-dev

And that's it, you have the files! Origin refers to the "remote", which is essentially a reference to where the online repository is stored, so if you had a different remote (unlikely if you're sticking with the basics), you could do 'git pull otherremote master', etc.

Now let's say you edited some files on your local computer, and it's looking good! You can update the online drive so I can see it with the following command format:

+ 'git add [MODIFIED FILES\FOLDERS'
    - for example say you modified dev2.py and added a file new.py. The command would be 'git add dev2.py new.py'
    - You can run git add multiple times, it'll just add the new files to the pending change list.
    - You can also call 'git rm *blah*' to remove files
+ 'git commit'
    - This commits your changes. Your online changes will not yet be

reflected online, but you need to do this to compile your changelist. A window will pop up asking you for a commit message. Whenever you update the repo you need to include a message describing what you changed. These can vary in detail from 'added file or did some stuff' to a paragraph describing things. You can view the changes so you don't need to be super specific unless it was a conceptual change that's hard to get from just viewing the code.

+ 'git push origin [BRANCH NAME]'
    - Same as pulling. 'git push origin ave-dev'

And that's it! The basic operations then are:

+ init (only need to do this once!)
+ pulling [get files from remote server to local build]
+ adding [add new OR modified files to changelist]
+ committing [Generate commit message and prepare to push]
+ push [Update online repo with local changes]

Those are the basics of using Git! Again, I'd go over online tutorials if you get stuck or let me know if you have any questions. Now that you have the files we can actually get to the meat!

4.0 GAME STRUCTURE AND SOFTWARE DESIGN

Now that you have the basic files and toolsets needed to get started we can get to the interesting part. The two software layers that we will be primarily developing in consist of Python, and external libraries. Python will act as our code engine and language. However the builtin libraries do not have much visual (Graphical User Interface or GUI) support, so we need to grab other libraries written in Python (and underlying languages) that are designed to provide an API (Application Program Interface) to generate visual effects like windows, buttons, etc. Think of the API as a set of additional functions that can be imported and called from our Python scripts. PyGame is the primary library I have been using as it’s pretty simple and allows for hardware acceleration and other performance enhancing properties.

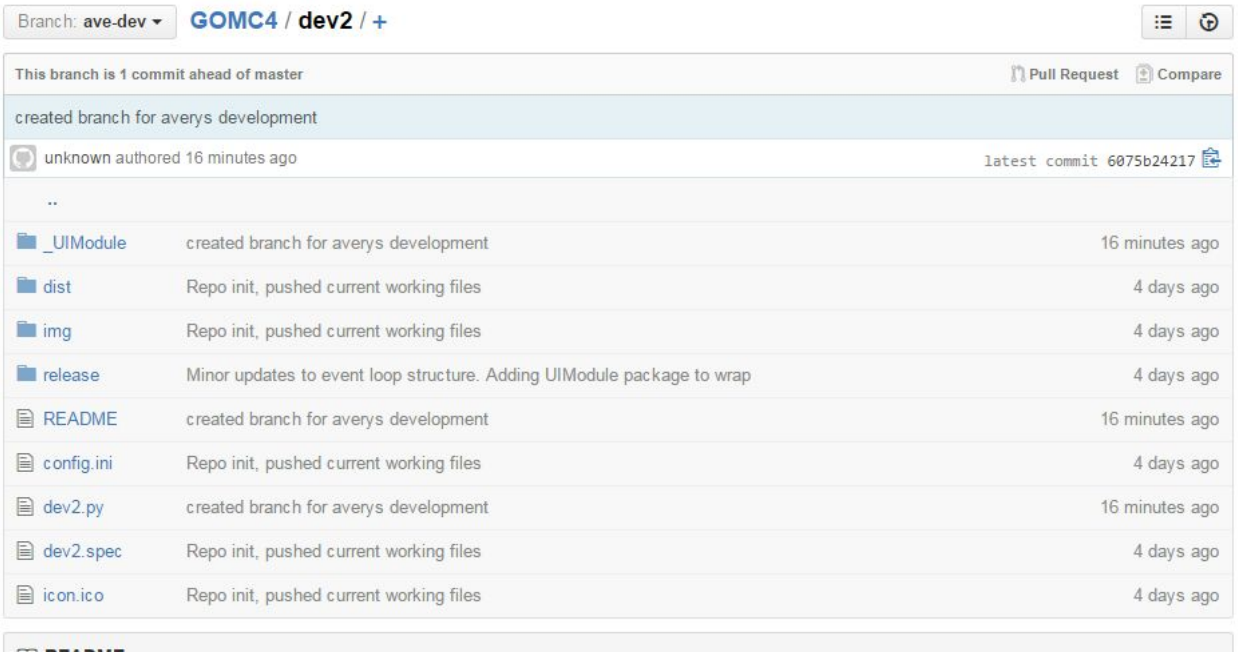
The biggest pro to using Python was that it was very easy to pick up the API and begin using it, and had little setup time. The biggest downside is that Python is not very performance orientated, meaning it runs slower and needs more resources than if I had written this in say C++. I’ll talk more about language considerations and whether or not I will be switching later but for now this setup has been working very well and allows for easy export of executable binaries.

