#5-1: Simulation

CS SCHOLARS – PROGRAMMING

Hw4 Overview

Week4 was a bit overpacked – sorry about that!

Can spend Wednesday finishing missing problems (`getCharacterLines`, `drawPixelArt`, `gradebookSummary`), then do Hw5 on Friday/Monday/Wednesday.

Quick review of Hw4 Written – Parsing Data.

Hw5 Overview

- A few core problems (three written, one programming)
  - **Recommendation:** try to complete and submit these by **this Friday**. We'll grade & give feedback on them over the weekend.
  - Can resubmit until following Wednesday though

- One larger project: Tetris or a self-designed project
  - Complete and submit by **next Wednesday**
  - Submissions will remain open until following Friday, but Wednesday is strongly preferred
  - If doing the self-designed project, contact Prof. Kelly by **this Friday**

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 after **events** (mouse-based and keyboard-based) based on **rules**

Simulations and Models

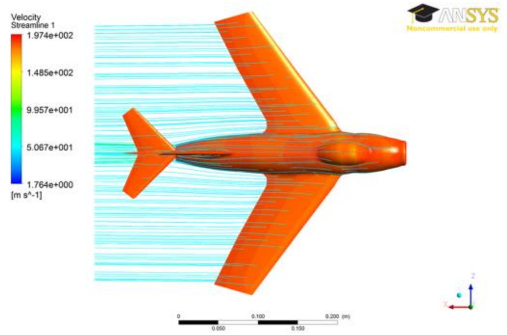
Simulations are Imitations of Real Life

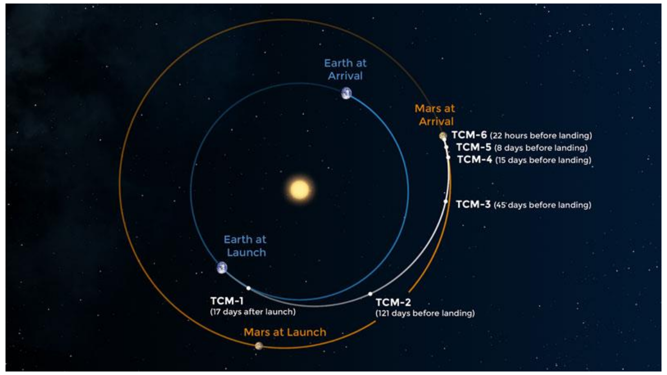
A **simulation** is an automated imitation of a real-world event.

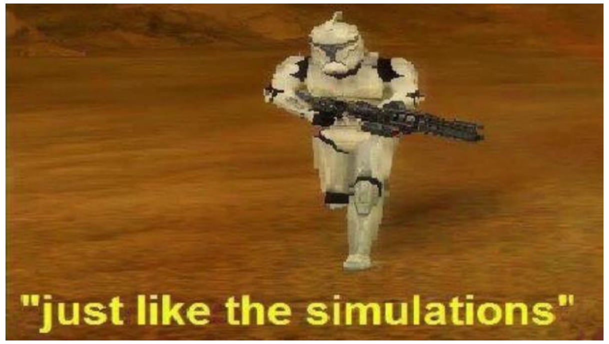
By running simulations on different starting inputs, and by interacting with them while they run, we can test how the event will change under different circumstances.

