Programming - Code Outline

This section will explain and lay out the process the software development team went through in writing our current code base. All the way from the first meet to now.

Table of Contents:

Recent Changes:

In The Beginning - League Meet One

Last season, which was our first year as a team, we were unable to abstract out the hardware definitions and configuration into a separate class. This season, however, that was the first thing we did. Compared to trying to figure it out last season, it seemed much more straight forward. We were very excited to see how much we have grown compared to last year, so it was a good start to the season.

Drive Class

Once we got the hardware class sorted out, we got to work on the drive function. Now that we had one year under our belt, we were able to organize our code with a little more foresight. Our drive function, for example, consisted mainly of update functions in the game loop. We separated them based on motor type, `updateAuxMotors`, `updateCrServos`, `updateServos`, and `updateDrive`. The last of which had a helper function `getDrivePower`. These will be outlined below.

`runOpMode`

Our `runOpMode` function is shown below. `h` is the name of our abstracted hardware class which will be laid out later on. `h` also handles telemetry (prefixed with `t`) — `h.tErr` and `h.tDrivePower` in this case. For now, we will focus on the code under the comment "Updates hardware," which are local functions.

```
@Override void runOpMode() {
    // Initiate hardware
    try {
        h.init(hardwareMap);
    } catch(Exception e) {
        h.tErr("HardwareMap", e);
        sleep(15_000);
        stop();
    }
    h.tStatus("Ready");
    waitForStart();
    h.tStatus("Running");
    try {
        // Update Loop
        while (opModeIsActive()) {
            // Updates gamepad
            h.updateGamepad(gamepad1, gamepad2);
            // Updates hardware
            updateDrive();
            updateAuxMotors();
            updateCrServos();
            updateServos();
            // Updates telemetry
            h.tDrivePower();
        }
        // Catches exceptions as plain-text
    } catch(Exception e) {
        h.tErr("Runtime", e);
        sleep(15_000);
        stop();
    }
}
```

Before that, regarding the try/catch blocks. This season, we vowed to use error handling for once, evident by the two try/catch blocks wrapping the majority of our code. Although it is rudimentary, it was a good start. Later on, we plan to advance our error handling into something more beneficial, especially during autonomous periods.

`updateServos`

Without getting into Hardware's — the abstracted hardware class — territory, the functions included (`update*`) are all housed within the same class. `updateServos` is shown below. It simply toggles the position of the servos we use to latch on to the foundation

```
private boolean a = true; // Toggle (actual value doesn't matter)
private void updateServos() {
    // Sets f-hook positions
    if (h.button_b) {
        h.fHook_l.setPosition(a ? 1.0 : 0.0);
        h.fHook_r.setPosition(a ? 0.0 : 1.0);
        a = !a;
        sleep(50);
    }
}
```

`updateCrServos`

Next is `updateCrServos`. This turns the servos clockwise or counterclockwise depending on the side. It uses nested ternary operators, which could probably be a little more readable. However, the resulting operation is relatively simple so we have not had issues on that front.

```
private void updateCrServos() {
    // Sets grabber positions
    h.grab_l.setPower(h._rTrigger != 0.0 ? 1.0 : (h.lTrigger != 0.0 ? -1.0 :
0.0));
    h.grab_r.setPower(h._rTrigger != 0.0 ? -1.0 : (h.lTrigger != 0.0 ? 1.0 :
0.0));
}
```

`updateAuxMotors`

Now, `updateAuxMotors`. We have two auxiliary motor sets, one is to move the robots top platform forward and back, `h.lSlide_l` and `h.lSlide_r`, the other moves the scissor contraption up and down, `h.scissor`.

```
private void updateAuxMotors() {
    // Sets scissor-lift's motor powers
    h.scissor.setPower(-h._lStick_y);
    h.lSlide_l.setPower(-h._rStick_y);
    h.lSlide_r.setPower(h._rStick_y);
}
```

`updateDrive` `getDrivePower`

The final one, `updateDrive`, is the most involved. It uses a helper function called `getDrivePower`. They are shown below.

```
private void updateDrive() {
    h.drive_lf.setPower(getDrivePower(0));
    h.drive_rb.setPower(getDrivePower(1));
    h.drive_lb.setPower(getDrivePower(2));
    h.drive_rf.setPower(getDrivePower(3));
}
```

```
private double getDrivePower(int motorID) {
    double power;
    double modL1 = 0.6;
```

```
    double modL2 = 0.3;
    switch (motorID) {
        case 0: power = h.lStick_y - h.lStick_x - h.rStick_x; break; // drive_lf
        case 1: power = h.lStick_y - h.lStick_x + h.rStick_x; break; // drive_rb
        case 2: power = h.lStick_y + h.lStick_x - h.rStick_x; break; // drive_lb
        case 3: power = h.lStick_y + h.lStick_x + h.rStick_x; break; // drive_rf
        default: power = 0; break;
    }
    power = power > 1 ? 1 : power;
    power *= h.lTrigger != 0 ? modL2 : (h.rTrigger != 0 ? modL1 : 1);
    return power;
}
```

The `mod*` variables are at the request of our driving team to include different levels of throttle, enabled by holding down a button. We include two levels (three if you include full speed — the default), `L1` runs at 60% and `L2` at 30%. As for the switch statement, the algorithm used to obtain the correct arrangement of `-` and `+` was pure trial and error. At first, we were sure there was a more elegant solution (note the line directly below the switch statement is necessary, as the value sometimes goes above 1) but were unable to find one, thus, trial and error it was.

