4. Practice - NetLogo Model Design

Tamás Takács, PhD student, Department of Artificial Intelligence

90 min read

January 25, 2025

Collective Intelligence

Last Practice

In our last practice, we covered:

- The Models Library and Benchmark Simulations
- The Fire model
- Extensions of the Fire model
- Phase Transitions and Tipping Points
- BehaviorSpace experiments
- The Info Tab
- Schelling's Segregation model
- Extensions of Schelling's Segregation model
- Plotting
- The Virus on a Network model
- Links
- Extensions of the Virus on a Network model

That’s quite a lot again! Now, let’s dive into how you can design and create your own NetLogo model completely from scratch.

Simple Economy Model

In 1996, Josh Epstein and Rob Axtell published one of the first definitive books on agent-based modeling and social science, *Growing Artificial Societies*, which featured artificial economic agents.

We will create a simple model of economic agents, inspired in part by Epstein and Axtell's work and a paper by Dragulescu and Yakovenko (2000).

Dragulescu, A., & Yakovenko, V. M. (2000). Statistical mechanics of money. The European Physical Journal B, 17(4), 723–729. doi:10.1007/s100510070114

The Rules

- 500 people start off with $100 each (starting with a uniform distribution).
- At every tick, each person gives $1 to another person randomly.
- If you run out of money, you can't give any more money away until someone gives you money.

Simple, right? Let's try to implement it in NetLogo!

First Step - `setup`

Designing the `setup` procedure is usually the first step, followed by the `go` procedure. These two components are not always required, but they are a style heavily utilized in NetLogo. Let's start with an empty project:

File > New

First, let's set our grid parameters before adding anything to the model. These can be edited to your liking; however, in this example, we will use the following parameters:

- **Location of Origin**: Corner, Bottom Left
- **max-pxcor**: 500
- **min-pycor**: 80
- **Patch size**: 1
- **Font size**: 10
- **Frame rate**: 30

One of the first things to add is a `setup` button next to the 2D environment grid. After adding the `setup` button, go to the **Code Tab** and create a basic skeleton for the `setup` function. A `setup` function should always include the `clear-all` and `reset-ticks` commands.

```
to setup
    clear-all
    ;; setup code
    reset-ticks
end
```

Looking at the first rule of the exercise, it requires us to create 500 agents and assign them $100 as a starting point. This can be implemented using the `create-turtles` command and defining agent attributes. Additionally, we can set them to a circular shape with green color and size 2 for convenience.

```
turtles-own [ wealth ]

create-turtles 500 [
    set wealth 100
    set shape "circle"
    set color green
    set size 2
]
```

Running the `setup` creates the turtles, but we cannot see them because all turtles are created at the `(0, 0)` patch by default. Let's modify the code so that the agents spawn in specific locations.

> **Note:** NetLogo will create all turtles at the (0, 0) patch if no location is specified in the command.

```
to setup
  clear-all
  create-turtles 500 [
    set wealth 100
    set shape "circle"
    set color green
    set size 5
    setxy wealth random-ycor
  ]
  reset-ticks
end
```

The goal is to visualize the wealth distribution. Rich agents should move to the right side of the grid, while poorer agents move to the left. Initially, all agents start in a random row with their x-coordinate (horizontal alignment) set to match their wealth.

Second Step - `go`

In the `go` procedure, turtles must transact their wealth if they have any. This can be done by calling a `transact` command (defined later). All `go` commands should end with the `tick` command.

```
ask turtles with [ wealth > 0 ] [ transact ]
```

This ensures only agents with wealth can give money to others. However, our grid size only allows a total wealth of $500 for a single agent. To avoid errors, we limit the simulation with the following condition:

```
ask turtles [ if wealth <= max-pxcor [ set xcor wealth ]]
```

This ensures that agents with wealth exceeding 500 do not move further right on the grid. The final `go` procedure looks like this:

```
to go
    ask turtles with [ wealth > 0 ] [ transact ]
    ask turtles [ if wealth <= max-pxcor [ set xcor wealth ]]
    tick
end
```

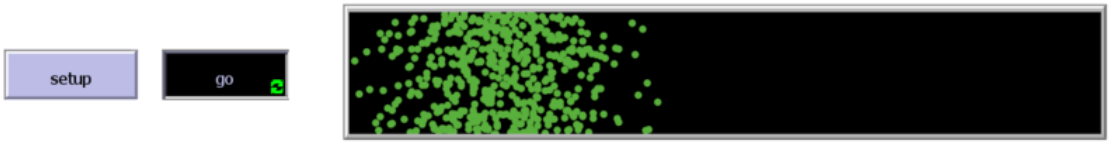
Don't forget to also add the `go` Button to the Interface Tab.

The `transact` Command

The `transact` command reduces one unit of wealth from the agent and gives it to a random agent. The `set` command is used to reduce wealth, while `ask one-of other turtles` is used to transfer wealth to a random recipient.

```
to transact
    set wealth wealth - 1
    ask one-of other turtles [ set wealth wealth + 1 ]
end
```

At this point, the simulation should work as intended. Agents will move horizontally across the grid, visualizing the wealth distribution as the simulation progresses.



Exercise: Run the `setup` command in the Interface tab and press the `go` button to start the simulation.

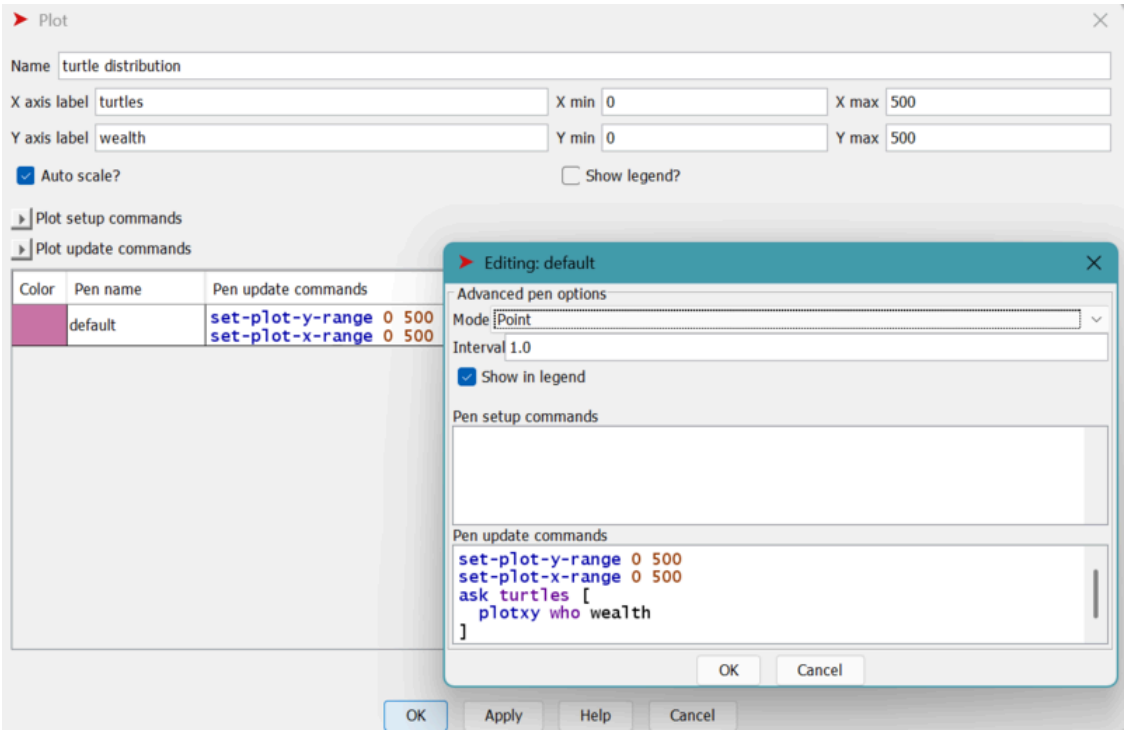
Analyze the wealth distribution in the system:

