Mobile Computing

App Development Project - Report

Student: Mike Pereira Gonçalves
ID: 0130898735

15.05.2017

Contents

List of Figures

List of Tables

1 Description

This report's focus lies on the actual implementation, called SpaceWars, of the previously created specification about a Star Trek-inspired space combat multiplayer game. For the full understanding of this project, it is advised to start with the specification before reading this report as details and features explained there won't be covered here anymore.

Shoutouts to Dren Gashi who also was involved in the creation of the specification and also is implementing his version of the game.

2 Specification Modifications

During the implementation phase, some smaller details about the specification had to be adjusted. Those modifications include aspects of the game which were either not fully thought out or were missing crucial information.

- **Multipeer Connectivity**:

  The `serviceType` has been changed to `mcspace-mpcdmv1`.

- **Spaceship**:

  The linear `damping` property of spaceships was added in the setup object configuration.

  Additionally in the setup message, the spaceship attributes were moved from the separate `ship_config` into the common `objects` section.

  Moreover, the `head` attribute of `ammo` was renamed to `available` and represents an array of torpedo IDs. This change was necessary in order to be able to send the current state of a game session to a client who for some reason got previously disconnected.

  The torpedoes shot by a spaceship are all of the same rectangular size and fly at a constant velocity. In order to reduce the fire rate, the delay between two consecutive torpedoes must be of at least 0.3 seconds.

Type	**Width**	**Height**	**Velovity**	**Shoot delay**	**linearDamping**
Torpedo	30	90	2000	$0.3s$	0

Table 1: Torpedo Properties Table

  Finally, the concept of a spaceship's shield wasn't fully covered. When the spaceship takes damage, it reduces its shield by the same amount. When the shield reaches 0 and thus becomes disabled, the ship can be destroyed by a single subsequent hit. With the exception of spaceships colliding with each other resulting in them being destroyed on the spot regardless of their shields, when the incoming damage is greater than the shield's value, the shield simply becomes disabled without killing the spaceship, for which another damage boost is necessary. This life mechanism can be compared to the one typically used on the Sonic video game franchise.

- **Meteoroid**:

  In order to be able to send the current game state to a client, the setup message does not only include the maximum but also the current health of meteoroids, respectively denoted `hp_max` and `hp`.

  Additionally, when destroying big meteoroids, there is a chance of a dilithium crystal appearing underneath. The spawn rate of this happening is denoted `spawn_rate` and is computed using this formula:

  $$\texttt{spawn\_rate} = 0.75 * mean(m)$$

  See specification section 2.4 to learn more about how to compute $mean(m)$.

- **Spacestation**:

  Because of their round shape, the dimensions of spacestations are represented by their radius instead of their width and hight. Additionally, their size has been increased from 150 to 200 radius.

  Moreover, the spacestation status message now also includes whether the station is currently transferring hp to its owner.

- **Move command**:

  Periodically, every client sends a move message to the server, stating his current position, rotation and velocity. Previously, the server would simply broadcast this message to all his clients. However, the number of messages sent from the server would increase exponentially with the number of connected clients. Therefore, all the move commands have been grouped into a single message which is then send to all the clients. See section 5 for the concrete structure of the JSON message.

- **Resume command**:

  This message type has been removed as it was sufficient to reuse the already existing `start` message.

- **Item respawn command**:

  This message type has been renamed to `object_respawn` and has the same structure as the object listing in the setup message.

- **Hp/Ammo/Kill commands**:

  These message types were all removed. It is sufficient to send a collision message to the clients who then compute the current life and ammo of the involved objects.

- **Gameover command**:

  The `winner` attribute is not necessary as it can be computed by going through the player list and checking for a `null` value in the `killed_by` property.

- **State Sync command**:

  The `paused` attribute has been removed. The default behaviour of the server is to pause the game whenever a `state_sync` is received, making this attribute redundant.

- **Survival game mode**:

  Instead of creating a completely separate game mode, CPU enemies have been added to the normal battle game mode. This makes it possible to play the game in singleplayer, as the user can just choose virtual enemies. They can also be chosen when playing with other friends, making the battle more lifeful.

3 Implementation

3.1 Architecture

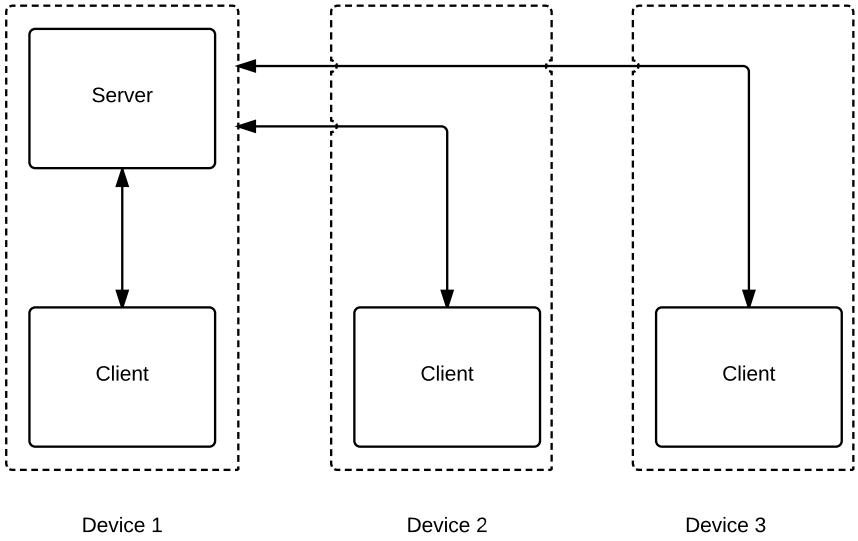

Figure 1: Server-Client architecture

SpaceWars uses a Server-Client architecture where one device simultaneously plays the roll of a server and client whereas the others are purely client-side. Together, they form a star topology with the server in the middle. Clients cannot directly communicate with each other. Messages must go through the server.