Hardware Class

The addition of a separate hardware class is definitively the biggest improvement to our code from the previous season so far. First we will lay out the initializations, including the constructor.

```
public Hardware(Telemetry telemetry, LinearOpMode linearOpMode) {
    t = telemetry; opmode = linearOpMode;
}
LinearOpMode opmode;
Telemetry t;

// Initialize hardware
DcMotor drive_lf, drive_rb, drive_rf, drive_lb;
DcMotor scissor, lSlide_l, lSlide_r;
Servo fHook_l, fHook_r;
CRServo grab_l, grab_r;

// Initialize visual detection
OpenCvCamera phoneCam;
SkystoneDetector sDetect;

// Initialize gamepad values
double lStick_x, lStick_y, rStick_x, lTrigger, rTrigger; // Gamepad 1
boolean button_b;
double _lStick_y, _rStick_y; // Gamepad 2
```

A big reason we were able to make this, is because we finally began to understand how to interact with objects, hence the `Telemetry` and `LinearOpMode` objects passed via the constructor. In our first year, we never would have thought to do that. Below the constructor are the other global variables assigned to null values.

`init` `initAuto`

We include two separate initialization functions, `init` and `initAuto`.

```
void init(HardwareMap hardwareMap) {
    // Defines drive motors
    drive_lf = hardwareMap.dcMotor.get("leftFront");
    drive_rb = hardwareMap.dcMotor.get("rightBack");
    drive_lb = hardwareMap.dcMotor.get("leftBack");
    drive_rf = hardwareMap.dcMotor.get("rightFront");

    // Drive motor setup
    drive_lf.setDirection(DcMotor.Direction.FORWARD);
    drive_rb.setDirection(DcMotor.Direction.REVERSE);
    drive_lb.setDirection(DcMotor.Direction.FORWARD);
    drive_rf.setDirection(DcMotor.Direction.REVERSE);

    // Defines scissor-lift hardware
    scissor = hardwareMap.dcMotor.get("scissor");
    lSlide_l = hardwareMap.dcMotor.get("slideL");
    lSlide_r = hardwareMap.dcMotor.get("slideR");
    fHook_l = hardwareMap.servo.get("hook1");
    fHook_r = hardwareMap.servo.get("hook2");
    grab_l = hardwareMap.crservo.get("block1");
    grab_r = hardwareMap.crservo.get("block2");
}
```

```
void initAuto(HardwareMap hardwareMap) {
    WebcamName webcamName = hardwareMap.get(WebcamName.class, "Webcam 1");
    int cameraMonitorViewId =
hardwareMap.appContext.getResources().getIdentifier("cameraMonitorViewId", "id",
hardwareMap.appContext.getPackageName());
    phoneCam = new OpenCvWebcam(webcamName, cameraMonitorViewId);
    phoneCam.openCameraDevice();
    sDetect = new SkystoneDetector();
    phoneCam.setPipeline(sDetect);

    phoneCam.startStreaming(320, 240, OpenCvCameraRotation.UPRIGHT);
}
```

The `init` function includes basic configuration of our hardware devices. `initAuto` presents a good opportunity to credit where we got the majority of out vision related code from, DogeCV. While we do eventually move to using TensorFlow, DogeCV was our main vision library for all of last season and early this season. So, to be honest, `initAuto`, aside from straightforward commands like "startStreaming," was beyond us. None of us have ever used OpenCV or any kind of video stream processing, so our only hope was to rely on DogeCV. We never end up getting this vision detection method working reliably (problems with lighting and no clue how to fix it), so this is simply a mention of the fact that it existed, and we tried...

`updateGamepad`

The rest of the hardware class consists of many telemetry function, which we will not go over here, and a function to update the gamepad values. `updateGamepad` is shown below.

```
void updateGamepad(Gamepad gamepad1, Gamepad gamepad2) { // to be referenced
externally
    lStick_x = gamepad1.left_stick_x;
    lStick_y = gamepad1.left_stick_y;
    rStick_x = gamepad1.right_stick_x;
    lTrigger = gamepad1.left_trigger;
    rTrigger = gamepad1.right_trigger;
    _lStick_y = gamepad2.left_stick_y;
    _rStick_y = gamepad2.right_stick_y;
    _rTrigger = gamepad2.right_trigger;
}
```

