3081W Project

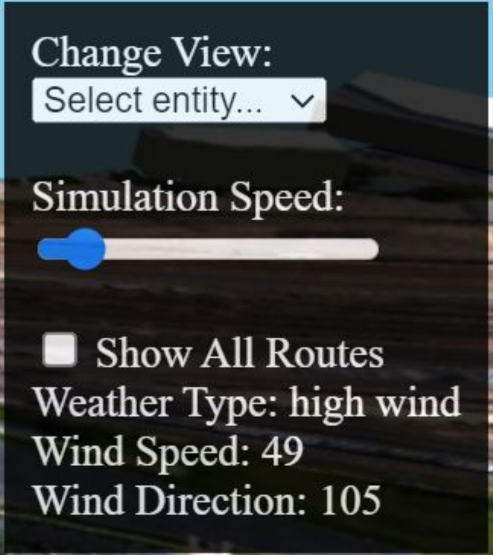
Tyler Pham, Rodrigo Alanis, Vincent Liu, Preston Zhu

Weather Feature

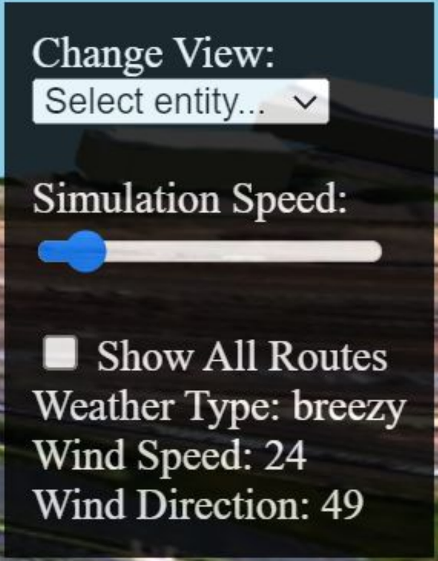
Random weather creates winds of varying speeds and direction that affect the movement of the entities in the simulation. The winds are separated into 4 different classes: sunny in which winds are low, breezy where winds are mild, high winds where some entities such as Drone are heavily affected but a Helicopter may still be able to move, and tornadoes in which all entities are unable to move.

Possible Weather

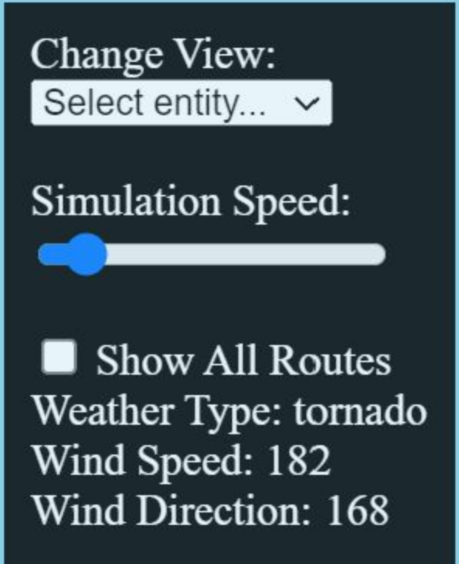
The current weather can be viewed in the menu section. Wind speeds range from 0 to 10 for sunny, 10 to 25 for breezy weather, 25 to 55 for high winds, and 100+ for tornadoes. This will be changed every 10 seconds



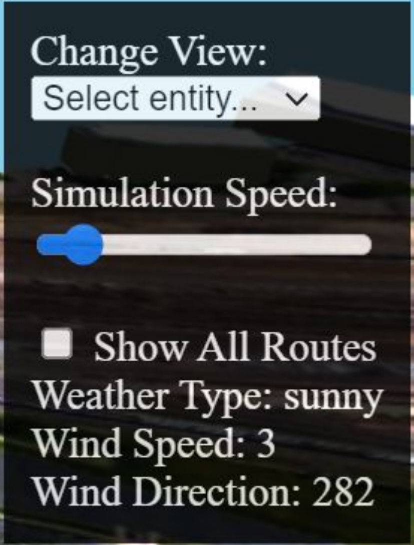
Weather Type: high wind
Wind Speed: 49
Wind Direction: 105



Weather Type: breezy
Wind Speed: 24
Wind Direction: 49



Weather Type: tornado
Wind Speed: 182
Wind Direction: 168



Weather Type: sunny
Wind Speed: 3
Wind Direction: 282

Why is this new feature important?