From an implementation point of view, the client does not know whether there is an active server running on the local device. The algorithm behaves exactly like the other clients. Messages are received through delegates and are sent by calling the `sendToServer()` function of the `ConnectionManager`, who then is in charge to relay this message to the correct destination.

Similarly, when the server wants to broadcast a message, he does so by calling the `sendToClients()` function. Once again, the `ConnectionManager` will make sure that the local client receives this message, whereas the other clients will receive it through `Multipeer Connectivity`. The interpretation step of the message is in both cases identical.

Since no two client can directly communicate with each other, it is clear that any incoming message is designated, and thus interpreted by the server. If the server is not running, then this implies that the incoming message comes from the server and thus should be interpreted by the client.

As for the message interpretation, the Command Design Pattern has been applied. Every message follows the JSON structure and has a `type` key which corresponds to a command type. Adding new commands is as easy as creating and registering a new class that extends the `Command` base class. This class is then in charge of interpreting the received JSON, extract the important key values and calling a delegate function from either the `ClientInterface` or `ServerInterface`.

Note: Even when a user is playing alone with the CPU, the same Server-Client infrastructure is running behind the scenes.

3.2 Scene Layers

The game scene is composed of multiple layers which all have their role. In the following is a list of all components per layer, as well as a small description:

- **World layer**: Spaceship, Torpedo, Meteoroid, Life orb, Dilithium, Spacestation, Blackholes

  Includes all game objects that the player can interact with. They all can be accessed by their unique IDs and will be shown on the Minimap.

- **Background layer**: Spacefield border, Starfield

  Includes all environment objects which are purely visual. A parallax scrolling effect is added to the whole layer.

- **Overlay**: Joystick, fire and pause button, Minimap, BarIndicator, Countdown, Pause pop-up

  Includes content that should always stay in the foreground and not be affected by player movements.

3.3 Object Manager

The `ObjectManager` is in charge of generating, and thus assigning game objects to the world, managing the individual layers, positioning new objects such that they don't collide with each ones, updating the Minimap, querying game objects by ID, delegating occured events to the `ClientInterface` respectively the `ServertInterface` (if available) and much more. It is supposed to be the main access point where all important objects can be accessed from.

3.4 World Generation

The world generation is a multi-step process. It all starts in the Lobby screen where the host is able to specify the game settings, including space field shape and size, and the number of each game object type that should be spawned in.

When transitioning to the Ship Selection screen, the server internally already starts with a JSON setup template containing the skeleton of every game object. Here is an example of such a template:

```
{
  "countdown":5,
  "type":"setup",
  "space_field":{
    "w":3000,
    "h":2000,
    "shape":"rect"
  },
  "objects":[
    {
      "type":"meteoroid2"
    },
    {
      "type":"meteoroid1"
    },
    {
      "type":"meteoroid1"
    },
    {
      "type":"life_orb"
    },
    {
      "type":"dilithium"
    },
    {
      "type":"cpu_slave"
    },
    {
      "type":"cpu_slave"
    },
    {
      "type":"cpu_master"
    }
  ]
}
```

For each meteoroid, the server chooses with a random distribution of 50% between `meteoroid1` and `meteoroid2`. For CPU enemies, there is a 1 in 6 chance to get a `cpu_master`. The remaining will produce a `cpu_slave`.

When every player has picked a spaceship type, the host clicks on the 'Start game' button. Here, the server would need to send the setup message to the other clients. However, the corresponding JSON has still not been fully generated.

Every game object comes with two initializers. It is possible to create them either by providing an object configuration in form of a JSON, or by giving the most important attributes as parameters and let the remaining ones be randomly picked. For instance, a unique ID must be assigned to every game object. In order to do so, the server creates an instance of the class `IDCounter`, which can either generate individual or a whole range of IDs without any overlapping.

Object configurations can only be generated after initializing them. Thus the server sends the setup skeleton, together with the ship selection choices of the players, to the local client, who then starts to generate and populate the space field with game objects. After completion, he sends the generated setup back to the server which then is able to broadcast it to the remaining connected clients. An example of a complete setup message can be seen in section 5.

Note: The server is the only decider on the configuration of the objects, such as their size, their health or damage, their position, etc. The client simply reads the data from the setup message and generates its objects according to it. Therefore it is possible that two server implementations differ in the object configuration generation, without causing any unexpected problems.

3.4.1 Object positioning

During the world generation and, more precisely, when spawning in new game objects, a spawn location needs to be selected in the space field. However, we want to avoid that objects collide with each other. More precisely, we want to ensure that they are at least a certain distance threshold away from each other.

The `ObjectManager` comes with a `getFreeRandomPosition()` function which, like the name suggests, returns a `CGPoint`. For this, he randomly picks a position on the space field and then checks whether the closest already placed game object is at least threshold distance away from said point. If not, a new random position is picked and the procedure is repeated until the termination condition is met. However, in order to decrease execution time on heavily over-saturated maps or to avoid dead locks all together, the threshold is linearly made smaller after each failed termination check.

