Carleton University COMP 4905 Honours Project Report

Student:

Yixiang Huang #101071528

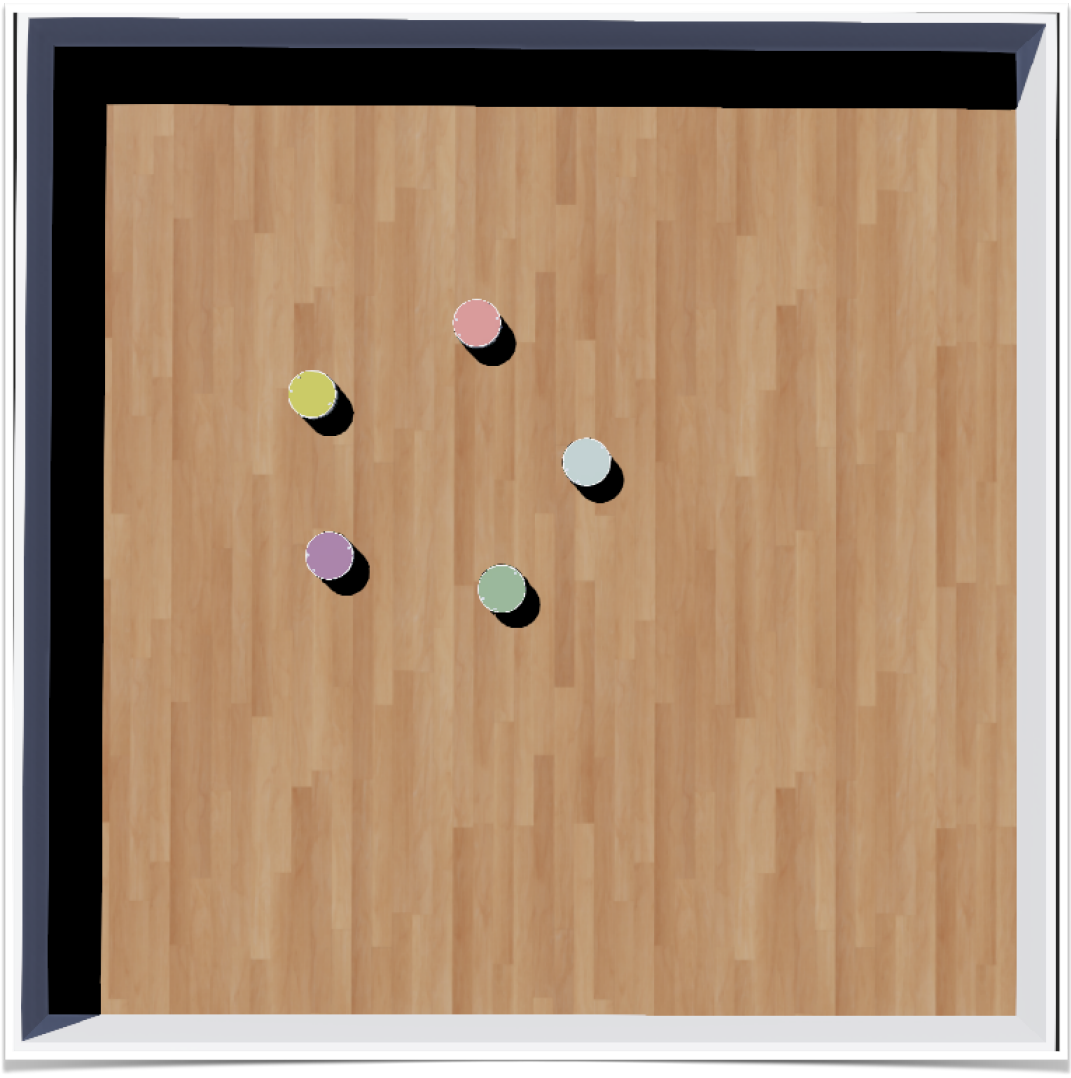
Supervisor:

Mark Lanthier

Date:

Fall 2021

2D Robot Light Show
Controller, Firmware, and Network



Abstract

This project was inspired by drone light shows that are recently very popular during festivals, hundreds of commercial drones each carrying a LED light, forming shapes, moving around to present stunning animations at night. These light shows are made with proprietary software and therefore not accessible, therefore I have been wanting to build my own DIY solution (although simpler) to perform a robot light show. The outcome of this project is an entire toolset for 2D robot light shows, including the software for designing animations, the choice of the middleware for communication, and the firmware that runs on robots.

The most important part of this project is the controller. It is responsible for providing an interface to design the frames, as well as controlling all robots to move simultaneously. At each frame, each robot will have different location and destination.

This report records in details about each of the components of the software I built, along with discussions about the strength and limitations of my approaches. Thanks to this project, I have learned a lot about building graphical interface efficiently with JavaFX, and gathered some new understanding of multi-thread computing in Java.

Acknowledgement

All codes in this project were developed on my own, with resources from online forums and publicly accessible materials. I sincerely appreciate the contributors to these resources.

Table of content

List of figures

Motivation

Intel Cooperation have been offering light shows which are performed by drones. In a total of 200 to 1200 drones [Intel Drone Light Shows, 2021], each of them carries a LED light which can change to any color and flies to predefined locations, performing stunning animations and offers eye-catching visual effects. These complex drone operations are all designed prior to the event, using Intel's proprietary software.

Intel Drone Light Show as part of Winter Games opening ceremony [Aviation Today, 2018]

Similar to the idea of drone light shows, but instead of making the hardware, I wanted to implement a simplified 2-dimensional light show in a simulated environment (Webots) with robots. In total of 10 to 50 robots each with a LED light, able to move to pre-defined locations and control the pixels (LED lights), forming a designated image (or animation).

Methodologies

1. Communication

The communication between the one instance of controller application and multiple instances of robot firmware, relies on networking. But due to complexities, robots are not directly connected to the controller application. Instead, both part separately connects to an event broker which allows them to communicate by sending and receiving messages.

1.1 Message

A message is formed by two parts, the payload and the topic.

The payload (an array of bytes, in this case it is a String) is the body of the message. Like texting, payload can be just encode/ decode to String for better readability. For example, a message from the controller to robots might be instructions for robots to perform some action. On the other hand, a message from robot to controller can be a signal of its current location or status.

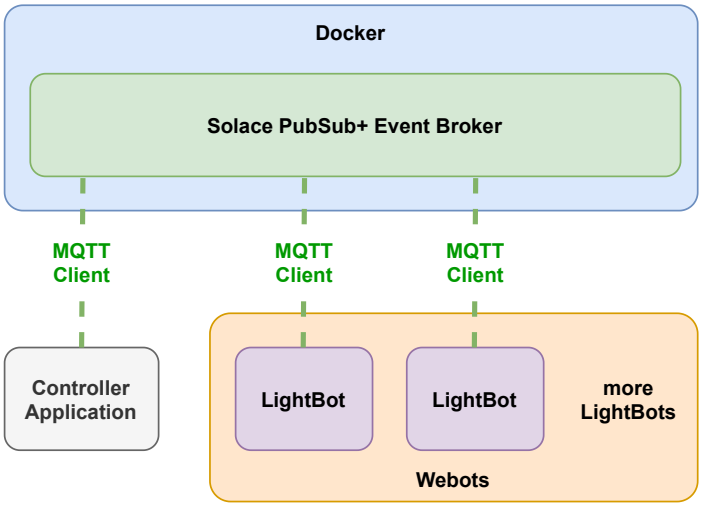
The topic of a message is set by the publisher when a message is sent, it decides who receive the message.