Before I jump into the code I’ll review the basic software data flow design being used. As common in game development, or anything relying heavily on user input, I am using something called an event loop model. What this means is that the primary core of the system will run in a loop catching user generated events. Events in this case would be mouse motion, clicking buttons or keyboard keys, etc. Each time the user interacts with the computer an event is generated and placed in a queue. Each loop iteration we grab the events that have been

generated since the last iteration (roughly at 60Hz is what I have it set at now, so every ~16.6ms) and handle them. For example we saw that the mouse was moved. We can check where the mouse moved to, and update the screen if it falls within a region that we think should respond (like mousing over a button makes it change color or being up text). That's the basic concept, and will be fleshed out further in section 4.2 as I discuss the main loop.

4.1 File Structure Overview

The image below is a screencap of what the current file structure looks like at the time that I am writing this. The dist and release directories won't matter much to you so don't worry about those. The img and _UIModule directories are more important. Img contains as it sounds like images being used such as button layouts or background images. _UIModule contains a submodule I'm writing that will contain an API for instantiating instances of buttons and other user interface devices. You can see me importing the classes and functions defined in that module in line 14 [1] within dev2.py. Speaking of, dev2.py is the 'main' script, or file that you run to execute the whole function. Ideally this will be broken down a bit into subfiles to make it more manageable, but for now everything is being tossed into it as I play with new features.



4.2 Main Program

Let's look at what our main program is. Line 88 within dev2.py is where execution starts. When you run the script within Python some system flags are set. If the file was run as a top level file (not called by another running python script) the __name__ flag is set as __main__. This if-statement checks for that flag and runs the code beneath it if it is true. The line beneath it then is where our code starts executing once run. Everything above that are global variables and functions. They are values and methods used from other parts of the code and allow for reusability of common functions, make the code look more clean, and let me pass data between different parts of the code.

Following here I'm going to do a line by line explanation of the basics.

pygame.init() # Initializes the pygame library.

- SideNote: In Python (and other object oriented languages like C++ and Java), the dot operator denotes that you are accessing something within an object's namespace. At a high level, think of pygame as a large package or bundle. Init is a specific item with that bundle. If I just called 'init()', Python would think I was trying to call a user defined function which doesn't exist (unless you import that function specifically from a module). The syntax for calling pygame functions and classes is: 'pygame.CLASS/FUNCTION.subfunction…..' etc. You can see a more clear example of this at line 92/93. As I pointed out earlier I imported my own submodule within the package _UIModule under the namespace 'Buttons'. You can view this module under Buttons.py within the _UIModule directory. Test is a function name within that directory, and Button is a class. A class is simple a collection of variables and methods. An object is simple an instantiation of a class. In other words, a class is like a template (not to be confused with templates, which are another construct with many object oriented languages). You typically need to have created a variable of that class's TYPE. This is more explicit in C++ or Java, but in Python since types are implicit it's a bit less apparent. You may have static methods within a class that allow you to call them without creating an object first. This may sound really wonky at first and took me some time to get used to, but the more you play with it the easier it becomes. If the current project is too messy for you to get the concepts, start with a simplified version and play with some online tutorials. The example of this within our project is the next line below Buttons.test(), where I create an object from class Button. I can now call functions and access class variables via x.var, or x.function(). I'm covering this to try to give you an abstract idea of how classes and object work as they will be how I will be implementing the user interface mainly.

