“Colony Ship” Activity Manual

[Fall 2018]

Introduction

Adventure and glory await you! With the recent discovery of the galactic warp gate network, an international coalition of world governments has been hard at work designing and constructing humanity’s first interstellar colony ship. Bound for Kepler-438b, an exoplanet only 472.9 light-years away from Earth in the constellation Lyra, you and your colleagues have been tasked with developing the software control systems that will guide the ship through the hazards of deep space. The fate of thousands of pioneering souls rest on your collective ability to design, implement, and test your code and work as a team!

Image Source: https://exoplanets.nasa.gov/alien-worlds/exoplanet-travel-bureau/

Contents

Your Mission

The ship will start near Earth within our local star system. Each star system acts like a level in a video game. The goal in each star system is to get to the next warp gate whilst avoiding getting hit by asteroids and other hazards. The over-arching goal of the endeavor, is to get to the habitable exoplanet in the Kepler-438 star system.

Every star system contains gravitationally significant bodies such as stars, planets, warp gates etc. as well as smaller objects such as asteroids and debris which are collision hazards that the ship must either avoid or destroy.

While each star system has its own local coordinate system, each star system is itself situated within the larger galactic coordinate system. The two coordinate systems are independent of each other and are used for separate purposes (i.e. navigating around debris within a star system and navigating between warp gates within the larger galaxy.)

First and foremost, the success of your mission is determined by whether or not you are able to reached Kepler-438b. Secondary objectives include minimizing how much damage the ship sustains during the voyage and minimizing the fuel cost of the warp gate path you follow through the larger galaxy.

To achieve these objectives, the ship's functions have been organized into four subsystems. Each subsystem corresponds to one development team and each student team is required to create a control algorithm for its assigned subsystem. A brief description of each subsystem follows:

Sensors – Observe and report nearby phenomena. The Sensors team will receive data from the ship's sensors about the current star system and must interpret, organize and format the information for other subsystem teams to use.

Navigation – Map a path through the galactic warp gate network towards the ship's destination. Navigation must keep track of the warp gate network and inform other subsystems which warp gate the ship should head to next.

Propulsion – Control the ship's thrusters and maneuver towards your destination. Propulsion is responsible for navigating within individual star systems (i.e. avoiding obstacles and moving from one warp gate to the next). Once the porpulsion team has been told where to go within a star system, it's up to them to figure out how to get there and directly control the various thrusters outputs.

Defence – Protect the ship from rogue space debris using a defensive torpedo turret. After receiving relevant sensor data, your challenge is to determine whether, when, and where to launch torpedoes.

Any one subsystem cannot work in isolation. Teams must dynamically pass each other information at run time in order to succeed. For instance, Defense won't know where the threats are if Sensors doesn't provide them with information about the ship's surroundings.

You have all been provided a class for each subsystem and a simple API. It is up to you to decide how and what information will be passed between subsystems and how each subsystem will perform its duties.

Workflow and Schedule

The overarching schedule of the activity will follow this pattern:

Introduction > Specification > Development Iterations > Sea trials

Following the introduction of the activity and an overview of the activity's goals (likely what's happening as you read this), the entire class will be split into ships. Each ship will be further divided into Subsystem Teams. You are encouraged to read this entire manual an familiarize yourself with at least the high-level details of every ship subsystem, even if it is not the one you are directly assigned to, as you will benefit from knowing what other teams are dealing with and how they might better interface with your subsystem.

During the **specification** stage, teams will design on paper a solution to their subsystem's problems, they will layout everything their system needs and everything they might need to pass to other subsystem teams. They will not code anything during this stage. Before entering the Development phase, you should have your subsystem specification reviewed by at least one other subsystem team within your ship. Only after these steps have been completed should coding and simulation cycles begin.

During the **development** stage, you will implement your specified solution and be able to test it live within a local simulation on your own development machine. Depending on what subsystem you are working on, you may need to mock up (or request that the relevant subsystem team) mock up a dummy solution to temporarily fill in the gaps for other subsystems that your work depends on but that are not yet completed.

Besides solving your own subsystem problems, you will likely want to be in regular contact with the other subsystem teams that are part of your ship as all of your solutions evolve as previous assumptions change. It is likely that the interface between subsystems will change several times and you will need to coordinate those changes lest your colleagues end up wasting time working on broken/outdated code.

