Space Ship Collaboration Student Manual

UWaterloo Ideas Clinic*

June 30, 2020

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 ships. 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!*

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

*Faculty & Staff: Derek Rayside, John Harris, Chris Rennick, Sanjeev Bedi. Co-op Students (alphabetical order): Taha Aziz, Mathew Del Rosso, Joanna Diao, Francisco Garcia-Gonzalez, Kaeun Kim, Margaret Malton, Nirosh Manohara, Zach Radford, Derrick Wang, Maggie Wong.

Contents

1 Engineer an AI to Fly to Kepler-438b!

Figure 1: Mission overview

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. Your spaceship has four subsystems:

Subsystem	**Description**	**Metrics**
Defence	Fire torpedoes at asteroids and debris to keep the ship safe. Needs info from Sensors to aim.	Kepler-438b Damage Torpedoes
Navigation	Plot course through galaxy. Decide which warp gate or planet to go to. Needs info from Sensors.	Kepler-438b Jump cost
Propulsion	Use thrusters to orient and propel ship towards destination. Needs destination from Navigation.	Kepler-438b Damage Fuel
Sensors	Interpret gravitational waves and electro-magnetic sensors to provide info to Defence and Navigation.	Kepler-438b Energy

Figure 2: Subsystem overview

1.1 Learning Objective: Teamwork

The subsystems need to work together in order for the ship to reach Kepler-438b. Making this happen will require teamwork in several dimensions:

- *Interpersonal.* Listen. Be nice. Many people want to show off their skills and knowledge. That's ok. But do so in a way that does not denigrate your classmates: also find ways to celebrate their strengths. You are now in the big leagues: everyone here has skills and abilities, but those might not be immediately obvious to you. Pay attention and find something nice to say. You are all in this together, and you will be together for the next five years. You are here to collaborate with your classmates — not to compete with them.
- *Version Control (Git).* Version control is a foundational technology for teamwork in software.
- *Division of Labour:* How to organize the work to ensure that the subsystems integrate. A common way that software systems failed in the twentieth century is that all of the subsystems could perform their functionality independently, but couldn't work together. Modern agile design techniques focus on integration and communication first, functionality second.

1.2 Working Outside Your Comfort Zone

This activity is designed to push you outside your comfort zone. By definition, that will feel uncomfortable. This activity is also designed to be fun. Welcome to the complex world of feelings!

Being able to work outside your comfort zone is a skill — a skill that few people in the world have. You were admitted to Waterloo, in part, because you have the potential to develop this skill. This activity is designed to help you realize your potential — and to realize that you have this potential.

Highschool was within your comfort zone. Maybe, at times, it enlarged your comfort zone — but it did that by pushing the boundary forwards from the inside. Now we are going to take the warp gate to the other side. Hold on!

	Highschool	**Spaceship Collaboration**
Pre-requisite Knowledge	Carefully provided	Uses ideas from classes you haven't taken yet.
Time	Just enough to do it perfectly.	Not enough.
Programming Environment	Taught to you.	Learn on your own.
Classmates	Not as talended as you.	Very talented.
Team Size	2–4	~ 16

Figure 3: Some ways in which this activity is outside your comfort zone

1.3 How to Manage Feeling Lost or Overwhelmed

You might feel uncomfortable during this activity. Don't worry! These are normal feelings. This activity is a safe space for you to explore managing these feelings. These feelings are very common amongst high-achievers — even amongst high-achievers over thirty years old! The key is to learn positive strategies to manage these feelings so that they don't ruin your fun and impede your progress.

We're not ready to demo! That's a 20th century feeling. It's time to let it go and embrace the century you were born in. The industry has learned from vast experience that this flavour of perfectionism does not produce the best software in most cases. There are two reasons people typically have this feeling: (1) the software is not integrated, and (2) the features are incomplete. A key insight of the late 20th century is that if we integrate first, then the system is always ready to demo — maybe some of the features still need to be improved, but the system does something. See §2 to learn the right technical skills to work together this way. Integrate first, then your team will always be ready to demo.

Lost ... Have no idea what to do? You're feeling lost. How are you going to manage this feeling? You have choices.

Negative:

- Run away! Start fiddling with your phone/computer.
- Attack others. Maybe if you can make them feel vulnerable by attacking them then they won't notice that you feel vulnerable.
- Attack yourself. See *Imposter Syndrome* below.

Positive: Find someone to talk to. Perhaps a teammate. Perhaps a classmate on another team. Perhaps a TA or instructor. They can help get you started. We all feel lost sometimes.

Overwhelmed! Too many things! What to do next?!?! You're feeling overwhelmed. You could choose a negative behaviour described above, or you could choose a positive strategy, such as:

- Start with something small and achievable.
- Reflect on your role on the team, and use that to focus your efforts.
- Ask someone who depends on your work what they want to prioritize.

Imposter Syndrome? Do you really belong here? If you have doubts, then you are probably feeling *imposter syndrome*. (If you really wanted to study Economics or English but your uncle convinced you to come here instead, that's a different issue — chat with your academic advisor.) Imposter syndrome is a thing that high-achieving people often suffer from: a feeling that you aren't good enough or don't belong — despite objective evidence to the contrary. You belong here. We admitted you. Our admissions processes are finely tuned by decades of experience. You can read more about how to manage these normal feelings in *Harvard Business Review*:
`https://hbr.org/2008/05/overcoming-imposter-syndrome`
`https://en.wikipedia.org/wiki/Impostor_syndrome`

1.4 Feelings Beyond This Activity

The previous discussion is about feelings you might have within this activity — feelings that this activity might provoke in you, by design. In that discussion, *find someone to talk to* means people doing the activity with you.

You might also have these kinds of feelings outside of this activity. Some degree of these feelings is pretty common in first year generally. Going to university is a big adjustment. That's normal.

There are many resources to help you learn how to manage these kinds of feelings. Talking to friends, classmates, family, older students, *etc.*, is always a good first step. The next step is to participate in UW Counselling Services' workshops:

`https://uwaterloo.ca/campus-wellness/counselling-services/seminars-and-workshops/coping-skills-seminars-online`

You can also speak with your academic advisor, in the First Year Engineering Office or in your home program. Here are some helpful links:

`https://www.engsoc.uwaterloo.ca/resources/mental-health/`
`https://uwaterloo.ca/engineering/current-undergraduate-students/engineering-counselling`
`https://uwaterloo.ca/campus-wellness/`

1.5 Assessment

Your instructor will tell you about the specifics of your assessment. However, the performance of the spaceship is strictly off limits for formal assessment: that's just for pride & glory. The technical ideas in this activity will be taught to you in future courses, and you will be evaluated on them properly at that time. For now we are just having fun with the technology and ideas while we build our teamwork skills.

Marks	**Pride & Glory**
• personal reflections • version control quiz • teamwork quiz • design case studies	• landing on Kepler-438b • minimizing ship damage • minimizing torpedoes spent • minimizing warp costs • minimizing fuel used • minimizing sensor energy • clever algorithms • blowing everything up along the way • looking good while doing it!

2 Team Design Methods

There are different ways in which a team can be organized to work on an open-ended design task. Experience has shown that some techniques work better than others. Understanding how to best organize the team's work and communication can improve your chances of success.

2.1 Traditional Design: Divide & Conquer

Divide & Conquer was made popular by the Roman emperor Julius Cesaer over two thousand years ago. It worked for him as a military strategy. It is also a useful technique in algorithms. Here's what it looks like:

1. ***Divide:***
   - *Split* the systems in to subsystems.
   - *Develop* each subsystem independently.
   - *Test* each subsystem in isolation.
2. ***Conquer:***
   - *Integrate* the subsystems together.
   - *Test* the system as an integrated whole.

From a teamwork perspective, it can work well for tasks such as harvesting crops, where everyone is doing roughly the same thing and there is relatively little communication required between squads.

But for teams designing complex system, Divide & Conquer tends to fail at integration time: the subsystems end up being incompatible.