- Based on your observations, hypothesize the probability density function (PDF) that the wealth distribution might follow.
- What factors could contribute to the shape of the wealth distribution?
- Provide a formal justification for your guess before verifying with additional experiments.

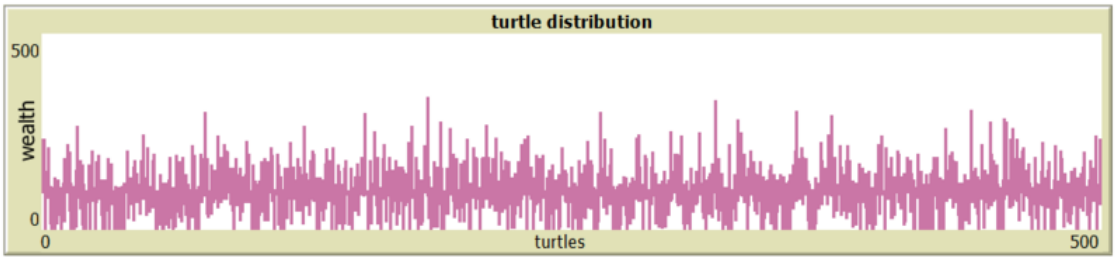
The simulation works, agents are moving around, and wealth is being redistributed. However, to better understand the inner dynamics of the model, we need additional insights through **plots** and **monitors** that provide valuable analytics.

Let's create a **point plot** that visualizes the wealth distribution. The **x-axis** will represent each agent's unique `who` ID, and the **y-axis** will represent their wealth. The plot will be named **"Turtle Distribution"**, and we will also include a **legend** displaying the total wealth in the system, which remains constant at $50,000 ($100 * 500 agents).

To achieve this, we need to plot each agent's `who` ID against their respective wealth as points on the graph. This can be done easily using the following configuration:



We set the plot's maximum x and y ranges using the `set-plot-x-range` and `set-plot-y-range` commands. These define the bounds of the plot: the **x-axis** for the agents' IDs and the **y-axis** for their wealth. Then, we ask the turtles to plot their x and y coordinates, which correspond to their `who` ID and wealth, respectively. This ensures each agent's wealth is represented as a point on the plot.



Exercise: Observe the wealth distribution in the simulation. Over time, the dynamics of the simulation result in a noisy distribution of wealth among the agents. As the simulation progresses through many steps, the wealth starts to become increasingly unequal.

- What part of the simulation dynamics contributes to this phenomenon?
- Why does wealth inequality emerge despite the random nature of the transactions?

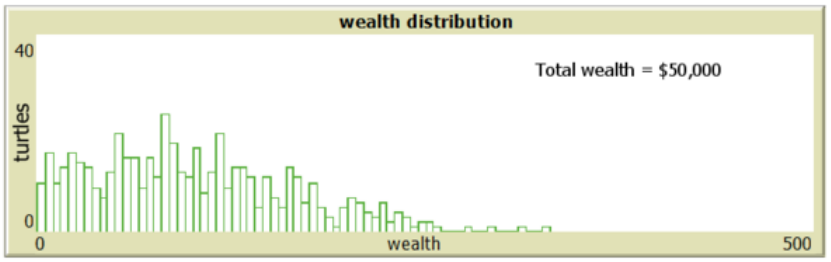
Exercise: Create another plot, but this time switch the roles of the x and y axes. In this plot, we aim to explore how different amounts of wealth are distributed among agents. Follow these steps:

- Use a **bar plot** with wealth as the x-axis and the number of turtles as the y-axis.
- Set the interval for the wealth bins to 5.
- Set the **maximum x value** to 500 (wealth) and the **maximum y value** to 40 turtles.
- Label the plot **wealth distribution** and add a legend indicating the total number of wealth ($50,000).

Think about what this plot reveals. How does the wealth distribution change over time?

Solution:

▶ **Click to show/hide solution**



For additional analytics, we want to calculate two key metrics:

1. **The wealth of the top 10% of agents** in the environment (to understand how much wealth is controlled by the richest agents).
2. **The total wealth of the bottom 50% of agents** (to assess the distribution of wealth among the poorer agents).

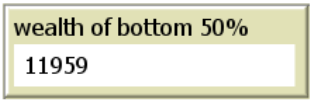
To achieve this, we can create two reporters in our NetLogo code: one to calculate the total wealth of the **top 10% of agents** and another to calculate the total wealth of the **bottom 50% of agents**.

```
to-report top-10-pct-wealth
  report sum [ wealth ] of max-n-of (count turtles * 0.10) turtles [ wealth ]
end
```

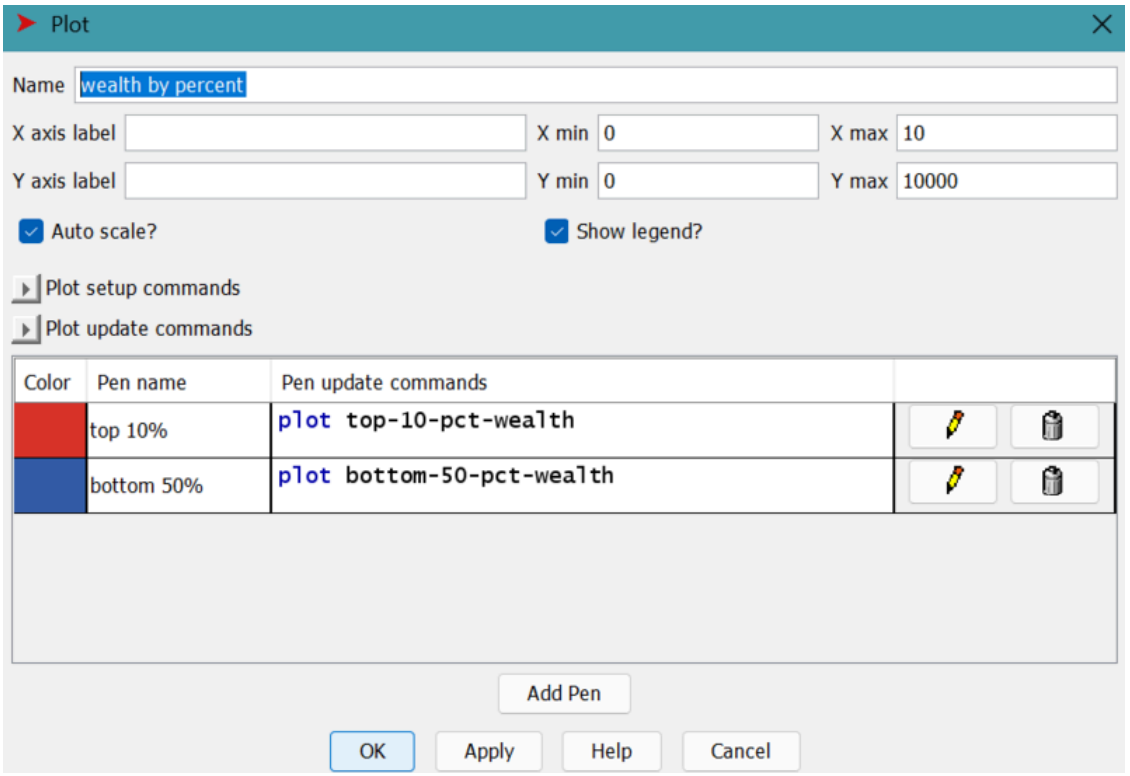
```
to-report bottom-50-pct-wealth
  report sum [ wealth ] of min-n-of (count turtles * 0.50) turtles [ wealth ]
end
```