Eventually, once all of the subsystem teams on your ship feel their solutions have sufficiently progressed, your ship can request to run a "Sea Trial": the instructor will pull all of your ship's subsystem controller code down onto a public computer and run a simulation using a new galactic map for the entire class to observe and learn from. In theory, naive or hard-coded solutions run on your local systems will not fare well when faced with new galactic terrain and hazards. Rather, robust and dynamic solutions that can better account for some uncertainty are preferable.

Although this activity is not meant to be competitive, the instructors will be making note of which ships are able to reach certain milestones first (e.g. leaving the first star system, passing through a hazard unscathed) and "bragging rights" are only meant to serve as minor entertainment incentive for some. However, the more important purpose of holding public Sea Trials is to allow all teams an opportunity to observe , speak to, and learn from other ships' teams over the course of the whole activity.

SourceTree and Git

Git is one of the most popular modern "version control systems" and is used around the world to coordinate work from multiple collaborators on distributed projects. (Though it also make an

excellent backup and distributed work system even when working alone.) It will be used in this project to separate each ship's work (via forks) and each subsystem team's development (via branches) as well as facilitate easily pulling everything together into a "release" version for "Sea Trials".

SourceTree is a GUI application we will be using to interact with our git repository. Although Git can be controlled entirely through console commands (and therefore automated for more sophisticated uses), using SourceTree for this activity will help to visualize the git workflow and help new users come to understand git terminology such as "pushing", "pulling", "branching", and "merging".

GitLab is a service that hosts Git repositories. (For example, you may have heard of other popular Git hosting services such as Github and Bitbucket.) For this activity, we will be using the GitLab service that the University of Waterloo runs for its students at http://git.uwaterloo.ca. You should be able to log in using your WatIAm user id and password.

Git Primer

Some vocabulary:

Repository – a data structure representing your entire project, all the files within it, and their complete history of committed changes.

Clone – creating a copy of a repository that lives on your computer instead of, for example, on a website's server.

Branch – a parallel version of your project that you can work on without interfering with other branches. Typically, you will working in your own "development" or "feature" branch and then commit changes back to a "master" branch that the rest of your team pushes/pulls from. Sophisticated projects can have many dozens of branches, each serving specific purposes, such as prototyping new features, fixing old bugs, code freezes in preparation for releases, maintenance, etc.

Commit – saving a change to a file from your local copy to your repository. It will record what changes are made and who made them. These should be done with a brief message explaining what has been done

Merge – takes the changes from one branch and applies them to another (hopefully, but not always, seamlessly)

Push – sending committed changes from your local repository to a remote repository

Pull – fetching changes and merging them to the file you're working on

Getting started:

1. Make sure SourceTree is installed and make an Atlassian account
2. Go to https://git.uwaterloo.ca
3. Select the activity's repository/project and copy the link it provides
4. Open SourceTree and select file -> clone/new
5. Paste the link you copied earlier and give the project a location on the local hard drive to clone a local copy of the repository to.
6. Go to Remotes -> origin then select the branch corresponding to your team

Each team will correspond to a development branch relating to their subsystem, this is where the instructors will pull from so make sure this code works. You may branch off from your own team's branch as much as needed, just make sure these branches are merged with your team branch before Sea Trials if you want the code tested.

The Unity Engine

We'll be using the Unity game engine for this activity. Unity is C# scriptable and has many features relating to 2D and 3D game making as well as an extensive UI with some drag and drop functionality. It will handle rendering, physics, and the main game loop for the activity.

Within the game engine, there are entities called game objects. The ship is one of these so called game objects and therefore has several properties such as a position in the world space and is able to accept game object components. Many kinds of such components are included with Unity, but you can also make your own using C#. The ship game object has a custom Ship component that will call the update methods you will be writing. You won't be using the Unity engine directly in any way save for some math related functionality. The Sandbox namespace that is provided at the top of your scripts will provide you with what you need to complete the activity.

Another aspect of the Unity game engine you need to know, is that it makes use of so called scenes. Effectively these are something like different levels in a game. They contain all the relevant game objects to a particular situation. For this activity, each solar system has its own scene. You may choose to place a ship into an individual scene and test the ship in that solar system in isolation. To see a full run of the entire trial you must run from the game core scene. The game core scene can be found in the assets folder in your project window. Make sure to remove any instances of the ship game object in all the other scenes as the game core will generate its own instance of the ship.

