EPTA Code Execution Flowchart

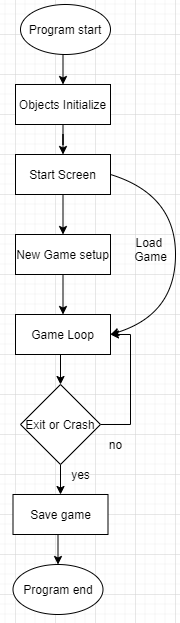
Brendan Fallon, July 4, 2020

Written to describe how the code makes the game work, explains why some of the design decisions were made that way, and also improvements that could be made to simplify things.

This document is a great place to start with how the overall program works.

This also outlines the software architecture and how it could be refactored to the NEVADA game engine.

# Game Core Code Path



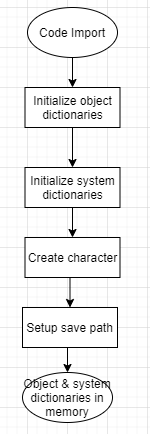
This all starts in the EngPhysAdventure Alpha v0.30.py (or whatever the version is). That’s the main file where everything is run from. All the game variables are initialized at the start with the import statements. Then at the end of the code after the function declarations the game execution begins.

We start with an opening screen (which can be skipped) then loads all the game attributes into memory if you’re doing a loading sequence. If not, the game will be initialized in the setup function where you start a new character (which can also be skipped for loading in or dev mode). Then it runs the core main game loop which is where the rest of the game runs out. It should be noted that there’s two versions running depending on if you’re in developer mode (which shows you errors so you can debug) or just in normal mode where it will catch errors and save data before crashing.

Everything should be under the main game loop in the future including the start screen and setup (maybe read some books on this). Start screen, setup/ending should just be another event function not in main.

It was designed this way because the game was initially designed linearly with a setup, main game loop, and ending. But I had to change it around so that you could restart the game, load in, or do multiple endings, so isn’t perfect because of the modifications to the original code (and the unwillingness to start from scratch on the main game functions and just modify incrementally to keep a usable product).

## Objects Initialize

The start of the program loads in all the entirety of the game elements into memory. How these are initialized, stored, and manipulated through the game is somewhat disorganized and there’s lots of room for optimization.

The import statements are what make the game initialize the variables at the start. In EngPhysAdventure Alpha v0.30 (or whatever version) the “from GameFunctions import \*” runs the entirety of the GameFunctions.py file which initializes all the game variables, lists, objects, etc. Everything before the first function declaration is the first code executed to initialize the world and game elements. The first and most important part is game object initialization by calling the Startup file to load in the main game object dictionaries. The StartUp.py file contains all the objects initial values as well as the constructors to make them into a dictionary.

Dictionaries are used to store a lot of the game objects and system information in key value pairs so that it’s easier to read and write game code about specific objects. The design decision negates having to store everything in a list and lookup array codes but at the cost of still having to lookup key names, having to follow a specific naming convention (lowercase of the object name), and not being able to handle duplicate keys very well.

This game setup (before function declaration) should be put in an initialization or the main file.

Many of the game functions should be methods within the object class and pass elements by reference. Game functions.py should simply call those methods/pass information correctly and store non-object related functions.

### Initialize Object Dictionaries

The game objects and startup file were designed to be able to have all items in convenient keyed dictionaries for easy lookup and a map 4D tuple that allows locations to be indexed by coordinate. This system works well for convenience but due to the restrictions on dictionaries in python there’s a couple problems such as not allowing duplicates or causing lots of key value errors due to casing. Another source of inefficiency is that the MAP tuple is passed back with many empty spaces that will still need to be looped through.

Instead of creating all the game objects from scratch every time it should be stored in a database or memory file, also that way things could be changed on the fly through versions.

Python **dictionaries** don't support **duplicate keys**. One way around is to store lists or sets inside the **dictionary, so make all dictionary values a list so it’s a dictionary of lists.** <https://stackoverflow.com/questions/10664856/make-a-dictionary-with-duplicate-keys-in-python>

Maps should also not be based on coordinates but should be an adjacency lists. Basically instead of referencing coordinates it references where it’s adjacent to and the value of distance.

