Final Design Report MTE 100/121
TKOM - Robot Project

Department of Mechanical and Mechatronics Engineering

A Report Prepared For:
The University of Waterloo

Prepared By:
Viyasan [redacted]
Ethan [redacted]
Ramy Wahib [redacted]
Farhaan [redacted]

150 University Ave W.
Waterloo, Ontario, N2L 3G1
December 2, 2025

Table of Contents:

List of Figures:

List of Tables:

I. Introduction

This project aims to encourage children to spend less time on their screens. Studies show that increased screen time can reduce the white and gray matter of children [1]. White and gray matter are aspects of the brain that are crucial for information processing and other cognitive functions [2]. The solution that was decided to be incorporated was to build a robot to encourage children to spend less time on screens and more time on drawing. This is because studies show that drawing increases white and gray matter [3]. Therefore, a robot that can encourage children to replace their screen time with drawing time will regulate children's brain health to normal and could potentially get white and gray matter levels above average. This research is the basis for the robot's design.

II. Scope

The main functionality of TKOM is allowing any user to draw a video game concept on paper, and be able to play it. To start, the user cuts out provided paper templates for a main character and for a paper belt. The character can be drawn as anything, and may include a backtrail if the user desires. The markers used for the paper belt are provided and each indicate a different aspect of the game. For example, green means ground, blue means add to score, red means remove from score or end the game, and yellow means end of map. These can be used to draw almost any 2-dimensional game imaginable, everything from simple platformers and mazes, to popular games like Flappy Bird, Geometry Dash, and Piano Tiles. After inserting the paper belt into the robot and starting the game, the user gets to pick from many game customization options such as simulated gravity, autoscrolling, and multi-jump. The provided controller is used to play these games using built-in joysticks and buttons.

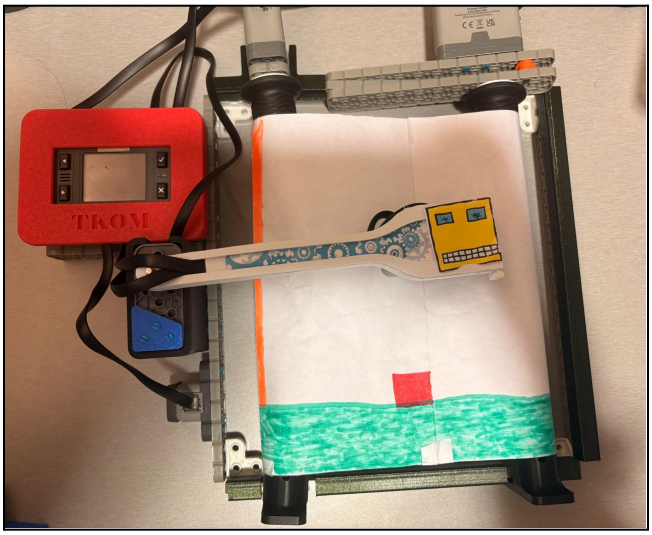


Fig. II.a Top View of TKOM Robot

A variety of sensors are used around the entire robot. As mentioned above, the user plays games using the joysticks and buttons on the handheld controller. The game option selection was done using the screen and buttons on the brain of the device. A bumper on the side of the robot is used to exit games and

eventually exit the program, allowing users to stop playing at any point. A distance sensor mounted below the main character is used to detect if a paper game is loaded into the device, and allows the user to start a game. This sensor is also used in our extended game mode, where a longer roll of paper can be wound around the rollers to provide the ability to play a long game. When this roll is fully unwound, the sensor detects the absence of paper and immediately ends the game before the paper can rip. Lastly, a color sensor mounted directly below the main character is used to detect the different game colors and execute game functions accordingly.

Three motors are used throughout the device. One is used to control the vertical motion of the main character. This is done by mounting a swing arm onto the motor which is mounted vertically, where the end of the swing arm represents the main character controlled by the user. The paper background is controlled using two motors, one on each roller. As these motors spin, the paper background moves horizontally behind the main character, representing forward and background motion by the character. The reason two motors were used instead of one, was for the extended paper mode. Since paper motion can be both forward and backward, the paper would not be able to rewind backwards onto an unpowered motor and would break the machine. The screen of the brain is also used to output the player's score at the end of a game.

The robot carries out various tasks throughout the game, most of which involve the color sensor. When this sensor detects a color, certain tasks need to be executed based on the specific game configuration. For example, when yellow is detected, the game is over. It will execute this task by stopping motor motion and outputting score to the screen. Another way of shutting down is by pressing the bumper on the side of the robot, which immediately stops the game and outputs score to the screen. If a button on the controller is pressed, based on the current state of the game as determined earlier by the color sensor, motion may take place of either the main character or game background. Tasks are generally considered complete immediately after a task is performed such as moving the game background or increasing main character velocity.

Some changes were made to the scope of this project based on a variety of factors. Originally the intention was to create a multiplayer system for competitive games like sports or collaborative games like platformers. This would add a social aspect to the robot which can increase user satisfaction. However, due to the limitation of a singular game controller, and the absence of electronic alternatives like additional bumpers, the idea was scrapped in favor of the current singleplayer design. Some software features were never added such as wall jumps and more intuitive menu options due to strict deadlines and the need to dedicate time towards fundamental bugs largely created due to inconsistent vex electronics. Other ideas such as a pay-to-play system were scrapped as well due to hardware limitations and time constraints. Concepts to make the main character move vertically using a rack and pinion or belt system were set aside in favor of the current mechanism due to simplicity, and therefore reliability which was able to speed up development time. Overall, the original scope was closely followed and all the fundamental aspects were implemented fully.

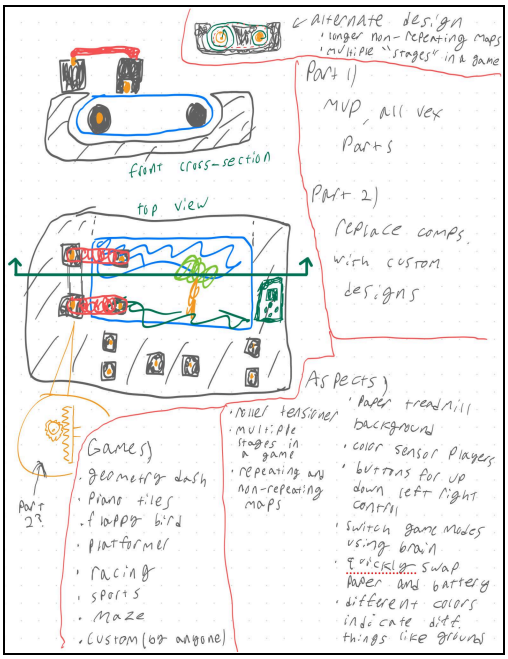


Fig. II.b Original Project Planning

III. Constraints and Criteria

Throughout the development of TKOM, several constraints and criteria have been formulated to ensure satisfactory performance that would indicate it is ready for mass-market adoption. Research by PWC indicates that 32% of consumers would abandon a product after just one negative experience [4]. This demonstrates that maintaining a spotless user experience is essential for optimizing market adoption, and ensuring long-term success.

The following four criteria were initially proposed to ensure the product would be ready for mass production and market adoption:

1. The product should be able to work with any adequately drawn background. Users cannot be expected to draw perfectly straight lines or perfectly color in an area, so the product should be able to handle minute errors or inaccuracies.
2. The character's movement should be snappy, fast, and accurate. This allows for faster paced gameplay and will reduce error from the device's end, making the limiting factor not the ability of TKOM, but rather that of the player's skill. If a player loses the game, they should not feel it was due to a hardware malfunction, but rather a fault of their own, and this criteria helps ensure that.
3. TKOM should integrate a structured reward system to improve user engagement and retention. Research from MIT Sloan School of Management shows that gamified elements such as points, leaderboards, and badges significantly boost long-term user retention in apps like Duolingo and Nike SNKRS [6]. Incorporation of gamified elements takes advantage of the user's intrinsic motivation to keep them more engaged in the product, which acts as an effective harm reduction technique when compared to the large amounts of dopamine released by social media apps such

as Instagram and Snapchat [7]. This brings TKOM one step closer to its ultimate goal of helping the population rid themselves of their harmful cellphone addictions.

4. The paper background on TKOM should be easily visible to the player. According to research done by Caroux and colleagues, games with higher visual clutter or distracting elements reduces player accuracy, thereby increasing its perceived difficulty [8]. If TKOM's games become too difficult, then the average user will not be able to effectively enjoy them; they would rather become frustrated with its difficulty, which may ultimately result in discontinuation of usage. This leaves the user much more vulnerable to relapse to their old cellphone addictions, which would be a failure from TKOM's end.

All of these constraints, save criteria 3 were eventually deemed to be too limiting in terms of scope, however. These constraints would lock the eventual product into a very narrow type of device, which reduces space for innovation and new ideas further down the engineering design cycle. As such, the following two criteria have been proposed to replace criteria 1, 2, and 4:

A. The product should run reliably. The rationale for this criteria is the same as #1 and #2, however it applies to a much broader range of mechanisms. It reduces the absolute need for a user-drawn looped background with a motor-controlled character arm while covering said mechanism with the same rigor as the previous constraints.
B. The product should be user-friendly. Operating TKOM should not require the reading of extensive documentation, as users generally are less inclined to read technical writing if they can help it. Additionally, studies have shown a product's perceived ease of use to be a strong indicator as to how long they are likely to keep using it. To effectively combat cellphone addiction, the user should be able to adopt a harm-reduced variation without significant trouble, and they should be able to stay with it long enough for their primary addiction to subside. By making the initial adoption and subsequent use as seamless as possible,

A. Guiding constraints

Throughout the engineering design cycle, it became evident that constraint A would be the primary guiding criteria. This is because the highest priority for TKOM was to have an enjoyable user experience so the user would be more inclined to use it over their cellphones. The most important part of the user experience, in this scenario, would be the quality of the actual game as it is the entire point of TKOM in the first place.

To optimize the game experience, it became paramount to ensure reliable operation of the product, and so several steps were taken to ensure TKOM would run reliably. For instance, the vex robot was meticulously tested, and the refresh rate of the tick system was optimized to just the right value so the robot could scan its sensors as frequently as possible, making functions like the color sensor more accurate, all while ensuring each tick would fit into its allotted time slot. An oscillation function was also added to motorPlayer to ensure the player object would linger near the top of the green area to ensure consistency across different playthroughs.

B. Nonguiding constraints

Unlike criteria A, constraint B was not especially useful in the development of TKOM, as the product is not conceptually difficult to understand. Most of its target demographic very likely has experience playing 2D platformer style games, so the instruction of playing and designing custom maps did not take much dedicated instruction. This constraint was only helpful when receiving the program's

initial parameters, where care was taken to ensure the input of initial values, especially numbers, was as clearly explained as possible given the space available on the VexIQ Brain.

IV. Mechanical Design and Implementation

A. Description

This robot, which will now be referred to as TKOM (The Kore Of Mental-health) was designed to engage the user in creativity through the form of drawing; which engages the brain, forming grey matter. The user interacts with TKOM by placing in their drawing of a gameboard, with each of the colours corresponding to a game element: green, being ground; blue, being score-up; orange, being end game; and red, being either player death or a player position reset—depending on the context the user gives the VEX brain. To insert the map, the user must first remove the black stabilizers at the bottom, then lift up the top tensioner and insert the map. To start up TKOM, the user must hit the checkmark button after inserting the battery into the brain. Then the user inputs their desired game functionality. Once TKOM is started up, the user can interact with the player and background by using the VEX controller. If the left front button is pressed, the player motor will move the player hand up and down. If the left control stick is moved left/right, the two rollers will pull/push the paper background forward/back due to the rollers being tightened by the tensioner coming back down, allowing for the rotation to transfer from the roller to the paper. During the game, the player arm will detect colour, deciding the motion that the player motor will do. If the bumper is pressed or if the colour orange is detected, the game will stop, and the VEX brain will display the score.

B. Components

B.I Player-Arm

B.I.a General Arm Design

One of the key components of any videogame is the main character. To translate this to our project, we decided on using a thin plastic plate (Player-Arm) connected to a motor, where the character is represented as the end of the plate closest to the center of our assembly. The rotation of the motor results in motion of the Player-Arm in a vertical motion in front of the paper belt.

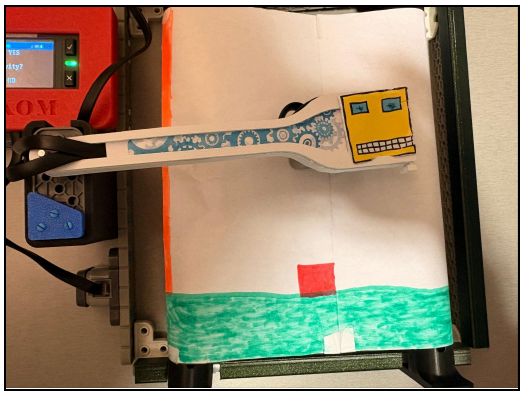


Fig. B.I.a Player-Arm

B.I.b Horizontal Motion Compensation

A major concern during the design was maintaining purely vertical motion of the character, contrary to a motor's rotational motion. Solutions were proposed such as using a rack and pinion gear mechanism, or a belt with roller system, however a design was settled on where the motor moves the Player-Arm in an angular fashion. This decision was made in order to simplify the overall design, decreasing design time and improving reliability. In order to compensate for the horizontal component of the rotational motion, the paper belt shifted forward and backward based on the horizontal component of the Player-Arm velocity at any given time. The rotation compensation should be of a larger magnitude at the extremes of the character motion due to the tangential motion of the Player-Arm being less aligned with the vertical direction. In order to accurately calculate this compensation, a calibration sequence was performed at the start of each game where the Player-Arm swung down until it hit a known obstacle. Since the location of this obstacle is known, the Player-Arm position angle is known, and so motor encoders can be properly calibrated.