Auto class

Our initial autonomous classes included just one instruction set and one helper class. First, we will go over the instruction set, `Auto`.

```
@Autonomous(name="Auto") public class Auto extends LinearOpMode {
    // Initializations
    private Hardware h = new Hardware(telemetry, this);
    private AutoBase a = new AutoBase(h, this);

    // Runs when initialized
    @Override public void runOpMode() {
        // Initiate hardware
        try {
            h.init(hardwareMap);
            h.initAuto(hardwareMap);
        } catch(Exception e) {
            h.tErr("HardwareMap", e);
            sleep(15_000);
            stop();
        }
        h.tStatus("Ready");
        waitForStart();
        try {
            // Instructions:
                // INSTRUCTIONS GO HERE

        // Catches exceptions as plain-text
        } catch(Exception e) {
            h.tErr("Auto Runtime", e);
            sleep(15_000);
            stop();
        }
    }
}
```

It is very similar to Drive's `runOpMode` function, just without the loop. Stripped down to code that's actually ran, it looks like this.

```
@Autonomous(name="Auto") public class Auto extends LinearOpMode {
    // Initializations
    private Hardware h = new Hardware(telemetry, this);
    private AutoBase a = new AutoBase(h, this);
```

```
    // Runs when initialized
    @Override public void runOpMode() {
        // Initiate hardware
        h.init(hardwareMap);
        h.initAuto(hardwareMap);

        h.tStatus("Ready");
        waitForStart();

        // Instructions:
            // INSTRUCTIONS GO HERE
    }
}
```

So yeah, pretty simple, the bulk ends up being the error handling. Sadly, we do not have that actual autonomous from this point in time. We wrote all the instructions mid-meet but neglected to save it. With just one autonomous class to work with, we had to switch between two sets anyways. They looked something like this.

```
// Instructions:
a.movF_(2_500, 1)
a.movR_(2_000, 1)
// OR
a.movF_(2_500, 1)
a.movL_(2_000, 1)
```

We will explain what exactly `AutoBase` and `mov*_` are next.

AutoBase Class

`AutoBase` is definitely the class we spend our most time in. It's purpose is to abstract out as many instructions as possible, this makes our autonomous instruction sets incredibly simple. This is important for in-the-field adjustments. First, we will go over the helper functions. We have many individual functions, but they fall into only a few templates. We will go over those.

`driveTargetPos` etc.

The first ones are drive-specific helper functions. These are things like `driveTargetPos` which is simply shorthand for setting the position of all the drive motors.

```
void driveTargetPos(double revlf, double revrf, double revlb, double revrb) {
    int mod = 28 * 20
    h.drive_lf.setTargetPosition((int)(revlf * mod));
    h.drive_rf.setTargetPosition((int)(revrf * mod));
    h.drive_lb.setTargetPosition((int)(revlb * mod));
    h.drive_rb.setTargetPosition((int)(revrb * mod));
}
```

We never actually get our motor's encoders working correctly for the first meet, so these functions are untested at this point in time. The purpose of the `* mod` is to convert the revolutions we give it to the ticks that the motors take as input. The other ones, also encoder related, are mode changing functions. So things like this.

```
void driveModeSRE() {
    h.drive_lf.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    h.drive_rf.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    h.drive_lb.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
    h.drive_rb.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
}
```

More shorthand.

```
private boolean drive_isBusy() {
    return h.drive_lf.isBusy() || h.drive_rf.isBusy() || h.drive_lb.isBusy() ||
h.drive_rb.isBusy();
}
```

The last two have to do with time based movement, the method that we used at the meet. As with most, they are just shorthand for the drive motors.

```
void drivePower(double plf, double prf, double plb, double prb) {
    h.drive_lf.setPower(plf);
    h.drive_rf.setPower(prf);
    h.drive_lb.setPower(plb);
    h.drive_rb.setPower(prb);
}
```

```
void halt(long t) {
    opmode.sleep(t);
    h.drive_lf.setPower(0);
    h.drive_rf.setPower(0);
    h.drive_lb.setPower(0);
    h.drive_rb.setPower(0);
}
```

`mov*`

Now onto the stuff we actually use in the instruction sets, that is, `mov*` and `mov*_`. `movF` and `movF_` are shown here.

```
void movF(double rev, double p) {
    driveModeSRE();
    driveTargetPos(rev, rev, rev, rev);
    driveModeRTP();
    drivePower(p, p, p, p);
    while (drive_isBusy()) {
        h.tDrivePos();
        opmode.idle();
    }
    halt(0);
    driveModeRWE();
}
```

```
void movF_(long t, double p) {
    drivePower(p, p, p, p);
    halt(t);
}
```

This set of functions would be considered a level of abstraction higher than the helper functions described earlier. This makes them rather straight forward, especially the time based ones, since those are already pretty simple without the extra abstraction. These elevate the autonomous to our highest level of abstraction and are the only functions currently used in `Auto`'s instruction sets. For an example of how we adjust these for different directions, here is `movL` and `movL_`.

```
void movL(double rev, double p) {
    driveModeSRE();
    driveTargetPos(-rev, rev, rev, -rev); // Added (-)'s
    driveModeRTP();
    drivePower(p, p, p, p);
    while (drive_isBusy()) {
        h.tDrivePos();
        opmode.idle();
    }
    halt(0);
    driveModeRWE();
}
```

```
void movL_(long t, double p) {
    drivePower(-p, p, p, -p); // Added (-)'s
    halt(t);
}
```

League Meet Two/Three

Here, we will outline changes from League Meet One to League Meet Two/Three (League Meet Two and League Meet Three occurred on the same day). The `Drive` class didn't actually change, so we will be starting from `Hardware`.

