Space Defense

Software Design Document

Authors

Name	Student Number
Ben Rodford	1313259
Matthew Tarantella	400071616
Nicholas Migliore	400071112
Jacob Anderson	400067644
Robert Holford	400088568

Revisions

Date	Description	Author(s)
2019/11/16	Initial design document	Ben, Matthew, Nicholas, Jacob, Robert
2019/12/24	Revision of Module Interface Specifications	Ben, Matthew, Nicholas, Jacob, Robert
2019/12/29	Editing and minor revisions (“version 1.0”)	Ben, Matthew, Nicholas, Jacob, Robert

Table of Contents

Introduction

Purpose

This software design document describes the design and architecture of the game Space Defense. This document serves as a design template upon which the game will be built. This draft of the document describes the ideal state for a functional version of the game, as the final product will have undergone improvements and minor changes.

Scope

Space Defense is a 2-dimensional, tile based video game for Windows and Linux platforms. Space Defense is a singleplayer, offline game with a campaign mode consisting of numerous levels and challenges. The gameplay will feature an overworld map for navigation, and a combat map that will include a player team, and a computer team. The player must defeat the computer team in order to win. The development will utilize the FOSS game engine, Godot, and utilize C# scripting. The goal of this project is to create an enjoyable game in a playable state.

Definitions, Acronyms, Abbreviations

- 2D: 2-dimensional, exists in a space with no depth.
- FOSS: Free and Open Source Software.
- Load: Open a save file to continue playing from the saved state.
- Non-isometric: The graphics will not attempt to create an illusion of depth
- Overworld: Area in a video game that serves to connect separate, individual levels
- Save file: A file containing information pertaining to a saved game state.
- Save: Record the current game state to be loaded at a later time, enable a single ongoing game across different sessions.
- Ship: A spaceship, the main units in the game.
- Tile-Based: Movements of sprites are limited to moving in fixed increments on a grid.
- Sprite: 2-dimensional bitmap image representing an object.

Overview

Title page, authors, revision, table of contents contain information pertaining to this document.

Introduction contains information that outline the general goal and concept of the project and this document.

Use Cases describes expected users of the game and the behaviour the manner in which these users will interact with the game.

Design Overview describes the structure of the video game's architecture, the interfaces implemented and utilized by the game, and any constraints that apply.

System Object Model is included for adherence to Design Document template. The game will be delivered as a single package, no internal interfacing is to be implemented..

Object Descriptions describes that attributes and methods of classes implemented within the game.

Object Collaborations describes the relationships between the implemented classes.

Data Design contains diagram-based information to describe the storage structure of the video game's data.

Dynamic model contains diagrams describing the usage and states of the game.

Appendix contains tools used in creation of the game.

Use Cases

Actors

Public Player

A public player is the end user of the game. They are expected to be able to play the game, or learn how to do so using instructions, but not understand how everything works. They will interact with the final product only.

Play Tester

A play tester is a public player that will play early versions of the game as to give feedback, and attempt to improve the game by finding bugs.

