Arduino Lab 3

Knut Ola Nøsen

***Abstract*—The project explores multitasking by creating a reaction-based game. The goal is to demonstrate how the architecture of an Arduino program can be influenced by its lack of proper multithreading. The program contains not only parallel operations, but also the use of Serial input for manipulating the game state.**

***Index Terms*—Arduino, RGB, LED, Random, RAMP, Sequential, Multithreading**

I. Theory

THE game centers around reaction time. Two players have a button each, and when the center RGB LED switches from red to green, the first player to press the button gets points based on the players reaction time. If the LED turns blue instead of green, the player who clicks a button will loose points. If noone clicks, the game will continue with a new round after 1 second. The game ends when a player reaches a score of 10 points. After each round is won/lost, a tone will play and leds flash to celebrate the winner or bully the looser.

II. Methods

A. Hardware

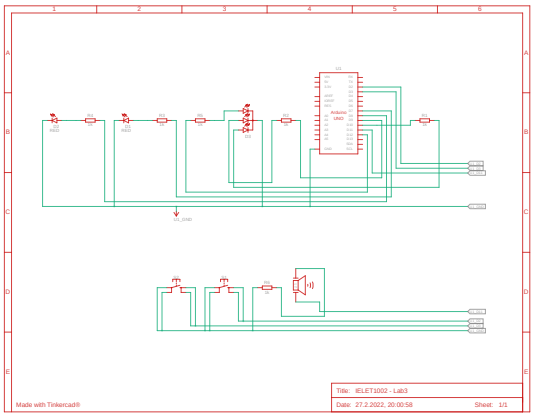

Fig. 1. Wiring Diagram

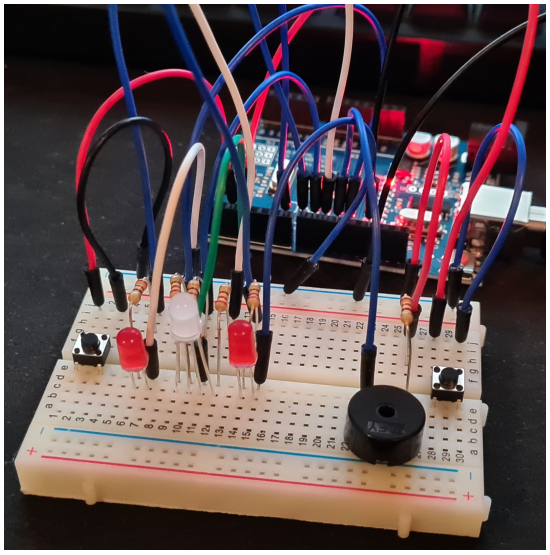

Fig. 2. Lab3 Circuit

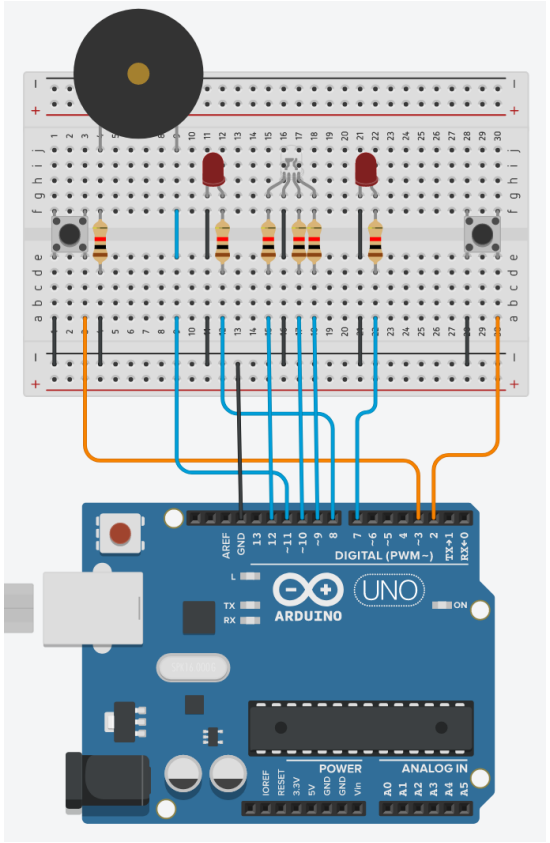

Fig. 3. Tinkercad

B. Software

Lets start by creating some configs to keep our code free of ”magic constants”. By keeping these configs in separate files, we avoid cluttering our logic in the main.cpp file.

