Autonomous Programming

2016-2017 FTC Challenge Workshops
VCU School of Engineering
September 24, 2016
Presented by:
Team 8297 Geared UP!
## Table of Contents

<table>
<thead>
<tr>
<th>Section #</th>
<th>Topic</th>
<th>Page #s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Autonomous game in VELOCITY VORTEX</td>
<td>3, 4, 5</td>
</tr>
<tr>
<td>2</td>
<td>Why use Android Studio?</td>
<td>6, 7</td>
</tr>
<tr>
<td>3</td>
<td>High Level Strategies for Autonomous Programs</td>
<td>8 - 11</td>
</tr>
<tr>
<td>4</td>
<td>Starting Menu for Autonomous Programs</td>
<td>12, 13</td>
</tr>
<tr>
<td>5</td>
<td>Linear v/s Event Driven Programming</td>
<td>14 - 18</td>
</tr>
<tr>
<td>6</td>
<td>Setting up the Overall Class Structure</td>
<td>19 - 21</td>
</tr>
<tr>
<td>7</td>
<td>Using Sensors to Improve Navigation</td>
<td>22, 23</td>
</tr>
<tr>
<td>8</td>
<td>PID (Proportional Integral Derivative)</td>
<td>24 - 26</td>
</tr>
<tr>
<td>9</td>
<td>Tips &amp; Tricks</td>
<td>27 - 30</td>
</tr>
<tr>
<td>10</td>
<td>Video of a successful Autonomous Program run</td>
<td>32</td>
</tr>
</tbody>
</table>
Autonomous Programming

Section #1
Autonomous game in VELOCITY VORTEX
Autonomous game in VELOCITY VORTEX

The Match starts with a 30 second Autonomous Period where Robots are operated via “Pre programmed instructions only” and teams are NOT allowed to control Robot behavior with the Driver Station or any other actions during the Autonomous Period.

<table>
<thead>
<tr>
<th>Scoring Achievement</th>
<th>Autonomous Points</th>
<th>Driver-Controlled Points</th>
<th>End Game Points**</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beacons Triggered</td>
<td>30 / Claim + Bonus Particle awarded (up to 2)</td>
<td>10 / Claim</td>
<td></td>
<td>1.5.2.1 and 1.5.4.2</td>
</tr>
<tr>
<td>- State of Claim is all lit at end of period</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cap Ball</td>
<td>5</td>
<td>-</td>
<td>10</td>
<td>1.5.2.2 and 1.5.4.1a</td>
</tr>
<tr>
<td>- On Playing Field</td>
<td></td>
<td></td>
<td></td>
<td>1.5.4.1b</td>
</tr>
<tr>
<td>- Off Playing Field below crossbar</td>
<td></td>
<td></td>
<td></td>
<td>1.5.4.2c</td>
</tr>
<tr>
<td>- Raised above crossbar</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Capped Center Vortex</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Particle</td>
<td>15*</td>
<td>5*</td>
<td></td>
<td>1.5.2.3 and 1.5.3.1</td>
</tr>
<tr>
<td>- Scored in Center Vortex</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Scored in Corner Vortex</td>
<td>5*</td>
<td>1*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robot Parked</td>
<td>5</td>
<td>-</td>
<td></td>
<td>1.5.2.4a</td>
</tr>
<tr>
<td>- On Center Vortex Base</td>
<td></td>
<td></td>
<td></td>
<td>1.5.2.4b</td>
</tr>
<tr>
<td>- Completely On Center Vortex Base</td>
<td>10</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- On Corner Vortex Ramp</td>
<td>5</td>
<td>-</td>
<td></td>
<td>1.5.2.4c</td>
</tr>
<tr>
<td>- Completely On Corner Vortex Ramp</td>
<td></td>
<td></td>
<td></td>
<td>1.5.2.4d</td>
</tr>
</tbody>
</table>
Autonomous in VELOCITY VORTEX (contd.)

Depending on the alliance, the ramp and beacons for each team will be either on the left (RED) or right (Blue) side of the starting position for each robot. Note the field is symmetrical along the diagonal, not along the sides!
Autonomous Programming

Section #2
Why use Android Studio?
Why use Android Studio?

<table>
<thead>
<tr>
<th>PROS</th>
<th>CONS</th>
</tr>
</thead>
</table>
| • Android Studio is applicable outside of FTC, as this IDE is used in the tech industry as well as schools/colleges.  
• Android Studio helps you learn the fundamentals of Java unlike App Inventor which is more “drag and drop”.  
• Android Studio provides the ability to customize and make programs that are specific to your robot at the deepest level (App Inventor tends to be restrictive in what you can do given its construct). | • It is difficult to use Android Studio without a basic understanding of Java and class structure/packages in general. Hence, it might not always the best option for rookie teams or those transitioning from FLL.  
• Unlike App Inventor, Android Studio does not have a built-in menu. |
Autonomous Programming

Section #3
High Level Strategies for Autonomous Programming
High Level Strategies for Autonomous

• Use utility methods for quicker and easier programming (non-programmer team members will also have an easier time learning the basics)

• Use an overall class structure (One main “autonomous” class with all methods. All individual programs will extend “autonomous”). See code snippet later on.