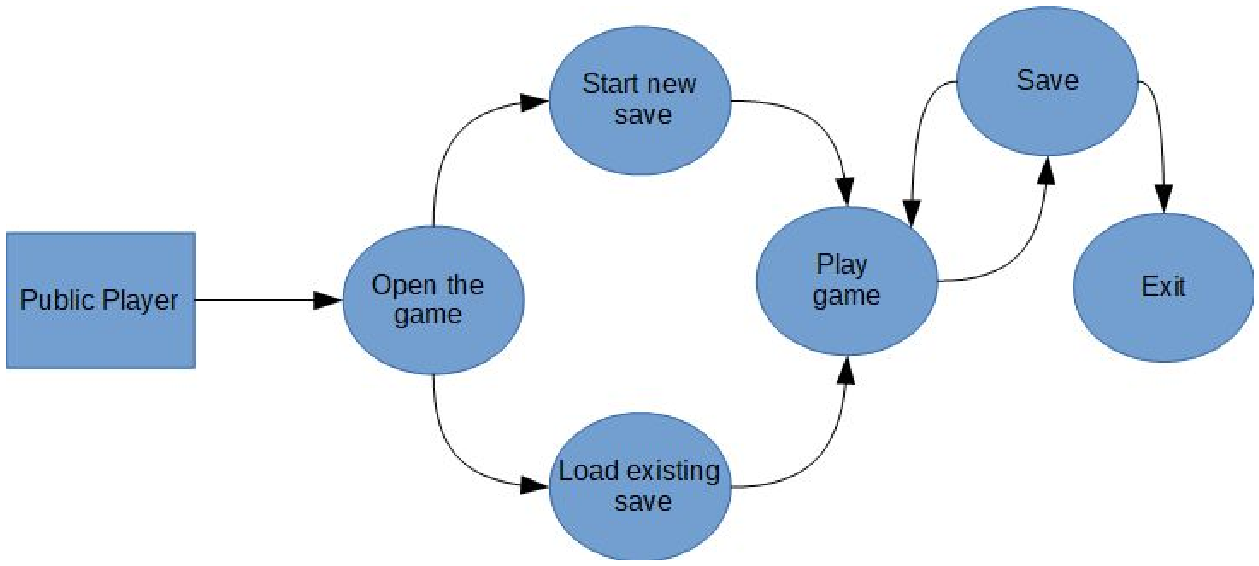
Developer

A developer is a public player that understands the workings of the game, and will play the game to test new content, and search for and fix bugs or otherwise unwanted behaviour.

List of Use Cases

- Open the game
- Start new save
- Load existing save
- Play the game
- Exit the game

Use Case Diagram



Use Case Descriptions

All users:

-Open the game. Open the executable and interact with home menu.
Goal: Allow users to access the game.
Success: Users can view and use the main menu.
Precondition: Game is downloaded on a system that meets requirements.
Trigger: Running the executable.
Rel: NA
Flow: User navigates to location of game shortcut or executable. User launches.
Assumptions: User can navigate their OS's filesystem.

-Start a new save. Create a game in an initial state.
Goal: Allow users to create a new save file.
Success: A valid save file is created.
Precondition: Game is running.
Trigger: User starts a new save.
Relations: NA
Flow: User presses "New Game" and enter the identity of the new game.
Assumptions: User is using a mouse/keyboard, or some other device/devices that can emulate the functionality of such.

-Load an existing save. Load a game in a previously saved state.
Goal: Allow users to continue a previously saved game.

Success: Game state of matches that of when the game was saved.
Precondition: A previously saved game exists.
Trigger: User loads an existing save.
Relations: NA
Flow: User presses “Load Game” and selects the save to load.
Assumptions: User is using a mouse/keyboard, or some other device/devices that can emulate the functionality of such.

-Play the game. Interact with game mechanics until exiting.
Goal: User plays the content of the game.
Success: Game mechanics perform as designed.
Precondition: User has started a new save or loaded one.
Trigger: Successful new save or load.
Relations: NA
Flow: User interacts with visual content.
Assumptions: User is using a mouse/keyboard, or some other device/devices that can emulate the functionality of such.

-Save the game. Create a save state that can later be loaded to resume progress from the same point.
Goal: Users saves the state of the game.
Success: Game state of when saved can be loaded later.
Precondition: User has started a new save or loaded one.
Trigger: User decides to save the game.
Relations: NA
Flow: User selects “Save Game”, then either saves a new file, or overwrites an existing one.
Assumptions: User is using a mouse/keyboard, or some other device/devices that can emulate the functionality of such.