Listing 1. Range.h

```
struct Range
{
   const int minValue;
   const int maxValue;
   Range(int minValue, int maxValue) : minValue(minValue), maxValue(maxValue)
   {
   }
};
```

Listing 2. PlayerConfig.h

```
struct PlayerConfig
{
   const byte ledPin;
   const byte buttonPin;
};
```

Listing 3. RgbLedConfig.h

```
struct RgbLedConfig
{
   const byte redPin;
   const byte greenPin;
   const byte bluePin;
};
```

Listing 4. ApplicationConfig.h

```

struct ApplicationConfig
{
   const int baudRate = 9600;
   const int buttonDebounceTime = 50;
   const byte randomSeedPin = A0;
   // NB! Update this with state.players
   const int numPlayers = 2;
   const PlayerConfig player1 = {.ledPin = 8, .buttonPin = 3};
   const PlayerConfig player2 = {.ledPin = 7, .buttonPin = 2};
   const RgbLedConfig rgbLed = {.redPin = 10, .greenPin = 9, .bluePin = 12};
   const byte buzzerPin = 11;
   const Range winnerBuzzerPitch = Range(750, 2000);
   const int winningPoints = 10;
   // 30% should be written as 0.3
   const float trickRoundProbability = 3.0 / 10.0;
   const int trickRoundDuration = 1000;
   const int looserBuzzerPitch = 220;
   const int roundCompletionAnnouncementDuration = 3000;
   // The frequency the LED’s should blink after a win/loss
   const int fastBlinkFrequency = 5;
   // How long it should take for a green light to appear
   const Range roundTimeMs = Range(3000, 6000);
} const appConfig;
```

This gives us a global const appConfig when we include the ApplicationConfig.h file in our main sketch.

Listing 5. Timer.h

```
#ifndef Timer_h
#define Timer_h
#include <Arduino.h>

class Timer
{
private:
   long startTime;

public:
   Timer()
   {
      reset();
   }
   void reset()
   {
      startTime = millis();
   }

   unsigned long getElapsedTime()
   {
      return millis() - startTime;
   }

   /*
      Will check if the time since last reset()
      call is greater than the given time
      Note that while loopWait() automatically resets after
      the given time, this function does not
    */
   bool isFinished(const unsigned long durationMs)
   {
      return getElapsedTime() >= durationMs;
   }
   /*
   Will return false until the given time has passed
   Then it will return true and start counting down the same amount again
   Example:
      Timer timer;
      while(true) {
         if(timer.isTimePassed(1000)) {
            // Will run every 1000ms
         }
      }
   */
   bool loopWait(const unsigned long durationMs)
   {
      if (isFinished(durationMs))
      {
         reset();
         return true;
      }
      return false;
   }
};

#endif
```

We also create a Timer class to make time tracking a bit easier and more intuitive.

Now that all our external files are set up, we can start looking at the main.cpp file.

Listing 6. main.cpp - imports

```
#include <Arduino.h>
#include <ezButton.h>
#include <Ramp.h>

#include <RgbLedConfig.h>
#include <Range.h>
#include <PlayerConfig.h>
#include <ApplicationConfig.h>
#include <Timer.h>
```

We start by importing our external files and libraries.

Listing 7. main.cpp - Player

```
class Player
{
public:
   const PlayerConfig playerConfig;
   int points = 0;
   ezButton button;
   Player(const PlayerConfig playerConfig) : playerConfig(playerConfig),
       button(ezButton(playerConfig.buttonPin))
   {
   }

   void changePoints(int points)
   {
      this->points += points;
   }

   void resetPoints()
   {
      this->points = 0;
   }

   int getPoints()
   {
      return this->points;
   }

   void setup()
   {
      pinMode(playerConfig.ledPin, OUTPUT);
      pinMode(playerConfig.buttonPin, INPUT_PULLUP);
      button.setDebounceTime(appConfig.buttonDebounceTime);
   }
};
```

Next we create a Player class that can keep track of its score, and put all information about that player in one place. This also lets us create a cleaner setup function with less room for error.

Listing 8. main.cpp - RgbLed

```
class RgbLed
{
   RgbLedConfig rgbLedConfig;

public:
   RgbLed(const RgbLedConfig rgbLedConfig) : rgbLedConfig(rgbLedConfig)
   {
   }

   void red(bool state)
   {
      off();
      digitalWrite(rgbLedConfig.redPin, state);
   }

   void green(bool state)
   {
      off();
      digitalWrite(rgbLedConfig.greenPin, state);
   }

   void blue(bool state)
   {
      off();
      digitalWrite(rgbLedConfig.bluePin, state);
   }

   void off()
   {
      digitalWrite(rgbLedConfig.redPin, LOW);
      digitalWrite(rgbLedConfig.greenPin, LOW);
      digitalWrite(rgbLedConfig.bluePin, LOW);
   }

   void setup()
   {
      pinMode(rgbLedConfig.redPin, OUTPUT);
      pinMode(rgbLedConfig.greenPin, OUTPUT);
      pinMode(rgbLedConfig.bluePin, OUTPUT);
   }
};
```

Properly managing the RGB LED turned in to a mess, and therefore i created helper functions to ensure that the correct state was always set. This however turned in to a number of functions, and for this reason i decided to create a RgbLed class. This also has a setup function like the Player class.

Listing 9. main.cpp - SerialCommand

```
enum class SerialCommand
{
   START = 's',
   STOP = 'q',
   RESET = 'r',
   HELP = 'h',
};
```

We create an enum of comments that the user may send to the arduino through the serial port.

