CS 386 — Game Dev — Simple Classic VG in Pygame

Project #2: Simple Classic Video Game in Pygame [after Prof. Shafae's version]
This project is to write a **Python + Pygame** program to build a classic game: that is, a video version of a simple well-known classic board game, or pencil-paper game, or card game, etc. (samples below).

Introduction

Game programming can be very difficult when the programmer is not familiar with the game. There is never anything simple about programming a game, however being intimately familiar with a classic (or traditional) game can aid in mapping the mechanics of the game into software. Using a classic game also has the advantage that the need for complex graphical code is lessened.

The objective of this assignment is to practice programming a game that the programmer understands very well. A further objective is to scale down the classic games to manageable sizes: eg, 6x6 chess/checkers board.

Well-Known Games

- Dice Chess
- Diagonal Checkers
- Chinese Checkers
- Concentration
- Mancala
- Yahtzee
- Pachisi/Parcheesi
- Go (11x11 board?)
- Snakes and Ladders
- Backgammon
- Othello/Reversi
- Blue and Gray
- Lines of Action
- Nine Men's Morris
- Lotto
- Scrabble
- Twixt
- Havannah (base-8?)
- Dots and Boxes
- Abalone
- Mastermind
- Labyrinth
- Terrace (6x6?)
- Quarto
- Continuo
- Quoridor
- Boku

If you want to use a game not on this list, please get the instructor's permission.
Note that word-related games typically require a dictionary of words, and subject/concept related games typically require a bit more than an encyclopedia.

Multi-Player

Your game should provide for two players to play, taking turns (a turn-based game). Please be sure to make it obvious (eg, on screen) who's turn it is.

Level-0 Brain Opponent

Your game should provide for one player to play a "computer brain". You only need a "level-0" brain; one that will only make a random, but legal, move/choice. It makes sense to design the Brain as a standalone cluster of functions (perhaps a class object) that cannot directly make actual changes to the Model's game state, but instead creates the equivalent of a player's input command/choice which is then passed as an argument in a method/function call to the Model, just like the Brain was another outside-the-box player; however, this is not required.

Based on the game rules, you will have to have the Brain determine what a legal move is, somehow. Often the simplest way is to ask the game Model if a given fully-random move is legal in the current game state, and keep asking until a move is adjudged legal by the Model -- and then make the move (via a Model call). At today's computer speeds, this is likely sufficient. If it turns out not to be fast enough, you could add functionality to the Model to answer more complex questions from the Brain.

To make it easier for the user to see what is going on, you will likely want to include a pause or delay after the user makes his/her move before the Brain is given control to make a move (because the Brain runs at CPU speeds and the user may otherwise not clearly see what happened to the game state.

Teams

You can have up to 3 members on a project team. You should consider using Agile "Stand-ups" to ensure progress. Less than 5 minutes time, progress board, and the 3 questions answered (sometimes embarrasing).

Game Design

You may wish to create a design document to help you understand the requirements of the game you selected. The game design document might be patterned after the following suggestion of topics:

o **Introduction:** What is your project? Describe in detail what your project is so a layperson can understand what the objective is as well as the utility. For instance, explain what kind of game it is, how it is played, who is the intended audience and any action or narrative that accompanies the game.

o **Design:** Detail the rules of the game, how it is played, example scenarios. Provide enough detail such that a savvy person can create a software architecture from this information. Factors to consider while writing your game design are:

– Rules
– Sources of uncertainty
– Win state, lose state
– Expected skills the player must have prior to starting the game
– Controls
– Expected duration of a game
– Scoring
– Visual representation of game state

o **Software Architecture**: If desired, detail the software design you used for your program. Provide details such as algorithms used, software model, (simplified, or hand-drawn) UML models, etc.

o **Bibliography:** A list of references you used to understand and accomplish this project. For instance, an online explanation of the rules, a strategy guide, a book about the game, etc.

o **Development Time**: Waiting till the deadline to create working code rarely works. Also, writing a batch of more than 10 lines of executable code before testing the batch rarely works well (Rule #2). The fastest way to code is two-fold:

1. Keep the code for different concepts in different buckets (the “Single Responsibility” part of “SOLID” S/W engineering development), and
2. Drastically reduce the amount of time spent tracking down run-time bugs by keeping your next testable batch of code well under 10 lines of executable code (so that the run-time bugs are easy to find).

