TODO

Table of Contents

Entries

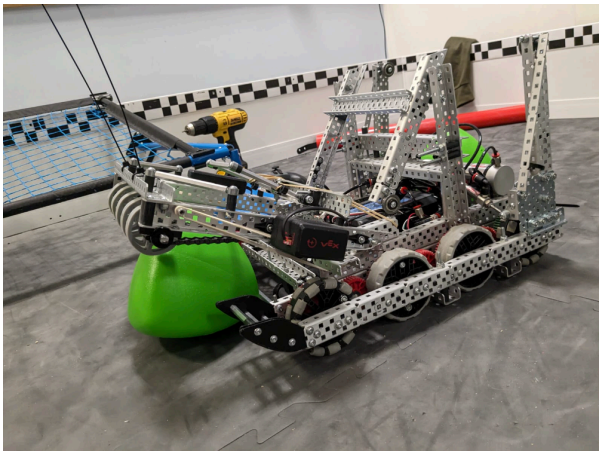
Appendix

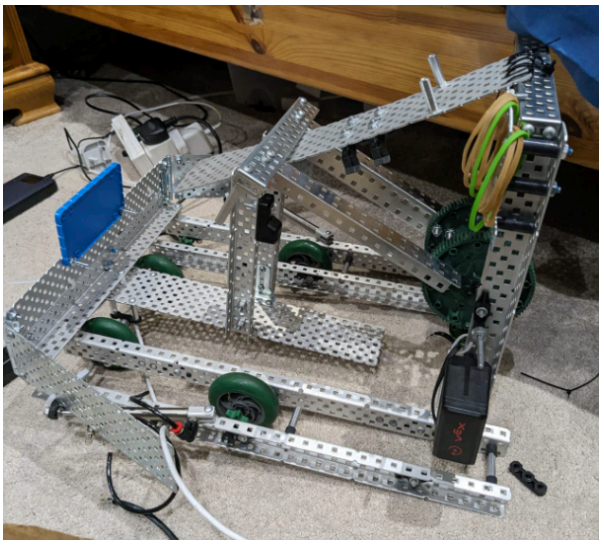
Team Introduction

Who are the Snowflakes?

We are the St Chris Snowflakes; we started out as a VRC team in September 2023, in our first season as a team in 'Over Under'. We set out to do our very best and we quickly found that we all loved the challenge of VEX and wanted to excel as far as we could; after a struggle, we qualified for UK nationals and siezed the oppurtunity to become the best we could. Our hard work payed off, and we went home with design award and a spot in the VEX World Championships; where we went in April and gained key experience to start this season.

Off of the back of Over Under we have acquired some very important skills necessary for competitive representation on both a national and global scale such as good building and coding practices which we hope to utilise in order to become even more competitive in all aspects of the V5RC competition where we weren't last year. We are also moving to a smaller team size as some of our members are moving on. This will mean we may have to work harder but also means a larger range of experience in other areas.





Team Introduction

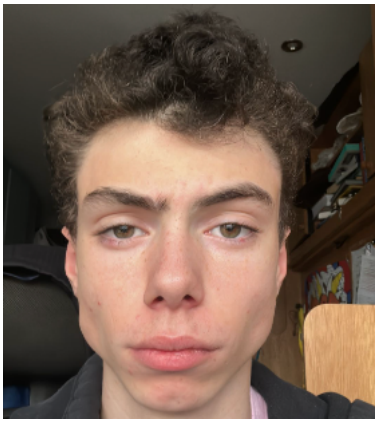
The members

Jonah Fitchew
- Co-Head Builder
- Driveteam

Hello I am Jonah Fitchew and my role within the team is the physical construction of the robot, documentation and assisting the driver during matches.

Aubert Seysess
- Co-Head Builder

Hello I am Aubert Seysses and my role within the team is to help design and virtually CAD the robot, also, I aid with the physical construction of the robot.



Daniel Dew
- Head Programmer
- Driver
- Driveteam

My role within the team primarily resides in the programming of the robot's code including autonomous and driver control. I also aid with the design and CAD phases of building. I am also the primary driver.

Daniel DaSilva
- Programmer

Hi I'm Daniel da Silva and my role within the team is to support the development of the robot's software and to help organise it into the logbook.

Team Introduction

Thomas Robb

- Head Tactics
- Driveteam

My role within the team is to brainstorm tactics and communicate with the other members to ensure that designs and tactics align. During competitions I am also responsible for taking note of performances, both our's and other team's; to help find possible alliances.

How To Use This Notebook

About this Notebook

TL;DR

For this season, we decided to deviate from the standard process for making engineering notebooks. We decided that, with the loss of our main logbooker we would have to share the notebooking duties; this meant that formatting could become inconsistent and we immediatly found that it took too long to format everything to the desired (exceptional) standard. Therefore, to cut down on time and improve the notebook's readability and functionality, we decided to adopt the *Notebookinator* template, which is an extension of the *typst* markup language.

Why Typst?

Several ways of creating notebooks for VEX exist, with most adopting visual editors such as google slides or hand writing their notebooks.
When deciding what we wanted to use for this season, we quickly ruled out hand writing the notebooks as mistakes could take valuable time to correct; neatness and clarity is often sacrificed; and the need for online collaboration is great. We previously used google slides with good results, however the formatting (e.g. colour coding, table of contents etc.) takes a significant ammount of time to maintain and can be very difficult to keep consistent when we all share equal role in notebook creation (as opposed to 1 person overseeing all notebook formatting).
We then landed on the possibility of using a markup language; and with the lack of flexibility from LaTex, Typst seemed like the best option. We had also noted a few teams success with using Typst, especially when using Notebookinator alongside it - for example team 53E (also the creators of Notebookinator) had a great Over Under notebook[1] using Typst.

Features

- Uniform formatting
- Notebookinator template
  - Easy cohesion with engineering design process
  - Built in components i.e. pros/cons tables
- Code blocks
- Built in table of contents
- Fully Digital
  - Neatness
  - Modern tooling
  - Easy submission
  - Cohesion with version control

[1] *Link to notebook*

The Snowflakes' Enginnering Ethos

Our Engineering Ethos

At its core, VEX Robotics is nothing but an engineering problem. It provides a goal, and the materials to get there. We believe that the key to success in Robotics intrinsicly lies in how you approach each problem; with open-mindness and the willingness to learn but most importantly the deterination to find the best solution possible.

> Engineering: The art of organizing and directing men, and of controlling the forces and materials of nature for the benefit of the human race.
>
> — Henry Gordon Stott

Our Engineering Design Process

For every new problem, we try to stick to an engineering method (similar to a scientific method) where different phases are used to maintain organisation. The process applies to all forms of design, including programming and sometimes even tactics.

Phases of Design

For each of the phases in our EDP, a corresponding icon is provided, these are used throughout the notebook to label a phase.

Identify Problem[1]
Each solution starts with a problem, this ranges drastically – for example, from 'We need a drivetrain' to 'this mechanism causes instability'. Problems can be raised by all members and, regardless of severity, it is something that must at least be addressed.

Brainstorm Solutions
Once the problem has been properly analysed, with root causes found, the team can move on to brainstorming solutions. Here, every team member can put forward possible solutions or fixes to the problem at hand, this is often accompanied with rough concept drawings. Additionally, finding use cases where each possible solution has been used effectively can be key to display a solution's viability.

Decide Solution
Once all possible solution have been brought to the table, one possible solution is picked to move to the next phase; to ensure that the decision is definitely the best one available, additional proceses can be used to decide the best (i.e. decision matrices). Ideally, all members offer their thoughts on the solutions.

Implement Solution
With one solution in mind, the solution can be expanded, now taking into account the smaller details while creating a plan of action for the designing and implementation of the solution. The solution is then designed and built – either physically or as a program.
Note that both the build and program icons are used during this phase.

