ME 102B: Kinetic Sand Table

Group 24

Rocklin Duong, Nilesh Kothari, Taylor Jazan, Rahul Kariyawasam

19th December 2024

1. Device Discussion

1.1 Opportunity

This project explores the creation of a motor-powered sand table, an engaging device that combines automation and user interaction for creating intricate patterns in sand. The opportunity lies in blending art and technology to develop a device that is functional, creative, and accessible. The sand table can serve as a captivating display piece, a creative tool, or an interactive installation, merging the worlds of technology and art in an innovative way.

1.2 High-Level Strategy

The sand table uses a dual-axis motorized system to move a steel ball over the sand's surface, guided by a magnet underneath. The control system includes an ESP32 microcontroller, two motors, a motor driver, a joystick, and limit switches. This design enables two operational modes: automated and manual.

From our initial desired functionality, we were able to accomplish nearly everything that we wanted. Though we were unable to implement online functionality, we did instead allow for manual control in both cardinal and diagonal directions.

1.3 Design

The design of the motor-powered sand table consists of a dual-axis system using linear rails and a timing belt mechanism. The motorized system allows for precise movement of a steel ball over a sand surface to create intricate patterns. Full size photos can be found in the appendix.

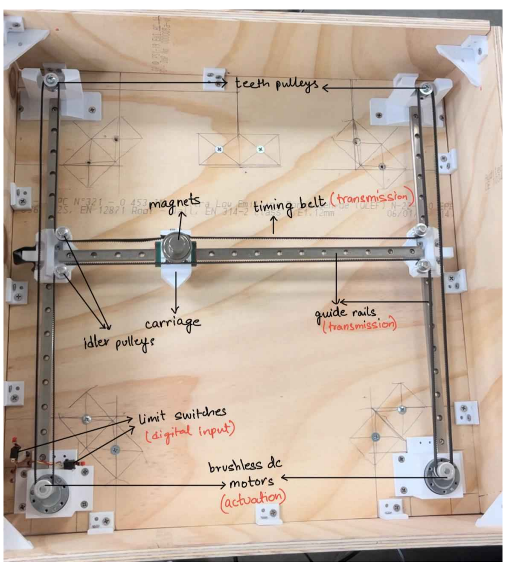

Figure 1: Photo of table with labels

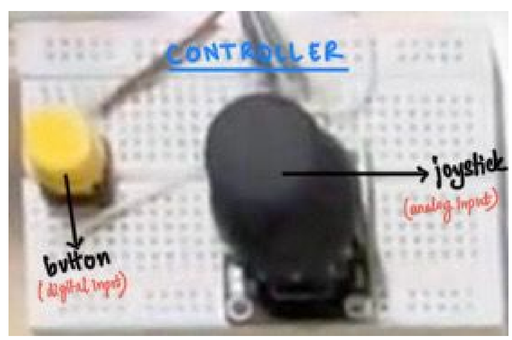

Figure 2: Photo of controller with labels

2. Calculations

In this section, we derive the key performance requirements and mechanical parameters for the linear rail and timing belt assembly used in the project. The system employs two brushed DC motors (for two degrees of freedom), a timing belt mechanism for translation, and ball bearings for load support. For simplicity and consideration of "worst case scenario," calculations are assumed for only one motor driving the assembly.

Note that these calculations do not take into account friction between the carriage(s) and the linear rails, or the friction due to the tension of the timing belt. However, these values are assumed to be negligible due to the use of quality ball bearings in the linear rail assembly, and a relatively low belt tension, respectively.

2.1 Required Torque for Each Motor

The required torque for each DC motor is determined based on the motor's maximum speed, the pulley radius, and the carriage acceleration. The system uses a pulley with radius $r = 6.5\,\text{mm} = 0.0065\,\text{m}$ and a motor shaft speed of $250\,\text{rpm}$ (revolutions per minute). The linear velocity v_{max} of the carriage is given by:

$$v_{\text{max}} = \omega \cdot r,$$

where ω is the angular velocity of the motor shaft in radians per second. Converting $250\,\text{rpm}$ to radians per second and plugging in:

$$v_{\text{max}} = 250 \times \frac{2\pi}{60} \cdot 0.0065 = 0.170\,\text{m/s}.$$

Assuming a rapid start-stop scenario (constant acceleration), the maximum acceleration a_{max} is estimated using kinematics:

$$a_{\text{max}} = \frac{v_{\text{max}}^2}{2L},$$

where L is the stopping distance. For simplicity, assume a realistic stopping distance $L = 0.01\,\text{m}$. Substituting values:

$$a_{\text{max}} = \frac{(0.170)^2}{2 \cdot 0.01} = 1.445\,\text{m/s}^2.$$

The torque T required at the motor shaft to achieve this acceleration is given by:

$$T = F \cdot r, \;\; F = m \cdot a_{\text{max}}$$

where F is the force required to accelerate the carriage. Given an assembly mass of $m = 0.071\,\text{kg}$, the required torque is:

$$T = F \cdot r = 0.071 \cdot 1.445 \cdot 0.0065 = 6.67 \times 10^{-3}\,\text{Nm}\,(6.67\,\text{mNm}).$$

This torque is well within the operating limits of the DC motors used, which have a maximum stall torque of 18 kg·cm, or 1.76 Nm.