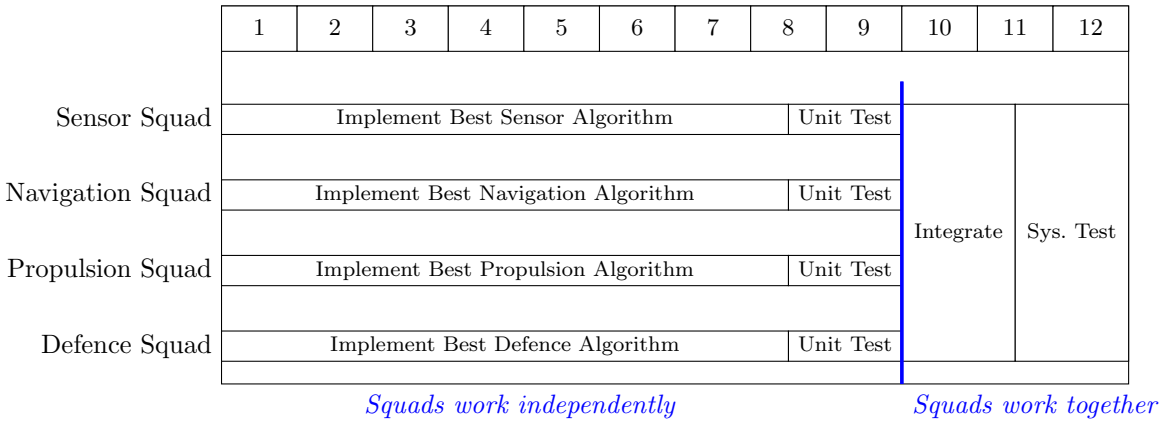


Figure 4: Gantt chart for *Divide & Conquer* teamwork approach

2.2 Modern Design: Integrate & Iterate

The solution to the problem of Divide & Conquer is to *integrate* first, before the subsystems are even built: first design the interfaces, then develop the algorithms. This approach is sometimes referred to *agile design*, and includes concepts such as *test-driven development* (TDD) and *continuous integration* (CI).

1. ***Integrate:*** The system as a whole should always be integrated.
   - *Test-First:* Agree on some tests before you write code.
   - *Interfaces:* Collaboratively design the subsystem interfaces (API).
   - *Mock Data:* Demonstrate system integration with mock data.
2. ***Iterate:***
   - *Generalize* to more test inputs.
   - *Improve* performance.
   - *Refine* interfaces as necessary.
   - *Sprint:* a limited time period in which to implement improvements.
     - If you can't do it within the sprint, defer it to a future sprint.
     - Always integrate and test on time.

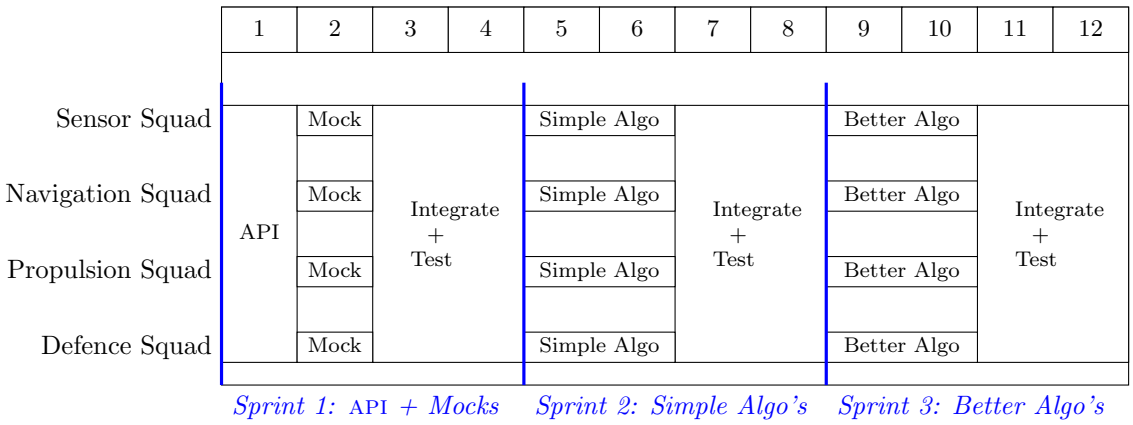


Figure 5: Gantt chart for modern *Agile* teamwork: Integrate & Iterate

3 The Godot Game Engine

This spaceshipe activity is built inside the Godot game engine. You need to know almost nothing about Godot to do this activity. Your instructor knows almost nothing about Godot. Some nice people from the UW Games Institute, who are experts with Godot, help us set this up. This section describes the few things that you will need to learn about Godot.

3.1 Godot is programmed with C#

You have probably never used C# before, so it is probably outside your comfort zone. That's ok. Part of the intent of this activity is to give you a safe opportunity to work outside your comfort zone [§1.2]. What you will learn is that C# is very similar to C, C++, and Java. And that all of these languages are also conceptually similar to Python — although the syntax is different. By the end of your degree, you will be able to pick up new languages fairly easily — especially new languages that are related to languages you already know.

C# is an *object-oriented* language. Maybe you never learned object-oriented programming before. That's ok. For most of this activity you just need to use objects/classes that have already been defined, and you will be able to figure that out.

3.1.1 C# Tips

The end of most lines of code (statements) must be marked with a semicolon;

```
//Single-line comments are indicated with double slashes.

/* Multi-line comments are indicated
using matching pairs of opening and
closing bracket-asterisk characters. */
```

3.1.2 Data Structures

Below are some C# data types you will likely encounter in this activity, it is not a comprehensive list so do not be afraid to research online documentation as needed.

string An array of characters. Used to represent words and sentences. Designated using double quotations marks:

```
string planetName = "Sirius";
```

A string can be compared to another string to see if they contain the same information:

```
planetName.Equals("Sirius")
```

would return true.

Strings can also be empty

```
string myEmptyString = "";
```

and null

```
string myNullString = null;
```

int An integer. Could be positive or negative, but no decimal. E.g. `int myInt = -6;`

float A number with a decimal, but with a limited number of digits that it can store (32 bits of precision). Sometimes referred to as a 'single precision floating-point value'. E.g. `float myFloat = 3f;`

double A number with a decimal, and with twice as many digits that it can store (64 bits of precision). Sometimes referred to as a 'double precision floating-point value'. Convert to **float** in C# like so:

```
float myFloat = Convert.ToSingle(0.123);
```

Vector2 An object containing two floats, x and y. In Godot, typically used to store a coordinate pair that represents position, velocity, etc. The Vector2 class itself provides numerous useful utility functions for comparing distances, angles, etc. Vector2 can convert to/from Vector3 by adding or removing a third coordinate $z = 0$. A useful member function which you may need is "angle_to (Vector2 to)" which gives the angle in radians between two vectors. To see more functions available to Vector2 in Godot, right click on Vector2 in Visual Studio Code and click "go to definition" or visit this link. `https://docs.godotengine.org/en/stable/classes/class_vector2.html`

Lists Collection of objects that can be accessed by index, similar to an array in c++ but can increase in capacity dynamically. `https://www.tutorialsteacher.com/csharp/csharp-list`

Queue Collection of objects that operates on the "first in, first out" principle. `https://www.tutorialsteacher.com/csharp/csharp-queue`

Dictionary Collection of key-value pairs, like an English dictionary which is a collection of words (keys) and associated definitions (values). `https://www.tutorialsteacher.com/csharp/csharp-dictionary`

For any object types that you are unfamiliar with, right click on the variable type in Visual Studio Code and click "go to definition". This will take you to the definition where you will find data members and methods that are available for you to use with that object.

Most built-in collection classes like tuples, lists, queues, and dictionary can be used together with C# "generics" functionality for more convenient data handling. For example, *List<string>* and *Dictionary<ulong, Vector2>* helps track/enforce what types of data a given collections contains.