1.2 Publisher and Subscriber

A subscriber register topics to subscribe with the event broker through its MQTT client. When a message with corresponding topic published to the network, a callback function in subscriber's program will be executed automatically to process the message.

A publisher can publish message to the network containing any payload and topic, and don't need to worry about the delivery of the message. The message broker guarantees everyone who subscribed to the topic receives the message. If no one in the network subscribed to the topic at the time of publishing, the message will be destroyed. In this implementation, both the controller and robots are both publisher and subscriber, forming a bi-directional communication channel.

1.3 Message Broker



[Fig. 1] Connections between controller and robot

Similar to a post office, the message broker is a server that distributes messages according to the topic attached to the message. In this configuration, as shown in [Fig. 1], all robots and the controller only connects to the broker via their own Java MQTT client. Implementation-wise, the event broker is a Solace PubSub+ event broker that runs in a Docker container, no configuration is required.

1.4 Topics

> Topics are a means of classifying information, and in practice they're simply strings that are composed of one or more levels.
> Topics have the format a/b/c/.../n, where a, b, c and so on to n are identifiers in a hierarchical scheme you've devised that allows you to classify your information. For example, if you're publishing information on household pets you might create topics like animals/domestic/cats and animals/domestic/dogs to organize the content you're sending out.
> [Solace Documentation, 2021a]

Messages are routed by the broker according to topics, a topic is a String that indicates which category the message belongs to. One message belongs to exactly one topic. Both the controller and robots subscribes to a certain set of topics at startup, the communication begins within these topics.

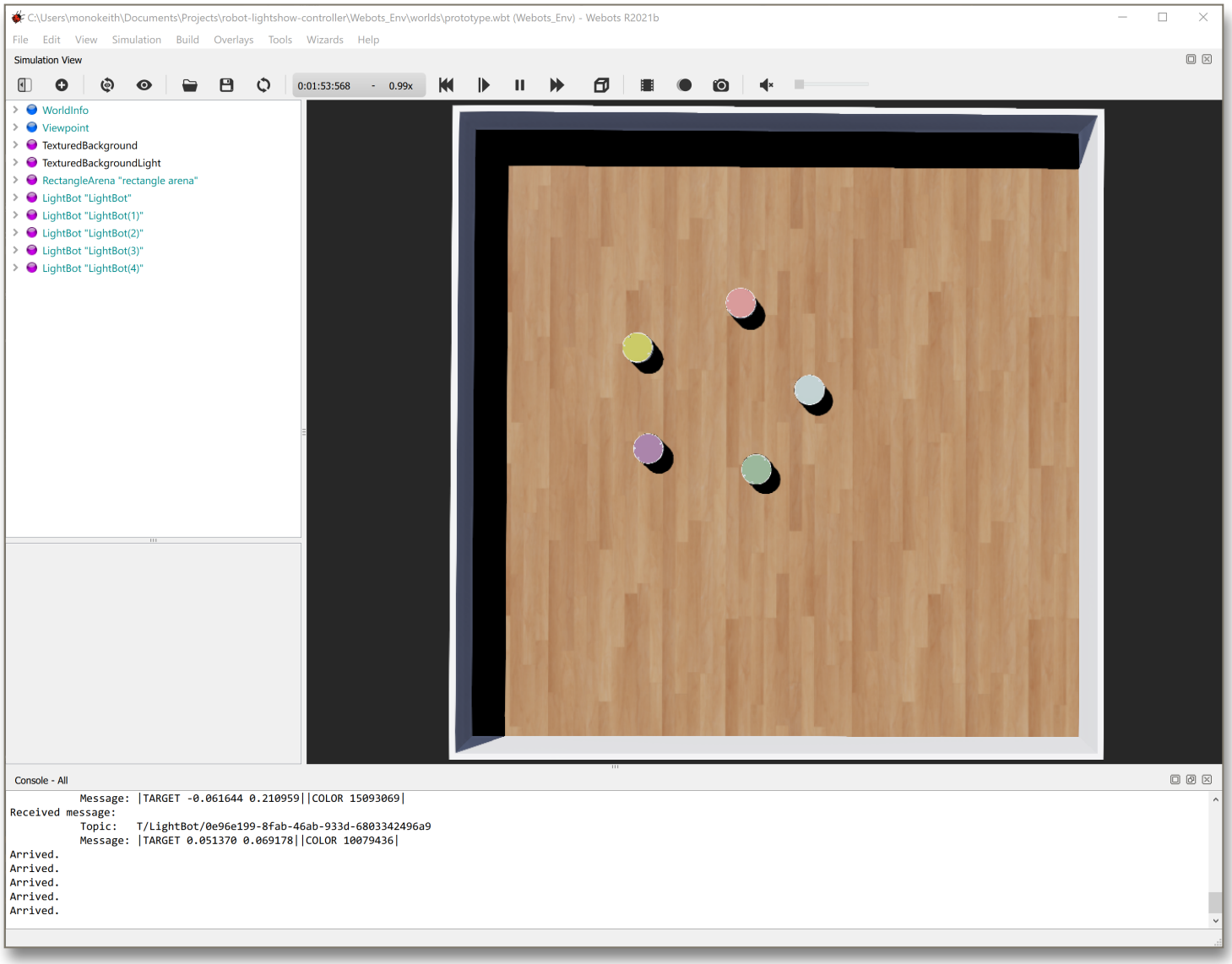
More specifically, the controller initially subscribe to the topic **T/LightBot/UUID** and wait for robots to report their UUID. The UUID of a robot was randomly generated as a part of the startup process, the robot also subscribes to the topic **T/LightBot/<UUID>**, which allows others (in this case it's the controller) in the same network to send a message containing one or many commands that will only receive by this robot. The controller maintains a set of connected robots, once the controller receive a message from this **T/LightBot/UUID** topic, it will trigger a registration of a BotPixel to this specific UUID.

There are two more topics for other purposes, **T/LightBot/Broadcast** and **T/LightBot/Arrival**. The Broadcast topic (subscribed by all robots) was useful if the controller starts after robots, or just need to check if any robot dies, it was used to trigger the robots to report their UUID, so the controller can refresh the connection status. The Arrival topic (subscribed by the controller) was used for robots to report its status when a requested action is complete.