The `max-n-of` reporter retrieves the **n turtles** with the **highest wealth values**, where `n` is 10% of the total number of turtles (in this case, `count turtles * 0.1`). Similarly, the `min-n-of` reporter retrieves the **n turtles** with the **lowest wealth values**, based on the `wealth` attribute.

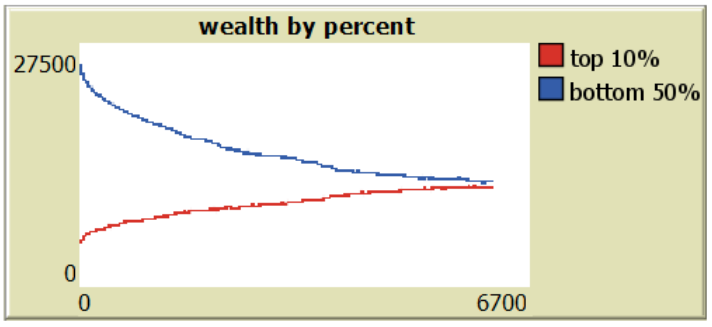
To complete the task, add two respective monitor elements to the interface.



We can also plot these values on a **Plot Element** to observe how the wealth distribution changes over time. With the reporters for the **top 10% wealth** and **bottom 50% wealth** already created, these values can be dynamically added to a plot during the simulation.



Which results in something like this:



As the simulation progresses, the lines on the plot will likely cross each other. Over time, the **top 10% of agents** will control the majority of the wealth in the economy, while the **bottom 50% of agents** will control only around **20% of the total wealth**. This illustrates the emergence of wealth inequality in the system. Is this always the case?

Exercise: Using the BehaviorSpace experiment tool, create an experiment named `wealth-distribution` to measure the **top 10%** and **bottom 50%** wealth values in the system.

- Ensure the experiment collects these metrics only at the **end of the simulation**; intermediate steps are not needed.
- Set the experiment to run for a total of **10 repetitions** to account for variability.
- Limit the simulation to **10,000 ticks**.
- Analyze the final wealth distribution from the results.

Based on the dynamics of this wealth redistribution system, the wealth distribution tends to evolve into a **Boltzmann-Gibbs distribution**, which is also referred to as an **exponential distribution** in statistical mechanics.

Key Characteristics of the Emergent Distribution:

1. The probability of an agent having wealth (w) follows: $[P(w) = \frac{1}{\langle w \rangle} e^{-w/\langle w \rangle}]$ where $(\langle w \rangle)$ is the average wealth in the system.
2. The random exchange of wealth between agents leads to this exponential distribution, as it mirrors the energy exchange dynamics observed in gas molecules in statistical mechanics.

Exercise: Create a `Slider` element to control the number of agents in the simulation dynamically. Follow these steps:

- Add a global variable (e.g., `agent-count`) to store the number of agents.
- Configure the `Slider` in the Interface tab with appropriate minimum, maximum, and default values (e.g., min: 10, max: 500, default: 100).
- Ensure the `setup` procedure uses the value of `agent-count` to create the corresponding number of turtles.
- Update the plot configurations dynamically to reflect changes in the number of agents:
  - Set the x-axis range of the plot to `0` to `agent-count`.
  - Ensure the plots scale dynamically as the number of agents changes.

Extending the Model

Exercise: Modify the rules of the system to allow agents to go into debt. Specifically:

- Remove the restriction that prevents agents with `0 wealth` from giving money.
- Allow agents to give money even if their wealth drops below `0`.

Observe the effect of this rule change on the wealth distribution over time:

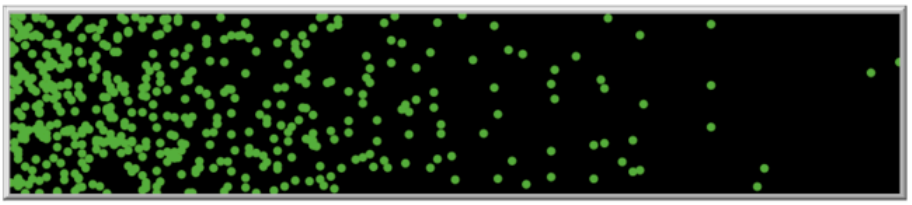
- Does the system still follow an exponential distribution?
- What happens to the wealth dynamics as debt accumulates in the system?

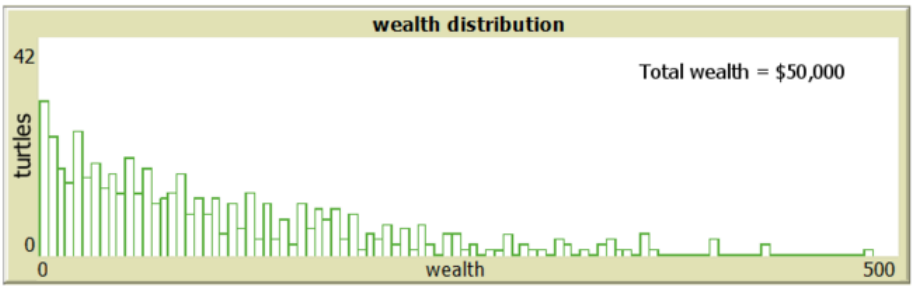
Solution:
▶ **Click to show/hide solution**

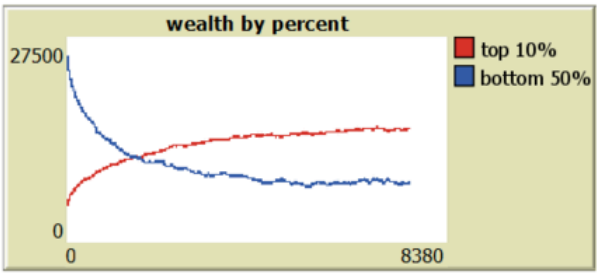
Exercise: Change the transaction rule so agents give out more money per transaction (e.g., from `$1` to `$5` or more).

