Orbiting Around Globes

Author: Zheyin Zeng
UID: U6545267
Date: Friday 27 September

This delivery has implemented stage b, c and d.

Problems

Here I will state the problems I need to tackle. Then the high-level ideas will be introduced in the *Design* section and the low-level solutions will be introduced in the *Implementation* section.

Overview

My goal is to design a flexible system instead of multiple ones. It should allow as many vehicles as possible to survive. Also, it should be able to handle a small number of vehicles as well.

Where Globes Are

The first task is to know where the globes are. It is easy to just know that as we can use *sensors* but it is difficult to have the **latest** knowledge where the globes are. If the positions are not latest, vehicles may head towards void when needing to charge. So how do we let all vehicles know the latest positions of globes? Let them keep flying around a known globe? That seems not very feasible for a large number of vehicles …

Spreading Messages

All vehicles are equipped with *sensors* to detect globes. But it is quite possible that for some vehicles the globes will never be in the range of their *sensors*. So there must be a way to allow these vehicles, which will probably not meet any globes for a long time, to know the positions of globes so that they know where to go if they need to charge, even if they have not actually met any globes before. In other words, those who find the globes should let others know where the globes are through *message passing interface*.

Coordination

Vehicles will be "freezing" if too many of them fly to the same globe (i.e. position). Thus, coordination between vehicles needs to be introduced. That is, do not let too many vehicles fly to the same globe at the same time.

Idling

"Idling" is a term for me to describe those cases: what vehicles should do right after they finish charging, and what they should do if they have high energy. If we do not control the vehicles at all in the cases above, the vehicles will fly around randomly (i.e. default behaviours). This will then possibly lead to the vehicles not being able to fly back to globes for charging. So we need to design a path for all vehicles where they fly along. This can ensure that vehicles always fly around the globe.

Multiple Globes

If a vehicle find (or receive) multiple globes, what should it do? Well, my solution is that a vehicle only focuses one globe even if it finds multiple globes around. This derives a few more problems: From the prescriptive of sending side, should we send the closest globe? Or the distance does not really matter? From the prescriptive of receiving side, what should it do if the globe information it is receiving is totally different from it received last time? (i.e. totally different globes, instead of great displacement of the same globe). The decision to send which globe must be considered.

Shrinking Size

My idea to shrink the size of vehicles is by introducing a leader vehicle which guides all other vehicles to vanish, and this involves a few problems: 1) all vehicles must agree on one leader; 2) the leader vehicle needs to have the knowledge of every single vehicle existing; 3) there needs to be a plan B if the leader vehicle dies.

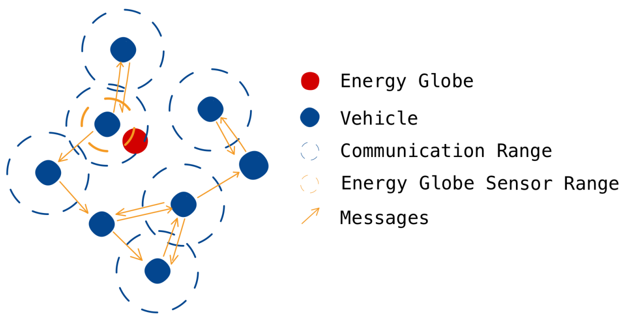
Design

Finding the Globes

Finding the globes is easy as all vehicles head towards the globes at the start of simulation. As a result, some vehicle will eventually obtain the information about the globes in most cases. But the fact is that not every vehicle does due to the fact that the collision-free system prevent vehicles being too close and the range of the sensors are not very far. Therefore, we cannot just rely on each individual vehicles to find the globes on their own. Instead, vehicles should help each other. That is, the ones which know the position of a globe should forward the messages to others. Next section will talk about this idea.

Epidemic-Alike Message Passing

Due to the fact that vehicles can send messages to all its nearby vehicles and vehicles can receive messages at anytime, my idea is that let the messages spread on the way epidemics spread. It is similar to the spread of a virus in a biological community.

