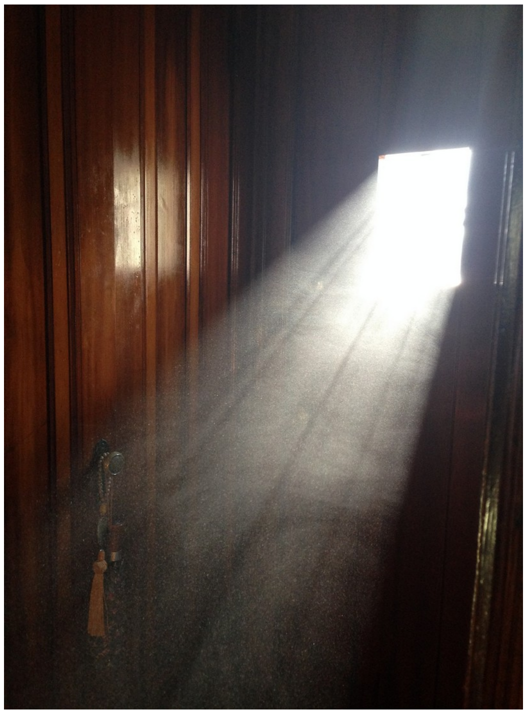
Dust, Actors and Twins

alan@mentalresonance.com



Dust is a software platform that enables the creation of highly scalable and distributable applications. It is an implementation of an approach to software known as Actors which are, as we shall see, a perfect match for representing events in the real or virtual worlds.

This document will introduce Actors and Dust and look at a small example of using them to model the world around a toy system of cars, roads, houses, weather and power plants. We shall see how this model is using both real real-time data (monitoring) and local data (simulation) and we will see how, in Dust, there really is no difference between the two.

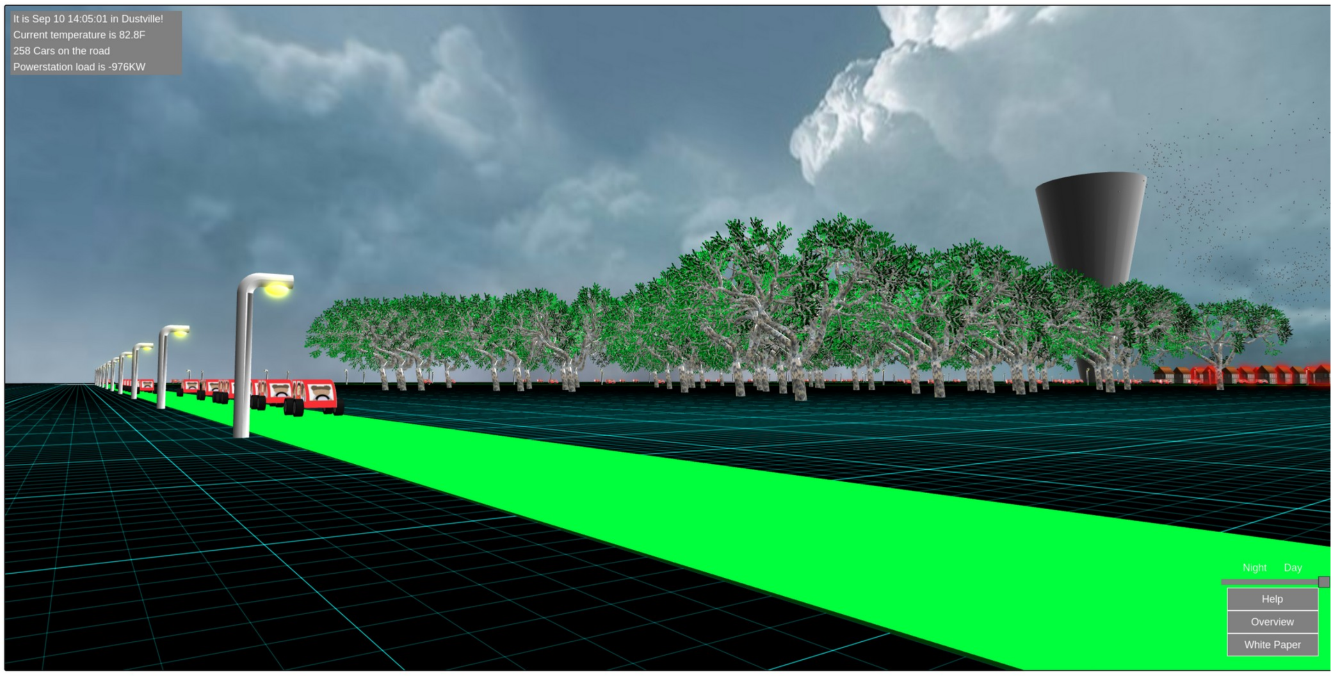
Dust is written in Java and lives on top of the Java ecosystem. It should not be necessary to know Java to understand the enclosed, but if you do know a little Java (or its close cousin, Groovy) then that is all the better.

This document is designed to give a technical overview of Dust and its relationship to digital twinning and is not intended for marketing purposes, so humor remains intact.

The first 10 pages of this paper describe the Dust system and its core architecture. The last 15 pages do a deeper dive into some of the actual Dust code which creates the toy model – *Dustville*.

It is always a good idea to have an example in mind before diving in to the details, so let us briefly describe the 'Twin' that will be used to explore Dust and twinning further.

Dustville



```
It is Sep 10 14:05:01 in Dustville!
Current temperature is 82.8F
258 Cars on the road
Powerstation load is -976KW
```

Dustville is a model of a very simple town. It has roads, and intersections and cars which traverse both. There is a group of houses which have solar panels to help generate power. The power is delivered by (or to) a power station.

There is also a flock of birds continually swooping around the power station.

The whole town has weather – it has sunlight and heat – and the houses respond to this by generating power and trying to keep cool with air conditioners.

Dustville is a cartoon town designed to demo the Dust system – not to be a realistic model of a real town. However the difference is simply one of scale – Dust applications can easily scale to any size, as we shall see.

Everything mentioned above – each bird, car, road, intersection, weather report and house is modeled using exactly the same kind of software object – an Actor. Actors run continually doing the work necessary to model the object they are twinning. Car Actors know where cars are and how fast the are traveling. Road Actors know how many cars are on them at any given time. Intersection Actors also model charging stations around their periphery and the Car Actors (being Electric Vehicles) know when they need to pull over and recharge. All of these actions impacts the power station, which responds accordingly (sometimes by getting angry).

