

1951Z

CODING NOTEBOOK



[Table of Contents](#)

What is coding to team 1961Z?

Coding is a very important aspect of every VEX robotics team. Thus, it is why we have spent months tuning and perfecting the program to make sure that our functions work as efficiently and fluidly as possible. We also utilize a programming scheme known as “crawl”, “walk”, and “run”. These serve as backups of our functions in the order of simplicity for the unfortunate case that it ends up breaking during a competition. The following notebook will go over the important features of our programming for our new robot “Riptide” which allows us to have a competitive advantage over other teams through heavy implementation of programmed functions to allow ease of driving in a match.

We also believe that sharing our knowledge of programming is the best way to further improve teams around us in terms of a competitive advantage. Thus, we have a short coding series on our Youtube Channel that goes through the different coding aspects that are required to learn to excel in VRC.

Link: <https://tinyurl.com/wwz98bn9>

Table of Contents

[Table of Contents](#)

[Auto-Indexing](#)

[Its Purpose:](#)

[Our Method:](#)

[The Code:](#)

[Testing](#)

[Assisted Driver Functions](#)

[Its Purpose:](#)

[Our Method:](#)

[The Code:](#)

[Testing](#)

[Movement Functions](#)

[Its Purpose:](#)

[Our Method:](#)

[The Code:](#)

[Testing](#)

[Unity Simulation](#)

[Purpose](#)

[Method](#)

[Code](#)

Auto-Indexing

Its Purpose:

The first major key was the code we coined AutoIndexing. The purpose of this code was so that the driver did not have to be too focused on the conveyor and indexing the balls properly so he could store 3 at one time. Instead, the code would take care of that for the driver and allow him to focus only on the base.

Our Method:

To achieve such code I first had my designers add sensors in the robot so we could track the movements of the ball in the robot accurately and have the code understand where the ball is placed. Once the sensors were added I got to work. The list of the sensors is one Line tracker at the bottom near the intake to detect when a ball has entered the conveyor. A limit switch at the top so the code can understand where to stop the first ball and another limit switch located near the hood to detect when a ball leaves the robot. To create this code I based it on complex logic statements including if statements and operators to determine how the code would function. Another major factor in this code was a ball counter we had running continuously which will be later discussed in the notebook. Using these systems I was able to create a working auto-indexing that greatly helps the driver and reduces the load while driving.

The Code:

```
int intakeToggle() {
    while (true) {
        if(Controller1.ButtonR1.pressing() && Controller1.ButtonR2.pressing() &&
(indexer.velocity(pct) < 100)){
            setIntakeSpeed(-100);
            indexer.spin(fwd, -100, pct);
            ballC = 0;
            whenToStop = 1;
            task::stop(scoreGoal);
        }
    }
}
```

This first section of the code deals with the controller presses and when the driver presses both right shoulder buttons at the same time it will flush out any balls in the conveyor and resets the ball count.

```
else if (Controller1.ButtonR1.pressing()) {
    intake.spin(directionType::fwd, intakeSpeedPCT, voltageUnits::volt);
    if(LineTrackerIntake.reflectivity() > 17){
        task intakingBalls = task(scoreGoal);
    }
}
```

This section of the code deals with the actual auto-indexing part of the program. The way it works is if the driver is pressing the top right shoulder button it will spin the intakes and if the Line sensor in the intakes detects a ball it will execute the auto-indexing code.

```

else if (Controller1.ButtonR2.pressing()) {
    intake.spin(directionType::rev, intakeSpeedPCT, voltageUnits::volt);
}
else if(Controller1.ButtonUp.pressing()){
    task intakingBalls = task(scoreGoal);
}
else {
    brakeIntake();
    if(whenToStop % 2 != 0){
        task g = task(outtake0Ball);
        whenToStop = 0;
    }
}

printf("leftfront: %i\n", whenToStop);
task::sleep(10);
}
}

```

The rest of the code deals with miscellaneous procedures and making other functionality like only outtaking the intakes are given to the driver so he can handle any scenario.

Now we will discuss how we create a ball counter to keep track of how many balls there are in the robot at all times.

```
int ballC = 0;

bool senseTop = false;
bool senseBottom = false;

int bcount() {
    while (true) {
        printf("ball count: %id\n", ballC);
        if (ballC <= 3 && ballC >= 0) {
            if (LineTrackerIntake.reflectivity() > 17 && senseBottom == false) {
                ballC++;
                senseBottom = true;
            } else if (LineTrackerIntake.reflectivity() < 10) {
                senseBottom = false;
            }

            /*if(trackerOuttake.pressing() && senseTop == false) {
                ballC--;
                senseTop = true;
            } else if (!trackerOuttake.pressing()) {
                senseTop = false;
            }*/
        }
        task::sleep(10);
    }
    return ballC;
}
```

The way this code works is that the ball count is initialized at the beginning of every match and runs for the duration of the entire match only being rested when the conveyor gets flushed out. The way ball count works are that if the line tracker in the intakes detects a ball it adds one to the ball count. The other commented code was so that if the top limit switch was pressed the ball count would decrease by one but there were physical issues with shooting once that was added so we had to remove the code and

the switch and remove balls code based whenever the driver hits a macro that is later discussed in the coding notebook.

Now we will move on to the actual logic of how the auto-indexing code works and a few clips of the robot in action to provide a better understanding.

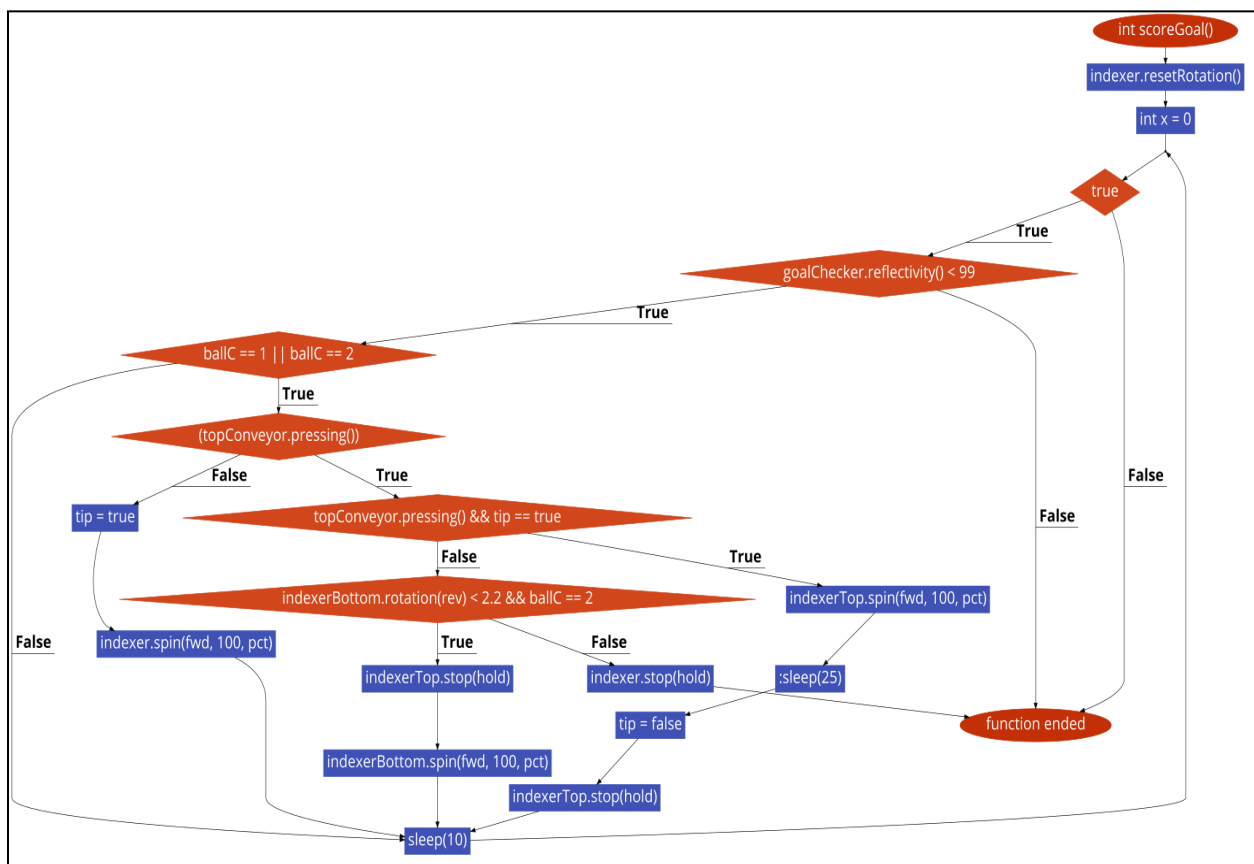
```
int scoreGoal() {
    indexer.resetRotation();
    int x = 0;
    while (true) {
        if (goalChecker.reflectivity() < 99) {
            if (ballC == 1 || ballC == 2) {
                if (!(topConveyor.pressing())) {
                    tip = true;
                    indexer.spin(fwd, 100, pct);
                } else if (topConveyor.pressing() && tip == true) {
                    indexerTop.spin(fwd, 100, pct);
                    task::sleep(25);
                    tip = false;
                    indexerTop.stop(hold);
                } else if (indexerBottom.rotation(rev) < 2.2 && ballC == 2) {
                    indexerTop.stop(hold);
                    indexerBottom.spin(fwd, 100, pct);
                } else {
                    indexer.stop(hold);
                    break;
                }
            }
        } else {
            break;
        }
        task::sleep(10);
    }
    return 1;
}
```