• Get the simplest options completed first (try to solve the simplest autonomous challenge first before attempting to add multiple motors for complicated tasks)

• Use as much sensor input as possible (example - we used the gyro sensor to detect if the robot had climbed on top a block/ball)
High Level Strategies for Autonomous (contd.)

• Create a “Menu Driven System” that lets the driver/on field coach select the program to run dynamically (our team used four touch sensors for selection, since the field was NOT a mirror image. Two more sensors were used for adding a delay, and the other two were used for selecting red/blue alliance)

• In VELOCITY VORTEX, the Red Alliance robot turns left to the ramp or push the beacons and the Blue Alliance robot turns right for performing the same action. By using a menu, a team can literally have one program and select the direction of turns based the alliance choice (instead of having multiple versions of the same program to run)
High Level Strategies for Autonomous (contd.)

Our recommendation is to extend the Linear Op mode for Autonomous programs. Why?

- Event Driven mode unnecessarily overcomplicates things for autonomous programs

- In most cases, the autonomous program will perform only one action at a time i.e. drive and then shoot a ball after stopping (instead of shooting a ball while driving) etc. Linear Op Mode is best suited for such activities and can be coded without worrying about setting up a state machine.
Autonomous Programming

Section #4
Starting Menu for Autonomous Programs
Starting Menu for Autonomous Programs

In order to effectively organize autonomous programs, it is preferred that a menu is designed to let the driver / on field coach select the appropriate program.

```java
if ((BLUEtouchSensor.isPressed() && REDtouchSensor.isPressed())) {
    continue;
} else if (BLUEtouchSensor.isPressed()) {
    telemetry.addData("==BLUE==BLUE==BLUE==BLUE==BLUE==", 0);
    return allianceColor.BLUE;
} else if (REDtouchSensor.isPressed()) {
    telemetry.addData("==RED==RED==RED==RED==RED==RED==", 0);
    return allianceColor.RED;
}
```

- This source code is part of a program that uses touch sensors, each one representing the alliance program, for menus.

- A series of “if statements” are created to identify the alliance to use by returning the alliance color based on the sensor pressed by the user. If both sensors are pressed simultaneously, the program ignores the command and returns to wait for the next sensor press event.
Autonomous Programming

Section #5
Linear v/s Event Drive Programming
Linear v/s Event Driven Programming

- With the introduction of the new Android based control system in FTC, there are two different programming options to choose from:
  - Event Driven (class - OpMode)
  - Linear Mode (class - LinearOpMode)

- Event Driven - the code that runs the robot is inside the loop method. As the robot runs, the loop method will be called repeatedly, where the program can constantly read input from gamepads or sensors. This is perfectly suited for TeleOp as it would make it easier to adjust to changes from the gamepad or on the field.

- Linear Op Mode - Unlike with the Event Driven mode, the main method in is executed just once. So, all the logic for a specific autonomous program should be included in this method directly or in methods invoked from inside this method. This is useful for autonomous, as it executes a set of instructions in sequence one at a time and then finishes.
### Linear v/s Event Driven Programming (Contd.)

<table>
<thead>
<tr>
<th>Linear Op Mode</th>
<th>Event Driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>State machine logic is actually implemented behind the scenes and hidden from the programmer</td>
<td>Requires understanding and use of a state machine and tracking of state transitions</td>
</tr>
<tr>
<td>All user code goes inside the runOpMode method and will be executed once in a sequence</td>
<td>All user code goes inside the runOpMode method and will be executed millions of times at processor speed</td>
</tr>
<tr>
<td>Simpler and easier to understand (great starting point for beginners!)</td>
<td>Requires the understanding of the abstract concept of a state machine and associated conditional logic</td>
</tr>
<tr>
<td>Supports only one task/action being performed by the robot at a given instant</td>
<td>Can be used to program parallel tasks for the robot (not a typical use case in autonomous)</td>
</tr>
</tbody>
</table>
Linear Op Mode - Example