Weighting of Project Issues

- 60% for a playable game
- 20% for a clearly defined win and lose state
- 10% for online instructions or tutoring
- 10% for a legal and random Brain opponent
- 5% for a reasonable README.txt file
- 5% for following the Submission rules

Prerequisites

In order to complete the exercise successfully, at a minimum, you will need to have the following:

- **Python** interpreter
- A text editor
- **Pygame** module

Students are encouraged to use the **Pygame** module for graphics rendering and mouse/joystick input.

Docs

Although no documentation is required, it is in the best interest of the student to document the design in writing before beginning to code and test each portion of the game. Typically, the designer would start with the initial Setup and then with the major user interactions, called "Use Cases", which is to say: use an outside-in approach. So, you would want to document what the user will be allowed to request from your (game) system (e.g., making moves, turning cards, etc.)

Internally, a good S/W design will have a 2-level architecture: a top-level using names and concepts from the user-game world, and a lower "CS" support level using computer science (or simple programming) names and concepts. For example, the CS level Model would include both the current state of the game and functions to update that state. Additionally, two other clusters of CS-level functions (or methods) would focus on displaying (outputting) portions of that game model to the user and in obtaining requests (inputs) from the user. These I/O clusters are called a View (for the output) and a Controller for the input. The Controller cluster will pass along well-formed requests (function/method calls) to the Model. The Model will modify its state based on calls from the Controller and will then call the View to ensure that the user is apprised of the state changes. (This example uses the industry-standard MVC architecture.)

Academic Rules

Correctly and properly attribute all third party material and references, lest points be taken off.

Submission Rules

Your submission must, at a minimum, include a plain ASCII text file called **`README.txt`** (e.g., title, contact info, files list, installation/run info, bugs remaining, features added) all necessary source files to allow the submission to be built and run independently by the instructor. [For this project, no unusual files are expected.] Note, the instructor doesn't necessarily use your IDE.

All source code files must include a comment header identifying the author, author's contact info (please, no phone numbers), and a brief description of the file.

Do not include any IDE-specific files, object files, binary executables, or other superfluous files.

If you need to include a text file, use either a .txt file, or a .pdf file if formatting is required. Please don't include a .doc or .docx file.

Place your submission files in a **folder named with this format:**

`X-pY_teamname` or `X-pY_lastname-firstinitial`

where **`X`** is the course **number** (e.g., **123** for CS-123) and **`Y`** is or the project **number** (eg, 9 for Project #9). For example in CS-**123** for Project #**9**, if you were a 1-person team and named were **Tim Crazy**, then you would use

`123-p9_Crazy-T`

for the folder name, or if your team members were **D**ewey, **C**heatam and **H**owe, then you would use

`123-p9_DCH`

for the folder name. If you only have a 2-person team, use "X" for the third team letter.

Then zip up this folder. **Name the .zip file the same as the folder name**. Please don't use a .rar extension.

`123-p9_Cruise-T.zip`

Note that if you have "file extensions" hidden on your files browser, you will not see the **.zip** file extension.

The project is due by the time and on the due date specified in the class's bulletin-board project assignment post. Turn in by **sending me email** (see the Syllabus for the correct email address) with the zip file attached.

The email subject title should also include the folder name.

NB, If your emailer will not email a .zip file (eg, because of file size), then change the file extension from .zip to .zap, attach that, and tell me so in the email.

Please include your name and campus ID at the end of the email (because some email addresses don't make this clear) – and if it's a team effort then include everybody on the team.. If there is a problem with your project, don't put it in the email body – instead tell me by putting it in the README.txt file.