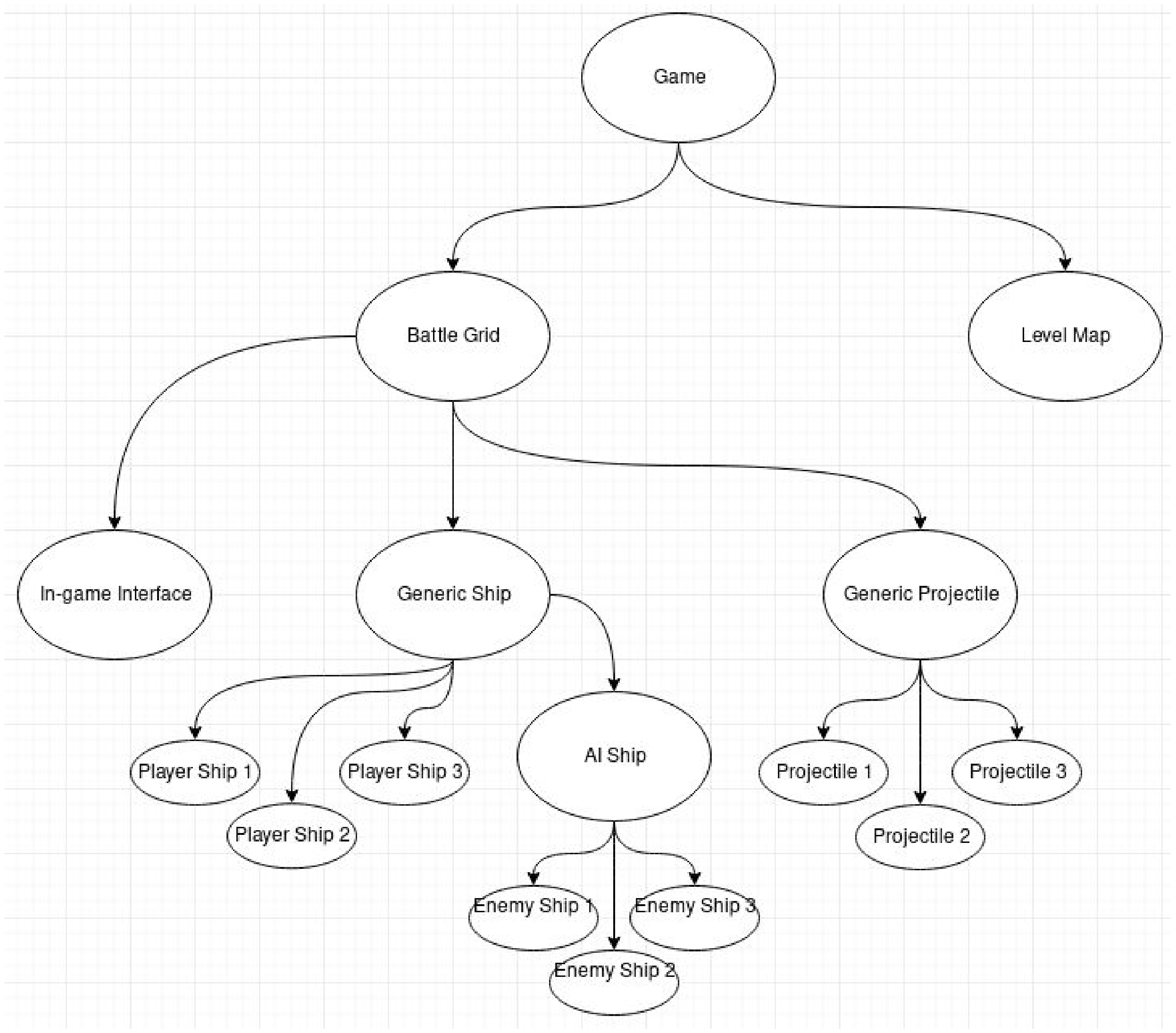
-Exit the game. Cleanly cease game function and return the user to their desktop.
Goal: Users can cleanly exit the game to desktop.
Success: Game stops running.
Precondition: Game is running.
Trigger: User exits game.
Relations: NA
Flow: User selects exit, then confirms an exit to desktop.
Assumptions: User is using a mouse/keyboard, or some other device/devices that can emulate the functionality of such.

Design Overview

Introduction

The design of the software is influenced by the node structure of the Godot development environment. The program is constructed with a tree of nodes, which may or may not have code attached. The diagram below shows the hierarchy and relation between pieces of code. Nodes not containing code, such as sprites or hitboxes, are considered properties of their parent node, which manages their usage. Nodes interact with other nodes through the node tree. Many design paradigms used are standards recommended by the Godot publishers and community.

System Architecture



Constraints and Assumptions

- Node interactions should be limited to parent/child interactions wherever possible.
- Nodes should access ancestral nodes by passing a request up the hierarchy, increasing encapsulation and limiting communication to neighbouring nodes.
- Child nodes should never directly perform operations on nodes other than themselves or their own children.
- All required assets are assumed to be present and accessible.
- The game should still run correctly if no user save files are available.
- Due to the turn-based system and generally low hardware requirements, processing time between individual actions or turns is not considered to be a limiting factor for playability.

Module Interface Specifications

Main game control module

Module

Game

Uses

Godot
System
Node2D

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	Inputs	Outputs	Exceptions
_Ready	none	none	

Semantics

State Variables

maxLevel: int
selectedLevel: int
playerPoints: int

unlocked: Array[String -> bool]

State Invariant

None

Assumptions and Design Decisions

- The Game module is responsible for setting up the initial state of the program and providing a main menu.
- This module is responsible for maintaining the state of the game as the user progresses.
- This module passes control to the Level Select module to get the user's choice of level.
- This module passes control to the Battle Grid module to play a level. Upon completion the Game module receives the result of the level.