Test Solution
Once the solution has been implemented onto the robot, we can begin testing the solution to find out how effective it is. This is a key phase as it shows us how the solution up to different scenarios.
Depending on its effectiveness, the results of a test may prompt us to move back into the implement phase, as changes sometimes have to be made. This creates a feedback loop that iteratively improves the solution until it meets our desired standards.

For our robot, we decompose the larger problem into a set of smaller, approachable problems. From there, each and every problem is approached using this EDP; this allows us to stay organised and avoid decision paralysis.

[1]Sometimes, if the problem is obvious (i.e. the need for a drivetrain), this phase is skipped due to mutual understanding.

? Example Entry 2024/01/01

Here's some content in this entry.

Here's an example of how you'd create a pro-con table:

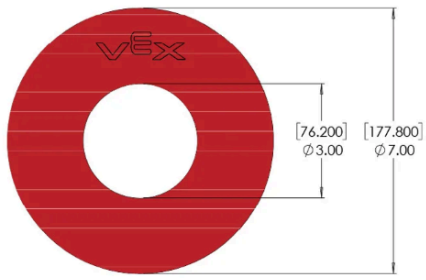
Pros	Cons
• pro number 1 • pro number 2 • pro number 3	• con number 1 • con number 2 • con number 3

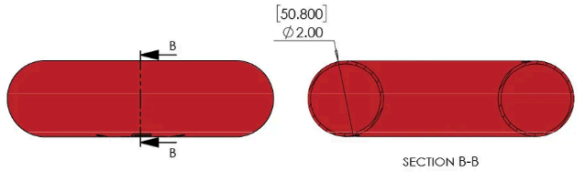
Now we'll generate 50 words of filler text!

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut.

The Game

Scoring & Game Objectives



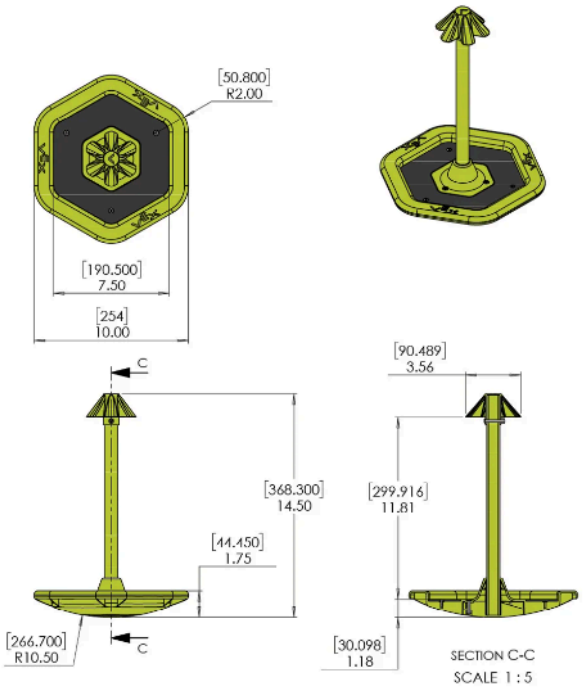


Game Elements: Rings
The majority of the scoring is done via these coloured rings.

- Outer diameter 7’’
- Inner diameter 3’’
- Height 2’’

Potential Challenges

- Rings cannot roll on the floor, so manipulation is strictly contact based
- Large surface area in contact with floor so potential difficulty in manipulation



Scoring Object: Mobile Stakes/Goals There are 5 mobile goals (‘mogos’) on the field, and they can be freely manipulated by teams.

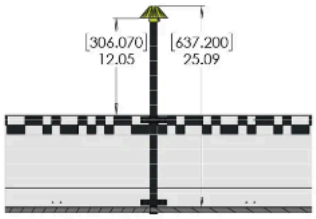
- 10’’ diameter Hexagonal bird’s eye view profile
- 14.5’’ height
- Rubber cap to make descoring more difficult

Potential Challenges

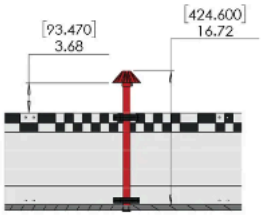
- Weighted bottom could make manipulation difficult
- Scoring would require an elevated mechanism.
- Rubber caps mean force must be required to score/descore

Designed by: Daniel Dew
Witnessed by: Jonah Fitchew

NEUTRAL STATIONARY GOALS:



ALLIANCE SPECIFIC STATIONARY GOALS:



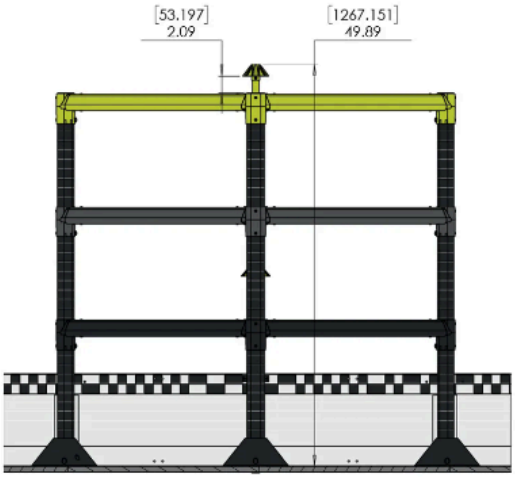
Elevated Stakes: Neutral and Alliance

There are 2 neutral stakes and 2 alliance stakes allowing for further scoring.

- Neutral stake 25.09'' tall
- Alliance stake 16.72'' tall
- Rubber caps (identical to mogo)
- Alliance stakes can only be scored by the corresponding alliance

Potential Challenges

- Stakes differ in height from each other (also from the mogo) meaning different or morphing mechanism to score on all.
- Placement (field perimeter) risks throwing rings out of the field (risking S1 infringement)
- Rubber caps mean force must be required to score/descore



High Stake and Ladder

In the center of the field, there is a 4′ ladder that teams can climb in the endgame to gain extra points. It also has a stake that can fit 1 ring at the very top.

- 49.89'' (4.165′) tall
- 3 tiers/rungs
- 4 sides

Potential Challenges

- Climbing structures requires lots of power and/or time
- High Stake would require extreme precision

Scoring Takeaways

- All scoring requires vertical capability
- Employing multiple methods of scoring (mogo, neutral/alliance stake) would require multiple systems or 1 complex system such as a lift
- Emphasis must be put on precision and reliability as there is little room for error

Designed by: Daniel Dew
Witnessed by: Jonah Fitchew

Rules Analysis

Format

To avoid simply regurgitating the rules (to people who already understand them), we are going to simply list some rules with a paraphrased description; then how it affects us; then potential solutions – if a rule presents no problem, we will not cover it.

e.g.

<RULE NUMBER>

- *Paraphrased rule description*

Problems

- This rule affects us like this
- It also affects us like this

Potential Solution

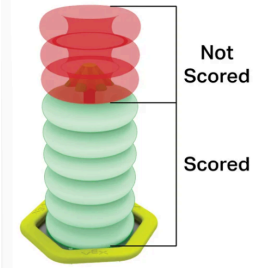
- This is one way we can mitigate the risk of infringement…
- This is another…

> **Note**
>
> Inspection, safety and general rules will not be covered, due to their relative simplicity.

Scoring rules

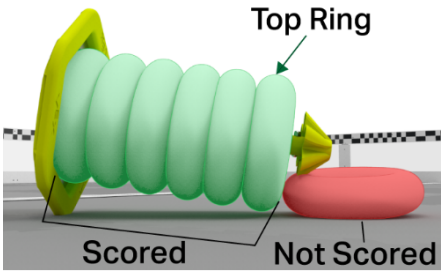
<SC3>

- To be considered scored on a stake, the ring must meet certain criteria:
  1. Ring must not be contacting robot *of same alliance*
  2. Ring is not contacting foam tile
  3. Ring is encircling the stake[1]
  4. Total ring count must not exceed max ring count of the stake (mobile: 6, neutral & alliance: 2, high: 1)



Problems

- Neutral/alliance stakes can contain more than 2, despite only 2 being counted
- Mogos with our rings on can be tipped to effectively descore some rings



Potential Solutions

- Driver may have to take care when scoring on neutral/ alliance stake
- Driver may have to guard or defend filled mogos

[1]Long description omitted, see *https://www.vexrobotics.com/high-stakes-manual#sc3*

<SC5>

- A mobile goal is considered placed in a corner when it meets the following criteria:
  1. Mogo is contacting floor/foam tile
  2. Mogo is upright
  3. Contact with robot is irrelevant

> **Note**
>
> Only 1 mogo can be considered placed in each corner, even if 2 meet the requirements.