The key to understanding Actors is that they work by passing messages to each other and responding to messages. A car (Actor) pulling over to an intersection (Actor) to charge sends it a message indicating how much power it is going to use. The intersection (Actor) sends this message to the power station (Actor) informing it of the increased load.

Clearly the cars, houses, roads etc. are being simulated by (somewhat) realistic, dynamically synthesized data. However the WeatherStationActor is actually a digital twin of a real weather station (in Northern California). The twin knows how to access the live data (we shall meet it later), convert the data to messages and pass them on to interested parties.

Finally there is a ‘Town’ Actor who handles more generic issues – e.g. increasing or decreasing traffic based on the time of day (or night) ...

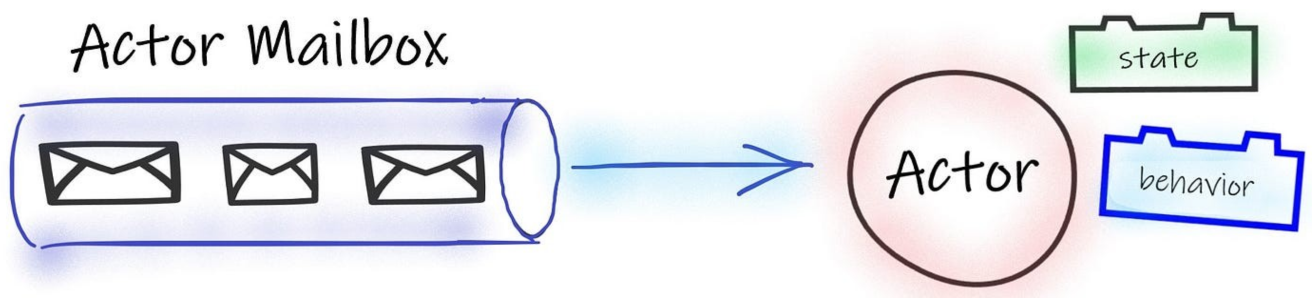
… it is Actors all the way down.

Actors

As we previously stated Dust is an implementation of the Actor paradigm – but what exactly is an Actor?

- Are a computing ‘unit’. They may have internal state but unlike object-oriented programming an Actor’s state is not directly accessible outside of the Actor.
- Actors can exchange messages with other Actors – this is the only way their influence can be spread
- On receipt of a message an Actor processes it, does whatever it is supposed to do, and then waits for the next message. Importantly an Actor is only ever processing a single message at any given time. Messages live in the Actor’s sequential ‘mailbox’ until processed by the Actor.
- The Actor system makes a guarantee that the order of messages in its mailbox sent from the same Actor will be in the order the sender sent them. So if Actor A sends message in order A1, A2, A3 to Actor C, and Actor B does likewise, then at some point Actor C’s mailbox may look like A1, B1, B2, A2, B3, A3 and Actor C will access these in this order. Note that messages may be interleaved from different Actors, but will always be in the correct order.
- Messages are sent fire-and-forget. There is no (automatic) acknowledgment of receipt but this can of course be added on (by sending a reply message).

That’s it. Messages have some recognizable ‘type’ which varies by application, and what an Actor does in response to a particular message is called its Behavior.



There is no large loop managing this. From a designer’s point of view there can be no collisions (attempting to manipulate the same piece of state at the same time) between Actors because all they do is send messages to each other. Actors sit and wait to receive messages, act on them, and sit and wait to receive more.

And there can be millions of inter-cooperating Actors.

Dust

So far we have motivated our discussion by the observation that if you want to simulate or monitor aspects of the real world then the Actor model comes with distinct advantages. In addition to those already discussed there is another very useful advantage that comes virtually for free:

There need be very little difference between monitoring and emulation code when the Actor approach is taken. Consider wishing to twin a car and a road. For this we would have a CarActor whose state includes which road it is on, its position in that road and its speed.

The road, obviously is modeled by an Actor. This RoadActor knows the location and speed of all of the cars on it (information that might be useful for TrafficLightActor). The CarActor will exchange messages with the road to keep it up to date and so that other cars on the road can also have an idea of road conditions.

All of this is clearly being drive by the CarActors. As they update their speed the roads and traffic lights update accordingly (and this data may well be fed into planning systems).

But where does the CarActor get its position and speed information from? There are (at least) two possibilities:

- Output of real-world sensors which can track cars and determine their position and speed. These would wrap this information into, let's say, a CarLocationMsg[1] which it sends to a corresponding RoadActor. Part of the RoadActor's job is to create a CarActor for a new car that may have just entered it.
- Output of a CarSimulatorActor. This might live in a parallel Application which contains some model a user wishes to explore. The CarSimulatorActor sends CarLocationMsgs to the RoadActor.

Note that ***the RoadActor is not concerned with who the sender of the CarLocationMsg was***. The core of this system remains the same – TrafficLightActors react to real traffic or simulated traffic in exactly the same way.[2]

1 You may have guessed (correctly) by now we have the convention the classes of Actors are named with an 'Actor' suffix and those of messages by a 'Msg' prefix.

2 Of course if TrafficLightActors sent messages to affectors which actually changed the traffic lights then one might want to simulate that as well otherwise some chaos might ensue and fun times be had.

Entities

Dust is an Actor system but it adds a layer to make it easier to create and manipulate Entities. What is an Entity? It is anything that might be twinned by an Actor.

Often Entities have a life cycle – they come into being, persist for a while and then are gone. Examples include:

- Physical objects – Cars have a comparatively brief life cycle if you are just twinning them while they are on a road, but may have a longer life cycle if you include them while they are parked.
- Intangibles – a car driving at excess speed should be watched more closely, so perhaps a tighter monitoring Actor should be introduced to apply focus to this car. But when the car begins to behave again this Actor may be disposed of.

Entities must have some form of unique ID. This may be innate to the entity (US companies have EIN numbers) or imposed by the application (some form of unique identifier is given to that brief focus Actor).

Digging in the Dust

Armed with our motivation and a characterization of Actors and Entities we can look more into Dust and see how applications are realized.

Dust and Java

(This section assumes some elementary knowledge of Java or similar languages – e.g. C#).

Dust is written in Java and runs on the JVM and so everything in Dust is a Java object. Java inheritance is used as it usually is in Java programs - for instance one type of an Actor may be a more specialized version of another Actor and all Actors inherit from a class, called – what else – Actor.

Messages can be any Java object but if there is a need to persist these messages or send them over a network then they need to be Serializable.