Access Routine Semantics

_Ready(): transition: displays the main menu on the screen

Local Types

None

Local Functions

```
MainMenu(): transition: displays the Main menu
PickLevel(): transition:
        LevelMap.generate_map(maxLevel)
StartLevel(level: int):
        transition: Grid.loadLevel( // get initial ship setup for level )
SaveGame():
        transition: writeFile(filename, maxLevel, playerPoints, unlocked)
LoadGame():
        transition: displays the Load menu;
                (maxLevel, playerPoints, unlocked) = readFile(selected filename)
OptionsMenu
        transition: displays the options menu
```

Level select map module

Module

LevelMap

Uses

Godot
System
Node2D

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	Inputs	Outputs	Exceptions
generate_map	Int maxLevel	none	
select_level	Array[int] level, int player_pos	none	
move_player	Array[int] level	none	
start_encounter	Int pos	none	
set_selectable	Array[int] level	none	
set_not_selectable	Array[int] level	none	

Semantics

State Variables

Array[Array[int]] levels

State Invariant

None

Assumptions and Design Decisions

- This module is only accessed by the main Game module and provides an interface for the user to select a level.
- It is assumed that the level map will be generated, with the generate_map function, before any other routines are called
- This module allows players to select levels that they can interact with through encounters, shopping, or random items.

Access Routine Semantics

```
generate_map(int maxLevel):
{
    for (i = {0 to maxLevel}):
        Set i to a type of node
    for (i = {0 to maxLevel}):
        Connect i to at most 2 other nodes, with one node having 0 current connections
    for (i = {0 to maxLevel}):
        draw node for level i
}
select_level(Array[int] level, int player_pos)
{
    Display level information on HUD
    If level[s] == 1:
        Show move button

}
move_player(Array[int] level )
{
    If level[s] == 1:
        Set player position to level[p]
    Show start encounter information for level
}
Start_encounter(int pos)
{
    Create battle grid based on level number
}
set_selectable(Array[int] level)
{
        level[s] = 1
}
set_not_selectable(Array[int] level)
{
        level[s] = 0
}
```

Local Types

None

Local Functions

None

Battle grid module

Module

Grid

Uses

Godot
System
TileMap

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	**Inputs**	**Outputs**	**Exceptions**
_Ready			
loadLevel	Array[ship]		
RangeCheck	range: int, pos: Vector2	Array[Vector2]	invalid_coord
_Input	@event: InputEvent		
playerTurn			
compTurn			
validMove	ship, Array[Vector2]	bool	
move	ship, Vector2		invalid_coord
validAttack	ship, Array[Vector2]	bool	
attack	ship, ship Array[Vector2]		invalid_coord

defeat			
victory			

Semantics

State Variables

selected: Vector2
gridSize: int
playerTurn: int
aiNodes: Array[Vector2]
playerNodes: Array[Vector2]
impassibleNodes: Array[Vector2]

State Invariant

|gridSize| % 2 == 0
32 <= |gridSize| <= screenHeight/4

Assumptions and Design Decisions

- The battle grid module is responsible for controlling the flow of a battle. This includes implementing the turn-based system, providing interactive control to the user during their turn, and executing the AI opponent’s actions on their turn.
- The grid module initializes the battle state according to the level provided by the Game module.
- This module coordinates with the Interface module to provide user interaction.
- This module returns the result of the battle to the main Game module.

Access Routine Semantics

_Ready()

- // Godot function, Called when the node enters the scene tree for the first time

loadLevel(array[ship]):

- transition: $\forall(x \in ship : addShip(x . position))$

RangeCheck(range, currentPos):

- output:=($\forall\ i : Vector2 \mid i . x \in \{currentPos . x - range \ldots currentPos . x + range\} \wedge$
  $i . y \in \{currentPos . y - range \ldots currentPos . y + range\} \wedge$
  $(i . y \notin \{currentPos . y - range, currentPos . y + range\} \vee$
  $i . x \notin \{currentPos .x - range, currentPos .x + range\}) \wedge traversibleTile(i) \Rightarrow output.append(i)$)

_Input(InputEvent @event)

- Godot function for player/game events

- Transition:
    - If no object is clicked: none
    - If cpu ship is clicked: drawRange(ship.GetRange(), WorldToMap(mouseClick.Position)), selected = ship
    - If player ship is selected and empty cell is clicked:
        - if the click is in range and a valid move, move(ship, WorldToMap(mouseClick.Position))
        - Otherwise do nothing
    - If player ship is selected and cpu cell is clicked:
        - If the attack is valid, attack(selected ship, clicked ship)
        - Otherwise do nothing
    - If escape is pressed, call menu
    - If GUI element is pressed or selected, run associated function
    - Otherwise do nothing

playerTurn

- Transition: Allows the player to makes moves. Terminates when the player is finished, calling compTurn, victory, or defeat

compTurn

- Transition: Computer makes moves. Terminates when the algorithm is finished, calling playerTurn, victory, or defeat

validMove(ship, path) //if $ship.getPosition())$ isn't going to be a thing, pass currentpos