B.I.c Visual Appeal

Another major concern of the design was aesthetic. Since the Player-Arm extended over the game background, it was important to make it visually appealing. The main way this was done was by introducing a thin slot for a paper drawing to be inserted into the Player-Arm upper surface. This drawing could include a design for the main character, as well as a background trail or design behind the character to help mask the rest of the arm. Another technique used to mask unappealing engineering was running the wiring for the sensors inside of the character arm, hidden from the view of the user. These wires ran to distance and color sensors mounted on the end of the Player-Arm.

Fig. B.I.c.1 Player-Arm Sensor Mounting

The color sensor was used for main gameplay functionality, such as detecting the ground in a game, and so was mounted directly below the character face to create an accurate “hitbox.” The distance sensor was used for detection if a game was loaded into the system, and so its position was not as crucial and it was mounted closer towards the vex brain along the Player-Arm. In order to have the arm blend into the paper below it, a few design considerations were used. Firstly, the arm was designed to be as this as possible in the plane of the paper belt in order to minimize obstruction of the game background. A size was chosen which was barely wide enough to house the wires from the two sensors while still maintaining structural integrity of the arm.

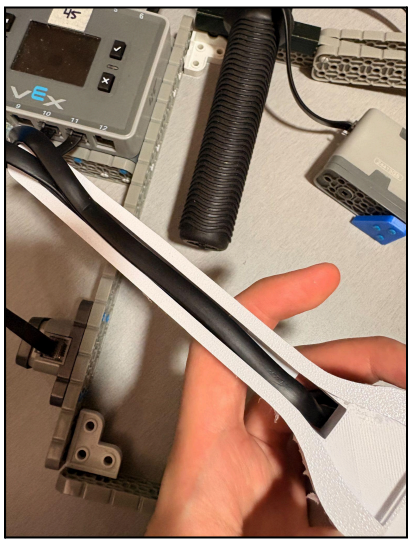


Fig. B.I.c.2 Player-Arm Hidden Wires

Another design decision was to make the arm shape use splines to create a smoother look. Since the part was to be 3D-printed, rather than machined, it was not a constraint to have the part be easily machined. This spline-based design made the arm blend into the background better and avoid casting harsh shadows which may interfere with the gameplay. This arm was also made out of a white plastic, making it blend into a paper background better.

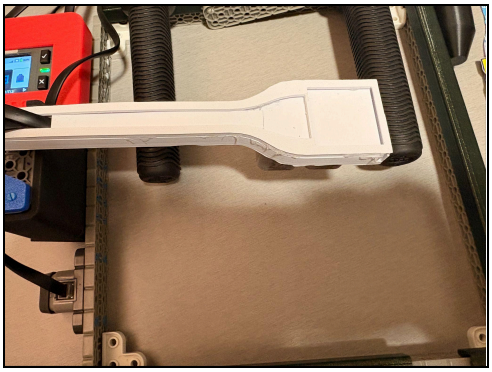


Fig. B.I.c.3 Empty Player-Arm

B.I.d Other Considerations

Some other considerations involved the thickness used for the Player-Arm. Making the arm too thin could result in rapid unplanned disassembly, especially under the accelerations of gameplay and rough play from young users. However, making the arm too thick would result in heavy weight which the motor would not be able to support and control. An iterative estimation approach was used to determine the appropriate dimensions. Another consideration was manufacturing. All parts were designed to be 3D-printed with standard FDM/FFF technology without the need of complicated assembly. Most parts were assembled through an interference-fit or with the use of standard vex pins, for easy disassembly for rapid prototyping.

B.II Background Roller

B.II.a Design/Role of Background Roller

The background roller acted as a medium between the motor and the paper background, in which power would be transmitted to move the paper left or right in response to the user input. To be able to power the drivetrain for the background there required high static friction, thus the rubber bicycle grips, refer to fig. B.II.a.1, were chosen to be the contact piece for the paper.

Fig. B.II.a.1 Bicycle Grip Rollers

To interface the bicycle grips with the motor, a 3D-printed end component was designed, as seen in fig B.II.a.2. A transition fit was used between the motor shaft and an interference fit was used with the bicycle grip, allowing for the motors output rotation to transfer to the grip and then the paper background.

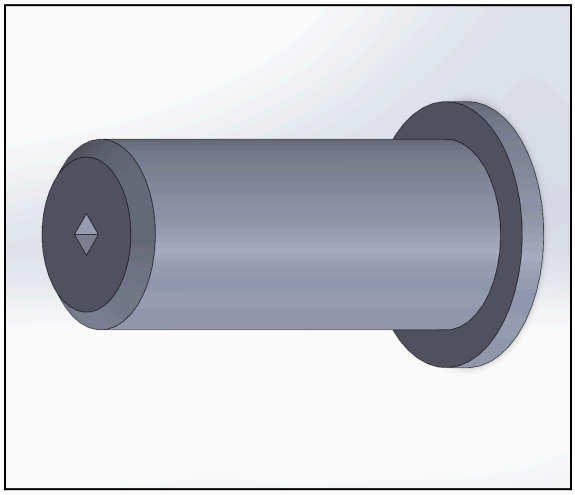


Fig. B.II.a.2 Bicycle Roller Insert

B.II.b Functionality

Throughout the initial phase of development in the project, the rollers worked perfectly, but as backgrounds were swapped back and forth we noticed that there would be issues with the rollers and the contact at the point where the background would be taped. Considering the limitation of the spacing of vex inserts and our mounting solution, we understood this to be both an oversight in our design and a limitation of vex placement.

B.III Top Tensionor

B.III.a Design/Role of Top Tensioner

The initial design for the top tensioner to utilize was proposed to be either a spring loaded system, or a top down insert. But to lower complexity and for user ease, a simple elastic tensioner was used. The components of it being an elastic band, two shafts, a free axle, and four plates. The band goes around the two shafts, which are both then joined to their two respective plates. These two plates are then joined back together by the free axle, which itself is on the main chassis of the robot. The user interacts with this piece when placing the paper background into the robot, shown in fig B.III.a.1.

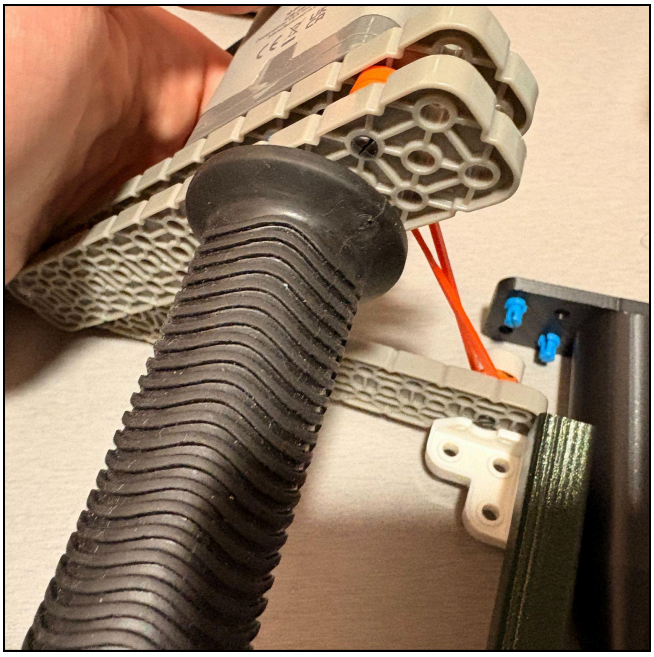


Fig. B.III.a.1 Top Tensioner Mechanism

B.III.b Functionality

Due to the simplicity of the design, the mechanism worked easily in isolation; but, in practice, this system tended to be difficult. This system is quite easy to use, in theory, as all the user has to do is lift up the tensioner to release tension as it shortens the distance between the two background rollers, allowing for the user to insert the paper background. In practice, the player arm gets in the way and is awkward, as you would want to insert the paper with two hands, yet one is being used to hold the tensioner up, this being an oversight from our team.

B.IV Bottom Tensionors

B.IV.a Role of Tensioner and Design Iterations

The role of the bottom tensioners was to provide stability to the paper belt as it moved back and forth. Our group initially planned on mimicking the top tensioner with the rubber grips but on a free axle rather than a motor; This proved challenging due to the instability it brought as this component would be relatively heavy and harm the amount of rotations which would be able to be transferred to the paper. The initial solution was to instead 3D-print a vex connector compatible tensioner. This would be inserted into the paper belt from the bottom of the chassis, after it was already tensioned from the top. This worked well as the static friction of PLA basic is significantly less than that of the bicycle grips, allowing for there to be little interference from the friction of the static tensioners. But, due to the static nature of the tensioner, the paper belt would move down and start tearing over the course of the game. To solve this issue, we redesigned it with a chamfer for the paper to rise up onto. This was originally designed to have a modular chamfer piece, refer to fig. B.IV.a.1.

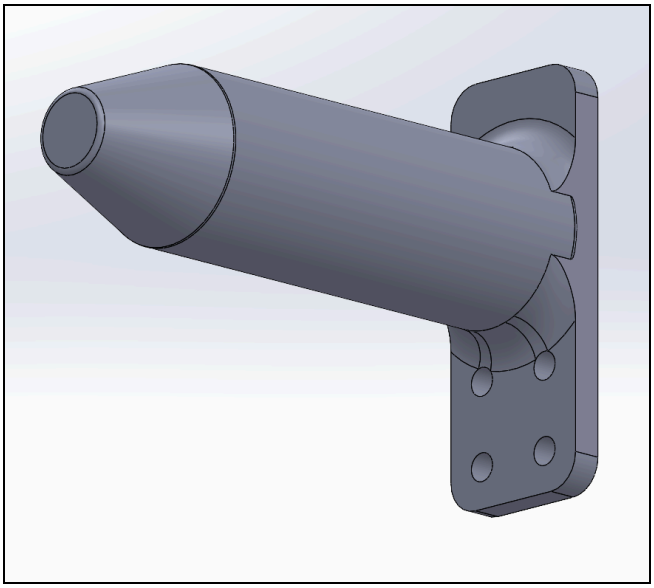


Fig. B.IV.a.1 Intermediate Design of Tensioner

This allowed for the user to set how compressed the paper was vertically. Due to it being modular, the piece would move left and right due to the paper constantly making contact with it. This was taken into consideration in the final redesign which also implemented an easy to remove and put on gate and lock mechanism, refer to fig B.IV.a.2.

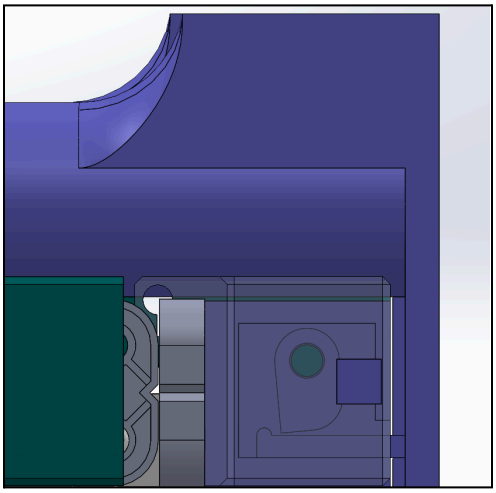


Fig. B.IV.a.2 Final Design of Tensioner

B.IV.b Final Design functionality

The final bottom tensioner utilized an insertion lock mechanism, this was done by making a three part component for the tensioner. One end would be the tensioner which interacted with the paper. But, the key change was that it wouldn't directly attach to the chassis, it instead had an extrusion below which would insert and lock into the second piece and third piece, refer to Fig. v.B.IV.b.1.

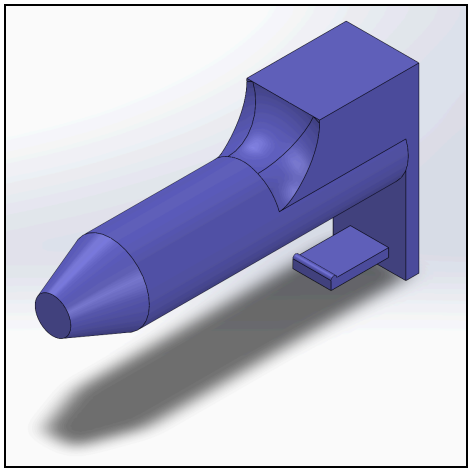


Fig. B.IV.b.1 Bottom Tensioner Final Design

The second piece was a box mount which would attach to the chassis; inside the mount there was a cut-out for insertion and an asymmetrical free axle gate, refer to fig. v.B.IV.b.2.