3.5 Screens

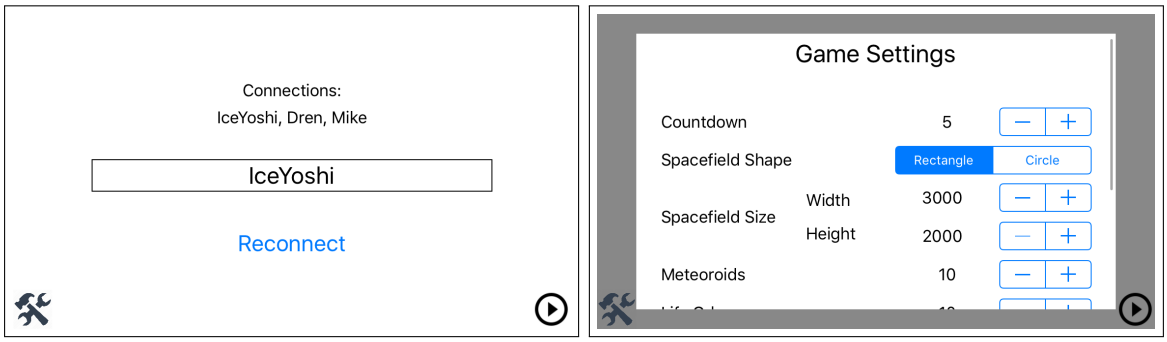

Figure 2: Lobby screen

When first opening SpaceWars, the user is confronted with the Lobby screen shown on figure 2. Here, he can type in his nickname and choose whether he wants to be the server (host) or client. When connected to other devices, a list of all participants is shown. Additionally, the host has the ability to view and change the game settings and to start the ship selection.

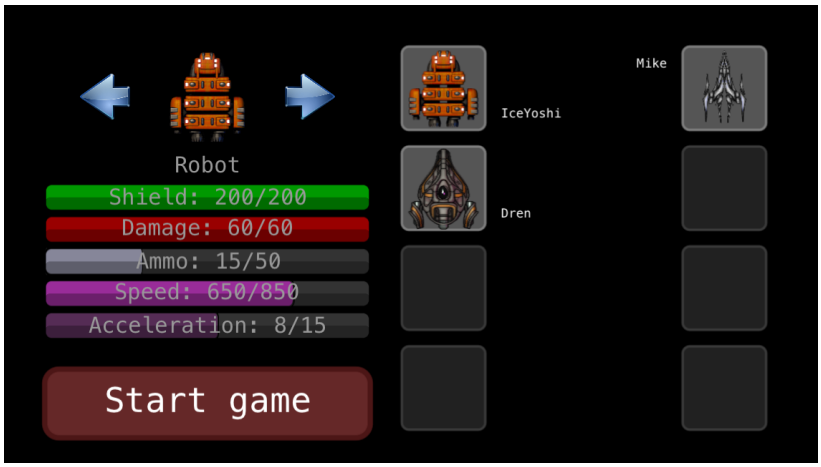

Figure 3: Ship Selection screen

In the Ship Selection screen at figure 3, the player can select between 3 different spaceship types, each with their strengths and weaknesses. On the right side of the screen, the selection choice from other players is being shown and updated in real-time. When everyone's ready, the host can generate the world and start the game by clicking on the 'Start game' button.

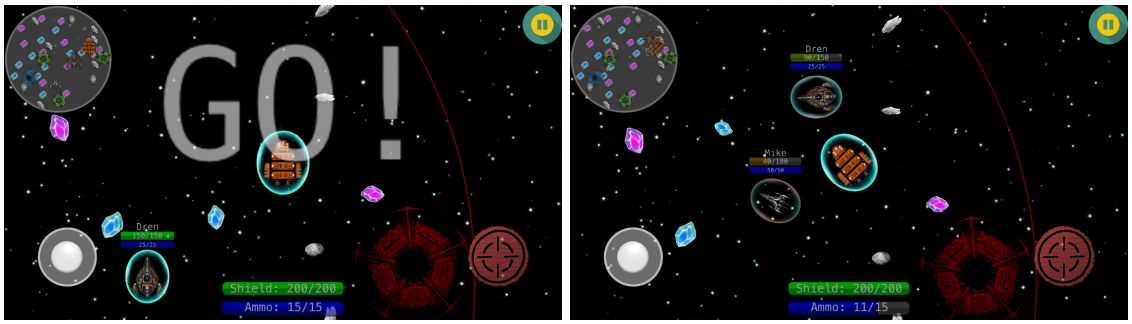

Figure 4: Gameplay screen

After the game scene has loaded, a countdown will start on every device like depicted on figure 4. The game begins when the host's counter reaches 0 (labeled "GO!"), after which a start message is sent to all the clients.

Here, the player can move around his spaceship, shoot torpedoes and pause the game. On the top-left side is a Minimap with the updated placement of all game objects. The game session ends either when no player spaceship remains or one remains with no CPU enemies left. When this condition is met, the host sends to all his clients a gameover message, resulting in a screen transition.