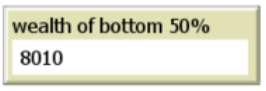
- What happens to the wealth distribution?
- Does inequality increase or decrease?

Solution:
▶ **Click to show/hide solution**







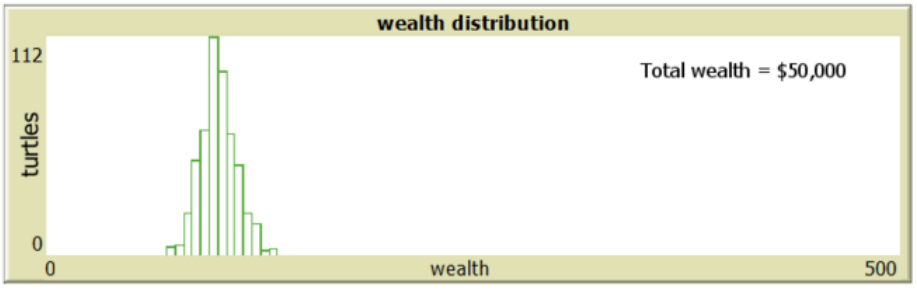


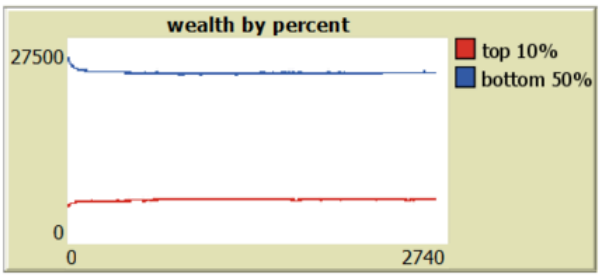
Exercise: Change the rules so that richer agents have a lower chance of receiving money, based on their wealth.

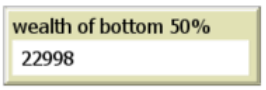
- Try different probability functions, such as `1 / wealth` or `1 / sqrt(wealth)`.
- Observe how this affects the wealth distribution and inequality.

Solution:

▶ **Click to show/hide solution**







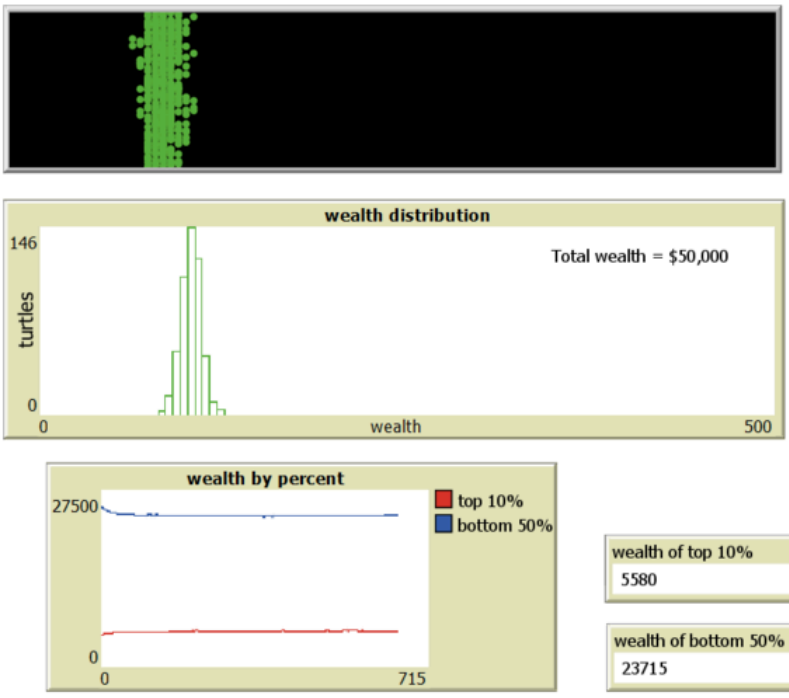
Total Wealth Proportional Probability:

Solution:

▶ **Click to show/hide solution**

Solution:

▶ **Click to show/hide solution**



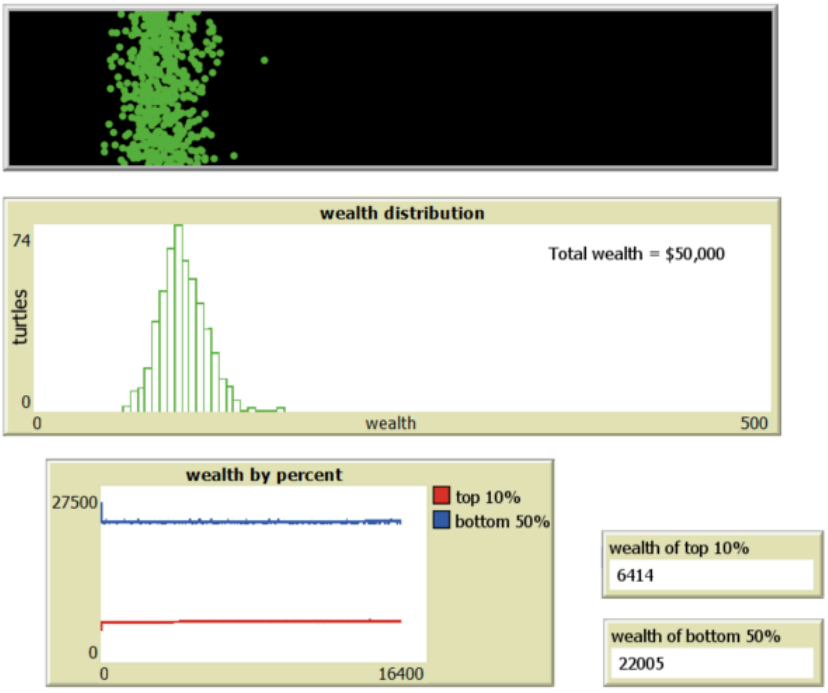
Exercise: Modify the giving rule so that wealthier agents give out more money based on the percentage of their wealth.

- Change the rule so that every agent gives out a fixed percentage (e.g., `5%`) of their current wealth in each transaction.
- Experiment with different percentage values (e.g., `2%`, `10%`, or `20%`).
- Observe how this rule affects the wealth distribution and inequality over time.