The first condition of the auto-indexing is to make sure that the line tracker that is in the front of the robot determines that there is no goal located in front of the robot. It then moves on to the second condition which determines if there are 1 or 2 balls in the

[Table of Contents](#)

conveyor via ball counter. Once that is met it will move on to the second condition which is if there is no ball at the top where the line sensor is meaning it's not pressed the whole conveyor will spin till it reaches that limit. Once touching I added a bit of delay as the ball would slip down if the code immediately stopped after the ball pressed the limit switch. If none of those conditions are met and there is a second ball in the conveyor only the bottom two rollers will spin ensuring the top ball stays in the correct position and that the second ball is stored as high as possible. Finally, the last condition is if a third ball is detected the code would automatically break as there is not enough space to fully flush three balls in the conveyor. The code is also written via an int function so we can run this function as a multi-thread in the Operator Control to make sure it doesn't interfere with other programs running in the driver.

Auto-indexing Logic flow chart:



Testing

This code and concept took me two weeks to perfect and make sure it was reliable that the driver could trust the program and it would function as promised. Below is a link to a few videos that show the different aspects of auto-indexing working (I was the one driving in the video, so the robot doesn't move that nicely):

<https://photos.app.goo.gl/Gmf3Ct47JiQ9X99fA>

<https://photos.app.goo.gl/VEsrkHSxDsefcozB6>

Assisted Driver Functions

Its Purpose:

To go along with Auto-indexing I created macros(assisted driver functions). The purpose of this code was so that the driver did not have to be too focused on the number of balls shot out of the conveyor. Instead, the code would take care of that for the driver and allow him to focus only on the base.

Our Method:

To achieve such code I based the majority of the ball counter code that was discussed previously. I also used the top line sensor because in some cases the balls have to be staggered when shot out of the robot so the top line sensor helped in controlling the balls so they always landed in the goal. Mostly this code was very straightforward to write and was based on how many revolutions the motor has spun. Using these systems I was able to create working macros that greatly help the driver and reduces the load while driving.

The Code:

```
int outtake1Ball() {
    indexer.resetRotation();
    int x = 0;
    int ballCount = ballC;
    if(ballCount >= 1){
        indexerTop.spin(fwd, 100, pct);
        task::sleep(100);
    }
    while (true) {
        if( !(topConveyor.pressing()) && ballCount > 1){
            indexer.spin(fwd, 100, pct);
            printf("done");
        }
        else if (indexer.rotation(rev) < 2 && ballCount == 1) {
            indexer.spin(fwd, 100, pct);
        }
        else {
            if(ballCount >= 1){
                indexer.spin(fwd, 100, pct);
                task::sleep(50);
                indexerTop.stop(hold);
                indexerBottom.stop(hold);
            }
            ballC--;
            break;
        }
        task::sleep(10);
        x+=10;
    }
    return 1;
}
```

The first section of the code deals with when the driver presses the top left shoulder button on the controller, it will call the outtake1ball() function. This causes the robot to check the number of balls that remain in the robot, if there are more than 1 one ball in the robot, the robot will spin the top indexer first, then spin the main conveyor until the top

limit switch which senses when the ball is in the top spot. Once the ball has been shot out the top rollers and conveyors will then stop holding the remaining balls in the robot. If there is only one ball in the robot, the conveyor and top roller will start spinning until it is sensed that the ball has left the robot which will then stop the roller and conveyors which then allows the robot to be ready to pick up more balls.

```
int outtake2Ball() {
    indexerTop.resetRotation();
    int x = 0;
    int ballCount = ballC;
    while (true) {
        if (fabs(indexerTop.rotation(rev)) < 9) {
            if (ballCount == 2) {
                indexer.spin(fwd, 100, pct);
            } else {
                indexerTop.spin(fwd, 100, velocityUnits::pct);
                if (!topConveyor.pressing()) {
                    while (x < 200) {
                        task::sleep(100);
                        x += 100;
                    }
                    indexerBottom.spin(fwd, 100, pct);
                } else {
                    indexerBottom.stop(brake);
                }
            }
        } else {
            indexer.stop(brake);
            ballC -= 2;
            break;
        }
        task::sleep(1);
    }
    return 1;
}
```

The second section of the code deals with when the driver presses the bottom left shoulder button on the controller, it will call the outtake2ball() function. This function works the same as before but the number of rotations that the conveyor spins before

balls are detected are increased so that 2 balls can be shot out of the robot. If the robot has two balls within the robot, it will shoot out both balls simultaneously or back to back. If there are three balls the robot will stagger the balls by 100 ms each to allow enough space for the third ball to be sensed and stopped by the robot so that it can be held for future use.

```
int outtake3Ball() {
    indexerTop.resetRotation();
    while (true) {
        if (fabs(indexerTop.rotation(rev)) < 9) {
            indexer.spin(fwd, 100, pct);
        } else {
            indexer.stop(brake);
            ballC -= 3;
            break;
        }
        task::sleep(1);
    }
    return 1;
}
```

The third section of the code deals with when the driver presses the left button of the left d-pad on the controller, it will call the outtake3ball() function. This function will spin the conveyor and roller for more rotations as the robot has to deposit three balls, but since there will be no balls remaining, the indexer system can spin the entire time and simply stop to allow the robot to be primed for further usage within the game.

```

int conveyorToggle() {
    while (true) {
        if (Controller1.ButtonLeft.pressing()){
            task m = task(outtake3Ball);
        }
        else if (Controller1.ButtonL1.pressing()) {
            task f = task(outtake1Ball);
        }
        else if (Controller1.ButtonL2.pressing()){
            task q = task(outtake2Ball);
        }
        else{
            //nothing needed in else but it just makes the code look a bit cleaner
        }
        task::sleep(10);
    }
}

```

This is a layout of all the tasks needed to be able to run all three macros at the same time so that the driver only needs to focus on picking up balls and scoring using the respective macros depending on the number of balls the robot has and how many balls the driver wishes to score.

