Prize Winner

Computer Programming, Apps & Robotics

Year 9-10

Paul Cyril
Lachlan Blake

The Heights School

Aim of Entry

The aim of this entry is to demonstrate how a laser and light sensors can be used to test the quality of water, and how computer processing can remove the effects of ambient light and plot the information visually.

Scientific Purpose

Turbidity is how cloudy or hazy a body of water is, caused by tiny particles which are invisible to the naked eye. Conventional turbidity sensors work by measuring how much light is scattered away from a focused beam of light. Therefore, they need to be in dark, well-enclosed containers to prevent interference from ambient light. However, this means that water must be flowing through the sensor to get constant readings.

It is possible to minimise the effects of ambient light by taking a measurement of ambient when the primary light source is off, and a measurement when the light source is on. By converting the measurements to the correct scale, it's possible to just subtract the effect of ambient light from the readings. This means the sensor does not need mechanical parts or complicated structures and improves the simplicity and versatility of the sensor. Transmitting data to an external device, like a PC or even a smartphone further reduces the cost of the device.

2020 Oliphant Science Awards
Student Work - DO NOT COPY

Potential Applications

Monitoring the sources of pollution in creeks, streams and rivers requires measurements to be taken frequently along the length of the river. This could be done through laborious monitoring by hand or by using large numbers of cheap monitors. The photo resistors, laser, wiring, mount and Arduino for this turbidity sensor can be brought for less than $60. This low-cost sensor could be mass produced, then placed along lengths of a river and continuously monitor water quality. Such data could even be used to help pinpoint dumping and leakage.

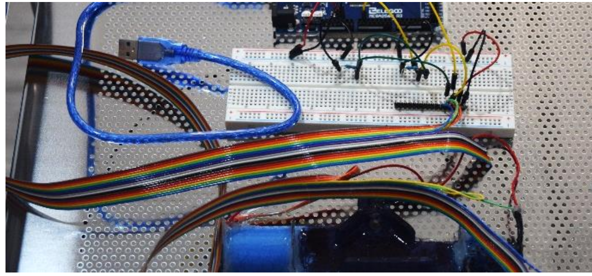
The application itself is designed to be flexible and versatile enough that other sensors can be easily added. Instead of preforming calculations on the Arduino, the Arduino sends raw data to a flexible desktop application on a computer. This allows diagnostics, debugging and calibrations to be done on the computer instead, making it easier to get information.

Equipment Needed

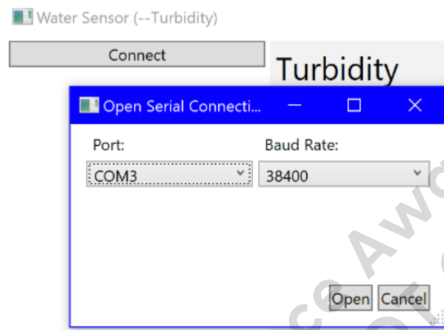
- Windows PC with USB port
- A copy of the application we wrote as an exe file, which can be downloaded from https://github.com/Teflae/Water-Sensor-Display/releases/tag/v1.0.
- An Arduino connected to a turbidity sensor (or applicable emulator).

Instructions

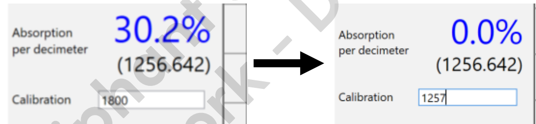
1. Place the sensor near the water body you want to test and the sensor and check that all the wiring is correctly in place.



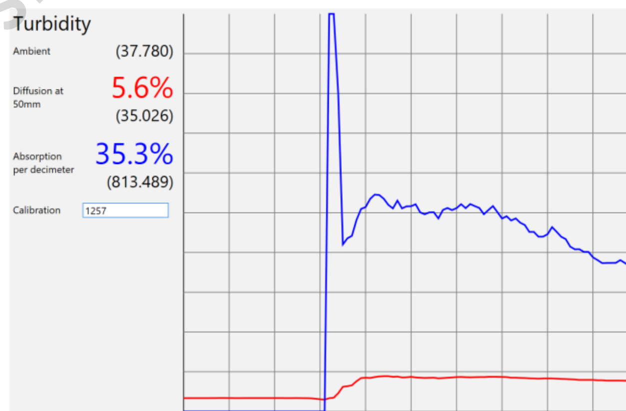
2. Run the application.
3. Connect the Arduino to the computer using the provided USB cable.
4. Press the ‘Connect’ button. If you have multiple COM ports check that the correct one is selected, then click ‘Open’:



5. Calibrate the sensor by setting the ‘Calibration’ text box (which defaults at 1800) to the number above it, so that the ‘Absorption per decimetre’ is at 0%:



6. Finally, place the sensor in the water. The black numbers are the raw values, approximately in Lux, of the photoresistors, while the percentages are based on the calibrated value. Red is diffusion, or how much of the laser’s light is detected as being scattered by the turbidity of the water, whilst blue is how much of the laser’s light was absorbed, or attenuated, by the water over 10cm:



