COVID-24

GOJON Baptiste • JALLERAT Mathieu • LE TOUZÉ Liam • YAACOUB Peter

Table of contents

Game rules

The rules of the game are simple. Infect and then kill as many people as possible in the shortest amount of time. To achieve this, the disease designer has access to a control panel that allows them to see the spread of their disease on a world map in near real-time.

In the conceptual phase, the designer can choose to start the disease spread in the desired region. Depending on the chosen region, the spread can be more or less rapid. For more difficulty, we recommend starting with Oceania, tough with its fighting kangaroos and hairy spiders. For simplicity, a zone in Asia can't hurt. It's like starting in Wuhan but better, you'll see.

The designer can also use their toolbox to improve the attributes of their disease:

- Lethality: the ability to kill the infected population.
- Infectivity: the ability to infect the healthy population.
- Camouflage: the ability to hide from cures.
- Reassembly: the ability to regroup against a cure.
- Heat resistance: the ability to spread in hot areas (≥ 20 °C).
- Cold resistance: the ability to spread in cold areas (< 20 °C).

Finally, the designer can access detailed population data in each zone of planet Earth. Thankfully, they can't infect other planets!

The end is never reached, and there will always be a certain number of genetically advantaged humans who will survive no matter what. The goal is not to eradicate everyone but to eradicate the weakest as quickly as possible.

Thus, strategy is key. Killing everyone too early would reduce the chances of the disease spreading. Killing too late would increase the chances of the disease being cured too soon.

Stay alert and don't forget your main goal: to annihilate humanity.

Objectives

Our project was inspired by the game Plague Inc (Figure 1). Our ambition was to design a game in a similar vein, where players create and spread a disease to eradicate the world population. Players can name their disease at the start of the game and choose the region of the world where it begins to spread. The chosen region will dictate the difficulty of the game. Over time, players accumulate points that they can invest to improve various aspects of their disease, such as infectivity, lethality, or resistance to different environments. The game is continuously evolving and never truly ends.

Firstly, we focused on designing an intuitive and attractive graphical interface, providing players with clear information on the percentage of the population that is healthy, infected, recovered, or deceased. We wanted to display a map with distinct countries or zones to infect. Next, we worked on developing a user interface that allows players to upgrade their disease as soon as they have accumulated enough points. Finally, our most complex objective was to design and implement partial differential equations to model the disease's progression based on its parameters, which evolve according to the upgrades made.

Figure 1 : Plague Inc: Evolved (Ndemic Creations)

During the programming, we maintained a non-exhaustive task list in our README.md file so that each group member could take ownership of a goal. This list is ordered by completion and alphabetically. It represents all the achieved and abandoned objectives, with details of the problems encountered and the solutions adopted.

Models

- ☑ **Add data to regions.**
  Once all data is stored in CSV files, the `World` model creates all regions and assigns their respective data via the `csv` module.

- ☑ **Implement the point system.**
  The system is attached to the disease model. Points continuously increase through a thread when the map is visible.

- ☑ **Implement `Region` and `World` models.**
  The `World` model is created when choosing the initial region and manages the creation and storage of different regions.

- ☑ **Get the news.**
  The news was written but not synced on our GitHub repository, so it was necessary to retrieve them. Initially, we considered a list, but eventually, a CSV file was created.

- ☑ **Retrieve data.**
  For the realistic aspect of our game, it was necessary to collect accurate and categorized data by chosen zones. Population data was already available, and Baptiste compiled data for areas and temperatures.

- ☑ **Revise the `Region` model.**
  The model now contains all necessary information: name, map, populations (infected, deceased, healthy, and recovered), area, temperature, and neighbors.

- ☑ **Review disease properties.**
  The disease properties of airborne and waterborne transmission were eventually removed due to low priority, particularly related to the Humidity.csv file.

- ☑ **Find a way to link neighboring regions.**
  Neighboring regions are stored in a CSV file. Although not all zones necessarily touch another (e.g., the Caribbean), we allowed creating maritime and air links between geographically and culturally close regions. Each zone is limited to four neighbors.

- ☐ **Add the Humidity.csv file.**
  This file was ultimately never added due to low priority.

- ☐ **Modify data over time.**
  Changes in data, such as population since 1960 or temperature due to climate change, were not pursued due to low priority.

Controllers

- [x] **Finish revising the Improvements file.**
  The file was simultaneously written by Mathieu and Peter, so it was necessary to review it for code and interface consistency.

- [x] **Evolve populations based on neighbors.**
  In parallel with Euler's disease propagation algorithm, we created an algorithm to propagate disease through neighboring zones, injecting a random component to simulate realistic uncertainties.

- [x] **Calculate colors in `MapController`.**
  We decided to determine zone colors based on infection and death rates, using red for infection and purple for death. Colors for health and recovery rates were possible but deemed aesthetically questionable and low priority.