Testing

This code was relatively easy to make and it was reliable enough that the driver could trust the program and it would function as promised. Issues arise with certain scenarios which need code-specific case statements that would catch these rare events. The issues usually were from conflicts with how the auto-indexing system worked, but once bug fixes were implemented the macro and auto-indexing system became robust. Without further ado, here is how the assisted driver function code works:

<https://photos.app.goo.gl/wh5gDPCYWm7XG2mX6>

Movement Functions

Its Purpose:

Movement functions are a very important aspect of the robotics competition as you want your robot to be as fast as possible and as accurate. Finding this balance is a challenge that every team faces every year and each takes a different approach. For me, I believed a combination of motion profiling, PID, and heading control allowed me to have the speed and accuracy that I desired from my robots. I have been working on this movement code for the past three years in robotics and I have finally achieved the goal that I have been working towards and I am proud to share the code with you.

Our Method:

This code has taken me years of research and reading research papers, looking at how others implemented movement functions, and most importantly hours of debugging. The code utilizes the two inertial and two trackers we have on robots and uses them to accurately determine where on the field the robot is located. I will be splitting the code into multiple parts one discussing the turn PID and the other the main moveForward function that I have created.

The Code:

```
struct PID{
    float current;
    float kP;
    float kI;
    float kD;
    float target;
    float integral;
    float error;
    float derivative;
    float lastError;
};
```

The main reason I created this structure was that PID was gonna be used in a lot of my functions and since the values for kP, kI, and kD were gonna be different for every function it would make more sense to create as a struct so I could easily reuse the code later on as you will see below.


```

PID sRotatePid;

int iRotatePid(int target) {
    sRotatePid.kP = 0.458; //0.458
    sRotatePid.kI = 0;
    sRotatePid.kD = 0.6;

    sRotatePid.current = get_average_inertial();
    sRotatePid.error = target - sRotatePid.current;
    sRotatePid.integral += sRotatePid.error;
    sRotatePid.derivative = sRotatePid.error - sRotatePid.lastError;
    sRotatePid.lastError = sRotatePid.error;

    return (((sRotatePid.error) * (sRotatePid.kP)) + ((sRotatePid.derivative) *
(sRotatePid.kD)) + ((sRotatePid.integral) * (sRotatePid.kI)));
}

```

This is where the struct comes into play as I am easily able to create the math behind how PID works. The way it works is that it takes the current position and then finds the Error which is the target - current value this value is considered as the P or proportional value. The next step is to calculate the I which is taking the integral of the error over a certain time frame so the way to calculate that is to constantly add the error after every loop this is the I or the integral value. Finally, the D value or derivative is calculated by using error - last error which you store at the end of every loop. After that to return the value you multiply each corresponding value with a certain constant that you tune over time with a certain parameter which are:

Parameter	Rise Time	Overshoot	Settling Time	Steady-State Error	Stability
kP	Decrease (faster)	Increase (further)	N/A	Decrease (more precise)	Worsens
kI	Decrease (faster)	Increase (further)	Increase (takes longer)	Decrease (more precise)	Worsens
kD	N/A	Decrease (closer)	Decrease (quicker)	N/A	Improves*

- [Wikipedia](#)

This is just the math behind the code. The next part will discuss how math is used to move the robot with precision and speed while turning.

```

void rotatePID(int angle, int maxPower) {
    resetFunction();
    double maxError = 0.1;
    int timer = 0;
    double minVelocity = 0.5;
    exit_function = false;
    while (fabs(get_average_inertial() - angle) > maxError && !exit_function) {
        int PIDPower = iRotatePid(angle);
        printf("heading inertial  %f\n", get_average_inertial());
        int power = abs(PIDPower) < maxPower ? PIDPower : maxPower * (PIDPower /
abs(PIDPower));
        leftDrive.spin(fwd, -power, velocityUnits::pct);
        rightDrive.spin(fwd, power, velocityUnits::pct);
        if (timer > 800 && fabs(leftDrive.velocity(pct)) < minVelocity) {
            exit_function = true;
        }
        wait(10, msec);
        timer += 10;
    }
    brakeDrive();
}

```

The code works by taking 2 parameters: the max speed desired by the user and the angle. Then it goes into a while statement that only exists once the angle determined by the inertial is less than the error set at and the exit function condition has been met. The way the power is calculated is that it puts the desired angle in the PID function above that is calculated every 10 msec. The next step is that it uses a ternary operator to determine if the value calculated by the PID is above the max speed set by the user and then if it caps it at that limit. Then it sets both the left drive and right driver with that velocity. Finally, the code checks with the exit condition, and as long as the speed of the leftDrive is less than the set minVelocity and the timer that is running during the code is above a certain threshold will exit stopping the drive. This code works in both ways that even if the robot overshoots the code will fix it and have it go in the opposite direction. I have spent a month tuning the PID constants and with those properly tuned the code is both super consistent and quick.

```

void moveForwardWalk(double distanceIn, double maxVelocity, double
headingOfRobot, double multiply, bool cancel = true) {

    static const double circumference = 3.14159 * 2.77;
    if (distanceIn == 0)
        return;
    double direction = distanceIn > 0 ? 1.0 : -1.0;
    double wheelRevs = ((distanceIn) / circumference);
    double wheelRevsDegree = ((360 * distanceIn) / circumference);
    distanceAtEnd = distanceAtEnd / circumference;
    resetFunction();
    double leftStartPoint = (leftTracker.rotation(rotationUnits::rev));
    double rightStartPoint = (rightTracker.rotation(rotationUnits::rev));
    double leftSpeed, rightSpeed;
    double offset = 0;

    int sameEncoderValue = 0;
    double distanceTraveled = 0;
    double PIDPowerL, PIDPowerR, PIDPowerHeading;

```

This is the beginning of the moveForward function that I have worked for years on. It takes many parameters but most stays constant unless need to be changed. The distanceIn allows the user to determine how many inches they want the robot to go. The maxVelocity parameter sets the max velocity the robot is allowed to go. The headingOfRobot is to determine the angle the user would prefer the robot to go. The multiply is so that the user can tune the input levels to the heading control. Finally, cancel is preset to true but if you don't want the function to cancel you can set it to false. The rest of this code deals with determining how many revolutions the wheels need to travel to reach the target distance and initializing variables.

```

while ((direction * (get_average_encoder() - rightStartPoint) < direction *
wheelRevs) || (direction * (get_average_encoder() - leftStartPoint) < direction *
wheelRevs)) {

    distanceTraveled = (get_average_encoder());

    if ((goalChecker.reflectivity() > 120 || (fabs(leftDrive.velocity(pct)) < 1)
|| (fabs(rightDrive.velocity(pct)) < 1)) && cancel == true &&
fabs(distanceTraveled) > 0.1) {
        ++sameEncoderValue;
    }

    if(sameEncoderValue > 10){
        break;
    }

    //y\ =\ 2\cos\left(\frac{x}{20}\right)\ +2

    PIDPowerHeading = iHeadingPid(headingOfRobot);
    headingError = fabs(PIDPowerHeading) < multiply ? PIDPowerHeading : multiply *
(PIDPowerHeading / fabs(PIDPowerHeading));

    if (direction == fabs(direction)) {
        headingError = headingError;
    } else {
        headingError = headingError;
    }

    printf("heading %f\n", get_average_inertial());
    printf("headingError %f\n", PIDPowerHeading);
    //printf("distanceTraveled %f\n", distanceTraveled);
}

```

The code then moves into the while loop that loops while the distance traveled is less than the calculated wheel revolution. The code then sets the variable distanceTraveled to the current position. Next, it calculates the heading control power of the robot using the heading PID loop. We also created print statements in the loop as a way to make sure the code is performing properly.