- Output := ($\forall\ x \in path \mid (x \in rangeCheck(ship.getRange(), ship.getPosition()) \wedge (traversibleTile(x)) \wedge (inRangeTile(x)) \Rightarrow true \mid otherwise\ false)$

move(ship, space):

transition: ship.position = space

exception: $(space \notin rangeCheck(ship.getRange(), ship.getPosition()) \vee (\neg traversibleTile(space)) \vee (\neg inRangeTile(space)) \Rightarrow invalid\ coord)$

validAttack (ship, path) //no account for ship ranged attacks currently. If we want ranged attacks, we'll need ship.getAttackRange and possibly a validRangedAttack()

- If player ship: output:= $(validMove(ship, path) \wedge path[last] \in aiNodes \Rightarrow true \mid otherwise\ false)$
- If ai ship: output:= $(validMove(ship, path) \wedge path[last] \in playerNodes \Rightarrow true \mid otherwise\ false)$

attack (atk, def, path)

- transition: (validAttack(atk, path) => attack(atk, def))
- exception: $(\neg validMove(atk, path) \Rightarrow invalid\ coord)$

defeat: transition: displays defeat screen
victory: transition: displays victory screen

Local Functions

_on_EndTurn_pressed(): runs AI turn before returning to player's turn.

SumOfPrevious : int -> int
SumOfPrevious(startNum) = (startNum * (startNum + 1))/ 2

inRangeTile: Vector2 -> bool
inRangeTile(coord):
$out := (coord.x \notin [0..gridSize] \vee coord.y \notin [0..gridSize] \Rightarrow false \mid$
$coord.x \in [0..gridSize] \wedge coord.y \in [0..gridSize] \Rightarrow true)$

traversibleTile: vector2 -> bool
traversibleTile(coord):
$out := (coord \in impassibleNodes \vee coord \in aiShips \Rightarrow false \mid$
$coord \notin impassibleNodes \wedge coord \notin aiShips \Rightarrow true)$

addShip(ship): self.ships.add(ship) //draw ships to screen

drawRange: Int -> Vector2
drawRange(range, currentPos):
Transition := $(\forall x : Vector2 \mid x \in rangeCheck(range, currentPos) => highlight(x))$

highlight: Vector2
highlight(pos) : transition:= Draw a highlight around the gridcells in pos (uses godot)

In-battle interface

Module

Interface

Uses

Godot
System
Node2D

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	**Inputs**	**Outputs**	**Exceptions**
_Ready()			
_Input	@event: InputEvent		

Semantics

State Variables

selectedAction: action

State Invariant

None

Assumptions and Design Decisions

- This module is responsible for the UI components during a battle.
- This module provides buttons to create a user input state, which is used by the Battle Grid module.

- This module allows displays selected ship data when a ship is selected
- If a friendly ship is selected before an enemy ship, a battle prediction will be shown
- This module controls the display of ship data, battle predictions, as well as battle results

Access Routine Semantics

```
_Ready():
    transition: draws the User Interface on the UI panel at the bottom of the screen.
_Input(inputEvent):
    transition:
        if (inputEvent.type == click):
            // get object (button on UI) at inputEvent.position;
            if (object.type in action):
                selectedAction = object.type
```

Local Types

action {move, attack, upgrade, None}

Local Functions

```
battle_predict(Array[Ship] playerShips, Array[Ship] enemyShips):
    return: compares strength of ships of both teams;
        returns ratio of relative strength as float [0..1]
```

Template ship module

Module

Ship

Uses

Godot
System
Node2D
Sprite

Syntax

Exported Constants

None

Exported Types

ShipType {attack, defense, support, flagship}

Exported Access Programs

Routine name	Inputs	Outputs	Exceptions
HP.get	none	Int HP	none
HP.set	hp: int	none	Type exception
firepower.get	none	Int firepower	none
firepower.set	firepower: int	none	Type exception
penetration.get	none	Int penetration	none
penetration.set	penetration: int	none	Type exception
armour.get	none	Int armour	none
armour.set	armour: int	none	Type exception
accuracy.get	none	Int accuracy	none
accuracy.set	acc: int	none	Type exception

evasion.get	none	Int evasion	none
evasion.set	evasion: int	none	Type exception
getRange	none	The range a ship can move in (int)	none
setRange	movement range: int	none	Type exception
damage	damage taken: int		

Semantics

State Variables

hp: int
firepower: int
penetration: int
armour: int
accuracy: int
evasion: int
movePoints: int
attackPoints: int

State Invariant

hp, firepower >= 0
penetration, armour, accuracy, evasion ∈ (0 .. 100)

Assumptions and Design Decisions

- This module provides all required definitions for a generic ship, which can be controlled by providing input.
- Different ships types are initialized by modifying this module’s initial values.