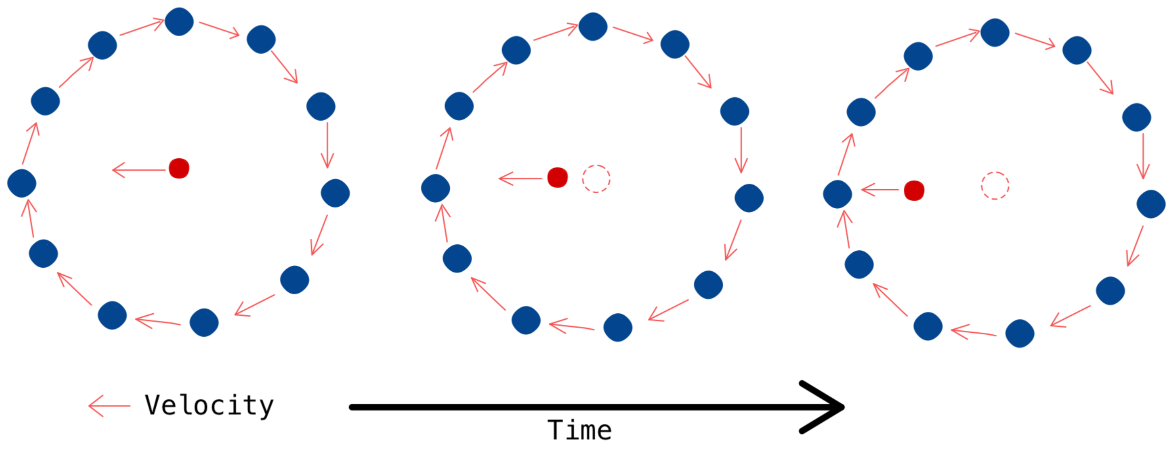


By passing messages on the way epidemics do, as long as there is one vehicle having found the globes and nearby vehicles are present, the globe information should be able to be received by all connected vehicles. The nature of this algorithm is to spread the message out as much as possible so it allows latest messages to be received by many vehicles as long as a new message is sent among the whole vehicle network. It is also the building-block of the whole system I have designed.

Orbiting & Charging

Orbiting refers to vehicles circle round the globes, and is what vehicles do in my design when they do not need to charge (i.e. having high energy). Once vehicles have the knowledge where the globes are, they orbit round the globes therefore. So why orbiting? First of all, orbiting is relatively easy to implement if we use *parametric equation* to draw points. Second, letting vehicles move around help messages being spread under the *Epidemic-Alike Message Passing* algorithm. Third, orbiting is ideal to be the candidate of the vehicle state “idling” — vehicles have high energy and do not need to charge.

But letting vehicles orbit properly requires extra care. Imagine that once vehicles get the position of globes, they start orbiting. What if they keep orbiting but do not update the globe position? The fact is that they will eventually be off the orbit centroid due to the fact that the globes move, as shown in the diagram below:
In this case, the vehicles adjust their orbit iff some vehicle in the network finds new globe position. This works well with large number of vehicles because the moving globes will eventually be within the range of sensors of vehicles so that globe position can be updated. But it does not work well with small number of vehicles as the probability of meeting a globe is much lower.



As a result, some vehicle have to periodically check the globe position — obtains new information about the globe position and brings it back to the vehicle network. So do we really need to explicitly design such a system to do so? Fortunately not, because it is natural for these vehicles which need to charge to do so. When vehicles have low energy, first they fly off their orbit and head towards to the last known globe position. Second, while they are charging, they can actually update the globe position. Last, when they finish charging and go back to orbit, they bring the new globe position as well, allowing the whole vehicle network to update globe position. This process repeats forever, enabling globe position to be updated frequently.