Heading PID Math:

```
PID sHeadingPid;

int iHeadingPid(float target) {
    sHeadingPid.kP = 4;
    sHeadingPid.kI = 0;
    sHeadingPid.kD = 1.6;

    sHeadingPid.current = get_average_inertial();
    sHeadingPid.error = target - sHeadingPid.current;
    sHeadingPid.integral += sHeadingPid.error;
    sHeadingPid.derivative = sHeadingPid.error - sHeadingPid.lastError;
    sHeadingPid.lastError = sHeadingPid.error;

    return (((sHeadingPid.error) * (sHeadingPid.kP)) + ((sHeadingPid.derivative) *
(sHeadingPid.kD)) + ((sHeadingPid.integral) * (sHeadingPid.kI)));
}
```

Distance PID Math:

```
PID sMovePid;

int iMovePid(float target) {
    sMovePid.kP = 0.13; //0.13
    sMovePid.kI = 0;
    sMovePid.kD = 0.2;

    sMovePid.current = get_average_encoder_deg();
    sMovePid.error = target - sMovePid.current;
    sMovePid.integral += sMovePid.error;
    sMovePid.derivative = sMovePid.error - sMovePid.lastError;
    sMovePid.lastError = sMovePid.error;

    return (((sMovePid.error) * (sMovePid.kP)) + ((sMovePid.derivative) *
(sMovePid.kD)) + ((sMovePid.integral) * (sMovePid.kI)));
}
```

[Table of Contents](#)

Speed PID Math:

```
PID sSpeedPid;

float iSpeedPid(float target, bool leftSide) {
    sSpeedPid.kP = 0.16; //2
    sSpeedPid.kD = 0;
    sSpeedPid.kI = 0;

    if(leftSide == true){
        sSpeedPid.current = leftDrive.velocity(pct);
    }
    else{
        sSpeedPid.current = rightDrive.velocity(pct);
    }
    sSpeedPid.error = target - sSpeedPid.current;
    if(leftSide == true){
        printf("error left %f\n", sSpeedPid.error);
    }
    else{
        printf("error right %f\n", sSpeedPid.error);
    }
    sSpeedPid.integral += sSpeedPid.error;
    sSpeedPid.derivative = sSpeedPid.error - sSpeedPid.lastError;
    sSpeedPid.lastError = sSpeedPid.error;

    return (((sSpeedPid.error) * (sSpeedPid.kP)) + ((sSpeedPid.derivative) *
(sSpeedPid.kD)) + ((sSpeedPid.integral) * (sSpeedPid.kI)));
}
```

Motion Profiling Math:

```
double increasing_speed(double starting_point, double current_position, double
addingFactor) { // how fast the robot starts to pick up its speed
    static
    const double acceleration_constant = 60.0; //tuned 80
    return acceleration_constant * fabs(current_position - starting_point) +
        (50);
}
```

[Table of Contents](#)

LeftDrive:

```
if (direction * (distanceTraveled - leftStartPoint) < direction * wheelRevs) {
    if ((fabs(distanceTraveled) < 0.8)) {
        if(direction == fabs(direction)){
            leftSpeed = maxVelocity;
        }
        else{
            leftSpeed = std::min(increasing_speed(leftStartPoint, distanceTraveled,
offset), maxVelocity);
        }
        leftSpeed = (direction * (leftSpeed - headingError));
        front_L.spin(fwd, leftSpeed, pct);
        back_L.spin(fwd, leftSpeed, pct);
    } else {
        PIDPowerL = iMovePid(wheelRevsDegree);
        PIDPowerL = fabs(PIDPowerL) < maxVelocity ? PIDPowerL : maxVelocity *
(PIDPowerL / fabs(PIDPowerL));
        printf("left Speed %f\n", PIDPowerL);
        PIDPowerL = ((PIDPowerL - headingError));
        front_L.spin(fwd, (PIDPowerL * 0.01) * 12, volt);
        back_L.spin(fwd, (PIDPowerL * 0.01) * 12, volt);
    }
    printf("left Speed Real %f\n", leftDrive.velocity(pct));
    if ((direction * (wheelRevs - 0.1)) > fabs(distanceTraveled) &&
fabs(distanceTraveled) > 0.1) {
        leftSpeed += iSpeedPid(leftSpeed, true);
    }
    leftDrive.spin(fwd, leftSpeed, pct);
} else {
    brakeDrive();
}
```

The next part is where the magic is done. In this loop, if the distance is less than a certain amount the robot will be set at max speed and while doing some testing because more weight is in the front of the bot when going backward the robot would tip if we just set at max speed so instead of using my old code I had written I used the motion profiling curve to have the robot accelerate to top speed when going backward. Then the motor would be set to that speed which would be

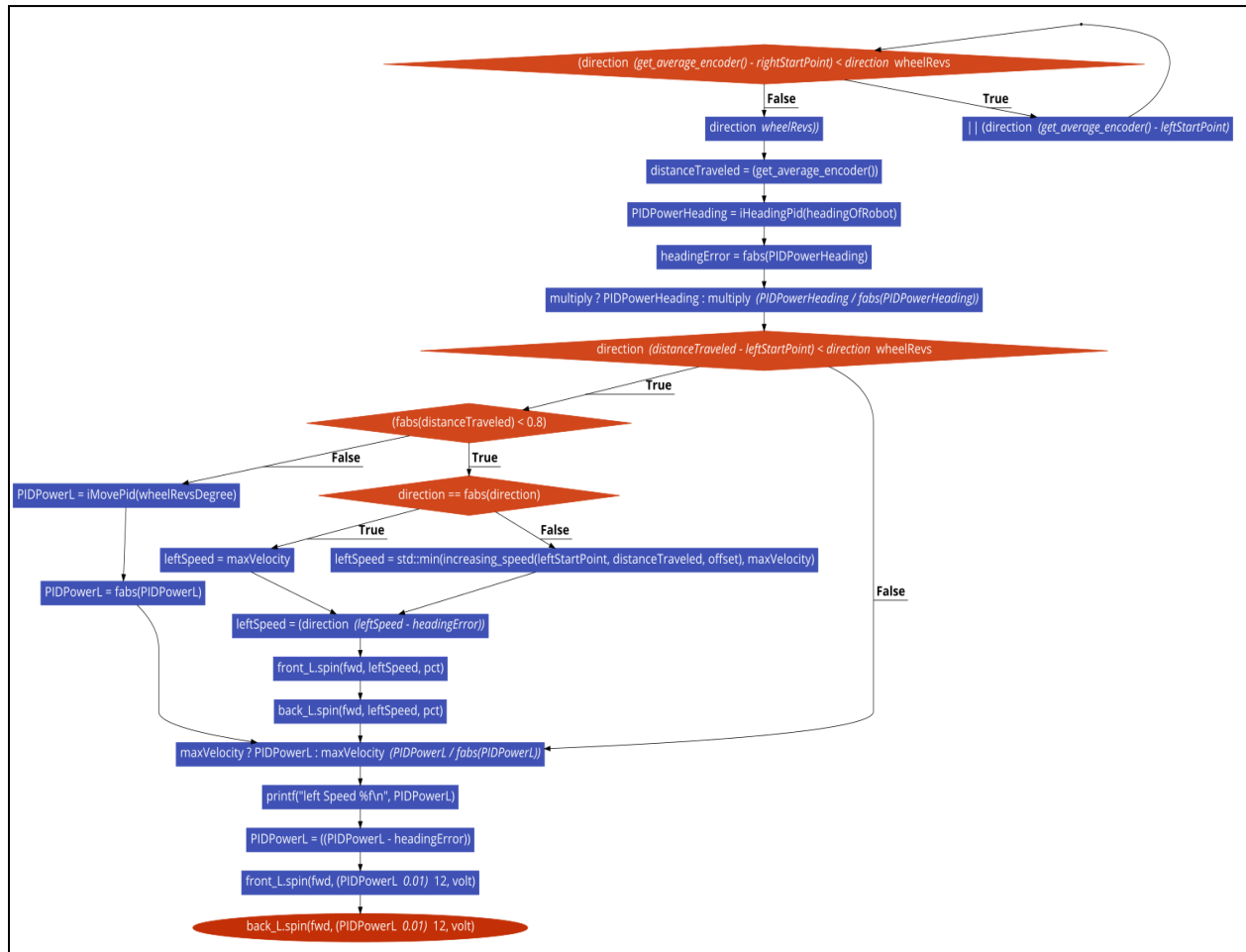
[Table of Contents](#)