Dust and Mutability

The durability and simplicity of the Actor model and Dust is you don't have to worry about shared state across threads causing problems by simultaneously being accessed by two threads. State in an Actor should be private or protected so it cannot be access from the outside. However there is the problem of messages – they are created in one Actor and passed to another so it is possible for both Actors to have access to the same 'state' – in this case the message, and in theory, could bang into each other.

Dust relies on cooperation here. If the sending Actor thinks the receiver might have a reason to modify the message it should send a *copy* of the message.

Creating and Addressing Actors

Actors can only be created by other Actors (with one exception) or by the system. When an Actor creates another it is the 'parent', and the created Actor is a 'child' of that parent.

When an Actor is created it should be given a unique name by its parent (unique within its siblings) or the creation process will give it one.

If you follow an Actor up the chain by following parents you end up at a system created actor whose name is 'user'. So analogous to a computer where files have names, and each one can be identified by a unique path through named (sub)directories which lead to it, one can do the same with Actors using '/' to separate the names. e.g.

```
        /user/carworld/roads/a27/car-8bh407
```

Here car-8bh407 is the name of a CarActor, it is the child of RoadActor a27 which is the child of an Actor called roads who exists to manage the road network, which is in turn a child of Actor carworld which manages the whole model who is a child of Actor user who is a

SystemActor who manages everything. So in this path there are 5 different Actors. Now consider another path:

```
/user/carworld/roads/a27/car-844ter
```

This is the path down to another CarActor which is currently on the same road. Again the path has 5 Actors but the first 4 are the same as before. So the two cars are siblings (share the same parent).

Now assume a little time has passed and our first care has turned onto a different road (say M1). The Actor at /user/carworld/roads/a27/car-8bh407 will have been destroyed, and a new Actor

```
/user/carworld/roads/m1/car-8bh407
```

created. Note that the CarActor at .../car-8bh407 is not the same Actor as before. That Actor was destroyed and a new one – with new state but most likely the same car behavior was created.

Suppose I am an Actor wanting to send a message to this new CarActor. I need to get what is called a reference to it and then send it a message:

```
context.actorSelection(“/user/carworld/roads/m1/car-8bh407”).tell(new SlowDownMsg(),self)
```

Let’s unpack this. context.actorSelection() says give me a reference to the Actor at the named path. References have many methods, one of which is:

```
tell(<message to send>, reference of sender)
```

‘self’ is an Actor reference which identifies me (the sender). So in this case ‘I’ (PoliceCarActor ?) have sent a SlowDownMsg() to the (Actor for the car whose license plate is 8bh407).

When the receiving Actor comes to process this message it is associated with a variable ‘sender’ which is a reference to the message-sending Actor. So it can easily send a reply:

```
sender.tell(new NoMsg(), self)
```

Here we have taken artistic license assuming we could get a car’s (artistic) license plate and use that as the unique identity for a car ***Entity***. It is important to realize this is not the unique name of the CarActor but is only a unique Actor within a given path. For instance, after ignoring the slow down message to CarActor /user/carworld/roads/m1/car-8bh407 we might find the creation of a new Actor in a very separate path:

```
/user/carworld/legal-proceedings/speeding/car-8bh407
```

The Actor at the end of this path is not a CarActor although he shares the name. His state is more likely to include ‘maxSpeed’ and ‘legalCosts’ than positional information.

Destroying Actors

Actors have a life cycle and often they might need to stop. They can be stopped in one of four ways:

- They stop themselves deliberately
- They receive a message telling them to stop themselves
- One of their ancestors on their path stopped
- They threw some kind of exception

Thus to a degree the tree of Actors is self managing, for if it were not and the tree has say a million Actors the programmer would find that a bit of a handful to manage.

In dust there are many useful patterns that don’t appear in usual programming approaches. For instance there is the ‘boredom’ pattern: “Create yourself and look for the following. If it hasn’t happened in two days then kill yourself”. While it sounds trite this is actually a very useful pattern.

The final cause – a bug in the code or an unexpected condition (undefended network error, say) is treated specially. A message is sent to its parent describing what happened and the parent has a choice:

- Just let it stop
- Tell it to continue with its state as is
- Tell it to restart itself with a fresh slate

This mechanism allows a degree of fault-tolerance – actors can get re-instated in the face of errors[3].

Remote Actors

We initially spoke of the scalability and distributability of Dust. Scalability simply comes from the use of Actors. These are very lightweight objects – millions per second may be created and destroyed and millions of messages sent per second. Dust uses as many native hardware threads its host may have (usually twice the number of processors) but then layers another much lighter threading layer on top of it. At this level there is one ‘virtual thread’ for each of the millions of mailboxes.

A Dust application runs in a process called an Actor System. Actor Systems are named and the name can be prepended to the path. So

```
/user/carworld/legal-proceedings/speeding/car-8bh407
```

is a reference to an Actor in my Actor System.

```
/next-town-over/user/carworld/legal-proceedings/speeding/car-8bh407
```

3 This is the famous Joe Armstrong ‘Just let it crash’ mode of operation. His thesis famously pointed out if you let things crash and restart them in the appropriate way you can often build much more reliable systems than trying to handle the error and recover. The Erlang language implements this.

is a reference to an Actor in a similar situation in a different Actor System (called next-town-over) on my host machine whereas:

```
dust://mars.com/mars-port/user/spaceworld/legal-proceedings/speeding/spaceship-8bh407
```

demonstrates the durability of human nature.

Reverse Scalability

Scalability implies growth – in terms of data, processing, number of Actors etc. But often in digital twinning there is the overlooked (and equally hard) issue of reverse scalability: what kind of processing can we put at the edge?