The API

You will have access to certain libraries and namespaces specified at the top of each subsystem script file. ***You are not allowed to change these.***

Each subsystem class has an update method called "**`SubsystemNameUpdate`**". (E.g., for `Defence` this method will be called **`DefenceUpdate`**. For `Sensors` it is called **`SensorsUpdate`**). These update methods are each called once in the main game loop which runs several times per second. Keep this in mind when writing your code. The order of execution is as follows:

```
Sensors.SensorsUpdate();
Defence.DefenceUpdate();
Navigation.NavigationUpdate();
Propulsion.PropulsionUpdate();
```

Variables declared within the update methods won't have their old value during a new method call. The time in seconds between method calls can be accessed like this:

```
float timeBetweenMethodCalls = SubsystemRef.deltaTime;
```

More information about the ship as well as an instance of each subsystem class is kept inside of a **SubsystemReferences** object on the ship. Because this is passed into your **SubsystemUpdate** method, you can access any public methods, fields, structs, or classes defined in any other subsystem class through that reference. For example, if Navigation wants to call a method in Propulsion's class called `SetDestination(Vector2 destination).` The code on Navigation's side would look like this:

```
SubsystemRef.Propulsion.SetDestination(newDestination);
```

Remember this is only an example. It's up to you and the other teams to agree how information will be passed.

Here's the complete list of fields in the `SubsystemReferences` class that you can access.

```
// References to all the subsystems

Sensors Sensors

Defence Defence

Navigation Navigation
Propulsion Propulsion


Vector3 solarPosition;  // the position within local space of the star system

Vector3 velocity; // current velocity of the ship

Vector3 forward;  // unit vector that points in the ship's forward direction

Vector3 back;     // opposite of forward

Vector3 right;    // unit vector pointing right

Vector3 left;     // unit vector left

Vector3 inwards;  // unit vector pointing into the screen

Vector3 outwards; // unit vector out of the screen

float deltaTime;  // time in seconds between method calls

string solarSystemName; // name of the solar system the ship is currently in
```

Note: `Vector3`objects can be easily cast to `Vector2`. The z component (the axis pointing into the screen) is simply dropped. For more information about `Vector2` and `Vector3`, library look up the relevant Unity API documentation. For math

Sensors Subsystem Controller

You are the eyes and ears of the starship. Your job is to provide the other subsystems with information about the world so that the ship can reach its destination whilst avoiding hazards. You are also tasked with identifying the habitable exoplanet, Kepler438b. Keep in mind that this planet is necessarily within the star system Kepler 438.

The `Sensors` subsystem is unique in that it must provide information to many other subsystems. Something that might help complete the task is to assign each member of your group to work on providing only one subsystem with information e.g. Saleem and Erik will write the code that handles information for `Navigation` while Maddy makes an interface for `Defence`. Use git to combine your code into one script. You may choose to simplify your code or remove redundant sections later on.

Your work will be called through the **`SensorsUpdate`** method provided in the `Sensors` class. Just like all `SubsystemUpdate` methods, it will be called several times per second. Keep this in mind when implementing your code.

This method receives two parameters. The first is the **`SubsystemReferences`** parameter and is common to all `SubsystemUpdate` methods. Refer to the API section of this manual to learn more about all the information provided in this parameter. The other one is the **`Data`** parameter of type `ShipSensors` and is unique to your subsystem. This is how you get a handle on the ship's sensors' data. The ship has two sensors: a long range gravity wave interferometer (GWI), and a limited range electromagnetic sensor (EMS). A section about each will follow.

The Gravity Wave Interferometer

In the real world, gravity wave interferometers use interfering wave patterns of high energy laser light to detect minute contractions and expansions of space due to gravity waves.

For this activity, the ship's GWI detects waves from gravitationally significant bodies in a star system such as warp gates, stars, planets, moons, etc. It has no range limit; that is it detects any and all gravitationally significant objects within the star system. Access the GWI through the `Data` parameter like this:

```
Data.GWInterferometer
```

The information about any object detected by the GWI is stored within a `struct` like this:

```
public struct GWI_Detection
{
        public float angle;
        public float waveAmplitude;
        public GravitySignature signature;
       (public string warpgateDestination;)
}
```

The **`angle`** tells you the direction the gravity wave came from relative to the global X-axis; counter-clockwise being the positive direction. The **`waveAmplitude`** is related to the distance between the object and the ship by this formula:

```
waveAmplitude = Data.GConstant / distance
```

The **`signature`** of the wave will most often let you know what kind of object was detected. The different kinds of gravity signatures are saved inside an `enum` called `GravitySignature`. Checking a given `objectSignature` might look something like this:

```
bool isPlanetoid = objectSignature == GravitySignature.Planetoid;
```

You may also chose to convert to a string like this:

```
bool isWarpGate = objectSignature.toString() == "WarpGate";
```

The exoplanet, Kepler438b, will have a planetoid signature.

