

Simulation – Model, View, Controller

15-110 – Friday 04/10

Learning Goals

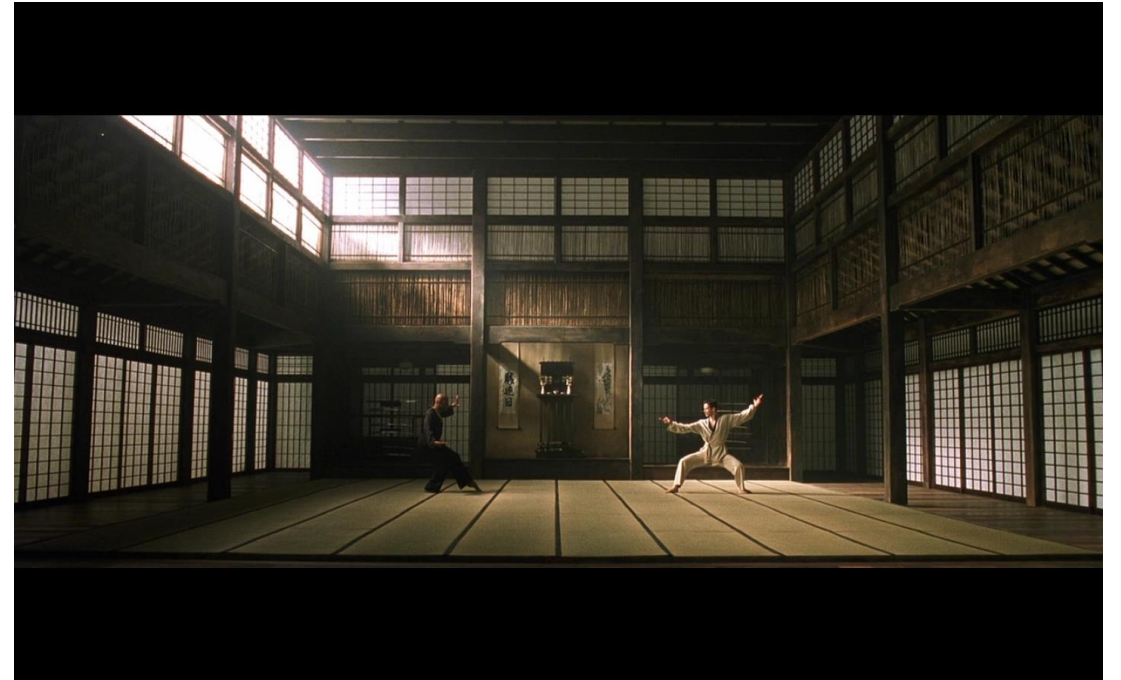
- Represent the state of a system in a **model** by identifying **components** and **rules**
- **Visualize** a model using graphics
- Update a model over **time** based on **rules**
- Update a model based on **events** (mouse-based and keyboard-based)

Simulations and Models

Simulations are Imitations of Real Life

A **simulation** is an automated imitation of a real-world event.

By running simulations on different starting inputs, and by interacting with them while they run, we can test how the event will change under different circumstances.



Simulation and COVID-19

Simulation has been used in the COVID-19 pandemic to demonstrate what 'flattening the curve' means, and how different policies will result in different infection and fatality rates.

This has a distinct effect on the choices governments make about which policies to implement.

An early simulation by The Washington Post helped many people understand the need for social distancing:

<https://www.washingtonpost.com/graphics/2020/world/corona-simulator/>

Free-for-all



Attempted quarantine



Moderate distancing

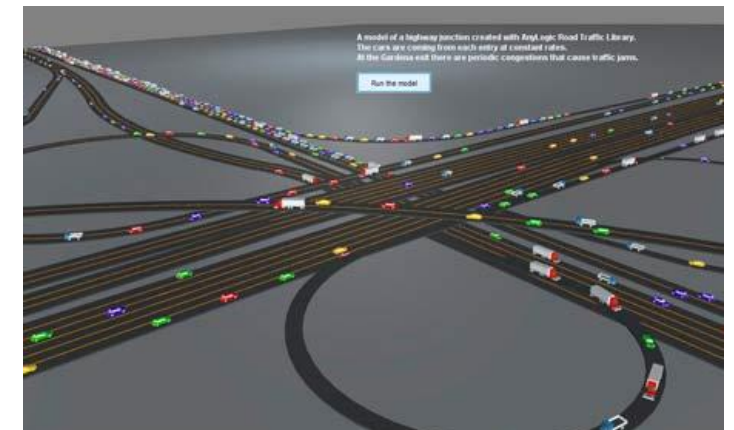
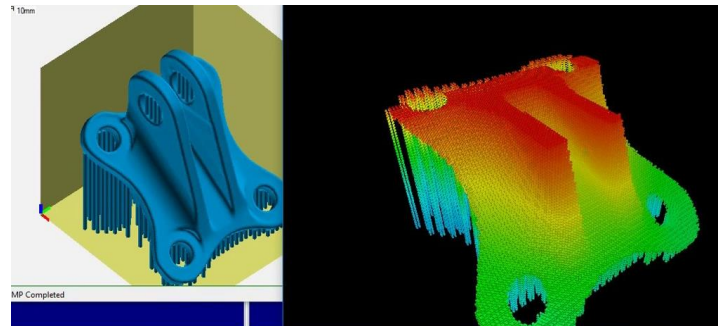
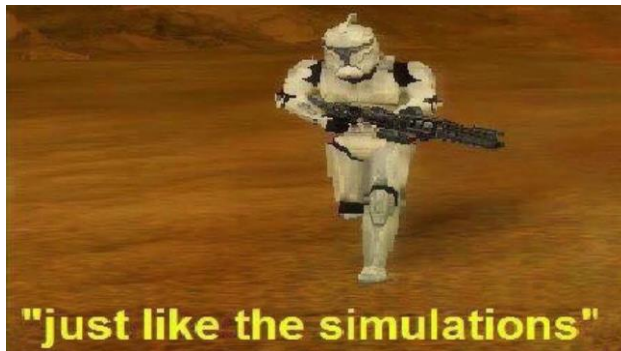
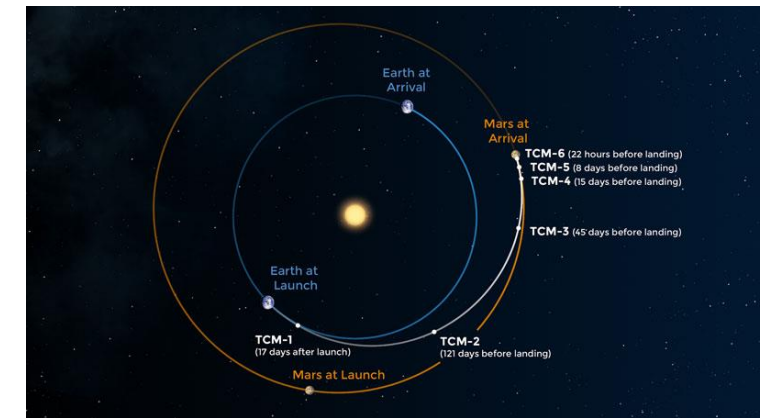
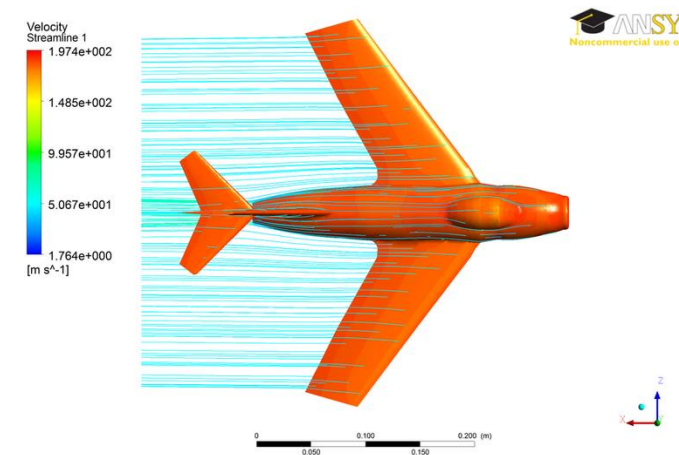


Extensive distancing



Examples of Simulations

Simulation is used across many different fields, including training people, testing designs, and predicting results.