`https://www.tutorialsteacher.com/csharp/csharp-generic-collections`

3.1.3 Mathf Library

Mathf is a built-in library which provides numerous useful mathematical functions such as *pow*, *max*, *clamp*, *Pi*, and *Deg2Red*, etc. For more information go to: `https://docs.microsoft.com/en-us/dotnet/api/system.mathf?view=netcore-3.1`

3.2 The Main Game Loop

Simple software programs commonly take some input, perform some computations, and then exit. Digital games are a unique form of software in that they are designed to provide players with a sense of continuous interactivity while simulating a dynamic game world, playing sound effects/music, and rendering flashy visuals to the screen. In order to achieve

this, most game software runs in a loop: cycling through a sequence of repeating steps many times per second until the player chooses to exit the game.

```
// Behind the scenes, Godot runs through a loop like this about 60 times per second
pseudo_code TypicalGameLoop()
{
    ReadPlayerInput(); //Is the player trying to move their character?
    UpdatePhysicsSimulation(); //Has the character bumped into anything?
    EvaluateGameRules(); //Does bumping into something have a consequence?
    RenderGameToScreen(); //Show that consequence on screen
}
```

In this activity, our main loop will be slightly different: when your ship is finally ready to fly, there won't be any manual player input. Rather, your code will respond to the changing environment around your ship and it will command the ships subsystems automatically. Our sandbox code will provide each of your subsystem update routines with access to object references relevant to your particular subsystem. For Defence, for example, the *turretControls* object allows you to aim and fire torpedoes.

```
// Our game loop will call each of your team's subsystem code in sequence each loop
pseudo_code GodotSpaceshipGameLoop()
{
    SensorsUpdate(shipSubsystems, shipStatusInfo, activeSensors, passiveSensors, deltaTime);
    NavigationUpdate(shipSubsystems, shipStatusInfo, galaxyMapData, deltaTime);
    PropulsionUpdate(shipSubsystems, shipStatusInfo, thrusterControls, deltaTime);
    DefenceUpdate(shipSubsystems, shipStatusInfo, turrentControls, deltaTime);

    UpdatePhysicsSimulation(); //Has the character bumped into anything?
    EvaluateGameRules(); //Does bumping into something have a consequence?
    RenderGameToScreen(); //Show that consequence on screen

}
```

Each subsystem routine is also provided with a reference to every other subsystem (via *shipSubsystems*) so that each subsystem routine can communicate and coordinate when necessary. Finally, *deltaTime* tells you how many seconds have elapsed since the last update loop.

3.3 The Sandbox API and Your Code

We refer to all of the behind-the-scenes code that manages the game simulation, keeps track of important statistics, and monitors for your successful landing on Kepler-438b as the *sandbox*. We've architected the sandbox such that the only files your team needs to touch are your ship's four *SubsystemController* files. Depending on what ship your team has been assigned to (let's use the *Nostromo* as an example), those files will be named:

- NostromoSensorsController.cs
- NostromoNavigationController.cs
- NostromoPropulsionController.cs
- NostromoDefenceController.cs

How and why each of your team's SubsystemControllers interact, how information is passed between subsystem code, and which squad is responsible for solving which problems is up to you and your team to decide.

As part of each subsystem's update function, an subsystem reference argument is passed in that your code can use to gain access to the other subsystem controllers on your ship. The below code example demonstrates what this looks like for the *Nostromo's* Sensors and Propulsion update functions:

```
//Assuming the NostromoSensorsController.cs file looks like this...
public class NostromoSensorsController : AbstractSubsystemController
{
    //Your Sensors squad decided to expose this Vector2 variable to other
    // subsystems by marking it "public"
    public Vector2 DesiredSolarSystemPosition;

    public void SensorsUpdate(NostromoSubsystems nostromoSubsystems, ...)
    {
        //Your fancy sensors code goes here...
    }
}

//Then your NostromoPropulsionController.cs file might look like this...
public class NostromoPropulsionController : AbstractSubsystemController
{
    public void PropulsionUpdate(NostromoSubsystems nostromoSubsystems, ...)
    {
        //Use the nostromoSubsystems object to
        //access the sensors subsystem controller from within
        //the defence subsystem's update method...
        var ourSensorsController = nostromoSubsystems.SensorsController;

        //Now we can access the exposed variable...
        Vector2 positionDifference = ourSensorsController.DesiredSolarPosition - this.GlobalPosition;

        //More fancy propulsion code here...
    }
}
```

Information about the current state of your ship is often useful for every subsystem controller and can be accessed via the *shipStatusInfo* object.

```
public class ShipStatusInfo {

    string currentSystemName; //The name of the solar system your ship is currently travelling within
    Vector2 positionWithinSystem; //Your ship's position relative to the center of the current solar system
    float shipCollisionRadius; //How large is your ship when approximated by a circle
    Vector2 linearVelocity; //What direction and how fast is your ship travelling within the current solar system
    float angularVelocity; //How quickly is your ship rotating
    Vector2 forwardVector; //A vector (in global coordinate space) pointing forward from your ship's bow
    Vector2 rightVector; //A vector (in global coordinate space) point starboard
    float torpedoSpeed; //How fast will torpedos travel when launched
    bool hasLanded; //Has your ship successfully landed?

}
```

3.4 Godot's Coordinate System

The coordinate system Godot uses for 2D games like ours has the positive X axis pointing to the right (on your display) and the positive Y axis pointing DOWN (on your screen). **PLEASE NOTE:** This is different than what you are likely used to from high-school math class where the Y axis typically points UP.

Godot uses radians to describe most rotations with positive rotations resulting in objects turning clockwise (on your screen).

Most of the coordinates described by the shipStatusInfo object are relative to the center of the current solar system your ship finds itself within. When your ship uses a warp gate to jump to a different solar system, it will end up at a new position relative to the center of the new solar system.

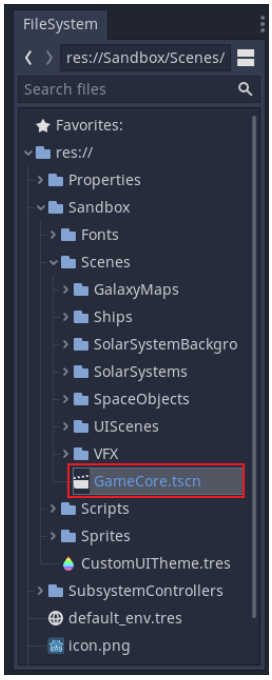
3.5 Changing Galaxies and Camera Focus

Generally speaking, we've architected the activity sandbox such that you won't need to know about anything Godot-specific beyond useful data structures like *Vector2* and utility function classes like *Mathf*.

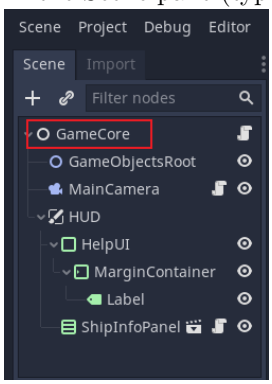
One Godot-specific technique you **do** need to understand however is how to choose which galaxy is loaded when you start the game and which ship the in-game camera initially focuses on when you start the game.

In both cases, you must:

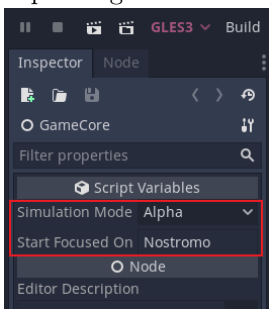
1. Use Godot's "FileSystem" pane (typically in the bottom left of the editor) to open the file "GameCore.tsc".



2. In the Scene pane (typically top left of the editor) select the root GameCore object.



3. In the Inspector pane (typically top right of the editor) you can choose which Galaxy to simulate using the "Simuation Mode" property. Select either Alpha, Beta, or Gamma depending on how complex/difficult you want the galaxy to be.



4. The "Start Focused on Ship Name" field can be used to set which ship the camera focuses on when the simulation begins. Typically you will set this to whichever ship your team is working on.