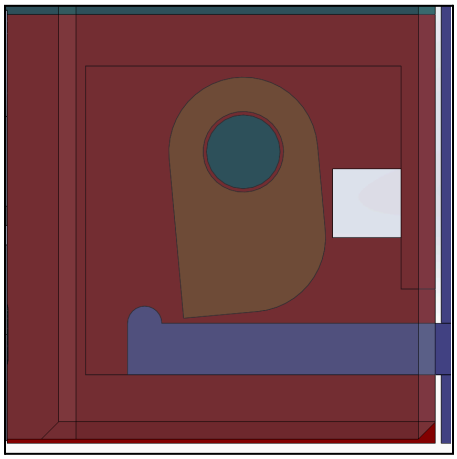


Fig B.IV.b.2 Box mount and Gate

The third piece was a stick with a widened top, it acted as the locking mechanism to the gate, refer to fig B.IV.b.3.

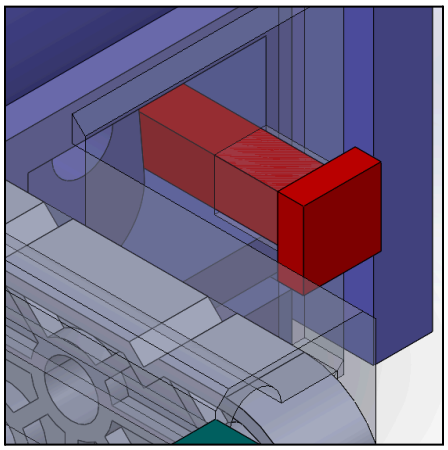


Fig. B.IV.b.3 Lock Insert

It worked by inserting piece 1 into piece 2 via the bottom extrusion into the cut-out, the extrusion would then push the gate back and slip through. After moving through, the gate would come back to being vertical due to gravity, then piece 3 would be inserted blocking the gate from moving toward the cut-out. Due to the asymmetry of the gate, the extrusion would get caught at the gate when piece 3 was in place, this allowed the system to be fully locked, Fig. B.IV.b.4.

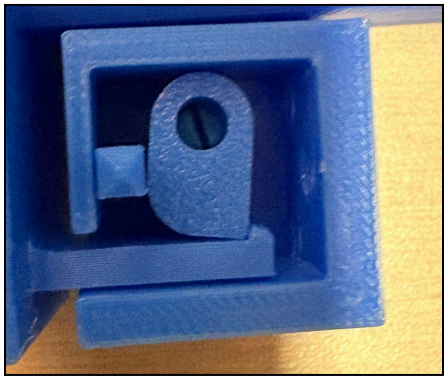


Fig. B.IV.b.4 Locked system

B.IV.c Success and failures

This final design had a key issue, it lacked radial stability, meaning once the background rollers started spinning the tensioners would be unstable and wobble. This proved problematic as the tension wouldn't be consistent causing the paper belt to struggle to move. The locking mechanism worked, but at the cost of the primary functionality. This issue wasn't able to be addressed before the demo, thus the previous design was utilized as it was a proven concept.

C. **Chassis Design**

C.I Motor mount and Position setter

A motor mount was designed for the Player-Arm. This was designed to both raise the motor and to provide it stability. It used a tolerance fit, allowing it to slide in and out easily, shown in Fig. 10.

Fig. C.I.a Mount and Position Setter

It maintained stability in the frame even when the player moved up and down. The raised height allowed for the player arm to be comfortably above the paper-belt, creating easier access for insertion and

removal of the paper-belt. Along with this piece was a blue position setter, visible in Fig 10, this acted as a mechanical constant for the Player-Arm to hit so it could recalibrate the origin position.

C.II 3D-printed part covers

The chassis was covered with 3D-printed covers, this allowed for it to be more engaging for the user while serving a structural purpose. The green covers on the top and the bottom, fig C.2.a, attached to the sides of the bottom tensioners and the top left motor, this was done so that they would be given support from the horizontal force which would be applied to them when the motors are running the belt to the left and right.

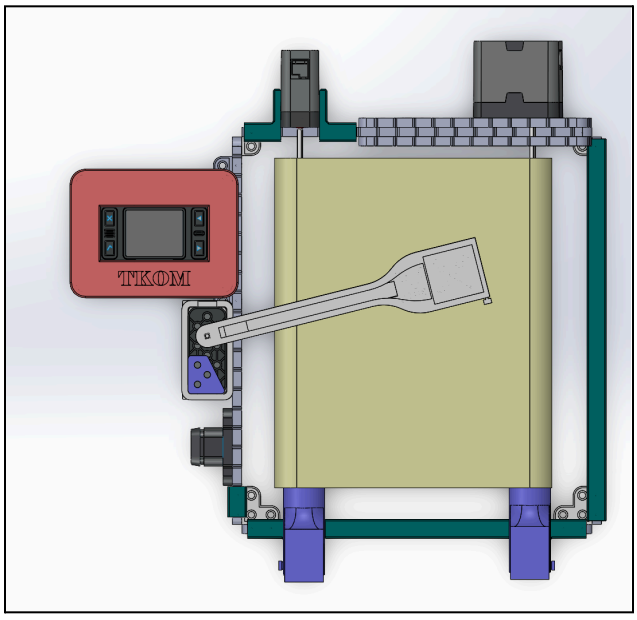


Fig. C.II.a Green Top and Bottom Covers

The top plate for the VEX brain, fig C.2.b, was done as a way to make the product more engaging for the user, make the user face the correct reading view due to the text TKOM, and it brought focus to the brain as a key component to engage with the product .

Fig. C.II.b Red VEX Brain Plate

C.III Corner Joints

Each end of the chassis was connected by a three by one white corner piece, fig C.3.a. This was done as it connected to both ends of the sides and would distribute force most evenly of the VEX parts, allowing for the structure to be rigid.

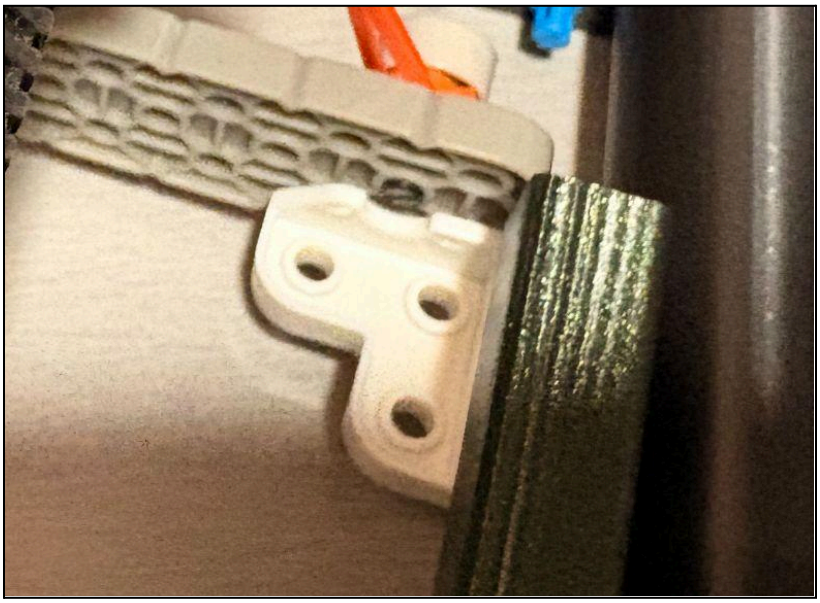


Fig. C.III.a White Corner VEX Component

C.IV Placement of bumper, VEX brain, and tensioners

The bumper was placed to the bottom left as it was most accessible to the user, imagining they're looking at the game from the reading view of the VEX brain. The VEX brain was placed above this to be just as accessible but also have closer proximity to the motors as to route the cables easily to the brain with little part wear. The tensioners were placed opposite to the drive train motors, which power the paper belt; this was done to allow for the tension to be the same from the top to the bottom, as seen in fig 14.

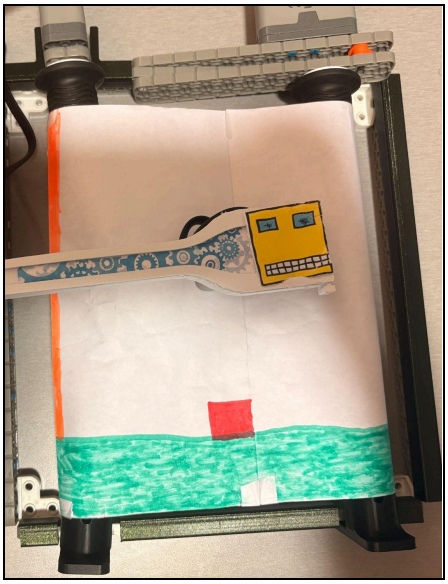


Fig. C.IV.a Top Down View

D. Motor Drive Design

D.I Placement of motors

The player motor was placed in the most centered area to allow for the Player-Arm's icon to sit in the centre, refer to Fig. 15. This allowed for easier origin calculations and created a standard for the user to create the game around. The drive-train motors were placed at the top at the distance of two pieces of paper in a hollow cylinder, allowing for an easy minimum size for the user to insert with ample tension, as seen in Fig. 3. The choice to place the drive-train on the top was so we could have games that focused on left right movement, we decided that there were more game opportunities that way compared to top down movement. The Player-Arm was purposefully placed perpendicular to the place which the drive-train was mounted as it allowed for x, y, and z axis motion for our player, allowing us to have left and right movement, player jump, and gravity, shown in fig 16.

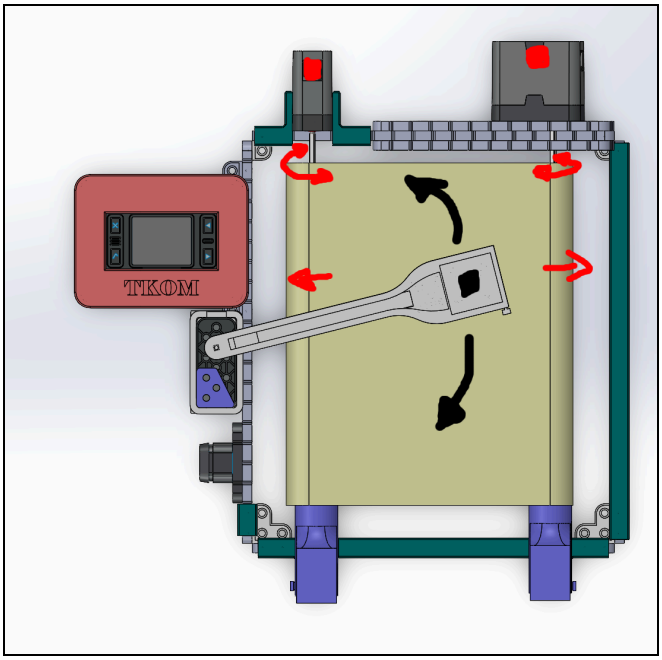


Fig. D.I.a Degrees of Motion

D.II Mobility of right motor

The right motor was placed onto a free-axle elastic tensioner. This was done to allow for the distance between the two drive train motors to decrease, allowing for easy insertion of the paper-belt, see fig. 3. Though this worked, it could have been more optimal as the user had to pull up the tensioner with one hand and insert the paper with the other, which proved to be difficult. Solutions were proposed such as a spring locking mechanism, but didn't go through due to time constraints.

V. Software Design and Implementation

To program the robot, a list of tasks was created. This was done by synthesizing the main idea of the robot, and breaking it down into smaller pieces. The main idea of the robot was as follows:

- STEP 1: Input parameters from users
- STEP 2: Spin Paper
- STEP 3: Check colours and adjust accordingly
- STEP 4: Check for Controller input and jump if button is pressed
- STEP 5: If player dies or bumper is pressed, stop spinning motors
- STEP 6: Output score for 5 seconds, then end program

However, since each task was generic, the code could not start being written yet. One main issue was player movement, as it was originally planned to simply spin the motor controlling the player (MotorPlayer) for a specific amount of time. However, due to its inaccuracy and difficulty to work with, a physics system was implemented instead. Additionally, most blocks needed helper functions to function.

Each task was then broken down and designed by a different group member, in order to be implemented into the code. All of the ideas were put together, to make the program one procedure. The final task checklist used for the demo was as follows:

Start Up and Initialization Phase:

- ☐ Hit start program button on Vex Brain
- ☐ Calls initparameters(askQuestions) on the brain until fully set up

Operation Phase:

- ☐ Speed of map is determined by the horizontal position of the left joystick + autoscroll constant defined by user upon startup
- ☐ If the left button up is pressed, the player will jump. User can define if player can jump once, twice, or infinitely many times
- ☐ When operating and not touching ground (green), the player will fall (go down automatically)
- ☐ When the player is off of paper, game end score displayed, after bumper hit, programStop
- ☐ When bumper hit mid game, game stop, score displayed, then after 5 seconds the program stops
- ☐ If playertouch red, the character either loses score, loses the game, or resets the stage. It outputs the score for 5 seconds and then closes.
- ☐ If player touches blue, score increase
- ☐ If player touch orange, gamestop, show score

These tasks were specifically designed to work well with the idea of a physics system in order to have smooth gameplay.