Problems

- Mogos can be knocked over to mitigate effect of corner

Potential Solutions

- Driver can guard/defend the corner, especially as robot contact is irrelevant.

<SC6>

- A mobile goal that has been placed in a corner will result in the following modifiers being applied to its scored rings:
  - Placed in **positive** corner:
    1. Values of all scored rings will be doubled
  - Placed in a **negative** corner:
    1. Values of all scored rings will be set to 0
    2. For each ring, an equivalent amount of points will be effectively removed from that alliance's score
    3. Points scored from auton bonuses and climbing cannot be removed

Examples included *here* .

Problems

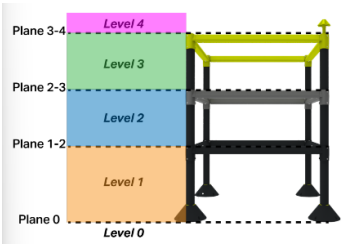
- Opposing rings scored in a positive corner can drastically change outcome of game due to 2x multiplier
- Ring scoring can be easily countered by placing them in negative corner

Potential Solutions

- Once again, large emphasis must be placed on defending scored rings and preventing them from being placed in a negative corner
- Putting emphasis on scoring on the elevated stakes could mitigate dependance on mogo scoring and corner defence/offence

<SC7>

- A robot has climbed to a level when the following criteria is met:
  1. Robot is contacting the ladder
  2. Robot is not contacting any other field elements
  3. Robot is not contacting any mobile goals



4. The robot's lowest point is above that level's minimum height

Problems

- Climbing must be completely independant, it cannot rely on lower rungs or the floor

Potential Solutions

- When considering climbing, large power consumption – due to independant climbing – must be considered, possibly with use of a winch and/or a PTO[2]

<SC8>

- **Autonomous Win point** is awarded to *any* alliance that have completed the following tasks (as long as they have not broken any rules):
  1. At least 3 scored rings of that alliance's colour
  2. A minimum of two 2 stakes on the alliance's side of the autonomous line with at least 1 ring of the alliance's color scored
  3. Neither robot contacting or breaking plane of alliance's starting line
  4. At least 1 robot contacting ladder

Problems

- Even if we can complete as many tasks as possible, AWP is still reliant alliance teammate, especially with no. 3

Potential Solution

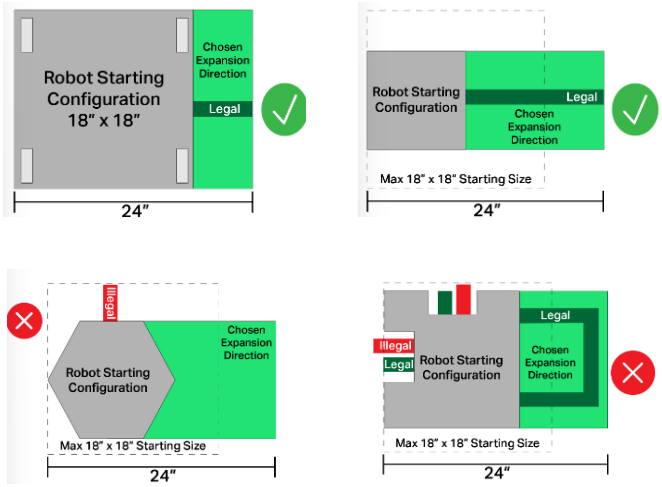
- Ensure prior coordination with teammate to ensure that they move off the line at the start[3]

Specific Game Rules

<SG2>

- Horizontal expansion is limited to an <u>additional</u> 6'' on **one** side.

> **Note**
>
> 6'' expansion is based on an 18'' x 18'' starting size, therefore robot can expand to the limit in **1** direction, then 6'' in the same direction.



[2]PTO: Power Take-Off

[3]If the team does not have a (working) autonomous, advice/technical help can be given to simply move off the line, ensuring AWP

Problems

- Mechanisims that rely on expansion must be contained within the footprint of the robot, or not expand over 6'' on one side only

Potential Solutions

- Design all expanding mechanisms to expand on one side only
- Use as little space of the 18'' x 18'' to maximise expansion capability

<SG3>

- **Vertical expansion is limited**; vertical expansion cannot break 2 or more levels of the ladder

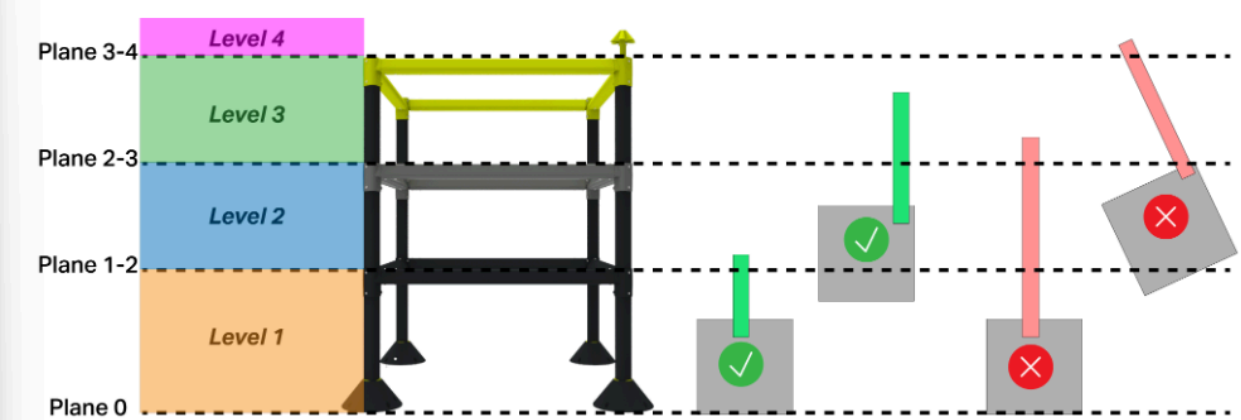


Figure SG3-1: Examples of legal and illegal vertical expansion.

Problems

- This rule makes climbing to the top with 1 movement impossible – unlike in Over-Under – instead teams have to climb the ladder like… a ladder, using each rung and not skipping levels

Potential Solutions

- When designing climbing mechanisms, multi-stage movements must be incorporated; making sure that the robot does not break 2 or more planes no matter the rotation[4]

<SG4>

- Keep scoring elements **in the field**, rings that exit the field will be given to the corresponding alliances to reintroduce into the game.

Problems

- Accidentally removing rings from when, for example, scoring on wall stakes results in a minor violation

Potential Solutions

- Driver can take extreme care when attempting to score on wall stakes
- Line-up guides can be designed to aid the driver

[4]This is because the planes are measured from the perspective of the field (see long explanation *here*)

- Lots of time on tuning the mechanisms to ensure they are not too powerful

<SG6>

- Possesion is limited to 2 rings and/or 1 mobile goal
  - Where rings scored on a stake do not count towards possesion count
  - Plowing multiple mobile goals is legal only when no mobile goals are possessed

Problems

- When attempting to rapidly score rings, this rule may be broken due to more than 2 rings being possessed
- Accidentally plowing a mogo while possessing one will result in a violation

Potential Solutions

- For both problems, driver care can be applied to avoid SG6 infringement
- A distance/colour sensor could be used in conjunction with an algorithm to stop manipulating rings once at the possesion limit
  - Using a colour sensor could allow for a colour sorting algorithm to only intake alliance's rings

<SG7>

- Dont cross the autonomous line during autonomous
  - Robots must not contact or break the plane of the autonomous line[5] during autonomous

Problems

- Accidentally crossing line due to lack of tuning or planning of the autonomous movements would result in the loss of ABP and AWP

Potential Solutions

- Extreme care and consideration must be used when planning out the autonomous movements

Primary Takeaways

Certain solutions appear more than once, meaning we can prioritise them to mitigate more risks at lower time/complexity costs.