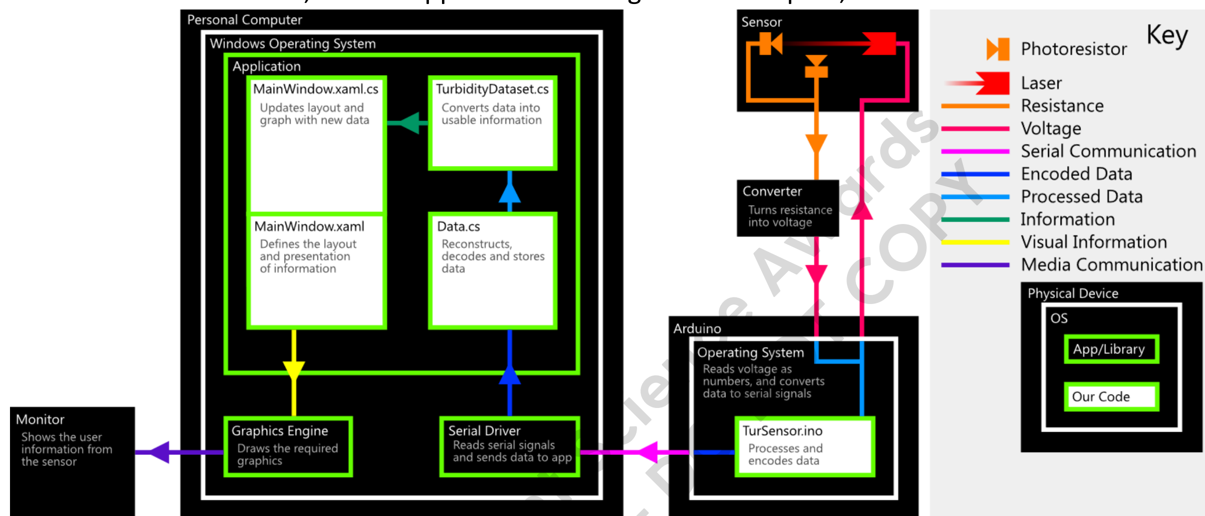
Note: We are currently working an emulator to simulate the sensor. Instructions will be at: https://github.com/Teflae/Water-Sensor-Display/tree/master/Emulator

2020 Oliphant Science Awards
Student Work – DO NOT COPY

Code

While it's possible to just look at a sample of water and say how turbid it is, eyesight alone cannot determine numerically how turbid it is, and can be influenced by external light factors. To get accurate measurements, the light levels can be measured by a light dependent resistor or photoresistor. However, a computer's monitor cannot directly find turbidity from the resistance of a submerged sensor. Instead, the resistance of the photoresistors is firstly converted into voltages to be measured by the Arduino, then sent through a USB (Serial) cable to the computer, which processes the data into information, and sends it to the monitor.

Whilst code for some stages already exists, we had to add hundreds of lines of our own code to bring it all together. This includes the code for the Arduino to get and send data, and a *Windows Presentation Foundation*, of WPF application running on a desktop PC, as shown below:



To allow us to edit the app at the same time, we've separated the code for the WPF application into multiple files to avoid editing the same file. In the following section the five most crucial files of code we've made are explained in detail.

Note: As with all applications bug fixes and new features will be added, possibly causing this hard copy to become outdated. Some unnecessary segments of code have been removed to reduce length. We upload the complete and latest version of the code to https://github.com/Teflae/Water-Sensor-Display.

TurSensor.ino

The Arduino runs this code. It is similar in structure to C++. First some global variables are defined. Integers *a* and *b* are the values returned by the sensors, *i* is used for timing purposes and increases by 1 for each measurement, *Lz* is how many measurements are taken before the laser is switched on or off, *Dz* is the delay, in milliseconds, between each measurement, Boolean *Lo* is whether the laser is on or off, and the string (text) *Data* is a buffer for storing data:

```
int a, b;
int i = 0;
int Lz = 10;
int Dz = 10;
bool Lo = false;
String Data = "";
```

In the *setup* function, a Serial connection is established and information about the sensor is sent to the computer (In future updates this allows the app to automatically determine what data it is getting), then configure the Arduino's output pins:

```
void setup()
{
```

```
  Serial.begin(38400);
  Serial.println("<'rate':'0.01','data':[{'x':'time'},{'x':'water'},{'x':'control','loop':'20'},{'x':'
turbidity','var':'absorb'},{'x':'turbidity','var':'diffuse'}]>\n");
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(8, OUTPUT);
  pinMode(7, INPUT_PULLUP);
}
```

The *loop* function is run repeatedly, taking measurements each time. It starts off by incrementing *i* and then checks if it's done enough measurements to toggle the laser.

```
void loop()
{
  i++;
  if (i >= Lz) {
    if (Lo) {
      Lo = false;
      digitalWrite(8, LOW);
```

Once it has completed taking *Lz* measurements with the laser off and on respectively, it sends the data inside its buffer as a batch. *Serial.println* appends a new line '\n' character to the end of the data, allowing the app to separate different batches:

```
      Serial.println(Data);
      Data = "";
    }
    else {
      Lo = true;
      digitalWrite(8, HIGH);
    }
    i = 0;
  }
```

Then it waits for *Dz* milliseconds and takes measurements using *analogRead* and adds the data it receives to the buffer. Each value is separated by the tab '\t' character:

```
  delay(Dz);
  a = analogRead(A0);
  b = analogRead(A1);
  Data += String(millis()) + '\t' + (digitalRead(7) ? "0" : "1") + '\t' + (Lo ? "1" : "0") + '\t' + St
ring(a) + '\t' + String(b) + '\t';
}
```

2020 Oliphant Science Awards
Student Work - DO NOT COPY

Data.cs

This C# file defines a class (tool and/or structure) for getting, correcting and storing data from the sensor.

```
using System;
using System.Collections.Generic;
using System.IO.Ports;
using System.Linq;

namespace Water_Sensor
{
    class Data
    {
```

Serial is the object used to access the computer's serial port, *Buffer* is the buffer of incoming data, *Dataset* contains a list of processed data, *DataType* stores the information about the sensor, *DataRecived* is an event that MainWindow.xaml.cs can 'subscribe' to, and *IsConnected* is used to check if the app is receiving serial data:

```
        private SerialPort Serial;
        private string Buffer = "";

        public List<TurbidityDataset> Dataset = new List<TurbidityDataset>();
        public string DataType = "Unknown";

        public event EventHandler DataRecived;
        public bool IsConnected
        {
            get
            {
                return Serial.IsOpen;
            }
```

```
        }
```