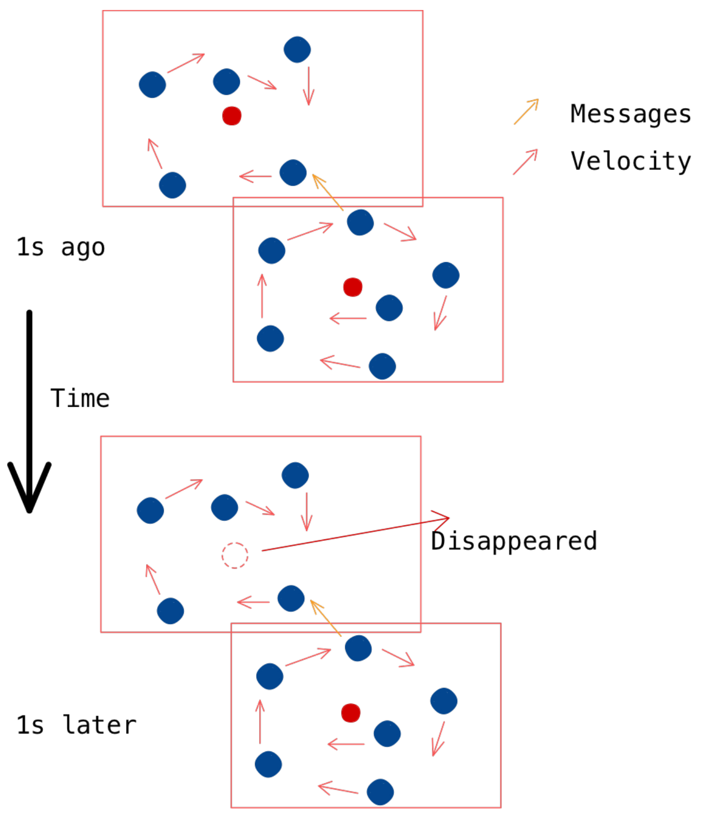
Coordination

If many vehicles head towards the same destination, they will be “freezing”. I have come up two solutions (but only one is taken). Let’s first talk about the one that are not taken.

One solution is that group all vehicles and let each group charge in turn but from my experiments it requires rigorous communication. Each vehicle need to have the latest knowledge of each other before charging. They need to know whether other groups need to charge or not. Otherwise, it would be the case that two or more groups go to charge at the same time. Furthermore, it may not be easy to handle a large number of vehicles since the first task is to group all of them. So, it is possible that some vehicles die beforehand before the completion of grouping (grouping does take time).

Another one is what I have implemented. Overall, it is easy. Vehicles keep orbiting and are not allowed to charge if nearby vehicles also want to charge. They only charge (head towards globe) if the vehicles receive some message that indicates completion of charging. You may think about the case that all vehicles want to charge and therefore none of them will not be able to charge. In fact, it will not happen. From my experiment, it is probably because of delay of message passing and new messages overriding old messages. I will detail this part in the *Analysis & Evaluation* section.

Just One Globe



We do not consider the distance between vehicle and globes and we always pick the latest incoming messages. That is, when a vehicle finds more than one globe around, always grabs one of them (and sends it to others). This is mainly due to the fact the closest globe for the sending side is not necessarily the closest one for the receiving side. Also, we cannot rely on the distance to decide behaviours of our vehicles (e.g. whether to charge; orbit round which globe) as globes would disappear and it is very hard to detect that. The bad case is that, assume we use the "closest-globe algorithm", and from the receiving side, if we always pick the closest globe (compares the incoming globe distance with last known one), the vehicle will be lost if the closest globe has just disappeared (see Figure 3).

Leader Vehicle

The leader vehicle exists for **stage d** and is responsible to guide all other vehicles to vanish in order to shrink the size of overall vehicles to certain one.

The first task is to let all vehicles agree on **one** vehicle which is the leader. This is important because vehicles will be confused if there are multiple leaders guiding them, causing the final size is much less than we expected. Fortunately, it does not require too much effort to achieve the goal due to the fact that in my design the vehicle which first finds the globe will eventually be the leader in the first 5 to 10 sec even if there are multiple globes! It's tricky but it is true. I will detail why it is a fact in the *Implementation* section. Here I just briefly explain: it is the nature of message passing interface.