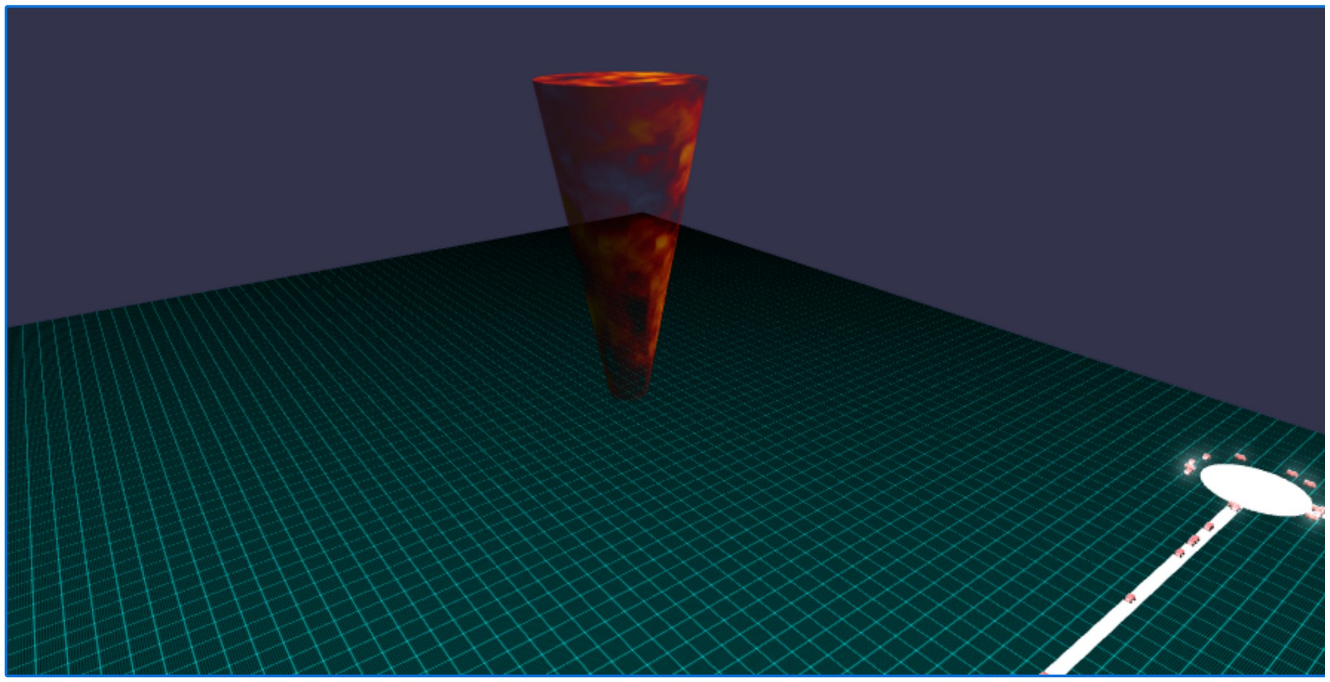
The answer to the ‘what kind of processing?’ is obvious – Dust and Actors - but can we do it?

Dust is written in Java and so can run wherever a Java Virtual Machine of the correct version can run. While Dust comes with many different modules the basic Dust-Core is quite small and is easily capable of running on a Raspberry PI and running some thousands of Actors quite happily.

In fact this was demonstrated with a predecessor to Dust running on a RaspberryPI 3B which was being used as a Wireless sniffer. It sent messages to an Actor System modeling and monitoring a building with rooms and floors and could be used as a proxy for the number of people in a given space.[4] Dust is lighter than its predecessor.

4 This was at the start of Covid.

Diving into Dustville



Intersections

IntersectionActor

We start with the Actor that represents an intersection – a place where several roads meet and which can deliver power to recharge batteries. An intersection's state consists of knowledge of where it is, and a list of references to RoadActors which represent the intersecting roads.

Since this is our first look at some code we will go gently and give textual descriptions of what the code is doing. Even if you've never coded before it should be quite possible to see what is going on.

```
class IntersectionActor extends Actor {

      int x, y

      double chargeLoad  = 0

      List<ActorRef> roads = []

      /**
       * Create intersection @ x,y
       * @param x
       * @param y
       * @return
       */
      static Props props(int x, int y) {
            Props.create(IntersectionActor, x, y)
      }

      IntersectionActor(int x, int y) {
            this.x = x
            this.y = y
      }

      @Override
      void preStart() {
            scheduleIn(new StartMsg(), 30 * 1000)
      }

      @Override
      ActorBehavior createBehavior() {
            (message) -> {
                  switch(message) {
                        case RegisterRoadMsg:
                              roads << sender
                              break

                        case GetRoadsMsg:
                              ((GetRoadsMsg)message).roads = roads
                              sender.tell(message, self)
                              break

                        case CarChargingMsg:
                              chargeLoad += ((CarChargingMsg)message).car.chargeCapacity
                              break

                        case CarFinishedChargingMsg:
                              chargeLoad -=
                                    ((CarFinishedChargingMsg)message).car.chargeCapacity
                              break

                        case StartMsg:
                              context.actorSelection("/user/powerstation").tell(
                                    new PowerLoadMsg(chargeLoad), sender)
                              scheduleIn(new StartMsg(), 30 * 1000)
                              break

                        default: super.createBehavior().onMessage(message)
                  }
            }
      }
}
```

The first line is the usual Java (or rather Groovy, in this case – think of it as Java without semicolons …) declaration of a new class called an IntersectionActor. It extends the Actor class and so will inherit Actor methods (and all the mailbox infrastructure etc).

The state of the intersection is its (x,y) coordinate location (we will be in a 3D world but think of it as Kansas), the electric load from charging at the intersection and a list of references to the RoadActors which represent the roads forming this intersection. How it gets these we'll come to in a minute.

Since we have to build infrastructure for each Actor instance we can't simply do a 'new IntersectionActor()'. The static props() method and the following IntersectionActor() constuctor are the mechanism by which a new Actor of this type is created.

Following this is a preStart() method. This is called immediately after the Actor has been created and is ready to receive message. Here we use the handy 'schduleIn()' method which sends an arbitrary message to an arbitrary Actor after an arbitrary amount of time (measured in milliseconds) . Here I am not telling it where to send the message, so by default it sends a StartMsg back to me in 30 seconds.

Finally we get to the meat of Dust: ActorBehavior createBehavior() {}.

This methods defines an Actor behavior – i.e. it takes the next message from its mailbox (if one is available; otherwise it waits) and processes it. Usually we handle it, as here, by a switch on the message class.

- RegisterRoadMsg - I simply add him (sender is his reference – see **Setup** below) to my list of road references.
- GetRoadsMsg – I simply file a response with my current list of roads.
- CarChargingMsg – I simply add a value to my current electrical load
- CarFinishedChargingMsg – I remove that value from the load
- StartMsg – I send a PowerLoadMsg carrying my current load to an Actor called powerstation. Then send myself another StartMsg in 30 seconds.

All other messages I send to my super-class which is the raw Actor for default processing.

That's it – I've defined all possible messages an intersection can receive and what it does when it gets them.