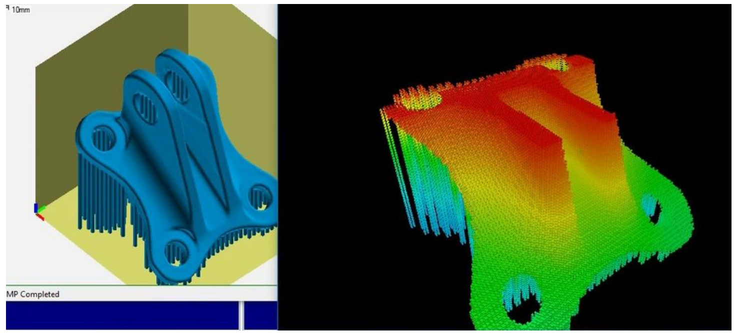
Examples of Simulations

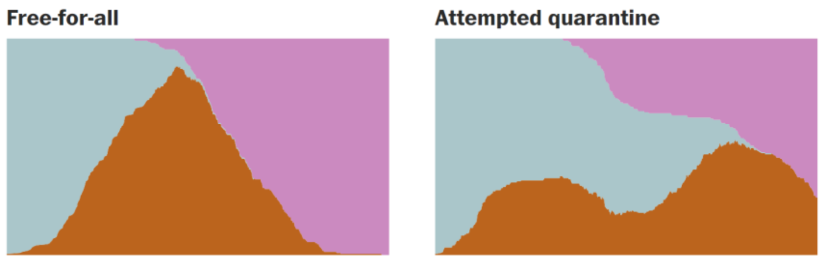
Simulation is used across many different fields, including training people, testing designs, and making predictions (like whether a flight plan will work, or how a pandemic will evolve over time).




Earth at Arrival
Mars at Arrival
TCM-6 (22 hours before landing)
TCM-5 (8 days before landing)
TCM-4 (15 days before landing)
TCM-3 (45 days before landing)
Earth at Launch
TCM-1 (17 days after launch)
TCM-2 (121 days before landing)
Mars at Launch







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 should change as time passes/events occur).

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.

You do: What are the components of this model? What are the rules?