- [x] **Replace `PopulationController`.**
  Initially unsure where to implement Euler's algorithm coded by Liam, we first created a separate controller and eventually implemented it in `MapController`.

- [x] **Verify coefficients for improvement parameters.**
  After establishing Euler's algorithm and disease improvements, we linked them for a coherent disease progression, neither too fast nor too slow. We proceeded through trial and error, conducting multiple tests for more or less coherent results.

- [x] **Ensure news scrolls correctly if the news list is modified.**
  Modifying the news list works as expected, but this modification functionality was not used in the code due to low priority and its time-consuming nature. The news list is predefined and does not change during the game.

- [ ] **Force a name for the disease.**
  We chose not to force a name for the disease. Although having an unnamed disease is unusual, there's no reason it can't be done. The default name for the disease is the game's name: COVID-24.

Views

- [x] **Fix font issue on Windows.**
  The line `font=("Courier Bold", 80)` works on macOS but not on Windows. We resolved the issue using `font=("Courier", 80, "bold")`.

- [x] **Limit button action to opening a single window.**
  Initially, we thought of disabling and re-enabling buttons once the window was displayed and disappeared, respectively. Although feasible in theory, it was impractical. We decided to hide the map window with `self.withdraw()` while another window was displayed and make it reappear with `self.deiconify()`.

- [x] **Revise the design and implementation of the `Information` view.**
  The view contains a lot of information, so we sorted and adjusted interface elements for visibility. We initially considered a `Scrollbar` but opted for the simple and quick solution of reducing font size.

- [x] **Test the map to improve performance.**
  Displaying images on `Tkinter` was not as trivial as we thought. Using the `Pillow` library, we devised a new method: first, we apply color modifications for each region, then combine all images into one for display. Reducing image dimensions without compromising quality was crucial.

- [x] **Verify the interface on Linux, Windows, and macOS.**
  Although the game was primarily tested on Linux, it was important to test on all used platforms. The team uses computers running Windows and macOS, while the target OS is Linux.

- [x] **Check the use of `self.state("zoomed")` for Windows.**
  The method works on Windows but not on macOS and Linux. The fullscreen mode, frequently used and somewhat complex, is managed by the `set_full_screen` method in a global class.

- [ ] **Add an event to exit fullscreen mode.**
  This was not done due to lack of time and low priority.

- [ ] **Create a settings window to disable music.**
  Initially, the idea was to create a dedicated window to disable music and exit the game. For low-priority reasons, we chose to add a "Quit" button visible above the map to exit the game.

Autre

- ☑ **Add music.**
  Like any intense game, we needed music to complement the interface. Peter composed a 5-minute music piece inspired by a science fiction theme.

- ☑ **Write the Docstring.**
  A tedious task, we used ChatGPT to speed up this process and adhere to standards.

- ☑ **Clean the code.**
  This step was never really necessary as it was continuously done during development. However, we used the code documentation step to review the cleanliness and readability of our codebase.

- ☑ **Revise the game name if starting in 1960.**
  We decided to keep the name COVID-24 and link the number to the game's publication year. For example, a new version released in 2025 would be called COVID-25.

- ☑ **Test the final game.**
  The final version on Linux was tested on a computer station at the Bibliothèque Marie Curie (BMC) and via the virtual desktop.

Non-trivial algorithm

The non-trivial calculation in our algorithm is the propagation of our virus within a region. For this, we use the HIRD model (SIRD in French), which divides the population into four categories (Healthy, Infected, Recovered, Deceased). To model the evolution of these populations, we use the following partial differential equations:

$$\frac{dS}{dt} = -\ \beta SI$$
$$\frac{dI}{dt} = \beta SI - \gamma I - \delta I$$
$$\frac{dR}{dt} = \gamma I$$
$$\frac{dM}{dt} = \delta I$$

S: Number of healthy individuals.
I: Number of infected individuals.
R: Number of recovered individuals.
M: Number of deceased individuals.
β: Infection rate.
γ: Recovery rate.
δ: Mortality rate.

Our algorithm in `MapController` starts by looping through all the regions to calculate the new population. It then defines all the necessary parameters (β, γ, δ), as well as the different populations (S, I, R, M).

Next comes the calculation using Euler's method, which relies on a local approximation of the solution using a first-order Taylor expansion. For this, we set our sequences S_n, I_n, R_n, M_n and take : $S_{n+1} = S_n + \frac{dS}{dt}(\beta, S, I) \times \Delta t$ (similarly for I, R, M).

The smaller the dt, the greater the accuracy, but the computation time increases. Therefore, a compromise must be found. Our algorithm performs a loop as long as the counter t, incremented by Δt at each iteration, is less than 10 (10 makes our game more dynamic). While in the loop, we calculate S_{n+1}, so that in the end, we get S_n, the healthy population 10 days later (similarly for I_n, R_n, M_n).