Setup

We begin with a list of pairs of x,y points – which determine the beginning and ends of the roads:

```
public static List<List<Integer>> roads = [
      [0, 0, 500, 0],
      [500, 0, 500, -500],
      [500, -500, 0, -500],
      [0, -500, 0, 0],
      [0, 0, -500, -600],
      [-500, -600, -500, -130],
      [-500, -600, 0, -500],
      [-500, -130, 0, 0],
      [0, 0, 0, -500],
      [0, -500, 500, -500],
      [500, -500, 0, 0],
      [-500, -130, -500, 500],
      [-500, 500, 0, 0]

]
```

Then we simultaneously construct RoadActors for each road while tracking the unique intersections (x,y coordinates):

```
roads.each {
      ActorRef road = context.actorOf(RoadActor.props(it, i), "road$i")
      ++i
      List key1 = it[0..1], key2 = it[2..3]
      [key1, key2].each {
            List<ActorRef> refs = intersections[it] ?: []
            refs << road
            intersections[it] = refs
      }
}
```

the RoadActors being imaginatively named road1, road2 etc. When it comes to building the intersections we have a slightly more useful naming scheme:

```
intersections.each {
      int x = it.key[0], y = it.key[1]
      ActorRef intersection = context.actorOf(
                  IntersectionActor.props(x, y), "intersection_${x}_$y"
            )
      ++i
      it.value.each { roadRef ->
            intersection.tell(new RegisterRoadMsg(), roadRef)
      }
}
```

Here the intersections are named by incorporating their coordinates, which are also the coordinates of the ends of the road – so when a CarActor comes to the end of the road it can easily get a path to the associated intersection Actor. Then for each road we send the intersection a RegisterRoadMsg but we send it *as though it had come from the corresponding RoadActor*. so in response to this message the IntersectionActors builds a list of references to their intersecting RoadActors.

Finally we generate 500 cars and their Actors. We position the cars at random intersections and give them random initial speeds.

```
      for (int j = 0; j < 500; ++j)
      {
            int road = ThreadLocalRandom.current().nextInt(0, roads.size())
            List coord = roads[road][0..1]
            Car car = new Car(j)

            car.chargeCapacity = ThreadLocalRandom.current().nextInt(80, 120)
            car.chargeRemaining = car.chargeCapacity
            car.speed =  ThreadLocalRandom.current().nextInt(1, 4)
            car.point = new Point(coord[0], coord[1])
            car.carRef = context.actorOf(CarActor.props(car), "car$j")
            car.carRef.tell(new AtIntersectionMsg(car), null)
      }
```

The Car object (as opposed to the CarActor) contains its location and speed and information about its charging state. The corresponding CarActor uses this object as part of its state.

Car Actor

CarActor is actually pretty passive. Here is its behavior (‘car’ is the Car object with position and speed information):

```
case AtIntersectionMsg:
      actorSelection("/user/intersection_${(int)car.point.x}_(int)car.point.y}")
                        .tell( new GetRoadsMsg(), self)
      break
```

When the CarActor gets an AtIntersectionMsg its job is to chose its next road to enter, but it has to learn what these are and so it sends a message to the corresponding IntersectionActor asking for that information. On receipt of that message back from the IntersectionActor:

```
case GetRoadsMsg:
      GetRoadsMsg msg = (GetRoadsMsg)message
      // No U turns
      List<ActorRef> roads = currentRoad ? msg.roads-currentRoad : msg.roads

      if (roads != []) {
            currentRoad = roads[ThreadLocalRandom.current().nextInt(0, roads.size())]
            if (car.chargeRemaining < car.chargeCapacity * 0.1) {
                  car.charging = true
                  car.speed = 0
                  oldPoint = car.point
                  car.point = chargePoint(car.point.x, car.point.y, 30.0d)
                  actorSelection(
                        "/user/intersection_${(int)oldPoint.x}_${(int)oldPoint.y}"
                  ).tell( new CarChargingMsg(car), self)
                  scheduleIn(new DoneChargingMsg(), car.chargeCapacity as Long * 1000L)
            }
            currentRoad.tell(new CarEnteringMsg(car), self)
      } else // Dead end
            context.stop(self)
      break
```

We extract the roads from the reply and remove the current road we are on. We then randomly choose one (there should always be at least one road available, if not we messed up and so we just stop the whole ball of wax).

Before moving on we check our battery level. If we are too low we remember where we were, set our speed to 0, mark ourselves as charging (so the UI will make us glow) and pull over to the side of the road. We tell the intersection we are charging and then we then compute how long the charging will take and schedule a DoneChargingMsg for that time.

In both cases we tell our new road we are entering it.

```
case DoneChargingMsg:
      car.chargeRemaining = car.chargeCapacity
      car.point = oldPoint
      car.speed = ThreadLocalRandom.current().nextInt(1, 4)
      actorSelection(
            "/user/intersection_${(int)car.point.x}_${(int)car.point.y}"
      ).tell( new CarFinishedChargingMsg(car), self)
      car.charging = false
      break
```

When we have finished charging we restore our position, chose a random speed and tell the intersection we have finished charging.

RoadActor

The RoadActor is responsible for handling a car's entrance and departure and the moving of the car at the right speed down the road. It also adjusts car speeds if they get too close.

Our road is bidirectional so we maintain two lists of cars – *coming* and *going*.

We'll split the createBehavior() method up into its individual message handler:

```
case CarEnteringMsg:
      CarEnteringMsg msg = (CarEnteringMsg)message
      if (msg.car.point.x == startX && msg.car.point.y == startY)
            going << msg.car
      else
            coming << msg.car
      msg.car.distance = 0
      msg.car.angle = angle
      update.cancel()
      self.tell(new UpdateMsg(), self)
      break
```