If the object is a warp gate the detection will also include a `string` with the name of the star system the gate leads to. This property is called **`warpgateDestination.`**

The sensor information is saved in a public `List` (think dynamic array) of `GWI_Detections`. Access it through the given **`Data`** parameter of type `ShipSensors` for your main method, `SensorsUpdate`.

`Data`**`.GWInterferometer`** is the `List`.

`Data.GWInterferometer`**`[0]`** is the first detection.

`Data.GWInterferometer[i]` is the detection at index `i` within the `List`.

`Data.GWInterferometer.Count` returns the number of detections in the `List`.

The Electromagnetic Sensor

The EMS is something of an all-purpose electromagnetic radiation detector, like a really sophisticated telescope that can perform analyses on blue/red shift, spectroscopy, and brightness at high speed. It provides information about all objects within a set range of the ship (e.g. 30m). This sensor should be used to detect smaller space objects such as asteroids and space debris. The EMS will also detect the gravitationally significant objects that the GWI detects, but only if they are within range.

`Data.EMSRange`

Each EMS detection is similar to the GWI detections as it comes in a struct like this:

```
public struct EMS_Detection
{
        public float angle;
        public float signalStrength;
        public Vector2 velocity;
        public int materialSignature;
}
```

The **`angle`**, much like the angle in GWI detections, indicates the direction the detection came from. The **`signalStrength`** is analogous to the `waveAmplitude` in the GWI detections. It is related to the distance of the object relative to the ship by the following formula:

`signalStrength = Data.EMConstant / distance`

The **`velocity`** indicates the velocity of the object. The **`materialSignature`** is a result of the sensor's spectroscopy capabilities. An object can be comprised of various materials. As such, you can't check this kind of signature the way you would a gravity signature. To check an objectSignature for a known material from the `SpaceMaterial enum`, use the following method:

```
bool Data.CheckSignatureForSpaceMaterial(int objectSignature, SpaceMaterial testMaterial);
```

Will return `true` if the given objectSignature contains the given `testMaterial`.

The exoplanet, Kepler438b, will have contain `SpaceMaterial`.`Water`, `SpaceMaterial`.`Common`, and `SpaceMaterial`.`Metal`.

As with the GWI, the EMS data is accessed through the **Data** parameter of type `ShipSensors` like this:

`Data.`**`EMSensor`** returns the `List` of EMS detections.

`Data.EMSensor`**`[0]`** is the first detection.

`Data.EMSensor`**`[i]`** is the detection at index **`i`** within the `List`.

`Data.EMSensor.`**`Count`** returns the number of detections in the `List`.

Defence Subsystem Controller

Keep the ship alive! Your team controls the turret system and it is your responsibility to destroy hazards and prevent the ship from receiving critical damage. Identify threats to the ship and calculate where to aim the turret to intercept oncoming debris.

The problem you must solve breaks down into the following key objectives:

- Design an information package that you can use to find targets
- Chose a target among the available choices

Just like every other subsystem team, you are provided with a subsystem class, `Defence`, and a corresponding `SubsystemUpdate` method called **`DefenceUpdate`**. It will be called several times per second along with the other `SubsystemUpdate` methods. Keep this in mind when implementing your code.

The `NavigationUpdate` method gets two parameters. The first is the **`SubsystemReferences`** and is common to all `SubsystemUpdate` methods. You can access the ship's position and velocity through this parameter among other useful information.

```
Vector2 shipVelocity = systemReferences.velocity;
```

```
Vector2 shipPosition = systemReferences.solarPosition;
```

Refer to the API section of this manual to learn more about all the information provided in this parameter. The other is unique to the `Defence` subsystem and is your handle on the ship's turret. **`TurretControls`** has only one field which is the **`aimTo`** position.

```
turrentControls.aimTo = nextTargetPositionVector;
```

The cannon will automatically fire once every second. Every time it fires, the turret will check what position you have given it to aim at and fire a missile towards that position. Keep in mind that the torpedoes you fire will take time to travel to their targets and you can only fire so frequently. To access the missile speed use the following reference:

```
Ship.missileSpeed
```

You will need to prioritize threats to be more effective, as you will not be able to shoot everything! You may consider firing at targets that are headed for the ship, closer targets, faster targets etc.

Navigation Subsystem Controller

Without you, humanity is quite literally lost. You must use your knowledge of the stars to guide the ship to its destination. For the ship to jump to a particular star system it must reach the warp gate that will take it there. The Sensors team can provide you with warp gate destinations in the form of star names.