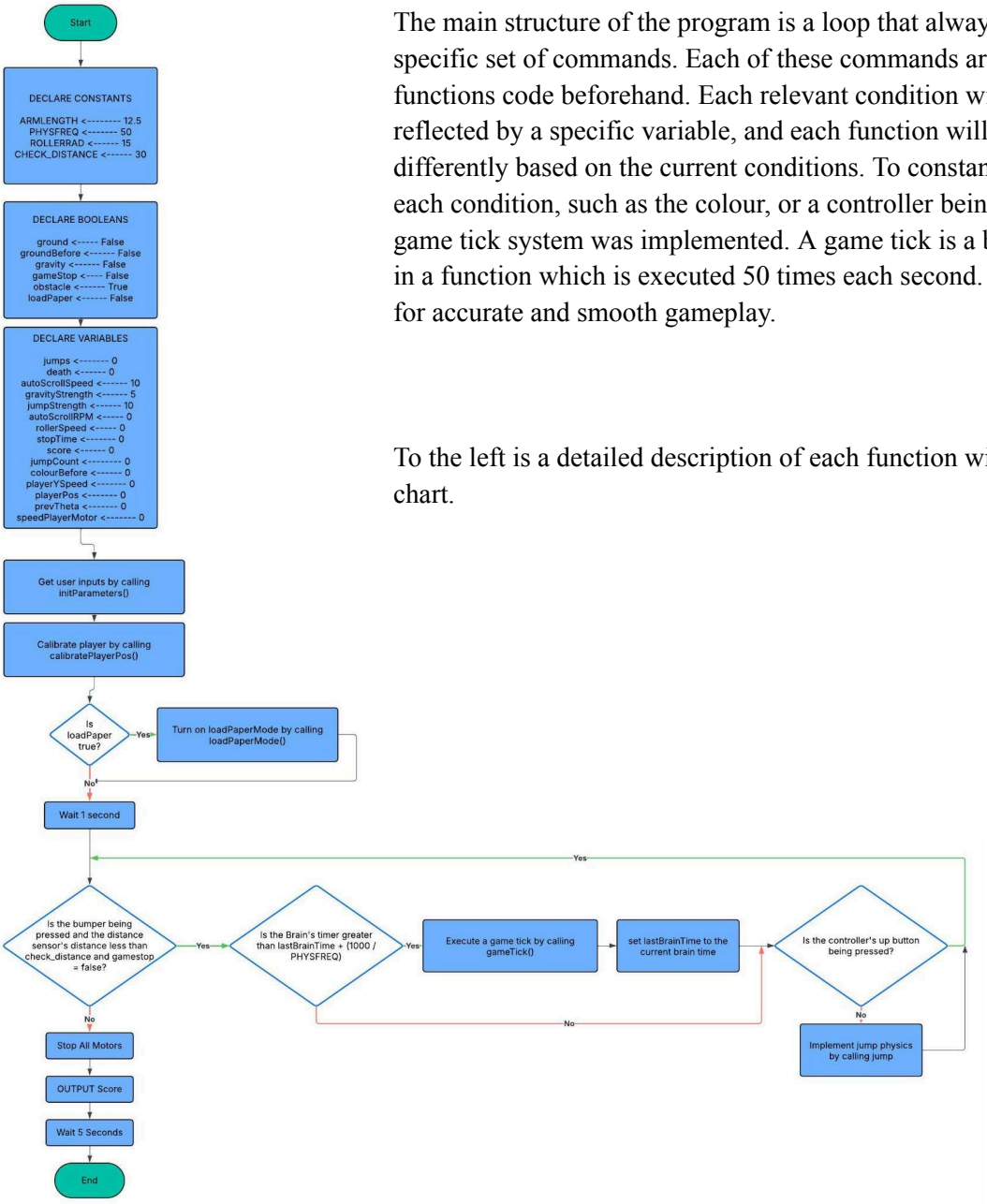
Shut Down Phase:

- ☐ While game running, if the bumper is hit, it exits to gamestop, then after 5 seconds of outputting score, it closes.
- ☐ If player falls off map -> gamestop -> then 5 seconds -> program stops
- ☐ Any other case where gamestop menu has come up

The choice of allowing the bumper to abruptly stop the game was to allow the user to smoothly use the robot without needing to wait for the finish line to be reached.

The full list of final blocks in the code were as follows:

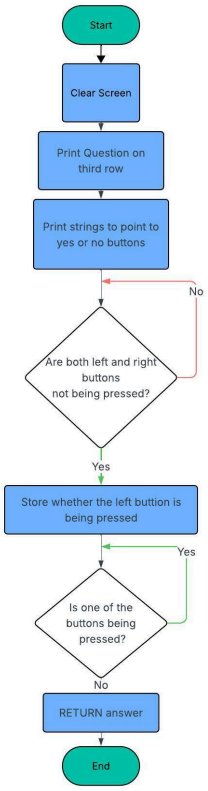
- AskQuestion()
- getValue()
- initParameters()
- colorCheck()
- initParameters()
- calibratePlayerPos()
- min()
- spinRollers()
- loadPaperMode()
- correctScrollSpeed()
- leftRightMove()
- upDownMove()
- deathAction()
- gameTick()
- jump()



Main Function Outline:

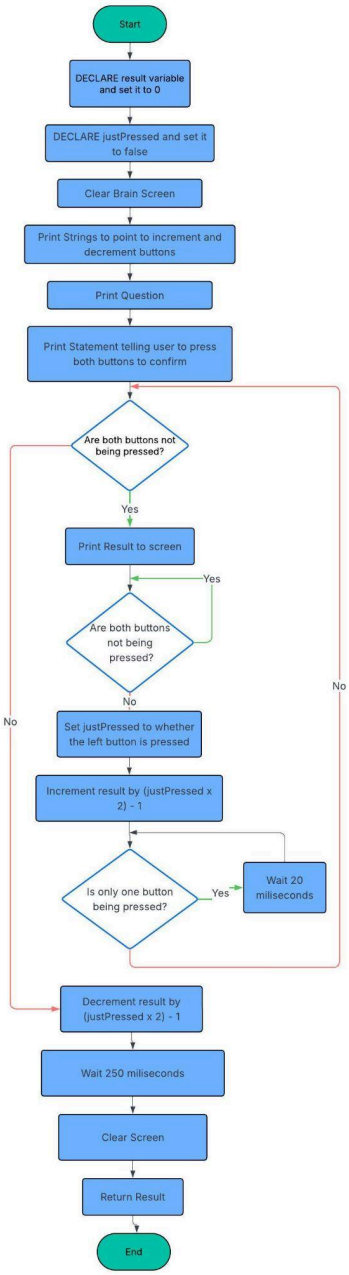
The main structure of the program is a loop that always runs a specific set of commands. Each of these commands are different functions code beforehand. Each relevant condition will be reflected by a specific variable, and each function will run differently based on the current conditions. To constantly check for each condition, such as the colour, or a controller being pressed, a game tick system was implemented. A game tick is a block of code in a function which is executed 50 times each second. This allows for accurate and smooth gameplay.

To the left is a detailed description of each function with a flow chart.



bool AskQuestion**(const char*** question**)**
Author: Ramy Wahib

This function is a helper function to format yes or no questions. It takes in a string literal as a parameter, since Vex IQ does not support strings as the parameter for a function. It clears the screen and formats the question, with a “YES” string pointing to the left button and a “NO” string pointing to the right button. Next, it waits until one of the buttons is pressed. When one button is pressed, it stores the state of whether the left button is being pressed in a boolean variable called “answer”. This is because one button was pressed to break out of the loop, and if it was the right button, then the user pressed “no”. Since the right button and the left button are always opposite booleans, grabbing the state of the left button is sufficient. It then waits for the user to release the button, and then returns the answer.



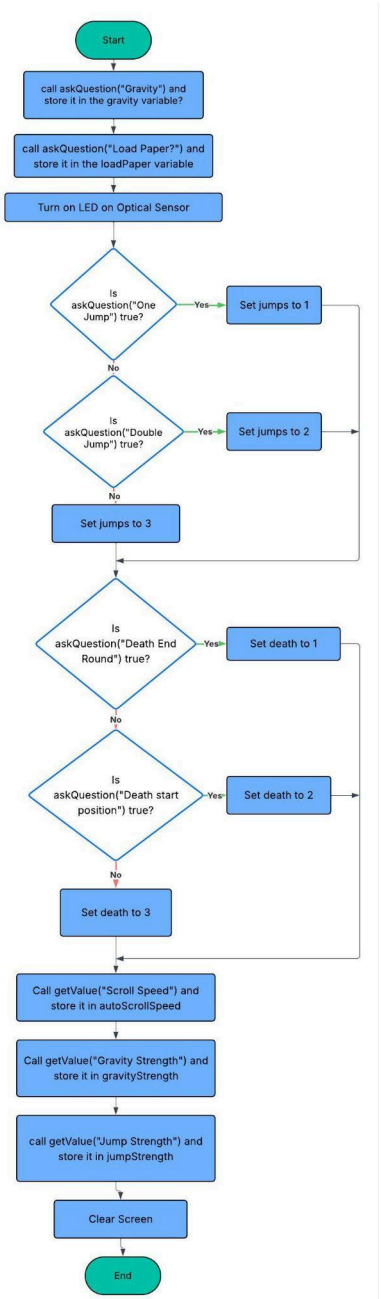
int getValue**(const char*** question**)**
Author: Farhaan Khan

This is another helper function with similar functionality to askQuestion. However, this is designed to format questions that request a numerical value, instead of a simple “yes” or “no”. It starts by declaring a result variable, which will be the value desired from the user. It also displays instructions for the user to input the value. While both buttons are not pressed simultaneously, it prints the result to screen. Another internal while loop checks if both buttons are not being pressed. This waits for the user to press a button. The loop is broken once a button is pressed. If the left button being pressed is true, its state would be 1, as C++ stores booleans as 0 or 1 [9]. Therefore, . This code keeps looping until both buttons are pressed simultaneously. This means that instead of creating a conditional to increment the result, it can just always be incremented by (2 * leftButtonState) - 1, as it would be 1 if it was true, and -1 if it was false. Once both buttons are pressed simultaneously, this code stops looping. However, it would be registered as a single press first, so the answer would be off by 1. To fix this, the result is decremented by (2 *

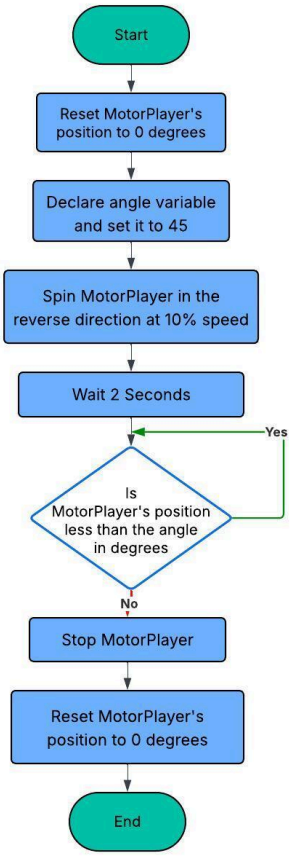
leftButtonState) - 1. Afterwards, the screen is cleared and the function returns the numerical value.

void initParameters**()**
Author: Viyasan Sribalamylvannan

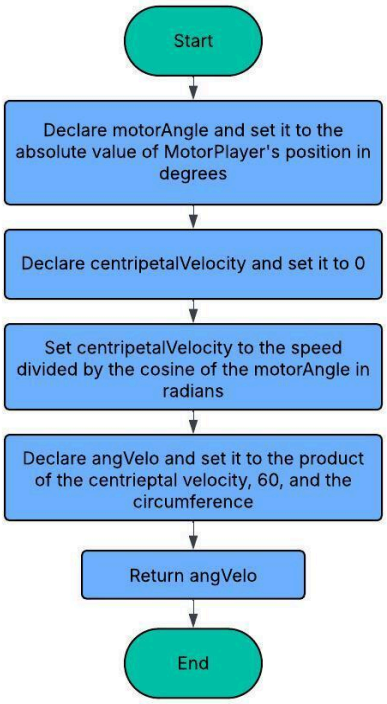
This function is used to initialize the variables needed for the rest of the program by inputting the values needed for the user. It is where both above helper functions are called. It starts by calling askQuestion for both “Gravity” and “Load Paper?” and storing them in the gravity and loadPaper variables, respectively. It turns on the LED for the optical sensor, which is more useful for later functions, yet it turns on here due to the functioning occurring early in the program. Afterwards, the function calls askQuestion for “One Jump” and checks to see if it is true or false. If it's true, it can go on to the next parameter and set the number of jumps to 1. However, if this is not the case, it can be either double jumps or infinite jumps. In this case, askQuestion must be called again for double jump. The result of this will decide whether the number of jumps will be 2 or infinite. If it is infinite, the variable jumps is equal to 3, yet 3 was designed to signify infinite. Afterwards, a similar process occurs with the result of hitting an obstacle. AskQuestion(“Death End Round”) is called, which is asking if hitting an obstacle should end the round. If this is false, the other two options are to reset the paper to the beginning of the map, or to simply decrease the score. askQuestion(“Death start position”) is called to determine this. The result of this will be stored in the death variable, where result 1 is the round ending, result 2 is the map resetting, and result 3 is the score decreasing. Finally, getValue is called three times, once for the scroll speed, once for gravity strength, and once for jump strength. Each value is stored in the global autoScrollSpeed, gravityStrength, and jumpStrength variables respectively. The final step is for the screen to be cleared.



