Software Notebook

All the nitty-gritty software details that were in development.

Written By:
Jaiden Grimminck (Software Lead)

Credits:
Jaiden Grimminck (8840-utils, 8840-app)

Table of Contents

Background

Last year (2022), we ended the season in software with an unsatisfactory ending. Our code was inefficient, and had been rewritten in quite literally an hour. We set off hoping to streamline our process, and to create complex code in order to learn more and to prove that we were capable of developing complex features for our robot, and to put our developing skills to the test.

We ended up developing three main tools. First off, better-downloader, streamlining our process of transferring code. We ended up opting for using GitHub for transferring code, but it was a good practice in code transfer. Last year, we developed easy-downloader (in an hour… see the pattern from last year?) which would compress our robot repository into a zip file and download it on our Windows laptop. This was very inefficient, which led to better-downloader being developed. It allowed real-time changes to code, detecting changed files and transferring them through a REST API. We won't cover much about this though since this wasn't used often in our development process, rather we'll cover the two other tools used in and with our robot.

The second tool is 8840-utils. We set out to create tools that we deemed were missing from WPILib and to create a way to streamline our development process into a quick and easy process, allowing code to be produced efficiently and with speed. 8840-utils introduced a custom robot framework, as well as logging, an "IO" system (we're not too sure what to call it… think of it more as a simulation system), a custom autonomous system, custom web modules, and so much more. This led us to efficiently develop our robot code in the 2023 season, and saved us lots of time.

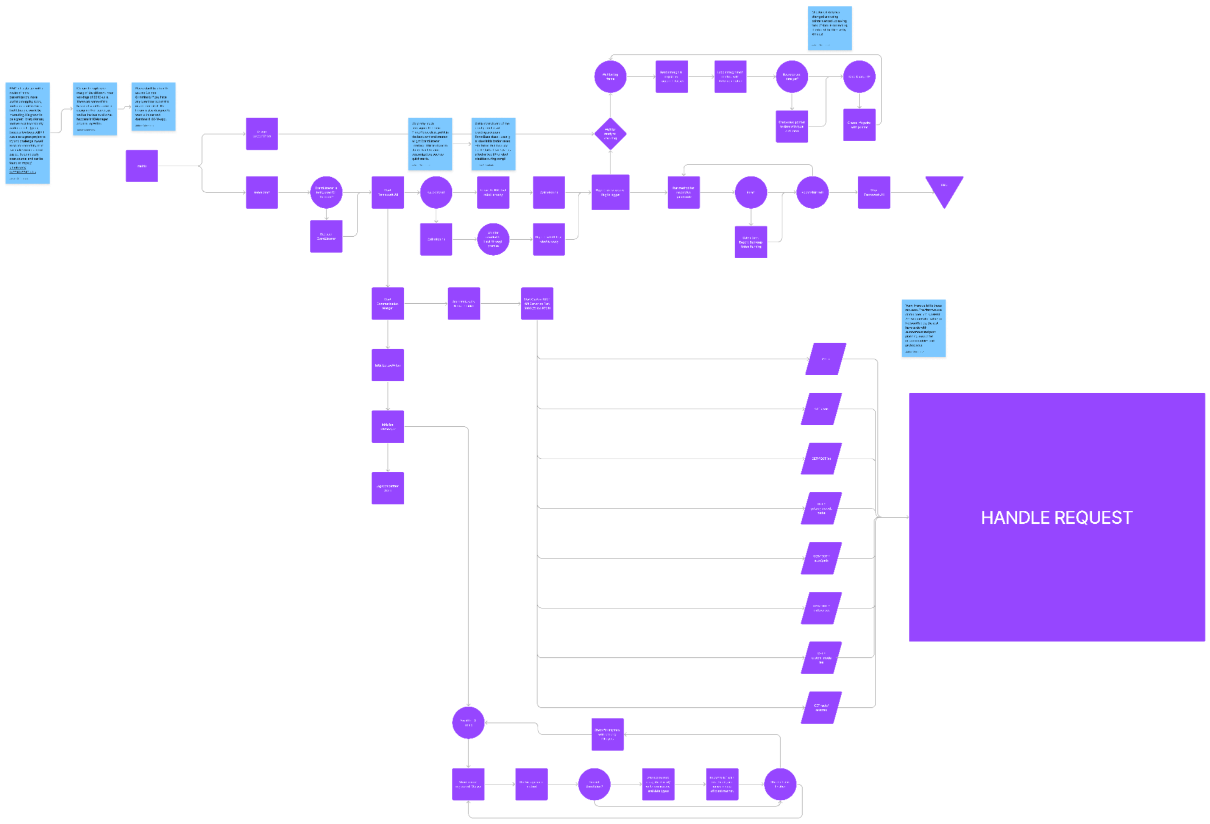
The third and final tool is 8840-app. This web dashboard built with ReactJS allows us to connect 8840-utils through multiple protocols, and introduces tools such as trajectory creation, parsing for custom modules, log viewers, neural networks, 3D field plotting, raycasting, and more.