2. Robots and Environment

2.1 Environment

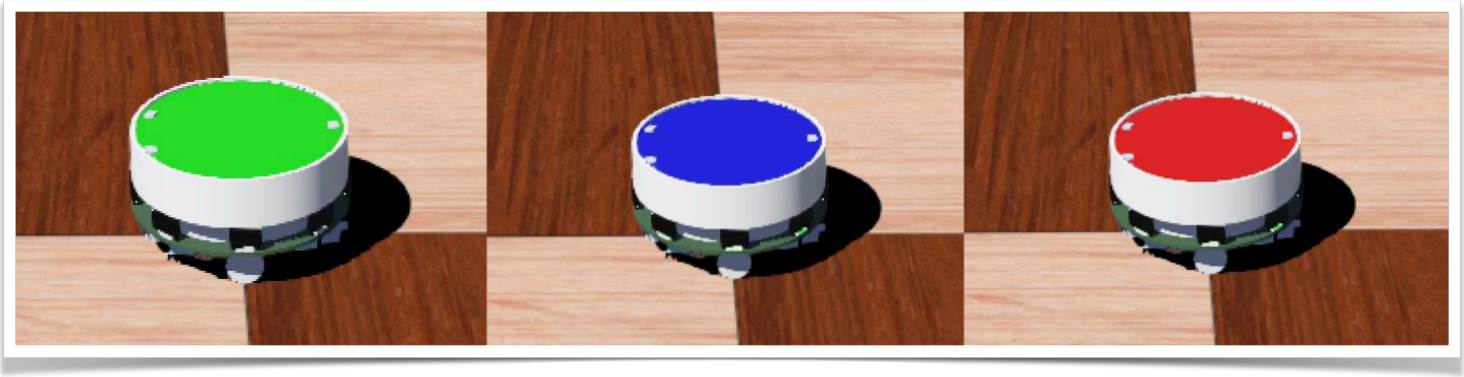
The environment that robots lives in is a simulator called Webots (shown in [Fig. 2]), it emulates an environment with real-world physics and offers support to multiple programming language. It is very convenient and efficient to test robots in this environment, most designs that worked in this simulator should offers similar effect in a real-world environment.



[Fig. 2] Test environment with 5 LightBots in webots

2.2 Hardware

The robot was modified from a Gctronic Elisa-3 robot [https://cyberbotics.com/doc/guide/elisa3], with added compass and GPS module, and re-packaged into a PROTO file. This robot have a rounded shape, and a big RGB LED on top, which makes it one of the best choices for a light show.



[Fig. 3] Screenshot of Elisa-3 in Webots showing different colors

2.3 Limitations of current design

A drawback of the current firmware implementation is that it is heavily relying on the GPS location, which is very accurate in the simulator. The robot relies completely on the accurate location given by the simulator to constantly adjust orientation when moving to destination location. However, in a real-world environment, GPS won't be able to offer this level of accuracy, some position estimation algorithm based on accelerometer or wheel spin is needed to provide this data. Since the main focus of this project is the controller software rather than the accuracy of movements, I did not spend time figuring this out.

2.4 Firmware

The firmware of the robot receives instructions from controller and perform corresponding actions. It was implemented in Java with a Paho MQTT library as the only dependency. The firmware and its dependencies were compiled and output as JAR files, which can be conveniently add to Webots's controller directory.

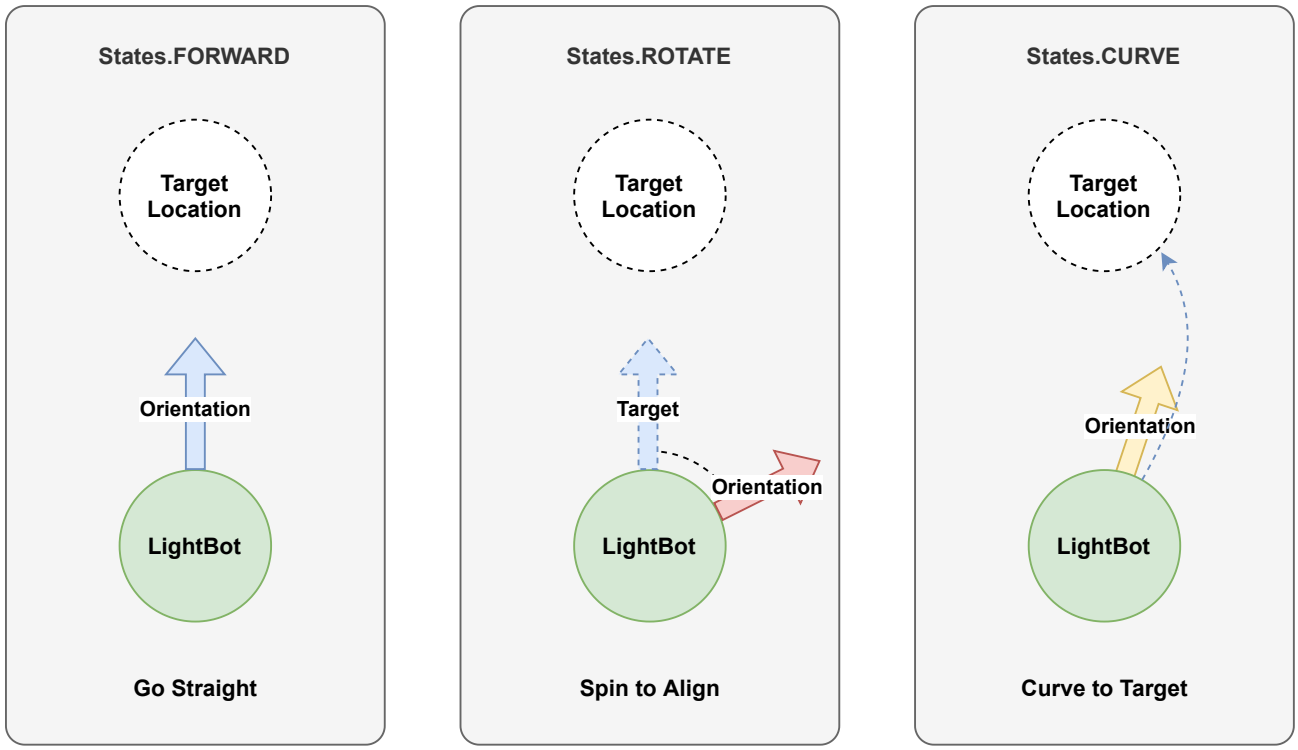
2.4.1 Threads

The firmware runs in two threads, a message processing thread and an action thread.

The action thread is the main thread started by Webots, and have a loop associated with time step in the Webots environment, the loop run once per time step. It is used for constantly adjusting wheel spin according to its current orientation and location.

The message processing thread waits for and process messages received from the MQTT client, and updates the state of the robot (for example, target location) accordingly.

2.4.2 Target Location



[Fig. 4] Possibilities of target location

In my implementation, the only way to set or update target location is through messaging. The message processor thread updates the target location recorded in the Location class.

The Location class is responsible for determining the orientation and distance to the goal location, it keeps track of the delta between the current location and the target location, and computes the shortest path every time step. These information will then be use by the action thread to adjust the movement of the robot. To ensure a smooth movement, when the orientation offset is small, the robot curves towards the goal, instead of stop and spin to re-align. This mechanism can ensure the robot only spins to align at the beginning of a movement, which is not noticeable because of its rounded shape.