Once the leader is identified, the second task is to let the leader vehicle have the knowledge of all vehicles so that it can guide which one should vanish. But theoretically we cannot design an algorithm that first obtains number of total vehicles and then does the "subtraction" as vehicles never know how many vehicles there are. So we should think about it the other way. We let the leader vehicle exclude certain vehicles according to those which are already included. That is, once the leader vehicle has the knowledge of adequate number of vehicles to be reserved, it starts to exclude.

The last task is to have a plan B if the leader vehicle dies. Unfortunately, I haven't come up a feasible answer to it so I have to say that the system for stage d is not very robust although the probability that the leader vehicle dies is `1 / (total no. of vehicles)` which is low.

Implementation

Local Storage

I would like to first talk about the local storage of vehicles because I use it so much in my implementation. Local storage refers to the variables declared in the declaration region of task `Vehicle_Task`. So each vehicle have its own copy of variables. It allows vehicles to store information locally for later use instead of fully relying on `Receive()` to get information needed to decide what to do. We need local storage due to the fact the vehicles may have nothing in their message queues but they do need to do something (e.g. orbiting) that relies on received messages. Following are the full story of local storage.

Message passing related local storage are:

Variable Name	Variable Type	Description
`Last_Msg`	`Inter_Vehicle_Messages`	The last received messages by this vehicle. Contains globe position which can be used to go back when vehicle fly off the vehicle network accidentally.
`Local_Charging`	`Boolean`	Indicates whether this vehicle is charging. This variable is local and is not sent to others. This is to ensure the behaviours of this vehicle is not affected by incoming messages.
`Reserved_Vehicles`	`Set (of Vehicle_No)`	The set is dedicated to Leader Vehicle. This contains those vehicles which should not be scheduled to destruct.

Other local storage:

Variable Name	Variable Type	Description
`T`	`Real`	The parameter of parametric equation for orbiting.
`Destruction`	`Boolean`	The configuration variable. Controls whether to activate destruction mode for stage d.

Local storage will be discussed further in the following sections.

The Loop

Manually controlling vehicles is driven by the loop inside each task. The code outside the loop is only executed “once” as opposed to the code inside the loop which is constantly running until vehicles die. This means that we do the initialisation (e.g. initialising the `Last_Msg` record) outside the loop and do calculation (e.g. calculating orbit) and take actions (e.g. see if it should/needs to charge) inside the loop.

Sending Messages

In my design, there are three cases where vehicles send messages (i.e. where we invoke `Send()`): 1) when they find globes; 2) when they have low energy and need to charge; 3) when they receive messages and they forward (send) them.

1) When vehicles find globes

If there are globes around, we pick the first globe and write the new globe information into local `Last_Msg` and sends it out. Note that in this `Last_Msg` the `Charging` flag is explicitly set to be False (meaning “I’m not going to charge, you can go then.”) instead of using `Last_Msg.Charging`. So why explicit `False`? This is because having found globes usually means that the vehicle has just fin-ished charging. This informs other vehicles which need to charge of that they can go.

If there are no globes around, the vehicles do not send messages. Instead, they try to receive messages — see next section *Receiving & Adjusting & Forwarding Messages*.

2) When vehicles have low energy and need to charge

We first let vehicles send messages to indicate that they are charging (that is, in the outgoing messages we set `Last_Msg.Charging` to be `True`). Vehicles which receive such messages are not allowed to charge even if they have low energy. Meanwhile, we set `Local_Charing` to be True as well in this case. We use `Local_Charing` to decide whether the vehicle should orbit around or head towards globes, like in the piece of code:

```
if not Local_Charging then
      Orbiting (Throttle => Full_Throttle * 0.5);
end if;
```

P.S. `Orbiting` is a helper procedure that lets vehicles orbit around the last known globe.

3) When vehicles receive messages and they forward them

This case implements *Epidemic-Alike Message Passing* algorithm. In this case receiving and sending are a bit interleaved because vehicles have to receive and adjust messages before forwarding them so I leave this part to the next section.