Access Routine Semantics

```
HP.get():
        return: HP
HP.set(val):
        transition: HP := val
firepower.get():
        return: firepower
firepower.set(val):
        transition: firepower := val
penetration.get():
        return penetration
penetration.set(val):
        transition: penetration := val
armour.get():
        return: armour
armour.set(val):
        transition: armour := val
accuracy.get():
        return: accuracy
accuracy.set(val):
        transition: accuracy := val
evasion.get():
        return: evasion
evasion.set(val):
        transition: evasion := val
getRange():
        return: range
setRange(val):
        transition: range := val
damage(val):
        transition: HP := HP - val; if (HP <= 0)
```

Local Types

None

Local Functions

move(Array[Vector2]): pass move request to Grid and move if possible
attack(Array[Vector2]): pass attack request to Grid and attack if possible

Template AI Ship module

Module

AIShip

Uses

Godot
System
Node2D
Sprite
Ship

Extends

Ship

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	Inputs	Outputs	Exceptions
playTurn	none	none	

Semantics

State Variables

sequence: Queue[turn]

State Invariant

None

Assumptions and Design Decisions

- This module extends the basic ship template by adding decision-making functionality for the computer player to use.
- Different AI ships are created by modifying this module's initial values.

Access Routine Semantics

```
playTurn():
    transition:
    {
        if (self.sequence is empty): makeSequence()
        if (not enough points remaining): break;
        if (turn.type == move): grid.move(turn.position);
        if (turn.type == attack): grid.attack(turn.position);
        turn := self.sequence.next();
        self.playTurn();
    }
```

Local Types

turnType {move, attack}
turn { type: turnType, position: Array[Vector2] }

Local Functions

```
makeSequence():
    transition: self.sequence = (multi-turn sequence of moves to achieve some goal)
                turn = self.sequence.next()
```

Generic projectile module

Module

Projectile

Uses

Godot
System
Node2D
Sprite

Syntax

Exported Constants

None

Exported Types

None

Exported Access Programs

Routine name	Inputs	Outputs	Exceptions
projectile_set	Vector2 position, Vector2 direction, float speed, int damage	none	

Semantics

State Variables

position: Vector2
direction: Vector2
speed: float
damage: int

State Invariant

speed > 0

Assumptions and Design Decisions

- This module provides all required definitions for a generic projectile.
- Different projectiles are created by providing this module with different initial values.
- This module will move a projectile across the screen based on its speed
- When the projectile moves into the same location as a ship, damage will be done to the ship based on the type of projectile

Access Routine Semantics

```
projectile_set(pos, dir, spd, dmg):
    transition: (position, direction, speed, damage) := (pos, dir, spd, dmg)
```

Local Types

None

Local Functions

```
move():
    transition:
        position := position + direction*speed;
        if (collision with entity):
            if (typeof(entity) == wall): remove self from parent //deletes self
            if (typeof(entity) == Ship): ship.damage(self.damage)
```

Data Design

Organization

root directory: main program files

- Godot configuration files
- Godot project files
- CSharp configuration files
- CSharp files for modules

  resources directory: imported assets

    graphics directory: contains graphics files

      backgrounds directory: large background images

      sprites directory: small sprite graphics files

      - sprite models made up of several sprites are organized into their own folders

    audio directory: audio files

      sounds directory: sound effects

      - sound effects are further grouped into related subfolders

      music directory: background music

  userdata directory: user configuration and settings files

    save directory: contains save files

Data files

Configuration

The user configuration/settings file is a JSON-like plaintext file, holding the last saved values for all persistent game settings. A default settings file is generated at the start of the program if the appropriate file is not present. This includes the following values:

Key	**Default Value**
resolution	(auto-detected else 480p)
difficulty	normal

music volume	100%
sound effects volume	100%

Saves

Saving the game creates a new file each time, recording the current state of the game in an encrypted JSON-like text file, with the goal of preventing users from being able to manually edit save files. A timestamp is also recorded in each save file so that saved states can be organized chronologically. The game can only be saved between levels. A save file that cannot be read for any reason is considered invalid and should not be displayed to the user. When a save file is loaded, the game takes on the state provided by the file and allows the user to continue playing. Values in a save file include the following:

Key	**Value**
time	current timestamp
level	highest accessible level
points	player’s accumulated points
(unlockable upgrades)	1 if unlocked else 0

