Australian National University

GLIDING IN SPACE

Assignment for all students of Systems, Networks and Concurrency 2017

This is a carefully evaluated and marked assignment. Thus extra care is expected on all levels.

Overview

Coordinating behaviours and resources in a concurrent and distributed way is at the heart of this course. The background this year is a swarm of physical vehicles in 3-d space.

The code framework which will model and simulate a swarm of vehicles showing default behaviours which keep them in motion, together as well as collision-free is provided to you. All vehicles have a local “charge” to keep them “alive”. For physical reasons yet unknown, the vehicles replenish their charge to full by passing “energy globes” in close proximity. Vehicles which run out of charge mysteriously disappear. Vehicles constantly consume a little bit of energy to keep their on-board systems running, and consume more substantial amounts of energy when accelerating or decelerating.

Sensors, Actuators & Communication

Each vehicles is operated by a dedicated task which has access to local sensors, communication interfaces and actuators.

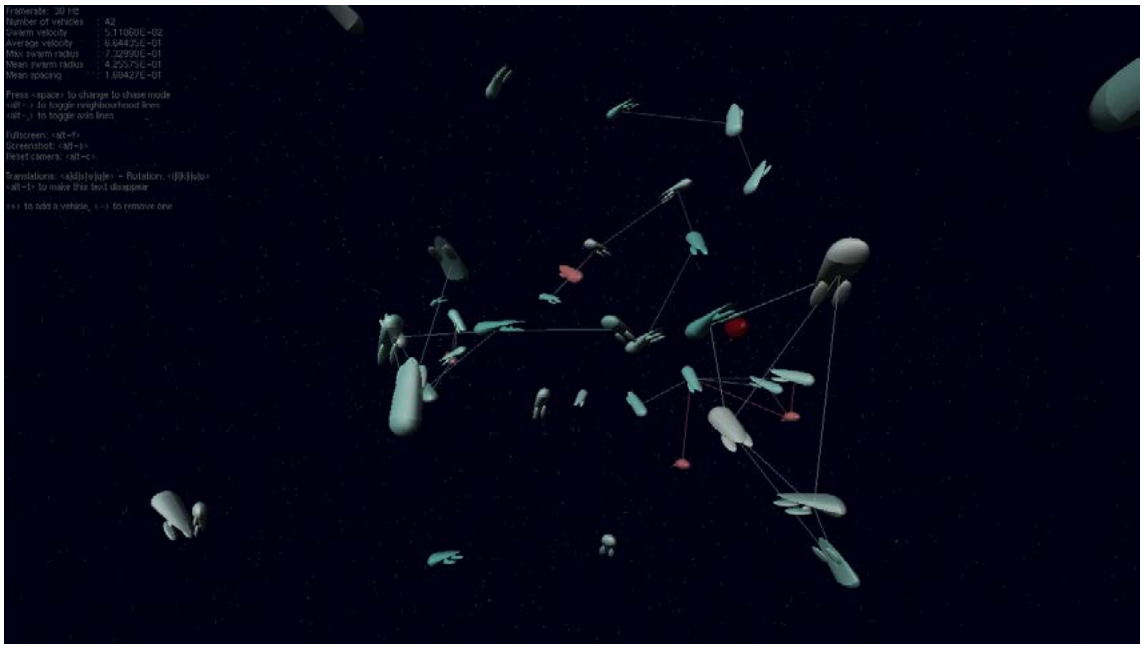
The sensors include position, velocity and acceleration as well as current charge. If the vehicle is close enough to one or multiple energy globes to utilize them, the sensors also display the position and current velocity of all close energy globes.

The actuator system consists of setting an absolute destination position and throttle value. The underlying cruise control systems automatically handles the steering and acceleration. Once the destination point has been reached the throttle automatically switches back to idle which means that default swarming behaviour takes over. The vehicles do not slow down when approaching the destination point and rather pass through the destination point. This helps to keep the controls fluent and the vehicles in motion. Collision avoidance reflexes are always active and prevent vehicles from crashing into each other. Note that destinations might become unreachable, if multiple vehicles are bound for the same destination.