Receiving & Adjusting & Forwarding Messages

Vehicles try to receive messages in every physics update. But if we just invoke `Receive()` to receive messages, the task may be blocked, causing the vehicle out of control. So, we first invoke `Message_Waiting` to see if we should invoke `Receive()` to retrieve messages and adjust & write the messages into local `Last_Msg` for later use.

Suppose that `Message_Waiting` has returned `True` and we have retrieved the messages. In my design, **stage b & c** only need two variables in the `Inter_Vehicle_Messages` record:

Variable Name	Variable Type	Description
`Charging`	`Boolean`	Whether this vehicle is going to charge.
`Globe`	`Energy_Globe`	Energy globe information.

And vehicles do not have to adjust these messages before forwarding them. They just forward them without any modification once receiving them.

But for stage d, we adds more variables to the `Inter_Vehicle_Messages` record:

Variable Name	Variable Type	Description
`Sender`	`Swarm_Element_Index`	Who sends this messages.
`Leader`	`Swarm_Element_Index`	The leader vehicle. At the beginning, all vehicle tasks initialise it with their own `Vehicle_No`.
`Target_Vanished`	`Swarm_Element_Index`	Destruction target.

Then we need to adjust the information above before forwarding it to others.

Sender

When a vehicle receives some messages, it writes its own `Vehicle_No` into `Last_Msg.Sender` before forwarding them. This variable is for the *leader vehicle* to count the number of distinct vehicles (i.e. the leader vehicle puts all IDs it meets into a `set` and check the size of the `set`).

Leader

`Last_Msg.Leader` in each vehicle will only be implicitly overridden by the oldest message (or the vehicle which first finds the globes) because of the nature of message passing interface. Within 2 to 5 sec, all vehicles will eventually have the same Leader in their own Last_Msg. And this is exactly how to let all vehicles agree on the same leader vehicle!

`Target_Vanished`

If the vehicle is leader vehicle, it will write the destruction target into `Last_Msg.Target_Vanished` when requirements are met (for details see *Taking Actions* section). Then the leader vehicle sends it out to guide certain vehicles to destruct themselves.

Taking Actions

P.S. This part is a bit interleaved with sending & receiving & adjusting messages.

Vehicles have to decide what to do depending on their current states and received messages. In this part I will talk about how vehicles decide what they do.

1) Orbiting

Orbiting round globes can be seen as drawing points round globes and then we let points act as destinations of vehicles. We identify points by using circle paramedic equation such that points make up of a circle. So if we set the destination of vehicles to be the points produced by circle paramedic equation, then the vehicles can therefore orbit round the globes.

Circle parametric equation:

$$x = r \cdot \cos(t) + a$$
$$y = r \cdot \sin(t) + b$$
$$z = c$$

Where r is orbiting radius and (a, b, c) is the position of a globe.

P.S. For the z position, we just use the z of the globe so that vehicles can orbit around the globe.

2) Charging & Coordination

Vehicles head towards the last known globe from `Last_Msg.Globe` at full throttle iff when they have low energy **and** `Last_Msg.Charging = False` which indicates that some vehicles are not currently charging. This prevent too many vehicles heading towards the globe at the same time. On the contrary, `Last_Msg.Charging = True` indicates that one or more vehicles are currently charging. In this case, vehicles keep orbiting even if they have low energy.

When vehicles finish charging, they do not immediately send messages to tell others they are done, but they implicitly do so. It is done in the following code:

```
if Has_Energy_Nearby (Energy_Globes_Around) then
      declare
            Lucky_Globe  : constant Energy_Globe :=
                                        Grab_A_Globe (Energy_Globes_Around);
            Outgoing_Msg : constant Inter_Vehicle_Messages :=
                                        (Sender => Vehicle_No,
                                        Globe => Lucky_Globe,
                                        Charging => False,
                                        Leader => Last_Msg.Leader,
                                        Target_Vanished => Last_Msg.Target_Vanished);
      begin
            Last_Msg := Outgoing_Msg; Send (Last_Msg);
      end;
end if;
```

Note that `Charging` is explicitly set to be `False`.