2.4.3 RGB LED

> In the case of an RGB LED (gradual field set to TRUE and color field containing an empty list), the value parameter indicates the RGB color of the LED in the range 0 (off or black) to 0xffffff (white). The format is R8G8B8: The most significant 8 bits (left hand side) indicate the red level (between 0x00 and 0xff). Bits 8 to 15 indicate the green level and the least significant 8 bits (right hand side) indicate the blue level. For example, 0xff0000 is red, 0x00ff00 is green, 0x0000ff is blue, 0xffff00 is yellow, etc.
> [Webots reference Manual, 2021a]

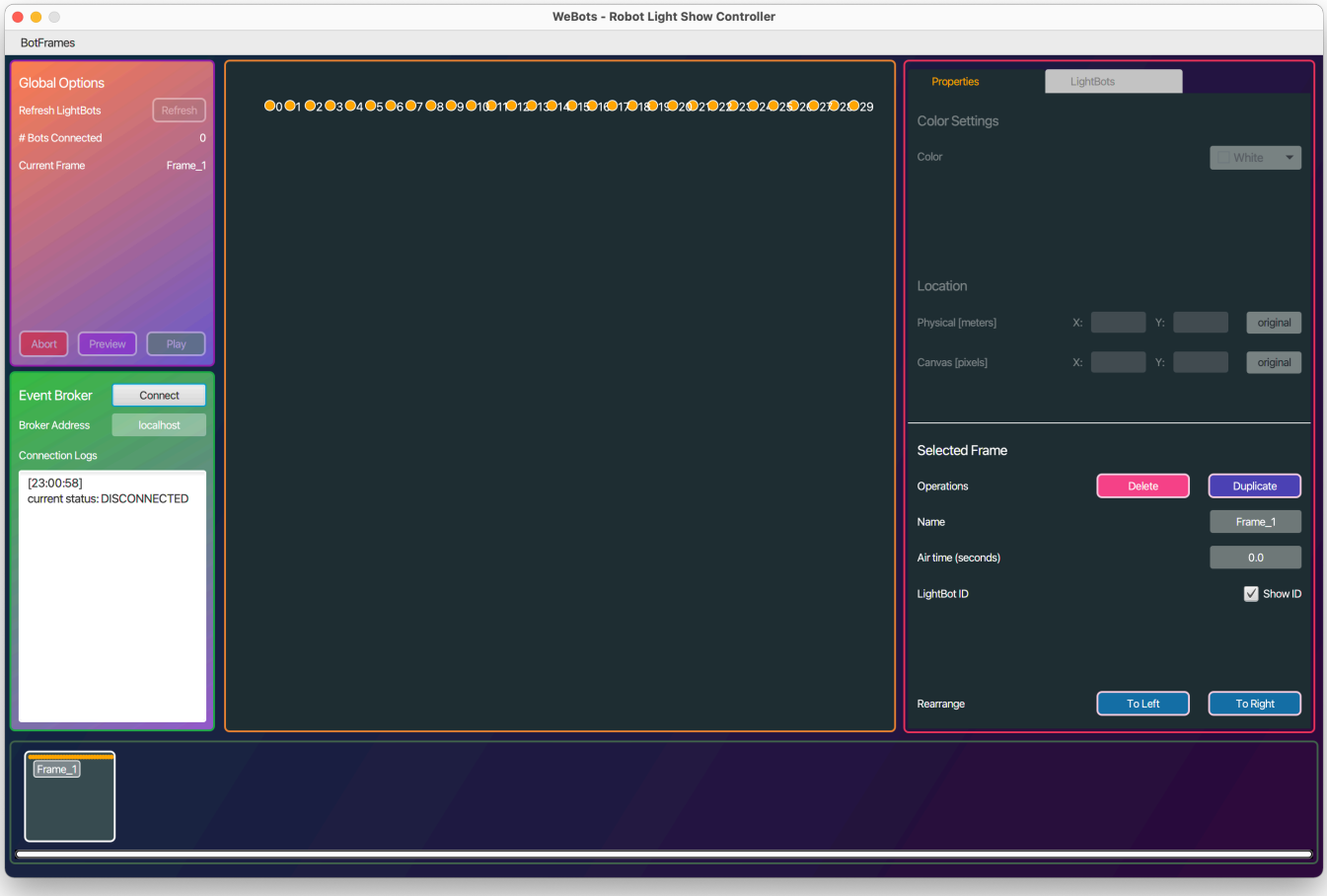
The LED on top of the robot is also directly controlled by messaging. As described in the documentation above, Webots takes a single integer representing the 8-bit per color RGB value. This value was calculated by the following formula:

$$RGB = (R \cdot 65536) + (G \cdot 256) + B$$

Eq.(1)

3. Controller application

3.1 Functionality



[Fig. 5] Screenshot of controller application

The controller application is a software build with JavaFX for purposes of both designing the animations and controlling the robots to play the animations.

Initially there are a fixed amount of circles on the screen, each circle represent a robot which represents a BotPixel, the canvas represent the floor in Webots. User can select one or more pixels by clicking on the circle, and de-select by clicking on empty space. When one or more pixels are selected, they can be drag-and-drop to relocate. When a single pixel is selected, user have an additional choice to type in the desired location's coordinates to be more accurate. Animations are formed by multiple frames,

all frames contains the same amount of BotPixels, representing the color and target location of each robot at a given point of time. When playing the animations, the controller will send locations of pixels of the current frame to the corresponding robots. Before continuing to the next frame, it waits until all robots have arrived their destinations.

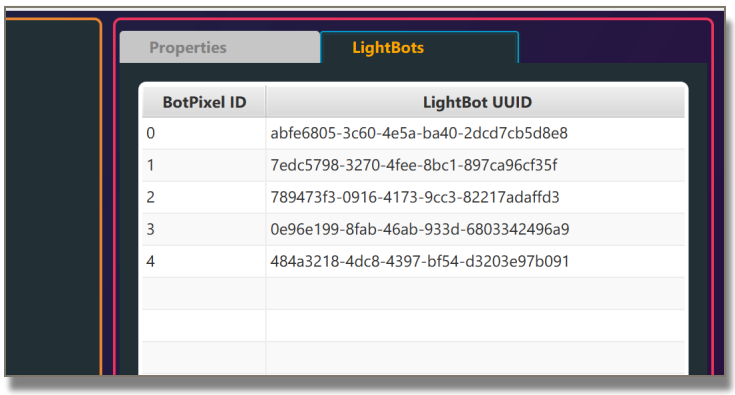
3.2 Design pattern

The graphical user interface of this application was built with JavaFX with help from Fxml and SceneBuilder. It follows the MVC design pattern in order to organize code. Object oriented programming is used wherever possible.

3.3 Connection

As indicated in section 1.3, the controller does not directly communicate with any robots, it only connect to the message broker with a single MQTT client. In this way, robots and controller can start with any order, rebooting robots or controller will not result in requiring to reboot the other. Because they all connect to the message broker, the controller also benefits from not having to manage all the connections and distribution of messages.

In the situation of controller starts after all robots have started, which means the controller missed all the UUIDs of robots, it simply sends a request of reporting UUID by sending the request message to the T/LightBot/Broadcast topic. In result, all robots receives this request and report their UUID as usual.



[Fig. 6] Mapping of pixel ID and robot UUID

A map which correlates a BotPixel to a connected LightBot's UUID is automatically maintained. This is useful when distributing target locations to robots. Instead of broadcasting to all robots, this ensures that only the correct robot receives the instructions, in order to save bandwidth and processing power.

3.4 Threads / Synchronized class

Multiple threads and synchronized functions were implemented in the controller. Two of these threads were responsible for sending and receiving messages.

3.4.1 Incoming / outgoing message queue

Synchronizing messages was critical. When a message is received by the MQTT client, a callback function executes to put it in a thread-safe message queue, which in this case it's just a LinkedList in a synchronized class. This is because when multiple messages arrives at roughly the same time, there are chances that multiple callback functions run at the same time. If these messages are being processed in parallel, there are chances of causing race condition, because the message processing function sometimes update the state machine. It is dangerous for this to happen, the program may get stuck in some weird state and then causes a freeze. Furthermore, the Paho MQTT client that I used was not intended to be called asynchronously, and doing so will cause internal race condition, which could leads to a complete freeze of the entire messaging thread.

