Curtin College

Bentley Campus

**User Documentation**

**Assignment 2:**

**Vampirism Epidemic Simulation**

Mamert Vonn G. Santelices

Student ID: 90026174

The following program creates a simulation of set population wherein a prompted number of humans and vampires populate the scene. Within the lifetime of one scene, would undergo several timesteps simulating interactions between the entities of set population. The data gathered from the program would helpfully inform the World Health Organization to predict the epidemic of the rare bat virus in the jungles of Africa.

The Simulation is consisting of two core classes: Scene class and Entity class. The Scene is populated by a set of Entity classes as it’s children. It contains all the functionalities shared by the entities. A Scene class can be overridden to write custom code for a \_ready function and a start function, which is primarily used by the program as the on ready instructions and loop instructions, respectively.

Moreover, the Scene contains several components that are essential to the Entity class. Such as, the instantiated Canvas class within Scene used in drawing DataFrames into an interactive scatter plot; an instantiated CollisionMap within Scene that checks for entity interactions within the default of 8 spaces of an entity, wherein the Von Neumann neighbourhood algorithm is utilised to present the collision of several diamond shaped region as an interaction; an instantiated OStream as an optional output stream to be printed both in the terminal and a file; and the list of event\_handlers that propagates events and changes that would mutate or delete an entity. Additionally, the list of \_entity\_types and \_event\_types can be overridden to respectively declare the following agents to be used on the scene and the following events propagated at the end of loops within the scene. Simply, Scene is the default class used for this program but can be inherited and overridden to perform different things.

Furthermore, the Entity class have several polymorphs. The Agents class, representing the active agents within the scene, is inherited by the Human class and the Vampire class, while the Consumeable class directly inherits from the Entity class as they are not active and are static objects within the scene. The Agent shares a moving functionality, using the collision map to searches for all vacant space from entity and move within one of those spaces randomly. The Agent class is constructed by passing a movement value, which are 4 and 8 for Human class and Vampire class, respectively. The movement value dictates the space within the grid that Agent class can travel within a single timestep. More importantly, it is important to understand that the Entity class does not store its data locally but directly stores it to the respective Dataframe, hence each class inheriting Entity class has a static Instance of a DataFrame for that class. Essentially, the Entity class acts as a mediator to a pipeline to the DataFrame containing the stored value. Therefore, when an Entity is created it then writes the data to a static map of that class type, and subsequently when the class is destroyed then the data is dropped within the directly DataFrame. Consequently, setters and getters are used to access the data of an Entity class which respectively modifies the data and returns an rvalue of the data from the DataFrame.

Lastly, the program uses a Signal class and an Event class to notify the instances of Entity of certain interactions, such as:

* **Human and human interaction**: some people are selfish, and others are helpful. When two humans interact there is a 40% chance that one human uses the other to gain 20 health, which means the other loses 20 health. There is also a 60% that the humans help each other, and both gain 10 health.
* **Human and vampire interaction**: vampires like biting. When a human and vampire meet the human has a 70% chance of being bitten by a vampire and becoming a vampire. The vampire has a 30% chance of being killed by the human.
* **Vampire and vampire interaction**: vampires like biting, so much so, they can’t help but sneak in a cheeky bite at any opportunity they get. As a result, when vampires come into contact, they bite each other, with the result being that both vampires lose 20 health.

Signals are used to buffer the result of an interaction between two or more Entities. It is done so by storing a callback function of the resulting interactions, since some of the given interactions are done randomly. As a result, it ensures that the two entities don’t perform the same interaction again. As shown in the find\_interaction\_from function in the CollisionMap, neighbors that had already performed an interaction is buffered on a dictionary and excluded from the neighbors found in the find\_nearest fuction of the corresponding entity for that interaction. The neighbors are then used to execute an interaction and buffer the resulting return value to the (excluded) dictionary. Finally, changes retrieved from the (excluded) dictionary are emitted to the current Entity class viewed on the loop allowing all resulting interactions to simultaneously occur within the current Entity class. Also, a Scene has a pointer to all its children (instance of Entity class) and all Entity class has a pointer to its parent scene enabling the Entity class to access the Scene’s components and vice versa. Additionally, the classes inheriting from Entity class contain a static DataFrame map contained in a InstanceDB class created when a scene calls use\_on\_scene on its entity\_types list. As a result, it creates the InstanceDB on a dictionary wherein the key value is the repr() value of Scene (the unique memory address of Scene)

Events function a little more differently. It buffers any mutation and deletion of Entity classes to prevent it from interfering with the CollisionMap when looking for interactions. It applies all the changes at the end of the loop to prevent undefined behaviour when executing/emitting Signals on the CollisionMap. It is important to know that the arrangement of the Events in the list of \_event\_types in Scene matters, as it guarantees that the closest event type of the list is prioritise and executed first. The following events use in the program are:

* **ConsumeEvent** – given a Consumeable instance, it randomly picks from the list of nearest Human instances from it to consume and receive its effect. Thereafter, deleting the Consumeable instance and dropping its data from the static Consumeable DataFrame
* **DeathEvent** – checks for Deaths given the condition is\_alive() return value of an Agent Class is False. For instance, Human instances return False for is\_alive() if they have zero or less health or an age greater than 70. In contrast, Vampire instance only returns False for is\_alive() if they have zero or less health
* **InfectEvent** – performs a mutation that changes a Human class to a Vampire Class, simultaneously dropping the data of the Human class from the static Human DataFrame and adding it to the static Vampire DataFrame. As well as, replacing the Human instance in the Scene.\_\_entities with a Vampire instance.
* **SlainEvent** – happens when a Vampire instance is slain by another Human instance, which deletes the Vampire instance from the Scene.\_\_entities and drops its data from the static Vampire DataFrame.

The program accepts the following arguments when running it: human\_count, vampire\_count, total\_timesteps, dimensions, and a file path to a seed.txt (deterministically recreate the results of something). Otherwise, if the values are not entered it uses the default values, which are: humans\_count=100, vampires\_count=10, total\_timesteps=20, dimensions=(256,256), seed=None. A seed.txt is produced when the program is runned so to re-run and reproduce the same results.

Perform the following inside of src directory:

$ python3 simulation.py [human\_count] [vampire\_count] [total\_timesteps] [dimensions] [seed]

A sweep execution can also be run to perform multiple simulations within a range of starting human count and vampire count. The sweep stores the results in the bin directory. The sweep parameters are starting\_human\_count, end\_human\_count, starting\_vampire\_count, end\_vampire\_count and dimensions. By default, the sweep parameter uses a 64x64 grid and a total of 100 timesteps.

Perform the following within the root of the directory:

$ sh sweep.sh [starting\_human\_count] [end\_human\_count] [starting\_vampire\_count] [end\_vampire\_count] [dimensions]