Listing 10. main.cpp - GameState

```
enum class GameState
{
   IDLE,
   RUNNING,
};
```

We also create an enum for the GameState, that the serial commands will change.

Listing 11. main.cpp - ApplicationState

```
118 struct ApplicationState
119 {
120    // NB! Update this with appConfig.numPlayers
121    Player players[2] = {
122       Player(appConfig.player1),
123       Player(appConfig.player2)};
124    RgbLed rgbLed = RgbLed(appConfig.rgbLed);
125    GameState gameState = GameState::IDLE;
126 } state;
```

Now that our datatypes are specified, we can create our ApplicationState struct, to keep track of the mutable state in our game. We put the players in an array to make manipulation of multiple players easier throughout the code, as well as making it easier to expand the game to more players in the future.

Listing 12. main.cpp - SerialMessages

```
130 void printHelp()
131 {
132    Serial.println("-------------------------------------------------------");
133    Serial.println("Available commands:");
134    Serial.println("");
135    Serial.println("s - Start the game");
136    Serial.println("q - Stop the game and show the winner");
137    Serial.println("r - Reset the game/score");
138    Serial.println("h - Show this message again");
139    Serial.println("-------------------------------------------------------");
140 }
141
142 void printWinner(Player &winner, int playerIndex)
143 {
144    Serial.println("-------------------------------------------------------");
145    Serial.println("Player " + String(playerIndex + 1) + " wins!");
146    Serial.println("Score: " + String(winner.getPoints()));
147    Serial.println("-------------------------------------------------------");
148    printHelp();
149 }
```

Because we require Serial input to control the game, we should also provide some feedback to the Serial port. To make this cleaner and reusable in the code, we define functions for doing this.

Listing 13. main.cpp - Reset

```
153 void resetIo()
154 {
155    state.rgbLed.off();
156    noTone(appConfig.buzzerPin);
157    for (Player &player : state.players)
158    {
159       digitalWrite(player.playerConfig.ledPin, LOW);
160    }
161 }
162
163 void resetGame()
164 {
165    state.gameState = GameState::IDLE;
166    for (Player &player : state.players)
167    {
168       player.resetPoints();
169    }
170 }
```

Due to the sequential nature of this program, it is beneficial to create a function for turning off IO that should be disabled after a section is done executing. By giving each part of the program a clean slate, we can focus on only the IO that we care about. We also create a function for resetting the game and player points.

Take notice that the for-loop uses the & sybmol next to the player variable. This means that the player variable is a reference to the actual player object. C++ is by default ”pass by value”, meaning that if we were to remove the & symbol, a copy of the player at a given iteration would be assigned to the variable. This has a clear performance issue, but more importantly it creates a rather interesting bug. In this for-loop we alter the players internal point value. If this is a reference however, only the internal point value of the copy will change, while the original player object stored in the array will remain unchanged. This was likely the most noteworthy discovery during this lab, and a rather fun issue to debug.

Listing 14. main.cpp - getBestPlayerIndex

```
174 int getBestPlayerIndex()
175 {
176    int bestPlayerIndex = 0;
177    for (int i = 1; i < appConfig.numPlayers; i++)
178    {
179       Player &player = state.players[i];
180       Player &bestPlayer = state.players[bestPlayerIndex];
181       if (player.getPoints() > bestPlayer.getPoints())
182       {
183          bestPlayerIndex = i;
184       }
185    }
186    return bestPlayerIndex;
187 }
```

Because we have choosen to put the players in an array, we require a bit of logic to find the best player from the list. Because of readability and the fact that we will need this logic in several places of the program, we will create a function to do this.

Listing 15. main.cpp - updateGameState

```
191
192 /**
193  * @brief Checks for serial commands
194  * @return true if game should keep running
195  */
196 bool updateGameState()
197 {
198    while (Serial.available())
199    {
200       const char input = toLowerCase(Serial.read());
201       const auto command = static_cast<SerialCommand>(input);
202       switch (command)
203       {
204       case SerialCommand::START:
205          resetGame();
206          state.gameState = GameState::RUNNING;
207          Serial.println("Game started!");
208          break;
209       case SerialCommand::STOP:
210       {
211          Serial.println("Game stopped!");
212          int bestPlayerIndex = getBestPlayerIndex();
213          printWinner(state.players[bestPlayerIndex], bestPlayerIndex);
214          state.gameState = GameState::IDLE;
215       }
216       break;
217       case SerialCommand::RESET:
218          resetGame();
219          Serial.println("Everything has been reset!");
220          break;
221       case SerialCommand::HELP:
222          printHelp();
223          break;
224       default:
225          break;
226       }
227    }
228    if (state.gameState == GameState::RUNNING)
229    {
230       int bestPlayerIndex = getBestPlayerIndex();
231       Player &bestPlayer = state.players[bestPlayerIndex];
232       if (bestPlayer.getPoints() >= appConfig.winningPoints)
233       {
234          Serial.println("Player " + String(bestPlayerIndex + 1) + " has the
                highest score: " + String(bestPlayer.getPoints()));
235          printWinner(bestPlayer, bestPlayerIndex);
236          state.gameState = GameState::IDLE;
237       }
238    }
239    if (state.gameState == GameState::IDLE)
240    {
241       resetIo();
242    }
243    return state.gameState == GameState::RUNNING;
244 }
```

Because the GameState can be changed at a whim by the serial commands, we need some constantly running checks to make sure that the buzzer for example does not keep ringing after the game is stopped. We also want to constantly check for user input. This logic makes sense to put into its own function, to be run in all of our sequential while-loops.