Coding a Simulation

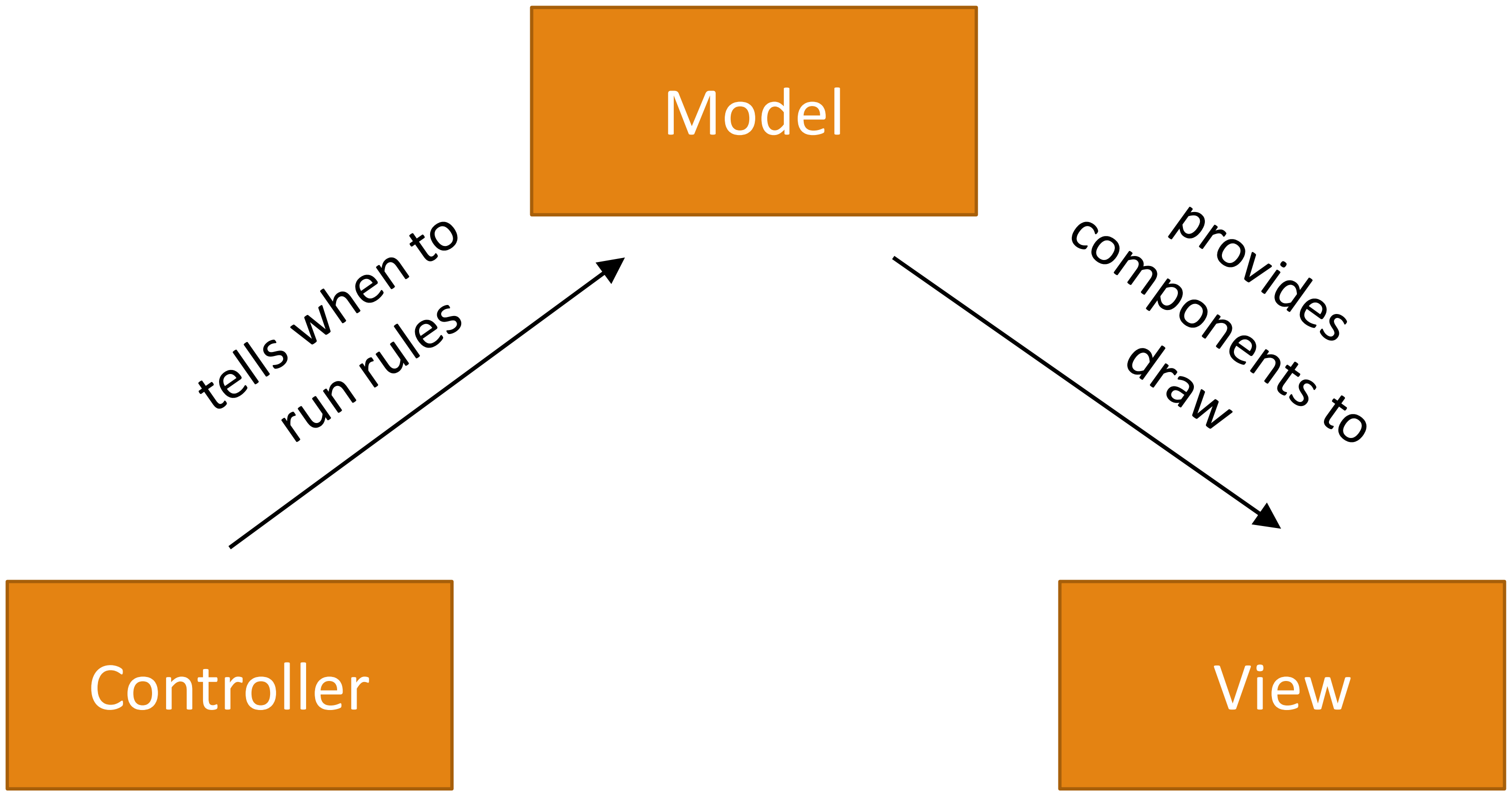
Simulation Parts in Code

We'll implement simulations in this class **graphically**, like in NetLogo, using Tkinter.

Our simulation code will be composed of three parts:

- A **model** which stores the core components in a shared data structure and implements core rules in functions
- Time and event **controllers** which tell the model when to run rules that update the components
- A graphical **view** which repeatedly displays the current state of the model

Model, View, Controller



Making the Components

We need to be able to pass the whole model around the code as a single variable. We'll do this by creating an **object** called `data` and adding components to that object.

These components will act just like variables; the only difference is that we'll use `data.componentName` instead of `componentName` by itself. It's similar to when we import a library or call a method on a list. For example, to store information about a circle that represents some part of the model, we could set:

```
data.x = 200
data.y = 200
data.r = 50
```

By storing all 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 components in a rule function, then display the updated data in a view function.

Note: you can do a lot more than this with objects! To learn more, check out the Advanced Programming slides from week2.

Displaying the Model

To display the whole model, we'll use Tkinter to draw graphics that represent the components visually. By referring to component values in `data` in the view function, we can make graphics that change alongside the model.

For example, if `data.x = 200`, `data.y = 200`, and `data.r = 50`, we could draw a circle with:

```
canvas.create_oval(data.x – data.r, data.y – data.r,
                   data.x + data.r, data.y + data.r)
```

We'll erase and re-draw the graphics window every time the rules of the simulation run. If we change 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** (or both!). We'll address the time controller first, then the event controller later.

The **time controller** will create a **time loop** and call a function that implements the model's rules within that time loop at equal time intervals. By calling this function continuously, we can simulate time passing.

If the model's rules change the model's components in `data`, this will simulate the model changing over time!

```
data.x = data.x + 5 # move the circle to the right
```

Simulation Functions

We'll use a new **simulation framework** that you can find linked on the course website to support our simulations. This framework manages the controllers for you; you just need to focus on implementing the model and the view. To do this, update three functions to build a simple simulation:

- `init(data)` makes the original components. `data` is the model object
- `timerFired(data)` runs the rules to update `data`.
- `redrawAll(canvas, data)` displays the model. `canvas` is a Tkinter canvas

This is different from the code we're used to because the functions **work together** instead of running in a sequential order.

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 **components** should hold any values that might change. In this case, that's the **color** of the circle. Set an initial component value in `init`.

The **rules** should describe how the model changes over time. In this case, we **change the color** in the shared data model with every call to `timerFired`.

The **view** should draw a circle in the middle of the window and set its color based on the color in the model. This is done in `redrawAll`.