Driver Skill

We have concluded that driving is a factor in nearly all the rules specifically targeted defence and offence, High Stakes is a skillful game that requires lots of practice from the driver. Putting emphasis on training our driver, using drills, friendly matches etc. must be a priority.

Control and Precision

We have also concluded that prescicion is key to avoiding rule infringement and also to maintain effectiveness. All mechanisms must be designed with extreme precision with lots of

[5] basically the halfway line

time allocate for fine-tuning to a) maximise effectiveness of mechanism and b) avoid breaking rules such as SG4 and SG7.

The Plan

This game and rule analysis has allowed us to form a plan on how we will approach the coming weeks as we organise ourselves to tackle the season.

Timeline Considerations

- The emphasis on driver practice means we will try our best to allow for plenty of driver practice
- The further emphasis on autonomous tuning means we will have to make ample room for autonomous testing in the timeline

Careful Design

- We will also be making sure that all our designs our designed with strength, prescicion and effectiveness in mind during all stages of the design process – this is especially prominent during the CAD phase

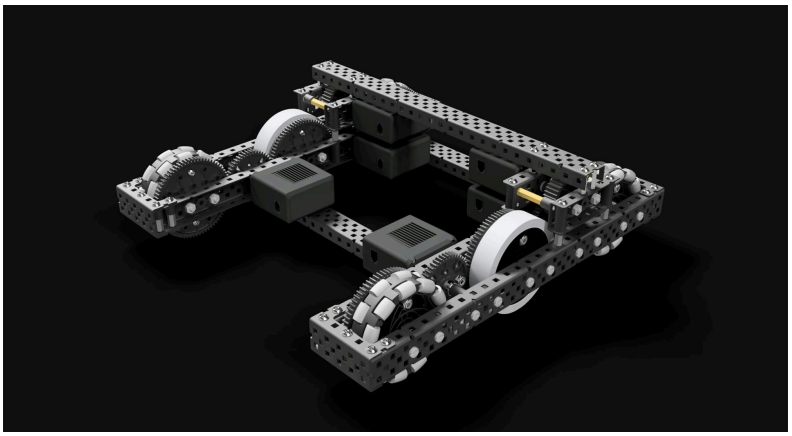
Brainstorming Drivetrain 2024/08/18

What are the different types of Drive?

The first solution to our somewhat obvious problem “We need a drivetrain” is the most conventional VEX drivetrain of them all - the Differential Drive, which features mirrored sides in a typical “tank” formation and allows for independant manipulation of each side in order to achieve the movement we want to see. Second is the Omni Drive which features one omni wheel in the centre of the drivetrain and allows for horizintal movement as well as the conventional forward and backwards. Third is the Mecanum Drive which takes advantage of the VEX mecanum wheels which allow for strafing and a limited form of horizontal movement.

Differential Drive

Pros	Cons
• Simplicity - Easy to build, program, and maintain • Powerful - Excellent for pushing and traction-heavy tasks. • Stability - Will be balanced and support all sorts of mechanisms	• Limited manouverability - Only supports forward, backward, and turning; no lateral (sideways) movement. • Not agile - Slow to make fine, precise movements or quick direction changes



Differential Drive

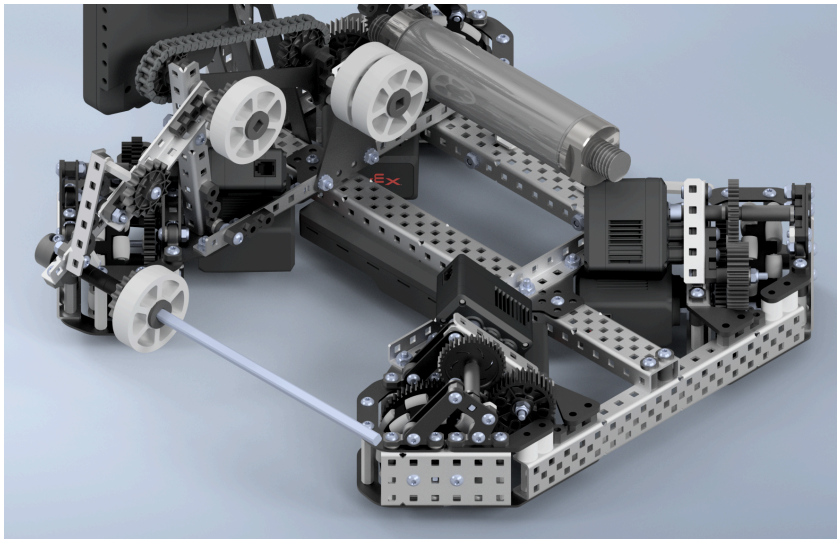
Holonomic Drives

Pros	Cons
• Holonomic Movement - Can move in any direction (forward, backward, sideways, diagonal) with ease. • Manouverability - good for accurate and small adjustments • Agility - Quick, fluid directional changes without rotating the robot.	• Lower traction - Limited pushing power due to low traction of the wheel in the middle • Complex programming - Requires more advanced coding for full control.

Variants of Drives:

Holonomic Drivetrains

X Drive:



Here we see a render of an XDrive done by in_ithica | 3818 on the VEX CAD discord server

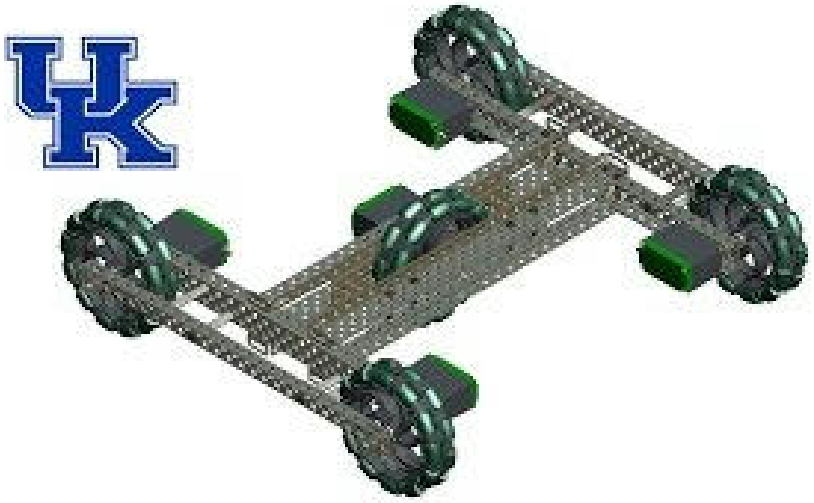
Pros	Cons

1. Higher Speed: X Drive is generally faster than both Mecanum and H-Drive, as it naturally provides more efficient power transfer for quick movements due to the 45-degree wheel orientation
2. Simplicity: Compared to Mecanum, an X Drive is simpler to build and program, as it doesn't require complex motor tuning or algorithms for strafing
3. Better Diagonal Movement: X Drive excels in diagonal motion without the power loss or inefficiencies found in Mecanum, making it smoother for navigating tight angles and corners

1. Lower Pushing Power: X Drive generally lacks traction and pushing force compared to Mecanum, making it less effective in scenarios requiring high torque or pushing resistance.
2. Inefficient Use of Space: X Drive's diagonal wheel layout takes up more space on the robot, which can limit the available area for other mechanisms compared to more compact configurations like H-Drive
3. Complex programming
4. Complex design – especially at the CAD stage

X Drive has traditionally been used by teams for extremely precise programming as it can provide an edge over more traditional drivetrains in terms of accurate movements in things like autonomous skills, however in a more traditional competition format with the goal of Tournament Champions, X Drive does not seem like the most popular choice. X Drives use a compound gear ratio to get around the difficulties in making the wheels diagonal, this can mean that it becomes quite difficult to build and maintain in comparison with more traditional drivetrains. Packaging the brain and pneumatic tank around this drivetrain can be difficult due to the odd motor placements.