also accounted for with heading control subtracting or adding a certain value to the speed of the motors. Next in the loop is where we used the PID for distance to have the robot decelerate in an exponential curve instead of a linear curve that we used to have. This allowed for faster acceleration and it being more consistent. Then the value would again be tweaked depending on the headingPID and if the robot was off-angle. Finally at the end of the loop is the idea to make the motors more consistent. We have a velocity PID to make sure the motors are actually at the value outputted to them and if not we correct the value of the motor to that certain value.

Right Drive:

```
if (direction * (distanceTraveled - rightStartPoint) < direction * wheelRevs) {
    if ((fabs(distanceTraveled) < 0.8)) {
        if(direction == fabs(direction)){
            rightSpeed = maxVelocity;
        }
        else{
            rightSpeed = std::min(increasing_speed(leftStartPoint, distanceTraveled,
offset), maxVelocity);
        }
        rightSpeed = (direction* (rightSpeed + headingError));
        front_R.spin(fwd, rightSpeed, pct);
        back_R.spin(fwd, rightSpeed, pct);
    } else {
        PIDPowerR = iMovePid( wheelRevsDegree);
        PIDPowerR = fabs(PIDPowerR) < maxVelocity ? PIDPowerR : maxVelocity *
(PIDPowerR / fabs(PIDPowerR));
        PIDPowerR = ((PIDPowerR + headingError));
        printf("right Speed %f\n", PIDPowerR);
        front_R.spin(fwd, (PIDPowerR * 0.01) * 12, volt);
        back_R.spin(fwd, (PIDPowerR * 0.01) * 12, volt);
    }
    printf("right Speed Real %f\n", rightDrive.velocity(pct));
    if ((direction * (wheelRevs - 0.1)) > fabs(distanceTraveled) &&
fabs(distanceTraveled) > 0.1) {
        rightSpeed += iSpeedPid(rightSpeed, false);
    }
    rightDrive.spin(fwd, rightSpeed, pct);
} else {
    brakeDrive();
}
distanceTraveledlast = distanceTraveled;
task::sleep(10);
```

Flow Chart of move forward code:



Testing

Testing was a major part in making this code come to life and it took many hours to debug the program and have it in the place where it is now. A lot of the testing can be credited to printf statements in the code and helping me realize what the code is calculating and how to fix it. There is one video of the robot doing short and immediate movements, and also a few videos on the movements that occur in autonomous.

<https://photos.app.goo.gl/eJyQkKmtLWmqoFEi7>

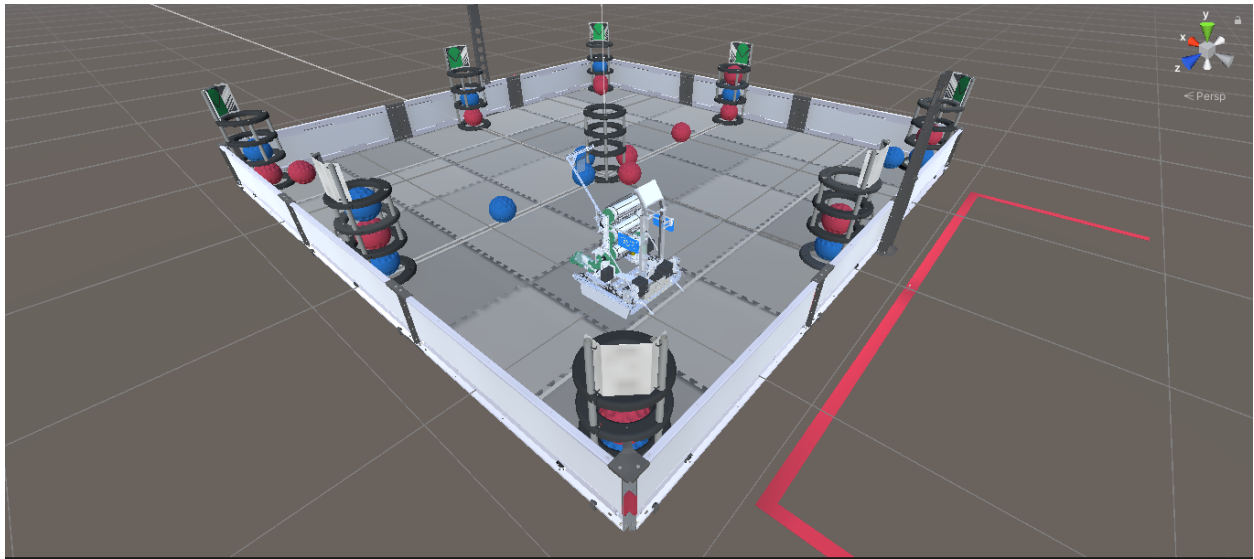
<https://photos.app.goo.gl/2J3CaCSzErHAWwmb7>

<https://photos.app.goo.gl/47rD99NMa7sK6e1b7>

Unity Simulation

Purpose

The main purpose of our Unity Simulation project was to create a physics-based environment that could be used to implement machine learning algorithms as well as artificial intelligence to let the robot create an autonomous path for both competition autonomous and skills autonomous path for the best possible score.



Method

The main method used was Unity's built-in physics engine to be able to calculate all ball physics, collision physics that may occur between the wall and the robot, the ball, and the robot, as well as the goal and the robot. We also used the built-in joint system, wheel physics system, and control system to make our simulation as accurate to our Mk II robot as possible. This would allow the algorithms to be training in an environment as close to real-life as possible.

A key factor to this included the ML-Agents which is provided by unity which is a machine learning platform compatible with unity to create AI within their software. This would be our main way to train our robot to be able to make decisions on its own through reward-based learning. Reward-based learning would award a point to the robot for every time it picked up a ball and then another point for every time it would score a ball with the main goal of the robot being to gain as many points as possible. Once training begins multiple robots will be used to try to find the most efficient method to earn these points. The result of this method would be the robot adapting to the field and finding the fastest path to be able to score balls which would create a path that we could then use to implement into our real-life robot.