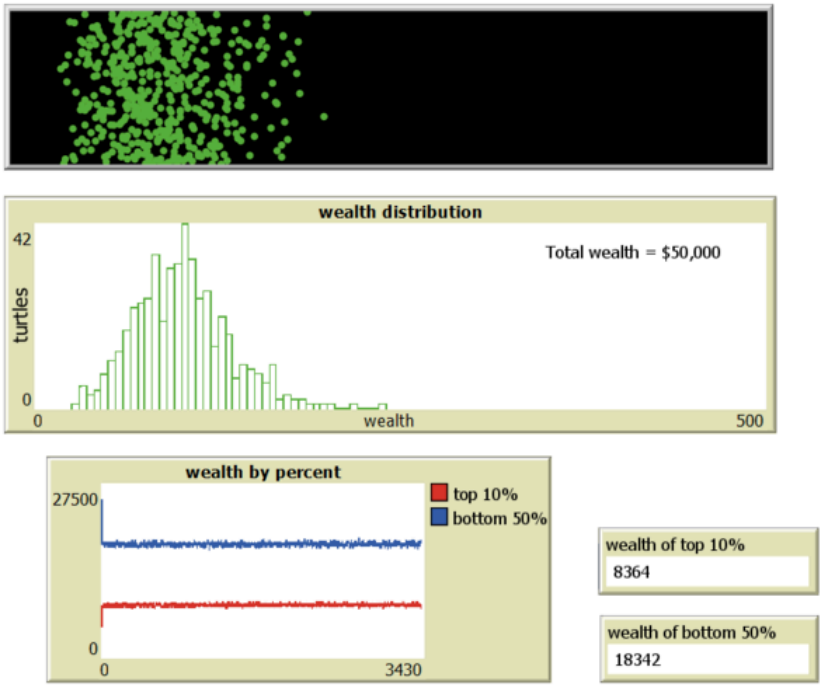
Solution:

▶ **Click to show/hide solution**

With 5%:



With 20%:



This percentage-based payment functions as a form of taxation, disproportionately impacting wealthier agents more than poorer ones. As the tax percentage decreases, the wealth distribution becomes more uniform, with agents converging toward similar living standards. Conversely, increasing the tax percentage flattens the wealth distribution, reducing inequality. However, if the tax rate becomes excessively high, the system reverts to an exponential **Boltzmann-Gibbs distribution**.

Fun Facts:

1. The original model of Epstein and Axtell was called the Sugarscape model, which can be found in the **Models Library**.
2. This redistribution phenomenon aligns with the broader concept in economics and sociology that, in **systems with random exchanges and without redistributive policies, wealth tends to become concentrated among a small fraction of the population.**
3. The simulation outcome mirrors the energy distribution among particles in an ideal gas, where energy is exchanged randomly during collisions.

Extra Exercise: Modify the model to introduce the following features:

- Add a new attribute called `transaction-cap` to turtles, representing the maximum amount an agent can give in a single transaction (based on their wealth).
- Introduce a reputation system: Each agent starts with a `reputation` value of 100. Agents with higher reputation are more likely to receive money.
- Update the `transact` procedure:
  - Agents give an amount based on their `transaction-cap` (e.g., up to 10% of their wealth).
  - The recipient is selected randomly but weighted by their `reputation` (higher reputation = higher chance).
  - Reputation increases for agents who receive money and decreases for agents who give money.
- Update the plots:
  - Create a plot to track the average reputation of agents over time.
  - Modify the wealth distribution plot to include both wealth and reputation effects.

Observe how introducing `transaction-cap` and reputation affects the wealth and reputation distributions over time. Try varying the initial values for `transaction-cap` and `reputation` to see their effects on inequality.

Flattening The Curve

The **COVID-19 pandemic**, one of the most significant global health crises in recent history, exposed the challenges of managing an unprecedented epidemic that disrupted economies and healthcare systems worldwide. The complexity of disease transmission, coupled with the uncertainty surrounding symptoms, prevention strategies, and public health guidelines, made decision-making **incredibly difficult for policymakers**.

In such critical situations, simulation models, particularly agent-based models (ABMs), can play a huge role in exploring potential scenarios and evaluating intervention strategies. While it is impossible to fully replicate the intricacies of a global epidemic, abstracting key elements into a model allows for a deeper understanding of disease dynamics and the potential impact of policy decisions.

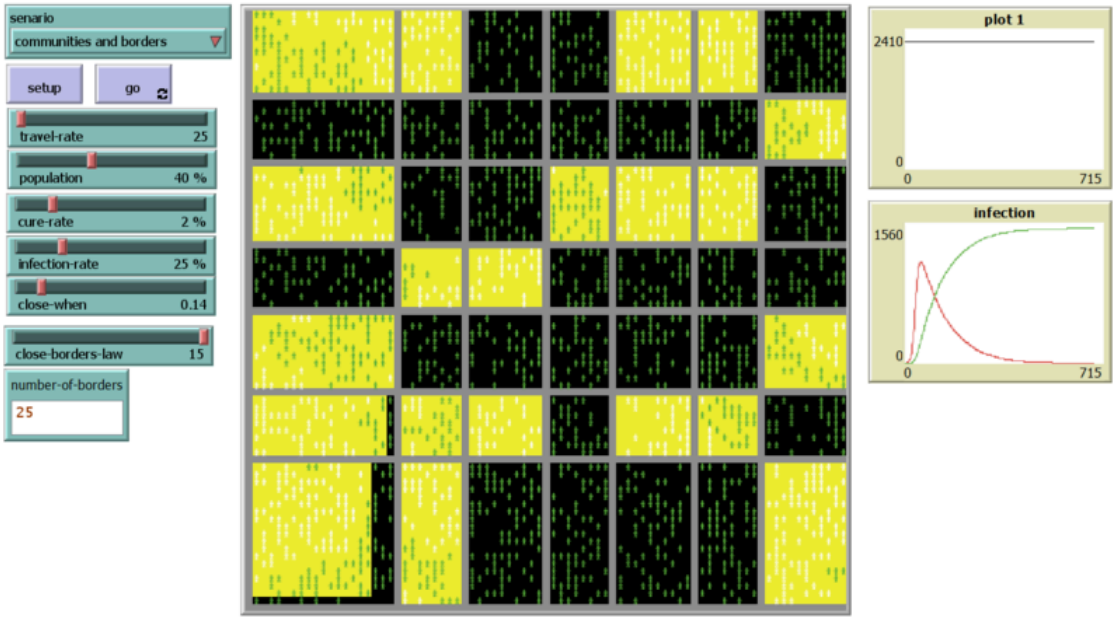
However, modeling public health crises comes with its own set of challenges. Simulations must account for limited data, uncertainty in hyperparameters, and the inherent variability of human behavior. Furthermore, developing and deploying these models requires rigorous validation and ethical considerations, as they inform decisions that directly impact millions of lives.

While not a substitute for real-world experimentation, agent-based models offer a valuable decision-support tool for analyzing scenarios, testing assumptions, and informing evidence-based policy-making during epidemics and other complex healthcare crises.

Modeling Commons

Modeling Commons is a website for sharing and discussing agent-based models written in NetLogo. There are at least 1,000 models contributed by modelers from around the world.

One notable model that gained attention during the COVID-19 pandemic was titled **"Covid-19: How quarantine can flatten the curve"**. This model, while relatively simple in design, focused on exploring the effectiveness of quarantine measures in reducing the spread of infection and "flattening the curve" within a population.