H Drive:



Here we see a CAD model of an H Drive found on the website Purdue Sigbots made by the University of Kentucky [1]

Pros	Cons

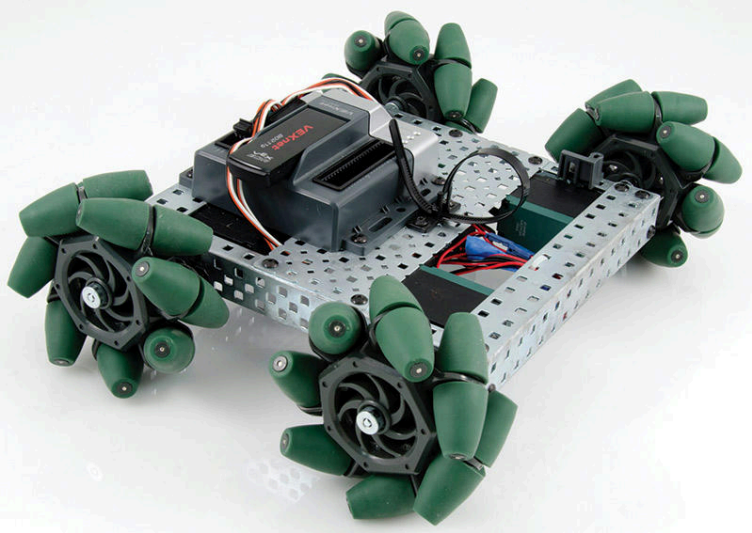
- Simplicity: The H Drive is easier to build and program compared to more complex holonomic drives like Mecanum or X Drive
- Good Strafing Ability: The central wheel allows for decent sideways movement (strafing), providing a balance between manoeuverability and straightforward design
- Efficient Use of Space: The parallel wheel layout in H Drive leaves more space for other components on the robot compared to an X Drive

- Limited Lateral Power: The central strafing wheel in H Drive has less traction and pushing power compared to the Mecanum or X Drive, leading to weaker side-to-side movement
- Less Agility: H Drive doesn't handle diagonal movement as smoothly as X Drive or Mecanum, reducing its manoeuverability in accurate and precise movements
- Vulnerability to Central Wheel Issues: The reliance on a single central wheel for strafing means that any failure or inefficiency in that wheel significantly impacts performance
- Programming Complexity: H Drives also consider some (although less than X and Mecanum) programming challenges
- Space: Having the wheel in the middle can result in less space for other subsystems such as odometry pods

H Drive is probably the most simple to build of all holonomic drivetrains with a single strafing wheel in the middle relying on the omni wheels rollers to provide lateral movement, the gear ratios used are the same as any traditional drive but one of the losses coming from the middle wheel is the lack of packaging ability in that area, no longer is the ability to use odometry pods to provide a higher level of accuracy during autonomous and no longer is the ability to have the pneumatic tank as low as possible to the ground due to the need for extra bracing and support of the middle wheel as inconsistencies and friction can have large effects.

Mecanum Drive:

Brainstorming Drivetrain 2024/08/18



Here is a live model of the Mecanum Drive found on the Servo magazine website about holonomic locomotion

Pros

- Holonomic Movement: Mecanum wheels allow for full omnidirectional movement, including forward, backward, sideways, and diagonal, giving excellent manoeuverability
- Good Pushing Power: Compared to other holonomic drives like X Drive or H Drive, Mecanum maintains relatively good traction and pushing power
- Versatility: Mecanum Drive offers a solid balance of movement options while still being able to handle various competition tasks, making it adaptable to different areas of teh competition like skills

Cons

- Complexity: Mecanum Drives are more complex to build and design than simpler drives like H Drive, requiring precise motor control and alignment for effective movement.
- Power Loss: Due to the angled rollers on Mecanum wheels, some power is lost during lateral movement, making it less efficient compared to a tank or X Drive
- Programming Complexity: Mecanum Drives are also harder to program

The Mecanum Drive is one of if not the most frowned upon drivetrain in the whole of VEX, as it utilises the otherwise useless Mecanum wheels which take up a large amount of room and are not very versatile. However in a Mecanum Drive you acquire the ability to complete control and assurance that movement is not the problem, however in building this drivetrain the ability to gear is almost completely gone, as in order to maintain the very much wanted 3-4 wide hole gap between the drivetrain C Channels the mecanum wheel leaves no room for a gear, admonishing any gear ratios wanted.

Use of Traction Wheels:

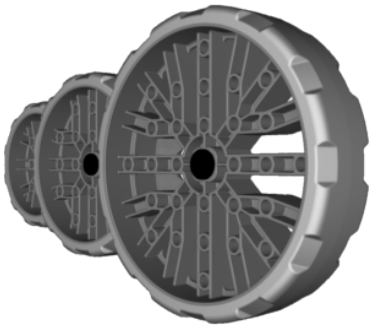


Image credit

For our drive an important consideration was the inclusion and amount of traction wheels which we wanted to use. Having more traction wheels increases our grip but also reduces our skidding which can mean that our bot will struggle to do tight turns quickly. However, not having enough traction wheels means we may be easier to push around and struggle more to push others.

To maximise grip whilst retaining agility, it is common to use traction wheels in the centre to reduce unwanted pushing forces, whilst combining them with other wheels such as omniwheels on the edges for greater turning ability.

Pros	Cons
• Increased Grip: More grip allows for more power, which means it's easier to push game elements or other bots around. • Resistance: Since they have a single degree of freedom traction wheels make our robot more resistant against being pushed when attacking our defending	• Rigidity: Having one degree of freedom means that the bot will struggle more to perform tight turns, which may be important to perform midmatch or during the autonomous which often relies on turns being consistent and accurate.

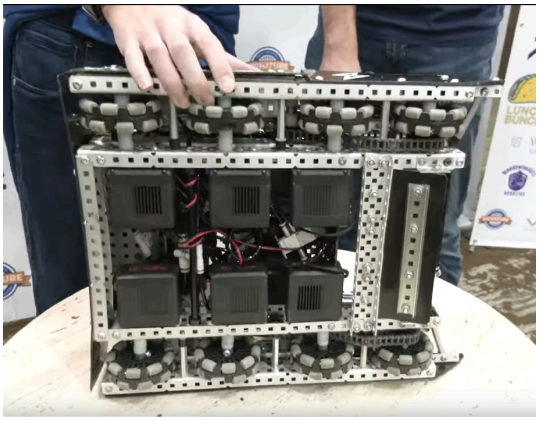
Overall, it is important to decide how many traction wheels we use and where. Ultimately, deciding this depends on a teams game strategy and the type of bot they are going for.

Citation on traction wheels

All Omniwheel Drive:

Brainstorming Drivetrain 2024/08/18

Having our drive consist of only omniwheels has some advantages such as allowing for a lot more skidding and tight manoeuvering. However, a lack of traction wheels means our bot will have less torque to push and also will be easier to push. This pushing power loss can be compensated for by playing more evasively in order to outmanoeuver opponents. A good example of an all omniwheel drive was 9364H's bot in OU *Pits and Parts* , who additionally engineered their bot for speed and agility to control the arena and punish double zoning in that game. See image on right for example (This image was cropped from Pits and Parts video cited above).



Number of Wheels:

Another important consideration is the amount of wheels, which we want on our drive. Similar to traction wheel ratio, having more wheel affects our torque-agility ratio. Having more wheels increases surface area in contact with the ground, which correlates with more traction. An increase in traction means that we have less skidding, which can be a detriment since skidding can be useful for turning, but it will also mean that the power is transferred more efficiently, which allows us to have more rpm or torque. See explanation for this on the right.