Simple Example Code

```
def init(data):
    # put variables in data here
    data.color = "red"

def redrawAll(canvas, data):
    # (200, 200) is center point
    # make sure to reference data for the parts that change!
    canvas.create_oval(200 - 50, 200 - 50, 200 + 50, 200 + 50,
                       fill=data.color)

def timerFired(data):
    import random
    # Let's pick a color randomly!
    newColor = random.choice(["red", "orange", "yellow",
                              "green", "blue", "purple"])
    data.color = newColor # update data to change the model
```

Activity: Make the circle grow

You do: open the simulation starter code and copy in the functions from the previous slide. Run the code to make sure it works, then modify the code in the three functions so that the circle **grows larger** as time passes.

Hint: you'll need to add one **component** to the model, the thing that is changing. You should change that component in `timerFired` and access it while drawing the circle in `redrawAll`.

Interaction Events

The second kind of controller is one that captures **events**.

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

When you take an action 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 Rules

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

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

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

These work like `timerFired(data)` – we update `data`, then the controller refreshes the view immediately afterwards. This lets us make visible changes to the model.

`keyPressed` Events

In `keyPressed`, the `event` parameter contains two values we can access with a `.` (like string or list methods and the `data` components):

- `event.char` is a string that holds the character pressed
- `event.keysym` is a string that holds the 'name' of the character, for characters we can't show in a string (e.g., Enter or BackSpace)

If we want to draw the last-pressed character in the middle of the screen, for example, we would store that character in `data`, then draw it in `redrawAll`:

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

Example Key Event

```
def init(data):
    data.color = "red"
    data.tmp = "" # need to hold partial strings

def redrawAll(canvas, data):
    canvas.create_oval(200 - 50, 200 - 50, 200 + 50, 200 + 50,
                       fill=data.color)

def keyPressed(event, data):
     # 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 = ""
```

Activity: move circle up/down

You do: take the simulation code from the last activity (growing circle) and update it so that the circle moves **up** when the user presses the up key and **down** when the user presses the down key.

Note: you should use `event.keysym`. You'll be able to check it against `"Up"` and `"Down"`.

`mousePressed` Events

In `mousePressed`, the `event` parameter holds the 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 and draw the circle based on the model location in `redrawAll`:

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

Example Mouse Event

```
def init(data):
    data.color = "red"

def redrawAll(canvas, data):
    canvas.create_oval(200 - 50, 200 - 50, 200 + 50, 200 + 50,
                       fill=data.color)

def mousePressed(event, data):
    import random
    newColor = random.choice(["red", "orange", "yellow",
                              "green", "blue", "purple"])
    # Check if the user clicked inside the circle
    # Is the distance between the center and the click less than the radius?
    if ((event.x - 200)**2 + (event.y - 200)**2)**0.5 <= 50:
        data.color = newColor
```

Activity: make circle shrink

You do: take your code from the previous activity and modify it so that the circle **shrinks** whenever the user clicks inside it.

You can start with the bounds check from the previous slide, but you'll need to change what happens in the conditional body!

Summary: Model, View, Controller

Throughout the process of building these simulations, we've structured code based on the **model, view, controller** framework.

Model: manages the components and rules of the thing we're simulating

View: displays the data in the model so that the user can look at it

Controller: manages time loops and events that provide changes to the model

Sidebar: Controller Functions – Time Loop

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

The **time** controller in the function `timeLoop` calls our function `timerFired`, then calls `redrawAll` to update the view. It simulates a time loop with the built-in function `canvas.after`. This function calls `timeLoop` again (like an infinite loop) but pauses before making the call. That lets us repeat infinitely without freezing the window.

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

You can also change the window size by changing `width` and `height` in the function call arguments.

Sidebar: Controller Functions – Event Loop

The event controller runs an **event loop** to capture the signals that the computer sends out, similar to the time loop discussed before. 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 controller will catch it and send the event data on to the correct rule function; then it will tell the view to update. This is done with a special kind of Tkinter function called `bind` and is provided in the starter code.

With Tkinter we can listen for and bind functions to lots of different event types. We'll care about just two: `<Key>`, a key press, and `<Button-1>`, a left mouse click. There are lots of other Tkinter events we can implement if we want them:

https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/event-types.html

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 after **events** (mouse-based and keyboard-based) based on **rules**