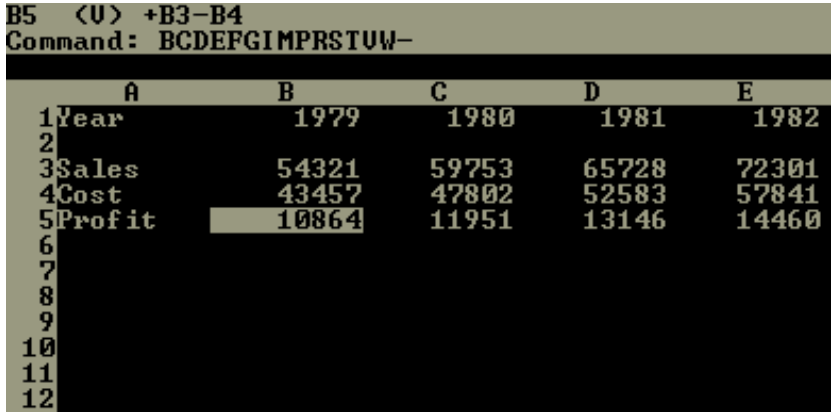
Bonus Slide

The First Killer App

In 1979, the application VisiCalc was produced by VisiCorp. This was a basic spreadsheet application that let the user modify values in a table and automatically re-calculate the results.

This was a huge development for business, as re-calculating tables by hand took teams of accountants long periods of time. The application became widely popular.

This incentivized many people to buy their own personal computers for the first time.



Unit 2: Modern CS Techniques

Simulation

15-110 – Monday 11/02

Announcements

- Project2 was due today
- Hw6 will be released soon, not due until 11/16
  - You get a week off to study for the exam
- Project1 revisions due **tomorrow noon**
- Quizlet7 on Wednesday
- **Exam2 next Wednesday, 11/11**
  - Same logistics as last time; practice materials available soon
- Reminder: sign up for Oral Assessment 2 - https://www.cs.cmu.edu/~110/oral/oral2.html

Final Project

The first problem on Hw6 involves ideating for the **final project.**

- Guide: https://www.cs.cmu.edu/~110/hw/final-project-guide.pdf

In this project you'll apply everything you've learned in 15-110 to design and implement a project of your choosing.

Ideation – due with Hw6 (Monday 11/16)

Plan – due with Hw7 (Monday 11/23)

Implementation & Reflection – due Friday 12/04 (last day of classes)

Discussion – complete between 11/30 – 12/14 as Oral Assessment 3

Learning Goals

- Represent the state of a simulated system in a **model** by identifying **components** and **rules**

- Read code that uses a **simulation framework** to update a model's components based on rules and visualize the model with graphics

- Use **Monte Carlo methods** to estimate the answer to a question

Simulations and Models

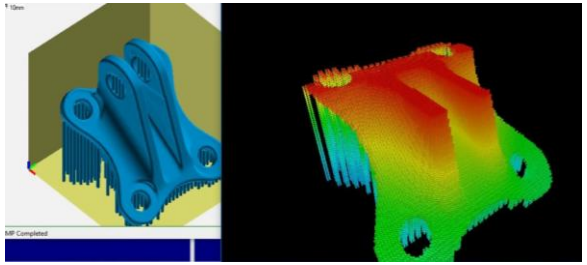
Simulations are Imitations of Real Life

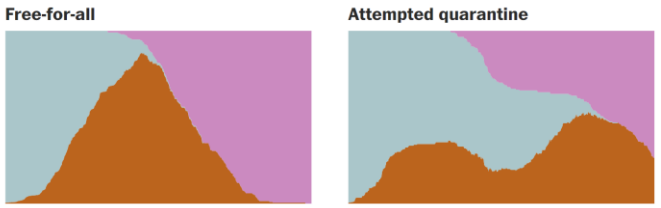
A **simulation** is an automated imitation of a real-world event.

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

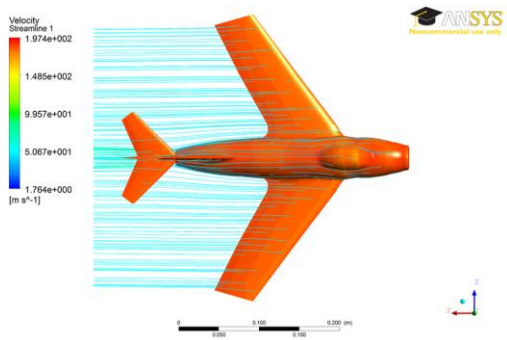
Simulation is used across many different fields, including training people, testing designs, and making predictions (like whether a flight plan works, or how a pandemic evolves).