Ships

Data for different ship types is loaded when the game is started. The ships data file is a JSON-like text file containing objects representing many ships, with each object containing values for the following:

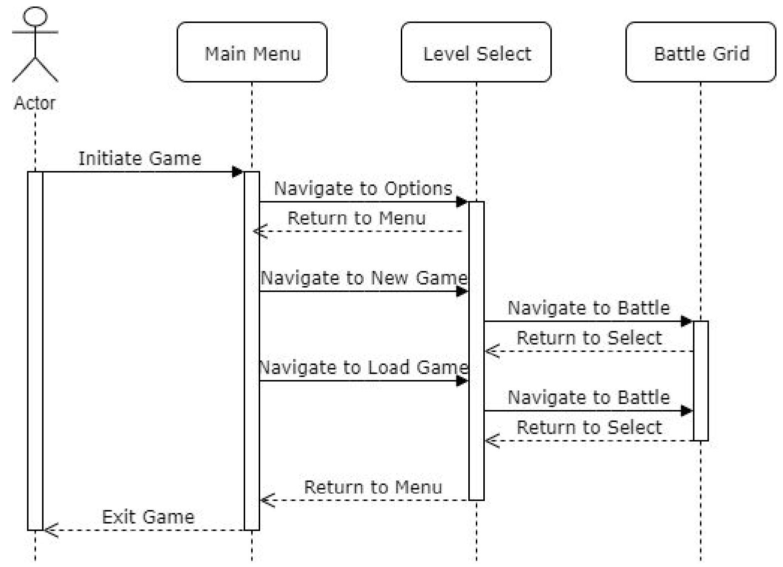
Key	**Value**
type	unique identifier for ship type
HP	
range	
firepower	
penetration	