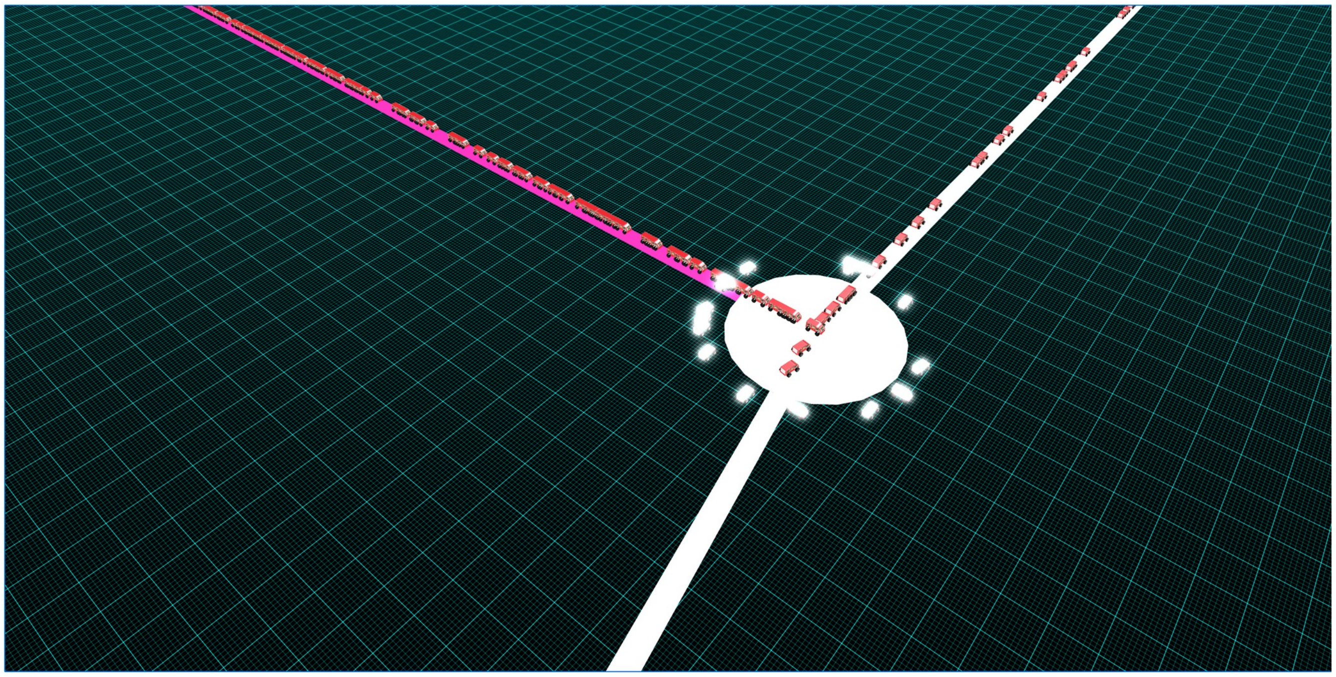
When a CarActor has chosen its next road it sends the RoadActor a CarEnteringMsg which contains information about the car's location. Using this it can put it onto the coming or going list. It adjusts the car's angle to that of the road and sets its distance down the road to be 0. It then tells itself to update immediately.

```
case UpdateMsg:
      [going, coming].each {stream ->
            boolean isGoing = stream.is(going)
            Car inFront = null
            List<Car> toRemove = []

            stream.each {
                  // Update position
                  if (! it.charging) {
                        it.distance = (float)(it.distance+it.speed*deltaT / 1000.0f)
                        it.point = getPoint(isGoing, it.distance)

                        // Close to end of road .. compute charge use here
                        if (length - it.distance < 1) {
                              it.point.x = isGoing ? endX : startX
                              it.point.y = isGoing ? endY : startY
                              it.chargeRemaining = (float)
                                    hargeRemaining - 0.02 * length)
                              it.carRef.tell(new AtIntersectionMsg(it), self)
                              toRemove << it
                        }
                        else {
                              if (inFront) {
                                    if (inFront.distance - it.distance < 10)
                                          it.speed = inFront.speed
                                    }
                              }
                              inFront = it
                        }
                  }
            }
            toRemove.each {
                  stream.remove(it)
            }
            toRemove = []
      }
      update = scheduleIn(new UpdateMsg(), deltaT)
      break
```

An UpdateMsg is processed at regular intervals and its main job is to move the car down the road (by increasing its distance) until it gets to the end. Note that if a car is charging it is viewed as still being on the road but it is not moved. If we are very close to the end of the road we move it to the end and adjust the charge in its battery by how far is has traveled along the road and we then tell the car (via a message to its CarActor) that it is at the intersection and we supply the message with the coordinates of the intersection. Finally we mark it for removal from this road. If it isn’t near the end we simply check to make sure it is not too close to the care in front and if it is adjust its speed. Finally we reschedule the update message.

```
case GetCarsRequestMsg:
      GetCarsResponseMsg resp = new GetCarsResponseMsg()
      resp.roadId = id

      [coming, going].each { stream ->
            boolean isGoing = stream.is(going)
            List<Car> cars = stream.collect {it.clone()}
            if (isGoing)
                  resp.going = cars
            else
                  resp.coming = cars
      }

      int totalCars = going.size() + coming.size()
      resp.density = totalCars / length
      sender.tell(resp, self)
      break
```

The UI periodically sends us a GetCarsRequestMsg and we reply to the Actor representing the UI a GetCarsResponseMsg which contains all the information about the cars as well as a measure of the ‘density’ of traffic. All of this information is passed back through a web socket to the 3 dimensional UI in the browser and it repositions all the cars appropriately. If the UI thinks the care density is too high it changes the road color accordingly.



Power Station Actor

The PowerStationActor is this simplest of all. Here is its code in its entirety.

```
class PowerStationActor extends Actor {

      Map<ActorRef, Double> loads = [:]
      double megaLoad

      static Props props() {
            Props.create(PowerStationActor)
      }

      @Override
      ActorBehavior createBehavior() {
            (message) -> {
                  switch (message) {
                        case PowerLoadMsg:
                              double load =
                                     ((IntersectionPowerLoadMsg)message).load
                              loads[sender] = load
                              megaLoad = loads.values().sum() as double
                              WebSocketCarsServer.powerStationLoad(megaLoad)
                              break
                  }
            }
      }

}
```

The PowerStationActor maintains a table of power loads indexed by Actor references. Whenever it gets a new PowerLoadMsg it simply updates it table with the new value, adds up all the values and sends the value directly to the UI via the Web Socket server.

Real Time Data

So far our model is a simulation of roads, cars and charging stations but we began by saying that Dust makes the intermixing of simulation and monitoring of real-time data simple, so we have added a simple example.

WeatherStationActor

The internet has a network of home weather stations accessible via an API. The WeatherStationActor allows other Actors to request access to this information and be delivered updates to the weather data at set periods. This is the entire WeatherStationActor:

```
class WeatherStationActor extends PubSubActor implements HttpClientActor {

	String url
	Long period

	static Props props(String url, Long period) {
		Props.create(WeatherStationActor, url, period)
	}

	WeatherStationActor(String url, Long period) {
		this.url = url
		this.period = period
	}

	@Override
	void preStart() {
		scheduleIn(new StartMsg(), 5000)
	}

	@Override
	ActorBehavior createBehavior() {
		(message) -> {
			switch(message) {
				case StartMsg:
					request(new HttpRequestResponseMsg(self,
							HttpService.buildGetRequest(url))
					)
					scheduleIn(new StartMsg(), period)
					break

				case HttpRequestResponseMsg:
				      HttpRequestResponseMsg msg = (HttpRequestResponseMsg)message
				      Map data = new Gson()
							.fromJson(
								msg.response.body().string(), Map)
							.observations[0]
					self.tell(new WeatherStationMsg(data), self)
					break

				default:
					super.createBehavior().onMessage(message)
			}
		}
	}
}
```

First notice we are not extending Actor, as usual. Instead we are extending PubSubActor and getting some behavior from HttpClientActor. Both of these are built into Dust.