The path created by the AI would then be recorded by checking the velocity that the robot gives the wheels in the physics system. Using the data, the velocity values can be fed to the real-life robot, which will then mimic the AI's path which will be the fastest path determined by its reward-based training process.

Along with the AI piece of the project, the robot will be fully drivable and functional along with a fully functional scoring system that can then be used as a driving practice simulator for teams that may not have access to their robots due to covid 19. The plan is to be able to push the Simulation out to the public for others to learn from and experience.

The process began with trying out basic projects to learn how unity worked at first with both Avi and Nikhil creating projects to test out the intaking system, drive system, and scoring as well. Once the basics were figured out then they were implemented into the main project. The process of building basic concepts to testing and pushing out the final AI will take 6 months as the project began in August.

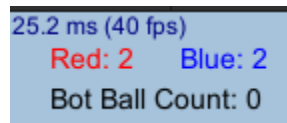
Code

- Display Text

```
public void Update()
{
    botBallCounter.text = "Bot Ball Count: " + botBall.ToString();
    scoreTextRed.text = "Red: " + scoreRed.ToString();
    scoreTextRed.color = Color.red;
    scoreTextBlue.text = "Blue: " + scoreBlue.ToString();
    scoreTextBlue.color = Color.blue;

    if(botBall == 2)
    {
        botBallCounter.color = Color.red;
        intakeCollider.enabled = false;
        //Debug.Log("collider is false");
    }
    else
    {
        botBallCounter.color = Color.black;
        intakeCollider.enabled = true;
    }
}
```

This section code relates to the creation of our display text which helps us know what is going on in the simulation at the time. The text includes our Frames Per Second Value, the amount of balls within the ball and the scoring system to check our goals. The FPS is used when doing graphical edits such as lighting to see what is increasing our performance thus increasing our FPS or what may be decreasing it. The scoring system will be used to see one's score while in use of the simulation.

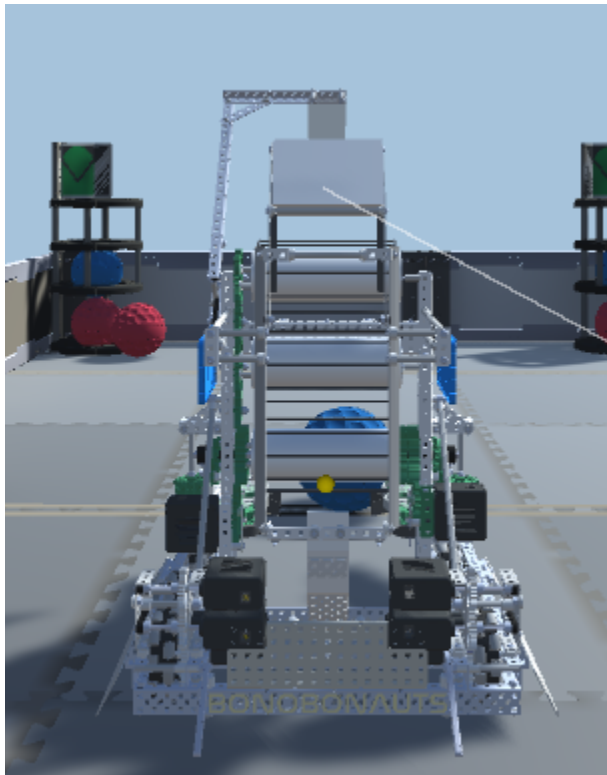


25.2 ms (40 fps)
Red: 2 Blue: 2
Bot Ball Count: 0

- Camera Follow

```
void LateUpdate()
{
    transform.position = player.transform.position - player.transform.forward
* cameraDistance;
    transform.LookAt(player.transform.position);
    transform.position = new Vector3(transform.position.x,
transform.position.y + yOffset, transform.position.z);
}
```

This section of code relates to our camera, which follows our robot to provide a third person view our robot when it is being controlled by the user. This was down by taking the position of the robot, offsetting the camera relative to the position, then aiming the camera at the robot at a set camera angle.



- FPS Calculations

```
void OnGUI()
{
    int w = Screen.width, h = Screen.height;

    GUIStyle style = new GUIStyle();

    Rect rect = new Rect(1, 0, w, h * 2 / 100);
    style.alignment = TextAnchor.UpperLeft;
    style.fontSize = h * 2 / 100;
    style.normal.textColor = new Color(0.0f, 0.0f, 0.5f, 1.0f);
    float msec = deltaTime * 1000.0f;
    float fps = 1.0f / deltaTime;
    string text = string.Format("{0:0.0} ms ({1:0.} fps)", msec, fps);
    GUI.Label(rect, text, style);
}
```

This code is how the FPS value is calculated, as it measures the millisecond value between each frame update and then divides one by the decimal millisecond value thus giving the frames per second within the simulation.

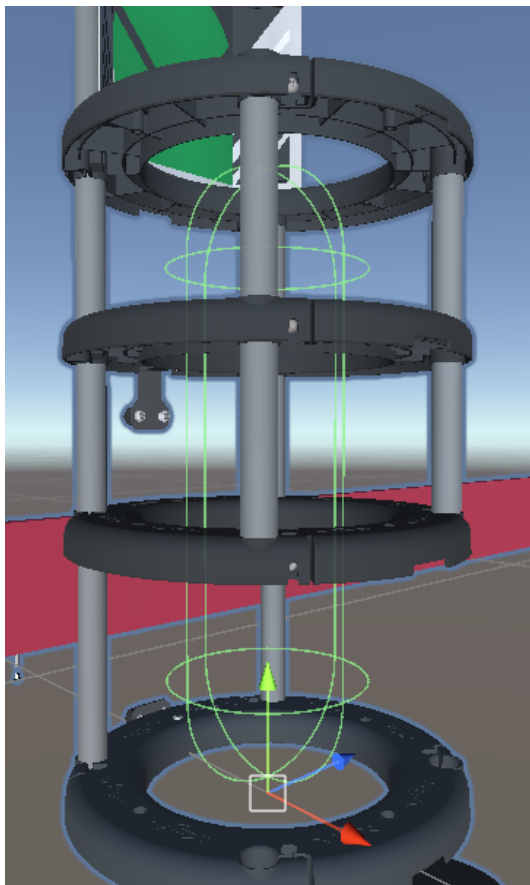
31.7 ms (32 fps)

-

- Scoring System, Goal Ball Detection

```
void Awake(){  
    ballCounter = GameObject.FindObjectOfType<Ball_Counter>();  
    bottom.SetActive(false);  
    bottom.SetActive(true);  
    top.SetActive(false);  
    top.SetActive(true);  
}
```

The way the scoring system detects balls in the goals is through using colliders set to a certain size in the middle of the goal. As shown with the green outlines in the image below.