Hardware Class

`initAuto` `startStream` `stopStream`

At this point, we are starting to understand the DogeCV library to the point where we can personalize some of the code.

```
void initAuto(HardwareMap hardwareMap) {
    WebcamName webcamName = hardwareMap.get(WebcamName.class, "Webcam 1");
    int cameraMonitorViewId =
hardwareMap.appContext.getResources().getIdentifier("cameraMonitorViewId", "id",
hardwareMap.appContext.getPackageName());
    phoneCam = new OpenCvWebcam(webcamName, cameraMonitorViewId);
    phoneCam.openCameraDevice();
    ssDetect = new SkystoneDetector();
    phoneCam.setPipeline(ssDetect);

    ssDetect.useDefaults();
    startStream();
}
```

The majority of the `initAuto` function remains the same, aside from some variable name changes and `userDefaults`. We understand the code a lot better now though, where as before, we didn't understand what most of the code was for or why it was necessary. We also added some quality-of-life functions shown below.

```
void startStream() {
    OpenCvCameraRotation direction = OpenCvCameraRotation.SIDEWAYS_LEFT;
    phoneCam.startStreaming(screenWidth, screenHeight, direction);
}
```

```
void stopStream() {
    phoneCam.stopStreaming();
}
```

Aside from this, we added many more telemetry functions for debug purposes, but those will not be outlines here.

AutoBase Class

`AutoBase` had a ton of work put into it. We finally got our encoders working properly. We also, finally, added vision functionality.

`findSkystone`

`findSkystone` was a big step for us, now that we understood DogeCV a little more, we were able to create this algorithm to detect the skystone. We just use one function for this and have a parameter to change the direction it searches in. It is shown below.

```
double findSkystone(int dir, double p) { // Searches for a skystone in the given
direction. When it finds one, it will move towards it
    mov(dir, p); // move left(3) or right(1)
    h.tSub("Scanning");
    while(h.ssDetect.foundRectangle().area() < 5500 || (dir == 3 ?
h.ssDetect.getScreenPosition().y < 73 : h.ssDetect.getScreenPosition().y > 160))
{
        h.tCaminfo();
        h.tRunTime();
        opmode.idle();
    }
    halt(0);
    return opmode.getRuntime();
}
```

The reason it returns the runtime, is because we chose to use that to determine which configuration the stones were placed; the more time it took, the closer the Skystone was to the wall.

`pickUp` `drop`

Now that we were able to find the Skystone, we needed a way to pick it up and drop it. We created two functions, `pickUp` and `drop`, to handle that. They are shown below.

```
void pickUp() { // Will extend platform, attempt to grab a block, then retract
the platform
    h.tSub("Picking up Block");
    platform(2.5, 0.6);
    h.grab_l.getController().setServoPosition(h.grab_l.getPortNumber(), 1);
    h.grab_r.getController().setServoPosition(h.grab_r.getPortNumber(), 0);
    opmode.sleep(400);
    platform(-2.3, 0.6); h.tSub("");
}
```

```
void drop() { // Will extend platform, release any held block, then retract the
platform
    h.tSub("Dropping Block");
    h.grab_l.getController().setServoPosition(h.grab_l.getPortNumber(), 0);
    h.grab_r.getController().setServoPosition(h.grab_r.getPortNumber(), 1);
    opmode.sleep(400);
}
```

They are relatively simple. Just some communication with `Hardware`. We had troubles figuring out how to interact with a CR servo as though it were a regular servo, but figured it out eventually.

`latch` `unlatch`

In order to grab the foundation in autonomous, we created `latch` and `unlatch`. As with `pickUp` and `drop`. We simply interacted with `Hardware` a bit.

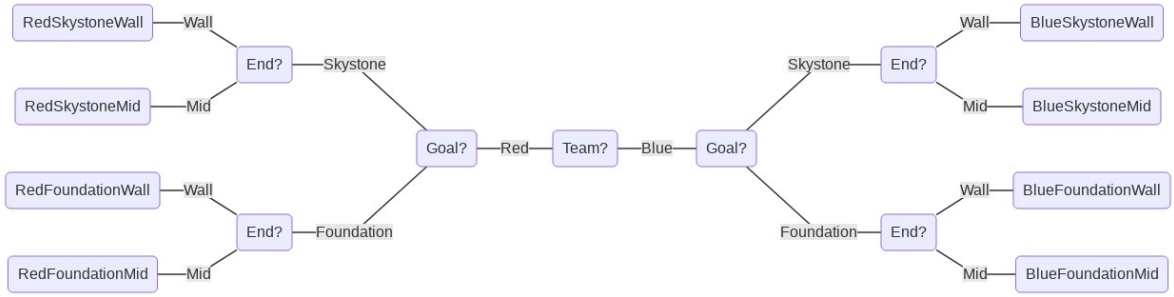
```
void latch() {
    h.fHook_l.setPosition(0.0);
    h.fHook_r.setPosition(1.0);
    opmode.sleep(400);
}
```

```
void unlatch() {
    h.fHook_l.setPosition(1.0);
    h.fHook_r.setPosition(0.0);
    opmode.sleep(400);
}
```

We also added some more helper functions, in preparation for better error handling and what not, but they generally follow the same format as League Meet One's section on helper functions.