Energy = Power * Time
$E=Pt$
Energy = Force * Distance
$E=Fs$
Therefore
$Fs=Pt$
$P=Fs/t$
$P=Fv$
Power = Torque * RPM

4 Wheels

Having 4 wheels is the minimum number of wheels a drivetrain can have. It sacrifices power for manoeuverability but is easier to implement since it doesn't require motor stacking.

Pros

- Agility: Slides more so it has greater theoretical turning ability
- Easy to implement: Since there are fewer wheels there's fewer gears to deal with allowing for more space for game specific components such as wings, intakes or lifts
- Lighter: Less components so robot will weigh less meaning motors have less load to move

Cons

- Less power: Fewer wheels so there's less traction, which means less pushing power
- Lighter: Less weight so it's easier to push around

6 Wheels

6 wheels is a balance between manoeuverability and power. However, it's trickier to implement since it requires motor stacking.

Pros	Cons
• Agility: Still slides but not as much as a 4 Wheel Drive • Power: Balanced power but still less traction than an 8 Wheel Drive	• More complex: Requires motor stacking so its design as a drivetrain is more complex, which can mean it's more difficult to implement game scoring subsystems.

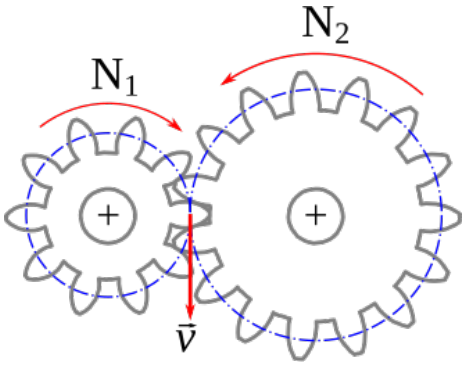
8 Wheels

An 8 Wheel Drive provides the most traction of the three options, which gives it the most power but means it will slide less and so will struggle with turns more.

Pros	Cons
• Most Power: Has the most power because of the greater traction • Heavier: More wheels and more gearing weigh more so it's more resistant to pushing and will push with more force	• Less Agile: The higher traction stops the robot from sliding as much so it will have less turning ability. • Heavier: The greater weight will put more strain on motors, which leads to higher chance of motor burnout. However, this can be circumvented by quickswaps or cooling the motors with a fan.

Overall, it's important to consider the strengths and weaknesses of the wheels chosen and amount of wheels, since the power lost from using fewer wheels can be compensated by using some traction wheels to create overall high performance or similarly an 8 wheel drive can become more manoeuverable if the drive includes some omniwheels or is entirely made of them. Therefore by considering what the wheels and wheel ammount can do for the drivetrain it is possible to create a drivetrain, that doesn't suffer from any major weakness within the scope of what our team is trying
to achieve.

A Look at Gear Ratios



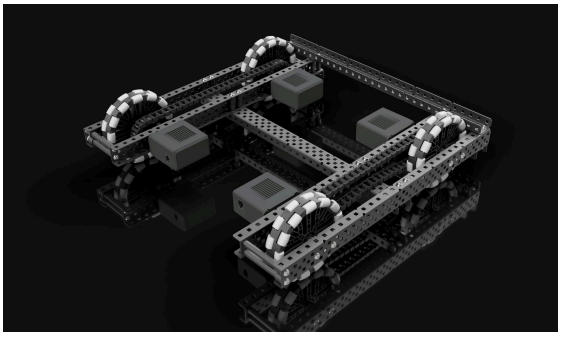
Another factor which contributes to the balance between torque and agility is the gear ratio chosen. If the force applied on the driven gear is further from its pivot compared to the distance from the driving gears pivot then the driven gear will turn with more force but it will turn slower compared to the driving gear. If this coupling is reversed then the driven gear will turn with less force but it will turn faster compared to the driving gear. The gear ratio is often described as either $\frac{\text{driven gear } \Omega}{\text{driving gear } \Omega}$ where Ω is angular velocity or $\frac{\text{driven gear teeth}}{\text{driving gear teeth}}$. Like with wheel type and amount the gear ratio of the drive can be used to further optimise a drive to have greater speed or greater pushing power. On the left is an example of a 2:3 gear coupling. *Image Credit*

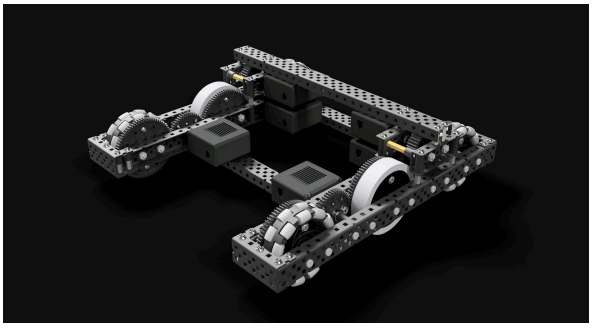
It's also important to consider that wheel size also acts like a gear and so affects torque-speed.

> **Note**
>
> Whilst it's not very commonly used in the context of V5, it's theoretically possible to have a gear transmission to switch between high torque and high speed, which may have a niche to save on motors when trying to make a bot capable of having high speed and pushing ability without commiting as many motors from 88W limit.

Direct Drives

It is also important to consider the viability of direct drives, where power is drawn directly from the motors. This can be useful as it has a lower build complexity allowing for more space and parts for other game scoring subsystems. They can be consisdered when you want to go with specific rpms such as 200 or 600, in order to have a unique torque-speed ratio compared to the rest of the competition. Below is a comparison between a direct drive and a 6 motor geared drive. *Direct Drive Geared Drive*





Drivetrain Selection Process

The process of selecting a drivetrain is a teams first thoughts when considering the needs for a robot in any given VEX season. The benefits and drawbacks of all types of drivetrain are considered at this point of the design cycle, in the context of the current game of course, many different options are considered as the different parts that VEX provides can be combined in many different ways in order to give each drivetrain a different characteristic as it were.

Considerations for any given drivetrain

At the beginning of any season the needs of the three main aspects of a drivetrain are pitted against each other in the context of the needs of the current game, these three main features are:

- Torque
- Speed
- Size

	Manouverabili	Stability	Ease of build	Size of Wheels	Gear ratio achievable	Total
Differential	3	5	5	4	5	22
Holonomic	4	4	2	4	3	17

For our current purposes and team requirements, a Differential Drive is the most optimal drive to consider using. This is because of their simplicity; which we feel is valuable this early in the season since most other drives suffer from odd motor placements or other restrictions which make designing around them more complex, the power they provide; which is always helpful for pushing tasks, and their stability means we are less likely to tip over mid match. In addition, we have deemed that the downsides of a Differential Drive are either not substantial or can be compensated for by other parts of the drive. For example the lost manoeuverability can be made up for by incorporating omniwheels into our drivetrain.

Choosing Wheel Ammount

Choosing the correct amount of wheels is an important consideration when designing the drivetrain. It is an extra way to achieve an optimal balance between torque and agility. The three main considerations for wheel ammount are:

- Traction
- Manouverability
- Complexity

	Manouverability	Traction	Complexity	Gear ratio achievable	Total
4 Wheels	5	2	5	3	15
6 Wheels	4	3	4	3	14
8 Wheels	3	5	3	5	16

From our analysis we decided that an 8 Wheel Drive is our best choice, since it synergises most strongly with a Differential Drive. Combining these two gives us great freedom for gearing, allowing greater control over torque and speed. Additionally, both elements will contribute to a higher power drive train. The simplicity of the Differential Drive compensates for the higher complexity of having 8 wheels. Unfortunately, neither of these design elements give us the greatest manoeuverability, which is the greatest problem with our design at this current stage. This can be addressed with the wheels we choose, since having 8 Wheels gives us the most flexibility over combinations of wheel types.

Wheel types

The combination of an 8 Wheel Drive and Differential Drive gives our bot great power, whilst also being relatively simple to build. The main consideration for the combination of wheels, which we choose is that they give our bot greater manoeuverability since that is the main problem with our current solution.