- In Linear Op mode, the runOpMode method is used to configure the robot parts (motors, servos, etc.), gamepad controls, and other robot definitions. In this code, the motors and servos are being set a default name, and left motor’s direction is set as reverse. Since all this code gets executed sequentially and only once, it can be used to manage the tasks in a flow from start to finish without worrying about tracking the robot status (the code execution point gives the status of the robot at any instant!).

```java
package org.firstinspires.ftc.teamcode;
import com.qualcomm.robotcore.eventloop.opmode.Autonomous;

/**
 * Created by goyindb on 9/10/16.
 */
@Autonomous(name = "ShortBuildAuto", group = "ShortBuildAuto")
public class ShortBuildAuto extends AutonomousVelvort {
    @Override
    public void runOpMode() throws InterruptedException {
        initializeHardware();
        setAttributes(1440, 3, (float)15.00);
        waitForStart();
        spinTurn(90, (float)0.5, moveDirection.RIGHT);
        sleep(1000);
        spinTurn(90, (float)0.5, moveDirection.LEFT);
        sleep(1000);
        spinTurn(90, (float)0.5, moveDirection.LEFT);
        sleep(1000);
        spinTurn(90, (float)0.5, moveDirection.RIGHT);
        sleep(1000);
    }
    exitOpMode();
}
```
Event Driven Mode - Example

Using Event Driven mode, the loop method is used to configure the gamepad controls in the example below. In this code from a TeleOp program, the Y button’s function is defined to reverse the spool on a robot. This loop () method is invoked repeatedly (at processor speed) to pick up any events (button press, joystick motion, bumper push, etc.) that get triggered by user action.

```java
public class DemoTeleop extends OpMode {

    DcMotor motorRight;
    DcMotor motorLeft;

    @Override
    public void init() {
        motorRight = hardwareMap.dcMotor.get("leftmotor");
        motorLeft = hardwareMap.dcMotor.get("rightmotor");
        motorLeft.setDirection(DcMotor.Direction.REVERSE);
    }

    @Override
    public void loop() {

        float throttleLeft = -gamepad1.left_stick_y; //Give the robot's left motor throttle
        float directionLeft = gamepad1.left_stick_x;

        float throttleRight = -gamepad1.right_stick_y; //Give the robot's right motor throttle
        float directionRight = gamepad1.right_stick_x;
    }
}
```
Autonomous Programming

Section #6
Setting up the Overall Class Structure
Setting up the overall Class Structure

This overall class WILL NOT contain your actual autonomous code, but will include utility methods, commonly used variables, etc. AutonomousVelVort is the class we developed for this purpose.

```
public abstract class AutonomousVelVort extends LinearOpMode {

    // Example methods used last season (Res-Q) and probably will use this season.
    // Important methods: moveStraight and spinTurn used for navigation.
    // Team plans to use a Proportional Integral Derivative (PID) for straight moves this season,
    // but will go through a simplified moveStraight that calculates the required distance and sets power to the motors.

    public void initializeGyro(GyroSensor gyro) throws InterruptedException {
    ...
    }

    public void initializeHardware() throws InterruptedException {
    ...
    }

    public void setAttributes(int ticksPerRotation, float wheelDia, float wheelBase) {
    ...
    }

    public void stopDriveMotors() {
    ...
    }

    public void moveStraight (double mPower, double distance, boolean direction) throws InterruptedException{
    ...
    }

    public void spinTurn(float degrees, float power, moveDirection turn) throws InterruptedException {
    ...
    }

    public void moveByTime(double power, moveDirection dir, long timeMill, boolean abortWhenTilted) throws InterruptedException {
    ...
    }

    // Additional methods...
```

Shown below are example methods that we used last season (Res-Q) and probably will use this season. The important methods were moveStraight and spinTurn used for navigation. Our team plans to use a Proportional Integral Derivative (PID) for straight moves this season, but we will go through a simplified moveStraight that calculates the required distance and sets power to the motors in the next few slides.
On the left, is an example autonomous program that turns the robot to the right by 90 degrees, left 90 degrees, left 90 degrees, and finally right 90 degrees. It invokes three methods from the “AutonomousVelVort” class:

- initializeHardware (hardware mapping)
- setAttributes (wheel dia, wheel base, etc.)
- spinTurn (turns using a given degree number, power, and direction).

The key takeaway from this - instead of extending LinearOpMode directly, this class extends our own “autonomous” class - “AutonomousVelVort” (which extends the FTC class LinearOpMode). This way, it inherits whatever was in the LinearOpMode class along whatever we added to it as part of the class “AutonomousVelVort”.

```java
public class ShortBuildAuto extends AutonomousVelVort {
    @Override
    public void runOpMode() throws InterruptedException {
        initializeHardware();
        setAttributes(1440, 3, (float)13.00);
        waitForStart();
        spinTurn(90, (float)0.5, moveDirection.RIGHT);
        sleep(1000);
        spinTurn(90, (float)0.5, moveDirection.LEFT);
        sleep(1000);
        spinTurn(90, (float)0.5, moveDirection.LEFT);
        sleep(1000);
        spinTurn(90, (float)0.5, moveDirection.RIGHT);
        sleep(1000);
    }
}
```
Autonomous Programming

Section #7
Using Sensors to Improve Navigation
Using Sensors to improve Navigation