2.2 Required Timing Belt Tension

To determine the required tension T_{belt} in the timing belt, we consider the force needed to overcome inertia during acceleration and account for belt elasticity. The tension in the belt can be approximated as:

$$T_{\text{belt}} = F_{\text{load}} + F_{\text{dynamic}},$$

where F_{load} is the force due to carriage assembly weight, and F_{dynamic} is the force due to acceleration:

$$F_{\text{load}} = m \cdot g = 0.071 \cdot 9.81 = 0.696\,\text{N}, \; F_{\text{dynamic}} = m \cdot a_{\text{max}} = 0.071 \cdot 1.445 = 1.026\,\text{N}.$$

The total belt tension required is approximately:

$$T_{\text{belt}} = F_{\text{load}} + F_{\text{dynamic}} = 0.696 + 1.026 \approx 1.722\,\text{N}.$$

2.3 Maximum Radial Loads on Ball Bearings and Motor Shaft

The system's ball bearings and motor shafts experience radial forces during operation due to belt tension and carriage movement. The maximum radial load on each ball bearing is approximately equal to the belt tension, as the force from the belt tension is transferred directly to the bearings.

$$F_{\text{bearing}} = T_{\text{belt}} = 1.722\,\text{N}.$$

The radial load on the motor shaft arises from the belt tension and the pulley geometry. The radial force F_{shaft} can be estimated using the belt tension and the belt angle of entry θ to the pulley. Assuming $\theta = 0°$ from our design (also a maximum load scenario):

$$F_{\text{shaft}} = T_{\text{belt}} \cdot \cos(\theta),\;\; F_{\text{shaft}} = T_{\text{belt}} = 1.722\,\text{N}.$$

The calculated loads are within the acceptable operating limits of our ball bearings and motor shafts, as the ball bearings of our system are rated for a maximum load of 200 N.

3. Diagrams

Full-scale photos of the below diagrams can also be found in the appendix.

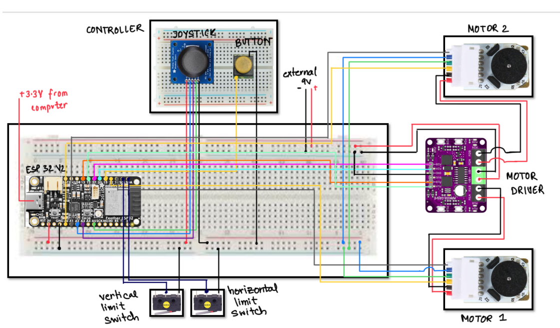

Figure 3: Circuit Diagram

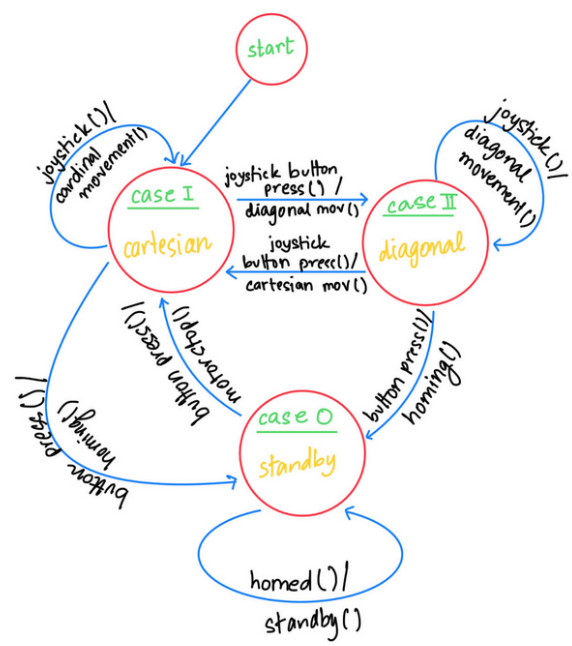

Figure 4: State Transition Diagram

4. Reflection

One strategy that worked well for our group was clear communication and task delegation. By setting realistic deadlines and holding regular check-ins, we maintained steady progress and avoided last-minute stress. However, we wish we had prototyped earlier in the process. Testing a mid-fidelity version of our design sooner would have helped us identify issues like motor strength and alignment earlier, saving time during the final build. Overall, balancing planning with hands-on testing is key to a smooth and successful project.