Adding two thread-safe message queue, one for incoming messages, one for outgoing messages, solves this by processing incoming messages synchronously.

```
// Queue to store messages to be published
private final LinkedList<BotMessage> publishQueue;

// Queue to save received messages
private final LinkedList<BotMessage> receiveQueue;
```

This can be done easily in Java by using the synchronize keyword as shown in following code (this syntax only works in Java):

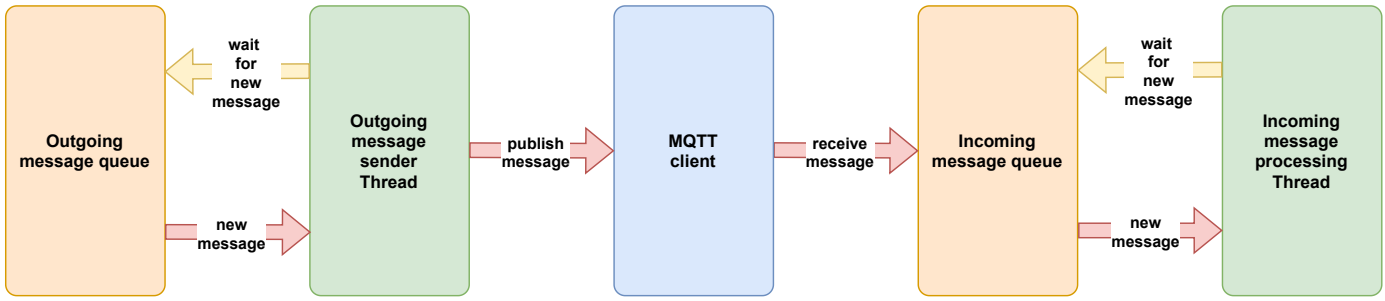
```
// Loop in publisher thread, iteratively publish messages
while(true){
    synchronized (publishQueue) {
        // Wait for message to publish
        while (transmitter == null || publishQueue.isEmpty()) {
           publishQueue.wait();
           if (terminateFlag) return;
        }
        // Call MQTT client to publish
        ……
    }
}
```

```
// Called when play animation thread publish targets
synchronized (publishQueue) {
    publishQueue.add(message);
    publishQueue.notifyAll();
}
```

By using synchronized keyword with the syntax above, access to the publishQueue is being synchronized, while the publishing thread is publishing the message, other threads cannot access the publishQueue. This is very similar to using a lock, but instead of a lock object that needs to be managed separately, this syntax allows to use the collection object itself as the lock.

The receiveQueue is also being used in the same way, since the MQTT client spawns a new thread when a message is received, using the synchronized block with a receiveQueue can synchronize the processing of received messages. And therefore, prevents conflicted state updates.

The process of sending messages also involves an outgoing message queue, but the purpose of this design was not to prevent race condition, it's only for achieving better performance. Sending a message requires communication to the event broker, network lag may cause congestions. In this case, since the delivery of a message is guaranteed and there's no need to wait for a response, it's faster to offload the process of sending messages to another thread. And this was achieved by setting up a thread-safe outgoing message queue, similar to the incoming message queue.



[Fig. 7] Simplified chart of message related threads

As shown in [Fig. 7], the incoming message queue and the outgoing message queue have their own synchronized mechanism, the controller can send and receive at the same time, while these 2 queue offers the buffer that enables MQTT client to send / receive at its own schedule.

3.4.2 Play animation thread

There is also another thread being generated and terminated during life-cycles of the program, the thread responsible for playing animation. When user click the 'play' button, a new thread is generated to run the procedures.

```
do {
    publishTargets();
    arrivalManager.waitForArrival();
    // Quit signal
    if (getGlobalState() != GlobalOptionControl.State.PLAYING)
        return;
    // Wait airtime
    double airTime = getCurrentFrame().getAirTime();
   sleep(airTime);
   } while (nextFrame());
```

The program will first publish all destinations to each of the robots, and wait for robots to arrive. When all robot arrives, the arrival manager releases the waitForArrival() function and the execution continues. If a frame is setup with an amount of airtime, the thread pauses. Finally it loops back if there's a next frame.

In the situation where the 'play' is aborted, the state will change and waitForArrival() returns immediately, which terminates this thread.

Airtime is useful in situations where user wants the frame to stay longer than robot's movements. For example, when all robots are not moving between frames, but their color changes, airtime ensures that user can see the color stays. For example, robots may be setup as a digital 8 to display any numbers, by switching the light on or off, airtime in this case would be the amount of time that the number stays visible before proceeding to next frame.

3.4.3 Arrival manager

The controller application have a dedicated synchronized class to wait for LightBot to arrive targeted destinations when a frame is being played. This is a critical section because it is being operated by multiple threads at the same time. It is responsible for both processing arrival messages, as well as pausing the play animation thread (3.4.2) until all robots arrives. Further more, the application also allow user to abort, in case when robots collides and stuck, that they will never arrive the destination.

This class offers a method to initiate the pending states by storing robots' UUIDs and their destinations. This method is being called only when robots gets new instructions.

```
public synchronized void setPending(Map<BotPixel, UUID> pixelMap) {
    locationMap.clear();
    pixelMap.forEach(
        (pixel, uuid) -> locationMap.put(uuid, pixel.getPixelLocation()));
    pending = locationMap.keySet();
    initialized = true;
    quitWaiting = false;
    notifyAll();
}
```

When a message about robot arriving, another method will be called to update the pending list, if robot arrives at correct location, otherwise a warning message will be presented.

```
public synchronized void arrive(UUID uuid, Point2D curLocation){
    if (!initialized) return;
    Point2D targetLocation = locationMap.get(uuid);
    if (targetLocation == null) {
        log(String.format("UNEXPECTED arrival from: %s", uuid));
        return;
    }
    double distance = targetLocation.distance(curLocation);
    if (distance > posAccuracy) {
        log(String.format("WARNING! misaligned LightBot: %s", uuid));
        return;
    }
    // Arrived
    pending.remove(uuid);
    log(String.format("remaining: %d, just arrived: %s", pending.size(), uuid));
    notifyAll();
}
```

Lastly, the most important waitForArrival() method appeared in (3.4.2).

```
public synchronized void waitForArrival(){
    while(!pending.isEmpty()){
        wait();
        if (quitWaiting){
            log("wait for arrival aborted");
            break;
        }
    }
    // Stop accepting arrive() once complete
    reset();
}
```