In real life, weather affects drones and helicopter movement. This feature makes the simulation more realistic, albeit with higher extreme weather chances. Rather than the drones and helicopters moving at fixed speeds, wind makes the simulation more varied and changes the movement speeds of the entities.

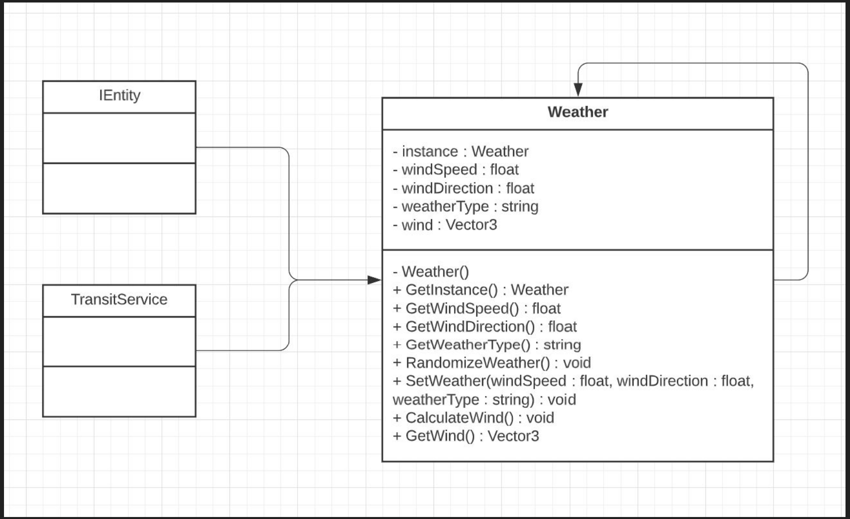
How does it add to the existing work?

The current simulation models a drone delivery system in ideal weather conditions (no wind). By creating a weather system we added to the existing work by making it more realistic and detailed for future runs.

In our simulation, if entities move against the wind, their speeds are slowed and may even be reduced to 0 if the wind speed is high enough. However, if entities move in the direction of the wind, the winds increase the speed of the entities and entities move faster.

Which design pattern did you use?

Weather is implemented using a Singleton. Weather should be universal throughout the simulation since it only encompasses the University's campus. Therefore all entities should reference the same object when determining the effects of weather.



How to interact with weather

Weather is not user interactable. The varying wind speeds are random with sunny having a 30% chance, breezy having a 30% chance, high winds with a 30% chance, and tornadoes having a 10% chance of occurring. The user can view which type of weather is occurring in the top right in the menu screen under the simulation speed slider.

Battery/Recharge Station Feature

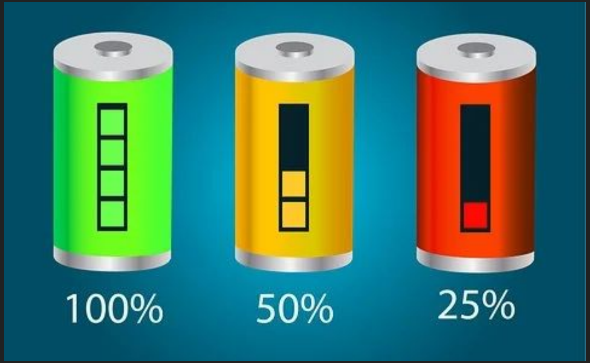
- Adds a battery to the drone, calculating whether it can complete the whole trip with sufficient remaining energy afterward
- <u>Sufficient remaining power:</u> travel to the nearest recharge station after a trip
    - Otherwise, replenish its power at the nearest recharge station.
- After recharging, it will move from its current position to complete the scheduled trip.
- The battery will deplete as it attempts to complete the travel request.
- There are three recharge stations spread out in the simulation.
    - Each station also has a repair drone.