The *Data* function, when triggered by its parent, creates a new 'instance' of the class. In theory, the app can connect to multiple sensors just by having multiple *Data* classes:

```
        public Data()
        {
            Serial = new SerialPort();
        }
```

The *Connect* function updates *Serial* with user entered values, subscribes to its *DataReceived* event and then 'opens' the serial port:

```
        public void Connect(string Port, int BaudRate)
        {
            Serial.PortName = Port;
            Serial.BaudRate = BaudRate;
            Serial.DataReceived += CheckSerial;
            Serial.Open();
            Serial.WriteLine("<In[water|water-turbidity]>");
        }
```

Disconnect disconnects the serial port:

```
        public void Disconnect()
        {
            Serial.Close();
        }
```

The function *CheckSerial* is triggered every time a new segment of data comes in, complete or in pieces:

```
        private void CheckSerial(object sender, SerialDataReceivedEventArgs e)
        {
```

It gets all the data received so far and splits it into 'elements' by newline characters:

```
            string[] input = Serial.ReadExisting().Split('\n');
```

Takes the incomplete data stored inside *Buffer*:

```
            input[0] = Buffer + input[0];
```

And sets the *Buffer* to its last element, which is either incomplete or empty:

```
            Buffer = input[input.Length - 1];
```

Then each element, excluding the last one, is processed:

```
            for (int i = 0; i < input.Length - 1; i++)
            {
```

If it is information (begins with '<') *DataType* is set:

```
                if (input[i].First() == '<')
                {
                    DataType = input[i].Substring(1, input[i].Length - 2);
                }
```

Else it's data, which must be split into individual values by tab characters. Then it's checked to see if it is complete (has more than 100 values) and then a new *TurbidityDataset* class is created from the data:

```
                else
                {
                    string[] spl = input[i].Split('\t');
                    if (spl.Length >= 101) Dataset.Add(new TurbidityDataset(input[i], spl));
                }
            }
```

Finally, the *DataRecived* event is triggered allowing its subscriber, the app's UI, to update its layout:

```
            DataRecived?.Invoke(this, new EventArgs());
        }
    }
}
```

TurbidityDataset.cs

This C# class stores and converts the raw data into information.

```
using System;

namespace Water_Sensor
{
    class TurbidityDataset
    {
```

Some constants are calculated beforehand.

2020 Oliphant Science Awards
Student Work - DO NOT COPY

The Arduino can read voltages up to 5V with 10-bit resolution, therefore multiplying the Arduino's output by *AnalogueToDigitalScale* returns the voltage:

```
        private static double AnalogueToDigitalScale = 5.0 / 1023.0;
```

Through experimentation, the formula to convert resistance of LDR to a Lux approximation is 7,000,000x^-1.233. Those numbers are defined here:

```
        public static double multiple = 7000000.0;
        public static double power = -1.233;
```

LaserMax is the user-calibrated maximum output of the laser, used for calculations:

```
        public static double LaserMax;
```

RawString is the unmodified data from the Arduino, *RawData* is an integer array storing the data as numbers and *TimeStamp* is the time the data was recorded:

```
        public string RawString;
        public int[] RawData;
        public int TimeStamp;
```

To get the entire picture around turbidity; 'Absorb' values are from the sensor directly pointed by the laser, measuring the absorption or attenuation of light by the water (how black it is), whilst 'Diffuse' values are from the sensor hidden from the laser, measuring the scattering of light by the water (how cloudy it is).

To account for ambient light; 'Control' values are taken when the laser is switched off, whilst 'Laser' values taken when the laser is on.

This means there are a total of four variables:

```
        public double ControlAbsorb, ControlDiffuse, LaserAbsorb, LaserDiffuse;
```

With which other values can be calculated from:

```
        public double Ambient, Absorb, Diffuse, Absorption, AbsorptionPerMeter, Diffusion;
```

The function *TurbidityDataset* does most of the post-processing. It receives the raw data from Data.cs:

```
        public TurbidityDataset(string raw, string[] spl)
        {
            RawString = raw;
```

Before the data can be used, it must be converted into integers:

```
            string[] StrData = spl;
            int length = StrData.Length;
            RawData = new int[length];
            for (int i = 0; i < length - 1; i++)
            {
                RawData[i] = Convert.ToInt32(StrData[i]);
            }
```

Then the timestamp is recorded:

```
            TimeStamp = RawData[0];
```

To get rid of some noise in our data, the data needs to be averaged. To do that, variables representing the data (e.g. *Control-Absorb, Control-Diffuse, Laser-Absorb, Laser-Diffuse)* must be defined*:*

```
            int ca = 0;
            int cd = 0;
            int la = 0;
            int ld = 0;
```

Then a *for* loop can be used to sum the data. Since the laser seems fade out slightly, only the last 5 control values are taken to prevent interference:

```
            for (int i = 25; i < 50; i += 5)
            {
                ca += RawData[i + 3];
                cd += RawData[i + 4];
            }
```

Another *for* loop is used to sum the values taken when the laser is on. Only the last 8 readings are taken to prevent interference from the laser turning on:

```
            for (int i = 60; i < 100; i += 5)
            {
                la += RawData[i + 3];
                ld += RawData[i + 4];
            }
```

2020 Oliphant Science Awards
Student Work - DO NOT COPY

Then to complete the averaging, the variables are divided by their respective amounts and converted to a lux approximation using the function *TurValueToLux*:

```
            ca /= 5;
            cd /= 5;
            la /= 8;
            ld /= 8;
            ControlAbsorb = TurValueToLux(ca);
            ControlDiffuse = TurValueToLux(cd);
            LaserAbsorb = TurValueToLux(la);
            LaserDiffuse = TurValueToLux(ld);
```