The Code

The code adheres to the typical NetLogo structure, with global variables, breeds, and agent-specific attributes clearly defined. The `borders` global variable serves as a flag to indicate whether the borders are currently closed.

The primary agents in the model are the `actor` breed, which are responsible for carrying and spreading the infection. The `borders` breed, on the other hand, is used primarily for visualization purposes and to act as immovable barriers that restrict movement within the environment. Each `actor` is equipped with an additional attribute, `days`, which tracks the number of days an agent has been infected with the virus.

```
globals [
  borders? ;; Flag to track if borders are closed
]

breed [actors actor] ;; Main agents representing the population
breed [borders border] ;; Temporary breed for creating border visuals

actors-own [
  days ;; Tracks the number of days an actor has been infected
]
```

The `setup` code primarily focuses on initializing the environment, using a somewhat convoluted approach to visualize the borders and set up the simulation. It involves generating the borders through algorithmic placement, spawning healthy agents within the grid, and designating one of the agents as the initial infected individual.

```
to setup
  clear-all
  reset-ticks
  clear-plot

  set borders? FALSE ;; Initialize borders as open

  ;; Scenario-based setup
  (ifelse senario = "base" [
    ;; Base scenario: spawn a proportion of the population on black patches
    ask n-of (count patches with [pcolor = black] * population / 100) patches [
      sprout-actors 1 [
        set shape "person"
        set color white ;; Healthy actors start as white
      ]
    ]
    ask one-of actors [set color red] ;; Infect one random actor
  ]
  ;; Communities and borders scenario
```

```
  senario = "communities" or senario = "communities and borders" [
    create-borders 10 [
      setxy 0 0
      set heading one-of [90 270] ;; Borders align horizontally or vertically
    ]

    ;; Draw initial border lines
    ask borders [
      repeat 4 [
        repeat sqrt (count patches) [
          set pcolor grey ;; Create a horizontal line of borders
          fd 1
        ]
        rt 90 ;; Turn to create the next segment
        repeat sqrt (count patches) + sqrt (count patches) / 4 [
          set pcolor grey ;; Extend the border vertically
          fd 1
        ]
      ]
    ]

    ;; Add additional borders for larger grids
    if sqrt (count patches) > 20 [
      ask borders [
        setxy sqrt(sqrt(count patches)) sqrt(sqrt(count patches)) ;; Place at center
        repeat 4 [
          repeat sqrt (count patches) [
            set pcolor grey
            fd 1
          ]
          rt 90
          repeat sqrt (count patches) + sqrt (count patches) / 4 [
            set pcolor grey
            fd 1
          ]
        ]
      ]
    ]

    ;; Create a central border
    create-borders 1 [
      setxy sqrt(count patches) / 2 sqrt(count patches) / 2
      set heading 90
    ]

    ;; Extend central borders
    ask border 10 [
      repeat 2 [
        repeat sqrt (count patches) [
          fd 1
          set pcolor grey
        ]
        rt 90
      ]
    ]

    ;; Clean up temporary borders
    ask borders [die]

    ;; Spawn actors based on population percentage
    ask n-of (count patches with [pcolor = black] * population / 100) patches with [pcolor = black] [
      sprout-actors 1 [
        set shape "person"
        set color white
      ]
    ]
    ask one-of actors [set color red] ;; Infect one random actor
  ]
  [stop] ;; End setup if no valid scenario
  )
end
```

The `go` Loop

The `go` loop is relatively straightforward but relies on several utility functions to handle various aspects of the simulation. Here's a breakdown of its functionality:

- The loop terminates if there are no infected agents (red-colored actors) remaining.
- Agents move around the map using the `travel` function.
- Infection spreads to neighboring agents through the `infect` function.
- If the proportion of infected agents surpasses the `close-when` threshold, borders are closed using the `close-borders` function.
- The `not-live` function handles deaths among infected agents.
- Recovery is managed via the `cure` function.
- Finally, the simulation advances to the next tick, and the loop continues.

```
to go
  ;; Stop if there are no infected (red) actors
  if not any? actors with [color = red] [stop]

  ;; Model dynamics
  travel ;; Move actors around
  infect ;; Spread infection
  if count actors with [color = red] / count actors > close-when [
    close-borders ;; Close borders when infection threshold is met
  ]
  not-live ;; Handle deaths from infection
  cure ;; Handle recovery

  tick ;; Advance simulation time
end
```

Traveling

The `travel` function is straightforward yet effectively simulates the concept of movement. It selects a proportion of agents standing on black patches (indicating areas not blocked by borders) based on the `travel-rate`. If the `travel-rate` is set to 100, all agents will travel; if it is set to 1, only 1% of the agents will travel. The exact number of agents selected to travel depends on the `density` global parameter.

All selected traveling agents are temporarily moved to patch `(0, 0)`. At this location, the patch agent (the patch itself) is asked to instruct all actors on it to move to a random patch that is both unoccupied and has a black color. This ensures that agents do not move back to their original location, facilitating valid movement across the grid.

```
to travel
  ;; Move a proportion of actors based on travel-rate
  ask n-of (count actors-on patches with [pcolor = black] * travel-rate / 100)
  actors-on patches with [pcolor = black] [
    setxy 0 0
  ]
  ;; Move actors to random unoccupied black patches
  ask patch 0 0 [
    ask actors-here [
      if not any? patches with [not any? turtles-here and pcolor = black] [stop]
      move-to one-of patches with [not any? turtles-here and pcolor = black]
    ]
  ]
end
```

Infecting

The `infect` function is straightforward, with no complex dynamics. It instructs all infected actors (red-colored agents) to check their eight neighboring patches. For each neighboring patch, if an actor is present and not already infected, there is a chance, determined by the `infection-rate`, that the actor becomes infected. If the neighboring actor is already infected, no action is taken.

```
  to infect
  ;; Infected actors (red) spread infection to healthy neighbors
  ask actors with [color = red] [
    ask neighbors [
      ask actors-here with [color = white] [
        if random 100 <= infection-rate [set color red]
      ]
    ]
  ]
end
```

Closing Borders

The `close-borders` procedure operates only in the `communities and borders` scenario and when the borders are still open. It selects a specified number (`number-of-border`) of patches to serve as initial border closures, changing their color to yellow. The procedure then iteratively propagates the closure by expanding the yellow area to neighboring black patches, repeating this process `close-borders-law` times. This ensures that travel, which can only occur on black patches, is restricted. Finally, the `borders?` flag is set to indicate that the borders are now closed.

```
to close-borders
  if not borders? [
    if senario = "communities and borders" [
      ;; Mark random patches as closed borders
      ask n-of number-of-borders patches with [pcolor = black] [
        set pcolor yellow
      ]
      ;; Expand borders by specified steps
      repeat close-borders-law [
        ask patches with [pcolor = yellow] [
          ask neighbors with [pcolor = black] [set pcolor yellow]
        ]
      ]
      set borders? TRUE ;; Borders are now closed
    ]
  ]
end
```