5. Appendix

5.1 Appendix A: Bill of Materials

								Total (Projected):	$ 251.83
Item Name	**Description**	**Purchase Justification**	**Serial Number / SKU**	**Price (ea.)**	**Quantity**	**Vendor**	**Link to Item**	**Notes**	**Subtotal**
DC Brushed Motors	43.8:1 Gear Ratio, 7A Stall Current, 18Kgf.cm stall torque, 16 CPR encoder resoltuion for motor shaft	Drives the timing belt and enables ball motion	FIT0186	$ -	2	N/A	https://www.dfrobot.com/product-634.html	Double check to see if it works properly. Do we have motor drivers for both? If yes, we can just use kit ones	$ -
Timing Belt Pulley	20 Teeth, 6mm bore, 2.03mm pitch, 6mm max belt width	Used to transfer motion from the motor to the timing belts	1375K119	$ 17.00	4	McMaster-Carr	https://www.mcmaster.com/1375K119	N/A	$ 68.00
Timing Belt Idler Pulley (Smooth)	5mm bore, 6mm max belt width	Used to hold the timing belt in place (is cheaper than teethed pulleys	3693N14	$ 9.06	4	McMaster-Carr	https://www.mcmaster.com/3693N14	N/A	$ 36.24
Timing Belt	2mm pitch, 6mm wide	Used to drive the motion of the magnet	B07XG9JN5B (ASIN)	$ 14.99	1	Amazon (Zeelo)	https://a.co/d/7cHCxBK	N/A	$ 14.99
Linear Slide Rail	400mm length, has scarriage block	Used for moving the magnet around	B09Z2D9LZT (ASIN)	$ 19.50	3	Amazon (iMetrix)	https://a.co/d/1tMzGAQ	N/A	$ 58.50
Wood	To be purchased and cut down/dimensioned into pieces of necessary size	Used to make the table	N/A	$ -	1	Home Depot	N/A	Will go in person to purchase at some point	$ -
Acrylic	Can also be cut down as necessary	Used to place on top of the sand to prevent sand from flying out/things from getting in	N/A	$ -	1	N/A	N/A	Will obtain from other people (Tom)	$ -
Sand	It's coarse and rough and it gets everywhere	The main ingredient for drawing patterns	B00J4YJ9HS (ASIN)	$ 7.99	1	Amazon (Be Good Company)	https://a.co/d/hfJLUFq	N/A	$ 7.99
Joystick	Voltage range of 3.3V to 5V,	Used for manual control of the magnet/ball	DFR0061	$ 5.30	1	DFRobot	https://www.dfrobot.com/product-349.html	N/A	$ 5.30
Neodymium Disc Magnet	5/8" diameter, 1/8" thickness	Sticks to the ball	DA2	$ 1.44	8	K&J Magnets	https://www.kjmagnetics.com/proddetail.asp?prod=DA2	8 Purchased in case we need more for "shimming"	$ 11.52
Neodymium Countersunk Magnet	32mm diameter, 6mm thickness	Screwed in to magnet carriage	B08VJ4KVBZ (ASIN)	$ 7.99	1	Amazon (DIYMAG)	https://a.co/d/ihnW4uJ	4 Pack	$ 7.99
Chrome Steel Bearing Balls	20mm diamter	Used to draw patterns in sand	B07Q3G7S9P (ASIN)	$ 8.79	1	Amazon (uxcell)	https://a.co/d/2ktCZXK	3 Per pack	$ 8.79
Potentiometers	10 kOhms	Used for circuitry	N/A	$ -	TBD	N/A	N/A	Have in kit	$ -
Breadboards	N/A	Used for circuitry	N/A	$ -	TBD	N/A	N/A	Have in kit	$ -
Resistors	Wide variety we can use from kit	Used for circuitry	N/A	$ -	TBD	N/A	N/A	Have in kit	$ -
Buttons	12mm diameter	Used for start/pause of drawing	1009	$ 5.95	1	Adafruit	https://www.adafruit.com/product/1009	15 Pack	$ 5.95
Motor Driver	N/A	Needed to drive the motors being used	N/A	$ -	2	N/A	N/A	Have in kit	$ -
M5-20 Screw	5mm diameter x 20mm length, flange bolt	Screws down the idler pulleys to the carriages	17543	$ 0.50	4	Bolt Depot	https://boltdepot.com/	N/A	$ 2.00
M6-20 Screw	6mm diameter x 20mm length, machine screws	Screws down the two extra teethed pulleys	5190	$ 0.32	2	Bolt Depot	https://boltdepot.com/	N/A	$ 0.64
M3-12 Screw	3mm diameter x 12mm length, machine screws	Screws down 3d printed parts to the carriages and the rails to the table	6833	$ 0.08	24	Bolt Depot	https://boltdepot.com/	N/A	$ 1.92
M3 Hex Nut	3mm diameter	Holds the center rail together	4773	$ 0.07	6	Bolt Depot	https://boltdepot.com/	N/A	$ 0.42
Wood Screws	#8 x 3/4" wood screw	Holds table together	3959	$ 14.48	1	Bolt Depot	https://boltdepot.com/	100 Per pack, unsure how many we'll need	$ 14.48
M6 Washer	6mm diameter	Evenly distrubtes pressure from screw onto teethed pulleys	4516	$ 0.06	2	Bolt Depot	https://boltdepot.com/	N/A	$ 0.12
Contact Switches	5 Amp current rating, 250V operating voltage	Used for homing	B08736NP44 (ASIN)	$ 6.98	1	Amazon (InduSKY)	https://a.co/d/icavpvx	10 Per pack	$ 6.98