Auto Classes

Notice the plural. We finally have multiple Autonomous classes! Very exciting! How many do we have? Eight! We will only go over two though. The eight classes were derived using this flow chart.



The specific classes shown will be `RedSkystoneWall` and `BlueFoundationMid`. By `Wall` and `Mid`, we mean ending the game on the tape close to the middle or close the wall. They are quite similar to the original template, but, of course, with instructions this time.

```
@Autonomous(name="Blue Foundation Mid") public class BlueFoundationMid extends
LinearOpMode {
    /*Initializations*/
    private Telemetry tele = telemetry;
    private __Hardware__ h = new __Hardware__(tele, this);
    private __AutoBase__ a = new __AutoBase__(h, this);

    // Runs when initialized
    @Override
    public void runOpMode() {
        // Initiate hardware
        try {
            h.init(hardwareMap);
        } catch (Exception e) {
            h.tStatus("Error");
            h.tErr("HardwareMap", e);
            sleep(15_000);
            stop();
```

```
        }
        h.tStatus("Ready");
        waitForStart();
        resetStartTime();
        try {
            h.tStatus("Running");
            /* Instructions - Blue Foundation Mid */
            h.tStatus("Latching");
            a.movB(1.0, 1.0);
            a.movR(9.0, 1.0);
            a.movB(6.0, 1.0);
            a.movB(0.8, 0.5);
            a.latch();

            h.tStatus("Unlatching");
            a.movF(7.0, 1.0);
            a.movF(1.4, 0.5);
            a.unlatch();

            h.tStatus("Line");
            a.movL(8.0, 1.0);
            a.movB(5.5, 1.0);
            a.movL(5.0, 1.0);

            h.tStatus("Done!");
            /* End */
        } catch (Exception e) { // Catches exceptions as plain-text
            h.tStatus("Error");
            h.tErr("Auto Runtime", e);
            sleep(15_000);
            stop();
        }
    }
}
```

```
@Autonomous(name="Red Skystone Wall") public class RedSkystoneWall extends
LinearOpMode {
    /*Initializations*/
    private Telemetry tele = telemetry;
    private __Hardware__ h = new __Hardware__(tele, this);
    private __AutoBase__ a = new __AutoBase__(h, this);
    private __Skystone__ s = new __Skystone__();

    // Runs when initialized
    @Override
    public void runOpMode() {
        // Initiate hardware
        try {
            h.init(hardwareMap);
            h.initAuto(hardwareMap);
        } catch (Exception e) {
            h.tStatus("Error");
            h.tErr("HardwareMap", e);
            sleep(15_000);
            stop();
        }
        h.tStatus("Ready");
```

```
        waitForStart();
        resetStartTime();
        try {
            h.tStatus("Running");
            /* Instructions - Red SkyStone Wall */
            a.movF(3.1, 1.0);
            double sTime = a.findSkystone(3, 0.6);//4.2 far stone 3.7 middle 2.2
end
            a.movF(3.0, 1.0);
            a.pickUp();

            h.tStatus("Moving to Foundation");
            a.movB(0.5, 1.0);
            a.trnR(1.0, 0.9);
            a.movR(6.0, 1.0);
            a.movF(sTime > 3.5 ? 12.7 : 10.0, 0.4);
            a.drop();

            h.tSub("Moving under Bridge");

            a.movB(4.0, 1.0);

            h.tStatus("Done!");
            /* End */
        } catch (Exception e) { // Catches exceptions as plain-text
            h.tStatus("Error");
            h.tErr("Auto Runtime", e);
            sleep(15_000);
            stop();
        }
    }
}
```

These numbers were all derived via trial and error. The foundation side instruction set is pretty simple, just instructions. The skystone side, however, has some variability in the form of `a.findSkystone` and `sTime`. Looking at the line utilizing `sTime`, `a.movF`, you can see that instead of three possible values (one per possible skystone position), there are two. We were able to get away with this by overlapping two of the. Since it uses time for the calculation, it is already pretty imprecise, so adding to that couldn't hurt. Sadly we had issues with lighting and were not able to test that theory. Other than that, the foundation autonomous' performed wonderfully. Aside from some hardware issues, there wasn't one instances that the foundation autonomous failed due to programming or imprecision. Which is kind of necessary for all autonomous', so we were happy to see such a success.

Qualifiers

Here we will outline large changes in the code used in the Qualifying Tournament.