screen = pygame.display.set_mode(size,HWSURFACE|DOUBLEBUF|RESIZABLE) # Create a screen object on which we can play all of our visual components. set_mode is a function within pygame.display which takes in a few parameters. I don't know how much Python you remember, but everything between the '(...)' are the parameters or inputs you are giving to that function. Here I pass it a variable defined in dev2.py, size, to tell set_mode what the resolution of the screen will be. The other inputs are configuring the screen to use hardware acceleration to increase performance, double buffer (which means update screen in a software buffer before writing to the physical screen, also a performance booster as writing to the screen is much slower than writing to memory), and allow the window to be resizeable. All references to this screen object will be done through the we just created: *screen*

Buttons.test()
x = Buttons.Button(screen, 80, 50, 400, 300, BUTTON_IMG) # Talked about these already

bg = pygame.image.load(filename)
bg = pygame.transform.scale(bg, (display_w, display_h)) # These two lines load an image (filename is another variable defined near the top of the file) from a provided file path and
scale it to fit within the screen dimensions.

exit_r = pygame.image.load(EXIT_RED).convert(24)
exit_g = pygame.image.load(EXIT_GR).convert(24)
exit_r = pygame.transform.scale(exit_r, (quit_w, quit_h))
exit_g = pygame.transform.scale(exit_g, (quit_w, quit_h))
exit_r.set_alpha(128)
exit_g.set_alpha(128) # This block creates the exit button. My UI module is not currently complete, so button interfaces are done all manually right now.
What this does is load two image files, a red and green exit button image into two different
variables and like the background image above scales them to pre-defined constants (again these are global variables defined at the top of dev2.py). The last two lines set alpha. If you've heard of RGB color schemes (Red Green Blue) where you select colors using 8-bit values (0-255) for red, green, and blue values, imagine alpha as the fourth component that controls transparency. This lets me make the buttons have various levels of transparency to look cool and stuff.

screen.blit(bg, (0,0)) # This function copies over a Surface (in this case our background image we created earlier) starting at location (0,0) (uses cartesian plot, 0,0 is top left corner of screen, first value is x and second is y). Remember the double buffering setting I mentioned earlier? This is what it does. When I 'blit' the image, it doesn't draw it to the screen (remember, this is a slow process and ideally should only be done once every loop,
not for every object we want to update). It copies the image to a memory buffer, think of it
as a screen that exists in software. Once we've updated everything in memory, we will call
pygame.display.update() or pygame.display.flip() [see line 229].

4.2.1 *Event Loop*

The event loop begins at line 188, while the game's runtime environment starts at line 123. Everything between the main start and line 123 only happens once, this is the initialization step.

Everything after 123 (in the while True) block happens in a loop. The event loop (line 188) is what I was referring to way back about each user input action generating an event. You can see at the start of the runtime loop I call *events = pygame.event.get()*. This grabs the queue of events, and I go through each event in the event loop which is just a large if else if else...statement.

5.0 DESIGN ROADMAP

As I've mentioned the code base is really messy right now. As I create better library support for user interface controls and move things out into subfiles it'll begin to look a lot nicer. The features I would to begin to add in the near future are:

1. Configuration File Control
2. User Interface Package
3. Character Unit List and Unit Specs
    a. A reconfigurable template that would make tweaking unit constants easy
4. Welcome screen/menu
    a. Add username/login (ties in with config file?)
    b. Requires (2), button support
5. Multiplayer support
    a. Python plugin to allow connecting to another running session of the game Looking into sample project, UDP or TCP modules.

Let me know what seems interesting to you. The config file would be pretty straight forward, it would just let users select options like cursor display, default screen size, etc and store them inside of a config.ini file (already in the repo) so that when they relaunch the game I can pull constants from there. I can show you what I had in mind for that. I think the third option would be most straightforward for you and require less complex software development stuff. Essentially the idea I had was either create an Excel document that stores each unit's traits and load that into Python so we can easily update constants, or do it within Python itself where we can create an object for each unit and feed in constants at runtime.

6.0 CONCLUSION

Please let me know if you want more elaboration on anything! I just sort of threw this together as a crash course and likely has a lot of gaps. I created a branch for you at https://github.com/KGilbert1993/GOMC4/tree/ave-dev/dev2 to play with (refer to Git section on branches).

Good luck and have fun!

7.0 NOTES

[1] Line numbers subject to change when files are updated