Note: All of our code is open-source, and follows rule R303. There is an exception for our robot code for 2023 (which is private for now), but we created it after kickoff. We made sure to release all of our code numerous times before kickoff, and have been constantly updating it since.

8840-utils

Custom Framework

One thing that was hard in developing code was the game modes, and all of their methods - we rather decided to use an interface to implement the methods, which required the methods to be written. This meant that we had to create our own version of the TimedRobot class to do so. Through this, we gained multiple advantages, such as choosing when the robot finishes enabling the robot code, and allowing us to learn about the HAL. Capturing the process follows this flowchart:



…Which is a bit big. Let’s break it down a bit. When the program starts, the user will assign an “`EventListener`” which is our class with all of our different game methods.

```
Robot.assignListener(new SomeEventListener());
```

Through this, the Robot class now knows what's assigned, and stores this for later. The Robot class now needs to be fed through WPILib. This is done pretty simply, just through starting the robot with it. (Note: Robot doesn't have to be used to use 8840-utils! It's only used if `EventListeners` want to be used, but all important loops such as Logging or IO are called in the `FrameworkUtil`, allowing other robot types to be used if wanted to.)

```
RobotBase.startRobot(Robot::new);
```

Since `Robot` extends `RobotBase`, we can do whatever we want with it, such as above. This then leads through a process of figuring out what to start. The program allows for "quick starting," which enables the "Code Ready" flag on the robot before the robot initialization method is called. Usually, this will be waited on, through a promise. The robot initialization can create a lambda that has to be cleared before continuing on in the robot code. For example, we use it in our robot code to wait until our swerve drive modules are ready.

```
Robot.getRealInstance()
          .waitForFullfillConditions(
            5000, //Time waiting (ms)
             () -> m_robotContainer.getDriveSubsystem().ready() //Condition
          )
          .onFinishFullfillment(() -> {
              //Reset odometry when the robot is ready!
              m_robotContainer.getDriveSubsystem().resetOdometry();
          });
```

Once finished, the robot will trigger the flag for the robot code. That's the basics of it, ignoring the initialization for the logger and the IO system.

Logging

We set out to create an easy way to log information, and to make sure it's data-efficient. A second goal that we made was that it would be human readable, or at least easily parsed by programs. We ended up with the Loggable class and the `AutoLog` annotation. Let's first go over how a class would be logged. We'll use the `SwerveGroup` class that's also in our library (that'll be covered later on in more detail) for an example.

```
public class SwerveGroup implements Loggable {
```

We first can implement the Loggable notation for the class, making sure that the class can be declared as a `Loggable` in the backend.

```
  public SwerveGroup(...) {
      //...
       //Add to logger
       Logger.addClassToBeAutoLogged(this);
   }
```

Then we just add the class to be logged. That's it for making sure that the class will be seen by the backend. We'll go over how this works in a second, but let's now talk about how data is saved, using the module speeds as an example. If we had a method called "`logSpeeds`" that would return an array of 4 doubles, each corresponding with a module's speed, all we have to do is add on the `AutoLog` annotation, like so:

```
@AutoLog(logtype = LogType.DOUBLE_ARRAY, name = "Swerve Drive Module Speeds")
  public double[] logSpeeds() {
      //Declare speed array
      double[] speeds = new double[4];

       //Loop through each module and add it to the speed array.
      loop((module, i) -> {
          speeds[i] = module.getState().speedMetersPerSecond;
      });

      //Return speeds.
      return speeds;
  }
```

Zooming in on the annotation, we see that it's pretty simple.

```
@AutoLog(logtype = LogType.DOUBLE_ARRAY, name = "Swerve Drive Module Speeds")
```

The `logtype` has to be equal to the return type of the method, and the name is how it'll be saved in the log. (Note: We're using an enum instead of detecting the return type - this will be changed in the future, but for now we're limiting down the return types, therefore using an enum.)

In the backend, the process is simple and intuitive. To stay consistent, logs are based on "cycles." Each "cycle" is about 1/32 of a second, a time that we deemed to be small enough to capture the changes, but big enough that it wouldn't cause memory issues with logs and processing. (The 1/32 of a second shows up a lot in our code! We use it often in timer tasks, timed game methods, IO, and more.) Simply put, each time a cycle is called, the program loops through each registered `Loggable`, then through each method that it contains. If the method has the annotation, it'll encode the return value to a string. Depending on if the data has been saved before or not, a "pointer" to the name will be created - pretty much a number that'll refer to the name. This saves data because the program doesn't have to save the name each log cycle, rather it only has to save one integer. The program also checks whether or not the data has changed from the last cycle - if not, the program will not send anything to save data. From our own tests, this has suggested a 4x reduction in file size. But where is this data sent? This is where the `LogWriter` comes into play.

```
ALC2
aSwerve Drive Module SpeedsD/0
d0/[0.0,0.0,0.0,0.0]
ALC3
d0/[1.1,1.1,1.1,1.1]
```

An example of how the logs are stored.