3) Destruction

Destruction is for stage d. This is carried out by the leader vehicle.

First of all, the leader vehicle has to be identified. This is done by two steps:

a) In the initialisation of each vehicle task, everyone writes their own `Vehicle_No` into `Last_Msg.Leader`. This variable will not be explicitly modified. It will only be overridden when vehicles try to receive messages, as shown in the following code:

```
if Messages_Waiting then
      declare
            Incoming_Msg : Inter_Vehicle_Messages;
      begin
            Receive (Incoming_Msg);
            -- replaces all local messages with incoming messages
            Last_Msg := Incoming_Msg;
            ......
```

b) Then we let vehicles send and receive message as usual … eventually the `Last_Msg.Leader` of **all** vehicles will be the same, which means that they all agree on the same leader vehicle. It is very trick and I will analyse the reason in the *Analysis & Evaluation* section.

Second, the leader vehicle gathers `Vehicle_No` from its incoming messages by storing them in its local `set`. Once it gets enough number of unique `Vehicle_No` within the `set`, it starts to guide those, which are **not within** the `set`, to destruct themselves by writing `Vehicle_No` into `Last_Msg.Target_Vanished` and sending it out.

Last, once a vehicle receives the messages where its Vehicle_No is equal to `Last_Msg.Target_Vanished`, it destructs itself by exiting the loop.

The three steps above repeat.

Analysis & Evaluation

Stage b

I have run my system twice under stage b with initial size of 128. Overall, the system is very robust for stage b as shown in the following tables:

Time Past	No. Of Vehicles Alive
5 min	~128
10 min	~127

It also supports a low number of vehicles with initial size of 8:

Time Past	No. Of Vehicles Alive
5 min	~8
10 min	~8

The reason why it is robust is because there is only one globe and every vehicle share the same globe. Even if the globe makes a great displacement, it will not confuse vehicles as they can always update globe info by message passing interface. Hence, the flying pattern looks very nice — they all orbit round the same globe.

Stage c

I have run my system twice under stage c with initial size of 128. The system starts to show instability and the performance is shown below:

Time Past	No. Of Vehicles Alive
5 min	~110
10 min	~100

From my observation, this is probably because in my design vehicles always take actions based on the **latest** messages they receive. Since there are multiple globes present, a vehicle may receive completely different globe information from different sources. And this is the reason why you see them keep “shaking” … and this somehow affects their decision to what to do.

With initial size of 64 the performance is much better:

Time Past	No. Of Vehicles Alive
5 min	~64
10 min	~63

Stage d

I have run my system serval times under stage d and most of time my system can shrink the size to a specific one, but sometimes it does not (~2 vehicles margin of error) when initial size is huge (say, above 100).

Defect

Why there is ~2 vehicles margin of error? This is because, in my design the leader vehicle first includes vehicles which should be reserved to the set `Reserved_Vehicles`. Once it includes enough number of vehicles, it starts to send messages to exclude (kill) those which are not within the set `Reserved_Vehicles`. However, the system may run into the case where vehicles which are already included die beforehand. This case therefore leads to the system over-shrinks the size.

Another defect is that I have no answer if the leader vehicle dies … though from all my tests I have not seen the leader vehicle died (but it is definitely possible).

Nature of Message Passing

As I mentioned before I will talk about the full story how to let all vehicles agree on one leader vehicle. This is tricky but essential. The first interesting fact is that, at the start of simulation all vehicles will not send any messages until someone finds the globe. The second is that, the vehicle which first sends the message will be the leader. That is, the `Last_Msg` of all vehicles will eventually be overridden by the **oldest** one (i.e. the message that is sent by the vehicle which first finds the globe) when vehicles constantly sending and receiving back and forth. Therefore, all vehicles will eventually agree on one vehicle — the leader.