3.6 Godot Debug Drawing

Nodes in Godot have a drawing function which enable you to visually debug your code. Your colony ship is a node, so inheritance allows you to access the DebugDraw function found in Godot Node2D definition (look at AbstractSubsystemController.cs for definition)

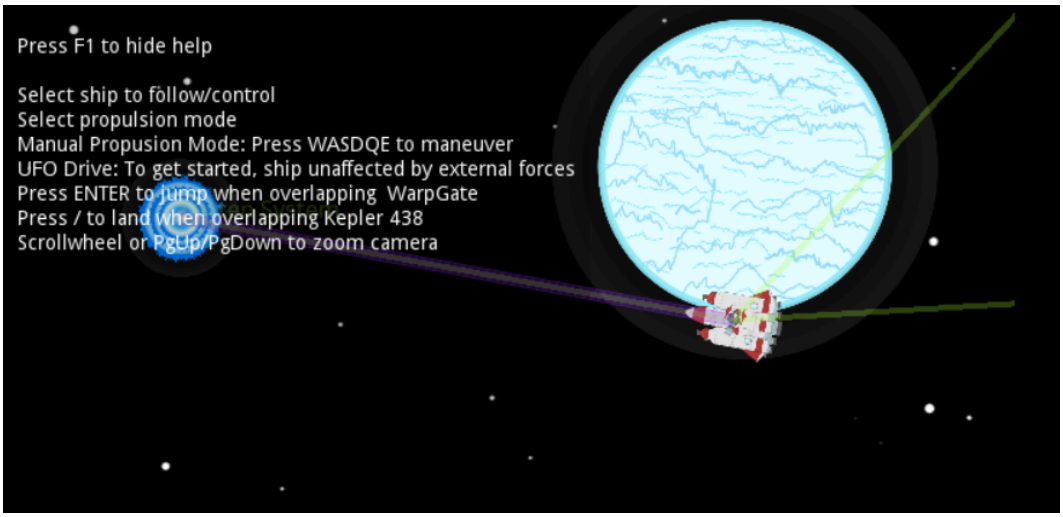
Below are definitions for useful drawing functions

```
public void DrawLine(Vector2 from, Vector2 to, Color color, float width = 1);
public void DrawString(Font font, Vector2 position, string text, Color? modulate = null);
```

Below is an example of the implementation of the DebugDraw function which was added as a member function to a subystemController class

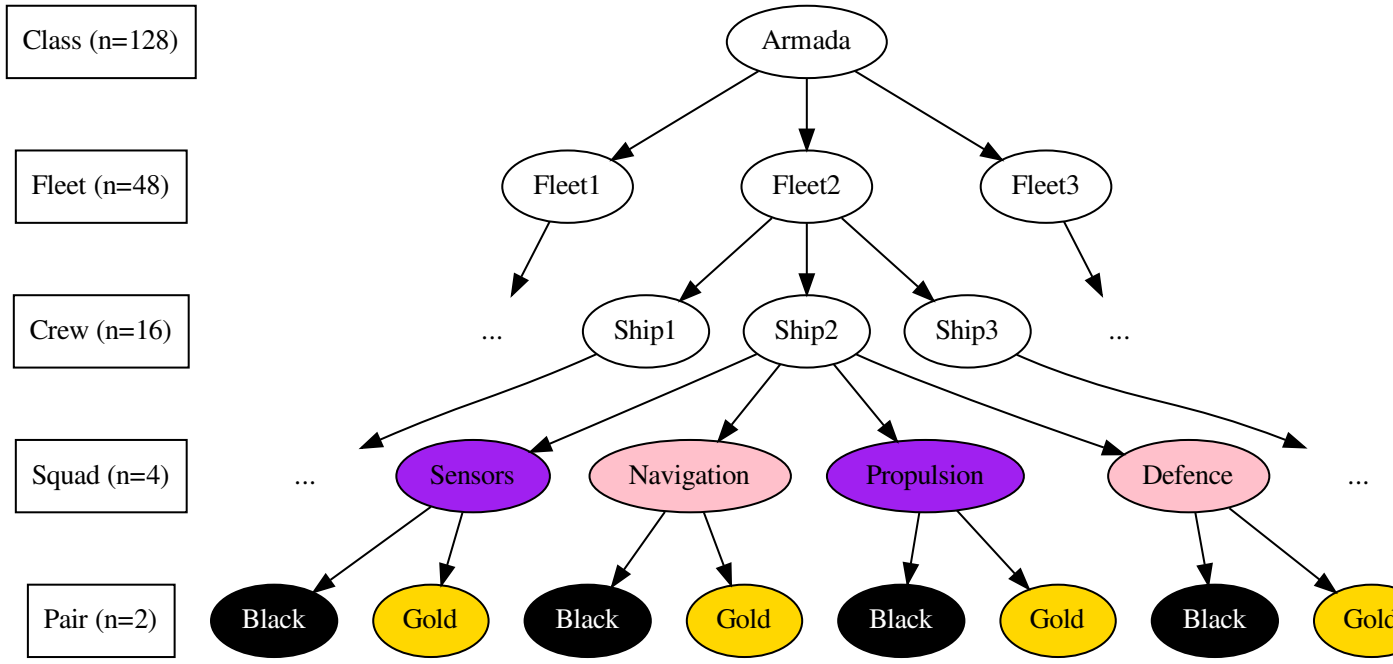
```
public override void DebugDraw(Font font)
{
    //iterate through all the unidentified gates
    foreach (var pair in unidentifiedGates)
    {
        //calculates the endpoint from the ship to a gate
        Vector2 endpoint = shipSystemPosition + Vector2.Right.Rotated(pair.Value) * 100f;

        //this will draw a purple line of width 5 from the ship's position to endpoint
        DrawLine(shipSystemPosition, endpoint, Colors.Purple, 5f);

        //displays purple text at endpoint
        DrawString(font, endpoint, "Unidentified Gate", Colors.Purple);
    }
}
```

By implementing code from above you can see a purple debugging line going from the ship to a gate!

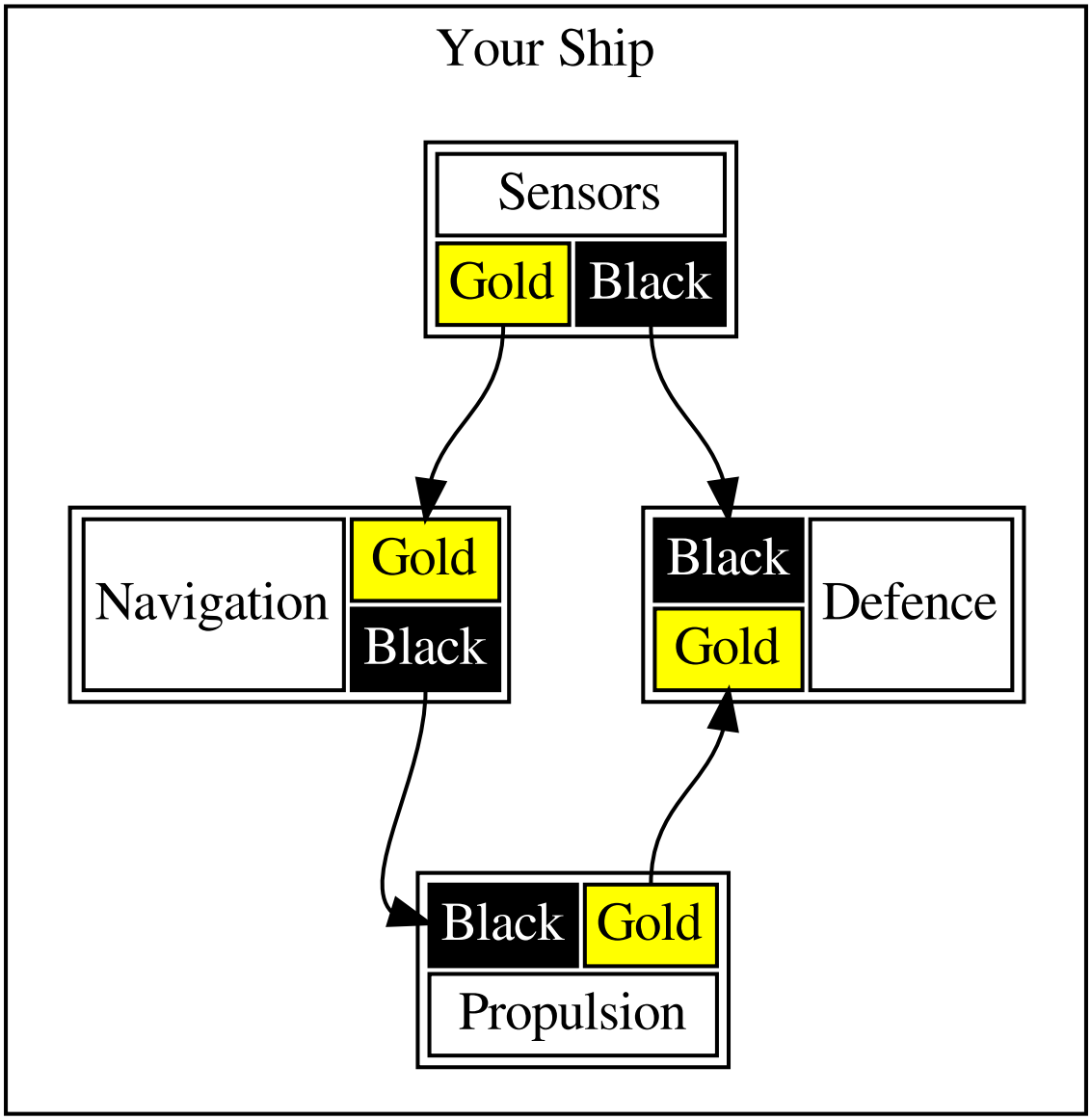

Press F1 to hide help
Select ship to follow/control
Select propulsion mode
Manual Propusion Mode: Press WASDQE to maneuver
UFO Drive: To get started, ship unaffected by external forces
Press ENTER to jump when overlapping WarpGate
Press / to land when overlapping Kepler 438
Scrollwheel or PgUp/PgDown to zoom camera

4 Class Organization

Each ship is staffed by a crew of roughly 16 students, divided into four squads: one squad for each subsystem. Each squad is split into two pairs: Black & Gold. Ships are organized into fleets of 2-3 ships. Crews within the same fleet will share knowledge with each other.