void calibratePlayerPos**()**
Author: Ethan Margolin



Since a physics system was going to be implemented in the program, the motor's start position needed to be constant every single time the program runs. This function was designed to carry out that task. It starts by resetting MotorPlayer's position to 0. A constant integer called angle is declared and set to 45. This is purely to improve readability of code. 45 degrees is the angle between the bottom of the map and the center. MotorPlayer is then spun in reverse for 2 seconds. This will guarantee MotorPlayer to be at the very bottom of the map, and not more since the mechanical casing around the motor was designed to prevent that. Next, MotorPlayer is spun at 10% speed until its position reaches 45 degrees. MotorPlayer is then stopped. Since it is guaranteed that MotorPlayer is at the center of the map, MotorPlayer's position can now be reset to have an accurate position 0 for MotorPlayer for the rest of the game.



double speedToRPM**(double** radius, **double** speed**)**
Author: Farhaan Khan

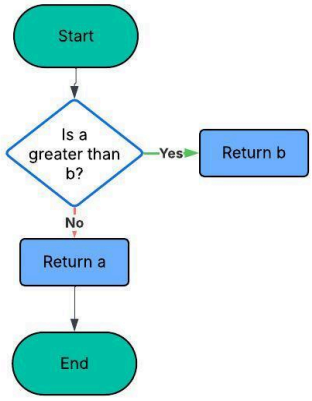
This function is a helper function that takes in the Y component of the speed of the player and converts it to RPM. It takes in two parameters: the radius of the fall and the current vertical speed of the player. It also declares a double variable called motorAngle and stores the absolute value of the player's position in it. Next, a double variable called centripetalVelocity, which is set to the Y component speed divided by the cosine of the motorAngle in radians. This is to resolve the Y component speed and figure out the centripetal velocity, which will be useful in later calculations. However, this speed is the linear velocity, yet angular velocity is needed. To convert it to angular velocity in RPM, the formula

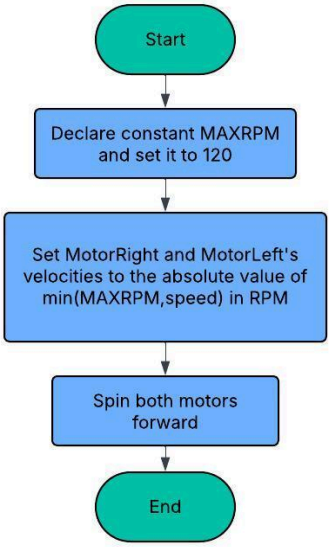
$$RPM = \frac{Linear\ Velocity \cdot 60}{2\Pi \cdot Radius}$$

is used [10]. The final variable to be declared is a double called angVelo, which stores the result of this formula when the parameters are input. Finally, the function returns the angular velocity.

double min(**double** a, **double** b)
Author: Farhaan Khan

This function is a simple helper function which aims to find the smaller number of two parameters. This will later be useful for input validation. It takes in two parameters, a and b, which are both doubles. If a is greater than b, it will return B, because it aims to return the smaller number. However, if b is greater, it will return a.



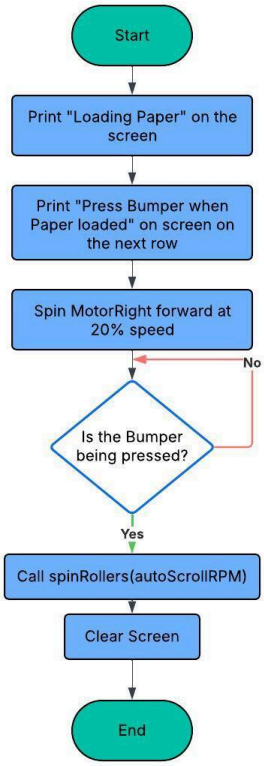


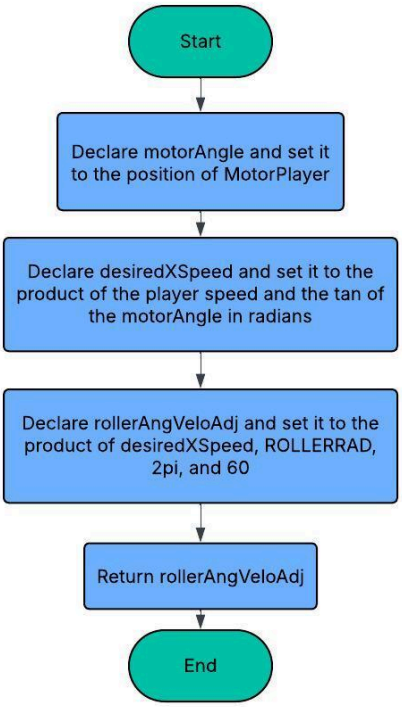
void spinRollers**(double** speed**)**
Author: Ethan Margolin

This function aims to spin MotorRight and MotorLeft to scroll the paper. It takes in the speed to spin it, which is stored as a double. A constant called MAXRPM is declared and set to 120. This is because VEX IQ motors cannot spin more than 120 RPM [11]. It then sets the speeds of both motors to the minimum of MAXRPM and the speed input. This is to make sure that if the speed is higher than 120, it will only set it to 120. Finally, it spins both motors forward.

void loadPaperMode()
Author: Ramy Wahib

This function helps the user load paper instead of making the user do it manually, in case the paper is long. It starts off by printing the instructions for the user on the screen. It then spins MotorRight forward at 20% speed, to allow the user to place the paper in order for it to roll around the Motor. While spinning, it constantly checks to see if the bumper is being pressed. Once the user is satisfied with the paper loaded, they can press the bumper, which will stop loadPaperMode. It will then clear the screen and start spinning MotorRight and MotorLeft normally to start the game.



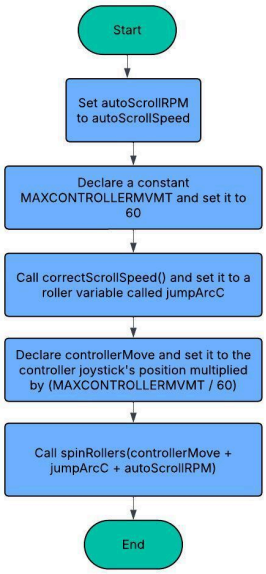


double correctScrollSpeed()
Author: Ethan Margolin

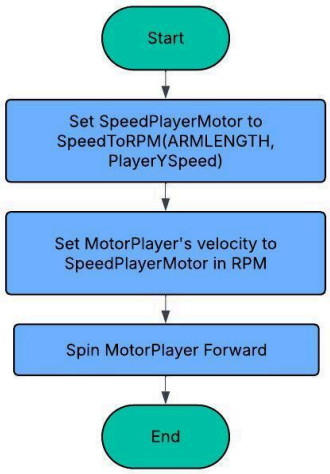
This function aims to smoothen the physics by taking the vertical speed of the player and using that value to return an appropriate speed to MotorRight and MotorLeft. This is so whenever the player jumps, the belt will compensate with the player jumping by changing its speed. Initially, a double variable called motorAngle is declared and set to the position of the player. Another double variable is declared and set to the horizontal speed of the player, which is the tan of the motorAngle in radians. The last double variable to be declared is rollerAngVeloAdj, which turns the horizontal speed of the player to the angular velocity of the motor. It is set to the desiredXSpeed multiplied by 60 and divided by the product of 2Π and the roller radius, ROLLERRAD. Finally, the function returns rollerAngVeloAdj.

void leftRightMove()
Author: Viyasan Sribalamylvannan

This function aims to calculate the correct speed to spin the motors with all the above factors taken into account, and spin the rollers with that speed. It starts by setting autoScrollRPM to autoScrollSpeed, to fix the value of autoScroll in RPM. It then declares another double called jumpArcC and sets it to correctScrollSpeed(). This is to add jump arc compensation to the final calculation. The final variable to be declared is a double called controllerMove, which is set to the controller joystick's position multiplied by 60 and divided by 100. This is so whenever the joystick moves by position 1, 0.6rpm is added to the speed. Finally, spinRollers() is called with the final speed of the sum of the above 3 variables to spin the motors at an appropriate speed that takes all conditions into account. This code was written with reference to [12].



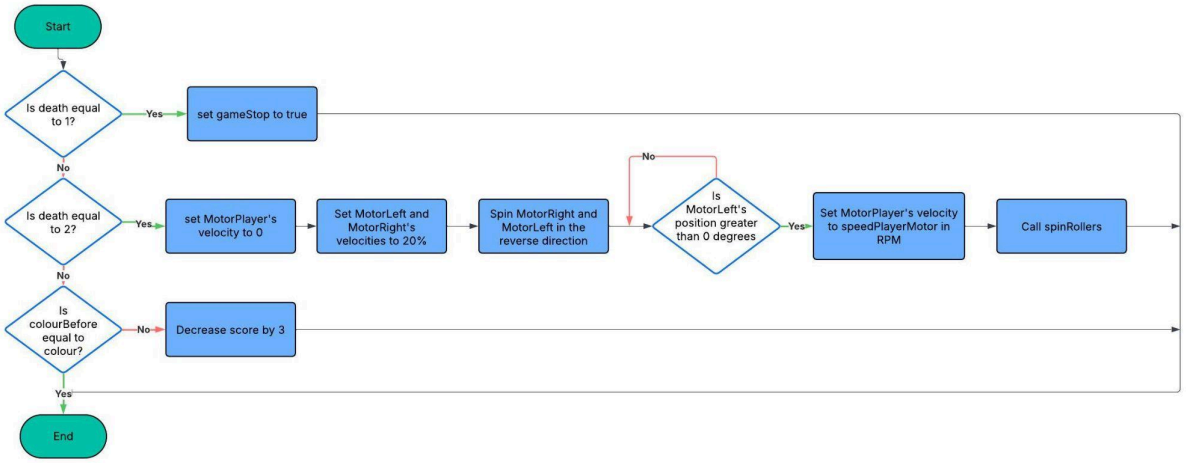
void upDownMove()
Author: Viyasan Sribalamylvannan



This function aims to update the vertical velocity of the player. It calls speedToRPM, inputting the arm length of the radius, stored in the constant ARMLENGTH, and the player'sYSpeed, to convert it to the correct angular velocity value needed to simulate this vertical velocity. It then sets MotorPlayer's velocity to this new angular velocity and spins MotorPlayer forward.

void deathAction**()**
Author: Ramy Wahib

This function aims to execute a specific action for hitting an obstacle depending on the input by the user. The value for the death variable was already set in the initParameters() function. If death is 1, the user wants the game to end when an obstacle is hit. This is done by setting gamestop to true, which will stop the main game loop. If death is 2, the user wants to reset the stage when an obstacle is hit. This is done by setting MotorPlayer's velocity to 0, to allow only the paper to move. Then, MotorLeft and MotorRight's velocities are set to 20% and are spun in reverse. While MotorLeft's position is not the same one it started with, it will keep spinning. Once MotorLeft's position is back to its initial one, the paper will be scrolled back to the beginning of the stage. Finally, MotorPlayer's velocity is set to the appropriate velocity in RPM and spinRollers are called to spin MotorLeft and MotorRight properly. If death is 3, it means the user wants the score to decrease. In this case, the score is decreased by 3.

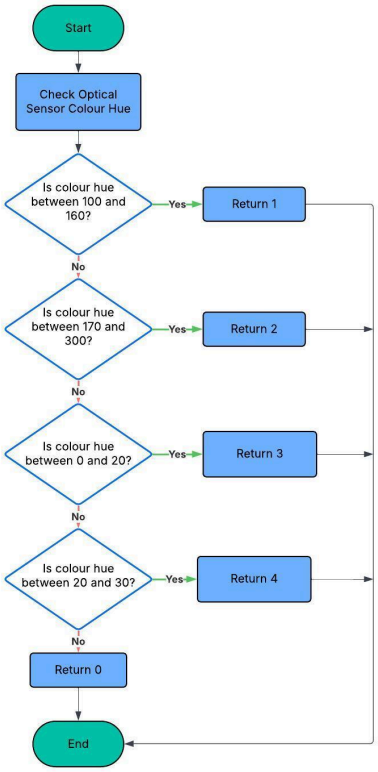


int colourCheck()
Author: Ramy Wahib

This is a helper function to allow other functions to know which colour is being detected. After testing the optical sensor, the following hue ranges were found:

HUE VALUES 100 - 160: GREEN
HUE VALUES 170 - 300: BLUE
HUE VALUES 0 - 20: RED
HUE VALUES 20-30: ORANGE

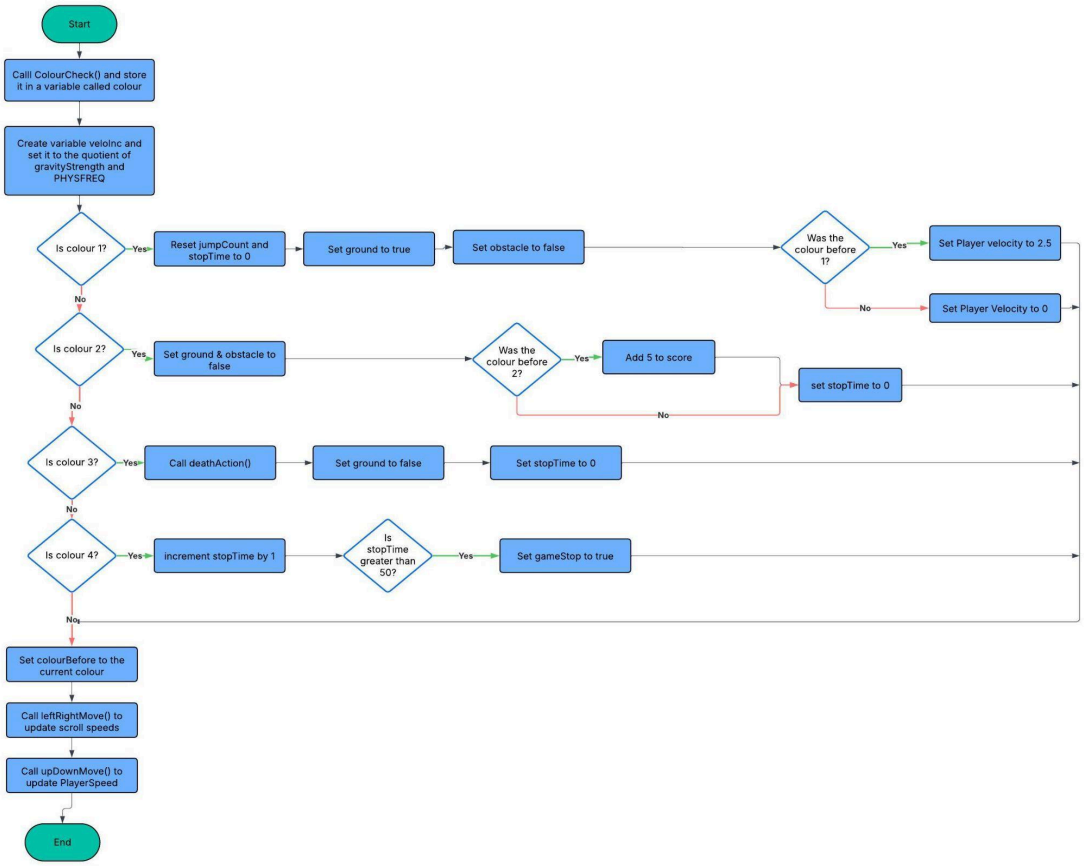
The function checks for each value. If it detects a hue value in the green range, it will return 1. If it detects a hue value in the blue range, it will return 2. If it detects a hue value in the red range, it will return 3. If it detects a hue value in orange range, it will return 4. Finally, if none of these colours are detected, it will return 0.



void gameTick**()**
Author: Farhaan Khan

This function aims to be carried out every game tick. This is to constantly update game conditions. Firstly, it calls colourCheck() and stores it in a new integer variable called colour. It also declares a new constant double variable called oscillationSpeed and sets it to 2.5. This is the vertical speed that the player will have when it is on ground. The reason it is not 0 is because if it did not go up while it was on ground, it would not be able to detect obstacles above ground. Another double variable called veloInc is declared and set to the quotient of gravityStrength and physFreq. This is the velocity that will be increased every tick. The full gravity strength needs to be implemented every second, so it's divided by PHYSFREQ, the number of ticks in a second, so each tick a specific velocity will increase, for smoother motion. Next, groundBefore to ground. If colour is 1, then it is green. Therefore, the ground variable is set to true, the obstacle variable is set to false. Additionally, jumpCount and stopTime are reset to 0. jumpCount will be explained further in jump(). stopTime will be explained when the colour is orange. While it is on ground, if groundBefore is false, then it sets the player speed to oscillationSpeed. It only does this if groundBefore is false because if it constantly sets the player speed while it is on the ground, the player would not be able to jump. If colour is 2, then it is on blue, which is a coin. Therefore, ground and obstacles are set to false. Since the score should only be decreased once every time it touches blue, it checks if in the previous tick, if the colour was blue. If it was not, then it just touched blue, therefore the score is increased by 5. stopTime is also reset to 0. If colour is 3, then it is on red, which is an obstacle. This time, it calls deathAction() to determine which action is appropriate. In the case that the user does not end the game, ground is set to false and stopTime is reset to 0. If colour is 4, it is on orange, which is the finish line.

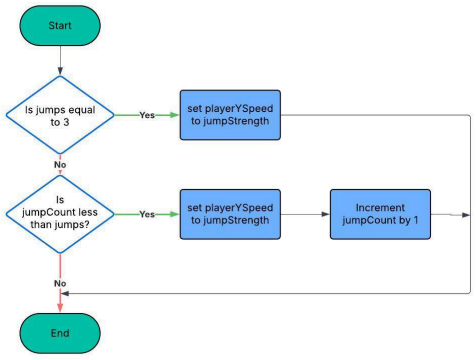
However, due to the weakness of VEX IQ optical sensors, orange was detected on unexpected parts of paper for a single tick. Therefore, stopTime increases with each tick of it being on orange. If stopTime reaches 50, then the program will know that it is on orange due to the finish line, and not due to inaccuracy of VEX IQ optical sensors. Therefore, it will set gameStop to true, ending the game. If it is on none of these colours, then normal air physics will be implemented through gravity. This means that both ground and obstacle will be set to false. After the colour checks, colourBefore will be set to the current colour so that it can be used in the next game tick. Finally, the motor speeds are updated if any of the conditions are changed. This is one by calling upDownMove() and leftRightMove() for every game tick.



void jump()

Author: Ethan Margolin

This function aims to make sure that the jump functionality is implemented smoothly, and the limit input by the user is set. Therefore, it first checks if jumps are equal to 3. If it is, it means the user requested infinite jumps. In that case, for every jump, the player's vertical speed will just be set to the jumpStrength input by the user. However, if there is a limit, it will check to see if jumpCount is less than the jump limit. If it is not, then the player's vertical speed will be set to jumpStrength and the jumpCount is incremented. This makes sure that if the maximum jump count is reached, no more jumps will execute.



Testing Procedures

Function	Tests ran + rationale	Expected program behaviour	Ensuring program behaviour
askQuestion()	To test askQuestion, its return value was stored in a variable in main, which was then displayed on the brain's screen. This would determine whether or not it was returning the correct value.	When the user pressed the "yes" button, the function should return true. Else it should return false.	The function worked on the first try, no extra effort required.
getValue()	The test procedure was similar to askQuestion, except the current value of the integer would be displayed on the brain's screen live. This way the user could see the updated value in real time.	As soon as the user lets go of the increment or decrement button, the screen should refresh to reflect the updated value.	Instead of timeout, the function would detect when the user releases the button to enable rapid triggering
initParameters()	Function was called exactly once in Main to see if all user-initialized inputs would be shown on display or not.	The brain should ask the user multiple yes/no questions, and ask for several numbers to be inputted. It should then begin the initialization sequence.	Care was taken to ensure every user-initialized value and any critical functions were called
colorCheck()	The function was called in an infinite loop in main. The output integer was converted to the appropriate color, which was displayed to the brain.	The brain should display the color of the object under the color sensor at any moment.	A range of hue values for each color was hard-written into the code, eg. red was 0 to 20.
calibratePlayerPos()	The function was called inside initParameters after all user-initialized variables were initialized	After initParameters had passed and just before the master loop began, the player arm should be perfectly parallel with the direction the paper rolled in	A rigid piece of plastic was attached close to the motor to ensure consistency across different runs
min()	The function was called in an empty main function	The brain should print the smaller of the two	The function worked on the first

	with two hard coded values, and its output was printed to the brain's screen	hard-coded values. For example if 5 and 15 were inputted into the function, the brain should display 5.	try, no extra effort required.
spinRollers(), leftRightMove() *(grouped together since one is helper function for another)*	Function was played in master loop after all values were initialized, and was tested with different autoscroll speeds and controller inputs to see how paper speed responded	Tested by inspection. Rollers should speed up as autoscroll speed was increased, and should slow down or speed up depending on controller input and if character arm was approaching or moving away from center line	Testing occurred parallel to others to see if any issues in movement of rollers would arise in different situations
loadPaperMode()	Function was called after initialization of variables to see whether or not game would immediately start in spite of function being called	When called, the function should wait and do nothing until the player presses the bumper	The function worked on the first try, no extra effort required.
correctScrollSpeed()	The value of this function is printed on the brain's screen after spinRollers() retrieves it	The value should grow as the player arm moves faster, and as it is further away from the centerline	The function worked on the first try, no extra effort required.
upDownMove()	playerYSpeed was printed to brain every tick, and rotational speed of player motor was observed	Player motor's angular velocity should correlate more or less linearly with playerYSpeed as determined by visual inspection	Player object's trajectory should roughly mimic projectile motion
deathAction()	During gameplay, the player object was purposely driven into the red (kill) objects to determine program behaviour upon death	The program should react to the red depending on what the user defined. For example, if the user wants red to return them to the start position, the game should pause until the rollers have rolled back to their original position	This function is called every game tick to minimize the chances the sensor would miss any red
gameTick()	An infinite loop was set up in main, along with a variable initialized to zero. Every time a tick should occur, the variable is incremented by 1, and that	The value should increase by about PHYSFREQ every second, as determined by inspection	The function worked on the first try, no extra effort required.

	updated value is printed to brain's screen to see how many total ticks have happened		
jump()	A live count of playerYSpeed was printed to the brain every game tick to see how jump() affected it	Every time the jump button was pressed, playerYSpeed should be reset to whatever jumpStrength is set to, unless the player is in mid-air and double/triple jump are both disabled	A variable tracking how many times the player has jumped was implemented to ensure they cannot jump infinitely many times midair

Table 4.1: Description of Testing Procedures of All Functions

VI. Verification

This section will discuss how TKOM met the constraints and criteria set upon it during the engineering design cycle. To recap, these constraints are the following:

A) The product should run reliably
B) The product should be user friendly
C) The product should have some sort of reward system

Criteria A

Criteria A was met by implementing several features that would correct any potential errors from the user or the machine. For instance, a distance sensor was installed next to the "player" color sensor that could detect if the rolling background has not been inserted into the rollers. If this is the case, then the program would stop and prompt the user to insert the paper.

Additionally, a custom 3d printed character arm was generated to maximize mechanical rigidity while minimizing weight, and by extension rotational inertia. By making the arm more rigid, the character would be less prone to sagging, reducing friction and interference with the paper roller. The reduction of rotational inertia would also make it easier for the motor to move the character, increasing the smoothness at which TKOM can play faster paced games.

Criteria B

Criteria B was implemented through guided instructions embedded in the input phase, and through a hot swappable paper roller design.

Guided instructions are displayed on the brain during the initialization to help the user input values that would dictate the rest of the game. For instance, when inputting integer values, the device displays arrows that indicate which buttons increment and decrement the value, and how to commit that value.

A hot swappable paper roller system was also implemented that would allow players to swap maps quickly. By disengaging one edge of the frame and pulling back one of the motorized rollers, the old map would have enough clearance for the player to simply slide it out and insert the new roller. A mechanism was also designed that would allow the stabilizers opposite from the motorized rollers to simply slide out after disengaging a pin. The mechanism was deemed to be too unsteady for the final iteration however, so it was exchanged for a rigid, fixed roller that was permanently attached to the frame.

Criteria C

Criteria C was met by implementing a coin and score system into the program. The program would start off at a score of zero, which would increment by 5 points every time the player touches a coin, or in this case, anything that is blue. A live count of the player's score is kept throughout the program, and is displayed at the end for 5 seconds either when the player's character dies, or the user decides to end the game. By implementing scores, players are incentivized to compete against themselves and their friends by progressively incrementing top scores.

Brief Redesign

To solve the sagging player arm, an extra 3d printed brace could be installed opposite the player motor to provide additional rigidity at the point of attachment. As for the paper stabilizer mechanism, rather than have a pin and swinging latch to hold the stabilizer in place, it would be simpler if the stabilizer had a clearance fit with the slot attached to the frame, and the two could be joined by a pin. This would allow for maximum rigidity while remaining modular and easy to take apart.

VII. Project Plan

In order to efficiently design our robot, tasks were distributed equally. Major tasks such as designing the general logic of our code, and general bot design, were done during group meetings. After each group meeting, smaller tasks were distributed as equally as possible across each member. If someone finished work early, or needed assistance with their task, additional meetings were scheduled and tasks were shared or redistributed. Whenever the group fell behind schedule additional group meetings were set up to get as much work done in as little time as possible.

In the original project plan, it was a goal to get a minimum viable product done by week one. This was mostly accomplished, with the exception of software and rollers for the paper belt. The goal of week two was originally to add some more advanced features, clean up the design, and create the majority of the software. The project plan was changed and instead the overall design was finalized, and basic software was created for testing. The goal of week three was to tune the design through testing. This was successful, and lots of changes were made to the robot based on testing and observations of issues. The goal of week 4 was to finish debugging, and work on visuals and the final demonstration. This was

successful, with the finalization of the code, addition of visual elements surrounding the robot, and the creation of templates for the paper belt and player arm drawings.

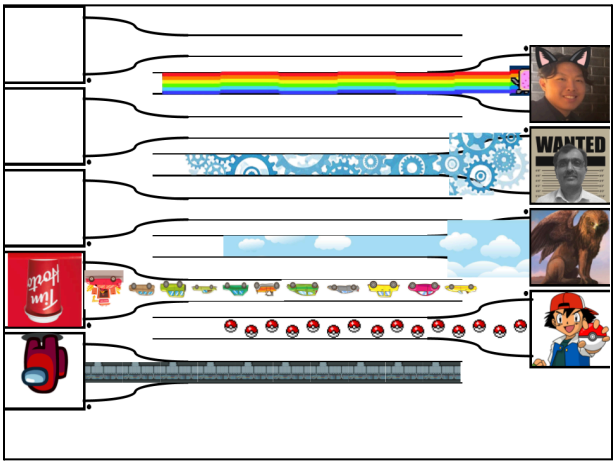


Fig. VI.I Main Character Template Page

Overall the project plan was followed, with the exception of the addition of some more advanced features. This was largely due to issues in reliability with the Vex IQ electronics provided.