Once converted, ambient light, absorb light and diffuse light can be calculated. The control values are subtracted from the actual readings, removing ambient light:

```
            Ambient = (ControlAbsorb + ControlDiffuse) / 2;
            Absorb = LaserAbsorb - ControlAbsorb;
            Diffuse = LaserDiffuse - ControlDiffuse;
```

Once the user calibrates the sensor, some easy-to-read percentages can be calculated:

```
            Absorption = 1 - Absorb / LaserMax;
            Diffusion = LaserDiffuse / LaserMax;
        }
```

The function *TurValueToLux* is defined down here so that it can be used repeatedly. It converts the Arduino's analogue readings to a Lux approximation, which makes subtracting ambient light possible.

```
        private static double TurValueToLux(int Analogue)
        {
            double RVoltage = Analogue * AnalogueToDigitalScale;
            double LDRVoltage = 5 - RVoltage;
            double LDRResistance = LDRVoltage / RVoltage * 10000;
            return multiple * Math.Pow(LDRResistance, power);
        }
    }
}
```

MainWindow.xaml.cs

This C# file initialises the app's user interface, listens for *DataRecived* events, updates the UI with information from individual *TurbidityDatasets* and then plots the data on a graph:

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;
using System.Windows.Shapes;

namespace Water_Sensor
{
```

In this class, the variables and storage methods which are used by multiple functions are defined:

```
    public partial class MainWindow : Window
    {
```

Data Sensor allows the program to connect with the data from the sensor. *Count* is the measurement of how many datasets there are from the sensor. *Diffuse_Data_Storage* and *Absorb_Data_Storage* are lists where the data from the sensor gets stored in for the graph to read and display. The integer variable *TurbidityGraphDataLength* is how many data points are displayed at one time on the graph:

```
        private Data Sensor;
        private int Count = 0;
        private List<double> Diffuse_Data_Storage = new List<double>();
        private List<double> Absorb_Data_Storage = new List<double>();
        private int TurbidityGraphDataLength = 100;
```

MainWindow is triggered when the application starts and runs the function to draw the graph axis and creates the *Data* class to access the sensor. It 'subscribes', or listens for *Sensor*'s *DataRecived* event:

```
        public MainWindow()
        {
            InitializeComponent();
            Draw_Graph_Axis();
            Sensor = new Data();
```

2020 Oliphant Science Awards
Student Work - DO NOT COPY

```
            Sensor.DataRecived += DataRecived;
        }
```

ConnectButton_Click is triggered when the '*Connect*' button is clicked. It first checks whether the sensor is connected to the app. If it is still connected, then it changes the connect button's text back to "*Connect*" and disconnects the sensor from the serial port.

```
        private void ConnectButton_Click(object sender, RoutedEventArgs e)
        {
            if (Sensor.IsConnected)
            {
                Sensor.Disconnect();
                ConnectButton.Content = "Connect";
                OutputTextBlock.Text = "";
            }
```

Else it opens the serial connection dialogue box (code on GitHub):

```
            else
            {
                SerialConnectionDialog Dialog = new SerialConnectionDialog();
                Dialog.Owner = this;
                bool IsAccepted = (bool)Dialog.ShowDialog();
```

If the dialogue box had been accepted (e.g. the user pressed 'Open'), it connects to the sensor to the specified port with the specified Baud Rate and then changes the button's text to "*Disconnect*". This code is nested inside a *try* loop, which instead of letting the application crash unexpectedly, it 'catches' any error and outputs an error message:

```
                if (IsAccepted)
                {
                    try
                    {
                        Sensor.Connect(Dialog.Port, Dialog.BaudRate);
                        ConnectButton.Content = "Disconnect";
                    }
                    catch (Exception ex)
                    {
                        OutputTextBlock.Text += ex.Message + '\n';
                    }
                }
            }
        }
```

2020 Oliphant Science Awards
Student Work - DO NOT COPY

DataRecived is triggered once the data has been processed:

```
        private void DataRecived(object sender, EventArgs e)
        {
            this.Dispatcher.Invoke(() =>
            {
                int l = Sensor.Dataset.Count;
```

For each new dataset, it adds *Absorption* and *Diffusion* values to its own lists to plot them on a graph:

```
                for (int i = Count; i < l; i++)
                {
                    Diffuse_Data_Storage.Add(Convert.ToDouble(Sensor.Dataset[i].Diffusion));
                    Absorb_Data_Storage.Add(Convert.ToDouble(Sensor.Dataset[i].Absorption));
```

Then checks whether the content in the list exceeded the length of the graph, and removes extras as necessary:

```
                    if (Diffuse_Data_Storage.Count > TurbidityGraphDataLength)
                    {
                        Diffuse_Data_Storage.RemoveAt(0);
                        Absorb_Data_Storage.RemoveAt(0);
                    };
```

Then converts the A*bsorption, Diffusion, Diffuse*, *Absorb* and A*mbient* data to string formats to be displayed in their corresponding *TextBlocks*:

```
                    PassiveTextBlock.Text = String.Format("({0:0.000})", Sensor.Dataset[i].Diffuse);
                    ActiveTextBlock.Text = String.Format("({0:0.000})", Sensor.Dataset[i].Absorb);

                    AmbientTextBlock.Text = String.Format("({0:0.000})", Sensor.Dataset[i].Ambient);

                    AbsorbtionTextBlock.Text = String.Format("{0:0.0%}",Sensor.Dataset[i].Absorption);
                    DiffusionTextBlock.Text = String.Format("{0:0.0%}", Sensor.Dataset[i].Diffusion);
                }
```

It then checks to see if new data actually came, before triggering *DrawTurbidityGraph*:

```
                if (Count < 1) DrawTurbidityGraph();
                Count = 1;
            });
        }
```

When DrawTurbidityGraph is triggered, it redraws the points on the turbidity graph. It first pre-calculates margins and chart area for performance; *Margin* is the spacing around the chart area from the edge of the canvas. *Xmin* and *ymin* are the minimum plotting distance that the points can be placed along the x and y axis. *Xmax* and *ymax* are the maximum plotting distance that the points can be places along the x and y axis. *Xarea* and *yarea* are the width and height of the chart area. *Step* is the spacing between each point along the x axis:

```
        private void DrawTurbidityGraph()
        {
            double margin = 10;
            double xmin = margin;
            double xmax = TurbidityGraph.ActualWidth - margin;
            double xarea = TurbidityGraph.ActualWidth - 2 * margin;
            double ymin = margin;
            double ymax = TurbidityGraph.ActualHeight - margin;
            double yarea = TurbidityGraph.ActualHeight - 2 * margin;

            double step = xarea / TurbidityGraphDataLength;
```

Then the polyline colours, variables and points are calculated and redrawn. *DiffuseColour* and *AbsorbColour* are the colours of the line plotted for their data points. *x* is the x value for the points. *Diffuse_Points* and *Absorb_Points* are the points which are plotted on the graph. *Diffuse_polyline* and *Absorb_polyline* are the elements drawn. *Draw_Graph_Axis* draws the grid and axis:

```
            Brush DiffuseColour = Brushes.Red;
            Brush AbsorbColour = Brushes.Blue;
            double x = xmin;
            Polyline Diffuse_polyline = new Polyline();
            Polyline Absorb_polyline = new Polyline();
            TurbidityGraph.Children.Clear();
            PointCollection Diffuse_points = new PointCollection();
            PointCollection Absorb_points = new PointCollection();
            Draw_Graph_Axis();

            int length = Diffuse_Data_Storage.Count;
```

A *for loop* is used to read the data stored in *Diffuse_Data_Storage* and *Absorb_Data_Storage lists*, verify the data and then convert them into *Points* to be drawn on the graph:

```
            for (int i = 0; i < length; i++)
            {
                double Diffuse_y = yarea - Diffuse_Data_Storage[i] * yarea;
                double Absorb_y = yarea - Absorb_Data_Storage[i] * yarea;
                if (Diffuse_y < ymin) Diffuse_y = ymin;
                if (Diffuse_y > ymax) Diffuse_y = ymax;
                Diffuse_points.Add(new Point(x, Diffuse_y));
                if (Absorb_y < ymin) Absorb_y = ymin;
                if (Absorb_y > ymax) Absorb_y = ymax;
                Absorb_points.Add(new Point(x, Absorb_y));
                x += step;
            }
```

Finally, the absorb and diffuse lines are drawn on the graph:

```
            Diffuse_polyline.StrokeThickness = 2;
            Diffuse_polyline.Stroke = DiffuseColour;
            Diffuse_polyline.Points = Diffuse_points;

            TurbidityGraph.Children.Add(Diffuse_polyline);
            Absorb_polyline.StrokeThickness = 2;
            Absorb_polyline.Stroke = AbsorbColour;
            Absorb_polyline.Points = Absorb_points;

            TurbidityGraph.Children.Add(Absorb_polyline);
        }
```

When triggered, *Draw_Graph_Axis* draws the grid and axis for the graph. Again, it first pre-calculates margins and chart area for performance:

```
        private void Draw_Graph_Axis()
```

2020 Oliphant Science Awards
Student Work - DO NOT COPY

```
        {
            double margin = 10;
            double xmin = margin;
            double xmax = TurbidityGraph.ActualWidth - margin;
            double xarea = TurbidityGraph.ActualWidth - 2 * margin;
            double ymin = margin;
            double ymax = TurbidityGraph.ActualHeight - margin;
            double yarea = TurbidityGraph.ActualHeight - 2 * margin;
            double xstep = xarea / 10;
            double ystep = yarea / 10;
```

Then it draws the X and Y axis for the graph:

```
            //Make the axis
            GeometryGroup axis_geom = new GeometryGroup();
            axis_geom.Children.Add(new LineGeometry(
                new Point(xmin, ymax), new Point(xmax, ymax)));
            axis_geom.Children.Add(new LineGeometry(
                new Point(xmin, ymax), new Point(xmin, ymin)));

            Path axis_path = new Path();
            axis_path.StrokeThickness = 1;
            axis_path.Stroke = Brushes.Black;
            axis_path.Data = axis_geom;

            TurbidityGraph.Children.Add(axis_path);
```

Then it draws the horizontal grid lines using another geometry group:

```
            // Make the horizontal grid.
            GeometryGroup xgrid_geom = new GeometryGroup();
            for (double x = xmin + xstep; x <= xarea; x += xstep)
            {
                xgrid_geom.Children.Add(new LineGeometry(
                    new Point(x, ymin),
                    new Point(x, ymax)));
            }

            Path xgrid_path = new Path();
            xgrid_path.StrokeThickness = 1;
            xgrid_path.Stroke = Brushes.Gray;
            xgrid_path.Data = xgrid_geom;

            TurbidityGraph.Children.Add(xgrid_path);
```

And finally, it draws the vertical grid lines:

```
            // Make the vertical grid.
            GeometryGroup ygrid_geom = new GeometryGroup();
            for (double y = ymin + ystep; y <= yarea; y += ystep)
            {
                ygrid_geom.Children.Add(new LineGeometry(
                    new Point(xmin, y),
                    new Point(xmax, y)));
            }

            Path ygrid_path = new Path();
            ygrid_path.StrokeThickness = 1;
            ygrid_path.Stroke = Brushes.Gray;
            ygrid_path.Data = ygrid_geom;

            TurbidityGraph.Children.Add(ygrid_path);
        }
```

The function *LaserMaxTuningTextBox_TextChanged*, is triggered by changing the calibration *TextBox*. Using a *try* loop and an *if* statement for input validation, it modifies *TurbidityDataset*'s *LaserMax* property, calibrating the app.