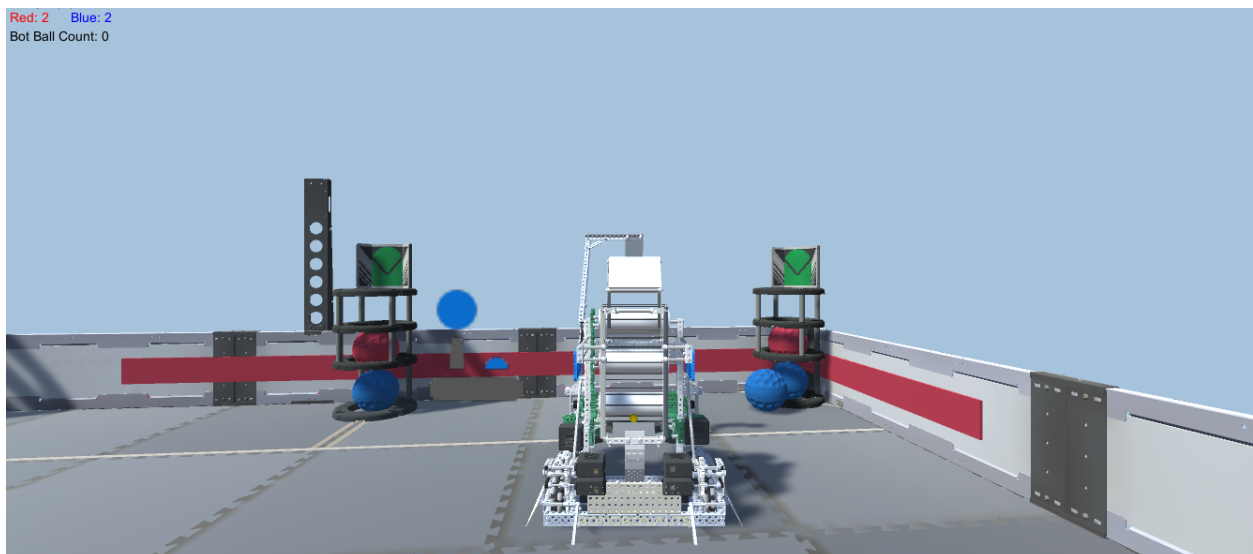



```

void Update() {
    Debug.Log("top ball" + goalList[0]);
    Debug.Log("ball size" + goalList.Count);
    if((int)goalList[0] == 1) {
        redBonus.SetActive(true);
        blueBonus.SetActive(false);
    }
    else if((int)goalList[0] == 2) {
        redBonus.SetActive(false);
        blueBonus.SetActive(true);
    }
    else if(goalList.Count == 0) {
        redBonus.SetActive(false);
        blueBonus.SetActive(false);
    }
}
}

```

Then, the goals check which ball out of the balls within the goal is the top ball. If the top ball is red then the goal is owned by red denoted by `redBonus.SetActive(true)` else it will be `redBonus.SetActive(false)`. The same is done if the ball that owns the goal is a blue ball. This method then checks which color has the bonus claimed by having a row of all red owned goals or all blue owned goals. As shown below there is no bonus as no set of three goals have been linked but the red score is shown as two red balls are in two goals and two blue balls are in two goals as well.



- Goal Ball Detection (Collision Enter)

```
void OnTriggerEnter(Collider collision) {  
    if(collision.CompareTag("redBall") == true) {  
        Debug.Log(collision.name);  
        goalList.Add(1);  
        ballCounter.redAdd();  
  
        if((int)goalList[0] == 1) {  
            redBonus.SetActive(true);  
            blueBonus.SetActive(false);  
        }  
    }  
    else if(collision.CompareTag("blueBall") == true) {  
        Debug.Log(collision.name);  
        goalList.Add(2);  
        ballCounter.blueAdd();  
  
        if((int)goalList[0] == 2) {  
            redBonus.SetActive(false);  
            blueBonus.SetActive(true);  
        }  
    }  
}
```

The OnTriggerEnter function call occurs when the ball first collides with the green outline also known as the hit box for the goal. When this occurs, the first collision that occurs starts the code which then checks the tag of the ball, which denotes whether the ball is a red ball or a blue ball. Based on this information the code then adds a number 1 to an ArrayList to show that there is a red ball within the goal. The ArrayList shows the amount of balls within a goal at any given moment with the number 1 representing red balls and the number 2 representing blue balls.

Ex: Goal 2: 1, 2, 2

This example shows one red ball and 2 blue balls where the goal is owned by red since it is in the most left position of the list.

Ex: Goal 2: 2, 1, 1

This example shows two red balls and one blue ball where the goal is owned by blue since it is in the most left position of the list.

The code is then able to use the data to determine where the row bonus occurs based on which goals are owned by either red or blue balls. The data also helps to determine how many of each type of ball are in every goal throughout the field to be able to determine an accurate score as if this was a real life competition.

- (Collision Exit)

```
void OnTriggerExit(Collider collision) {  
    if(collision.CompareTag("redBall") == true) {  
        //Debug.Log(collision.name);  
        goalList.Remove(1);  
        ballCounter.redMinus();  
  
        if(goalList.Count == 0) {  
            redBonus.SetActive(false);  
            blueBonus.SetActive(false);  
        }  
    }  
    else if(collision.CompareTag("blueBall") == true) {  
        //Debug.Log(collision.name);  
        goalList.Remove(2);  
        ballCounter.blueMinus();  
  
        if(goalList.Count == 0) {  
            redBonus.SetActive(false);  
            blueBonus.SetActive(false);  
        }  
    }  
}
```

When the function OnTriggerExit is called, the code is then able to determine when a ball has been taken by the robot. Since the code checks whether or not a blue ball or red ball has been removed it is able to adjust the score accordingly along with the home row bonus score as well. Along with decreasing score it also removes a spot for a ball in the respective goal list.

Ex: Goal 3 (before): 1, 1, 2 | Goal 3 (after): 1, 1

The example shows one blue ball being removed out of the list as it was the most bottom within the list.

-

- Goal Ball Detection functions

```
ArrayList ballList = new ArrayList(2);

public void ballListAdd(int id)
{
    ballList.Add(id);
    Debug.Log("array size" + ballList.Count);
}

public void ballListRemove()
{
    Debug.Log("array size "+ ballList.Count);
    ballList.RemoveAt(0);
}

public int topBall()
{
    return (int)ballList[0];
}

public void redAdd() {
    scoreRed++;
}

public void redMinus() {
    scoreRed--;
}

public void blueAdd() {
    scoreBlue++;
}

public void blueMinus() {
    scoreBlue--;
}
```

Above is shown the code for the goals counting system as it calls different functions in another file to make sure that the code remains clean and understandable. The names of the function have been named so that they are understandable by anyone who is trying to use this simulation

[Table of Contents](#)

for their own personal uses as well to make development easier as the names are easier to remember. The names are representative of what the function will do for the over all simulation depending on where the code is called (primarily called in the goal ball detection script)

- Robot User Control

```
void FixedUpdate()
{
    if (this.transform.localPosition.y < 0)
    {
        // If the Agent fell, zero its momentum
        this.rb.angularVelocity = Vector3.zero;
        this.rb.velocity = Vector3.zero;
        this.transform.localPosition = new Vector3(0, 1.5f, 0);
    }

    if (Input.GetKey("w"))
    {
        rb.velocity = new Vector3(0f, 0f, speed);
        transform.localEulerAngles = new Vector3(0, 0, 0);
    }
    else if (Input.GetKey("a"))
    {
        rb.velocity = new Vector3(-speed, 0f, 0f);
        transform.localEulerAngles = new Vector3(0, -90, 0);
    }
    else if (Input.GetKey("s"))
    {
        rb.velocity = new Vector3(0f, 0f, -speed);
        transform.localEulerAngles = new Vector3(0, 180, 0);
    }
    else if (Input.GetKey("d"))
    {
        rb.velocity = new Vector3(speed, 0f, 0f);
        transform.localEulerAngles = new Vector3(0, 90, 0);
    }
}
```

Using Unity's built in control system, the simulation is able to read both keyboard values as well as a gamepad's values such as an Xbox controller, PS4 controller, or any other generic gamepad.