VIII. Conclusions

In conclusion, the problem of encouraging users to draw more and use screens less was successfully solved in this project. The design was also capable of meeting the reliability constraint by careful mechanical and structural design. The user-friendly constraint was also met through guided instructions in the software side and hot swappable rollers in the mechanical side. Finally, the design was capable of meeting the reward system by implementing a score system in the program software to have the user feel rewarded after the game is over.

IX. Recommendations

This section will cover any changes that would be made to TKOM in hindsight, given the successes and shortcomings of the current product. Although the product's overall function is acceptable and somewhat robust, there are several changes that could have been made to further enhance reliability and ease of use by the end user.

Mechanical Changes

The biggest area of improvement when it comes to mechanical design would be the motorized rollers moving the paper back and forth. Due to mechanical flexibility, if the map inserted was too short, the rollers may bend inwards slightly and slowly push the paper out and towards the stabilizers. This would sometimes cause the rollers to jam, and the paper to move inconsistently. To solve this issue, the paper stabilizers could be connected to the motorized rollers to give them two points of contact on the frame, drastically increasing rigidity in the process. This would be quite challenging however, since the rollers need to be adaptive in length to account for variations in map sizes.

Additionally, the design for pin-engaged paper stabilizers, which will be referred to as the smart stabilizers, had too much give in the overall mechanism. Because of this, the decision was made to continue using rigid ones that were directly connected to the frame. The current design for the smart stabilizers could be improved by eliminating the swing arm, and instead having the stabilizer slide into a slot that would be directly attached to the frame. The two would have a clearance fit to allow for easy disassembly, and a pin would be used to attach the two, ensuring the stabilizers could not come out in the middle of gameplay.

Software Changes

One area of improvement with software could be the initialization of user-defined program variables, more specifically user-defined integers. Currently, returning an integer requires the user to use the buttons to increment or decrement 0 until they have their desired value.They must individually press the button every time they want to increment or decrement, which can become tedious if they want to enter a higher value. To solve this, a loop can be added that keeps incrementing a value if the user holds down a button for a set duration of time.

Appendix A: C++ Code

```
#pragma region VEXcode Generated Robot Configuration
// Make sure all required headers are included.
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include <string.h>


#include "vex.h"

using namespace vex;

// Brain should be defined by default
brain Brain;


// START IQ MACROS
#define waitUntil(condition)
do {
wait(5, msec);
  } while (!(condition))

#define repeat(iterations)
for (int iterator = 0; iterator < iterations; iterator++)
// END IQ MACROS


// Robot configuration code.
inertial BrainInertial = inertial();
motor MotorPlayer = motor(PORT5, false);
bumper BumperB = bumper(PORT9);
controller Controller = controller();
optical Optical10 = optical(PORT10);
distance DistanceSensor = distance(PORT11);
motor MotorRight = motor(PORT1, false);
motor MotorLeft = motor(PORT6, false);


// generating and setting random seed
void initializeRandomSeed(){
wait(100,msec);
```

```
43  double xAxis = BrainInertial.acceleration(xaxis) * 1000;
44  double yAxis = BrainInertial.acceleration(yaxis) * 1000;
45  double zAxis = BrainInertial.acceleration(zaxis) * 1000;
46  // Combine these values into a single integer
47  int seed = int(
48      xAxis + yAxis + zAxis
49    );
50  // Set the seed
51    srand(seed);
52  }
53
54  // Converts a color to a string
55  const char* convertColorToString(color col) {
56  if (col == colorType::red) return "red";
57  else if (col == colorType::green) return "green";
58  else if (col == colorType::blue) return "blue";
59  else if (col == colorType::white) return "white";
60  else if (col == colorType::yellow) return "yellow";
61  else if (col == colorType::orange) return "orange";
62  else if (col == colorType::purple) return "purple";
63  else if (col == colorType::cyan) return "cyan";
64  else if (col == colorType::black) return "black";
65  else if (col == colorType::transparent) return "transparent";
66  else if (col == colorType::red_violet) return "red_violet";
67  else if (col == colorType::violet) return "violet";
68  else if (col == colorType::blue_violet) return "blue_violet";
69  else if (col == colorType::blue_green) return "blue_green";
70  else if (col == colorType::yellow_green) return "yellow_green";
71  else if (col == colorType::yellow_orange) return "yellow_orange";
72  else if (col == colorType::red_orange) return "red_orange";
73  else if (col == colorType::none) return "none";
74  else return "unknown";
75  }
76
77
78  // Convert colorType to string
79  const char* convertColorToString(colorType col) {
80  if (col == colorType::red) return "red";
81  else if (col == colorType::green) return "green";
82  else if (col == colorType::blue) return "blue";
83  else if (col == colorType::white) return "white";
84  else if (col == colorType::yellow) return "yellow";
85  else if (col == colorType::orange) return "orange";
86  else if (col == colorType::purple) return "purple";
87  else if (col == colorType::cyan) return "cyan";
88  else if (col == colorType::black) return "black";
89  else if (col == colorType::transparent) return "transparent";
90  else if (col == colorType::red_violet) return "red_violet";
91  else if (col == colorType::violet) return "violet";
```

```
 92   else if (col == colorType::blue_violet) return "blue_violet";
 93   else if (col == colorType::blue_green) return "blue_green";
 94   else if (col == colorType::yellow_green) return "yellow_green";
 95   else if (col == colorType::yellow_orange) return "yellow_orange";
 96   else if (col == colorType::red_orange) return "red_orange";
 97   else if (col == colorType::none) return "none";
 98   else return "unknown";
 99   }
 100
 101
 102  void vexcodeInit() {
 103
 104  // Initializing random seed.
 105  initializeRandomSeed();
 106  }
 107
 108
 109  // define variable for remote controller enable/disable
 110  bool RemoteControlCodeEnabled = true;
 111  // define variables used for controlling motors based on controller
inputs
 112  bool eButtonsControlMotorsStopped = true;
 113
 114  // define a task that will handle monitoring inputs from Controller
 115  int rc_auto_loop_function_Controller() {
 116  // process the controller input every 20 milliseconds
 117  // update the motors based on the input values
 118  while(true) {
 119  if(RemoteControlCodeEnabled) {
 120  // check the ButtonEUp/ButtonEDown status to control MotorPlayer
 121  if (Controller.ButtonEUp.pressing()) {
 122          MotorPlayer.spin(forward);
 123          eButtonsControlMotorsStopped = false;
 124        } else if (Controller.ButtonEDown.pressing()) {
 125          MotorPlayer.spin(reverse);
 126          eButtonsControlMotorsStopped = false;
 127        } else if (!eButtonsControlMotorsStopped) {
 128          MotorPlayer.stop();
 129  // set the toggle so that we don't constantly tell the motor to stop whe
 n the buttons are released
 130          eButtonsControlMotorsStopped = true;
 131        }
 132      }
 133  // wait before repeating the process
 134  wait(20, msec);
 135    }
 136  return 0;
 137  }
 138
```

```
 139  task rc_auto_loop_task_Controller(rc_auto_loop_function_Controller);
 140
 141  #pragma endregion VEXcode Generated Robot Configuration
 142
 143
//----------------------------------------------------------------------------
 144  //
 145  // Module: main.cpp
 146  // Author: Ramy Wahib, Viyasan Sribalamylvannan, Farhaan Khan, Ethan
Margolin
 147  // Created: November 18, 2025
 148  // Description: TKOM robot
 149  //
 150
//----------------------------------------------------------------------------
 151
 152  // Include the IQ Library
 153  #include "iq_cpp.h"
 154  #include <cmath>
 155
 156  using namespace vex;
 157
 158  // ===== constants =====
 159  const double ARMLENGTH = 12.5;
 160  const int PHYSFREQ = 50;
 161  const double ROLLERRAD = 1.5;
 162
 163  // ===== player-initialized variables =====
 164  int jumps = 0;
 165  // 1 -> only one jump
 166  // 2 -> double jump
 167  // 3 -> infinite jumps
 168
 169  int death = 0;
 170  // 1 -> death ends round
 171  // 2 -> Death resets round
 172  // 3 -> Death decreases score
 173
 174  int autoScrollSpeed = 10; // rpm
 175  double gravityStrength = 5; // cm/s^2
 176  int jumpStrength = 10;
 177  double autoScrollRPM = 0;
 178
 179  // ==== booleans ====
 180  bool ground = false;
 181  bool groundBefore = false;
 182  bool gravity = false;
 183  bool gamestop = false;
 184  bool obstacle = true;
```

```
185 bool loadPaper = false;
186
187 // ===== internal variables =====
188 int rollerSpeed = 0;
189 int stopTime = 0;
190 double score = 0;
191 int jumpCount = 0;
192 int colourBefore = 0;
193 double playerYSpeed = 0; // cm/s
194 double playerPos = 0; // cm relative to centerline
195 double prevTheta = 0;
196 double speedPlayerMotor = 0;
197
198 // Asking the question
199 bool askQuestion(const char* question) {
200 // Reset screen
201 Brain.Screen.clearScreen();
202
203 // Formatting the way to ask question
204 Brain.Screen.setCursor(3, 1);
205 Brain.Screen.print(question);
206 Brain.Screen.setCursor(1, 1);
207 Brain.Screen.print("<-- YES");
208 Brain.Screen.setCursor(6, 1);
209 Brain.Screen.print("<-- NO");
210
211 // Wait for press
212 while (!Brain.buttonLeft.pressing() && !Brain.buttonRight.pressing()) {
213   }
214
215 bool answer = Brain.buttonLeft.pressing();
216
217 // Wait for release
218 while (Brain.buttonLeft.pressing() || Brain.buttonRight.pressing()) {
219   }
220
221 Brain.Screen.clearScreen();
222 return answer;
223 }
224
225 void nextStage(int spinSpeed) {
226 // Move roller 10 cm
227 while (MotorPlayer.position(turns) < 0.5) {
228   }
229 }
230
231 int getValue(const char* question) {
232 // Author: Farhaan Khan
233 // gets an integer intput from the user
```

```
234
235  int result = 0;
236  bool justPressed = false;
237
238  // format brain screen
239  Brain.Screen.clearScreen();
240  Brain.Screen.setCursor(1, 1);
241  Brain.Screen.print("<-- INC");
242  Brain.Screen.setCursor(2, 1);
243  Brain.Screen.print(question);
244  Brain.Screen.setCursor(4, 1);
245  Brain.Screen.print("press both to confirm");
246  Brain.Screen.setCursor(5, 1);
247  Brain.Screen.print("<-- DEC");
248
249  // while loop runs until user presses both buttons
250  while (!(Brain.buttonLeft.pressing() && Brain.buttonRight.pressing())){
251  // update screen
252  Brain.Screen.setCursor(3, 1);
253  Brain.Screen.print("value: %d     ", result);
254
255  // Wait for press
256  while (!Brain.buttonLeft.pressing() && !Brain.buttonRight.pressing()) {
257  wait(20, msec);
258      }
259
260  // increments if left is pressed
261  // decrements if right is pressed
262      justPressed = Brain.buttonLeft.pressing();
263      result += (int) justPressed * 2 - 1;
264
265  // Wait for release
266  // or both pressed
267  // uses xor operand "^"
268  while (Brain.buttonLeft.pressing() ^ Brain.buttonRight.pressing()) {
269  wait(20, msec);
270      }
271    }
272
273  // correct value
274  // if both were pressed, technically registered as a single press first
275  // value would be off by 1 otherwise
276    result -= (int)justPressed * 2 - 1;
277
278  wait(0.25, seconds);
279  Brain.Screen.clearScreen();
280  return result;
281  }
282
```

```
283 void initParameters(){
284 // Authors: Viyasan Sribalamylvannan
285 // Initializes user-initialized variables at beginning of game
286 // calles callibration function
287
288   gravity = askQuestion("Gravity?");
289   loadPaper = askQuestion("Load Paper");
290
291 // turn on led
292   Optical10.setLight(ledState::on);
293
294 // initialize jump variable
295 if (askQuestion("One jump?")){
296     jumps = 1;
297   }
298 else if (askQuestion("Limit to double jump?")){
299     jumps = 2;
300   }
301 else{
302     jumps = 3;
303   }
304
305 //initialize death variable
306 if (askQuestion("Death End Round?")){
307     death = 1;
308   }
309 else if (askQuestion("Death start position?")){
310     death = 2;
311   }
312 else {
313     death = 3;
314   }
315
316 // get integer inputs from user
317   autoScrollSpeed = getValue("Scroll speed?"); // rpm
318   gravityStrength = getValue("Gravity strength:"); // cm/s^2
319   jumpStrength = getValue("Jump strength?");
320
321 Brain.Screen.clearScreen();
322 }
323
324 int colorCheck()
325 {
326 // Author: Ramy Wahib
327 // 0 -> sentinel value
328 // 1 -> blue --> Coin
329 // 2 -> green --> ground
330 // 3-> red --> Obstacle
331 // 4-> orange --> FINISH LINE
```

```
 332
 333  if(Optical10.hue() > 100 && Optical10.hue() < 160) // Green
 334    {
 335  return 1;
 336    }
 337  else if(Optical10.hue() > 170 && Optical10.hue() < 300) // Blue
 338    {
 339  return 2;
 340    }
 341  else if(Optical10.hue() >= 0 && Optical10.hue() <= 20) // Red
 342    {
 343  return 3;
 344    }
 345  else if(Optical10.hue() > 20 && Optical10.hue() <= 30) // Orange
 346    {
 347  return 4;
 348    }
 349  else{
 350  return 0;
 351    }
 352  }
 353
 354  void calibratePlayerPos(){
 355  // Author: Ethan Margolin
 356  // moves playerMotor dwon for 2 seconds to reference position
 357  // reference position is same no matter where player motor is upon start
 358  // then adjust back to centerline and reset position
 359
 360    MotorPlayer.setPosition(0,degrees);
 361  int angle = 45;
 362    MotorPlayer.spin(reverse, 10, percent);
 363  wait(2,seconds);
 364    MotorPlayer.spin(forward,10,percent);
 365    MotorPlayer.setPosition(0,degrees);
 366
 367  while(MotorPlayer.position(degrees) <= angle){}
 368    MotorPlayer.stop();
 369
 370    MotorPlayer.setPosition(0,degrees);
 371  }
 372
 373  double speedToRPM(double radius, double speed){
 374  // Author: Farhaan Khan
 375  // converts total speed of an object attached to motor to angular
velocity
 376
 377  const double PI = 3.14159265;
 378  double motorAngle = abs(MotorPlayer.position(degrees));
 379
```

```
380  double centripetalVelocity = 0; // cm/s
381
382  // uses cosine since input is vertical component
383  centripetalVelocity = (speed / cos(motorAngle * PI / 180));
384  double angVelo = centripetalVelocity * 60 / (radius * 2 * PI); // rpm
385
386  return angVelo;
387
388  }
389
390
391
392
393
394
395
396  double min(double a, double b){
397  // Author: Farhaan Khan
398  // helper function to return smallest of 2 inputs
399
400  if (a>b){
401  return b;
402    }
403  else{
404  return a;
405    }
406  }
407
408  void spinRollers(double speed){
409  // Author: Ethan Margolin
410  // sets speed of motors depending on input
411  // negative input spins motor backwards
412
413  const int MAXRPM = 120;
414  if (speed >= 0){
415  // positive input; spin forward
416      MotorRight.setVelocity(min( MAXRPM, speed) ,rpm);
417      MotorLeft.setVelocity(min( MAXRPM, speed), rpm);
418      MotorRight.spin(forward);
419      MotorLeft.spin(forward);
420    }
421  else{
422  // negative input; spin backwards
423      MotorRight.setVelocity(min( MAXRPM, -speed),rpm);
424      MotorLeft.setVelocity(min( MAXRPM, -speed), rpm);
425      MotorRight.spin(reverse);
426      MotorLeft.spin(reverse);
427    }
428  }
```

```
 429
 430  void loadPaperMode()
 431  {
 432  // Author: Ramy Wahib
 433  // pauses program to allow user to insert paper
 434  const int MOTORSPEED = 20;
 435
 436  // output to brain screen
 437  Brain.Screen.setCursor(1,1);
 438  Brain.Screen.print("Loading Paper...");
 439  Brain.Screen.setCursor(3,1);
 440  Brain.Screen.print("Press Bumper");
 441  Brain.Screen.setCursor(4,1);
 442  Brain.Screen.print("when paper loaded");
 443
 444  // spin motor slightly forward
 445  // allow user to place paper for it to spin around motor
 446    MotorRight.spin(forward, MOTORSPEED, percent);
 447  while(!BumperB.pressing()) {}
 449    spinRollers(autoScrollRPM);
 450  Brain.Screen.clearScreen();
 451  }
 452
 453  double correctScrollSpeed(){
 454  // Author: Ethan Margolin
 455  // makes physics more accurate by correcting for unintended horizontal
 456  //    movement of player motor as it is rotational (polar)
 457
 458  const double PI = 3.14159265;
 459  double motorAngle = MotorPlayer.position(degrees);
 460
 461  // uses angle of arm and player's y velocity component to get x component
 462  // need to convert theta to tan too
 463  // cos(theta) = opp / adj = playerYSpeed / desiredXSpeed
 464  double desiredXSpeed = playerYSpeed * tan(motorAngle * PI / 180);
 465
 466  // turns horizontal speed into angular velocity of motor
 467  double rollerAngVeloAdj = desiredXSpeed * 60 / (2 * PI * ROLLERRAD);
 468
 469  // returns angular velo
 470  return rollerAngVeloAdj;
 471  }
 472
 473  void leftRightMove(){
 474  // Author: Viyasan Sribalamylvannan
 475  // helper function that collects all possible inputs controlling paper
roller mo
 vement
 476  // combines them and rolls motor at that speed
```

```
 477
 478  const int PI = 3.14159265358;
 479  const int MAXCONTORLLERMVMT = 60; // motor only accounts for half of
motor RPM
 480  // motor's max RPM is 120
 481  double jumpArcC = correctScrollSpeed();
 482  double controllerMove = (double) Controller.AxisB.position() *
MAXCONTORLLERMVM
 T / 100;
 483
 484    spinRollers(controllerMove + jumpArcC + autoScrollSpeed);
 485  }
 486
 487  void upDownMove(){
 488  // Author: Viyasan Sribalamylvannan
 489  // helper function that collects all possible inputs controlling player
arm movement
 490    speedPlayerMotor = speedToRPM(ARMLENGTH, playerYSpeed, false);
 491
 492    MotorPlayer.setVelocity(speedPlayerMotor, rpm);
 493    MotorPlayer.spin(forward);
 494  }
 495
 496  void deathAction(int colourBefore, int colour)
 497  {
 498  // Author: Ramy Wahib
 499  // handles different events that may occur if player dies (hits red)
 500  // specific event that occurs is decided by user in initParameters()
 501
 502  if(death == 1)
 503      {
 504  // if death stops program entirely
 505        gamestop = true;
 506      }
 507  else if(death == 2)
 508      {
 509  // if death resets player position
 510        MotorPlayer.setVelocity(0,percent);
 511        MotorLeft.setVelocity(20,percent);
 512        MotorRight.setVelocity(20,percent);
 513        MotorLeft.spin(reverse);
 514        MotorRight.spin(reverse);
 515  while(MotorLeft.position(degrees) > 0){}
 516        MotorPlayer.setVelocity(speedPlayerMotor, rpm);
 517        spinRollers(autoScrollRPM);
 518      }
 519  else{
 520  // death == 3
 521  // death decrements score
```

```
522 if(colourBefore != colour)
523       {
524 score -= 3;
525       }
526     }
527
528 }
529
530 void gameTick(){
531 // Author: Farhaan Khan
532 // Helper function that calls many critical functions for game operation
533 // occurs <PHYSFREQ> times every second
534
535 int colour = colorCheck();
536 double oscillationSpeed = 2.5;
537 double veloInc = gravityStrength / PHYSFREQ;
538
539 // update stats based on color
540   groundBefore = ground;
541
542 // handles behaviour based on color player is on
543 if (colour == 1) {
544 // Ground (blue)
545     ground = true;
546     obstacle = false;
547     jumpCount = 0;
548     stopTime = 0;
549 // set velocity to 0 if not on ground the tick before
550 if (groundBefore == false){
551       playerYSpeed = oscillationSpeed;
552       jumpCount = 0;
553     }
554   }
555
556 else if(colour == 2)
557 // Coin (green)
558     {
559       ground = false;
560       obstacle = false;
561 if(colourBefore != 2){
562 score += 5;
563       }
564       stopTime = 0;
565     }
566 else if(colour == 3) {
567 // Obstacle (red)
568       deathAction(colourBefore,colour);
569       ground = false;
570       stopTime = 0;
```

```
571      }
572  else if(colour == 4) {
573  // Finish Line (orange)
574      stopTime++;
575  if(stopTime >= 50)
576      {
577        gamestop = true;
578        ground = false;
579        obstacle = false;
580        playerYSpeed = 0;
581      }
582    }
583
584  else {
585  // White (empty background)
586      ground = false;
587      obstacle = false;
588    }
589
590  if (!ground){
591  // enables gravity if player is not on ground
592      playerYSpeed -= veloInc;
593    }
594
595    colourBefore = colour;
596
597  // update motor speeds
598    leftRightMove();
599    upDownMove();
600  }
601
602  void jump();
603
604  int main() {
605  // Initializing Robot Configuration. DO NOT REMOVE!
606  vexcodeInit();
607    initParameters();
608
609  // Calibrate position
610    calibratePlayerPos();
611  const int check_distance = 30;
612  int lastBrainTime = 0;
613
614  if(loadPaper)
615    {
616      loadPaperMode();
617    }
618
619  wait(1,seconds);
```

```
 620
 621  while(!BumperB.pressing() && DistanceSensor.objectDistance(mm) <
check_distance && gamestop == false)
 622    {
 623
 624  // update physics every N msec
 625  if (Brain.Timer.time() >= lastBrainTime + (1000 / PHYSFREQ)){
 626        lastBrainTime = Brain.Timer.time();
 627        gameTick();
 628      }
 629
 630  // jump
 631  if (Controller.ButtonEUp.pressing())
 632      {
 633        jump();
 634      }
 635
 637    MotorRight.stop();
 638    MotorLeft.stop();
 639    MotorPlayer.stop();
 640  Brain.Screen.setCursor(1,1);
 641  Brain.Screen.print("Score: %f", score);
 642  wait(5,seconds);
 643
 644  Brain.programStop();
 645  }
 646
 647  void jump() {
 648  // Author: Ethan Margolin
 649  // Handles logic when player presses jump
 650  // resets player's y velocity to jump strength to simulate jump
 651  // only does so if player still has jumps left
 652  // number of jumps possible depends on player input when initializing
 653
 654  if (jumps == 3){
 655  // infinite jumps
 656      playerYSpeed = jumpStrength;
 657    }
 658  else if (jumpCount < jumps){
 659  // not infinite jumps
 660  // value of jump correlates to how many jumps allowed
 661      playerYSpeed = jumpStrength;
 662      jumpCount++;
 663    }
 664  }
```