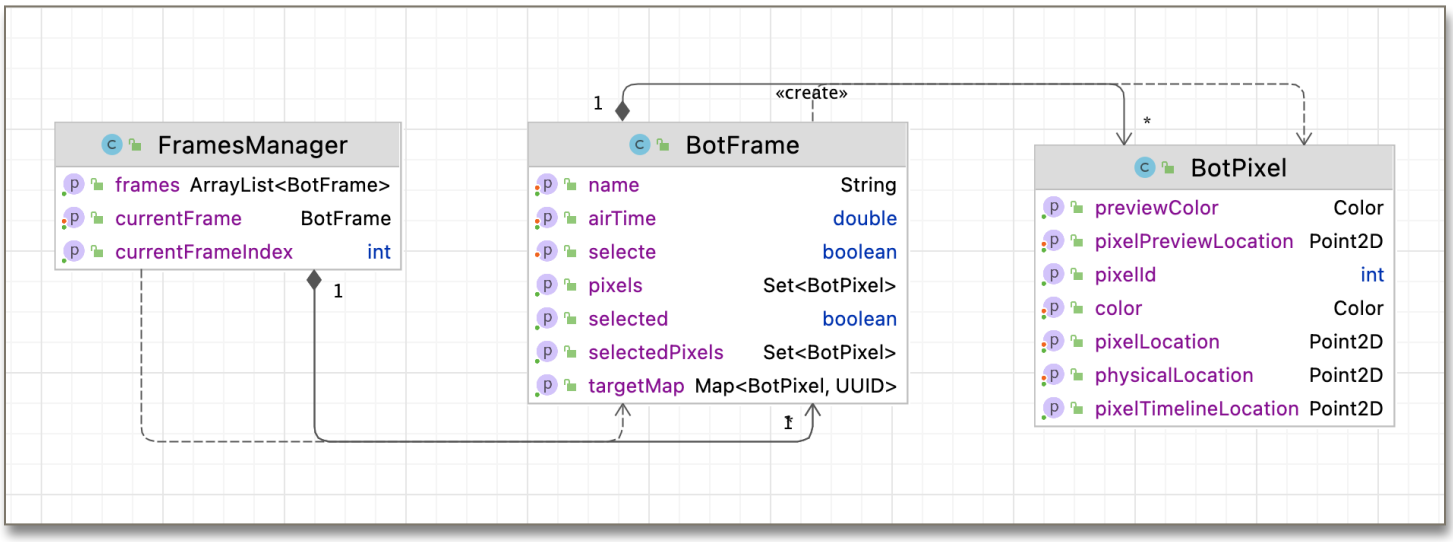
This function will pause the caller's thread by calling wait(), only when the status of pending robots updates, the notifyAll() function in arrive() method wakes up this thread. In situation of either all robot returns, or user abort the play, this function returns immediately and allows execution of the play thread to continue.

3.5 Serialization / Deserialization

The last part of the implementation was allowing user to save and load frames as files. This was achieved by utilizing the Gson library.

> Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson can work with arbitrary Java objects including pre-existing objects that you do not have source-code of.
> [Gson, 2021]

This library automatically serializes an object into json, and can do the same in reverse. Although there was one limitation, it does not work on reference of an object.



[Fig. 8] UML class diagram of BotFrame related classes

In this case, the hash set selectedPixels, should hold the same reference of selected BotPixels as they in the pixel set. However, when it is being deserialized, Gson does not keep track of this situation and

creates a deep copy of the same pixel. So the selections would not be recognized when loading from a file. Currently I have not found a solution to this. A possible workaround is to recognize BotPixels by their pixelIds.

Apart from this issue, another difficulty during the process was that TypeAdapters were required for both the Color class and Point2D class, which they are not natively supported. A type adapter will be matched and used when Gson serialize the BotFrame object as a tree. This was setup by the following syntax:

```
GsonBuilder builder = new GsonBuilder();
builder.registerTypeAdapter(Point2D.class, new PointAdapter());
builder.registerTypeAdapter(Color.class, new ColorAdapter());
```

A TypeAdapter is a class that implements both the JsonSerializer and JsonDeserializer interface. Each of these interfaces have only one function, to convert the object to a JsonElement, or in the reverse direction.

```
public class ColorAdapter implements JsonSerializer<Color>, JsonDeserializer<Color>
{

    private static final String R = "R", G = "G", B = "B";

    @Override
    public JsonElement serialize(Color color, Type type, JsonSerializationContext
context) {
        JsonObject jsonObject = new JsonObject();
        jsonObject.add(R, new JsonPrimitive(color.getRed()));
        jsonObject.add(G, new JsonPrimitive(color.getGreen()));
        jsonObject.add(B, new JsonPrimitive(color.getBlue()));
        return jsonObject;
    }

    @Override
    public Color deserialize(JsonElement jsonElement, Type type,
JsonDeserializationContext context) throws JsonParseException {
        JsonObject jsonObject = jsonElement.getAsJsonObject();
        double r = jsonObject.get(R).getAsDouble();
        double g = jsonObject.get(G).getAsDouble();
        double b = jsonObject.get(B).getAsDouble();
        return Color.color(r, g, b);
    }
}
```

A Json element can contain many Json objects as either primitive types or nested Json elements. In this case the Color class is simple enough, that it only requires 3 values to restore the state.

Results

1. Experience

This is a valuable experience, I've never had any chance like this to implement such an 'huge' project all on my own. It again reminds me the importance of following design patterns as well as planning before coding.

During the project, I've learned building graphical user interface in JavaFx with SceneBuilder and fxml. This tool made it so much easier to design a UI comparing to the old way of coding in the dark with plain code. I learned that the UI in JavaFx could be very similar to HTML and CSS, as they are very similar in terms of syntax and properties.

The project also had given me the chance to work with messaging and event-driven architecture. I've heard the usage and examples a lot during the COOP terms at Solace, but I haven't had the chance to actually use it to build something until this project. Messaging made connection between one controller and multiple robots so much easier. At the end of the term, I recognized the potential of this technology, and really think it could play an important role in the field of robotics. It is especially useful when a large number of data needs to be routed to multiple independent systems, in real time.

I was also able to transfer knowledges from SYSC 3303 in to the project. When the system consists of multiple subsystems, the most elegant way to implement is to let each subsystem have its own thread. This vastly lower the complexity of the system. Since multiple subsystems exists in both the controller and the LightBotFirmware, they both were benefit from adopting such design.

One of the most interesting experience was using Gson to serialize and deserialize Java objects. Although they did a terrible job in documenting this tool, it worked really well.

I really appreciate this opportunity for me to improve and apply what I have learned to build something fun.

2. Deliverable

At the end of the term, I completed the controller application, firmware, and built a test environment in Webots. The result is a fully working toolset for 2D robot light show. Although there are lots of room for improvements, it is functional to be used.