```
        private void LaserMaxTuningTextBox_TextChanged(object sender, TextChangedEventArgs e)
        {
            try
            {
                double lmt = Convert.ToDouble(LaserMaxTuningTextBox.Text);
                if (lmt > 100 && lmt < 100000)
```

2020 Oliphant Science Awards
Student Work - DO NOT COPY

```
                {
                    TurbidityDataset.LaserMax = lmt;
                }
            }
            catch (Exception) { }
        }
```

TurbidityGraph_SizeChanged is triggered when the size of the graph changes, allowing it to be redrawn to fit:

```
        private void TurbidityGraph_SizeChanged(object sender, SizeChangedEventArgs e)
        {
            DrawTurbidityGraph();
        }
    }
}
```

MainWindow.xaml

This *xaml* file is the design aspect for the application.

```
<Window x:Class="Water_Sensor.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:Water_Sensor"
        mc:Ignorable="d"
```

First, it sets the title as well as default sizes of the application window where *Height* is the height of the window and *Width* is the width of the window. The min variations determine the minimum size for the window. All the code for the application is located inside the applications *Grid*:

```
        Title="Water Sensor (--Turbidity)" Height="650" Width="1200" WindowState="Normal"
MinWidth="600" MinHeight="400">
    <Grid>
```

Then it defines the rows and columns upon which elements will be aligned.

```
        <Grid.RowDefinitions>
            <RowDefinition Height="150"/>
            <RowDefinition Height="1*"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="200"/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
```

Next, it creates button to connect to sensor which triggers the void "*ConnectButton_Click*" when clicked which triggers in *MainWindow.xmal.cs*. For each element in the design, its location, type, style and size are specified so that the program knows where to load everything and what they look like when the application loads.

```
<Button x:Name="ConnectButton" HorizontalAlignment="Stretch" VerticalAlignment="Top" Margin="4,4,0,0"
Click="ConnectButton_Click">Connect</Button>
        <Button x:Name="TestButton" HorizontalAlignment="Left" VerticalAlignment="Bottom"
Margin="4,4,0,4" Click="TestButton_Click" Visibility="Hidden">$Test_Get</Button>
```

And then defines the text block which displays the status of the sensor.

```
        <TextBlock x:Name="StatusTextBlock" TextWrapping="Wrap" Margin="4,28,4,4" />
```

To support multiple sensors in the future, each sensor will have its own *grid* inside a *stackpanel*:

```
        <StackPanel Grid.Column="1" Grid.RowSpan="2">
```

TurbidityGrid is the grey box that contains all the information from the sensor and graph. The colour, size and layout of the grid is defined first:

```
            <Grid x:Name="TurbidityGrid" Height="500" Margin="4,4,4,4">
                <Grid.Background>
                    <SolidColorBrush Color="Black" Opacity="0.05"/>
                </Grid.Background>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="80"/>
                    <ColumnDefinition Width="120"/>
                    <ColumnDefinition Width="1*"/>
                </Grid.ColumnDefinitions>
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto"/>
                    <RowDefinition Height="32"/>
                    <RowDefinition Height="48"/>
```

2020 Oliphant Science Awards
Student Work - DO NOT COPY

```
                <RowDefinition Height="32"/>
                <RowDefinition Height="48"/>
                <RowDefinition Height="32"/>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="1*"/>
            </Grid.RowDefinitions>
```

Then it defines the *TextBlock* labelling the sensor the grid is displaying (a.k.a. turbidity). In addition, defines all the text blocks which display the data. E.g. *ambient label, ambient data, passive label, active label, diffusion data, absorption data* and *passive data:*

```
            <TextBlock x:Name="TurbidityLabelTextBlock" Grid.Row="0" Grid.Column="0" 
Grid.ColumnSpan="2" HorizontalAlignment="Stretch" VerticalAlignment="Center" FontSize="24" 
Padding="4,4,2,2">Turbidity</TextBlock>
            <TextBlock x:Name="AmbientLabelTextBlock" Grid.Row="1" Grid.Column="0" 
Grid.ColumnSpan="1" Margin="4,2,2,2" VerticalAlignment="Center">Ambient</TextBlock>
            <TextBlock x:Name="AmbientTextBlock" FontSize="20" Grid.Row="1" Grid.Column="1" 
Grid.ColumnSpan="1" Margin="2,2,2,2" HorizontalAlignment="Right"/>
```

It defines labels for the diffusion and absorption data:

```
            <TextBlock x:Name="PassiveLabelTextBlock" FontStretch="Normal" TextWrapping="Wrap" 
Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="1" Grid.RowSpan="2" Margin="4,2,2,2" 
VerticalAlignment="Center" >Diffusion at 50mm</TextBlock>
            <TextBlock x:Name="ActiveLabelTextBlock" TextWrapping="Wrap" Grid.Row="4" 
Margin="4,2,2,4" Grid.RowSpan="2" VerticalAlignment="Center">Absorption per decimeter</TextBlock>
```

Then it defines *TextBlock*s which displays diffusion, absorption and passive data:

```
            <TextBlock x:Name="DiffusionTextBlock" FontSize="36" Grid.Row="2" Grid.RowSpan="1" 
Grid.Column="1" Grid.ColumnSpan="1" Margin="2,2,2,2" Foreground="Red" 
HorizontalAlignment="Right"></TextBlock>
            <TextBlock x:Name="AbsorbtionTextBlock" FontSize="36" Grid.Row="4" Grid.Column="1" 
Margin="2,2,2,4" TextAlignment="Left" LineHeight="72" Foreground="Blue" 
HorizontalAlignment="Right"></TextBlock>
            <TextBlock x:Name="PassiveTextBlock" FontSize="20" Grid.Row="3" Grid.RowSpan="1" 
Grid.Column="1" Grid.ColumnSpan="1" Margin="2,0,2,4" HorizontalAlignment="Right"></TextBlock>
            <TextBlock x:Name="ActiveTextBlock" FontSize="20" Grid.Row="5" Grid.Column="1" 
Margin="2,0,2,4" TextAlignment="Left" LineHeight="72" HorizontalAlignment="Right"></TextBlock>
```

It also defines *TextBox* to calibrate the sensor as well as an associated label. When the *TextBox* is edited it Triggers *LaserMaxTuningTextBox_TextChanged* function.

```
            <TextBlock x:Name="LaserMaxTuningLabelTextBlock" TextWrapping="Wrap" Grid.Row="6" 
Margin="4,2,2,4" Grid.RowSpan="1" VerticalAlignment="Center">Calibration</TextBlock>
            <TextBox x:Name="LaserMaxTuningTextBox" Grid.Row="6" Margin="8,8,8,8" Grid.RowSpan="1" 
Grid.Column="1" TextChanged="LaserMaxTuningTextBox_TextChanged">1800</TextBox>
```

And finally, it defines a *canvas* element, on which a graph can be drawn. When the size of this element is changed, it Triggers the *TurbidityGraph_SizeChanged* function:

```
            <Canvas x:Name="TurbidityGraph" Grid.RowSpan="8" Grid.ColumnSpan="1" Grid.Column="2" 
SizeChanged="TurbidityGraph_SizeChanged"/>
        </Grid>
    </StackPanel>
  </Grid>
</Window>
```

2020 Oliphant Science Awards Student Work - DO NOT COPY

Photos, Screenshots and Videos

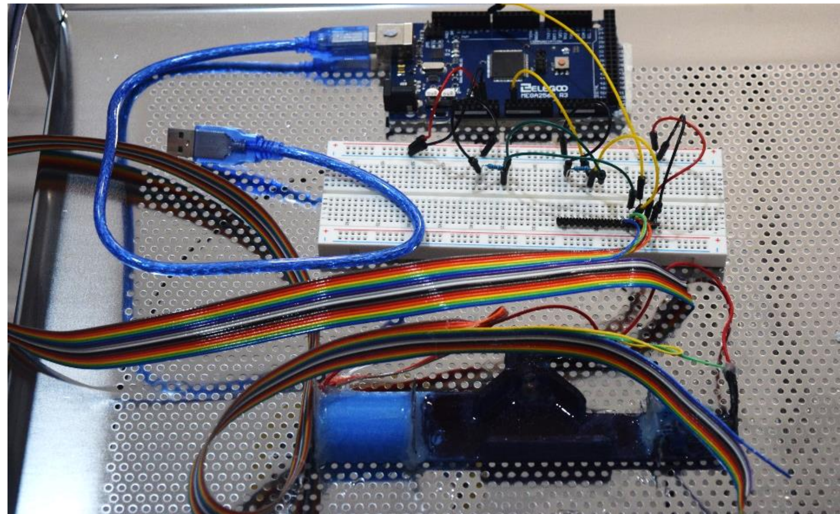


Figure 1: Showing the blue and black sensor constructed, the voltage dividers on a breadboard, and an Arduino.

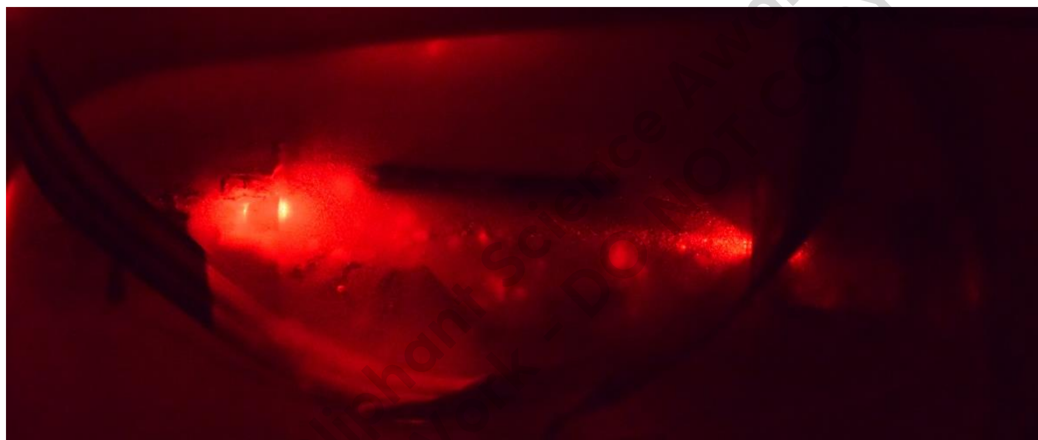


Figure 2: Showing the sensor in operation. The laser is on the right.

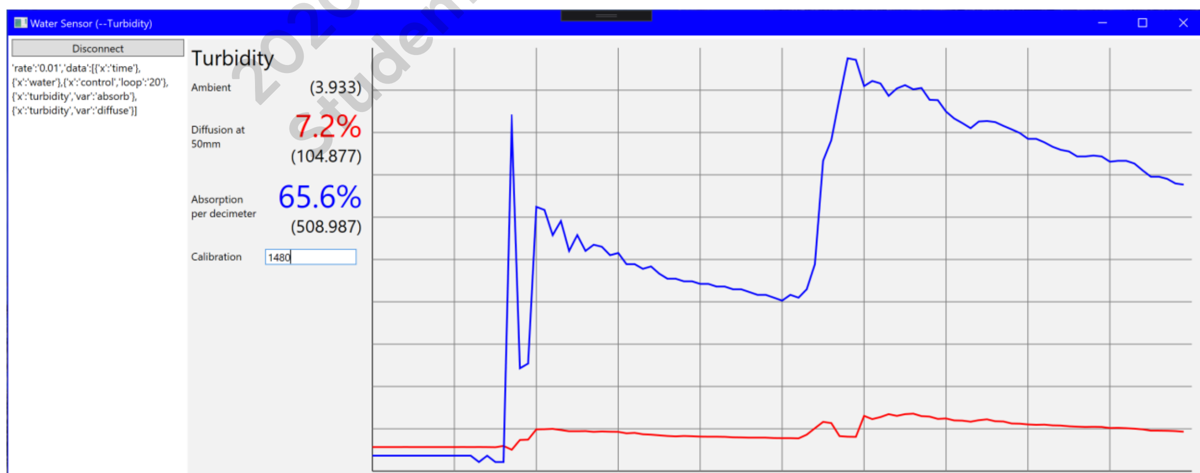


Figure 3: A screenshot of the app running whilst plotting data from the sensor. Whilst the water was not technically turbid, to show the operation of the app the sensor was placed in a container of water flour. The container was shaken twice, first gently and then roughly. This unsettled the flour, causing two corresponding peaks on the graph.

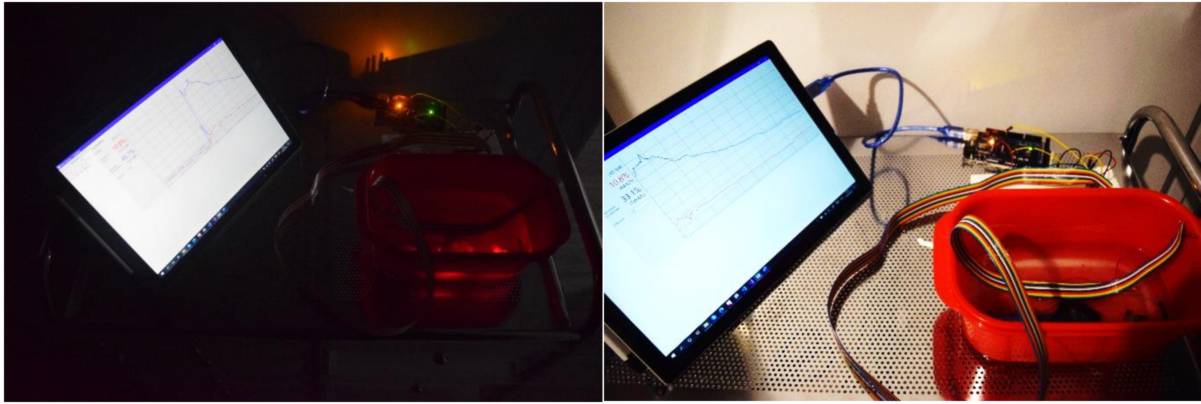


Figure 4: The entire system. Left: the sensor measuring turbidity and the application plotting it on a graph. Right: the system running after the light was turned on immediately after taking the photo on the left. Notice that there is no sudden change in the graph as the light was turned on.

Bibliography

Mangal, K., NA. *C# List Class.* [Online]
Available at: https://www.geeksforgeeks.org/c-sharp-list-class/
[Accessed 18 July 2020].

Microsoft, 2020. *How to publish events that conform to .NET Guidelines (C# Programming Guide).* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/how-to-publish-events-that-conform-to-net-framework-guidelines
[Accessed 18 July 2020].

Microsoft, 2020. *SerialPort Class.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/api/system.io.ports.serialport?view=dotnet-plat-ext-3.1
[Accessed 12 July 2020].

Microsoft, 2020. *Windows Presentation Foundation.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/framework/wpf/
[Accessed 6 July 2020].

mybotic, 2020. *Laser Diode Module Tutorial.* [Online]
Available at: https://www.instructables.com/id/Laser-Diode-Module-Tutorial/
[Accessed 6 July 2020].

net-Informations, NA. *Rounding Number to 2 Decimal Places.* [Online]
Available at: http://net-informations.com/q/faq/round.html
[Accessed 19 jULY 2020].

RodStevens, 2014. *Graphing Difficulties.* [Online]
Available at: http://csharphelper.com/blog/2014/09/draw-graph-wpf-c/
[Accessed 15 July 2020].

Water Science School, 2020. *Turbidity and Water.* [Online]
Available at: https://www.usgs.gov/special-topic/water-science-school/science/turbidity-and-water?qt-science_center_objects=0#qt-science_center_objects
[Accessed 21 July 2020].

2020 Oliphant Science Awards
Student Work - DO NOT COPY

Wikipedia, 2020. *Nephelometer.* [Online]
Available at: https://en.wikipedia.org/wiki/Nephelometer
[Accessed 21 July 2020].

Williams, D., 2015. *Design a Luxmeter Using a Light Dependent Resistor.* [Online]
Available at: https://www.allaboutcircuits.com/projects/design-a-luxmeter-using-a-light-dependent-resistor/
[Accessed 17 July 2020].

Acknowledgements

Thank you to Tim Trainor and Tom Cridland from The Heights School for your assistance, support and encouragement towards us throughout the project.

2020 Oliphant Science Awards
Student Work - DO NOT COPY