A PubSubActor accepts subscription messages from other Actors, and its default behavior is to take whatever messages it receives and broadcast them to its subscribers.

An HttpClientActor implements the request() method we takes an HttpRequestResponseMsg which in turn takes a HTTP request. Here the request is a GET on the given url. When the HTTP request completes it sends the requester (myself) the HttpRequestResponseMsg but with the response filled in. In this case we are expecting a JSON object consisting of a list of Maps. So we parse it out and wrap it in a WeatherStationMsg and send that to myself.

WeatherStationActor does not have any defined behavior for a WeatherStationMsg so it passes it on to its superclass – the aforementioned PubSubActor which simply passes it on to any Actors who have registered an interest. These include:

HouseActor

A house reacts to temperature based on the (live) external temperature and how well it is insulated. If it gets too hot it turns on the AC which add more load to the power plant. I'll leave an understanding of the code to the reader who by now should be more than capable of plumbing the depths.

```
class HouseActor extends Actor {

      double load, insulation

      ActorRef powerstationRef

      static Props props() {
            Props.create(HouseActor)
      }

      HouseActor() {
            insulation = ThreadLocalRandom.current().nextDouble(0.8d, 1.2d)
            load = ThreadLocalRandom.current().nextDouble(4d, 8d)
      }

      @Override
      void preStart() {
            // Subscribe to the weatherstation
            context.actorSelection("/user/weatherstation")
                  .tell(new PubSubMsg(WeatherStationMsg), self)
            powerstationRef = context.actorSelection("/user/powerstation")
      }

      @Override
      ActorBehavior createBehavior() {
            (message) -> {
                  switch(message) {
                        case WeatherStationMsg:
                              double temp = insulation * (double)((Map)
                                   ((WeatherStationMsg)message).data.imperial).heatIndex
                              log.info "${self.path} apparent temp = $temp"
                              powerstationRef.tell(
                                    new PowerLoadMsg(temp > 80 ? load : 0.0d),
                                    self
                              )
                              break

                        default:
                              super.createBehavior().onMessage(message)
                  }
            }
      }
}
```

Birds of a Feather

Consider the flocking behavior of birds (every power plant should have birds circling it). How could we simulate a flock of birds using Actors?

Since there is no overall ‘master’ in a flock of birds, how do they flock in real life if all they have is local information (of the birds nearby)? It turns out they do this by following a simple set of rules (which varies by species, apparently). Generally they try to keep some distance between themselves and the birds nearby, and they tend to head in the same general direction as all their neighbors. To a first order approximation we can simply consider the direction issue.

The problem to be solved then is computing the general direction of a neighborhood of a given bird, but this means first of all recognizing these birds. In reality a bird can see (or feel, or hear) its neighbors. We could arrange for all birds to broadcast their position in a message and then use the Dust broadcast bus mechanism which each bird can listen to and so learn where all the others are, but this is inefficient. Most birds will not be near any given bird.

The technique we use is to divide *space* into identical volumes and assign an Actor to each volume. A bird Actor knows its own position and if we give the volume Actors (which we call Hoods – short for neighborhood) suitable names – e.g. hoot_x_y_z where x, a and z are the coordinates of that Hoods center, then a bird Actor can easily compute which Hood it is in and alert the Hood Actor to its presence, position and current direction. The Hood Actor therefore has all the ‘local’ information for birds within it[5] and so can periodically tell all the birds in it the current general direction. The birds then adjust their path to better align with this new direction and the process repeats.

Like Cars and Roads the Hoods have to handle hand-offs, and those on the outside of the overall volume need to prevent birds from leaving the overall flocking space. These details aside the Bird and Hood Actors are amazingly simple:

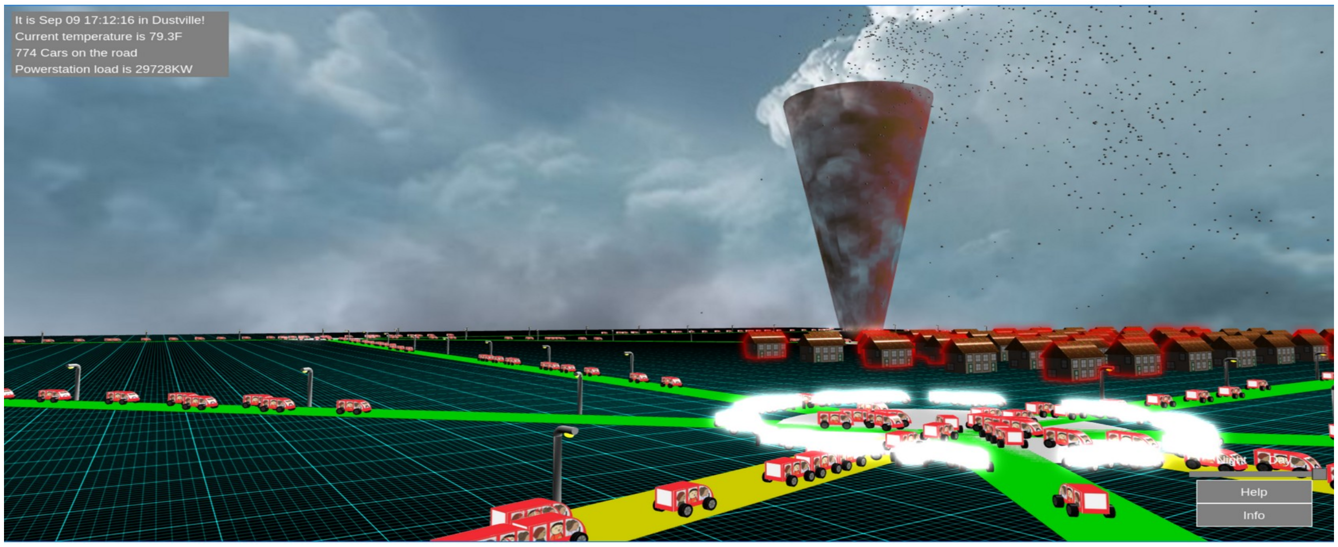
5 This isn’t quite true. If a bird is flying close to the edge of a Hood then its neighborhood might include some birds from an adjacent Hood. But in practice this seems not to matter much.