Simulations vs. Real-world Experiments

Simulations share a lot in common with real world experiments. Major differences include:

- Experiments run in **real time**; simulations can be **sped up, slowed down, or paused**.
- Experiments can be **expensive**; simulations are fairly **cheap**.
- Experiments include **all possible factors**; simulations only include **factors we program in**.

Example Simulations

You can explore simulations across a variety of fields on the site NetLogo.

- [Ant colony movements](#)
- [Flocking behavior](#)
- [Gravitational forces](#)
- [Climate change](#)
- [Fire spreading](#)
- [Rumor mills](#)

Simulations Run on Models

How do we program a simulation? You need to design a good **model**, which will mimic the part of the real world you want to study. The simulation represents how the system represented by the model changes **over time**, or how it changes **based on events**.

Models are composed of two parts:

- The **components** of the system (information that describes the world at an exact moment).
- The **rules** of the system (how the components change as time passes).

Components are like variables, and rules are like functions!

Example Model

Problem: how will increasing the price of bread over the course of a few months affect how many people buy bread?

Model Components: current price; delta change in price; overall consumer count; distribution of consumer incomes

Model Rules: supply/demand relationship for bread; relationship between income and max amount willing to pay

Activity: Design a Model

Problem: say we want to track how many birds are in a local area over time.

What are the components of this model? What are the rules?

Coding a Simulation

Simulation Parts in Code

We'll implement simulations in this class **graphically**, like in NetLogo. We'll use Tkinter again to do this!

Our simulation code will be composed of three parts:

- Making the **initial components**, by storing the starting component values in a shared data structure
- Implementing **time and event controllers**, which run the model's rules to update the components when called
- Graphically drawing a **view**, which will repeatedly display the current state of the components

Making the Components

We'll represent our components in code in a **dictionary** called `data`. The keys will take the place of variable names, while the value will be the actual component values.

For example, to store the price of bread, we could set `data["price"] = 5.00`.

By storing all of the components in one structure, we can pass the same structure around to all the functions we write, using **aliasing**. This will let us update data in one function, then display the updated components in another.

Displaying the Model

To display the whole model, we'll use Tkinter to draw graphics that represent the components visually. By referring to values in `data` in the view function, we can make graphics based on pre-defined components.

We'll erase and re-draw the graphics window every time the rules of the simulation run. By changing the components a little bit at a time, this makes the display appear to update smoothly.

Running the Rules

We can run the simulation rules in two ways: either **over a period of time**, or **when events happen**. We'll address the time controller first, then the event controller later.

The **time controller** will create a **time loop** and call a function (we'll call it `runRules`) within that time loop at equal time intervals. By calling this function continuously, we can simulate time passing.

In the actual rules function, we'll update the values in **data**, to change them over time.

Simulation Functions

To implement the three simulation parts, we'll use a new **simulation framework** that you can find linked on the course website. In this framework, you can update three functions that correspond to the three parts:

- `makeModel(data)` makes the original components. `data` is the model dictionary
- `runRules(data, call)` runs the rules to update `data`. The integer `call` represents the number of times `runRules` has been called
- `makeView(data, canvas)` displays the model. `canvas` is a Tkinter canvas

Simple Example – Color-Changing Ball

Let's start with a simple simulation. Say we want to draw a circle and have the color of the circle change over time.

The **model** should hold any component values that might change. In this case, that's the **color** of the circle.

The **rules** should describe how the model changes over time. In this case, we **change the color** every call to `runRules()`.

The **view** should draw a circle in the middle of the window, and set its color based on the color in the model.

Simple Example Code

```
def makeModel(data):  
    # put variables in data here  
    data["color"] = "red"  
  
def makeView(data, canvas):  
    # (200, 200) is center point  
    canvas.create_oval(200 - 50, 200 - 50, 200 + 50, 200 + 50,  
                       fill=data["color"])  
  
def runRules(data, call):  
    if data["color"] == "red":  
        data["color"] = "green"           # switch from red to green  
    elif data["color"] == "green":  
        data["color"] = "blue"           # switch from green to blue  
    else:  
        data["color"] = "red"           # switch from blue to red
```

Interaction Events

The second kind of rules are ones that run when an **event** occurs.

An **event** represents a single user interaction with the computer system. Events come in many forms: **keyboard presses**, **mouse clicks**, touchpad gestures, touchscreen presses, button presses, etc...