	Manouverability	Traction	Total
All Traction Wheels	1	4	5
Mixture	3	3	6
All Omniwheel	4	1	5

We have decided to go for a combination of 6 Omniwheels and 2 Traction Wheels, positioned third from the front. Having this wheel in this position means we still resist pushing from the sides but allows us to have greater manoeuverability thanks to the 6 omniwheels. Our experience from last season told us that placing the traction wheels there was best for turning and according to *purduesigbots* central placements helps reduce unwanted sideways movement. Another consideration was to place the wheels in a diagonal configuration, but experience told us that middle back was ultimately the superior choice.

Drive Gearing

The final consideration for our drive train will be the gearing we go for. This is the final way in which we can alter our drive train in order to get its torque and speed within desired parameters. We decided to use blue cartridge motors to give us 600 rpm, which we can gear down to 450rpm, with 3:4 gearing, to further increase torque but still have a good speed. We chose 69.85mm (2.75in) wheels to have greater power, so we don't compromise acceleration from the higher rpm. We didn't go for direct drive since we didn't want 200rpm or 600rpm.

What is a Programming 'Approach'

When tackling any project, thorough planning and thought is required to allow the team to effectively solve it. Defining a standard approach to certain aspects of the challenge can allow certain beneficial procedures to become instinctual, therefore allowing the team to become more efficient.
This is especially prominent when looking at programming, code on large projects can become completely unorganised and hard to read – which is particularly disastrous if you are in a team (where some or most of the members might not intuitively understand the programming language itself).

Common Approaches To Programming

Some aspects of code are typically defined by some form of rule or guideline, examples include:

- Variable naming conventions
- Usage of subprograms or classes
- Choosing a language
- Version or project management
- Program structure
- Safe and secure programming

TL;DR

Standardised programming – through usage of the above – helps to improve:

- Team communication
- Debugging
- Team/project management
- Readabilty of code
- Enjoyment of writing code

Brainstorming Approaches

Here, we will define what approach(es) we will use when coding, aiming to improve useability, team communication and debugging capabilities.

Language and Software

One of the most important decisions to make is the programming langauge and its accompanying software and libraries. For robotics, the two most common langauges are C++, Python and now, Rust which is making an appearance with *vexide*[1], but the software used alongside it is also very important.

Python with VEXCode V5 or RoboMesh Studio:

- The only current way to code with python is using VEXCode V5 or RoboMesh Studio

Pros	Cons
• Python is exceptionally easy to read and write thanks to its variable type declaration and readable syntax • Devices can be set up using the built in GUIs	• Using GUI for devices can be restricting • There is no allowances for *any* libraries, something that python is known for • Multiple files are not supported, therefore less orginisation through program structure is possible • No choice in editor, therefore no choice for extensions, themes, formatting etc. • Fairly limited device API, less control over devices

C++ with VEXCode or VEXCode Pro

- VEX offers an alternative to python with C++ in the IDEs VEXCode and VEXCode Pro
- VEXCode Pro allows users to use multiple files, along with header files and other c++ functionalities.

Pros	Cons
• C++ offers much more raw functionality as it is a lower level language[2] • GUIs are still available to configure devices	• Still no access to custom libraries • Rigid IDE, no access to extensions to improve workflow • Fairly limited device API, less control over devices • Cannot declare devices both in code and with GUI, must choose one or the other

C/C++ with PROS

[1] An open source Rust runtime for v5 robots
[2] A language that is closer to manipulating raw memory, giving users more control over memory

- *PROS* is an open source development envirionment for VEX founded by Purdue University
- Allows users to write code in C++ or C (C++ is generally pereferred)
- Integrated within existing IDEs, e.g. VSCode or Atom

Pros

- Allows full C/C++ functionality
- Multiple files supported, including header files
- Full template and library functionality
- Hot/Cold linking: only changed code is uploaded, allowing for fast uploads even wirelessly
- Built into existing IDEs, so extensions, formatting themes etc. is all supported and controlled by user
- Significantly improved device API, allowing for fine control over all devices/ components, including serial inputs/ outputs and direct control over radio
- Easy to get started, but with in depth capabilities for niche applications
- Huge ammounts of documentation e.g. *Purdue SIGBots wiki* or *API docs*

Cons

- Can be harder to understand concepts

Rust with Vexide

- *Vexide* is an open source 'no-std'[3] Rust runtime for vex V5
- It is a successor to *pros-rs* , which binds Rust code to the PROS API.
- Allows users to code using Rust, while supplying a CLI to manage projects or interact with a device
- Vexide will be included in a family of vex based applications, such as *vex-v5-qemu* , a CPU level simulation for PROS and Vexide code (includes node-based GUI for device configuration).

⚠ Warning

Vexide is still considered experimental, it has a small base of contributors that are working to make it more and more usable.

[3]'no-std' is a type of package that limits the use of standard libraries. The V5 brain runs without an OS, meaning std libraries impossible to use.

Pros	Cons
• Rust is a langauge designed around memory safetly, including things like variable 'ownership' to avoid memory leaks. • Vexide will eventually work seemlessly with a range of other projects developed by the vexide team.	• Rust is not an easy or intuitive language to learn – we have very limited experience with Rust • Because of it being so new, vexide does not have a stable base of users – meaning less documentation and support • No-std means basic mathematical functions are not accessable[4] • CLI is still limited especially when comparing it to PROS

Variable Naming

Naming conventions are very useful when writing *and* reading code. They can make long, complicated names easy to read; or can help clarify the intent or context around a variable.

Variable requirements:

Most languages (C++ included) require variables to adhere to certain rules:

- Must not start with a digit
- Only alphanumeric values or underscores
- No spaces/whitespaces
- No isolated keywords (e.g. 'if', 'for', 'import' etc.)[5]

Common types of variable naming:

camelCase:

Explanation:

- Variable names start with lowercase
- All new words within the variables start with an uppercase

Example:

```cpp
int dateNow() {
  // Get date
}
bool thisIsAVariable = true;
int currentDate = dateNow();
```

Pros/Cons:

Pros	Cons

[4]There are ways around this
[5]Dependant on language

- Easy to understand multiword variables
- Some programs recognise camelCase, allowing them to display variables with whitespaces
- Satisfying variable 'shape'

- Variable names can become long
- Some words can look confusing e.g 'A' in 'thisIsAVariable' ('A' can be hard to see)

snake_case

Explanation:

- Using underlining to represent whitespace
- Words typically start with lowercase

Example:

```cpp
int date_now() {
  // Get date
}
bool this_is_a_variable = true;
int current_date = date_now();
```

Pros/Cons:

Pros

- Very easy to understand understand multiword variables
- Very easy to see where whitespace is supposed to be

Cons

- Variable names can get very long
- Somewhat difficult to program with due to frequent use of '_'

> **Note**
>
> The difficulty from frequently using '_' can be circumvented by using a program such as Auto HotKey to rebind '_' to something such as "Shift" + "Space". However, this work around may not be worth it for the express purpose of making a naming convention easier.

Boolean 'is' naming

Explanation:

- Start all booleans with 'is'
- Often times subprograms that return booleans start with 'get' ('getIs…')

Example:

```cpp
// (Using camelCase)
bool getIsSaturday() {
  // is it a saturday?
}
bool isSaturday = getIsSaturday();
```

Pros/Cons:

Pros	Cons
• Easy to differentiate between regular procedures and functions that return a boolean value • Works best with camelCase but can also work with other variable naming conventions	• Variable names can get very long • Doesn't help identify between procedures and functions that return other data types

> **Note**
>
> Procedures are defined as subprograms, which don't return **any** value, whilst functions are defined as subprograms, that return a value of a **specified** data type.[6]

Pronouncable names

Explanation:

- Using abreviated words that are pronunciable and easy to make sense of

Example:

```cpp
int getCurrDate() {
  // Get date
}
bool thisIsAVar = true; // variable = var
int currDate = getCurrDate(); // current = curr
```

Pros/Cons:

[6]Languages such as Python do not require such precision in subprogram declaration mitigating the difference between functions and procedures. However, most other languages require procedures to be declared with the **void** keyword to specify they don't return a value.