armour	
accuracy	
evasion	

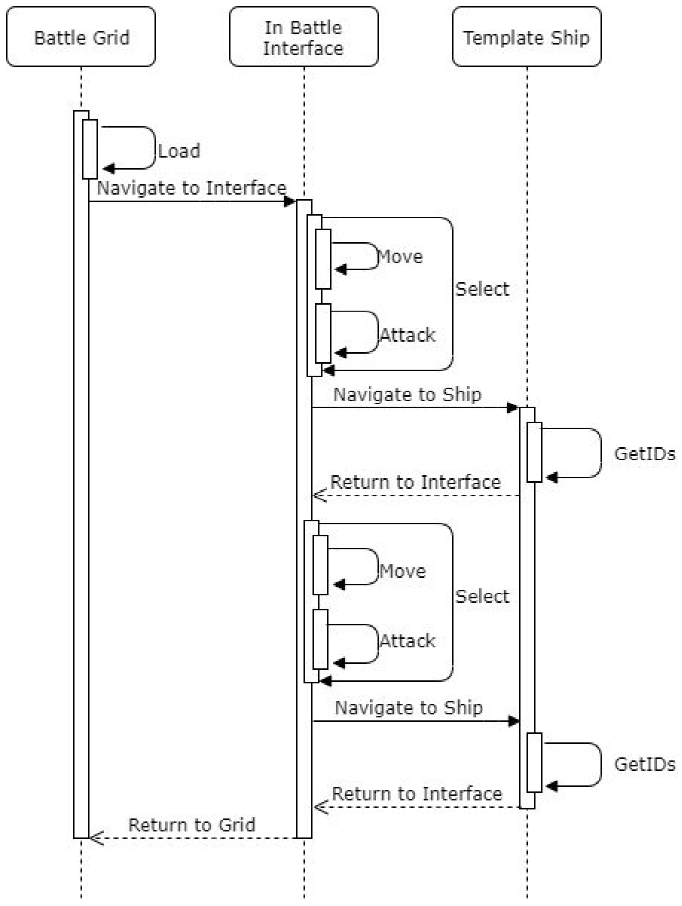
Dynamic Model

Sequence Diagrams

Main Menu Sequence

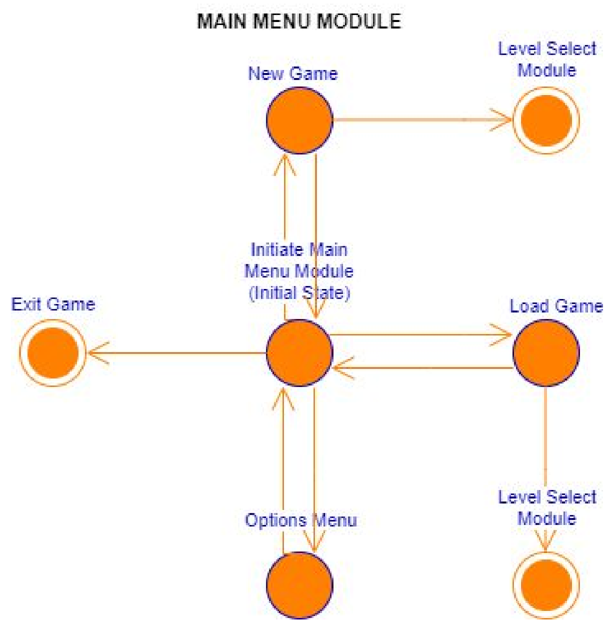


In-battle Sequence

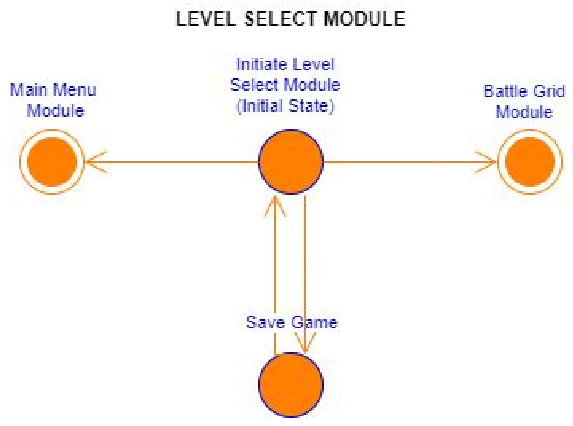


State Diagrams

Main Menu

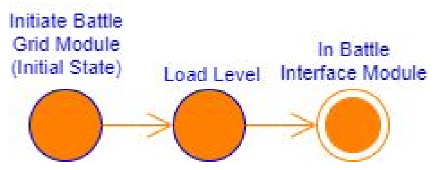


Level Select Menu

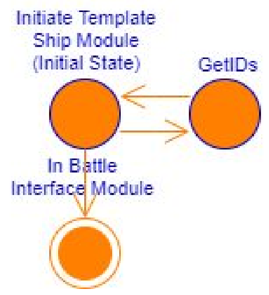


Game Board

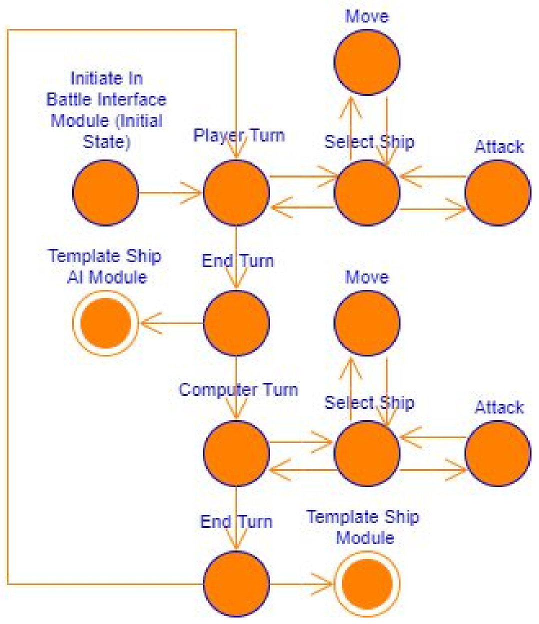
BATTLE GRID MODULE


Initiate Battle Grid Module (Initial State)
Load Level
In Battle Interface Module

Template Ship Module / AI Ship Module


Initiate Template Ship Module (Initial State)
GetIDs
In Battle Interface Module

In Battle Interface Module



Appendix

Design Critique

We believe the modular design imposed by using Godot is beneficial for creating such a game as this. Each entity is able to manage its own functions, but most interactions between entities is relayed through their shared parent, the Grid module. In this way, ships are able to act independently, while the Grid facilitates ensuring that all actions must be legal and that interactions always occur when they should. The documented modules are adequate to develop a functional prototype of the game, and should be easily scalable to add new functionality and modules as development and testing continues.

Tools

draw.io

https://www.draw.io/
Used for diagrams and charts.

GIMP

https://www.gimp.org
Used for drawing and editing graphic assets.

GitHub

https://github.com
Version control and code/file organization.

Godot

https://godotengine.org/
The development tools and engine used to create the project. The Godot IDE is used for editing scenes and most code, as well as compiling the software.

Google docs

https://docs.google.com
Used for live collaboration for documentation.

Krita

https://krita.org/en/
Used for drawing and editing pixel graphics.

Visual Studio Code

https://code.visualstudio.com/
External code/file editor with available C# integration.