The `LogWriter` class allows for customization of how the data is processed. We've made a few preset ones, one that updates NT (`NTWriter`), one that writes to a file (`FileWriter`), and an empty one (`EmptyLog`). The user can make their own `LogWriter`, and change how the data is handled. We hope to make an encoded one in the future to allow for even more data optimization, but that's an off-season project. Each `LogWriter` has two key methods: "`saveLine`," and "`saveInfo`." The "`saveLine`" is for any messages. Instead of using the normal println that Java offers, users of the library can use "`Logger.Log`" in order to make sure that this data goes through this method - it also provides timestamps and more to make sure the user knows when each message is sent.

The second method, "`saveInfo,`" is for info such as logging. In the `FileWriter` class, it's not used very differently from `saveLine`, it just saves the data. In the `NTWriter` class, it rather creates a separate NT entry for the info and messages.

Assigning the `LogWriter` is quick and easy, and is done in the “`main`” method of the user’s code with:

```
Logger.setWriter(new FileWriter("default"));
```

IO System

(Or whatever it is.)

The IO system is a way to go back and forth between real objects, and simulated objects. It’s a way to see what components are doing in both simulated circumstances, as well as real circumstances. Each “`IOLayer`” will have at least one read method and name, as well as potential write methods. The `IOCANCoder` class in our library is a good example of this. The `IOCANCoder` has the IO permission `READ_WRITE`, which means that there are methods that can be written to, and methods that can be read from. This is declared with the class annotation of:

```
@IOAccess(IOPermission.READ_WRITE)
public class IOCANCoder extends IOLayer {
 //...
}
```

In the constructor of the class, `super` is called in order to make sure that the method is initialized.

To declare the name of the class, a method is used. The advantages of this are allowing for name changes, such as if different components are being used, or you want to differentiate where each component is being used. This is done through `getBaseName`, for example, in `IOCANCoder`:

```
public String getBaseName() {
      return "CANCoder";
}
```

This will lead to all `IOCANCoder` instances being put under the category of “`IOCANCoder`” in NT. Before we get to what it actually looks like in NT, we can show an example of a read method in the `IOCANCoder` class. In the `IOCANCoder`, it has both a reference to a `CANCoder`, as well as a cache value. This cache value is used in simulation, and the real value is used on the robot. We can take a look at getting the absolute position of the CANCoder.

```java
/**
 * Returns in degrees the position of the encoder
 * @return The position of the encoder in degrees
 */
@IOMethod(
  name = "absolute position",
  method_type = IOMethodType.READ,
  value_type = IOValue.DOUBLE
)
public double getAbsolutePosition() {
    //Check if the robot is real, and if the encoder is null.
    //If it is, log a warning.
    if (Robot.isReal() && encoder == null) {
        Logger.Log(
           "[" + getBaseName() + "]" +
           "WARNING: CANCoder is null, and you're getting the absolute position."
        );
    }
    //If the encoder is null, or the encoder port is less than 0
    //(used mainly for testing), return the cache.
    //If the robot is real, return 0 if it satisfies the above conditions.
    if (encoder == null || this.encoderPort < 0) return isReal() ? 0 : this.cache;

    //Return the absolute position of the encoder,
    //or the cache if the robot is not real.
    return isReal() ? encoder.getAbsolutePosition() : this.cache;
}
```

Ignoring all of the fail-safe conditions of the method, we can see that it's pretty simple to declare the `IOMethod` and to return the value. The IOMethod annotation takes in a name (what the value will be saved as), a method type (read or write), and a value type (either what the input to the method is, or what the output of the method is). The main part of the method is the last line. The `isReal` method is part of the `IOLayer`, and tells whether or not the IOLayer is "real" or simulated. This is edited by both the code, as well as through NT. If the program detects a change in the NT value, it'll send it over to the `IOLayer`.

Through this, it's easy to create a complex system, allowing easy editing and data viewing through NT and other programs as well, such as web dashboards and more.

REST API

One very useful way for us to communicate with the robot is through a REST API. This allows us to send big pieces of information, such as autonomous paths, files, and more without keeping it saved in NT. We also made a REST version of NT, allowing editing and more just through a simple API request.

In order to do this, we combined the built-in Java API for HTTP Servers with our own methods to create a structure similar to Express for Node.JS. For example, here's a simple status page:

```
server.route(new Route("/", new Constructor() {
  @Override
  public Route.Resolution finish(HttpExchange req, Route.Resolution res) {
      return res.send(Element.CreatePage(Element.CreateHead(
              new Element("title").setTextContent("8840-lib")
      ), Element.CreateBody(
              new Element("h1").setTextContent("8840-lib server is running!")
      )));
  }
}));
```

Unfortunately, this is more of a backend thing, so there's not much to write about except for very technical details. If you want to know more, ask us! We'll be happy to share more.