Take notice that the code under the SerialCommand::STOP case is wrapped in curly braces. This is because we needed to create a variable scoped only to that case. If we didn't add the curly braces, the variable would be scoped to the entire function, and because that would mean a conditional decleration inside the same scope, it would be a syntax error.

Listing 16. main.cpp - fancySoundFunction

```
// Represents half of the period (a single on and off cycle) of a blink
int frequencyToHalfPeriodDelayTimeMs(const int frequency)
{
   return 1000 / frequency / 2;
}

// Pass the buzzer frequency through this to make a more interesting sound
// The function was arbitrarily choosen
int fancySoundFunction(const int frequency)
{
   long x = millis() / 20;
   return frequency + 500 * cos(x + sin(x));
}
```

To make the game more interesting, we want the winner sound to be something other than a simple linear ramp. To do this, we jump into GeoGebra and select an arbitrary mathematical expression to modulate the buzzer tone. The code vaguely represents this graph:

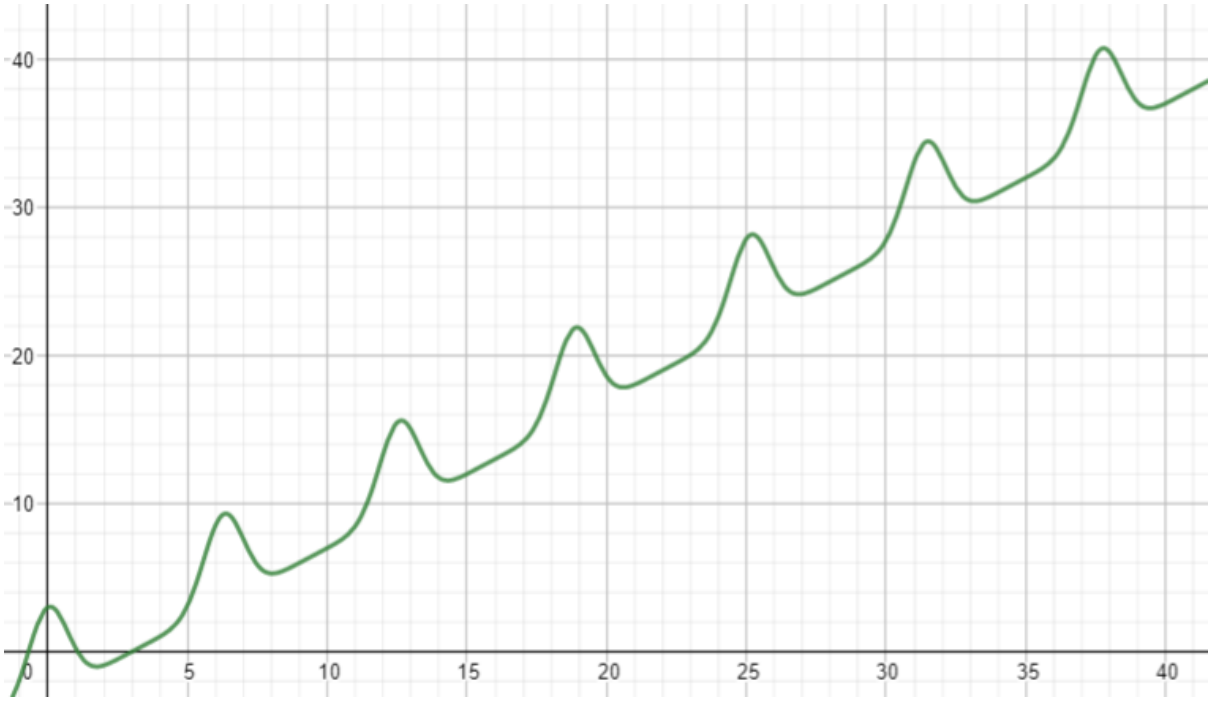

Fig. 4. Sound Function

Listing 17. main.cpp - printPoints

```
void printPoints()
{
   Serial.print("Score: ");
   for (int i = 0; i < appConfig.numPlayers; i++)
   {
      Player &player = state.players[i];
      if (i != 0)
      {
         Serial.print(", ");
      }
      Serial.print(player.getPoints());
   }
   Serial.println("");
}
```

Before we implement the logic for celebrating the winner and mocking the looser, we create a function to print the current score of all players to the serial port.