Just like every other subsystem team, you are provided with a subsystem class, Navigation, and a corresponding SubsystemUpdate method called **`NavigationUpdate`**. It will be called several times per second along with the other `SubsystemUpdate` methods. Keep this in mind when implementing your code. The `NavigationUpdate` method gets two parameters. The first is the **`SubsystemReferences`** and is common to all `SubsystemUpdate` methods. Refer to the API section of this manual to learn more about all the information provided in this parameter. The other is called **`StarInfo`** and is of type `GalaxyMap`. `StarInfo` contains a `List` of stars that look like this in code:

```
public class GalaxyStar

{

    public string name;

    public Vector2 galacticPosition; //using galactic coordinates
}
```

As you can see, each star has two properties available to you: the star's name, and its coordinates in galaxy space. This is different to solar system space, which all other groups will be using for their calculations. Don't confuse the two coordinate systems. The `GalaxyStar` objects do not exist within the context of any solar system. As the navigation team, you'll be operating on a higher decision making level.

You might want your strategy to resemble something like the following steps:

1) Generate a graph of stars acting as nodes that can be connected based on StarInfo
2) Update said connections based on the warp gates in the current star system
3) Calculate most promising path to the destination system
4) Determine which warp gate will take you to the next star along that path

Alternatively you might want to set up a few simple decision making rules. Here are some examples:

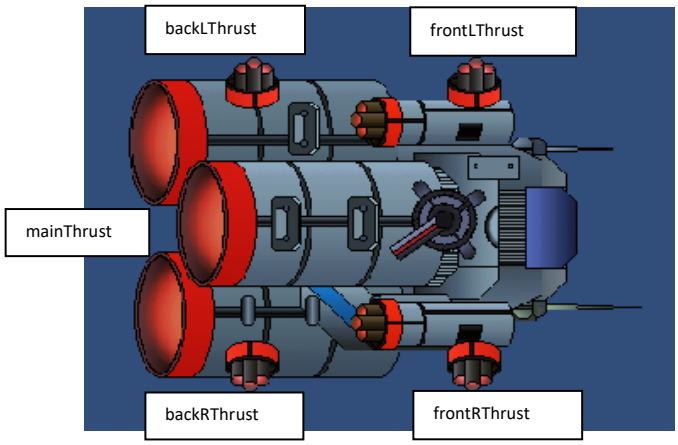
- Don't go to a star system we've already been to unless we have no other choice
- Always go to the second closest warp gate
- Travel to a random warp gate
  (WARNING: some warp gates will lose the game if passed through)

How you go about steering the ship is up to you. Good luck!

Propulsion Subsystem Controller

You control the ship's thrusters. Get the ship to the next warp gate to make it to the next star system. Try to avoid veering too far off course or spinning out of control.

Just like every other subsystem team, you are provided with a subsystem class, PropulsionSubsystemController, and a corresponding `SubsystemUpdate` method called **`PropulsionUpdate`**. It will be called several times per second along with the other `SubsystemUpdate` methods. Keep this in mind when implementing your code. The `PropulsionUpdate` method gets two parameters. The first is the **`SubsystemReferences`** and is common to all `SubsystemUpdate` methods. Refer to the API section of this manual to learn more about all the information provided in this parameter. The other is unique to the `Propulsion` subsystem and is your handle on the ship's various thrusters.



`ThrusterControls` has five fields of type `float` – one for each thruster: a main thruster at the back and four side thrusters for steering. Apply the correct amount of torque to orient the ship to your destination (Tip: consider looking into "PD control").

```
thrusterControls.mainThrust = (your float);
thrusterControls.portBowThrust = (your float);
thrusterControls.starboardBowThrust = (your float);
thrusterControls.portAftThrust = (your float);
thrusterControls.starboardAftThrust = (your float);
```

Here are some numbers you might need for calculation purposes:

The four thrusters are placed at (± 0.75m, ± 1m) relative to the ship's center. The X-axis being oriented along the port-starboard direction and the Y-axis being oriented along the aft-bow direction of the ship.

The main thruster has a limit of 100N force output. The four maneuvering thrusters have a limit of 50N force output. The ship has a mass of 100kg. That said, this is a physics simulation and not all

Example Communication Path

The below diagram is meant only as an example of how different ship subsystems might communicate with each other. It is not exhaustive nor is it the *only* way to organize your flow of data. Ultimately, you must discuss with the other subsystem teams to decide who is going to provide what information to whom (and when/how).