8840-app

Connection To Robot

The web dashboard is connected in many ways, through the REST API and through Network Tables. We use pynetworktables2js for the NTv3 connection to the robot, and we just use the `fetch` API in JS for the REST API.

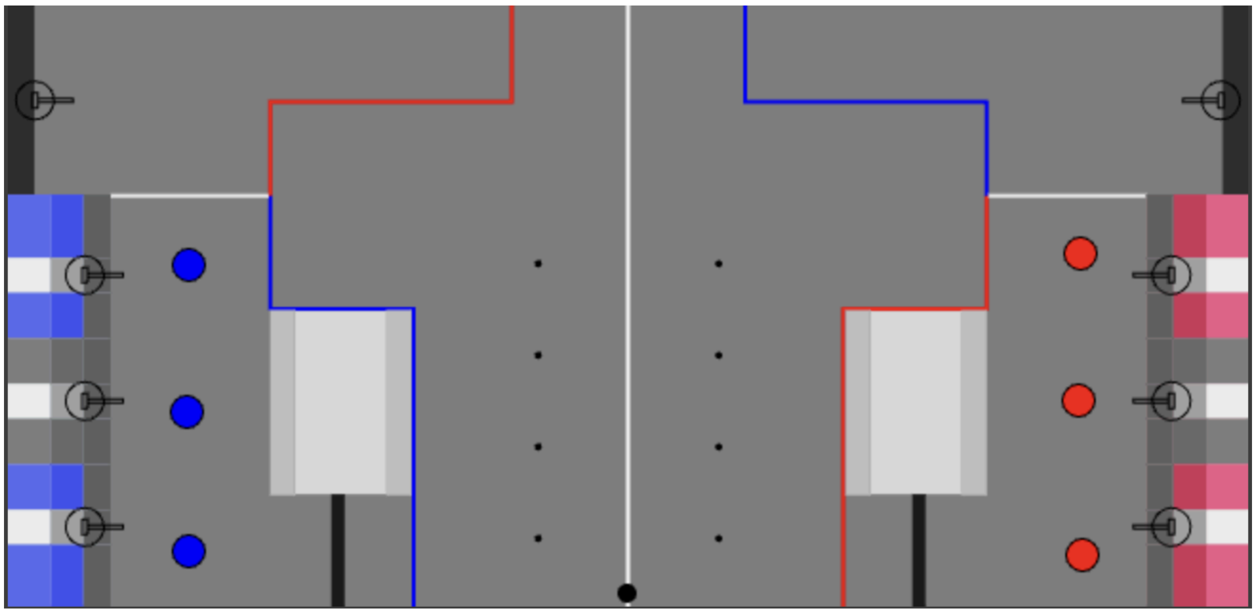
We also allow for easy customization in the web dashboard. The REST API is used for seeing if the robot is connected to the web dashboard, and the NT connection is used for general data gathering purposes. We set the autonomous path and other information through the REST API, so if we aren't running pynetworktables2js at the moment, we can still set the autonomous paths.

Field

2D Field

Display

The display is a simple layout of the field.



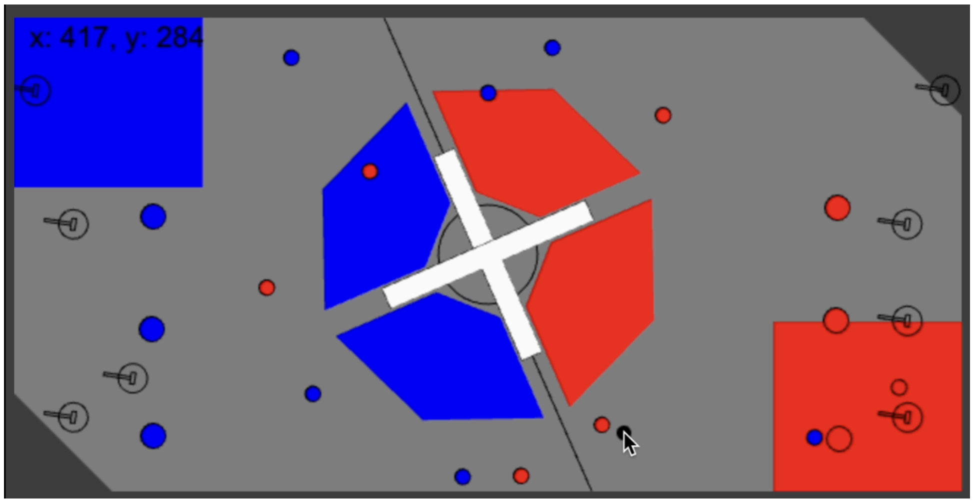
It’s rendered using 2d primitives, mainly rectangles and lines.