Listing 18. main.cpp - indicateWinner

```
281 void indicateWinner(Player &winner)
282 {
283    bool ledState = true;
284    ramp buzzerRamp;
285    const auto rampTargetValue = appConfig.winnerBuzzerPitch.maxValue -
           appConfig.winnerBuzzerPitch.minValue;
286
287    Timer blinkTimer;
288    const int blinkDelayTimeMs =
           frequencyToHalfPeriodDelayTimeMs(appConfig.fastBlinkFrequency);
289    buzzerRamp.go(rampTargetValue, appConfig.roundCompletionAnnouncementDuration);
290    for (; buzzerRamp.isRunning();)
291    {
292       if (!updateGameState())
293       {
294          return;
295       }
296       buzzerRamp.update();
297       const auto buzzerFrequency = buzzerRamp.getValue() +
              appConfig.winnerBuzzerPitch.minValue;
298       tone(appConfig.buzzerPin, fancySoundFunction(buzzerFrequency));
299
300       if (blinkTimer.loopWait(blinkDelayTimeMs))
301       {
302          ledState = !ledState;
303          digitalWrite(winner.playerConfig.ledPin, ledState);
304          state.rgbLed.green(!ledState);
305       }
306    }
307
308    resetIo();
309 }
```

Because the exercise requires the use of a for-loop, we have a loop-hole *punintended*. A for loop has 3 sections where the middle one is an evaluation. If we leave the two first empty, we get the same effect as a while-loop. Notice the use of the blinkTimer. It allows us to make the led blink, without using delay. This way we can not only implement fancy audio such as depicted above, but also have a very low responsetime when user input is detected from the serial port.

Listing 19. main.cpp - indicateLooser

```
313 void indicateLooser(Player &looser)
314 {
315    Timer blinkTimer;
316    bool ledState = true;
317    Timer announcementTimer;
318    const int blinkDelayTimeMs =
           frequencyToHalfPeriodDelayTimeMs(appConfig.fastBlinkFrequency);
319    announcementTimer.reset();
320    while
           (!announcementTimer.loopWait(appConfig.roundCompletionAnnouncementDuration))
321    {
322       if (!updateGameState())
323       {
324          return;
325       }
326       tone(appConfig.buzzerPin, appConfig.looserBuzzerPitch);
327
328       if (blinkTimer.loopWait(blinkDelayTimeMs))
329       {
330          ledState = !ledState;
331          digitalWrite(looser.playerConfig.ledPin, ledState);
332          state.rgbLed.red(!ledState);
333       }
334    }
335    resetIo();
336 }
```

Notice that because we are not waiting for a RAMP in this function, we need an additional timer. This is easy and clean to implement however, because we created our Timer class.

Listing 20. main.cpp - waitForButtonsToBeUnpressed

```
340
341 // Wait for both buttons to be released
342 // This is needed because the ezButton lib would
343 // remain pressed after the game finished and
344 // another game started.
345 // Adding a delay between games did not fix the issue.
346 void waitForButtonsToBeUnpressed()
347 {
348    while (true)
349    {
350       bool anyButtonIsPressed = false;
351       for (Player &player : state.players)
352       {
353          player.button.loop();
354          if (player.button.isPressed())
355          {
356             anyButtonIsPressed = true;
357             break;
358          }
359       }
360       if (!anyButtonIsPressed)
361       {
362          break;
363       }
364       delay(1);
365    }
366 }
```

Before we implement the game loop itself, we need one last helper function. This was the result of a rather cumbersome debugging session. The ezButton library seems to behave strangely when a round ends, and keeps reporting the button being pressed during the first update. Why? I don't know. Most likely it has something to do with the debouncing, and frankly i had better things to do. Without anything mentioned about it in the documentation, i decided to add a sanity check before starting a new round of the game. Now, the game will not continue until all the buttons have been released. The fix is rather simple, but such a solution should not be used in production code.

Listing 21. main.cpp - startGame

```
370 int randomInRange(const Range range)
371 {
372    return random(range.minValue, range.maxValue);
373 }
374
375 void startGame()
376 {
377    Timer roundTimer;
378    Timer trickRoundTimer;
379    Timer blinkTimer;
380    const int roundTimeMs = randomInRange(appConfig.roundTimeMs);
381    const int trickRoundTimeMs = appConfig.trickRoundDuration + roundTimeMs;
382    const bool trickRound = random(0, 100) <= (appConfig.trickRoundProbability *
           100);
383    waitForButtonsToBeUnpressed();
384    while (true)
385    {
386       if (!updateGameState())
387       {
388          return;
389       }
390
391       const bool roundFinished = roundTimer.isFinished(roundTimeMs);
392       const bool trickRoundFinished = trickRoundTimer.isFinished(trickRoundTimeMs);
393
394       if (roundFinished)
```

```
        {
            if (trickRound)
            {
                state.rgbLed.blue(true);
            }
            else
            {
                state.rgbLed.green(true);
            }
        }
        else
        {
            state.rgbLed.red(true);
        }

        for (Player &player : state.players)
        {
            player.button.loop();
            const bool buttonPressed = player.button.isPressed();
            if (buttonPressed)
            {
                if (roundFinished)
                {
                    // Triggerhappy
                    if (trickRound)
                    {

                        player.changePoints(-2);
                        printPoints();
                        indicateLooser(player);
                        return;
                    }
                    // Winner
                    else
                    {
                        int reactionTimeMs = roundTimer.getElapsedTime() - roundTimeMs;
                        float reactionTimeSec = reactionTimeMs / 1000.0;
                        // Add a small amount of time to avoid dividing by zero
                        int points = 1 / (reactionTimeSec + 0.000000001);
                        Serial.println("Reaction time: " + String(reactionTimeSec) + "
                            seconds");
                        player.changePoints(points);
                        printPoints();
                        indicateWinner(player);
                        return;
                    }
                }
                else
                // Tricked
                {
                    player.changePoints(-1);
                    printPoints();
                    indicateLooser(player);
                    return;
                }
            }
            if (trickRound && trickRoundFinished)
            {
                Serial.println("Time expired");
                return;
            }
        }
    }
}
```