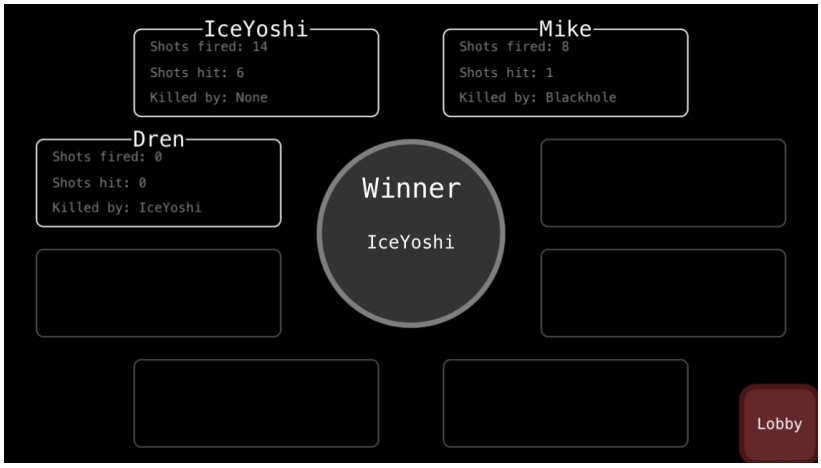

Figure 5: Gameover screen showing game statistics

Finally, in figure 5, the winner (if any) of the game session is shown, as well as game statistics per player like torpedo shot and hit count. With a click on the 'Lobby' button, the player enters the Lobby screen and the journey repeats itself.

3.6 Camera

The camera always follows the player's spaceship. However, instead of directly locking on the player, it changes its offset according to the spaceship's velocity in order to be slightly ahead and to show a bigger area in front of the user.

By performing a pinch gesture, it is possible to zoom the view point in and out. In order to avoid accidental pinch gesture recognition while simultaneously controlling the joystick and firing torpedoes, it is only possible to perform said action if at least two touch points are not on top of any overlay content.

Once the player's spaceship gets destroyed, it is possible to drag the camera using the pan gesture and thus to spectate the remaining match. In this mode, it is possible to click on an adversary's spaceship to make the camera follow the selected target.

3.7 Controller

The spaceship steering controller is inspired by a real-world joystick. Using touch gestures, it allows the player to precisely move his spaceship around. The distance of the controller head from its local origin indicates the applied thrust value, ranging between 0 and 1, giving the player even further movement control over his spaceship. On top of that, the controller comes with a "dead zone" of 30% where the player can rotate but not apply thrust. This is useful when aligning the spaceship in order to shoot at a target without moving towards it.

3.7.1 CPU

Figure 6: CPU enemies

In the game options, the host is able to choose whether CPU enemies should be included in the game session. Those enemies come with reserved spaceships that cannot be chosen by normal players. As the name indicates, they are completely autonomous and behave as if they were real players: They move towards other players and, once close enough, start to shoot at them. Their position, velocity and rotation is transmitted, similarly to the ones of other players, over periodic move messages.

The underlying implementation is quite simple. The server (host) assigns a virtual controller for each CPU enemy. Like a normal controller, the only two parameters that can be changed are angle and thrust. When provided with a list of spaceships with their current

position, the virtual controller simply choses to move towards the target that is the closest. In order to make the CPU easier/harder, it is possible to define a shoot rate and thrust limit. For instance, in the default settings, the thrust is limited to 20% and the fire rate at only one torpedo every three seconds.

3.8 Countdown

When transitioning from the Ship Selection to the gameplay screen, there is a countdown which primarily serves two purposes:

- It gives the player time to get familiar with the environment and to plan his first actions.
- It adds a way to synchronize the game start on all devices, regardless of the needed loading time. In fact, once the host's countdown reaches 0, he sends a start message to all his clients, whose countdown are purely visual and stop at 1, waiting for said message to arrive. In the other case, if a client's countdown lags behind and receives a start message, he simply skips the remaining count and jumps directly to 0.

The final "GO!" label can be seen in figure 4 on the left side.

3.9 Minimap

The Minimap is an overlay element that shows an overview of the complete space field during the gameplay. It is automatically updated when objects move/rotate, are removed from the scene or when they are respawn at another location.

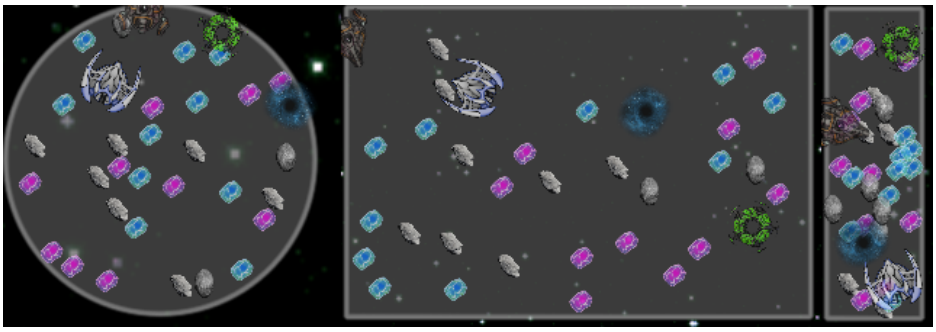

Figure 7: Minimap size adaption for three different space field aspect ratios

When first initialized, a certain overlay screen space is allocated to the Minimap. Like illustrated on figure 7, it tries to fill in that space, while conserving the aspect ratio of the space field.