All measurements for the field are stored in a large object, and then are used by the Canvas API in JS to render each of the objects. This also allows other components to access this measurement object for various purposes. Each of the measurements are stored in our custom `Unit` class, which allows us to keep track of units with ease. It also allows for custom conversion, for example inches to pixels.

This also allows for easy scaling, and conversion back and forth from pixels to real life units.

Physics

One version of the field component above supports physics. We don’t have physics for this year since there is no real need, but for the 2022 field it supports bouncing the balls off the different field elements.

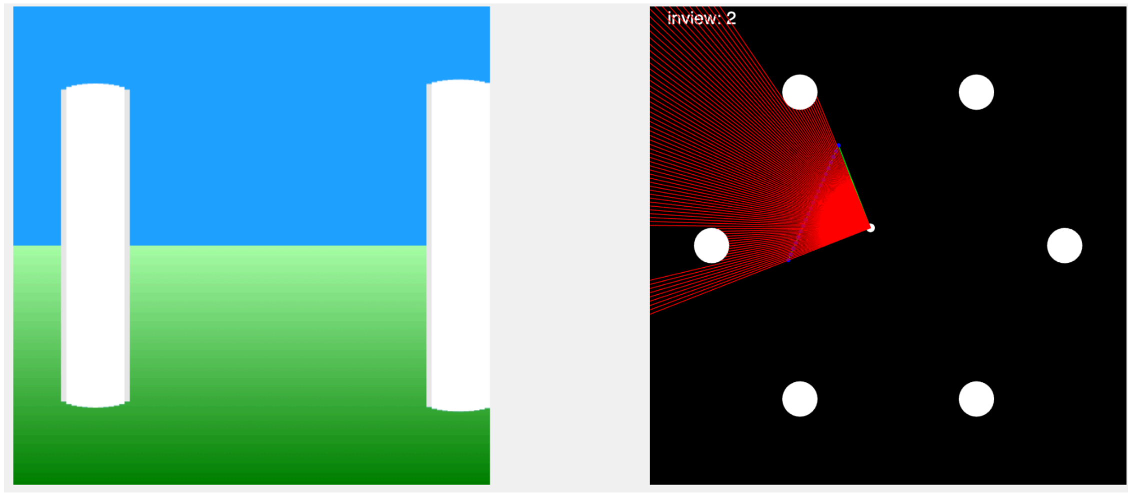


Screenshot from the Rapid React field with the balls bouncing around.

Using this physics engine, we can support things such as simulation and more. We haven't dived deep into usage for this though, but we've been experimenting with Deep-Q Learning.

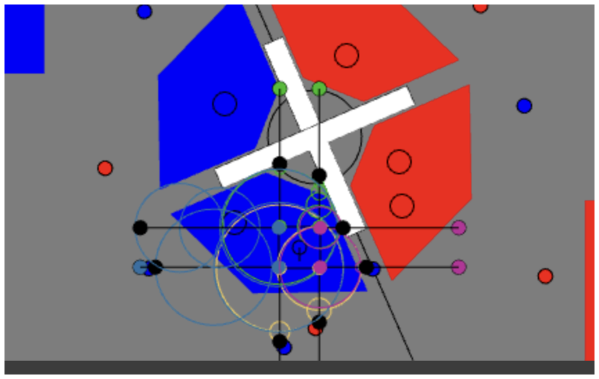
Raymarching

Along with the physics engine, we developed raymarching with these physics objects. This allows us to simulate distance sensors, and develop ways to train AIs with distance and other factors. In order to develop raycasting, we used math from shaders. Here's an example of the same math being used for 3D simulation:



On the right: the 2D map, with the raymarching occurring.
On the left: the 3D projection of what the raymarching detects.

Luckily, this could be applied to the distance sensors as well, but instead of casting many rays, we can just send out one for each distance sensor.

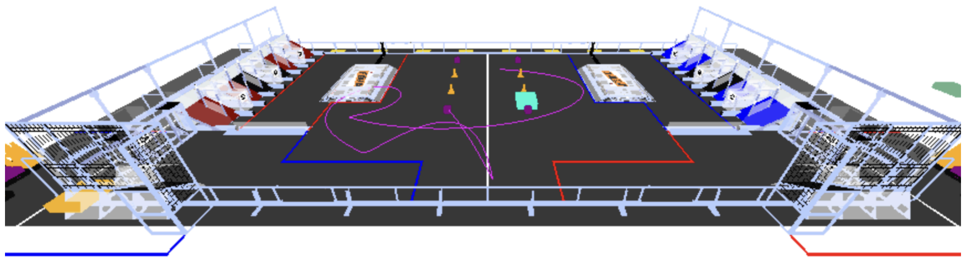

Example of distance sensors detecting physics objects, with the raymarching display visualized.

3D Field

In addition to the 2D field, we also decided to use Three.JS to visualize the field in 3D since the models are available online.