Not Living

The `not-live` function, which handles the process of dying, is also relatively simple in this simulation. Infected actors (red-colored agents) increment their `days` counter at every tick to track how long they have been infected. As the number of days increases, the probability of dying also increases. The chance of death reaches a maximum of 20%, simulating a higher mortality risk over time. This essentially means that each agent has the following probability of dying:

$$P(\text{not-live}_i) = \begin{cases} 0 & \text{if days\_infected}_i \leq 30 \\ \left(\frac{\text{days\_infected}_i - 30}{100} \right) \times \frac{1}{5} & \text{if days\_infected}_i > 30 \end{cases}$$

```
to not-live
  ;; Infected actors (red) age and may die over time
  ask actors with [color = red] [
    set days days + 1
    if random 100 <= days - 30 [
      if random 100 < 20 [die] ;; Chance of death increases with time
    ]
  ]
end
```

Cure

The cure process can occur for actors that are infected. The probability of an infected actor iii being cured can be represented by the following formula:

$$P(\text{cure}_i) = \left(\frac{\text{cure\_rate}}{100} \right) \times \frac{1}{2}$$

Exercise: Experiment with the hyperparameters of the model.

- Adjust the values for each hyperparameter (e.g., `population`, `infection-rate`, `travel-rate`, `cure-rate`, and others).
- Log the results of each simulation run, including the shape of the infection curve and the total number of deaths.
- Analyze how each hyperparameter influences the dynamics of the simulation, particularly the infection curve and mortality rate.

Exercise: Adjust travel and border rules to minimize deaths.

- Fix the infection-related parameters (e.g., `infection-rate`, `cure-rate`, etc.) to their default values.
- Modify the travel and border-related rules (e.g., `travel-rate`, `close-borders-law`, and `number-of-borders`).
- Run the simulation and log the results for each configuration, focusing on the total death count.
- Can you find a set of hyperparameters that minimizes the death count?

Exercise: Modify the infection dynamics by allowing agents to infect others within a variable radius.

- Add a slider `infection-radius` in the Interface tab to allow the user to control the radius of infection.
- Modify the `infect` procedure so that it checks all agents within the specified radius, not just neighbors.
- Run simulations with different values of `infection-radius` and observe how the infection spreads over time.

Solution:
▶ **Click to show/hide solution**

Exercise: Add a vaccination strategy to the model.

- Create a slider `vaccination-rate` that determines the percentage of healthy actors vaccinated at the start.
- Vaccinated actors should be immune to infection and represented by a unique color (e.g., blue).
- Modify the `setup` procedure to randomly vaccinate a proportion of the population based on `vaccination-rate`.
- Analyze the effect of vaccination on the infection curve and total deaths.

Solution:
▶ **Click to show/hide solution**

Extra Exercise: Add `super-spreader` agents with higher infection rates.

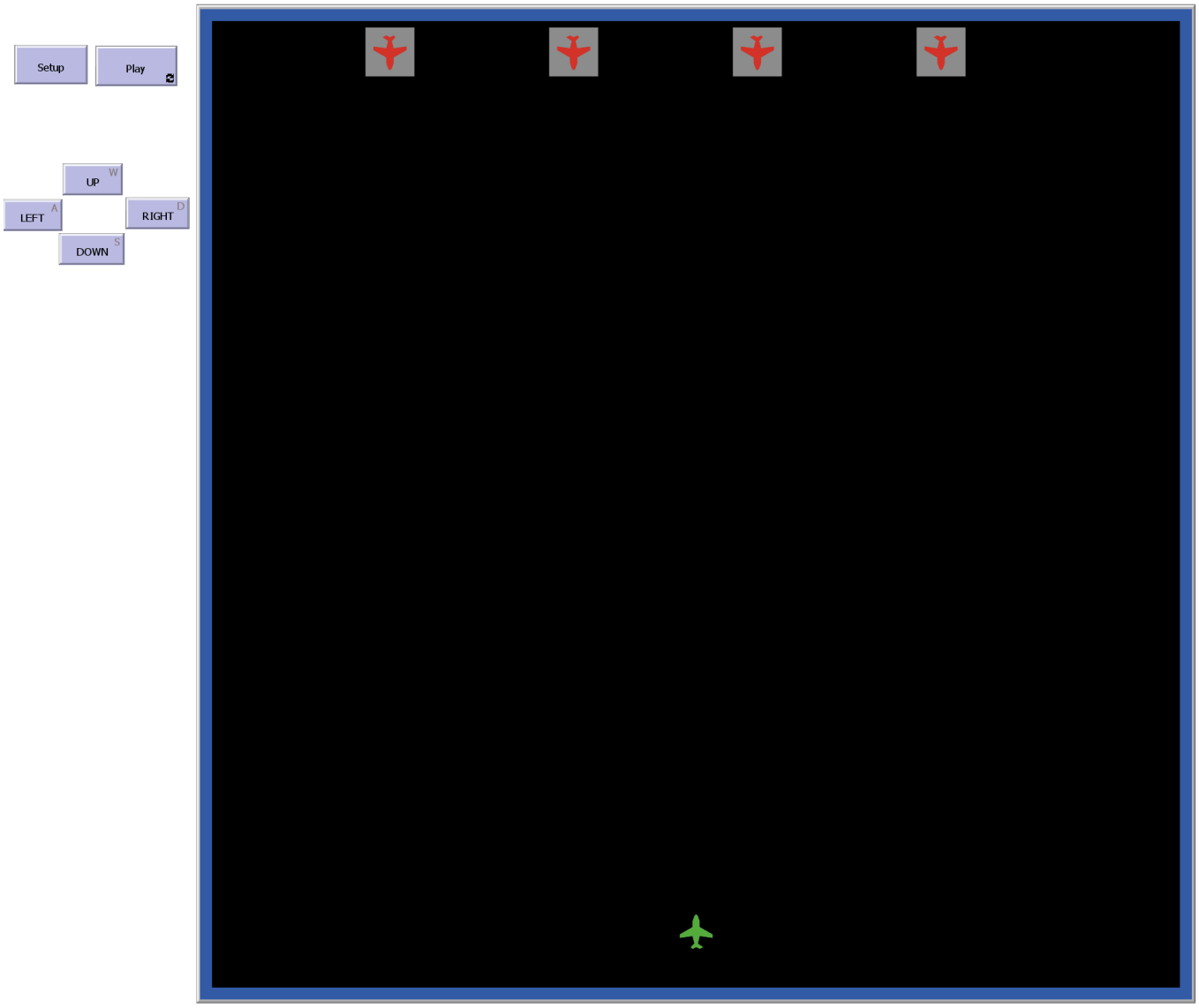
- Create a certain percentage of the population as `super-spreaders` at the start of the simulation.
- Super-spreaders should have a higher infection radius and infection rate compared to regular agents.
- Modify the `setup` and `infect` procedures to account for super-spreaders.

Game Development?

The implementation and development of this project were carried out by Ramesh Maddegoda in June 2021.