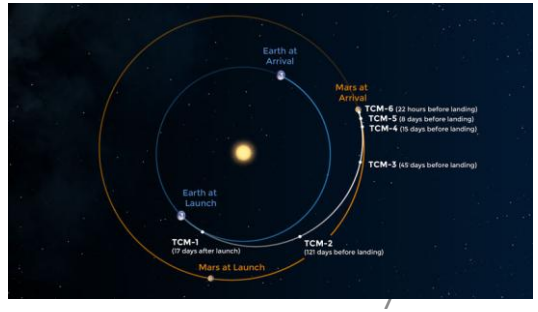












Simulations vs. Real-world Experiments

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, which is focused entirely on modeling and simulation.

- 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 showcases 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 much ice cream can an ice cream shop expect to sell on a given day?

Model Components: amount of ice cream sold; price; temperature; number of flavors available

Model Rules: some number of people will consider buying ice cream every day; when it is hotter outside, people buy more ice cream; when prices go up, demand goes down; an increased number of flavors increases the number of people who buy ice cream (to a certain point)

Activity: Design a Model

Problem: we want to track how many birds are in a local area over the course of a year, to see how the population changes.

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

Important: Simulations Rely on the Model!

Simulations are powerful, but they can also be suspect to error and bias, because **the results are influenced by what is included in the model**.

Example: you could build a fancy simulation of an amusement park to test different park configurations and estimate how much profit could be expected from each arrangement. But if your model doesn't include a variable for weather, the results may all be overly-optimistic.

Try investigating any simulations you might interact with to see what biases and errors they might include.

Coding a Simulation

Simulation Parts in Code

There are multiple ways to implement simulations!

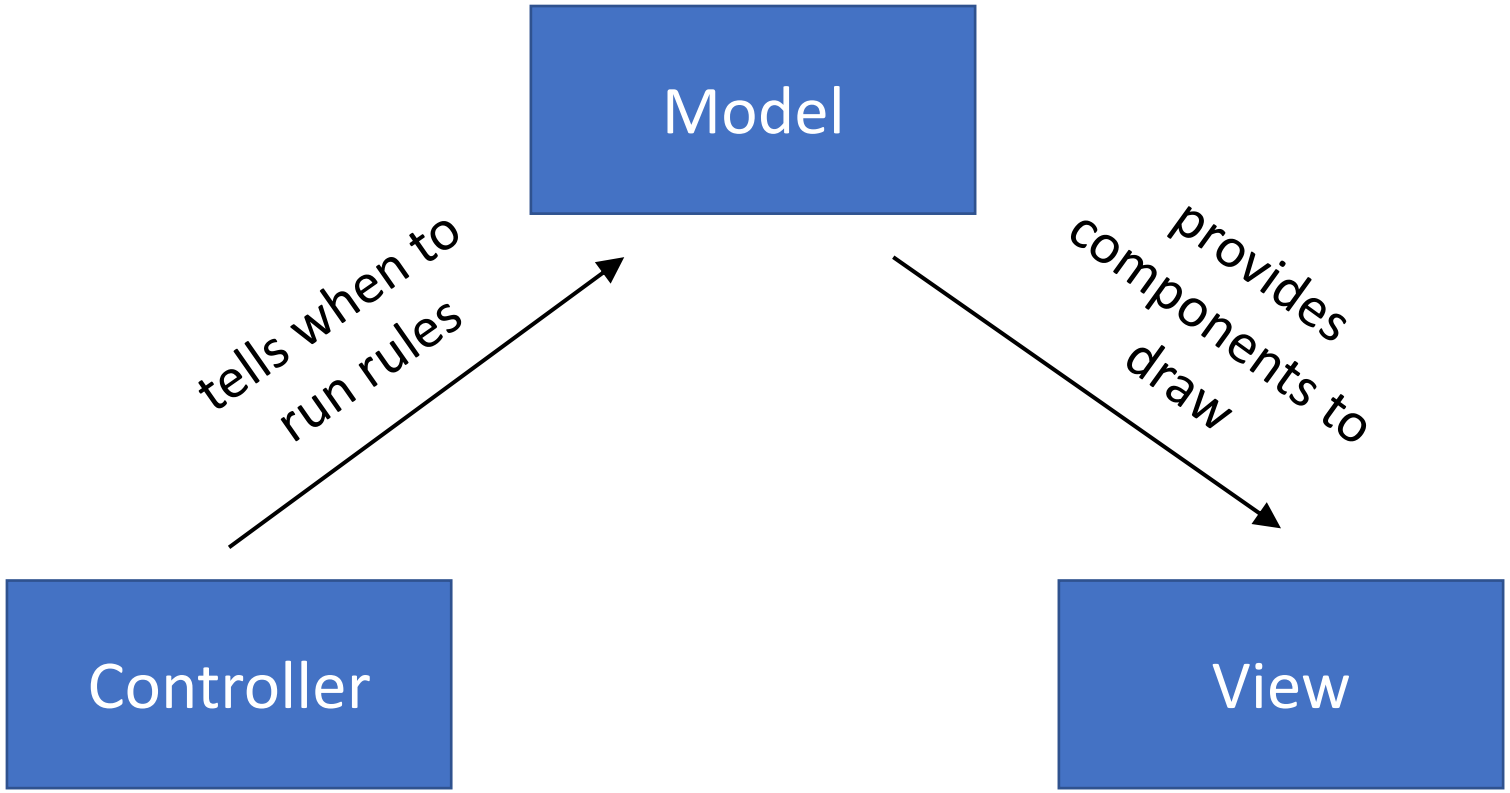
To provide more structure in interpreting the simulation code, we'll give you a **simulation framework** that implements simulations graphically, like in NetLogo, but using the graphics library `Tkinter`.

