Class 09

Simulation Architecture

Modeling Environments

Simulation Architecture

- At this moment in the semester, we are mostly done learning new C++.
  - Anything we learn will just be minor details building onto what we already know.
- We can now talk about how we should be structuring more capable simulations.
- We have written C++ programs with very basic structures thus far, but we can do much better.

Simulation Architecture

- Our goals are simple:
  - We want to *configure* our simulations without needing to recompile them
  - We want to encapsulate our simulations within a *model*

Configuring Simulations

- Configuring simulations is done by passing data into our simulations *when we execute the program*.
- Using *std::cin* however is extremely clunky; what if we need to supply a few dozen different parameters?
- We can use *configuration files*

Configuring Simulations

- A configuration file is a file that contains data that is processed by the simulation, and that data is used to set variables, parameters, etc..
- Because we are reading data from a file to set values within our program, we only need to change the configuration while when we want to set a variable to a different value.
- We will use a data format called JSON (JavaScript Object Notation) to define simulation parameters.
- We will use a tool called *nlohmann_json*
  - https://github.com/nlohmann/json

Configuring Simulations

- What goes into a configuration file?
  - Discrete, explicit values
- What does not go into a configuration file?
  - Dynamic values
  - Functions
  - Conditions/control constructures (if, for, while, etc.)

Configuring Simulations

```
{
    "number_of_entities": 2000,
    "simulation_end_time": 10.0,
    "delta_time": 0.1
}
```

- This configuration specifies that the simulation will have 2000 entities, will run for 10 seconds, and will step in increments of 0.1 seconds.

Encapsulation within a Model

- So far, other than defining a few classes to support our simulations, we have written the bulk of our programs in the main.cpp file
- The flow of our simulations is dictated by the code in the *main* function
- But what if do not have, or even want, a *main* function?

Encapsulation within a Model

- We can implement a class that we call *the model*
- This class will hold onto and manage the entire state of the simulation, as well as control the logical flow of the simulation.
- This is everything that our main function has been doing, but instead that code is tucked away into a class.
  - It will create our entities, environment, manage time, etc..
  - It handles everything
- Therefore, *assuming we even have a main function*, all it would do it create a model and interact with it.

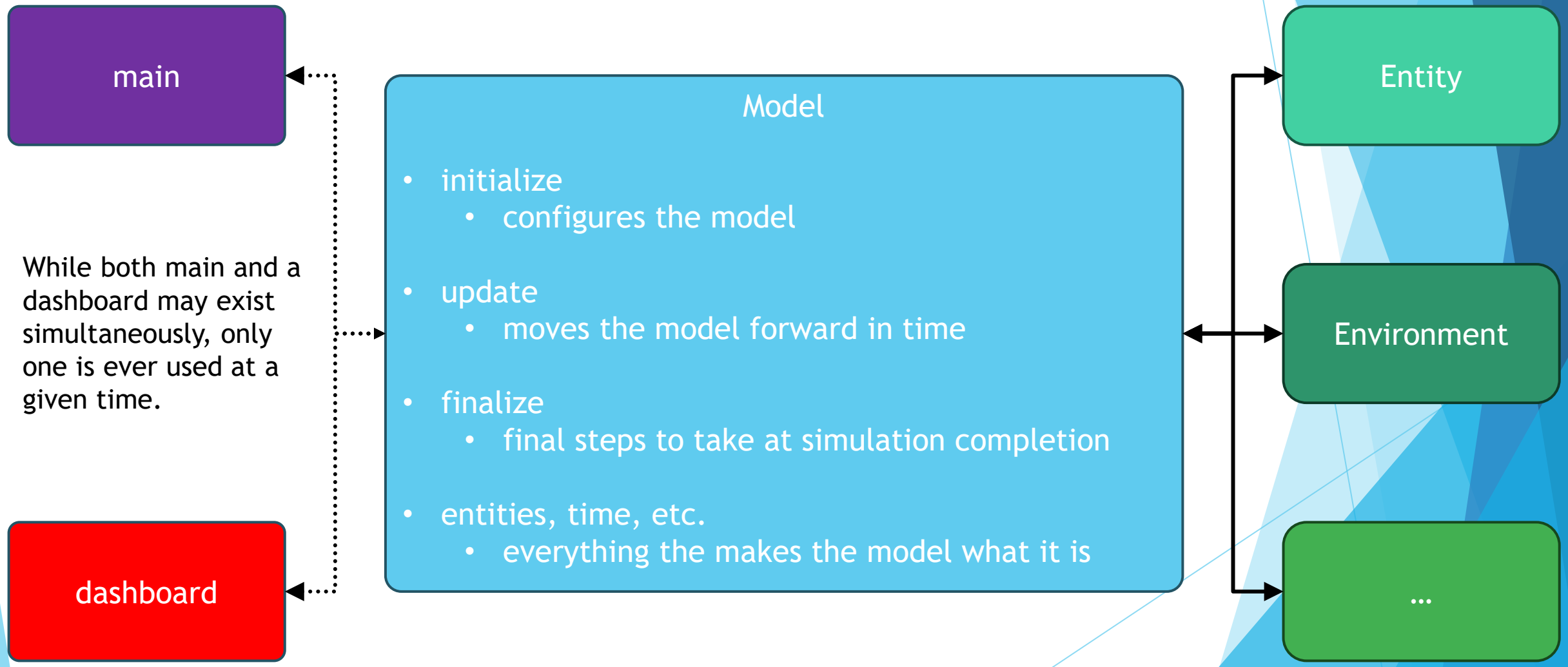
Encapsulation within a Model

- Why wouldn't we have a main function though? Well, it depends.
- ***We ultimately want to run our models and visualize them.***
- To do this we need to run the model and make sure it *outputs* some data. There are problems with this approach though:
  - What if it takes an obnoxiously long time to run the simulation to completion?
  - What if you do not want the simulation to explicitly end after some amount of time?
  - What if the simulation ends prematurely and the data is incomplete?
- These two problems yield two requirements: we want to visualize our data *as it is being produced* and we want to run our simulations *potentially without a defined end time*.

Encapsulation within a Model

- Our dashboards have suffered with the fact that it needs to run our simulation to completion, read the output, store that data in memory, and then display it one slice at a time.
- For adequately sized data this dashboard process is fast and performant (especially if the simulation is not faster-than-real-time (FTRT)), but we constantly are required to run the entire simulation to see even the beginning of it.
- For larger, more complex data we need to **transfer large amounts of data through memory,** which may not be feasible at all!

Simulation Architecture



Simulation Architecture

- **main** and **dashboard** are called consumers, or users, of the model.
- They are responsible for creating, initializing, updating, and finalizing the model.
- As the model updates, the consumer will pull data from the model and do something with it.
  - **main** will typically log the data
  - **dashboard** will typically visualize the data