Finally we get to the startGame function, which is our game loop. Here all of our choices pay off. From the use of three timers, the convenience of the updateGameState function, the RGB LED logic as helper functions, and the players inserted into an array all make this code very readable. Imagine if we had to manually disable

the 3 other colors on the RGB in each branch, or write a separate function that checked and updated each player as well as its button. With our current setup, this rather complex logic has become easy to implement. Also note that the updateGameState logic is identical to indicateWinner and indicateLooser. Consistency makes it easier for someone new to enter a codebase with which they are unfemiliar.This is great if you are working in a team, or need to get back to your own code after some time.

Listing 22. main.cpp - Setup

```
461 void setup()
462 {
463    Serial.begin(appConfig.baudRate);
464    randomSeed(analogRead(appConfig.randomSeedPin));
465    for (Player &player : state.players)
466    {
467       player.setup();
468    }
469    state.rgbLed.setup();
470    pinMode(appConfig.buzzerPin, OUTPUT);
471    printHelp();
472 }
```

Now that we have completed our game loop, we will create the void setup function Again we see the benefit of putting our players in an array. If we wish to add more players in the future, we are not at risk of forgetting to call setup for their IO, as the for-loop does not care about the number of players involved.

At the end of the setup function, we also call printHelp to show an initial set of instructions to the user.

Listing 23. main.cpp - loop

```
476 void loop()
477 {
478    if (updateGameState())
479    {
480       startGame();
481    }
482    else
483    {
484       delay(1);
485    }
486 }
```

And here we arrive at the last piece of code in the project. The main loop. Because everythin has been defined in its own function, it is very clean and easy to follow and expand upon.

III. Discussion

Because the solution i picked for this lab was very similar to lab2, the same discussion applies:

The current code works excellently, however it does have a weakness. While the nested nature of this code makes for very few instances of ”state” and globals, it does prevent us from running continous updates on anything while a piece of code is executing. In a larger project, i believe it would be beneficial to avoid while and for-loops, in favour of a more flat architecture with state machines. That way, the project remains scalable, and we can easily add continous checks without risking spagheti code and human errors due to forgetting to call an updater during a special loop.

In this project this weakness becomes a real problem, as we had to create helpers such as resetIo to keep our sanity. Perhaps the next lab will present a case where the current solutions’ drawbacks far outweigh its benefits, and a new solution must be choosen to address them.