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

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



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

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

Simulation Functions

You can find the simulation framework linked on the course website. 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:

- `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 (how much time passed)
- `makeView(data, canvas)` 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.

Example with GenAI: Growing circle

Let's start with a simple simulation. Say we want to draw a circle and have the circle grow over time. Copy our starter code into GenAI and prompt it to create a growing circle with time.

The **components** should hold any values that might change. In this case, that's the **size** of the circle. We will set an initial component value in `makeModel`.

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

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

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 `runRules`, then calls `makeView` to update the view. It simulates a time loop with the built-in function `canvas.after`. This function calls `timeLoop` again (like recursion) but pauses before making the call. That lets us recurse infinitely without freezing the window.

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

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

You try: Ball sinking in water

In the previous activity, we created a simulation of a circle that grows in size with time. Now we want to simulate a sinking ball in water!

Work with a friend to prompt the GenAI to:

- Use the starter code to build a simulation of a ball sinking in water over time. Use physics laws of gravity and buoyancy.
- Every time the simulation is run, the ball changes size randomly.
- On the simulation, we want to be able to see the radius of the ball as well as the time it takes to sink.
- Make the ball purple :)

Interaction Events

In the previous slides, we learned about how to use controllers that tell the model to change over **time**. 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).

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 in the previous lecture. 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; that function 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://web.archive.org/web/20190512164300id_/http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/event-types.html

Event Rules

To deal with Key and Mouse events, we'll introduce two new rule 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** that contains the information about the event.

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

Monte Carlo Methods and Testing Simulations

Randomness in Simulation

Most simulations use randomness in some way; otherwise, every run of the simulation will produce the same result.

Using randomness in a simulation means that the same simulation might have multiple different outcomes on the same input model. A single run of a simulation is not a good estimate of the true average outcome.

To find the truth in the randomness, we need to use probability!

Law of Large Numbers

The Law of Large Numbers states that if you perform an experiment multiple times, the average of the results will approach the **expected value** of the true answer as the number of trials grows.

This law works for simulation as well! We can calculate the expected value of an event by simulating it a large number of times and averaging the results.

We call programs that repeat simulations this way **Monte Carlo methods**, after the famous gambling district in the French Riviera. We're gambling with the accuracy of the answer.

Monte Carlo Method Structure

If we put our simulation code in the function `runTrial` and want to find the odds that a simulation 'succeeds', a Monte Carlo method might take the following format:

```
def getExpectedValue(numTrials):
    count = 0
    for trial in range(numTrials):
        result = runTrial() # run a new simulation
        if result == True:  # check the result
            count = count + 1
    return count / numTrials # return the probability
```

Monte Carlo Example

Every year, SCS holds the Random Distance Run. The length of this race is determined by rolling two dice. **What is the expected number of laps a runner will need to complete?**

```
import random
def runTrial():
    return random.randint(1, 6) + random.randint(1, 6)

def getExpectedValue(numTrials):
    lapCount = 0
    for trial in range(numTrials):
        lapCount += runTrial()
    return lapCount / numTrials
```

Using Simulations

Once we've programmed a robust simulation, we can **change the starting state** to see how it changes the simulation. This is especially useful when we want to **predict** certain things about the world.

We can check predictions more quickly by making `timeRate` smaller (calling the simulation more often).

Let's use the sinking ball as an example of how to make predictions by using Monte Carlo methods with simulations.

Activity: Monte Carlo Methods

You do: What are the odds that the (random-sized) ball sinks within 0.5 seconds?

1. Use GenAI to help you write the code to run a single trial of a sinking ball.
2. Then, modify the code from the previous slide to calculate `getExpectedValue`.

Learning Goals

- Represent the state of a simulated system in a **model** by identifying **components** and **rules**

- Read code that uses a **simulation framework** to update a model's components based on rules and visualize the model with graphics

- Use **Monte Carlo methods** to estimate the answer to a question