Hardware

The main change was the switch from DogeCV to TensorFlow in order to find the Skystone. DogeCV was extremely inconsistent in different lighting. Combined with our fundamentally unreliable movement algorithm made it too risky to use in matches. TensorFlow, however, uses machine learning, removing the hurdles of coding for different lighting. Here is the new `initAuto` function and its related code.

```
// Initialize Vision
float SkystonePos;
float SkystoneLeft;
float SkystoneRight;
int SkystoneArea;
double SkystoneConfidence;
List<Recognition> updatedRecognitions;
VuforiaLocalizer vuforia;
TFObjectDetector tfDetect;
private static final String VUFORIA_KEY;
```

initAuto

```
void initAuto(HardwareMap hardwareMap) {
    /* Initiate Vuforia */
    VuforiaLocalizer.Parameters parameters = new VuforiaLocalizer.Parameters();
    parameters.vuforiaLicenseKey = VUFORIA_KEY;
    parameters.cameraName = hardwareMap.get(WebcamName.class, "Webcam 1");
    vuforia = ClassFactory.getInstance().createVuforia(parameters);

    /* Initiate TensorFlow Object Detection */
    int tfodMonitorViewId =
hardwareMap.appContext.getResources().getIdentifier("tfodMonitorViewId", "id",
hardwareMap.appContext.getPackageName());
    TFObjectDetector.Parameters tfodParameters = new
TFObjectDetector.Parameters(tfodMonitorViewId);
    tfodParameters.minimumConfidence = 0.8;
    tfDetect = ClassFactory.getInstance().createTFObjectDetector(tfodParameters,
vuforia);
    tfDetect.loadModelFromAsset("Skystone.tflite", "Stone", "Skystone");

    if (tfDetect != null) tfDetect.activate();
}
```

updateTfDetect

```
void updateTfDetect() {
    updatedRecognitions = tfDetect.getUpdatedRecognitions();
    if (updatedRecognitions == null) return;
    if (updatedRecognitions.size() > 0) {
        int i = 0;
        for (int j = 0; j < updatedRecognitions.size(); j++) {
            if (updatedRecognitions.get(j).getLabel() == "Skystone") {
                i = j;
                break;
            }
        }
        if (updatedRecognitions.get(i).getLabel() == "Skystone") {
            float objectRight = updatedRecognitions.get(i).getRight();
            float objectLeft = updatedRecognitions.get(i).getLeft();
            float objectHeight = updatedRecognitions.get(i).getHeight();
            float objectWidth = updatedRecognitions.get(i).getWidth();
            float objectConfidence = updatedRecognitions.get(i).getConfidence();

            SkystoneLeft = objectLeft; SkystoneRight = objectRight;
```

```
            SkystonePos = objectLeft + (round(100 * ((objectWidth) / 2)) / 100);
            SkystoneArea = round(objectWidth * objectHeight * 100) / 100;
            SkystoneConfidence = objectConfidence;
        }
    } else if (updatedRecognitions.size() == 0) {
        SkystonePos = 0;
        SkystoneArea = 0;
        SkystoneConfidence = 0;
    }
}
```

Auto Classes

The main change to our foundation autonomous was switching the foundation's orientation at the end of autonomous. It used to be facing the same as it was when the round started, but now we have the robot put it up against the wall, giving our alliance more room to move on foundation side. We also added some handy comments to help navigate the instructions at a glance. ('-' = 1 revolution, '>' = 1/6 power, '=' = 1/3 power)

Instructions for red-foundation-mid autonomous

```
h.tStatus("Running");
/* Instructions - Red Foundation Mid */
a.movB(1.0, 0.9, 0.8); // Back -]
a.movL(11.5, 0.9, 3.8); // Left -----------]
a.movB(5.4, 0.9, 2.5); // Back -----]
a.movB(1.1, 0.5, 1.2); // Back -)
a.latch(); // Latch
a.customTrn(1.0, 0.3, 3400); // Custom Turn
a.movB(3.8, 1.0, 1.8); // Back ----)
a.unlatch(); // Unlatch
a.movF(1.0, 1.0, 0.8); // Forward -]
a.movR(5.4, 1.0, 1.5); // Right ----]
a.movF(9.0, 1.0, 3.5); // Forward ---------]
h.tStatus("Done!");
/* End */
```

AutoBase Class

Our encoder-based movements methods now require a time. This is used as a failsafe in case the motors are not able to completely reach the target distance.

`mov*`

```
void movF(double rev, double p, double t) {
    ElapsedTime elapsedTime = new ElapsedTime();
    driveModeSRE();
    driveTargetPos(rev, rev, rev, rev);
    driveModeRTP();
    drivePower(p);
    // While all the drive motors are busy and failsafe time is not reached
    while (drive_isBusy() && (elapsedTime.seconds() < t || t == 0)) {
        opmode.idle();
    }
    halt(0);
    driveModeRWE();
}
```