5 Team Organization



6 Team Design Decisions

There are several design decisions that affect multiple subsystems, and which need to be addressed at the beginning to ensure that the subsystems integrate. While your team needs to make some decisions to get started, your team might refine their choices as they progress through the mission. To start, your team will need to set up a meeting to discuss some of these important things listed below, as well as consider how implementation of these choices will affect the success of this mission:

Data Structure for Velocity. There are at least two choices:
Vector3D: x, y, magnitude (Cartesian coordinates)
Vector2D: angle, magnitude (Polar coordinates)

Units for Angles. There are at least two choices: *Degrees* or *Radians*. You might find that different parts of the Sandbox API use different units.

Push vs Pull. For every communication link between two subsystems, your team will need to decide if that link should be:

push where the producer calls a method, or writes to a field, in the consumer (in this example defence is the consumer and sensors is the producer)

```
//Assuming the NostromoDefenceController.cs file looks like this...
public class NostromoDefenceController : AbstractSubsystemController
{
    //Consumer exposes this list field as public and it will get set later in the SensorsUpdate function
    public List<EMSContactReport> emsContactData = new List<EMSContactReport>();

    public void DefenceUpdate(NostromoSubsystems nostromoSubsystems, ...)
    {
        foreach (var contact in emsContactData)
        {
            //data gets used here for example
        }
    }
}

//Then your NostromoSensorsController.cs file might look like this...
public class NostromoSensorsController : AbstractSubsystemController
{
    public void SensorsUpdate(NostromoSubsystems nostromoSubsystems, ...)
    {
    //Grab a reference to the Defence subsystem controller...
    var ourDefenceController = nostromoSubsystems.DefenceController;

    //Data gets set here
    ourDefenceController.emsContactData = newlyGeneratedData;
    }
}
```

or

pull where the consumer calls a method, or reads a field, in the producer (in this example defence is still the consumer and sensors is still the producer).

```
//Assuming the NostromoSensorsController.cs file looks like this...
public class NostromoSensorsController : AbstractSubsystemController
{
    //Producer exposes this list as public and will set it later in the SensorsUpdate function
    public List<EMSContactReport> emsContactData = new List<EMSContactReport>();

    public void SensorsUpdate(NostromoSubsystems nostromoSubsystems, ...)
    {
        //Compute sensor results...

        //data gets set here
        emsContactData = newlyComputedResults;
    }
}

//Then your NostromoDefenceController.cs file might look like this...
public class NostromoDefenceController : AbstractSubsystemController
{
    public void DefenceUpdate(NostromoSubsystem nostromoSubsystems, ...)
    {
        //Grab a reference to our ship's SensorsController
        var ourSensorsController = nostromoSubsystems.SensorsController;

        //Now the consumer can 'pull' the exposed data
        foreach (var contact in ourSensorsController.emsContactData)
        {
            //Pulled data gets used here
        }
    }
}
```

The difference between push and pull is subtle and of seemingly little importance for the current activity, however it is good to begin considering these design decisions as a matter of practice. They will become increasingly important as you begin to work with larger and more complex software systems.

Method Signatures & Field Types. Every communication link between two subsystems will involve one or more methods or fields. Meet to decide on what they should be called, what their parameters should be, what their return types should be, *etc.*

This list of team-level design decisions might not be complete: there might be other design decisions that multiple squads need to agree on. There are, of course, several design decisions to be made within each subsystem: some of those are described below in the respective sections for each subsystem.

7 Sensors Subsystem

Tell all other subsystems what is in the current solar system.
Receive requests from other subsystems about where to scan.

Metrics: { *Kepler438b* arrive safely at destination; *Energy* minimize energy usage from sensors

	Black Pair	**Gold Pair**
1. API + Mocks	*Listen* to Introduction	
	Talk with navigation and propulsion about required sensor data, storage format, and sharing. You can simply output the hard coded data for Galaxy Alpha (7.4) to get started. Be sure to talk about the strategies used with Team Gold. git commit -m 'Navigation and Propulsion sensing' Sensors*.cs git pull; git push	Talk with defence about required data, storage format, and sharing. You can simply output hard coded data for Galaxy Alpha (7.4) to get started. Be sure to talk about strategies used with Team Black. git commit -m 'Defence sensing' Sensors*.cs git pull; git push
	Together: determine how to send hard coded data to other subsystems	
	Share: knowledge with other Sensors Squads in your Fleet	
	Refine: your code based on learning from other Sensors squads	
	Demo: Galaxy Alpha with rest of ship	
2. Simple Algo	*Branch:* create new sensors branch in Git	
	Use the passive sensors to coarsely identify directions toward warp gates, planets, and large bodies	Use the active sensors to gather detailed information including the distance, direction, and composition of hazards and debris, as well as identify the destinations of warp gates
	Together: send sensed data to the other subsystems	
	Share: knowledge with other Sensors Squads in your Fleet	
	Refine: your code based on learning from other Sensors squads	
	Merge to Master: release your new code to the rest of your ship	
	Demo: Galaxy Beta with rest of ship	
3. Better Algo	Parse active sensor readings of planetary bodies to identify Kepler 438b	Optimize energy budget for active sensors by limiting scan range, arc, frequency
	Together: work to optimize energy budget	
	Merge: combine latest Black & Gold code	
	Share: knowledge with other Sensors Squads in your Fleet	
	Refine: your code based on learning from other Sensors squads	
	Merge to Master: release your new code to the rest of your ship	
	Demo: Galaxy Gamma with rest of ship	

7.1 Detailed Explanation

You are the eyes and ears of the starship. Your job is to provide the other subsystems with information about the space around your ship so that it can reach its destination whilst avoiding hazards. You are also tasked with identifying the habitable exoplanet Kepler438b once you arrive in the Kepler 438 solar system.

NOTE: Kepler 438b is the only planet that is composed of 70% common elements, 20% metals, and 10% water. Other planets may exhibit similar signatures but may include/lack small amounts of other elements.

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 code will be called through the **SensorsUpdate** method provided in your ship's SensorsController 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 several parameters. The first two are the **SubsystemReferences** and **ShipStatusInfo** parameter and are common to all SubsystemUpdate methods. Refer to the API section of this manual to learn more about all the information provided in these parameter.

The next two are the **ActiveSensors** and **PassiveSensors** parameters and are unique to your subsystem. This is how you learn about the ship's surroundings. The ship has two sensors: a passive, long range gravity wave interferometer (GWI), and an active, short-range electromagnetic sensor (EMS). A section describing each follows.

7.2 Passive Sensors: 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 a very long range but only reports the direction that a wave is arriving from and the gravity signature of that wave. Thus, the GWI tells you in which directions you will find Warp Gates but it does not tell you how far away they are or what other solar system a particular Warp Gate connects to. Because the ship's passive sensors are merely interpreting incoming gravity waves, they cost only minimal energy to operate continuously.

Access the GWI readings through the *PassiveSensors* parameter like this:

```
List<GWIReading> gwiReadings = passiveSensors.PassiveReadings;
```

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

```
public struct PassiveSensorReading
{
//A unique identifier to keeping track of the same object from one loop to the next
public ulong ContactID;

//The angle (from the global X axis) that the wave is arriving from
public float Heading;

//A characteristic signature that indicates what kind of
//celestial object generated this wave (i.e. WarpGate, planetoid)
```

```
public GravitySignature Signature;
}
```

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 object's GravitySignature 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.