The controller application:

- Connect to the message broker and maintain a list of 30 connected robots
- Create new frames by duplicating the previous one
- Rearrange a frame to left or right by 1 position at a time
- Change name of a frame
- Airtime: how long a frame stays when playing the animation
- Edit pixel location and pixel color
- Play animation by iterating through each frames, abort (cancel) a playing animation
- Preview a frame
- Load and save frames as JSON files

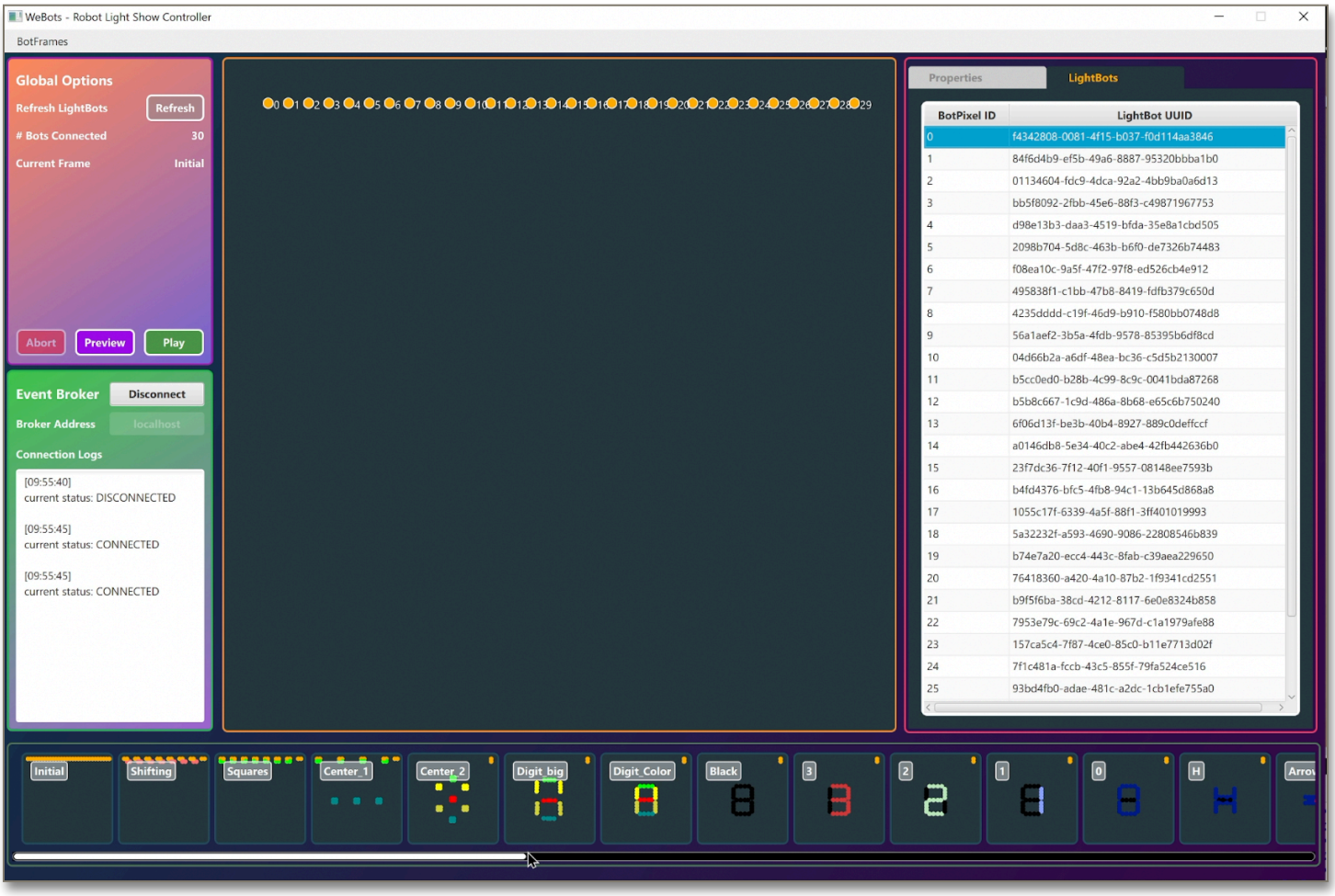
The firmware:

- Packaged as 2 JAR files, which could be executed by Webots natively
- Connects to the message broker automatically with a random UUID
- UUID command: Report UUID and location
- TARGET command: Set a target location, the robot will immediately move towards the coordinate
- COLOR command: Change color of the robot's LED

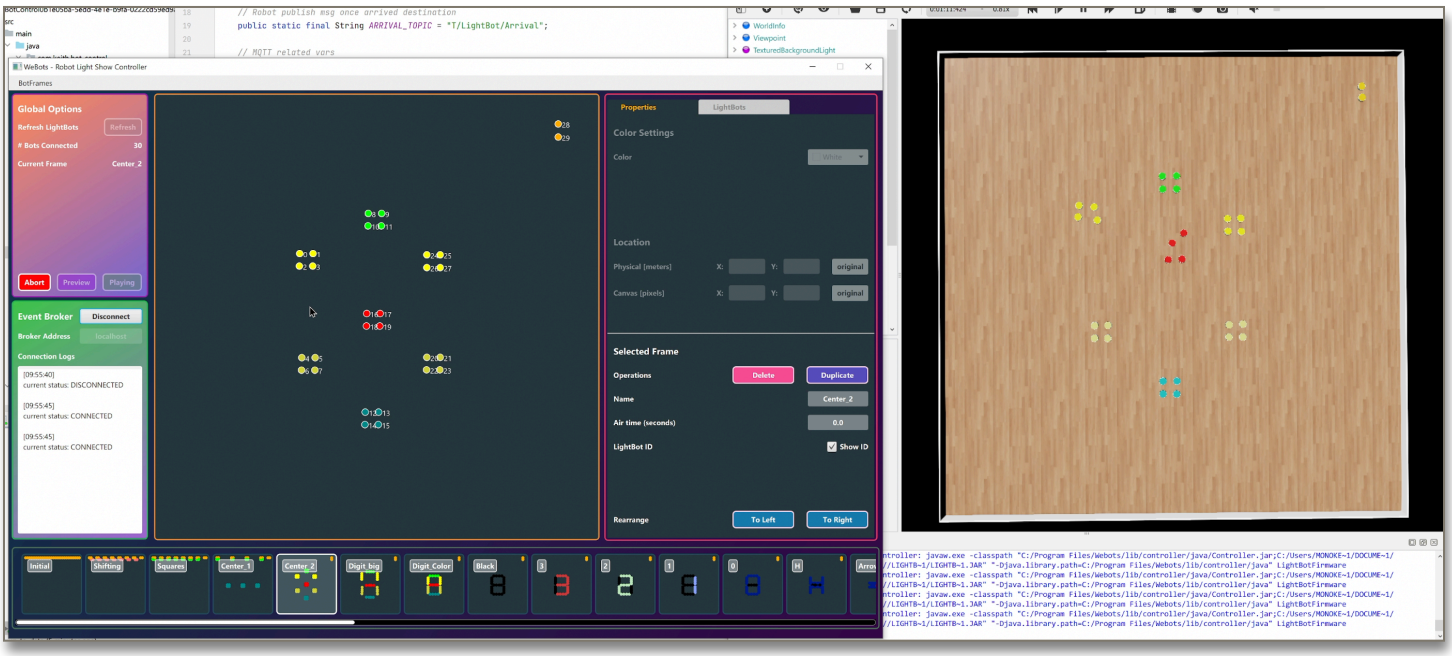
Environment in Webots:

- Modified Elisa-3 robot to contain GPS and Compass
- 3 meters by 3 meters square floor
- 30 robots in a row on edge of the floor