Battery/Recharge Station Feature

- Edge case: drone runs out of energy mid-trip due to the weather conditions decreasing its speed
- Solution: a repair drone from the nearest recharge station will be dispatched to the dead drone's location to replenish its energy.
- The repair drones will not run out of power mid-trip due to weather conditions
  - Illustrates a possible solution to this logic problem.
- From a business perspective, we could implement repair drones with solar power, so they recharge while moving

Why is this new feature important?

- Helps us model the drone system with an added **energy constraint**.
- When a drone cannot complete a trip with sufficient remaining energy, it will travel to a recharge station to recharge.
- The weather conditions interaction could fully deplete a drone's energy.
- The solar repair drone solution factors in possible monetary cost constraints on the energy system.



How does it add to the existing work?

- The existing work only accounts for traveling to and delivering that robot.
- Battery/Recharge station system forces each drone to factor in the energy costs of a trip.
- A repair drone was added to solve the possibility of a drone failing to complete a travel request due to weather conditions

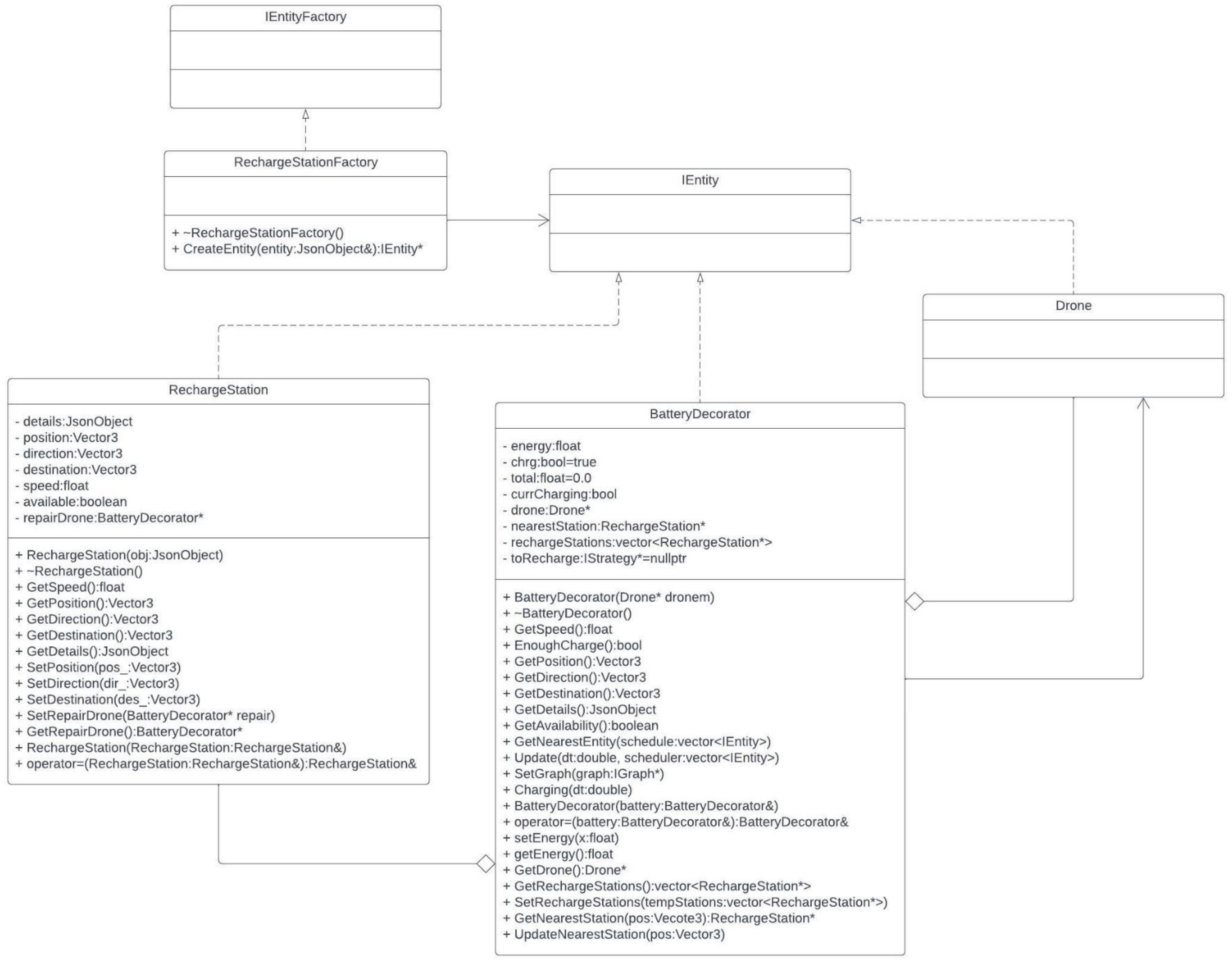
Which design pattern did you use?