3.10 BarIndicator

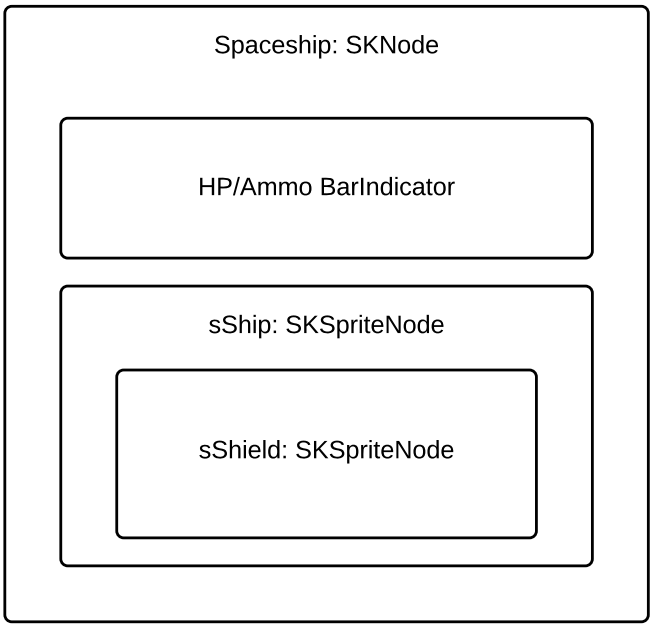

Figure 8: Spaceship Node nesting

A BarIndicator is a visual representation of a numeric value that ranges between a minimum and maximum bounds. It is used in the Ship Selection screen in order to show the characteristics of the individual spaceship types.

During gameplay, BarIndicators show the current health and ammo values of spaceships and, once damaged, the health of meteoroids.

When initializing, it is possible to give it 2 colors. As the value decreases, there will be a gradual transition between one color to the other.

The player's BarIndicators are static at the bottom center of the screen and remain there for the whole duration of the game. The adversaries' BarIndicators are drawn on top of their spaceship and thus move with the sprite node.

One technical challenge consists in making the BarIndicator move with the target node, but not let it rotate with him. In figure 8, we can see how the individual objects are nested with each other. For instance, when moving the main `Spaceship` class node, all its children will also move along with it. Same goes for the rotation.

The proposed solution is to override the `zRotation` getter and setter of the main class and to propagate those changes to the `sShip` sprite node, thus making the BarIndicator immune to the applied rotation. This solution does however not work for rotations coming from a `SKAction` or produced by physic computations as they seem to completely bypass getter and setter. This can be resolved by introducing public functions inside the `Spaceship` class who then perform the desired actions over the `sShip` sprite.

3.11 Blackhole

A Blackhole is a round game object that pulls any nearby spaceship in. When colliding with it, the ship gets damaged and a suck-in animation is shown, before reappearing once again at a predefined spawn position, usually at the center of the universe. Finally, an invulnerability time of two seconds is applied on the spaceship, making it possible to avoid collision with another object at the spawned position.

When multiple spaceships enter a Blackhole simultaneously, they simply collide with each other, resulting in a mutual destruction.

The animation duration is based on a given `delay` attribute and is defined in seconds, after which the player regains control over his ship again. The Blackhole also gets temporarily disabled during that time. This makes it possible to use this object as a means of teleportation in order to escape an adversary as he won't be able to directly follow the player due to this short inactivity time. As a trade-off, this action will also slightly damage the spaceship's shields.

3.12 Spacestation

Every player (except of CPU enemies) has his individual stationary space station. When the owner spaceship is on top of it, he slowly gets his shields repaired. However, after some regeneration time, the station gets temporarily disabled, thus avoiding someone camping at his station.

During this regeneration time, the player can still be shot at. Additionally, if an adversary touches a foreign station (tinted in red), it instantaneously gets disabled for the same amount of time.

3.13 Torpedo

Torpedoes are spawned in with a constant velocity of 2000. Furthermore, their `alpha` value decreases per frame by the value of `torpedoAlphaDecay` which is per default set to 0.02. The torpedo is removed from the scene when the condition `alpha < 0.4` is met.

In other words, when starting at $alpha = 1$, the torpedo will travel for exactly 30 $frames$, or half a second, at a speed of 2000 before disappearing. For the player, it seems as if the torpedo simply fades away over time.

3.14 Pause/Resume

During gameplay, any player may hit the pause button, resulting in a series of `pause` messages in order to pause every player on that game session. By design, only the host is able to resume the game. While paused, the name of the player that caused that action to happen is being displayed.

However, the game may also automatically pause when the host notices a connection loss between one of his clients. Again, it is the responsibility of the host to decide when to

Figure 9: Pause screen pop-up

resume the game, independently of whether the lost client rejoins. More about connection loss in section 3.16.

3.15 Collisions

Collisions between game objects are only enabled on the host's device. When such an event occurs, it is broadcasted via a message to all clients, who then animate and execute any necessary collision handling code.

As every client has a global view over the health and damage of each game object, they can compute the outcome of a collision without needing an extra message to update object data. The only exception to this rule are the Spacestations. There, it is not only necessary to know when a spaceship collides with it, but also for how long. Therefore, it was agreed on that the host just sends a Spacestation status message every time health is being transfered (per default once per second as long as the spaceship stays on top of an active Spacestation).

All in all, this should eliminate the risk of a desynchronisation between two devices, that would otherwise be possible if a collision would occur on one device but not on the other. Because of the small latency of $100ms$ between move messages, it is almost not perceivable by the player that the collision decisions happen in a distributed manner.

3.16 Connection loss

During gameplay, a connection loss between two devices may happen at an arbitrary point in time. There may be many possible reasons for such a connection cut. For instance, the application on the local device may have crashed, or the owner may have closed it. There might have been a notification or incoming call that moved the application in the background. Perhaps the communication channel is over-saturated or too far away from the source. At any case, the application must be able to handle such a situation.

Let us start with some background information. In order for two devices to find each other over Multipeer Connectivity, one of them must be advertising its service (server) while the other must be browsing for such an advertiser (client). Once the client finds a server that accepts the incoming connection, he automatically stops looking for other peers. This is not the case with the server, as many clients can be logically connected to the same server, forming a star topology.