Pros	Cons
• Drastically shortens variable names	• Not all abbreviations will make sense to everyone • Not using standard english can make documentation and understanding code harder: text autocompletion in modern IDEs means that long names aren't a problem to type.

Unit classification

Explanation:

- Suffixing all variable names (where applicable) with a unit (e.g. rpm, lbs, kgs etc.)
- Using an underscore to seperate unit
- Best used with snake_case

Example:

```cpp
int getMotorSpeed() { // Dont need unit classification for subprograms
  // Get RPM
}
int motorSpeed_rpm = getMotorSpeed(); // suffixed with '_rpm'
```

Pros/Cons:

Pros	Cons
• Units are always known • Conversions can be easier because input/output is documented in the name	• Takes longer to document all variables' types • Increases variable length

Using Subprograms and Classes

- Subprograms are smaller blocks of code that can be run anywhere in user code
- Classes are structures that allow for variables to be 'owned' and can drastically help organisation[7]

Pros	Cons

[7]We may do a deep dive on classes at a later point

- Organised and readable code
- Code can be run multiple times using less lines
- Classes allow for even further organisation
- Classes can help mitigate developer mistakes

- Classes can take time to write
- Variables defined in different files can lead to null pointers during initialising[8]

Version Control

An equally important issue for programming as a discipline is Version Control. This is important because it allows multiple people to work in parallel on the project from anywhere to develop one feature or several and then merge changes into one main branch of the project. In addition, Version Control softwares allow developers to mess with experimental changes without the fear of ruining the project, and it also acts as safety net incase a changeset that enters the main branch ends up breaking the project, since it can easily be rolled back to a stable build. Another advantage of Version Control is that it allows for new and old code to compared for performance over whatever parameters we test - allowing us to easily prove if our new solution is an improvement.

Examples of Version Control Software

There are many different Version Control Software. For this logbook, we are using GitHub - which is built off Git, since it has direct support for Typst and Notebookinator. Other examples include:

- Beanstalk
- PerForce
- Apache Subversion
- Mercurial

Program Structure

As well as using subprograms and classes for organisation, the usage of libraries and splitting the code up into different files helps keep the software for the robot structured and helps prevent accidental changes when working on different parts of the code. Our experience from last year taught us that its best practise to have the code for driver control, match autonomous and autonomous skills on seperate files. In last years game OU this principal was taken further since we had different routes for our left and right side autonomous for matches.

Warning

Global variables have to be initialised in 1 file to avoid Null Pointer Exceptions since the C++ compiler doesn't specify initialisation order.

[8]Basically, the C++ compiler does not have a specified order for intialising variables in different files, meaning if 1 variable depends on an unititialised variable (from another file) it can throw a memory error

Safe Programming

When working in medium level languages such as C++, the necessity for optimising your code and ensuring you handle computer memory and pointers properly increases. Failing to do so effectively, will cause crashes and problems on your machine and at a large scale can even throw the world into *disarray* .
If we use C++ again this year, we must make sure to initialize variables in the right order and ensure we handle memories and pointers properly. General things we can do ensure this are: using smart pointers, using tools such as static analyser; and finally using STL containers.

Smart Pointers

These help reduce the amount of manual memory management. The Microsoft *documentation* states that they are crucial to the RAII or Resource Acquisition Is Initialization programming idiom. The basic premise of RAII is to ensure finite resources are not wasted and so must control their usage and destroyed once its no longer useful - ie when a variable goes out of scope. The usage of Smart Pointers greatly reduces the chance of bugs and memory leaks since memory is automatically deallocated when the resource is no longer used.
Below is a comparison of a raw pointer vs smart pointers.

```cpp
void UseRawPointer()
{
    // Using a raw pointer -- not recommended.
    Song* pSong = new Song(L"Nothing on You", L"Bruno Mars");

    // Use pSong...

    // Don't forget to delete!
    delete pSong;
}

void UseSmartPointer()
{
    // Declare a smart pointer on stack and pass it the raw pointer.
    unique_ptr<Song> song2(new Song(L"Nothing on You", L"Bruno Mars"));

    // Use song2...
    wstring s = song2->duration_;
    //...

} // song2 is deleted automatically here.
```

> **Note**
> For the majority of C++ programming, smart pointers aren't necessary to manage especially within the context of VEX programming.

How Do We Decide Our Programming Approach?

While we typically use quantative methods for deciding aspects of our robot, such as the drivetrain (for example using descision matrices); we will avoid using such methods when deciding our programming approach. While we believe taking care when picking a programming approach is *very* important, much of it is still personal preference and can be decided by simply evaluating what fits us best.

Deciding Our Programming Appzroach

Here we will dsipaly the choices we made on the different aspects of the programming approach.

Language and Software

> **Final Decision**
>
> We ultimately decided to code in C++ using PROS

There were many factors involved with this, but experience played a big part in us choosing this: we used python with VEXCode for the majority of last year, but we found it was very limiting, with the lack of multiple file support and library support; we also used PROS/C++ last year for nationals and worlds, and we found it to be very reliable, even if we didn't utilise libraries to the fullest.
I did a significant ammount of research on Rust with vexide, talking to the main developers about the state of the project – I found that while it seemed like a very interesting project, there was not currently enough surrounding support to justify using the more unfamiliar language: Rust.

Variable Naming

> **Final Decision**
>
> We decided to use the following variable naming conventions:
> - camelCase
> - 'is' boolean variable naming
> - Pronounciable names

Variable naming is mostly inconsequential, so we chose what we were most familiar with. It is important to stick with this, as collaborative programming is much easier when the conventions are stuck to.
Arguments can be made to say that stricter rules must be followed, for example this *video* states why long variable names dont matter, and that unit classification is important. However, in VEX, code changes are often made in a hurry, and its easier to focus on a few instinctual rules than to have strict ones that force the programmer to pour over the code after rushed

changes are made to fix formatting – we are also unlikely to go above 1-2 people making changes to the code, so punctuality matters less.

Version Control

As programmers get more experienced, it becomes more and more apparent that version control is a ***must have***; as projects grow larger, mistakes become easier, making the ability to revert code into a stable state is essential.

Final Decision

We decided to use the most common form of version control: Git

Git, using *GitHub* , allows us to easily collaborate and use code accross multiple devices, with branching functionality and the ability to manage multiple projects as an orginisation[1].

Note

Small changes that we need to implement on another device can be shared in a discord channel, to avoid cluttering the version control.

Other approaches:

These are approaches that don't require descisions (you either use them or you don't)

- Subprograms and classes will be used where applicable – especially subprograms[2]
- We will prioritise uisng multiple files to aid with orginisation
- We will adhere to general safe programming rules, especially avoiding unititialised variables, as the compiler doesn't catch them and can be extremely difficult to debug.

Sticking To The Rules

With all these 'rules' it may seem like it will be impossible to follow these in practise; however we can implement procedures to help aid this process. We have taken inspiration from how compa nies maintain their often huge codebases.

Example

Fun Fact: all of Meta's code (including instagram, facebook, Meta VR etc.) is all kept on the same git codebase, meaning zuckerburgs original facebook code is available to *all* meta developers

- Before commiting to the git, we will make sure that all code follows the formatting rules

[1]Our orginisation: *https://github.com/29457A-SNowflakes*
[2]Classes, in the context of VEX can become more of a nuisance than an aid, so they must be used sparingly

- If code changes have to be made in a rush – for example during a competition – commits to the git will be flaged as bad code (often using the :poop: emoji[3]), and will be formatted at a later date.

[3]This is actually an industry standard flag for bad code

Glossary

Example word

This is an example word which will appear in the glossary.

Bibliography

Bibliography

[1] BRLS, “Purdue SIGBots Wiki — wiki.purduesigbots.com.” 2022.