Battery - Decorator:

- The decorator pattern allows us to add new functionality to our existing drone class without altering the original functionality.
- The battery decorator is a wrapper class that attaches to a drone entity.

Recharge Station - Factory:

- The recharge stations were implemented into our simulation using a factory pattern.
- The factory pattern allows us to create similar entities while reducing redundant code.


IEntityFactory
RechargeStationFactory
+ ~RechargeStationFactory()
+ CreateEntity(entity:JsonObject&):IEntity*
IEntity
Drone
RechargeStation
- details:JsonObject
- position:Vector3
- direction:Vector3
- destination:Vector3
- speed:float
- available:boolean
- repairDrone:BatteryDecorator*
+ RechargeStation(obj:JsonObject)
+ ~RechargeStation()
+ GetSpeed():float
+ GetPosition():Vector3
+ GetDirection():Vector3
+ GetDestination():Vector3
+ GetDetails():JsonObject
+ SetPosition(pos_:Vector3)
+ SetDirection(dir_:Vector3)
+ SetDestination(des_:Vector3)
+ SetRepairDrone(BatteryDecorator* repair)
+ GetRepairDrone():BatteryDecorator*
+ RechargeStation(RechargeStation:RechargeStation&)
+ operator=(RechargeStation:RechargeStation&):RechargeStation&
BatteryDecorator
- energy:float
- chrg:bool=true
- total:float=0.0
- currCharging:bool
- drone:Drone*
- nearestStation:RechargeStation*
- rechargeStations:vector<RechargeStation*>
- toRecharge:IStrategy*=nullptr
+ BatteryDecorator(Drone* dronem)
+ ~BatteryDecorator()
+ GetSpeed():float
+ EnoughCharge():bool
+ GetPosition():Vector3
+ GetDirection():Vector3
+ GetDestination():Vector3
+ GetDetails():JsonObject
+ GetAvailability():boolean
+ GetNearestEntity(schedule:vector<IEntity>)
+ Update(dt:double, scheduler:vector<IEntity>)
+ SetGraph(graph:IGraph*)
+ Charging(dt:double)
+ BatteryDecorator(battery:BatteryDecorator&)
+ operator=(battery:BatteryDecorator&):BatteryDecorator&
+ setEnergy(x:float)
+ getEnergy():float
+ GetDrone():Drone*
+ GetRechargeStations():vector<RechargeStation*>
+ SetRechargeStations(tempStations:vector<RechargeStation*>)
+ GetNearestStation(pos:Vecote3):RechargeStation*
+ UpdateNearestStation(pos:Vector3)

How to interact with Battery/Recharge Station

The battery and recharge station are not user interactable. The user will schedule a trip, and the simulation will automatically model the battery and recharge station system. If a drone calculates it cannot complete a travel request with sufficient remaining energy, it will stop by a recharge station before attempting a scheduled trip. The battery depletes at a constant rate as it attempts a trip. The repair drones will also automatically travel to and fix dead drones. The user will not have to interact with anything to enable the battery and recharge station system.