5.2 Appendix B: CAD

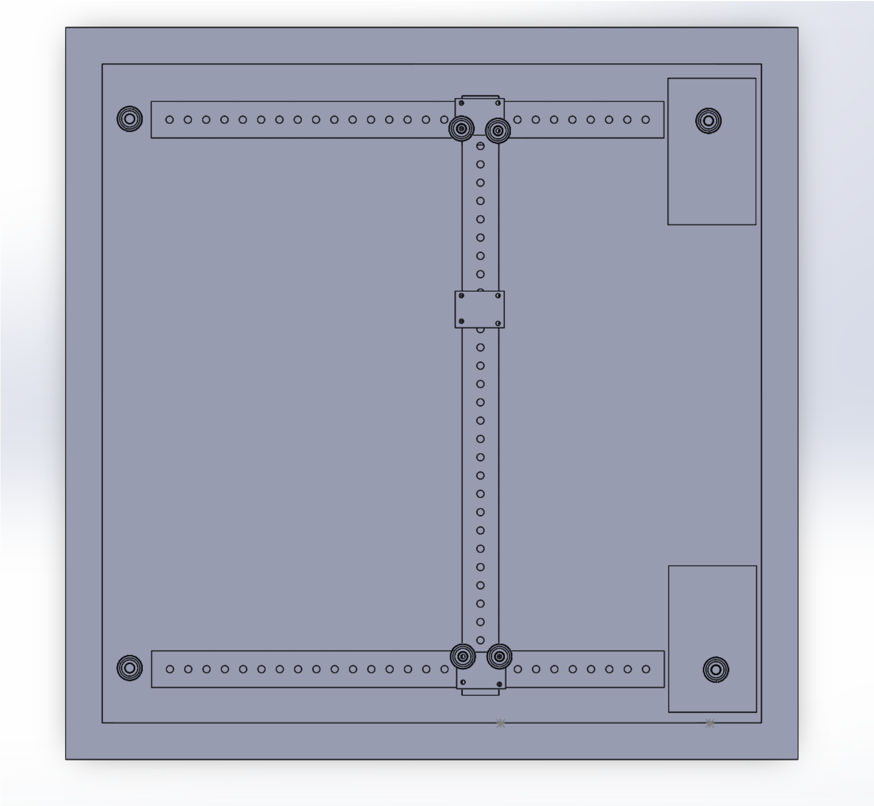

Figure 5: Top view of the sand table design showing the X and Y axes components.

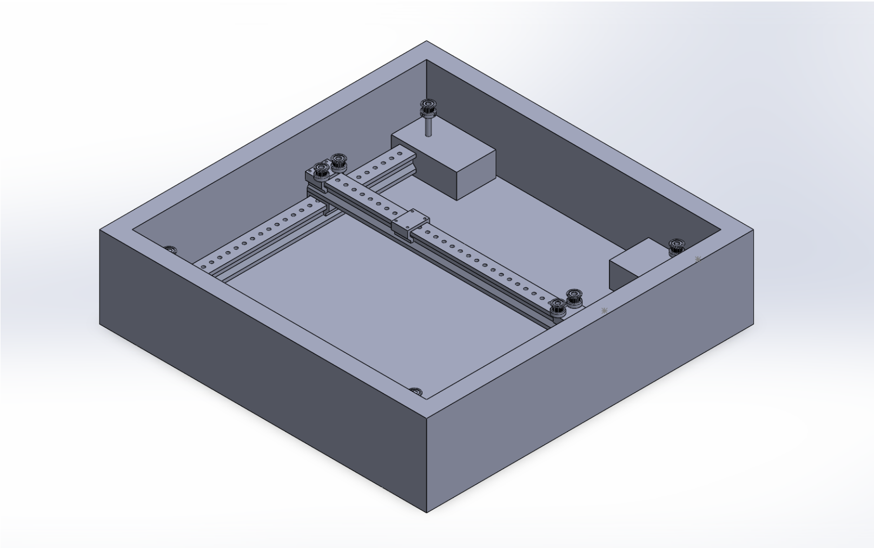

Figure 6: Isometric view of the sand table design, displaying the frame and carriage system.