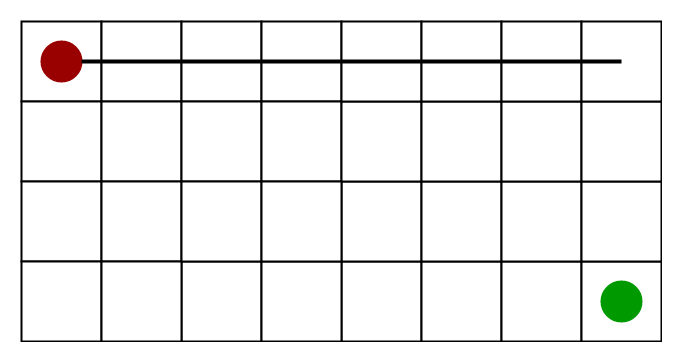
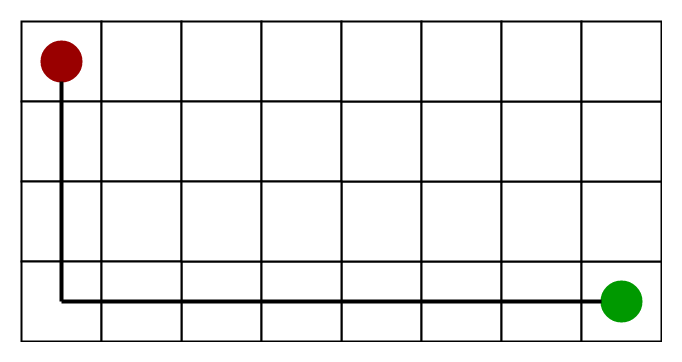
Quick aside on A\* search algorithm:

<https://www.geeksforgeeks.org/a-search-algorithm/>

Algorithm to find shortest path from source to target.

We take the spot with the minimum f for all squares/nodes we can move, where f = g + h, where g = the movement cost to the square, and h = the absolute distance from the source to the target.

H can be calculated exactly using Euclidian geometry or approximated using heuristics such as the Manhatten coordinate distance or the diagonal distance (longer of x or y).



## Game Loop

The game loop is the heart of controlling any game and EPTA is no different. The game loop only runs when the player is alive. This was designed this was from the start when the game was more linear and this would lead to the end. The first part of the game loop is getting a command from the player. If an auto script is running it will enter in the command automatically but if not it waits for the player to type in a command (because it’s a text based adventure we don’t have to deal with frame rates or other types of event inputs).

If the player dies in this loop it enters the end function which deals with happens when the game ends.

Whatever the command is it gets sent to the parser which then uses a rudimentary InfoCom-style parser with a bunch of short cuts to try to interpret what the user is trying to do. This is a basic form of natural language processing (NLP) and in some ways makes a text adventure harder to design/play as the input text can have a lot more variation (as compared to a controller or well defined set of keys) even though you may be trying to do the same thing. In addition to the lack of visuals this control barrier, I believe, is what makes the interactive fiction (IF) genre less accessible to new players then other game genres.

If the parser does interpret a command it is then responsible for calling that game command. That command called the command in GameFunctions which will call methods in GameClasses.py to update the world.

The main loop then checks the game world state to see if story, quests, or events are triggered. If they are then they update the game world.

The final part of the game loop checks if the player is entering a nested loop it loads the base game state and sets up the nested loop.

I believe aside from the other components this game loop is exactly how any game loop runs: take in inputs, update the world, output the changes. It will be the entire game around it which should change to support this game loop containing the entire game structure.

Game loop should run always while the game is running

Game Functions:

Combat/Stats

Talking/Questing

Interiors

# Lessons Learned

Never use global variables (pass by reference with all functions in object method)

1. Game loop: Add everything to game loop structure

2. Save states: Modulazerize and decouple save states/objects

3. Maps: Adjacency list and proper graph theory with list dictionaries (or some nesting) for duplicates

4. Object oreinted decoupling, encapsulation, polymorphism for all objects

Objects even for interfaces, quests, EVERYTHING