Appendix B: References

[1] O. Eric, “The negative effects of new screens on the cognitive functions of young children require new recommendations,” *Italian Journal of Pediatrics*, vol. 47, no. 1, Nov. 2021, doi: https://doi.org/10.1186/s13052-021-01174-6.

[2] H. E. Hulst *et al.*, “Cognitive impairment in MS: Impact of white matter integrity, gray matter volume, and lesions,” *Neurology*, vol. 80, no. 11, pp. 1025–1032, Mar. 2013, doi: https://doi.org/10.1212/wnl.0b013e31828726cc.

[3] C. Mukherjee, K. Cahill, and Mukesh Dhamala, “Action Video Gaming Enhances Brain Structure: Increased Cortical Thickness and White Matter Integrity in Occipital and Parietal Regions,” *Brain Sciences*, vol. 15, no. 9, pp. 956–956, Sep. 2025, doi: https://doi.org/10.3390/brainsci15090956.

[4] Shftrs, “The Importance of Quality Work: How Bugs Impact Customer Experience, Revenue, and the Bottom Line,” *Shftrs.com*, Feb. 26, 2024. https://shftrs.com/articles/the-importance-of-quality-work-how-bugs-impact-customer-experience-revenue-and-the-bottom-line (accessed Dec. 03, 2025).

[5] Y. T. (Tony) Chen, “Gamification in Marketing to Increase Customer Retention,” *dspace.mit.edu*, Jun. 01, 2023. https://dspace.mit.edu/handle/1721.1/151418

[6] J. R. Lewis and J. Sauro, “Effect of Perceived Ease of Use and Usefulness on UX and Behavioral Outcomes,” *International Journal of Human-Computer Interaction*, vol. 40, no. 20, pp. 1–8, Sep. 2023, doi: https://doi.org/10.1080/10447318.2023.2260164.

[7] Octavio, “Dopamine, social media and digital validation,” *NetPsychology*, Jun. 07, 2025. https://netpsychology.org/dopamine-social-media-and-digital-validation/

[8] L. Caroux and A. Mouginé, “Influence of visual background complexity and task difficulty on action video game players’ performance,” *Entertainment Computing*, vol. 41, no. March 2022, Nov. 2021, doi: https://doi.org/10.1016/j.entcom.2021.100471.

[9] A. Pomeranz, “4.9 — Boolean values – Learn C++,” *www.learncpp.com*, Feb. 04, 2025. https://www.learncpp.com/cpp-tutorial/boolean-values/

[10] “RPM to Linear Velocity Calculator,” *Cfm-calculator.com*, 2025. https://cfm-calculator.com/conversion/RPM-to-Linear-Velocity-Calculator.php (accessed Dec. 03, 2025).

[11] VEX Robotics, “Vex IQ Motor Maximum RPM,” *VEX Forum*, Jul. 21, 2019. https://www.vexforum.com/t/vex-iq-motor-maximum-rpm/63811 (accessed Dec. 03, 2025).

[12] VEX Robotics, “VEXcode IQ C++: vex::controller::axis Class Reference,” *Github.io*, 2025. https://johnholbrook.github.io/iqcpp-doxygen/classvex_1_1controller_1_1axis.html (accessed Dec. 03, 2025).