`findSkystone`

```
double findSkystone(int dir, double p) {
    /**Searches for a skystone in the given direction.
     * When/if finds one, the robot will move towards it.
     * Checks for Skystone based on the position on the screen,
     * area, and confidence.
     */
    driveModeSRE();
    driveModeRUE(); // Needed for returning encoder pos
    ElapsedTime elapsedTime = new ElapsedTime();
    mov(dir, p); // move left[3] or right[1]
    // Three second buffer. Needed because
    while (elapsedTime.seconds() < 3.0) {
        h.tRunTime(elapsedTime);
        opmode.idle();
    }
    do {
        if (h.tfDetect != null) h.updateTfDetect();
        h.tRunTime(elapsedTime);
        opmode.idle();
        // Failsafe in case it doesn't detect anything
        if (elapsedTime.seconds() > 8.3) {
            h.tStatus("Failed");
            halt(0);
            return -1.0;
        }
    } while ((dir == 3 ? h.SkystonePos < 460 : h.SkystonePos > 420) &&
h.SkystoneArea < 40_000 && h.SkystoneConfidence < 0.75 &&
opmode.opModeIsActive());
    halt(0);
    return Math.abs(h.drive_lf.getCurrentPosition() / 560); // 560 = motor tick
rate
}
```

We also had to add a turn method that turns on a circle rather than turning in place. This is used when orientating the foundation. It uses time, which adds some error, but with just one instance of it, it is not too noticeable.

`customTrn`

```
void customTrn(double leftPower, double rightPower, long t) {
    drivePower(leftPower, rightPower, leftPower, rightPower);
    opmode.sleep(t);
    halt(0);
}
```

Super Qualifiers

For the Super Qualifying Tournament, we made some pretty substantial changes regarding code organization, core algorithms used to accomplish tasks, and improving reliability.

Drive

We also expanded the scope of our speed switching algorithm at the request of the drivers. Now it incorporates unique speeds for strafing in addition to forward and backwards and turning. The code is shown below.

```
double getPower(int i) {
    double power;
    switch (i) {
        case 0: power = -h.lStick_y + h.lStick_x + h.rStick_x; break; //
drive_lf
        case 1: power = -h.lStick_y + h.lStick_x - h.rStick_x; break; //
drive_rb
        case 2: power = -h.lStick_y - h.lStick_x + h.rStick_x; break; //
drive_lb
        case 3: power = -h.lStick_y - h.lStick_x - h.rStick_x; break; //
drive_rf
        default: power = 0; break;
    }
    // 0.2 is buffer, constants are speed modifiers for different situations
    // Strafing speed : Zero input speed
    double mod_one = (h.lStick_x >= 0.2 || h.lStick_x <= -0.2) ? 0.3 * 1.9 :
0.3;
    double mod_two = h.rTrigger != 0 ? 0.65 : 0.8; // Speed lvl 1 : Speed lvl 2
    power *= h.lTrigger != 0 ? mod_one : mod_two;
    if (power > 1) power = 1;
    return power;
}
```

AutoBase

First improvement was the algorithm used to detect the Skystone. In the Qualifiers we were not able to confidently say we could detect Skystones, so for the Super Qualifiers, it was top priority. First step was changing the core algorithm. Before, we had the robot start from the far Skystone and work it's was in at a slow and constant pace, tracking the position of any Skystones to make sure it is aligned. This was flawed in both these ways, so we got rid of both of them. Rather than a slow constant pace, detecting all the way through, we had the robot step from stone to stone in increments, only activating the Skystone detection when we are stopped. With this method, we were also able to get rid of trying to track the position, because the robot would always, consistently be aligned with the target stone through the use of encoders. The new algorithm is shown below.

`findSkystone`

```
double findSkystone(boolean blue, double p) {
    /**Scans for the skystone, starting from the stone farthest from
     * the wall and incrementally approaches the wall, stopping after
     * scanning the third stone. Returns -1 if it didn't find a skystone.
     * A value from {1, 2, 3} is returned if success, 1 corresponds to
     * the farthest stone from the wall, 3 the third farthest.
     */
    h.tSub("Finding Skystone");
    driveModeSRE();
    for (int i = 1; i <= 3; i++) {
        if (!opmode.opModeIsActive()) return -1;
        ElapsedTime elapsedTime = new ElapsedTime();
        h.tfDetect.activate();
        while (elapsedTime.seconds() < 2.5 && opmode.opModeIsActive()) {
            h.updateTfDetect();
            h.tCaminfo(1);
            if (h.sArea > 50_000) {
                h.tSub("Success");
                halt(0);
                h.tfDetect.deactivate();
                return i;
            }
            opmode.idle();
        }
        h.tfDetect.deactivate();
        if (i == 3) {
            h.tSub("Failed");
            halt(0);
            return -1.0;
        }
        if (blue) movR(2.55, 2.0, 2.0);
        else movL(2.55, 2.0, 2.0);
    }
    h.tSub("Failed");
    halt(0);
    return -1.0;
}
```

We were also able to simply a lot of the common code, for instance, we refactored out the encoder-based movement algorithm to a single central function with necessary variables. This algorithm and `movF` which makes use of it is shown below.