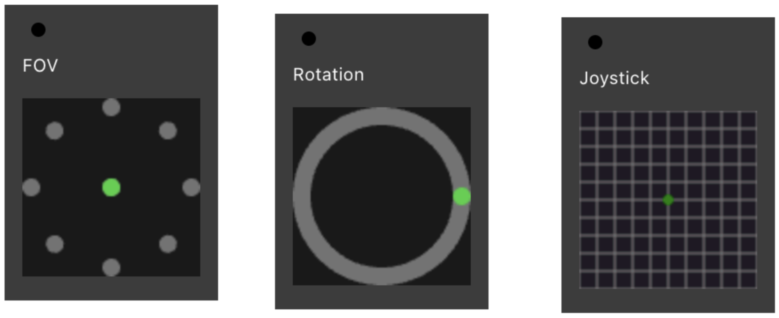
Using the NT connection through pynetworktables2js, we could link the poses of the robot in the 3D field. The field also can be used to visualize the trajectory of the path.

Through the local storage, paths can be sent directly over to the 3D field for quick visualization. If the web dashboard is connected to the robot, using the REST API it can receive all paths that are loaded for the current autonomous. It's a great way to make sure that the current path is doing the right thing, as well as understand how the robot may move.


Example of the 3D field with a path visualized (seen in pink).

Simulated Controls

The simulated controls were very pivotal in the development of the swerve code. Making these controls allowed us to develop code even though we didn't have controllers at times. The controls were very easy to make, just a few displays. Each display took the inputs from the users, then used the connection to NT through pynetworktables2js to send it to the robot. The robot then could take these inputs and send it through the drive system.



A few of the simulated controls.

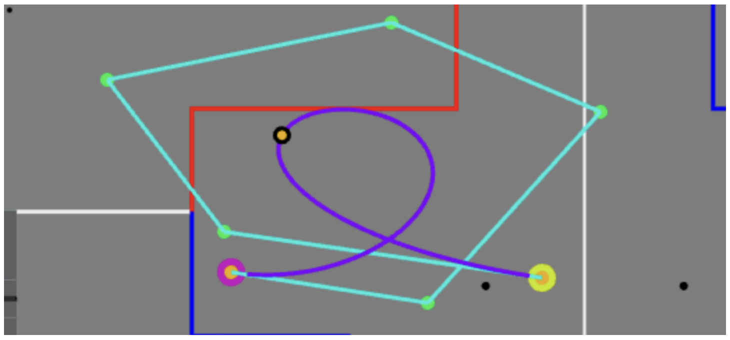
Path Planner

In the off season we decided to make our own trajectory plotting program (we were not creative in naming it). Instead of using Electron, we decided to attempt to make it work in React, and on the web. We used this software to create our autonomous paths for competition and testing. Through this, we developed numerous algorithms to make processing easier and to allow for complex paths.

How does it work?

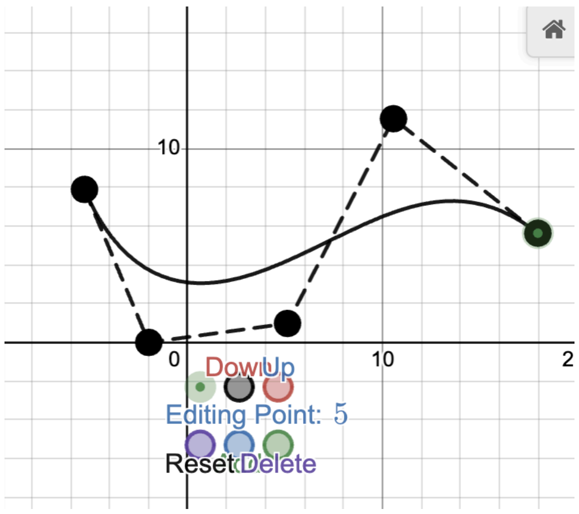
The component from first glance looks a lot like the 2D field. If you said it was the 2D field, you're not wrong. The advantages of using React allows us to use the same exact component for both the 2D physics field as well as for the path planning field, only with differences in what's enabled and disabled.

The path uses two types of points: "soft points" and "hard points." Hard points are where the robot <u>has</u> to be, as well as for where the robot slows down at. The soft points only influence the path - they can be described as having a "pull" towards the path. This allows the path to have curves, and to be translated in many different ways.


Using the soft points to create a loop-de-loop.

If you're immersed in coding or math, you may know that these curves are called Bézier Curves. These curves are mainly used in computer graphics and related fields, and also are perfect for this project.