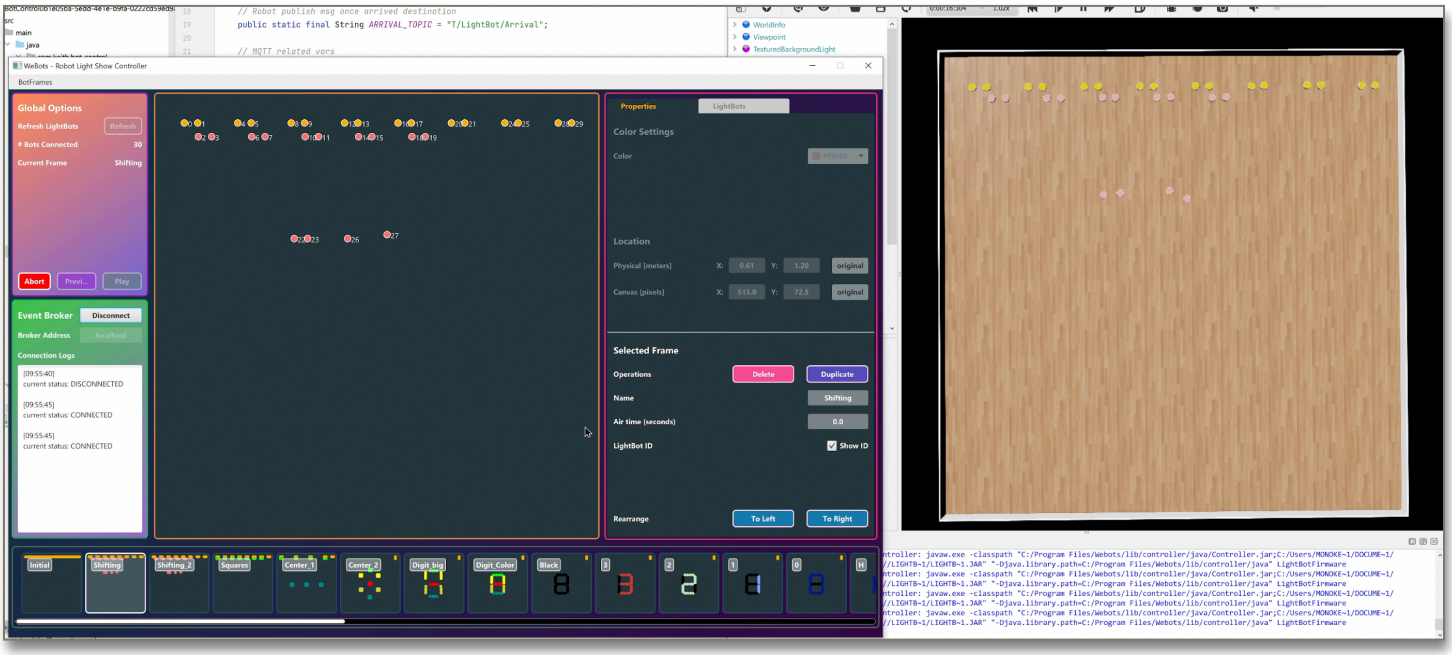
3. Snapshots of deliverable (testing the product)



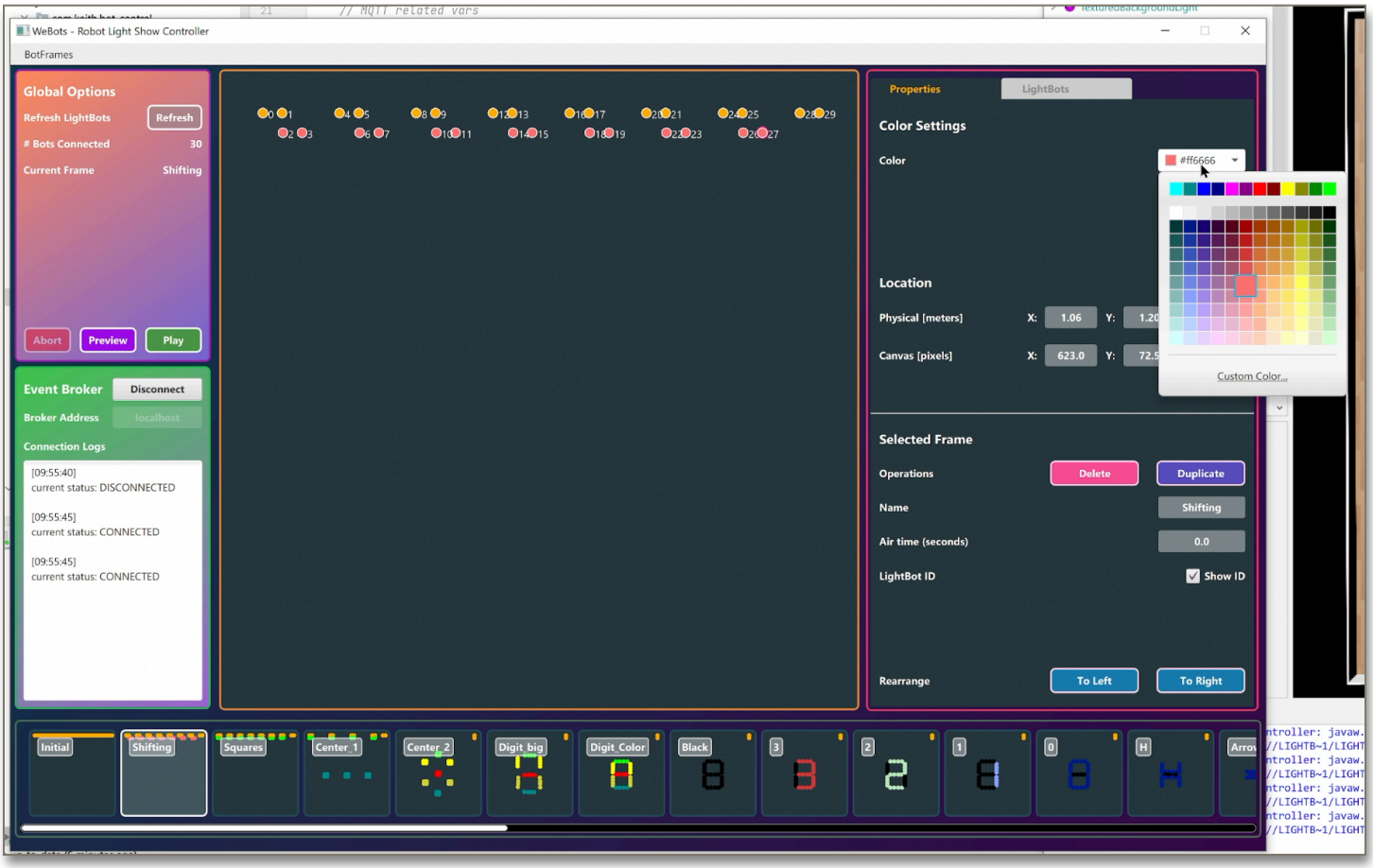
[Fig. 9] All robots connected, mapping of pixel ID to UUID shown



[Fig. 10] Playing animation, robots moving towards target



[Fig. 11] Preview a frame, robots moving towards target



[Fig. 12] Editing a pixel, change color

Reference

Intel Drone Light Shows (2021). Web Archive (Dec 15, 2021)
https://inteldronelightshows.com

Aviation Today (2018). Web Archive (Dec 15, 2021)
https://www.aviationtoday.com/2018/02/23/intel-drones-placeholder/

Solace Documentation (2021a). What are Topics?, Web Archive (Dec 15, 2021)
https://docs.solace.com/Basics/Understanding-Topics.htm

Webots reference Manual, (2021a). LED, Web Archive (Dec 15, 2021)
https://cyberbotics.com/doc/reference/led?tab-language=java

Gson, (2021). Gson, Web Archive (Dec 15, 2021)
https://github.com/google/gson