Bird Actor

```
ActorBehavior createBehavior() { (message) ->
{
	switch(message)
	{
	case InitMsg:
		currentHood = HoodUtils.hood(position)
		currentHoodRef = context.actorSelection("/user/${HoodUtils.hoodName(position)}")
		currentHoodRef.tell(
			new AddBirdMsg(new Bird(self.name, position, direction, speed)),
			self
		)
		self.tell(new StartMsg(), self)
		break

	case StartMsg:
		Vector3D newPosition = position.plus(direction.scale(speed)), newHood
		newHood = HoodUtils.hood(newPosition)
		position = newPosition

		if (position.inCube()) {
			if (! newHood.equals(currentHood)) {
				currentHoodRef.tell(new RemoveBirdMsg(), self)
				currentHoodRef = context.actorSelection(
					"/user/${HoodUtils.hoodName(newHood)}"
				)
				currentHoodRef.tell(
				new AddBirdMsg(
					new Bird(self.name, position,direction,speed)),
					self
				)
				currentHood = newHood
			}
			else {
				currentHoodRef.tell(
					new UpdateBirdMsg(new Bird(self.name, position, direction, speed),
					self
				)
			}
			currentHoodRef.tell(new GetAveragesMsg(), self)
		}
		else {
			// Let direction changes pull us back ...
			position = position.plus(direction.scale(speed))
		}
		scheduleIn(new StartMsg(), 750)
		break

	case GetAveragesMsg:
		Vector3D avgs = ((GetAveragesMsg)message).direction
		if (avgs.length()) {
			direction = avgs
		}
		break

	default: super.createBehavior().onMessage(message)
	}
}}
```

Hood Actor

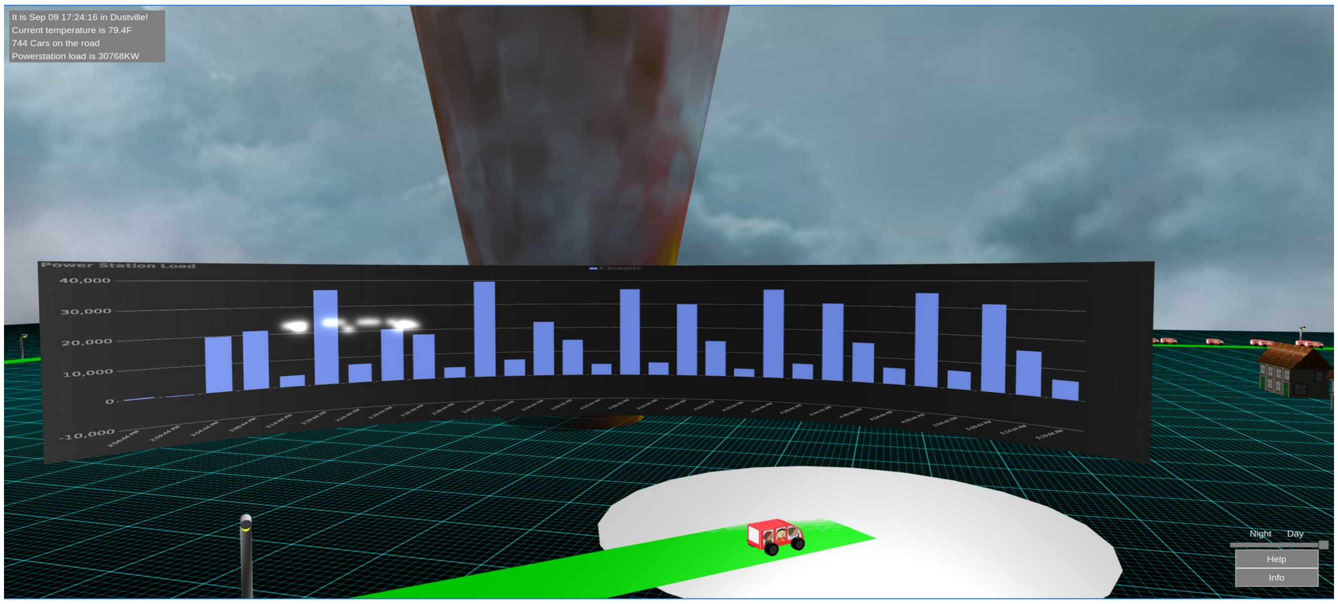
```
ActorBehavior createBehavior() {
       (message) -> {
              switch(message)
              {
              case AddBirdMsg:
                     birds[sender.name] = ((AddBirdMsg)message).bird
                     break

              case UpdateBirdMsg:
                     birds[sender.name] = ((UpdateBirdMsg)message).bird
                     break

              case RemoveBirdMsg:
                     birds.remove(sender.name)
                     break

              case GetBirdsRequestMsg:
                     sender.tell(
                        new GetBirdsResponseMsg(self.name, birds.values().toList()),
                        self
                     )
                     break

              case StartMsg:
                     averageDirection = new Vector3D(0, 0, 0)
                     averageSpeed = 0
                     int flock = birds.size()
                     if (flock) {
                            birds.each {
                              averageDirection =
                              averageDirection.plus(((Bird)it.value).direction)
                                   averageSpeed += ((Bird)it.value).speed
                            }
                            averageDirection = averageDirection.scale(1.0d/flock).normalize()
                            averageSpeed = averageSpeed / flock
                     }
                     scheduleIn(new StartMsg(), 2000)
                     break

              case GetAveragesMsg:
                     ((GetAveragesMsg)message).direction = averageDirection
                     ((GetAveragesMsg)message).speed = averageSpeed
                     sender.tell(message, self)
                     break

              default: super.createBehavior().onMessage(message)

              }
       }
}
```


It is Sep 09 17:12:16 in Dustville!
Current temperature is 79.3F
774 Cars on the road
Powerstation load is 29728KW

3D UI and User Interaction

While Actors spend most of their time messaging other Actors at some point the real world has to intrude – for instance coupling the DustVille Actors to a UI. Dust contains an Actor type that allows the creation of maps between conventional APIs and Actor messages. In particular we can couple 3D UIs (built on BabylonJS) in the browser via websockets with Actors representing the twins. Each CarActor 'drives' a unique car in the UI, birds flock by the power station etc etc.

Using cursor keys or the mouse a user can move through the 3D space. Also clicking on various objects (cars, roads, intersections, houses and the power station) sends a message to the corresponding Actor asking for its state – the state is returned in another message and is displayed by the UI:


It is Sep 09 17:24:16 in Dustville!
Current temperature is 79.4F
744 Cars on the road
Powerstation load is 30768KW