To first learn how these curves can be used, we created a Desmos graph to figure out how we could create a curve using an infinite number of potential control points.



https://www.desmos.com/calculator/lg0uwiwaxy

In this, we used the formula:

$$B(t,i,n) = \left(\frac{n!}{i!(n-i)!}\right)(1 - t)^{(n-i)}t^{i}$$

Where *t* is between 0 and 1, *i* is the control point index (from 0 to *n*) and *n* is the number of control points subtracted by 1.
Using this function, we can calculate each point through the following coordinates, incrementing *t* by a small value from 0 to 1.

$$\left(\sum_{i=0}^{n} x_i * (B(t,i,n)),\ \sum_{i=0}^{n} y_i * (B(t,i,n))\right)$$

Where x is the list of x-coordinates of the control points, and y is the list of y-coordinates of the control points.
Through this, it was easy to create the Bézier curves, using a $\Delta t = 0.005$. Once we had the paths, it seemed simple enough to implement the positions. We wanted to implement PID controls in order to make the robot slow down as it approaches the hard points, but here's where the problem comes in:

Distances in Bézier Curves don't have a formula.

If we increased t, in straight sections it'll move faster than curved ones, leading to not equal distances between each t value. In order to solve this, we created a "lookup table." Using a small Δt, we could loop through each point, find the distance between there and the last point, add to total distance, and add to the lookup table.

This creates the basic pseudocode:

```
DEFINE TABLE = {}
DEFINE TOTAL DISTANCE = 0
DEFINE LAST POINT = null
DEFINE T = 0
DEFINE DELTA T = [SMALL VALUE]

INCREASE T BY DELTA T:
  IF LAST POINT IS NULL:
      SET LAST POINT TO FIRST POINT
      CONTINUE

  CALCULATE CURRENT POINT FOR T
  FIND DISTANCE FROM LAST POINT AND CURRENT POINT

  ADD DISTANCE TO TOTAL DISTANCE
  ADD CURRENT POINT TO TABLE, WITH KEY OF TOTAL DISTANCE
  SET LAST POINT TO CURRENT POINT

RETURN TABLE
```

After creating the lookup table, we'll need a lookup function with the distance as the input, and the point as the output. We can achieve this by finding the closest point in the curve, or at least the two points that the distance is between. Through this, we can calculate a "weight" of how close the distance is to each point - through this weight, we can translate the closest point by a tiny bit in order to find a "more accurate" position on the bézier curve.

Using this lookup function, we can start generating a path using distances. First, we can split up the different curves by the hard points. Second, we create a distance tracker, and third, we can create a PID controller. The PID isn't necessarily complicated for now, it'll get a tiny bit more complicated later. We first figure out the distance to the next hard point on the curve, and compare it to the distance tracker. Using this, we can feed it to the PID controller and get a distance to move and add it to the distance tracker. When the distance tracker gets pretty close to the hard point, we can move the goal to the next hard point and repeat the process. Unfortunately, we'll find some undesirable behavior here: the robot goes too fast, the robot speeds up too fast, and at the end it can go extremely slow.

Complicating the PID controller by a tiny bit, we can add in a minimum speed, maximum speed, and maximum acceleration. If the output of the PID controller is greater or lesser than the min/max speed, we can clamp it, and make sure that it moves in those bounds. If we record the last movement the distance tracker did, we can compare it to the current movement to find the acceleration. If the acceleration is too high, we can clamp it down to make sure that the robot doesn't speed up too fast.

Through recording the changes to the distance tracker, we can create a list of points on where the robot should be, then save it.

In the path planner, there's also a feature called the timeline. The timeline mainly allows editing of when the robot is moving, and when it's stopped. It's not really used often, we hope to add more functionality in the future. This combined with the distances, we can finalize the trajectory by adding the time element to it. The list is generated with the assumption that a fixed autonomous is being used (from 8840-utils), containing information such as if the robot is driving, positions it should be at, and more. This list can then be handled in the user's program.

Through the path planner, we were able to learn a lot in path generation, as well as developing our coding skills. It was also immensely helpful for understanding how the robot can move, and how we can improve our autos.

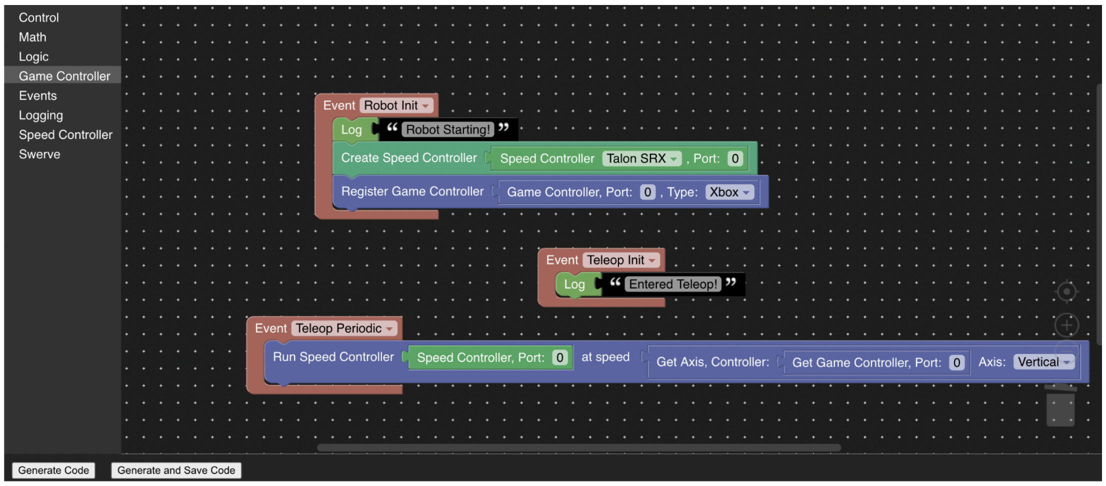
Interconnected Components

Till

Till is one of the more in development features. This is a side project for our team in order to help expand FIRST to beginners, as well as helping our new members understand how coding a robot works. Built with Blockly, Google's block-coding API, we created our twist on FRC programming.