- Sensors will be important in VELOCITY VORTEX. Unlike the past few seasons, scoring high in TeleOp depends on how much and what a team can accomplish during the autonomous period.
- As much as possible, utilize at least 2 ways to measure something (for instance, use a color sensor to hit a line as well as encoders to set a max value in case the sensor misses the line).
- **Example Usage (** used in the past; + plan to use this season):**
  - Gyro Sensor [z axis if the robot is on top of debris and tilted at an angle] *
  - Gyro Sensor [for turning] *
  - Color Sensor [for sensing beacon] +
  - Touch Sensor [finding the wall or selecting programs as mentioned earlier] *
  - Motor Encoder [probably the most important sensor on your robot] *
  - Ultrasonic sensor [for finding objects around the field] *
  - Limit switch [could be used for an arm] +
  - MR Rangefinder [Same use as ultrasonic sensor]
  - Optical Distance Sensor [Part of the range finder, for very close objects]
  - Compass Sensor [to identify the magnetic heading]
  - Android Phone Camera [along with Vuforia software for image recognition] +
Autonomous Programming

Section #8

PID
(Proportional Integral Derivative)
PID (Proportional Integral Derivative)

• A Proportional Integral Derivative is an algorithm commonly used in the industry to improve automation control that utilizes error and feedback to improve accuracy. Its main purpose is to minimize error with the assistance of feedback coming from sensors. One common application is a home temperature control thermostat unit.

• Why PID - For the purposes of our Autonomous program, a PID algorithm was implemented to manage the difference between the rotations of our robot’s DC drive motors so that drift over a long distance (lateral movement to the right or left) could be minimized. As a result, the robot can now drive straight with minimal drift over long distances! (Note – implementing PID algorithm for a track based robot is complicated!)
PID (Proportional Integral Derivative)

SP = Set Point (Desired target Value/input); P = Proportional Gain, I = Integral Gain, D = Derivative Gain
Autonomous Programming

Section #9
Tips & Tricks
Tips & Tricks

• CONSISTENT SET UP - When testing, make sure to be as consistent as possible in setup and other factors you can control.

• KEEP LAPTOP HANDY - Keep the computer that you are using for downloading changes close to the robot so it is less of a hassle to unplug/plug the robot controller.

• DO NOT OVER-TWEAK – Don’t tweak distances and angles too much. Overcorrection can ruin the accuracy of your program.

• VERSION CONTROL SOURCE CODE - Save a different version of your program, or take a screenshot, before you change ANYTHING. If you, by mistake, do something that messes up your program, then you can revert back to an older version that still works.

• MONITOR BATTERY VOLTAGE - If your program starts behaving erratically in the middle of a test session, the battery is probably what is causing it. Make sure the driver station reads at least 12.6 volts for accurate results.
Tips & Tricks (contd.)

- **USE TELEMETRY** - (crude) debugging statements can be added by invoking the methods from “telemetry” class shown below. Telemetry is especially helpful when debugging the robot and the messages show up on the driver station phone (be aware that they scroll!).

  ```java
  telemetry.addData("GYRO Initialization COMPLETE", 0);
  telemetry.update();
  ```

- **PAUSE ROBOT BETWEEN MOVES** - Always pause the robot for a short time (0.5 – 1.5 secs) to bring the robot to a complete stop and account for its inertia. This can dramatically increase the overall accuracy of your robot.

- **CONSITENT STARTING POINT** - Make sure to set up the robot up at the same place every time to reduce user error and increase consistency. Use features such as tile edges or lines on the field for reference since they remain consistent across all fields (home and competition).
The 5 Stages of Debugging

At some point in each of our lives, we must face errors in our code. Debugging is a natural healing process to help us through these times. It is important to recognize these common stages and realize that debugging will eventually come to an end.

**Denial**

This stage is often characterized by such phrases as "What? That's impossible," or "I know this is right." A strong sign of denial is recompiling without changing any code, "just in case."

**Bargaining/Self-Blame**

Several programming errors are uncovered and the programmer feels stupid and guilty for having made them. Bargaining is common: "If I fix this, will you please compile?" Also, "I only have 14 errors to go!"

**Anger**

Cryptic error messages send the programmer into a rage. This stage is accompanied by an hours-long and profanity-filled diatribe about the limitations of the language directed at whomever will listen.

**Depression**

Following the outburst, the programmer becomes aware that hours have gone by unproductively and there is still no solution in sight. The programmer becomes listless. Posture often deteriorates.

**Acceptance**

The programmer finally accepts the situation, declares the bug a "feature", and goes to play some Quake.
Autonomous Programming

Section #10
Video of a successful Autonomous Program run
A successful Autonomous Program Run!
Questions?
Contact Info for Team Geared UP!

Email:
tmgearedup@gmail.com

Website:
gearedup.ashburnrobotics.com

Twitter:
@TeamGearedUP