The vehicles are also equipped with a message passing system which allows to broadcast a message which will be received by all vehicles in close proximity. This is asynchronous and there is no feedback whether any message has been received by any vehicle – unless another vehicle actively sends a message in response of course.

Finally there is also a function which allows direct access to the underlying, secret clock of the world. `Wait_For_Next_Physics_Update` will put the task to sleep until anything actually happened (which includes communication). This relieves the vehicles from busy waiting on the world to change.

All the above controls are found in `Sources/Vehicle_Interface/vehicle_interface.ads`.



The Animation

The provided graphical animation of the swarm offers third person views as well as the view from one of the vehicles while it is passing through the swarm. The communication range can be visualized by drawing connecting lines between all vehicles which are currently in communication range. The colours represent their charging state as well as the control state. Turquoise vehicles are currently following their swarming instincts are not explicitly controlled by the associated task. The colour saturation reflects the level of charge. Once vehicles go into manual control (throttle and destination is set) they turn to a more red colour schema. The energy globe(s) are dark ruby coloured spheres.

Design Constraints

The final solution which is requested from you should be deployable on actual vehicles. Hence only the provided interfaces to physical sensors and forms of communications are allowed. "Looking underneath the hood" is still encouraged of course if you want to see how such a simulation can be implemented on a computer system. Bypassing the provided interfaces and using information from inside the simulator is obviously counterproductive for any future physical deployment and hence not allowed. Nevertheless the first stage of the assignment will allow you to introduce additional means of communication which can not necessarily be physically implemented.

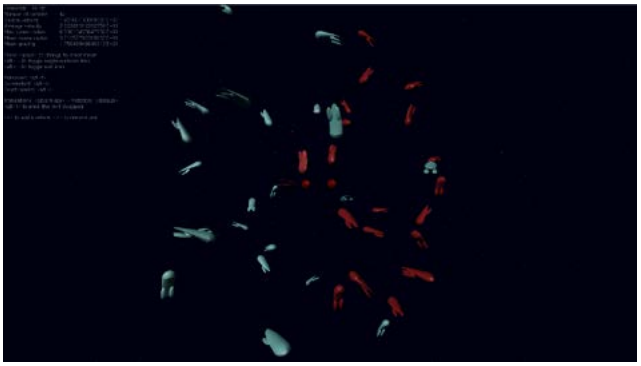
Timing Constraints

All calculations inside the vehicle tasks have an implicit deadline given by the update from the underlying physics engine. This deadline is not hard as seen by the local tasks, yet many tasks overrunning this deadline concurrently will slow down the simulator. Simulated time is not affected by this – only the update time intervals will become larger.

Design Goals

The overall design goal is to keep as many vehicles alive as possible for as long as possible. As energy globes can only be discovered locally, communication is required, as well as coordination between the vehicles as all swarm members heading for the some destination at the same time will not lead to many of them making it there.

The task can be solved in four stages:



a. Allowing a central coordinator.
b. Fully distributed.
c. Multiple energy globes.
d. Going full speed.

Stage *a* still allows for a central coordinator to be introduced and all tasks are allowed to communicate with this entity (or multiple thereof). The implementation of those central instances can employ shared memory based as well as message based forms of communication. Some are obviously questionable to impossible in a physical deployment of your system, yet this stage might help you to develop ideas which can then be considered for the second stage.

Stage *b* does not allow for a central coordinator and all planning and scheduling now needs to be done on the individual vehicles only using local communication. This is hard. If you are confident that you are up for stage *b* straight out then you do not need to implement stage *a*.

Stage *c* requires further coordination between vehicles as multiple energy globes are to be considered. Assume that you do not know how many globes are in existence, yet utilize the additional redundancy which is detected at runtime to enhance the overall robustness of your swarm charging method. To test this you will need to go to the package `Swarm_Configuration` and change `Configuration` to `Dual_Globes_In_Orbit`.