The server keeps his advertising service running throughout the whole game session. However, the peer accepting condition depends on the current state of the game. When in the Lobby screen, every incoming client connection request will be accepted without condition. However, once past the Lobby state, the peerID `displayName` is compared to the ones registered for that particular game session. This name is not the nickname that a particular player has typed in. Instead, the device's `UUID`[1] is being used in order to have a unique string for any device and thus being able to recognize a certain player that got disconnected before.

As a consequence, when a client gets disconnected from the server, only he will be able to rejoin the game. Such a scenario is further illustrated in section 4.

Once the client has regained a stable connection with the server, he automatically sends a name message. If the server is not in the lobby state, he will respond with a `state_sync`, which includes all the necessary information for the client to return to the current game session.

One aspect has not been covered in this particular implementation of SpaceWars. If the server device is the one having connection issues, the current game session is simply considered to be over.

It would be feasible to also recover from this scenario, but due to the time constrains of this project, this challenge can be considered as future work. Here are the steps that need to be addressed:

- Every clients needs to have a global view of the game session. This includes:
  1. A map from every device UUID to the corresponding player
  2. Current state of Spacestations (time left until being disabled/enabled)
  3. Keeping track of game statistics
- Initially, we created a star topology where the server was at the center. If he disapears, the whole connection topology gets disconnected. There must be a leader election in order to elect a new server. The other clients must then start to browse for the new server. This step should not be difficult, as every client already received a unique ID for that game session, thus it would be sufficient to just pick the lowest one, who would play the role as a server.
- There needs to be some sort of a timeout for the other clients when searching for the new server, as he might as well have crashed, resulting in a deadlock. The exact procedure would need to be further developed.

[1] UUID: Universally Unique Identifier

3.17 Movement interpolation

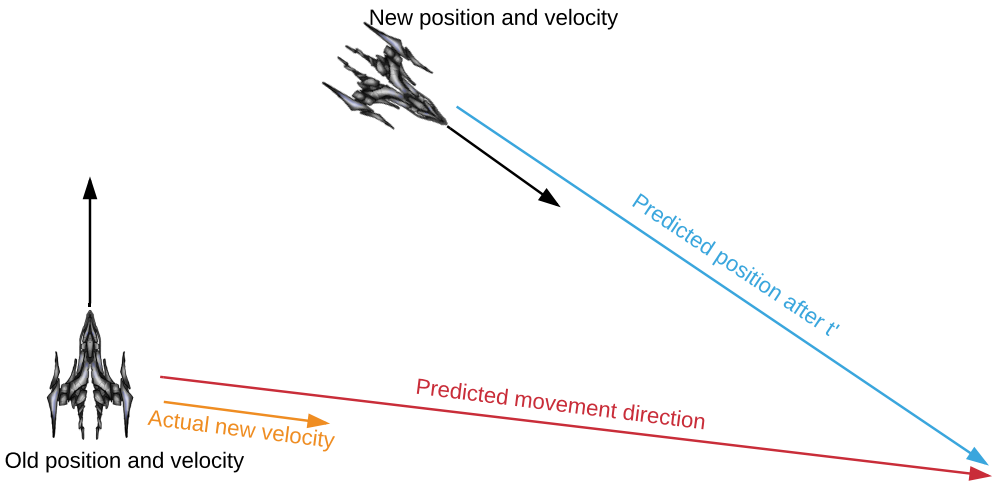

Figure 10: Movement interpolation

Every client sends his current position, together with his rotation and velocity in form of a move message to the server, which collects this data and broadcasts it to all his clients. Per default, those actions are done at a frequency of 10 messages per second.

While it would be sufficient for a client to just update the data of every spaceship upon receiving such a message, the resulting movement would not look smooth, as it would effectively just be teleporting the ships to the new position.

In order to improve the movement animation, let us consider figure 10. The client proceeds as follows:

1. First, by knowing the time delay in seconds Δt between two consecutive move messages and the frames per second rate fps at which the game is running locally, we can compute the mean amount of game frames that happen between two move messages like this:

   $$frames = \Delta t * fps$$

   For our typical case, we have: $\Delta t = 0.1s$ and $fps = 60$, thus: $frames = 6$.

2. Now we take the newly received velocity vector and multiply its length by $frames$, keeping his heading direction. If we add this computed vector to the new position, we get the predicted position for the spaceship after Δt (shown in blue).

3. Next, we subtract the old position from the new predicted position to get the predicted movement vector (indicated in red)

4. Finally, we divide this created vector by $frames$ and change the spaceship's current velocity to the resulting vector.

As for the rotation of the spaceship, a `SKAction` is executed for the same duration Δt, resulting in a smooth rotation speed.

3.18 Sound effects

SpaceWars comes with a main looping background song track, as well as multiple sound effects for common actions like shooting and hitting torpedoes, clicking on buttons, the last three seconds of the countdown timer, gameover transition, getting pulled from a Blackhole, collecting Dilithium Crystals and Life orbs, recharging at a Spacestation as well as destroying meteoroids and spaceships.

3.19 Optimizations

During the development of SpaceWars, a big performance hit has been noticed whenever `SKShapeNodes` were involved. After some research, it seems like shape nodes drastically increase the amount of draw calls and are constantly being updated, thus re-rendered for each frame.