Can you create simple games in NetLogo? **Absolutely**! By leveraging the built-in tools and features, you can design simulations that use keyboard inputs such as `W`, `A`, `S`, and `D` for movement, as well as button clicks to trigger specific events within the environment. NetLogo also supports **other mouse and keyboard inputs**, allowing for a wide range of interactive possibilities.

Let's explore what creative games can be built in NetLogo!



Let's just quickly go through what makes this game being a thing possible. Let's look at some global variables, agent attributes and breeds.

```
breed [ players player ]
breed [ player-bullets player-bullet ]
breed [ enemy-bullets enemy-bullet ]
breed [ landing-zones landing-zone ]
breed [ enemies enemy ]
breed [ explosions explosion ]
breed [ final-statuses final-status ]

; Global variables
globals [
  mouse-was-down
  stop-game
]

; Private varaiabled
players-own [
 health
]
```

```
enemies-own [
 health
]
```

There are several breeds utilized in this code, each serving a distinct and specific purpose. Let's break them down:

- **Player**: Represents the main agent controlled by you.
- **Player-bullets**: These are the bullet agents fired by the player.
- **Enemy-bullets**: Bullets fired by enemy agents targeting the player.
- **Landing-zones**: Designated areas where enemy agents spawn.
- **Enemy**: Represents the hostile agents that the player must defeat.
- **Explosion**: Simulates explosions using orange agents that scatter outward from the point of impact.
- **Final-statuses**: Used to display emojis or symbols on the screen at the end of the game by coloring specific patches.

The `setup` command is straightforward but crucial for initializing the game. It creates the player agent and positions it at the bottom of the grid. The player is assigned a plane-shaped appearance, customized color, heading, and a health attribute, which is shared by both `players` and `enemies` as agent-specific attributes. Additionally, it sets up the landing zones where enemy agents will spawn and creates the enemy agents with similar properties to the player. Finally, the setup command draws the blue borders around the grid to define the play area visually.

```
to setup

 clear-all
 reset-ticks

 set stop-game false

 setup-players
 setup-landing-zones
 setup-enemies

 set mouse-was-down false

  ask patches with [ count neighbors != 8 ]
    [ set pcolor blue ]
end
```

```
to setup-players
    create-players 1
     ask players [
        set shape "Airplane"
       set color green
       set size 3
       setxy 40 5
       set heading 0
       set health 100
     ]
end
```

```
to setup-enemies
  ask enemies [
    set shape "Airplane"
    set size 3
    set color red
    set heading 180
    set health 100
  ]
end
```

```
to setup-landing-zones
  let x 0
  create-landing-zones 4
  ask landing-zones [
    set shape "square"
```

```
    set size 5
    set color grey
    set x ( x + 15 )
    setxy x  (max-pycor - 3)
    hatch-enemies 1 [
      create-link-from myself [
        set color black
      ]
    ]
  ]
end
```

The `stop-game` command is responsible for managing the game's state, determining when the game should end based on specific conditions. The `mouse-was-down` command tracks the status of mouse clicks, enabling interactions or triggering events within the simulation based on user input.

The Play Loop

The `play` loop is designed to run indefinitely but can be terminated when the `stop-game` global variable is set to `true`. This variable is triggered when either all enemy planes' health reaches 0 or the player's health is depleted.

```
to play
    tick
    if stop-game = true [
        stop
    ]

    player-rules
    player-bullet-rules
    enemy-bullet-rules
    enemy-rules
    explosion-rules
    check-mouse-button
end
```

The `player-rules` command is straightforward: it checks if the player's health has reached 0. If so, it triggers the `game-over` command, which creates the emoji effect and sets the `stop-game` variable to `true`, effectively ending the game.

```
to player-rules
  ask players [
   if health <= 0 [
     game-over
   ]
   set label round(health)
   facexy mouse-xcor mouse-ycor
 ]
end
```

```
to game-over
  hatch-final-statuses 1 [
     setxy 40 40
     set shape "face sad"
     set size 15
     set label ""
     set color yellow
  ]
  set stop-game true
end
```

The `enemy-rules` command follows a similar structure to `player-rules` but includes additional logic for enemy behavior. It checks whether an enemy's health has dropped to 0, and if so, triggers the `explode` command and removes the agent. If an enemy is far from the player (beyond a distance of 50), it patrols the environment in a predefined motion. However, if it comes within 50 units of the player,

it begins to approach and fire bullets. The explosion effect for enemies is also handled within this command. Let's take a closer look at the `explode` command.

```
to explode
  hatch-explosions 25 [
    set shape "Default"
    set color orange
    set size 2
    set heading random 360
    set label ""
  ]
  sound:play-note "Gunshot" 0 64 2
end
```

The `explode` command simply hatches turtles of the `explosion` breed, creating the visual representation of the explosion. However, it does not handle their movement. The movement and behavior of these explosion agents are managed by the `explosion-rules` command, which is executed in the `play` loop.

```
to explosion-rules
    ask explosions [
    fd 0.01
    if [pcolor] of patch-here = blue [
      die
    ]
  ]
end
```

The `x-bullet-rules` command controls the bullet behavior in the model. It asks all bullet agents to move forward by one unit. The bullets are removed under specific conditions:

- if they reach the borders of the grid
- if they enter the landing zones
- or if they collide with enemy bullets within a radius of 3.

Additionally, if a bullet hits an enemy within a radius of 3, it reduces the enemy's health by 0.01. When bullets collide or hit their targets, they trigger the `explode` command, which hatches `explosion` agents to create a visual effect, similar to the explosions caused by enemy deaths.

```
to bullet-explode
  hatch-explosions 3 [
    set shape "Default"
    set color grey
    set size 1
    set heading random 360
    set label ""
  ]
  sound:play-note "Gunshot" 50 64 2
end
```

The `check-mouse-button` command is another key part of the model. It monitors whether the mouse button was pressed during the previous tick. If it detects that the mouse was clicked, it hatches a bullet from the player's position.

Movement

Lastly, the `go-up` command handles player movement, allowing the player to move upward when the `W` key is pressed.

```
to go-up
  ask players [
    set heading 0
    if ycor < max-pycor [
      set ycor (ycor + 1)
    ]
```

```
  ]
end
```

The player's vertical position (`ycor`) stays within the defined bounds, preventing the player from exceeding the grid's upper limit.

Extra Exercise: Add Power-ups

- Create a new `power-up` breed that spawns randomly on the grid.
- Make power-ups provide benefits to players, such as increased speed, additional health, or temporary immunity.
- Ensure power-ups disappear after a certain time or after being collected by the player.

Extra Exercise: Add Obstacles

- Create stationary or moving obstacles that block bullets and player movement.
- Ensure obstacles are randomly generated during the `setup` procedure of the game.

Extra Exercise: Multiplayer Support

- Add support for multiple players controlled by different keys (e.g., `WASD` for Player 1 and `Arrow Keys` for Player 2).
- Implement cooperative gameplay where players work together to defeat enemies.

Extra Exercise: Add difficulty levels:

- Implement a difficulty scaling system where the number and health of enemies increase as the player progresses.
- Optionally, increase the speed or frequency of enemy bullets over time.

Tamás Takács

January 25, 2025