Object-oriented programming

We organized our code using the Model-View-Controller (MVC) architecture, meaning the view has access to a controller that has access to the models. Of course, this architecture is not the only one that exists, but it is intuitive and works well with Tkinter, an imperative GUI library.

Our excessive use of private attributes and "get" and "set" methods is not incidental. This allows us to modify the internal structure of the model without having to change the code in the controllers that access it.

We schematized all our classes in a simplified UML diagram that contains only the attributes, not the methods or the Tkinter widgets (Figure 2). We also do not represent the `__main__` entry point nor the `Global` class.

Moreover, the diagram does not show it, but we only create one instance of `Disease` and `World`, which is then propagated from controller to view, and from view to controller. Obviously, the view never directly accesses the model.

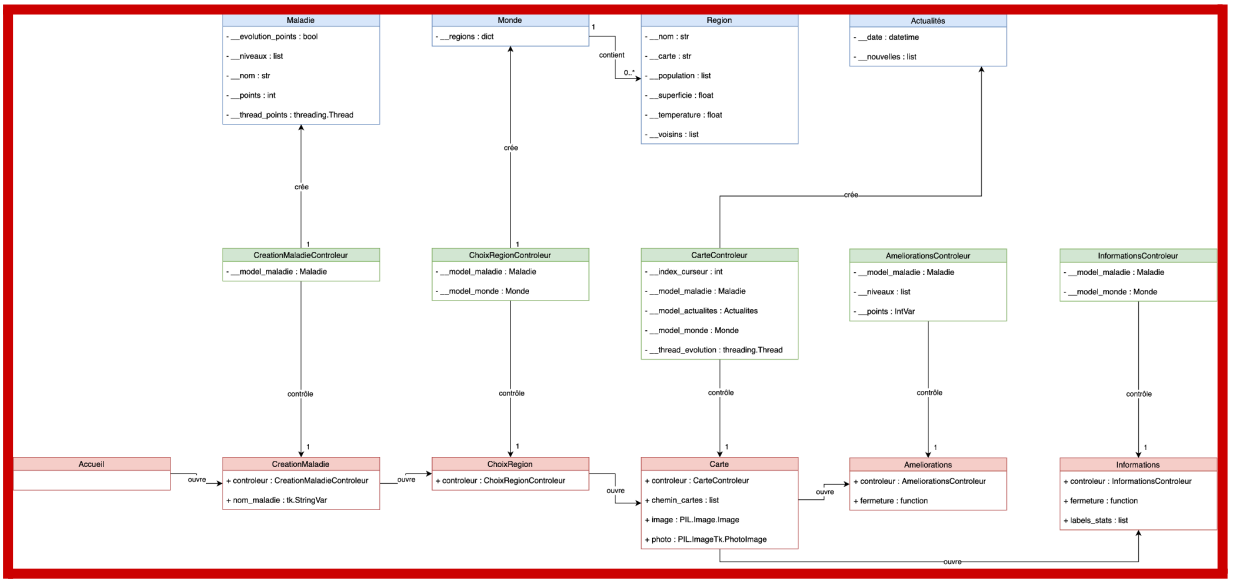

Figure 2 : Simplified UML class diagram (in French)

For the Human-Machine Interface, we used Tkinter. In hindsight, we might have used other modules, such as Pygame, for better performance. However, this option would have cost us time during the learning process.

Additionally, we tried to limit the number of views without concentrating all the widgets in one space (Figure 3). If the map areas were individually selectable, we could have done without the `Information` view. We could also have combined the `DiseaseCreation` and `RegionChoice` views, but starting the game smoothly was essential to avoid overwhelming the user from the beginning.

Figure 3 : The different views of COVID-24 (in French)

Sources

Data

- Admin-Lk. (2020, août 27). Le climat, une machine qui nécessite de l'énergie. Encyclopédie de L'énergie. https://www.encyclopedie-energie.org/le-climat-une-machine-qui-necessite-de-lenergie/

- United Nations Statistics Division. (2016). Population by sex, annual rate of population increase, surface area and density [PDF]. https://unstats.un.org/unsd/demographic/products/dyb/dyb2016/Table03.pdf

Algorithm

- INSA de Rouen. (2010, juin 16). Rapport de recherche de l'INSA de Rouen sur la Modélisation de la propagation d'un virus. https://moodle.insa-rouen.fr/pluginfile.php/21120/mod_folder/content/0/Rapport_P6-3_2010_01.pdf?forcedownload=1

- Physique TSI1 Troyes. (2020, 27 mai). Introduction à la méthode d'Euler en python [Vidéo]. YouTube. https://www.youtube.com/watch?v=-d7qrNkPDtQ