Stage *d* will test your algorithm to the limit and speeds up the energy globes. Without a well coordinated swarm message exchange design, chances are dropping drastically to hit an energy globe. To test this you will need to go to the package `Swarm_Configuration` and change `Configuration` to `Dual_Globes_In_Orbit_Fast`.

Criteria

The first criterion is to get over the initial phase without anybody dying. This can be a impossible, if still nobody found an energy globe before the initial charges run out. Don't worry about this case – this is just nature and you cannot do anything about it.

Yet once one or multiple globes are found you need to make sure that the information is spread effectively.

Now comes the real challenge of how to coordinate the vehicles. Find ways to coordinate their paths. This might be leading to different strategies in stage *a* and stage *b*.

The Programming Framework

The provided code has been successfully compiled and tested on the lab computers and:

- **Linux**: Ubuntu version 10, 14.04, and lab computers. Depending on the Linux version, it might be necessary to install glutg3 and glutg3-dev (via a package manager if you are lucky).
- **Mac OS**: PowerPC and Intel (32 and 64 bit), tested on 10.5, 10.6., 10.8, 10.10 and 10.11. No further installations should be necessary.
- **Windows**: tested on XP, Vista, Win7 and Win10. The freeglut library needs to be in the same directory as the executable (already placed to the right spot in the provided project).

Please note the different project files which refer to different operating systems and select the one fitting your computer when opening the project.

Parts of the provided framework are based on the Globe_3D project (an OpenGL 3D engine), which is maintained by Gautier de Montmollin.

Deliverables

You need to submit a report (in pdf) as well as your code (only the source files inside the directory `Student_Packages` - please do not submit the whole framework or your binary files). The report is your chance to convince us that your concept is great – even though your code might not have run according to expectations. Conversely, even though your code might be a strike of genius, we might not have recognized it without your great report along with it.

Report

- Documentation of your design. Specific emphasis should be given to explain your design decisions. Give reasons for each of those. Make clear which constraints you employed as 'driving concepts', which have been considered, and which have been purposefully ignored (for instance to allow for a cleaner, easier maintainable design).
- Provide documentation of your test runs. Give a precise motivation for each of your tests.

The following questions might help you to evaluate and describe your design:

- How does your design scale?
- Do you provide for graceful degradation in case that parts of your system become unresponsive (a mysterious vehicle disappearance case)?
- Do you consider your design dependable and maintainable and why (or why not)?

If you require help in technical/scientific writing, please do not be too shy to ask for it.

Code

Submit only the manipulated packages (vehicle task & message structure) together with the packages which you added on top. Your code will evaluated according to common professional practice. We do not enforce a specific coding schema, but request consistency and a general high standard on the basic coding level. Make sure all your identifiers have good names, all scopes and access constraints are set as tight as possibly, and full use has been made of compile time checks. All your code will be read. We do not have the capacity to provide detailed comments on all code, but will refine the provided feedback on individual request in the limits of our time.

General

Use graphical or any other means to express your ideas as precisely as you can. Overall assignment time is five weeks (which includes two weeks of semester break), so we expect a work of precision and care. Exact due date and submission procedures will be announced on the website and forums.

Guidelines for the marking are:

- (30%) Functionality of your program - does it implement stage *a, b*, *c or d*?
- (30%) Elegance and strictness of your design.
- (30%) Clarity of the report.
- (10%) Provided documentation about test runs.

Expect to be limited to a credit (CR) range marks if your design only considers stage *a* solutions and to find yourself in high-distinction (HD) range if your design is a convincing and compete submission for stage *d*. Allow yourself plenty of time to come up with a solid concept first. Without a clear idea you are bound for chaos in this assignment.

This is only a guide. We will feel free to give you more marks if your code is outstanding or your report is an outstanding example of technical writing.