5.3 Appendix C: Code Screenshots

The order of the screenshots go in order of Main.ino, then MotorControl, JoystickHandler, and EncoderHandler. For the last three classes, the .hpp file is shown then the .cpp file.

```cpp
#include <MotorControl.hpp>
#include <JoystickHandler.hpp>
#include <EncoderHandler.hpp>

// Define pin constants
const int manualButton = 15;    // Define button for manual control/pattern toggle
const int diagCardButton = 4;   // Define button for diagonal/cardinal directionality
const int xpin = 34;            // Define x-axis pin
const int ypin = 39;            // Define y-axis pin
const int ain1 = 12;            // Pin 1 for motor A
const int ain2 = 13;            // Pin 2 for motor A
const int bin1 = 33;            // Pin 1 for motor B
const int bin2 = 27;            // Pin 2 for motor B
const int AAPin = 32;           // Encoder A pin for motor A
const int BAPin = 14;           // Encoder B pin for motor A
const int ABPin = 26;           // Encoder A pin for motor B
const int BBPin = 25;           // Encoder B pin for motor B
const int vlimButton = 20;      // Vertical limit switch pin
const int hlimButton = 22;      // Horizontal limit switch pin


int state = 0;

// Define booleans
bool toggleControl = true;      // Checks if button was pressed
bool toggleCardinality = false; // Checks if was last in cardinal control or diagonal control
bool homingSequence = false;    // Checks if the homing sequence is currently running
bool verticalLimit = false;     // Checks if the vertical limit switch was pressed
bool horizontalLimit = false;   // Checks if the horizontal limit switch was pressed
bool checkControl = false;      // Enables debounce and begins service
bool checkCardinality = false;  // Enables debounce and begins service

hw_timer_t* timer = NULL;

// Create class objects
MotorControl motCon(ain1, ain2, bin1, bin2);
JoystickHandler joyHan(xpin, ypin);
EncoderHandler encoderHandler(AAPin, BAPin, ABPin, BBPin);

// Encoder interrupt functions
void IRAM_ATTR enc1() {         // Encoder updater 1
  encoderHandler.handleEncoder1();
}

void IRAM_ATTR enc2() {         // Encoder updater 2
  encoderHandler.handleEncoder2();
}

void IRAM_ATTR manual() {       // Check for manual control
  // Serial.println("Manual control button pressed");
  checkControl = !checkControl;
}

void IRAM_ATTR direction() {    // Check for direction toggle
  // Serial.println("Joystick button pressed");
  checkCardinality = !checkCardinality;
}