7.3 Active Sensors: Electromagnetic Radar

In order to learn more about celetial objects and to be able to sense smaller obstacles at all, it will be necessary to actively scan the space around your ship with your EMS radar.

The EMS radar is an all-purpose electromagnetic radiation detector that can perform analyses on blue/red shift, spectroscopy, and brightness at high speed. It provides information about all objects within a chosen range and chosen arc segment around your ship. This sensor can be used to detect smaller space objects such as asteroids and space debris as well as query celetial objects like Warp Gates about what other solar systems they're connected to.

In order to generate these detailed scans, the EMS Radar must first actively send out a pulse of energy before interpreting the rebounding information. As such, the wider and farther of an area you choose to actively scan more frequently, the greater the total energy cost to your ship. While excessive energy usage will not have a direct impact on the operation of your ship, it is a target for optimization and comparison against competing ship teams.

You can perform a directed scan like so:

```
List<EMSReading> emsReadings = activeSensors.PerformScan(heading, arcAngle, range);
```

The EMS sensor will return a list of EMSReadings for every object within the designed arc/range **but** every active scan costs energy proportional to the total area of space scanned. Thus, particularly proud and glorious sensor squads may endeavour to optimize their sensor energy budget by being smart about how wide, how far, and how often they use their active sensors.

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

```
public struct EMSReading
{
//For keeping track of the same object each loop
public ulong ContactID;
//The angle of the reading (relative to global x axis)
public float Angle;
//The strength of the reading, proportional to distance via activeSensors.GConstant
public float Amplitude;
//The velocity of the detected object in global refrence frame
public Vector2 Velocity;
```

```
//The collision radius of the detected object
public float Radius;
//A more detailed description of the object's material composition
public string ScanSignature;
//E.g. a Warp Gate's destination solar system name
public string SpecialInfo;
}
```

The angle, much like the angle in GWI detections, indicates the direction the detection came from. The signalStrength describes the distance to the object from the ship via the following formula:

```
distance = activeSensors.GConstant * Amplitude
```

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 a ScanSignature, it is necessary to split the reported string by '|' and ':' characters. The C# String.Split() function will be particularly helpful in this task.

7.4 Galaxy Alpha Data

Because so many other subsystems rely on the sensors squad to feed them information, below are listing of objects within the different system of Galaxy Alpha that you can hard code into your initial SensorsController implementation and feed to other squads immediately before having to understand your active or passive sensors.

Sol System :
Alpha Centauri warp gate (800,0)
Asteroids (800,200), (800, -200), (600, 0)

Alpha Centauri System :
Kepler 438 warp gate (312.978, 386.366)
Asteroids (-395, -271.967), (153.251, 166.202)

Kepler 438 System :
Planet Kepler 438 (1276.38, 107.665)

8 Defence Subsystem

Objective: Fire torpedoes at collision hazards.

Metrics:

Kepler438b	arrive safely at destination
Health	minimize damage to the ship from collision hazards
Ammunition	minimize the number of torpedoes used

Steps: Use Git at every step. This is spelled out for you at the beginning, but left implicit thereafter. 1st iteration is on master branch.

	Black Pair	Gold Pair
1: API + Mocks	*Listen* to Introduction	
	API design with Sensors/Black git commit -m 'api' Defence*.cs git pull; git push	Fire a torpedo straight ahead git commit -m 'fire' Defence*.cs git pull; git push
	Together: Fire a torpedo at an arbitrary hazard	
	Share: knowledge with other Defence Squads in your Fleet	
	Refine: your code based on learning from other Defence squads	
	Demo: Galaxy Alpha with rest of ship	
2. Simple Algo	*Branch:* create new defence branch in Git	
	Aim at nearest hazard	Develop scatter/chain shot
	Together: fire appropriate number of torpedoes at nearest hazard	
	Share: knowledge with other Defence Squads in your Fleet	
	Refine: your code based on learning from other Defence squads	
	Merge to Master: release your new code to the rest of your ship	
	Demo: Galaxy Beta with rest of ship	
3. Better Algo	Aim at hazard on collision course	Optimize hit/shot ratio
	Merge: combine latest Black & Gold code	
	Share: knowledge with other Defence Squads in your Fleet	
	Refine: your code based on learning from other Defence squads	
	Merge to Master: release your new code to the rest of your ship	
	Demo: Galaxy Gamma with rest of ship	

8.1 Defence API

Defence controls the ship's Turret. The Turret has 4 torpedo tubes. Each torpedo has a timer that starts after being launched. When the timer ends, the torpedo explodes. If the torpedo hits an object before the timer ends, it will also explode. After firing, each torpedo tube needs to cool down and reload before it can fire again.

Here are some useful code examples you should pay attention to:

- `turret.aimTo = new Vector2(1f,1f); //Aims the turret at a position in local space`
- `turret.TriggerTube(tubeIndex, fuseDuration); // and for tubes 1, 2, 3`
- `turretControls.GetTubeCooldown(0) // and for tubes 1, 2, 3`

Just like every other subsystem team, your task is to implement your subsystem's controller code (ShipNameDefenceController) and its corresponding DefenceUpdate method.

DefenceUpdate will be called several times per second along with the other SubsystemUpdate methods. Keep this in mind when implementing your code. The DefenceUpdate method is passed three parameters. ShipStatusInfo tells you about the current state of your ship such as its position, heading, and velocity.

```
Vector2 shipPosition = shipStatusInfo.positionWithinSystem;

Vector2 shipVelocity = shipStatusInfo.linearVelocity;
```

Refer to the API section of this manual to learn more about all the information provided in this parameter.

DeltaTime tells you how many seconds have passed since the last update loop. It will normally be a small fraction (e.g. 1/60) of a second.

The TurretControls parameter is unique to Defence and is your means of commanding the ship's turret. By populating the TurretControls.aimTo field...

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

... the turret will instantly and automatically point its torpedo tubes at that position in space. For convenience, this aimTo position is absolute (in solar system coordinates). It is not relative to either the ship's position or heading.

To fire a torpedo, call the TriggerTube method...

```
 turretControls.TriggerTube(0, 0.5f);
```

The first parameter designates which torpedo tube you want to fire (i.e. 0-3) and the second parameter sets the fuse duration if a torpedo is launched. (I.e., the torpedo will explode after 0.5f second regardless of whether it impacts a target.) Because torpedos have an explosion radius, setting a shorter fuse can be used to cause a torpedo to explode in open space and potentially destroy multiple obstacles at once.

The explosion radius of torpedos can be checked via a static property...

```
float explosionRadius = Torpedo.ExplosionRadius;
```

To check whether or not a particular torpedo tube is ready to fire again, you can check each tube's cooldown via...

```
float tube0Cooldown = turretControls.GetTubeCooldown(0);
```

Which tube cooldown you check is determined by the passed parameter (i.e., 0-3).

Keep in mind that the torpedoes you fire will take time to travel to their targets and you can only fire so frequently. Thus, you may need to intercept fast-moving targets by firing at their anticipated future positions rather than simply firing at their current position.

You can reference the torpedo launch speed via the static property...

```
float speed = Torpedo.LaunchSpeed;
```

You will need to prioritize threats to be most effective, as stopping the ship to shoot everything will take too long! Consider firing only at targets that are headed for the ship, closer targets, faster targets etc.

8.2 Optimization & Advanced Tactics

Once your squad is able to fire at designated targets, you might consider developing different strategies for how and when you use torpedoes. The most obvious opportunity for optimization is making sure you only fire torpedoes when absolutely necessary. Coordinate with the rest of your ship (particularly Sensors and Propulsion) and determine which targets need to be cleared.

Rather than simply firing at one target after the next, you may also consider developing different firing patterns such as "scatter shot" or "chain shot". With scatter shot, you can fire multiple torpedoes simultaneously covering a wider arc in front of your ship. By leveraging the combined explosive radius, scatter shot will be particularly useful when many threats are flying towards your ship at once.

A chain shot tactic might involve firing two torpedoes in the same direction but with slight time delays between launches. If they are both timed to explode at the same point in space, the first torpedo can destroy initial obstacles while the subsequent torpedo can immediately clear up any shrapnel or debris that the first explosion generates.