When you trigger an event on your computer, a **signal** is sent from the computer hardware to any programs that are currently running. That signal has information about the type of the event (key press vs. mouse click), plus any additional information that might be useful (which key was pressed).

Event Loop

Similar to the time loop that we used before, we'll need to run an **event loop** to capture the signals that the computer sends out. However, events occur **irregularly**, unlike regularly-timed rules.

To implement this event loop, we'll have our simulation system constantly **listen** for events. When an event occurs, the simulation system will catch it, then send it on to a function we write specifically to handle that kind of event. This is done with a special kind of Tkinter function called **bind**, and is provided in the starter code.

Tkinter Events

With Tkinter, we can listen for and bind functions to lots of different event types.

We'll care about just two: `<Key>`, a keyboard press, and `<Button-1>`, a mouse click.

There are lots of other Tkinter events we can implement if we want them:

<https://effbot.org/tkinterbook/tkinter-events-and-bindings.htm#events>

Event Handlers

To deal with Key and Mouse events, we'll introduce two new functions to our simulation framework:

- `keyPressed(data, event)`
- `mousePressed(data, event)`

Each of these takes `data` (our components data structure), and `event`, an **event object**, which contains the information about the event.

These work like `runRules(data, call)` – we update `data`, then refresh the view immediately afterwards. This lets us make changes to the model.

keyPressed Events

In `keyPressed`, the `event` parameter contains two values we can use:

- `event.char` is a string containing the character pressed
- `event.keysym` is a string holding the 'name' of the character, for characters we can't type (e.g., Enter or Backspace)

If we want to draw the last-pressed character in the middle of the screen, for example, we would need to store that character:

```
def keyPressed(data, event):  
    data["text"] = event.char
```


Example Key Event

```
def makeModel(data):  
    data["color"] = "red"  
    data["tmp"] = ""  
  
def makeView(data, canvas):  
    canvas.create_oval(200 - 50, 200 - 50, 200 + 50, 200 + 50,  
                      fill=data["color"])  
  
def keyPressed(data, event):  
    # build up a color string one char at a time until user presses Return  
    if event.keysym != "Return":  
        data["tmp"] += event.char  
    else:  
        # move the color into data["color"]  
        data["color"] = data["tmp"]  
        data["tmp"] = ""
```

mousePressed Events

In `mousePressed`, the `event` parameter holds pixel location where the user clicked on the canvas.

- `event.x` is the x location
- `event.y` is the y location

If we want to move a circle around the canvas to be centered wherever you click, we'd need to store the center location:

```
def mousePressed(data, event):  
    data["cx"] = event.x  
    data["cy"] = event.y
```

Example Mouse Event

```
def makeModel(data):
    data["cx"] = 200
    data["cy"] = 200
    data["size"] = 50
    data["colors"] = [ "red", "orange", "yellow", "green", "blue", "purple" ]
    data["colorIndex"] = 0

def makeView(data, canvas):
    i = data["colorIndex"]
    color = data["colors"][i]
    canvas.create_oval(data["cx"] - data["size"], data["cy"] - data["size"],
                      data["cx"] + data["size"], data["cy"] + data["size"],
                      fill=color)

def mousePressed(data, event):
    # Check if the distance between clicked point and center of circle is in circle
    if ((event.x - data["cx"])**2 + (event.y - data["cy"])**2)**0.5 <= data["size"]:
        data["colorIndex"] = (data["colorIndex"] + 1) % len(data["colors"])
```

Supporting Functions – Time Loop

The starter code we provide helps the simulation run smoothly. You don't need to understand this code, but here's some more info if you're interested.

The **time loop** works by using the built-in function `canvas.after`. This function lets us repeatedly call the same function (like recursion), but pauses before making the call. That lets us recurse/loop infinitely while not freezing the window.

The function `runSimulation(width, height, timeRate)` actually sets up this time loop. You can speed up/slow down the simulation by changing `timeRate`.

You can also change the window size by changing `width` and `height` in those parameters.

Learning Goals

- Represent the state of a system in a **model** by identifying **components** and **rules**
- **Visualize** a model using graphics
- Update a model over **time** based on **rules**
- Update a model based on **events** (mouse-based and keyboard-based)