```

```
void setup() {
  Serial.begin(115200);                         // Start serial communication
  pinMode(manualButton, INPUT_PULLUP);          // Setup manual control button pin with pull-up
  pinMode(diagCardButton, INPUT_PULLUP);        // Setup diagonal/cardinal control pin with pull-up
  pinMode(xpin, INPUT);                         // Setup x-axis pin as input
  pinMode(ypin, INPUT);                         // Setup y-axis pin as input

  pinMode(ain1, OUTPUT);                        // Setup motor control pins as outputs
  pinMode(ain2, OUTPUT);
  pinMode(bin1, OUTPUT);
  pinMode(bin2, OUTPUT);

  pinMode(vlimButton, INPUT_PULLUP);            // Limit switches with pull-up
  pinMode(hlimButton, INPUT_PULLUP);

  pinMode(AAPin, INPUT_PULLUP);                 // Activate encoder pins
  pinMode(BAPin, INPUT_PULLUP);
  pinMode(ABPin, INPUT_PULLUP);
  pinMode(BBPin, INPUT_PULLUP);

  attachInterrupt(digitalPinToInterrupt(AAPin), enc1, CHANGE);            // Encoder pin interrupt functions
  attachInterrupt(digitalPinToInterrupt(BAPin), enc1, CHANGE);
  attachInterrupt(digitalPinToInterrupt(ABPin), enc2, CHANGE);
  attachInterrupt(digitalPinToInterrupt(BBPin), enc2, CHANGE);

  attachInterrupt(digitalPinToInterrupt(manualButton), manual, FALLING);
  attachInterrupt(digitalPinToInterrupt(diagCardButton), direction, FALLING);

  digitalWrite(ain1, LOW);                      // Turn off all motors
  digitalWrite(ain2, LOW);
  digitalWrite(bin1, LOW);
  digitalWrite(bin2, LOW);

  timer = timerBegin(1000000);
  timerStop(timer);
  timerRestart(timer);

  // Serial.println("Setup successful!");
}

void loop() {
  // Serial.println("New loop initiated");

  while (homingSequence) {                  // Run homing sequence while true
    if (!(digitalRead(vlimButton))) {
      verticalLimit = true;
    }

    if (!(digitalRead(hlimButton))) {
      horizontalLimit = true;
    }
    homing(motCon);                         // Do not go to switch statement until homing sequence is done
    encoderHandler.resetEncoders();
  }
```

```
  }

  // if (!(digitalRead(vlimButton))) {
  //   Serial.println("Vertical limit reached");
  // }

  // if (!(digitalRead(hlimButton))) {
  //   Serial.println("Horizontal limit reached");
  // }

  if (checkForButtonPress(checkCardinality)) {                    // Event checker
    Serial.println("Cardinality changed");
    toggleCardinality = !toggleCardinality; // Change state

    checkCardinality = false;
  }

  if (checkForButtonPress(checkControl)) {                        // Event checker
    Serial.println("Control changed");
    if (toggleControl) {                    // If was true, enable homing sequence
      homingSequence = true;
      verticalLimit = false;
      horizontalLimit = false;
      toggleControl = false;
      toggleCardinality = false;
    } else {
      toggleControl = true;
    }
    checkControl = false;
  }

  // Serial.println(digitalRead(hlimButton));
  // Serial.println(digitalRead(vlimButton));

  switch (state) {  // Top left = -11800 bottom right = 10700 enc 1 // top left =
    case 0:   // Standby mode enabled
      if (homingSequence) {                  // If homingsequence is currently running, do not begin pattern mode (here for redundancy)
        Serial.println("Uh oh");             // This code block should in theory never be triggered
        homingSequence = false;
        break;
      }

      pattern(motCon);

      motCon.off();
      Serial.println("Pattern Mode");
      toggleCardinality = false;

      state = toggleControl + toggleCardinality;
      break;
    case 1:   // Standby mode disabled / Cardinal manual mode
      joyHan.cardinalInput(motCon);
      Serial.println("Currently accepting cardinal inputs");
      // Serial.println(encoderHandler.returnEncoder1());


      // if (verticalLimit) {
      //   Serial.println("Vertical limit has been reached");
      // }
```

```
      state = toggleControl + toggleCardinality;
      break;
    case 2:   // Diagonal manual mode
      joyHan.diagonalInput(motCon);
      Serial.println("Currently accepting diagonal inputs");

      // if (verticalLimit) {
      //   Serial.println("Vertical limit has been reached");
      // }

      // if (horizontalLimit) {
      //   Serial.println("Horizontal limit has been reached");
      // }

      state = toggleControl + toggleCardinality;
      break;
  }
}

// Homing function
void homing(MotorControl& MC) {