When, where, and how you employ these advanced firing tactics is up to you and your crewmates to discuss. Just be mindful of your cooldowns!

9 Navigation Subsystem

Decide which planet or warp gate in the current solar system the ship should fly to. Tell Propulsion about it.

Metrics:

Kepler438b	arrive safely at destination
Energy	minimize cumulative jump cost
Time	minimize how long it takes to reach your destination

	Black Pair	**Gold Pair**
1. API + Mocks	*Listen* to Introduction	
	Work with Sensors to determine how to identify local warp gates. Be sure to talk about strategies with team gold as well. git commit -m 'Large object navigation' Navigation*.cs git pull; git push	Work with Propulsion to determine how to reach a given warp gate. Be sure to talk about strategies with team black as well. git commit -m 'Small object navigation' Navigation*.cs git pull; git push
	Together: Talk to other teams, learn about navigation requirements	
	Share: knowledge with other Navigation Squads in your Fleet	
	Refine: your code based on learning from other Navigation squads	
	Demo: Galaxy Alpha with rest of ship	
2. Simple Algo	*Branch:* create new navigation branch in Git	
	Harcode a solution for Galaxy Alpha. Help team Gold with implementing Dijkstra's algorithm	Hard code a solution for the Dijkstra's graph in Wikipedia. Then move on to implementing a solution with Dijkstra's algorithm
	Together: Navigate through Galaxy Beta	
	Share: knowledge with other Navigation Squads in your Fleet	
	Refine: your code based on learning from other Navigation squads	
	Merge to Master: release your new code to the rest of your ship	
	Demo: Galaxy Beta with rest of ship	
3. Better Algo	Travel through the warp gates and get to your destination using the code developed.	Travel through the warp gates and get to your destination using the code developed.
	Merge: combine latest Black & Gold code	
	Share: knowledge with other Navigation Squads in your Fleet	
	Refine: your code based on learning from other Navigation squads	
	Merge to Master: release your new code to the rest of your ship	
	Demo: Galaxy Gamma with rest of ship	

9.1 Detailed Explanation

Without you, humanity is quite literally lost. You must use your knowledge of the Galaxy Map to guide your ship to its destination. The Galaxy Map describes how each solar system is connected to its neighbours along interstellar laneways. Each solar system is only connected to a few of its neighbouring solar systems and so your ship will need to make multiple jumps in order to reach its final destination. The cost of travelling along any given laneway has an associated fuel cost and, while some laneways are less expensive to travel along than others, these costs are not necessarily proportional to how 'long' a laneway may appear. Together with your fellow crewmates, you will need to decide whether you want to minimze fuel costs, time taken, or the total number of jumps in your journey.

For your ship to jump out of its current solar system, it must reach one of the local warp gates that is connected to a neighbouring system. While your knowledge of the Galaxy Map can tell you what solar system you want to travel to next, you must work together with the Sensors team as only they can identify which local warp gate leads where.

Your task is to code your ship's NavigationController class and implement the NavigationUpdate method. NavigationUpdate will be called several times per second along with the other SubsystemUpdate methods.

The NavigationUpdate method is passed several parameters:

- `shipStatusInfo` can be queried to learn about the current state of your ship (e.g. its position in the local solar system, it's velocity).
- `galaxyMapData` is unique to Navigation and contains information about the various interstellar laneways between solar systems, and the fuel cost to travel along each one.
- `deltaTime` indicates how much simulation time has elapsed since your update function was last called.

The `GalaxyMapData` object you receive contains an array of `GalaxyMapNodeData` objects and an array of `GalaxyMapEdgeData` objects. Each `GalaxyMapNodeData` object contains the following information:

- `systemName` - The name of a solar system
- `edges` - An array of galaxy map edges connecting that solar system to its neighbours
- `galacticPosition` - The system's galactic position. **Note:** a system's galactic position refers to a different coordinate space than the within-system position coordinates Sensors, Propulsion, and Defence will be mainly dealing with. The distance between systems are impossible to travel between by any means other than warp gates.

Listed below are the different solar systems that your ship can travel to:

- Alpha Centauri
- Aquarii
- Barnard's Star
- Cygni
- Groombridge
- Indi
- Kepler 438
- Kruger
- Luyten
- Quaid
- Sirius
- Sol
- Wolf 359
- Yennefer

For convenience, these solar system names can also be found in the `SolarSystemNames.cs` file as string constants. For example, if you type

```
SolarSystemNames.BarnardsStar
```

the complier will replace it with the string"Barnard's Star System".

As the Navigation squad, your decision making challenges are at the long-term, strategic level: *Given the solar system we're in, where do we want to jump next?*

You might want your strategy to resemble something like the following steps:
1) Use your knowledge of the Galaxy Map to generate a graph of connected solar systems
2) Calculate a route from your current solar system to a desired destination solar system
3) Determine the next warp gate your ship must jump through
4) Coordinate with your fellow subsystem squads to find and travel through each desired warp gate

To begin integrating with your fellow squads early, consider some quick and dirty placeholder strategies you could implement before you have a complete Navigation soluation:
• Travel to a random warp gate
• Always go to the second closest warp gate
• Don't go to a solar system we've already been to unless we have no other choice

9.2 Different Galaxies

The testing sandbox has three different galaxies built in: Alpha, Beta, and Gamma. Each galaxy contains a different set of solar systems connected by a different configuration of warp gates. Generally speaking, the difficulty and complexity of each galaxy increases from Alpha, to Beta, to Gamma.

To configure which galaxy you will simulate when you start the game:
1) Open the GameCore.tscn scene in the Godot
2) In the Scene tab, select the GameCode node
3) In the Inspector tab, under the Script Variables group, change the Simulation Mode to either Alpha, Beta, or Gamma

Once the simulation is running, the overall structure of a particular galaxy can be previewed by pressing the TAB key in game.

9.3 Squad Milestones

In the Alpha galaxy, there are only three solar systems and there is only one path from Sol to Kepler 438. The Alpha galaxy is meant as a starting point for your squad's navigation solution and allows you to simply target the next obvious solar system in sequence and focus on determining how you'll pass that information to your ship's other squads.

The Beta galaxy contains multiple branching paths and most solar systems contain multiple possible warp gates. Eventually, your squad will want to be able to calculate an optimal path through any arbitray galaxy but, conveniently, the Beta galaxy bears a striking resemblence to the example node graph on the Wikipedia page describing Djikstra's Algorithm (See `https://en.wikipedia.org/wiki/Dijkstra's_algorithm`); a procedure for calculating optimal paths through arbitrary node graphs.

The Gamma galaxy is the most complex, dangerous, and unpredictable of the three galaxies. At this point, your squad will want to have implemented a pathfinding algorithm that can handle arbitrary node graphs. Use you knowledge of the Galaxy Map to chart a

course to your new home and keep your fingers crossed that the rest of your ship's crew can follow your directions!

10 Propulsion Subsystem

Fire the thrusters to get to the destination planet/warp-gate in the current solar system. Receive the target destination from Navigation.

Metrics:

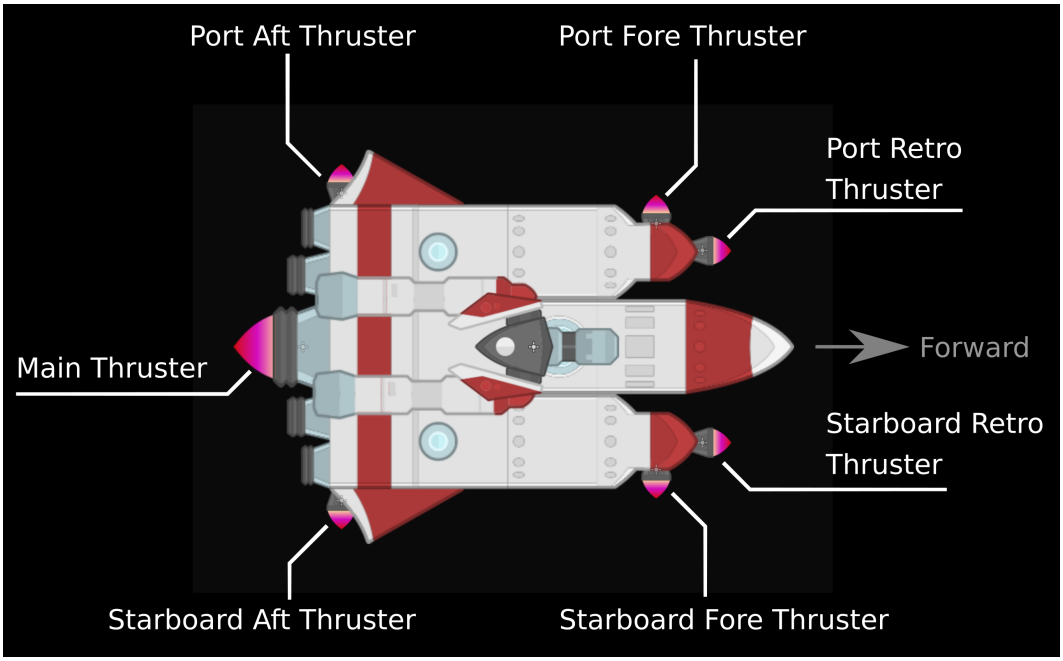
Kepler438b	arrive safely at destination
Health	minimize damage to the ship from moving too quickly through warp gates
Energy	minimize the energy costs to operate thrusters