IV. Large Images

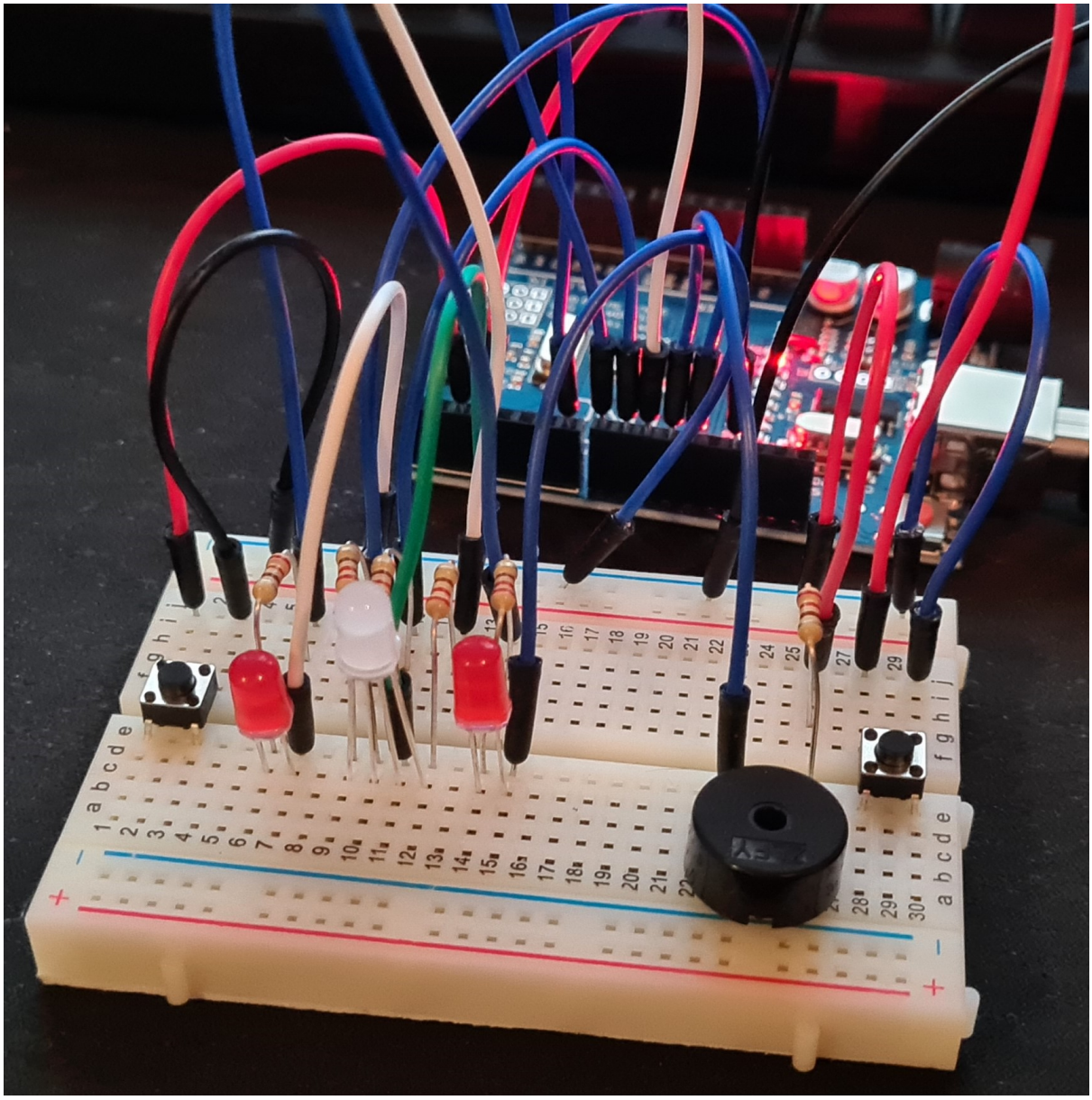

Fig. 5. Large Lab2 Circuit

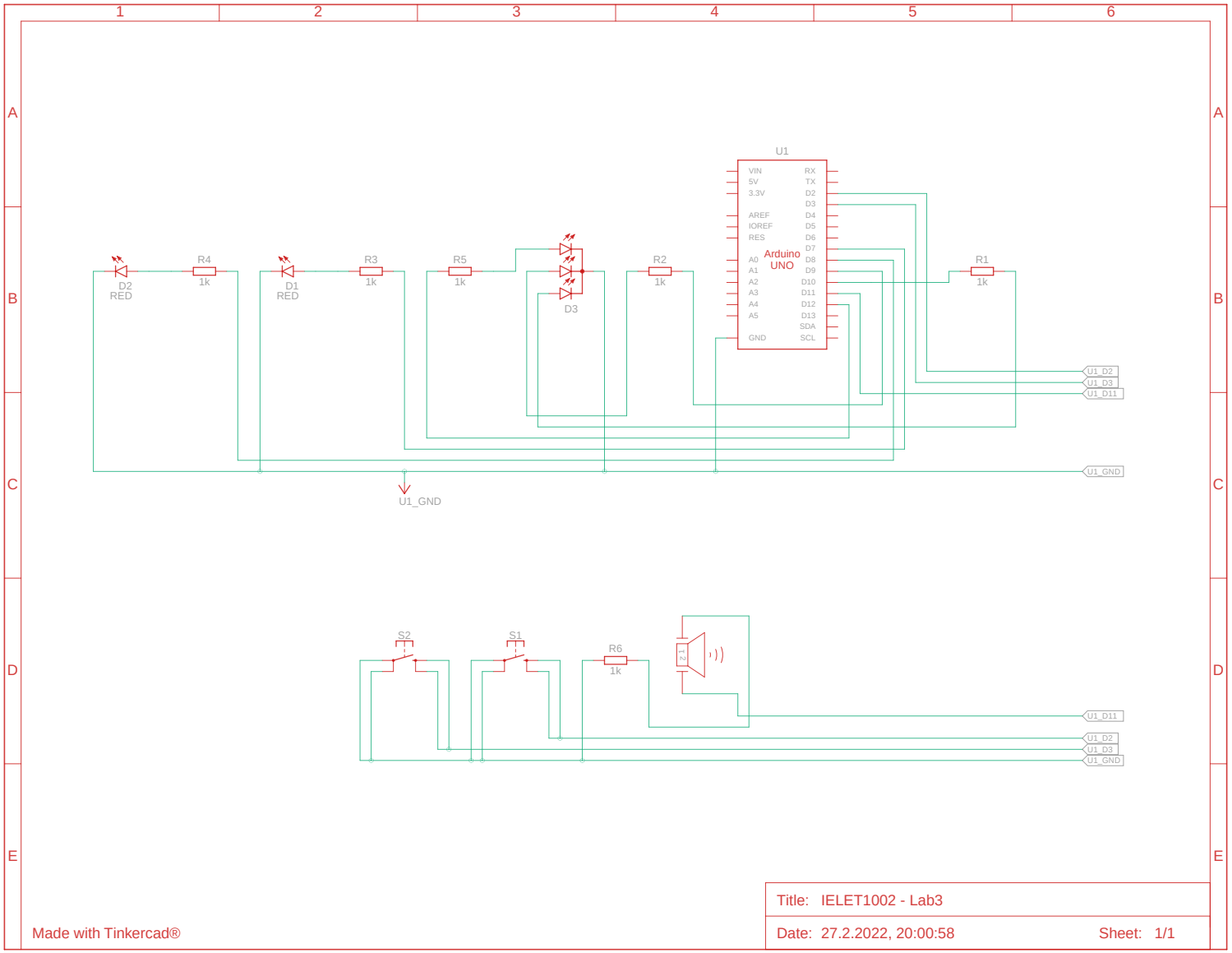