  if (!verticalLimit) {                   // Don't continue until vertical limit has been reached
    // Serial.println("Approaching vertical limit");
    MC.down();                            // Move motors down
    return;
  }

  if (!horizontalLimit) {                 // Don't continue until horizontal limit has been reached
    // Serial.println("Approaching horizontal limit");
    MC.left();                            // Move motors down
    return;
  }

  MC.off();

  homingSequence = false;                 // Exit loop
  verticalLimit = false;
  horizontalLimit = false;
}

bool checkForButtonPress(bool variable) {
  if (variable == true) {
    timerStart(timer);
    while (timerReadMillis(timer) < 200) {
      continue;
    }
    timerStop(timer);
    timerRestart(timer);
    return true;
  }
  return false;
}

void pattern(MotorControl& MC) {
  while (encoderHandler.returnEncoder1() < 10500) {
    enc = encoderHandler.returnEncoder1();
```

```
void pattern(MotorControl& MC) {
  while (encoderHandler.returnEncoder1() < 10500) {
    enc = encoderHandler.returnEncoder1();

    while (encoderHandler.returnEncoder1() > enc - 10800) { // Move up
      if (checkForButtonPress(manualButton)) {
      toggleControl = true;
      return;
    }

      MC.up();
    }

    enc = encoderHandler.returnEncoder1();

    while (encoderHandler.returnEncoder1() < enc + 1000) {
      if (checkForButtonPress(manualButton)) {
      toggleControl = true;
      return;
    }

      MC.right();
    }

    while (digitalRead(vlimButton)) {
      if (checkForButtonPress(manualButton)) {
      toggleControl = true;
      return;
    }

      MC.down();
    }

    enc = encoderHandler.returnEncoder1();

    while (encoderHandler.returnEncoder1() < enc + 1000) {
      if (checkForButtonPress(manualButton)) {
      toggleControl = true;
      return;
    }

      MC.right();
    }
  }

  homingSequence = true; // If the end of the pattern is reached, initiate homing sequence agai
}
```

```
#ifndef MotorControl_h
#define MotorControl_h

class MotorControl { // Class Declaration
  public:
    MotorControl(int a1, int a2, int b1, int b2); // Constructor

    void right(); // Motion commands
    void left();
    void up();
    void down();
    void upright();
    void upleft();
    void downright();
    void downleft();
    void off();

  private:
    // Pin storage
    int ain1;
    int ain2;
    int bin1;
    int bin2;
};

#endif
```

```
#include <MotorControl.hpp>
#include <Arduino.h>

MotorControl::MotorControl(int a1, int a2, int b1, int b2) { // Constructor
  ain1 = a1;
  ain2 = a2;
  bin1 = b1;
  bin2 = b2;
}


// 1h 2l for CW, 1l 2h for CCW
void MotorControl::right() { // 1 CCW 2 CCW
  digitalWrite(ain1, LOW);
  digitalWrite(ain2, HIGH);

  digitalWrite(bin1, LOW);
  digitalWrite(bin2, HIGH);
}

void MotorControl::left() { // 1 CW 2 CW
  digitalWrite(ain1, HIGH);
  digitalWrite(ain2, LOW);

  digitalWrite(bin1, HIGH);
  digitalWrite(bin2, LOW);
}

void MotorControl::up() { // 1 CW 2 CCW
  digitalWrite(ain1, HIGH);
  digitalWrite(ain2, LOW);

  digitalWrite(bin1, LOW);
  digitalWrite(bin2, HIGH);
}

void MotorControl::down() { // 1 CCW 2 CW
  digitalWrite(ain1, LOW);
  digitalWrite(ain2, HIGH);

  digitalWrite(bin1, HIGH);
  digitalWrite(bin2, LOW);
}

void MotorControl::upright() { // 1 OFF 2 CCW
  digitalWrite(ain1, LOW);
  digitalWrite(ain2, LOW);

  digitalWrite(bin1, LOW);
  digitalWrite(bin2, HIGH);
}
```

```
void MotorControl::upleft() { // 1 CW 2 OFF
  digitalWrite(ain1, HIGH);
  digitalWrite(ain2, LOW);

  digitalWrite(bin1, LOW);
  digitalWrite(bin2, LOW);
}

void MotorControl::downright() { // 1 CCW 2 OFF
  digitalWrite(ain1, LOW);
  digitalWrite(ain2, HIGH);

  digitalWrite(bin1, LOW);
  digitalWrite(bin2, LOW);
}

void MotorControl::downleft() { // 1 OFF 2 CW
  digitalWrite(ain1, LOW);
  digitalWrite(ain2, LOW);

  digitalWrite(bin1, HIGH);
  digitalWrite(bin2, LOW);
}

void MotorControl::off() { // Turn off motors
  digitalWrite(ain1, LOW);
  digitalWrite(ain2, LOW);

  digitalWrite(bin1, LOW);
  digitalWrite(bin2, LOW);
}
```

```
#ifndef JoystickHandler_h
#define JoystickHandler_h

class MotorControl;

class JoystickHandler { // Class Declaration
  public:
    JoystickHandler(int xp, int yp); // Constructor

    void cardinalInput(MotorControl& MC) ; // Handles inputs for cardinal directions
    void diagonalInput(MotorControl& MC) ; // Handles inputs for diagonal directions

  private: // Pin storage
    int xpin;
    int ypin;
};

#endif