	Black Pair (Translation Control)	**Gold Pair** (Rotation Control)
1. API + Mocks	Listen to Introduction	Listen to Introduction
	Meet Prop.Gold	Meet Prop.Black
	setup / install / git clone	setup / install / git clone
	API design with Nav.Black	UFO drive straight up
	Integrate + Test	Integrate + Test
	UFO drive to Nav's mock point with Prop.Gold	UFO drive to Nav's mock point with Prop.Black
	Integrate + Test	Integrate + Test
	Demo: Galaxy Alpha with rest of ship	
2. Simple Algo	*GitLab:* create new Issue+Branch in GitLab for 'Propulsion BoomBoom Controller'	
	Translation BoomBoom Controller	Rotation BoomBoom Controller
	Integrate + Test with Prop.Gold	Integrate + Test with Prop.Black
	GitLab: Create Merge Request Ask another Propulsion Squad to Review Respond to review Merge + Close Merge Request	
	Demo: Galaxies Alpha+Beta with rest of ship	
3. Better Algo	*GitLab:* create new Issue+Branch in GitLab for 'Propulsion Proprotional Controller'	
	Translation Proportional + Derivative Controller	Rotation Proportional + Derivative Controller
	Integrate + Test with Prop.Gold	Integrate + Test with Prop.Black
	GitLab: Create Merge Request Ask another Propulsion Squad to Review Respond to review Merge + Close Merge Request	
	Demo: Galaxies Alpha+Beta+Gamma with rest of ship	

10.1 Overview

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 controller class (PropulsionSubsystemController) and a corresponding SubsystemUpdate method (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 is passed several parameters. The first is a ShipStatusInfo object that describes the local situation of your shipe 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 (ThrusterControls) is unique to the Propulsion subsystem and is your means of commanding the ship's various thrusters. The final parameter (deltaTime) describes the number of seconds since that last update call.



The names and arrangement of the ship's various thrusters.

10.2 Mock Control: UFO Drive

Remember that the first step in modern agile design is to get the system as a whole integrated and working for one simple test case. The first simple test case for your spaceship is the Sol solar system.

The spaceship has a special UFO drive that allows you to set the velocity directly (i.e. kinematic control), without having to calculate the dynamic forces that would be used to control the ship with the regular thrusters.

Gold's first task is to use the UFO drive to fly diagonally (an arbitrary direction, just to show that the UFO drive works).

```
//Enable the UFO drive override
thrusters.IsUFODriveEnabled = true;
// fly down and to the right at a speed of 141 pixels per second
Vector2 velocity = new Vector2(100, 100);
thrusters.UFODriveVelocity = velocity;
```

Black's first task is to work with Navigation.Black to define the interface between Propulsion and Navigation. See §6 for some of the issues to consider. Navigation needs to tell Propulsion where the ship wants to go before Propulsion can work to get the ship there.

Together, **Black** and **Gold** can then update their code so that the UFO drive takes the ship to the destination determined by Navigation. One possible strategy is to set the UFO drive velocity to the distance between the target and the ship's current position. This strategy will make the ship slow down as it approaches the target, so it should land on the target smoothly.

10.3 Simple Algorithm: BoomBoom Contoller

Having successfully used the UFO drive to reach target destinations, next it's time to "take off the training wheels" and take control of the ship's thrusters directly. The ship has seven thrusters: four lateral ones at the corners for turning and strafing sideways, a main one at the rear for moving forwards, and two more pointing forward to allow for slowing down (and additional help turning if necessary).

A Simple BoomBoom Controller. This is the idea of how old home thermostats work to control the temperature in your home. In pseudo-code, the idea looks something like this:

```
target = 20; // room temperature
current = read_thermometer();
if (current < target) {
    furnace = 1; // too cold: turn on furnace
} else if (current >= target) {
    furnace = 0; // too hot: turn off furnace
}
```

Black does Translation Control: write a BoomBoom controller to make the ship go forwards, backwards, and side-to-side. Given the ship's current heading, will moving forwards or backwards get it closer to the target?

```
// forwards
thrusters.MainThrottle = 1;
thrusters.PortRetroThrottle = 0;
thrusters.StarboardRetroThrottle = 0;
// backwards
thrusters.MainThrottle = 0;
thrusters.PortRetroThrottle = 1;
thrusters.StarboardRetroThrottle = 1;
```

Gold does Rotation Control: write a BoomBoom controller to turn the ship left and right. Compare the ship's current heading to the target.

```
// turn clockwise/right (towards starboard side)
thrusters.StarboardForeThrottle = 0;
thrusters.PortAftThrottle = 0;
thrusters.PortForeThrottle = 1;
thrusters.StarboardAftThrottle = 1;

// turn counter−clockwise/left (towards port side)
thrusters.StarboardForeThrottle = 1;
thrusters.PortAftThrottle = 1;
thrusters.PortForeThrottle = 0;
thrusters.StarboardAftThrottle = 0;
```

BoomBoom controllers are very simple algorithms that just turn actuators on or off at full power. This is often fine for home furnaces because the human occupants don't really notice the slight temperature overshoots that occur when the thermostat keeps the heat blasting all the way up to the set temperature.

When using BoomBoom control with your ship's thrusters however, this lack of finesse has more drastic drawbacks. Note how BoomBoom control affects your ship's behaviour but don't spend too much time trying to reach Kepler 438b using BoomBoom control. The next section describes a more subtle (and much better) approach to dynamic control.

10.4 Better Algorithm: Proportional Controller

Where a BoomBoom controller would simply turn on a furnace full blast, a *proportional* controller turns on the furnace more or less depending on how cold it is inside. If it is very cold inside, then it will turn the furnace on strongly. If it is just a bit cool inside, then it will turn the furnace on just a little bit. A proportional controller is smoother and more fuel-efficient.

The difference between the target temperature and the actual temperature is the *error*, and the amount the furnace gets activated will be *proportional* to the error.

```
target = 20; // room temperature
current = read_thermometer();
k = 0.1; // a constant value, determined by tuning
if (current < target) {
    // too cold: turn on furnace
    error = target − current;
    furnace = error ∗ k;
} else if (current >= target) {
    furnace = 0; // too hot: turn off furnace
}
```

Black develops a proportional control algorithm for moving forwards/backwards and port/starboard (translation control). Note that the retro (reverse) thrusters are weaker than the main (forward) thruster, so they might use different constant factors.

Gold develops a proportional control algorithm for turning left and right (rotation control).

10.5 Even Better Algorithm: Prop. + Derivative Controller

An even smoother control algorithm also considers how fast the error is being reduced towards the target (its derivative). By incorporating this information, a PD controller can also start to slow down as it approaches it's desired target.

```
target = 20; // room temperature
current = read_thermometer();
kP = 0.10; // a constant value, determined by tuning
kD = 0.05; // a constant value, determined by tuning
if (current < target) {
    // too cold: turn on furnace
    error = target − current;
    derivative = (error − previousError) / deltaTime;
    P = kP ∗ error;
    D = kD ∗ derivative;
    furnace = P + D;
} else if (current >= target) {
    furnace = 0; // too hot: turn off furnace
}
```