Using this we can use the keyboard for testing purposes and the gamepads to mimic the use of an actual controller. Above is shown the keyboard code for determining how the robot will move.

```
void Awake()
{
    controls = new BotController();

    controls.Controller.ForwardBackward.performed += ctx => moveL =
ctx.ReadValue<Vector2>();
    controls.Controller.ForwardBackward.canceled += ctx => moveL =
Vector2.zero;

    controls.Controller.LeftRight.performed += ctx => moveR =
ctx.ReadValue<Vector2>();
    controls.Controller.LeftRight.canceled += ctx => moveR = Vector2.zero;
}

public void FixedUpdate()
{
    float leftMotor;
    float rightMotor;

    leftMotor = maxMotorTorque * (moveL.y + moveR.x);
    rightMotor = maxMotorTorque * (moveL.y - moveR.x);
}
```

The code above shows what is needed for a gamepad to be compatible with our simulation in order to take joystick input as well as trigger input for the simulation. By reading the values of the joystick which ranges from 0 to 1 with decimal values in between, the user is able to get an immense amount of control over where the robot will be moving within the simulation. The ForwardBackward call refers to the straight and backward movement of the bot. The LeftRight call refers to which direction the bot will be rotating depending on the joystick values the user is inputting.

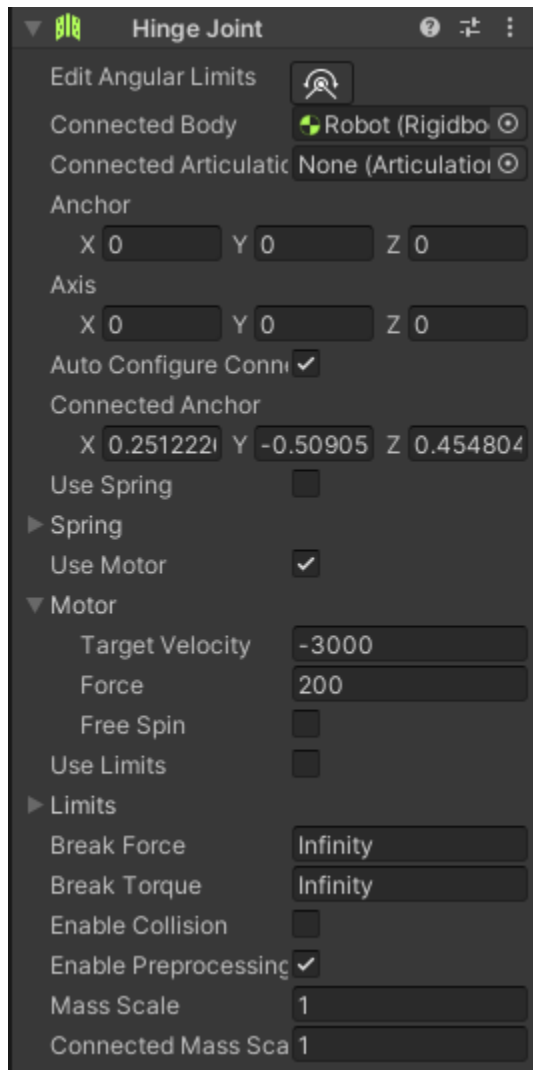
-

- Robot Axle Code

```
void Update()
{
    leftAxleInfo.frontWheelRotate.Rotate(0, 0, -leftAxleInfo.frontWheel.rpm /
60 * 360 * Time.deltaTime);
    leftAxleInfo.backWheelRotate.Rotate(0, 0, -leftAxleInfo.backWheel.rpm /
60 * 360 * Time.deltaTime);
    leftAxleInfo.middleWheelRotate.Rotate(0, 0, -leftAxleInfo.middleWheel.rpm
/ 60 * 360 * Time.deltaTime);
    rightAxleInfo.frontWheelRotate.Rotate(0, 0, -rightAxleInfo.frontWheel.rpm
/ 60 * 360 * Time.deltaTime);
    rightAxleInfo.backWheelRotate.Rotate(0, 0, -rightAxleInfo.backWheel.rpm /
60 * 360 * Time.deltaTime);
    rightAxleInfo.middleWheelRotate.Rotate(0, 0,
-rightAxleInfo.middleWheel.rpm / 60 * 360 * Time.deltaTime);
    transform.localEulerAngles = new Vector3(0, transform.localEulerAngles.y,
0);
}
```

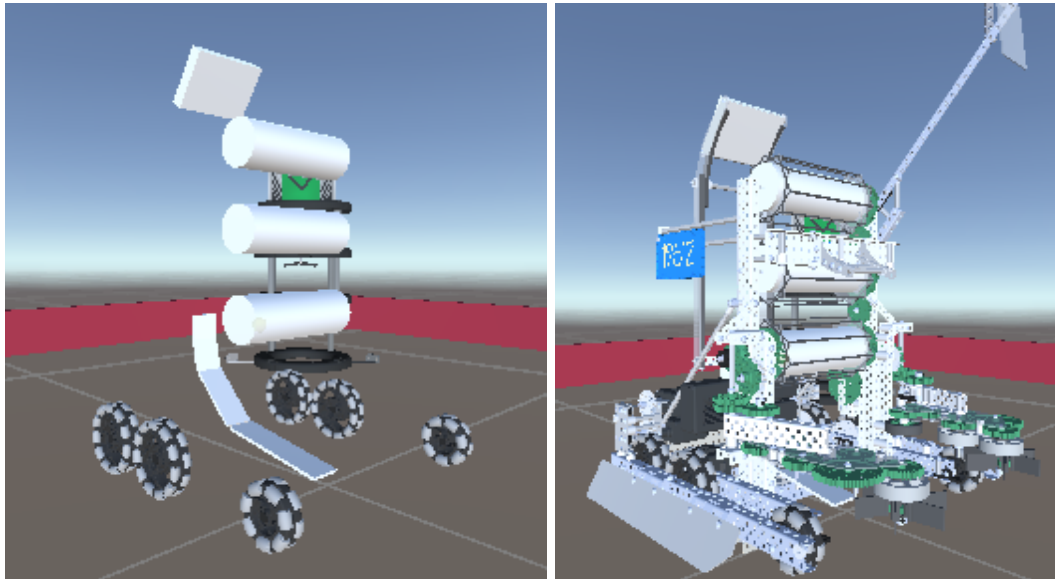
The method through which the robot moves is by taking axles linked to wheels and setting an rpm to the wheels depending on what the inputs are from the user. This is how the bot moves and will be moved throughout the simulation.

- Robot Functionality



The intakes work through using a Hinge Joint which allows the intakes to spin given a set Target Velocity and force. These forces are tuned to create similar intaking times to what real life intaking times are like. This required recording numerous videos of our in person robot and averaging the time over a few trials and then fine tuning the values to get an intaking experience close to that of real life.

- Robot Graphics



The way the graphics are handled is that the main functional parts of the robot, anything that moves or collides is a basic 3-D object to save on performance when rendering and detecting collisions as it is reduced in comparison to the simulation rendering and detecting every screw or nut thus causing performance issues. Then the main robot CAD is overlaid as an image that is simply rendered out using Unity's lighting system. This makes sure that all physics simulations are done with the basic 3-D parts and anything graphics is done for pure aesthetic.