Developer’s Guide: Victory Ratio

Overview:

Victory Ratio is a competitive medieval strategy game written in python using the pygame framework. Each player has 27 units. Nine of each of three different classes; axe, sword, and spear. Each class get’s advantage against one other class, so that each has a strength and a weakness. Units can be placed in groups of up to nine, and may each move and/or attack once per round. Player’s take turns moving units until all units have moved. For combat, the attacking unit gets to deal damage first, the weakened defending unit then strikes back. The game ends when one player has no units remaining. Game is played at a single computer and only requires the use of the mouse.

Installation:

Assuming knowledge of how to install python and an interpreter, all that is needed to add to run this game is pygame and numpy. This can be installed from the command line in your development environment by calling “pip install pygame” and “pip install numpy”. In the case of the developer using Anaconda as their interpreter, it is important to use the “pip” command instead of “conda” when installing numpy, as there is a known issue with Anaconda’s interaction with numpy which may cause issues at compile time.

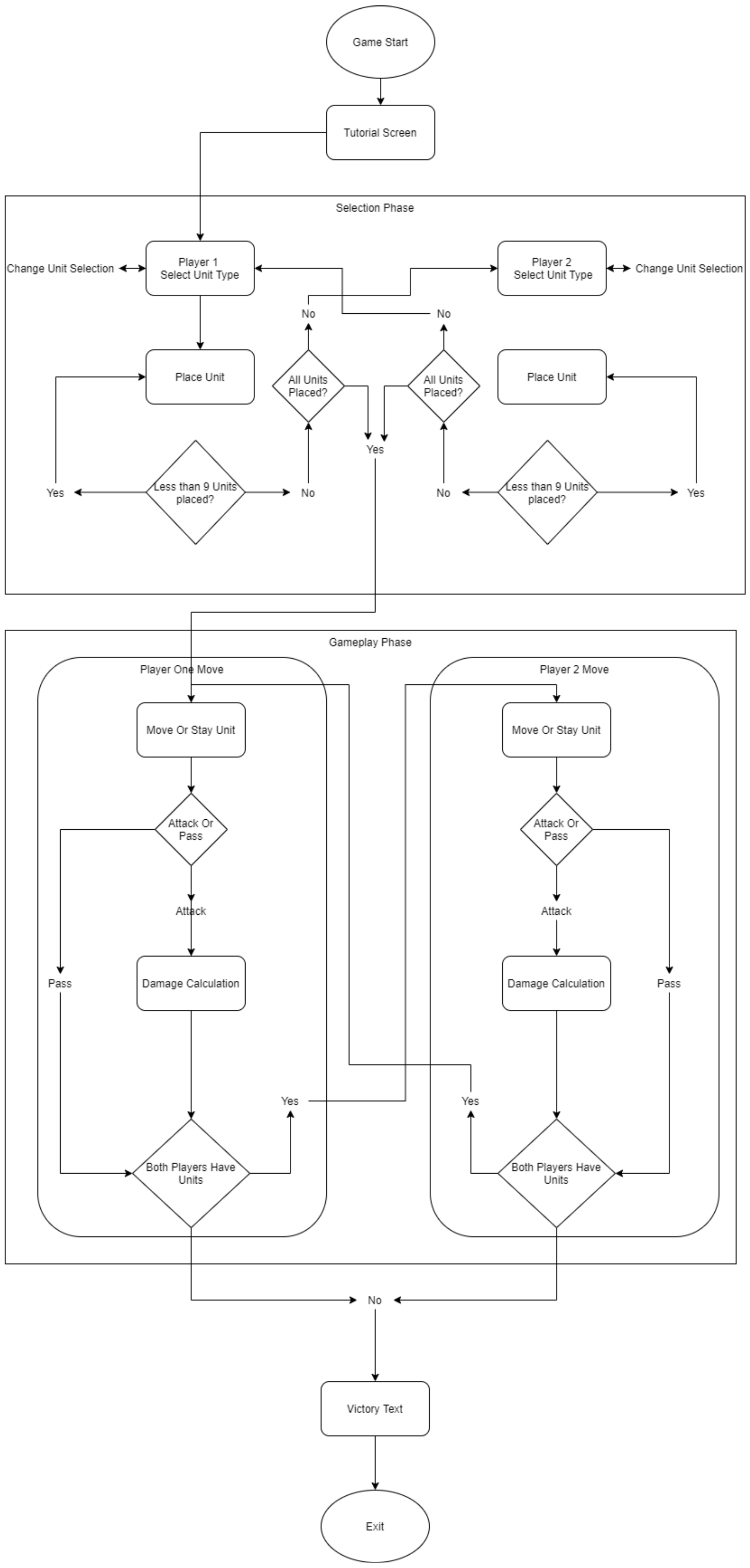
File Structure:

Code Flow

A brief synopsis of how the code operates. “main.py” contains the “while” loop that will keep the game running. It starts off before that creating the map and GUI elements. The GUI elements in “gameplay_UI.py” are used to control the state of the game, such as the tutorial state, unit selection phase, combat phase, etc;. While this is being called, the “board.py” file contains the Map and Pool classes that control how the gameplay elements are rendered. “tiles.py” controls the individual gameplay elements managed by the Map class, while Unit_Selection does the same for the elements managed by Pool.

main.py

The file that the game runs from. Handles calling methods and instantiating classes needed to display the game. Also handles transitioning between phases of gameplay as indicated by this chart.