The Editor

The editor on our web-dashboard is built with Blockly. Blockly was immensely helpful for easily creating blocks as well as compiling the blocks into a custom programming language, Till. Using their API, we were able to make basic blocks, such as speed controllers, game controllers, and more.



```
Event Robot Init
    Log "Robot Starting!"
    Create Speed Controller  Speed Controller Talon SRX , Port: 0
    Register Game Controller  Game Controller, Port: 0 , Type: Xbox

Event Teleop Init
    Log "Entered Teleop!"

Event Teleop Periodic
    Run Speed Controller  Speed Controller, Port: 0  at speed  Get Axis, Controller:  Get Game Controller, Port: 0  Axis: Vertical
```

Simple example of a program made

We originally planned on making it ourselves, but it ended up being too time consuming, and this was a great alternative.

The Interpreter

The interpreter lives in 8840-utils. The code built using Till is compiled into a file - it's easy and understandable, so anyone who opens up the files should understand what the program is doing. In the interpreter, we first load the file and find all of the "events," such as teleoperated periodic or robot init. We then store it until one of these methods should be called. When it is, we run the selected code through another process that picks out the commands, evaluates it, then does the desired action. It's pretty simple, and due to the many methods made before in 8840-utils such as `ControllerGroup` or `GameController`, it became effortless.
We hope to expand Till more in the future to bring easier and faster coding to FIRST, as well as provide beginners and more a way to easily start their programming career.

Custom Modules

Custom modules are a helpful way to bring displays to 8840-app. It's basically a recreation of the Canvas API in JavaScript in Java, as well as incorporating NT and other details. We'll talk mostly about the canvas since it's the most used in the custom modules.

Compiler

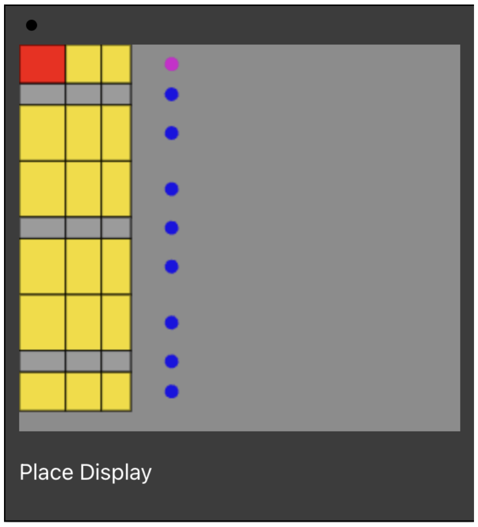
Each of the commands were compiled into two things: `CanvasSuppliers` and commands that took in the `CanvasSuppliers`. These were encoded into strings, then sent along with other information to the interpreter. All of the `CanvasSupplier` inputs were encoded into base64, which led to easy encoding and decoding. Here's an example of how a rectangle is drawn, from our code from this year:

```
//multValX/Y is a multiplier for the x/y.
 rect(
   IfElse(
       nt_value(getCustomBaseNTPath() + "Placing Display/side"),
  CanvasSupplier.IfOperation.EQUAL, s("red"),
       calc(
           MAX(), Calculation.SUBTRACT,
           calc(n(bound.getX() * multValX), Calculation.ADD, n(bound.getWidth() *
  multValX))
       ),
       n(bound.getX() * multValX)
   ), //The x of the position
   n(bound.getY() * multValY), //The y of the position
   n(bound.getWidth() * multValX), //Width
   n(bound.getHeight() * multValY), //Height
   true //Should the rectangle be filled? True for yes, false for no.
 );
```

This uses NT values, if statements, calculations, and more to create the command. The main cons of using this is that numbers and strings need to be passed through the function `n()` or `s()` in order for it to be parsed due to the nature of the interpreter. Once you get past that, it's pretty easy to use, and was especially helpful in developing different components of our robot.

Interpreter

The interpreter is pretty simple. It takes in each command and each `CanvasSupplier`, then recursively decodes each one. The commands are paired up with their respective functions in JS's Canvas API. The `CanvasSupplier` in JS only calculates values when it's needed, which can lead to functions such as `MAX()` existing in the Java side, which can be used as 100% of the width/height. We can get displays like this:



The grid display for this year.