It is recommended to avoid using them whenever possible. However, after some experimentation, there was a noticable performance improvement when adding shape nodes to `SKEffectNodes` and then set the `shouldRasterize` property to `true`. This way, the node and all its children are being cached and not constantly updated. Note that this only works for static content which does not change. Furthermore, this caching may lead to unexpected visual glitches. For instance, when applied to the space field border, half of it simply wasn't being rendered in, thus appearing invisible.

4 Use Case

Consider the following scenario:

Dren and Mike want to play SpaceWars together and start the application on their personal devices. Mike chooses the nickname 'IceYoshi' and hosts the game. Dren on his turn writes his own name as a nickname and joints the host, who then responds with his own name. However, Dren changes his mind and quickly alters his nickname to 'Drega'. This change is signalized via a message to the host who then updates his lobby view.

Mike adjusts the game settings and then clicks on the ship selection button, resulting in a transition to said screen on both devices. Mike chooses 'Human' as his spaceship race, and Dren decides to go with 'Skeleton'.

After hitting the start button, the game scene loads, and after waiting for the countdown to reach 0, the fun can start. Unfortunately, only a few seconds into the game, Dren receives an important phone call that he cannot dismiss. SpaceWars goes in the background and automatically sends a pause message to the host.

After finishing the call, Dren tries to reopen the game, but closes it accidentally. Therefore he opens a fresh instance of the application and, as expected, is once again at the Lobby screen.

In the meantime, Joe opens SpaceWars on his device. He chooses to impersonate Dren by calling himself 'Drega' and tries to join the already existing game session. However, the host notices that the device ID does not match any player in that game session and thus refuses this connection.

On the other hand, Dren enters his nickname and hits join. The host recognizes the device ID and responds with a `state_sync` message which contains the setup with all game objects and their most up-to-date position. Dren's device loads the game scene which is exactly at the same state as where he had left off. All what's left to do is for the host to click on resume and they are once again playing on the same game session.

It so happens that both players are sucked in a Blackhole at the same time, resulting in a mutual collision and thus destruction. The game ends by showing the gameover screen with the game statistics. It's a draw.

Mike suggests to start a new game session, and this time wait for Joe to also join in.