Game Start
Tutorial Screen
Selection Phase
Change Unit Selection
Player 1 Select Unit Type
Player 2 Select Unit Type
Change Unit Selection
No
No
Place Unit
All Units Placed?
All Units Placed?
Place Unit
Yes
Yes
Less than 9 Units placed?
No
No
Less than 9 Units placed?
Yes
Gameplay Phase
Player One Move
Player 2 Move
Move Or Stay Unit
Move Or Stay Unit
Attack Or Pass
Attack Or Pass
Attack
Attack
Pass
Damage Calculation
Damage Calculation
Pass
Yes
Yes
Both Players Have Units
Both Players Have Units
No
Victory Text
Exit

Which game controls are displayed are also controlled by the state of the game, such as the unit pool being hidden when the game begins.
Various defs stored in main.py are used to keep track of what happens when the user clicks the screen. Validity of the clicks are determined based on the phase of the game in conjunction with collision detection of the Tile class from tile.py. Animation is also handled here under animate_combat.
The “while” loop towards the bottom will continue to run until the user clicks the ‘x’ in the top right side of the window. The last line run is the clock, which is locked to 60 ticks. What this means is the game runs at 60 frames per second. If the computer running it cannot run that fast, the game will appear to run slower, but with the framerate locked a more powerful computer will not run things at crazy speeds.

gameplay_UI.py

Contains classes for a few gameplay elements not handled by tile.py. The file contains Pass_Button, Instruction, and Tutorial as separate classes.
Pass_Button can detect if it is clicked, and renders the image of the button itself. If there is a desire to reposition it, this can easily be done by modifying the blit under the def show_button, however as written currently you would have to change the check_collision def accordingly.
Instruction can be instantiated and told to display different text at the bottom of the screen. It is intended to be used to let the player know whose turn it is.
Tutorial displays an instructive banner that shows on game launch. It requires no sort of detection, and so goes away based on code in main.py.

board.py

This file controls the data structures responsible for presenting and controlling game elements. Map, Pool, and Unit_Selection are all classes contained within board.py
Map handles visual representation of the matrix of tiles that are displayed to the user. It has methods that loop through it to adjust traits attributed to each tile, such as units contained, if that tile is valid to place units or to attack, and what to display. Changing the map’s row and column count here should carry throughout the code, if the developer would like a larger or smaller map.
Pool contains a list of Unit_Selections that represent placeable units for the placement phase of the game. Code handles tracking which unit has been selected as well as deselecting units at the change of phase. A developer who wishes to change the total number of units available to a player or later add units to the game can do so here.
Unit_Selection handles the individual unit types of the pool. Handles rendering and confirming selection of the unit, as well as tracking the count in the pool.

tile.py

This file contains the Tile class and it’s subclasses Forest, Plain, and Water.

The Tile class handles functionality of individual tiles on the map. Stores the location on the screen as well as the index in the map matrix. Stores rendering information for the tile as well as the groups of units placed on the tile. Contains dictionaries handling units placed on the tile, stats of the tile, the strengths and weaknesses of the different unit types, and whether or not a tile is valid for movement or attack. Definitions within handle adding or subtracting units, detecting clicks, displaying units, combat calculations between a unit on this tile and a unit on an selected tile, and animation of the combat.
The Forest, Plain, and Water subclasses can call any method of the Tile class, but initiate with the defensive traits and images appropriate to the tile, and can be changed here. Adding a new tile type involves adding a new subclass in the same manner of these with the desired stats.

Known Issues:

Turn Order

Sometimes when there is an uneven number of unit groups between player 1 and 2, the turn order can pass incorrectly. Would recommend looking into swap_turn and new_round in main.py for a solution.

Adding Unit to Wrong Group

If the user changes units they are placing, such as from axe to spear, then tries to place the spear holder in a place with an axe unit, the spear count will decrease and additional axe units will be placed instead. The user should be blocked from placing a unit there.

User Experience Improvements

Can be made more intuitive by doing things such as putting the pass button on player 2's side of the field during his turn. This should easily be changed under Pass_Button in gameplay_UI.py. Could also change from telling the user it is player 1 or player 2's turn to Blue or Red's turn.

Future Development:

At inception, this was to be an educational game. As an educational resource, it was intended to help develop math skills by giving the user information about how the game calculates damage so that the user might apply fractions and ratio math to determine whether a move is good or not. So presentation of exactly how combat damage is calculated would be key. Unfortunately Victory Ratio would be difficult to make available, as pygame cannot be deployed to the web, and is difficult to deploy as an executable. It would be recommended that this project be redone in Kivy in order to make it deployable. Considering how much of the code is unrelated to pygame, this should not be the most costly conversion.

Implementing A* pathfinding in the game could be key to improving the code to find valid movement of units as well as one day implementing AI in the game.
Other quality of life improvements would include sound and additional animations, such as movement and perhaps a pop up screen of a more detailed battle animation. New units can be added to the game very easily, so it could be enjoyable to add an archer that can attack from far away or a horsemen that can move faster.