Fig. 6. Large Wiring Diagram

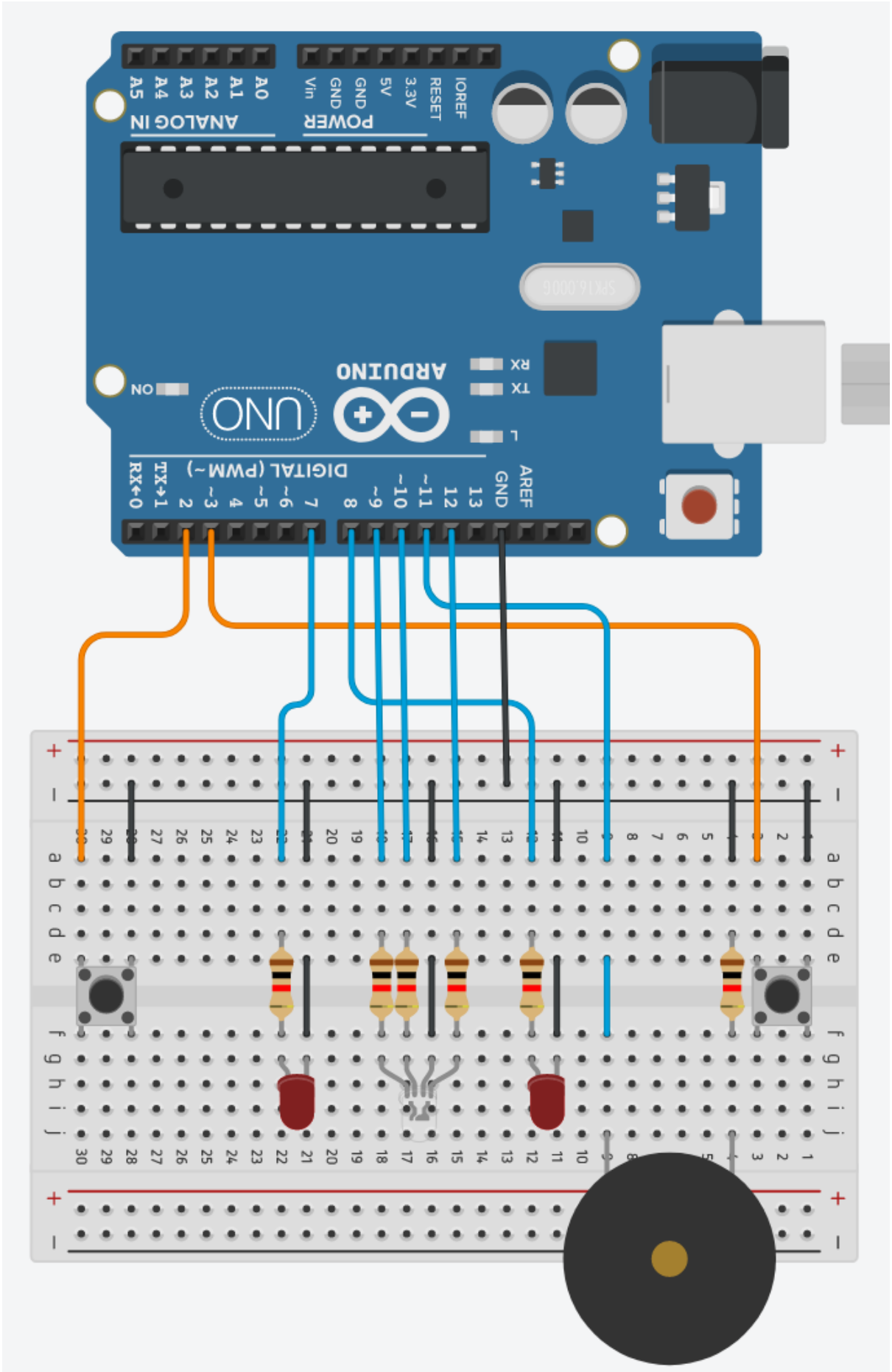

Fig. 7. Large Tinkercad