5 Appendix

Here is an updated sample of all possible messages that can be send to- and from the server. This time around, the shown values are taken from a real-world use case.

```
{ // Client -> Server
  "type":"name",
  "val":"Mike"
}

{ // Server -> Client
  "type":"name",
  "val":[
    {
      "pid":1,
      "name":"IceYoshi"
    },
    {
      "pid":2,
      "name":"Dren"
    },
    {
      "pid":3,
      "name":"Mike"
    }
  ]
}

{
  "type":"start_ship_selection"
}

{
  "pid":2,  // pid only when Server -> Client
  "type":"ship_selected",
  "ship_type":"skeleton"
}
```

```
33
34 {
35   "type":"setup",
36   "pid":2,  // pid of the client that receives this message
37   "countdown":5,
38   "space_field":{
39     "shape":"circle",
40     "r":1500
41   },
42   "objects":[
43     {
44       "dmg":50,
45       "rot":5.143643,
46       "size":{
47         "w":93,
48         "h":93
49       },
50       "id":32,
51       "hp":87,
52       "type":"meteoroid2",
53       "hp_max":87,
54       "pos":{
55         "x":1911,
56         "y":1255
57       }
58     },
59     {
60       "size":{
61         "w":300,
62         "h":300
63       },
64       "hp":750,
65       "acc":2,
66       "pos":{
67         "x":309,
68         "y":930
69       },
70       "speed":150,
71       "rot":0,
72       "name":"COM",
73       "damping":0.2,
74       "id":63,
75       "ammo":{
76         "max":64,
77         "available":[
78           64
79         ],
80         "min":64
81       },
82       "type":"cpu_master",
83       "hp_max":750,
84       "dmg":80
85     },
86     {
87       "dmg":83,
```

```
      "rot":0.3719037,
      "size":{
        "w":76,
        "h":38
      },
      "id":9,
      "hp":142,
      "type":"meteoroid1",
      "hp_max":142,
      "pos":{
        "x":1247,
        "y":786
      }
    },
    {
      "hp_gain":17,
      "id":35,
      "rot":2.557928,
      "type":"life_orb",
      "size":{
        "w":43,
        "h":59
      },
      "pos":{
        "x":2287,
        "y":2611
      }
    },
    {
      "size":{
        "w":108,
        "h":132
      },
      "hp":40,
      "acc":15,
      "pos":{
        "x":2426,
        "y":800
      },
      "speed":800,
      "rot":0,
      "name":"COM",
      "damping":0.8,
      "id":57,
      "ammo":{
        "max":58,
        "available":[
          58
        ],
        "min":58
      },
      "type":"cpu_slave",
      "hp_max":40,
      "dmg":30
    },
```

```
    {
      "size":{
        "w":49,
        "h":68
      },
      "id":41,
      "rot":4.743699,
      "type":"dilithium",
      "ammo_gain":13,
      "pos":{
        "x":1456,
        "y":166
      }
    },
    {
      "rot":4.687587,
      "inactive":60,
      "size":{
        "r":200
      },
      "rate":15,
      "owner":3,
      "id":157,
      "active":10,
      "type":"spacestation",
      "pos":{
        "x":1956,
        "y":785
      }
    },
    {
      "size":{
        "w":108,
        "h":132
      },
      "hp":100,
      "acc":15,
      "pos":{
        "x":2063,
        "y":2498
      },
      "speed":850,
      "rot":0,
      "name":"Dren",
      "damping":0.95,
      "id":2,
      "ammo":{
        "max":140,
        "available":[
          103,108,116,105,107,114,97,91,132,120,127,136,111,94,121,124,118,13
              1,98,123,106,95,104,138,133,119,92,135,137,129,102,130,125,109,
              113,140,99,134,101,117,100,93,96,110,115,128,122,139,112,126
        ],
        "min":91
      },
```

```
      "type":"skeleton",
      "hp_max":100,
      "dmg":20
    },
    {
      "size":{
        "w":130,
        "h":158
      },
      "hp":200,
      "acc":8,
      "pos":{
        "x":2388,
        "y":1216
      },
      "speed":650,
      "rot":0,
      "name":"Mike",
      "damping":0.75,
      "id":3,
      "ammo":{
        "max":156,
        "available":[
          154,144,153,146,143,145,152,155,142,150,156,151,148,147,149
        ],
        "min":142
      },
      "type":"robot",
      "hp_max":200,
      "dmg":60
    },
    {
      "delay":5,
      "pos":{
        "x":716,
        "y":1575
      },
      "spawn_pos":{
        "x":1500,
        "y":1500
      },
      "size":{
        "r":129
      },
      "min_range":43,
      "id":44,
      "strength":3.44,
      "type":"blackhole",
      "max_range":387,
      "dmg":34
    },
    {
      "size":{
        "w":108,
        "h":132
```

```
      },
      "hp":150,
      "acc":10,
      "pos":{
        "x":1984,
        "y":617
      },
      "speed":700,
      "rot":0,
      "name":"IceYoshi",
      "damping":0.9,
      "id":1,
      "ammo":{
        "max":89,
        "available":[
          71,69,74,67,66,65,83,88,86,73,70,72,79,77,68,84,80,78,76,87,85,89,8
              1,82,75
        ],
        "min":65
      },
      "type":"human",
      "hp_max":150,
      "dmg":30
    }
  ]
}

{
  "type":"start"
}

{ // Move Client -> Server
  "type":"move",
  "pos":{
    "y":1215.968505859375,
    "x":2387.930908203125
  },
  "pid":3,
  "rot":0,
  "vel":{
    "dx":0,
    "dy":0
  },
  "sqn":54
}

{ // Move Server -> Client
  "type":"move",
  "sqn":0,
  "objects":[
    {
      "pid":1,
      "rot":0,
      "vel":{
        "dx":0,
```

```
        "dy":0
      },
      "pos":{
        "x":1984,
        "y":617
      }
    },
    {
      "pid":45,
      "rot":0,
      "vel":{
        "dx":0,
        "dy":0
      },
      "pos":{
        "x":431.0001,
        "y":2244
      }
    },
    {
      "pid":47,
      "rot":0,
      "vel":{
        "dx":0,
        "dy":0
      },
      "pos":{
        "x":1257,
        "y":1703
      }
    },
    {
      "pid":49,
      "rot":0,
      "vel":{
        "dx":0,
        "dy":0
      },
      "pos":{
        "x":2180,
        "y":1632
      }
    },
    {
      "pid":51,
      "rot":0,
      "vel":{
        "dx":0,
        "dy":0
      },
      "pos":{
        "x":2528,
        "y":1890
      }
    },
```

```
    {
      "pid":53,
      "rot":0,
      "vel":{
        "dx":0,
        "dy":0
      },
      "pos":{
        "x":932.0001,
        "y":2323
      }
    },
    {
      "pid":55,
      "rot":0,
      "vel":{
        "dx":0,
        "dy":0
      },
      "pos":{
        "x":1085,
        "y":2177
      }
    },
    {
      "pid":57,
      "rot":0,
      "vel":{
        "dx":0,
        "dy":0
      },
      "pos":{
        "x":2426,
        "y":800
      }
    },
    {
      "pid":59,
      "rot":0,
      "vel":{
        "dx":0,
        "dy":0
      },
      "pos":{
        "x":1064,
        "y":2546
      }
    },
    {
      "pid":61,
      "rot":0,
      "vel":{
        "dx":0,
        "dy":0
      },
```

```
      "pos":{
        "x":666,
        "y":2408
      }
    },
    {
      "pid":63,
      "rot":0,
      "vel":{
        "dx":0,
        "dy":0
      },
      "pos":{
        "x":309,
        "y":930.0001
      }
    }
  ]
}

{
  "type":"station_status",
  "enabled":false,
  "station_id":157,
  "transfer":false
}

{
  "type":"pause",
  "pid":1 // pid only when Server -> Client
}

{
  "type":"state_sync",
  "state":"playing",
  "setup":{
    // Setup with up-to-date object data
  }
}

{
  "type":"object_respawn",
  "object":{
    "dmg":43,
    "rot":4.609109,
    "size":{
      "w":80,
      "h":80
    },
    "id":159,
    "hp":75,
    "type":"meteoroid2",
    "hp_max":75,
    "pos":{
      "x":1789,
```

```
      "y":973
    }
  }
}

{
  "type":"collision",
  "id1":5,
  "id2":55
}

{
  "type":"gameover",
  "players":[
    {
      "shots_fired":7,
      "shots_hit":5,
      "killed_by":"Meteoroid",
      "pid":1
    },
    {
      "shots_fired":0,
      "shots_hit":0,
      "killed_by":"COM",
      "pid":2
    },
    {
      "shots_fired":0,
      "shots_hit":0,
      "killed_by":"Blackhole",
      "pid":3
    }
  ]
}

{
  "type":"fire",
  "pos":{
    "x":1331,
    "y":2450
  },
  "pid":61,
  "rot":-1.507627,
  "fid":62
}
```