`movEncoder`

```
void movEncoder (List<Double> rev, double p, double t) {
    ElapsedTime elapsed = new ElapsedTime();
    driveModeSRE();
    driveTargetPos(rev.get(0), rev.get(1), rev.get(2), rev.get(3));
    driveModeRTP();
    drivePower(p);
    while (drive_isBusy() && (elapsed.seconds() < t || t == 0) &&
opmode.opModeIsActive()) {
        opmode.idle();
    }
    if (elapsed.seconds() >= t) h.tSub("Timed Out");
    halt(0);
    driveModeRWE();
}
```

`mov*`

```
void movF(double rev, double p, double t/*= 0*/) {
    h.tSub("Moving Forward");
    movEncoder(Arrays.asList(-rev, -rev, -rev, -rev), p, t);
} void movF(double rev, double p) { movF(rev, p, 0); }
```

We also scrapped the time-based custom turn in favor of one that uses encoders. The main issue with this was figuring out the math involved, but after some research and trial-and-error, we derived a consistent algorithm. This algorithm is shown below.

`cTrn*`

```
void cTrnL(double radius, int degrees, double p, double t) {
    h.tSub("Custom Turn Left");
    ElapsedTime elapsed = new ElapsedTime();
    double mod = 6.6;
    driveModeSRE();
    // (radius * [degrees to radians]) * constant
    double outerArc = radius * (Math.PI / 180) * degrees * mod;
    double innerArc = (radius - 1.5) * ((Math.PI / 180.0) * degrees * mod);
    driveTargetPos(-innerArc, -outerArc, -innerArc, -outerArc);
    driveModeRTP();
    double lP = (1 - (1.5 / radius)) * p;
    drivePower(lP, p, lP, p);
    while (drive_isBusy() && (elapsed.seconds() < t || t == 0) &&
opmode.opModeIsActive()) {
        opmode.idle();
    }
    if (elapsed.seconds() >= t) h.tSub("Timed Out");
    halt(0);
    driveModeRWE();
}
```

Another case of refactoring we are very proud of, is the abstraction of all our autonomous into just two algorithms with just two boolean variables `blue` and `mid`. This let's us make our eight autonomous classes extremely short and all the values are located in just two places (Very exciting! No more changing the same value in 4 different places!). The relevant methods are shown below.

FoundationBase

```
/* Instructions - Foundation */
a.movB(3.8, 1.2, 2.8); // Back ----]
a.movB(2.8, 0.5, 3.6); // Back ---)
a.latch(); // Latch

if (blue) a.cTrnL(1.9, 90, 2.0, 3.5); // Blue: Custom Turn Left 90 degrees
else a.cTrnR(1.9, 90, 2.0, 3.5); // Red: Custom Turn Right 90 degrees

a.movB(8.0, 1.0, 2.9); // Back --------]
a.unlatch(); // Unlatch
a.movF(1.0, 1.0, 0.7); // Forward -]

if (mid){
    if (blue) a.movL(6.4, 1.5, 3); // Blue Mid: Left ------])
    else a.movR(6.4, 1.5, 3); // Red Mid: Right ------])
}

a.movF(8.5, 1.0, 3.5); // Forward ---------]


h.tStatus("Done!");
```

SkystoneBase

```
/* Instructions - SkyStone */
a.movF(3.5, 1.0, 2.4); // --]

// sPos = Position of skystone, -1 if failed
double sPos = a.findSkystone(blue, 0.6);

if (sPos == -1) a.movF(1.7, 1.0, 1.5); // Failed: Forward --]
else {
    a.movF(2.2, 1.2, 1.5); // Success: Forward --]
    a.pickUp(); // Success: Pick Up
    a.movB(1.3, 1.2, 0.7); // Success: Back -]
}

if (blue) a.trnL(1.0, 2.0, 1.0); // Blue: Turn Left 90 degrees
else a.trnR(1.0, 2.0, 1.0); // Red: Turn right 90 degrees

if (!mid) {
    if (blue) a.movL(6.2, 2.0, 2.4); // Blue Wall: Left ------]]
    else a.movR(6.2, 2.0, 2.4); // Red Wall: Right ------]]
}
// 1 = farthest from wall, 3 = nearest
a.movF(sPos == 1 ? /*1 =*/8.5 : (sPos == 2 ? /*1 =*/10.5 : /*1 =*/12.5), 3.0,
2.6);
if (sPos != -1) {
    a.drop(); // Success: Drop
    a.movB(3.3, 3.0, 1.5); // Success: ---]]]
}

h.tStatus("Done!");
```

Autonomous Class Examples

```
@Autonomous(name="B Found. Mid", group="Foundation")
public class _BlueFoundationMid extends LinearOpMode {
    private FoundationBase base = new FoundationBase(this, telemetry);
    @Override
    public void runOpMode() {
        base.init();
        // Runs Foundation instruction set
        base.run(true, true); // Blue, Mid
    }
}
```

```
@Autonomous(name="R Skystone Mid", group="Skystone")
public class _RedSkystoneMid extends LinearOpMode {
    private SkystoneBase base = new SkystoneBase(this, telemetry);
    @Override
    public void runOpMode() {
        base.init();
        // Runs Skystone instruction set
        base.run(false, true); // Red, Mid
    }
}
```

```
@Autonomous(name="R Found. Wall", group="Foundation")
public class _RedFoundationWall extends LinearOpMode {
    private FoundationBase base = new FoundationBase(this, telemetry);
    @Override
    public void runOpMode() {
        base.init();
        // Runs Foundation instruction set
        base.run(false, false); // Red, Wall
    }
}
```