```

```cpp
#include <MotorControl.hpp>
#include <JoystickHandler.hpp>
#include <Arduino.h>

class MotorControl;

JoystickHandler::JoystickHandler(int xp, int yp) { // Constructor
  xpin = xp;
  ypin = yp;
}

void JoystickHandler::cardinalInput(MotorControl& MC) {
  int xaxis = analogRead(xpin);
  int yaxis = analogRead(ypin);

  int xval = map(xaxis, 0, 4095, -512, 512);
  int yval = map(yaxis, 0, 4095, -512, 512);

  // Enable deadzone
  if (abs(xval) < 100 && abs(yval) < 100) {
    // Serial.println("Joystick idle ");
    MC.off();

    return;
  }

  // Check joystick movement and call corresponding movement functions
  if (abs(xval) >= abs(yval)) { // Override y input if x > y
    if (xval > 100) {
      MC.right();
      // Serial.print("x: ");
      // Serial.println(xval);

    } else if (xval < -100) {
      MC.left();
      // Serial.print("x: ");
      // Serial.println(xval);

    }
  } else { // Override x input if y > x
    if (yval > 100) { // Note that y is flipped (positive is down, negative up) can change if we want
      MC.down();
      // Serial.print("y: ");
      // Serial.println(yval);

    } else if (yval < -100) {
      MC.up();
      // Serial.print("y: ");
      // Serial.println(yval);

    }
  }
}
```

```
void JoystickHandler::diagonalInput(MotorControl& MC) {
  int xaxis = analogRead(xpin);
  int yaxis = analogRead(ypin);

  int xval = map(xaxis, 0, 4095, -512, 512);
  int yval = map(yaxis, 0, 4095, -512, 512);

  // Enable deadzone
  if (abs(xval) < 100 && abs(yval) < 100) {
    // Serial.println("Joystick idle ");
    MC.off();

    return;
  }

  // Check joystick movement and call corresponding movement functions
  if (xval > 0) { // Moving right
    if (yval > 0) { // Moving down
      // Serial.println("Diagonal down right")
      MC.downright();
    } else {
      // Serial.println("Diagonal up right")
      MC.upright();
    }

  } else { // Moving left
    if (yval > 0) { // Moving down
      // Serial.println("Diagonal down left")
      MC.downleft();

    } else { // Moving up
      // Serial.println("Diagonal up left"
      MC.upleft();

    }
  }
}
```

```
#ifndef EncoderHandler_h
#define EncoderHandler_h

class EncoderHandler { // Class Declaration
  public:
    EncoderHandler(int AAPin, int BAPin, int ABPin, int BBPin); // Constructor

    void handleEncoder1();      // Handle encoder 1
    void handleEncoder2();      // Handle encoder 2
    void resetEncoders();       // Zero out encoder values

    int returnEncoder1();       // Return encoder 1 tick value
    int returnEncoder2();       // Return encoder 2 tick value

    int returnRotation1();      // Return encoder 1 rotation value
    int returnRotation2();      // Return encoder 2 rotation value

  private: // Pin + prev state storage
    int encoderCount1;          // Raw encoder count for motor 1 (A)
    int lastAAState;            // Previous state of AA

    int encoderCount2;          // Raw encoder count for motor 2 (A)
    int lastABState;            // Previous state of AB

    int encoderAAPin;           // Encoder Pins
    int encoderBAPin;
    int encoderABPin;
    int encoderBBPin;

    int cpr;                    // 700 encoder ticks per revolution for our motor
};

#endif
```

```cpp
#include <EncoderHandler.hpp>
#include <Arduino.h>

EncoderHandler::EncoderHandler(int AA, int BA, int AB, int BB) { // Constructor
  encoderCount1 = 0;
  lastAAState = 0;

  encoderCount2 = 0;
  lastABState = 0;

  encoderAAPin = AA;
  encoderBAPin = BA;
  encoderABPin = AB;
  encoderBBPin = BB;

  cpr = 700;
}

void EncoderHandler::handleEncoder1() {
  int currentAAState = digitalRead(encoderAAPin);
  int currentBAState = digitalRead(encoderBAPin);

  // Check transitions for encoder 1
  if (lastAAState == LOW && currentAAState == HIGH) {
    if (currentBAState == LOW) {
      encoderCount1++;
    } else {
      encoderCount1--;
    }
  } else if (lastAAState == HIGH && currentAAState == LOW) {
    if (currentBAState == HIGH) {
      encoderCount1++;
    } else {
      encoderCount1--;
    }
  }

  lastAAState = currentAAState;
}

void EncoderHandler::handleEncoder2() {
  int currentABState = digitalRead(encoderABPin);
  int currentBBState = digitalRead(encoderBBPin);

  // Check transitions for encoder 2
  if (lastABState == LOW && currentABState == HIGH) {
    if (currentBBState == LOW) {
      encoderCount2++;
    } else {
      encoderCount2--;
    }
  } else if (lastABState == HIGH && currentABState == LOW) {
    if (currentBBState == HIGH) {
      encoderCount2++;
```

```
  lastABState = currentABState;
}

void EncoderHandler::resetEncoders() {
  encoderCount1 = 0;
  encoderCount2 = 0;
  lastAAState = digitalRead(encoderAAPin);
  lastABState = digitalRead(encoderABPin);
}

int EncoderHandler::returnEncoder1() {
  return encoderCount1;
}

int EncoderHandler::returnEncoder2() {
  return encoderCount2;
}

int EncoderHandler::returnRotation1() {
  return (encoderCount1/cpr);
}

int EncoderHandler::returnRotation2() {
  return (encoderCount2/cpr);
}
```

5.4 Appendix D: Additional Images/Photos

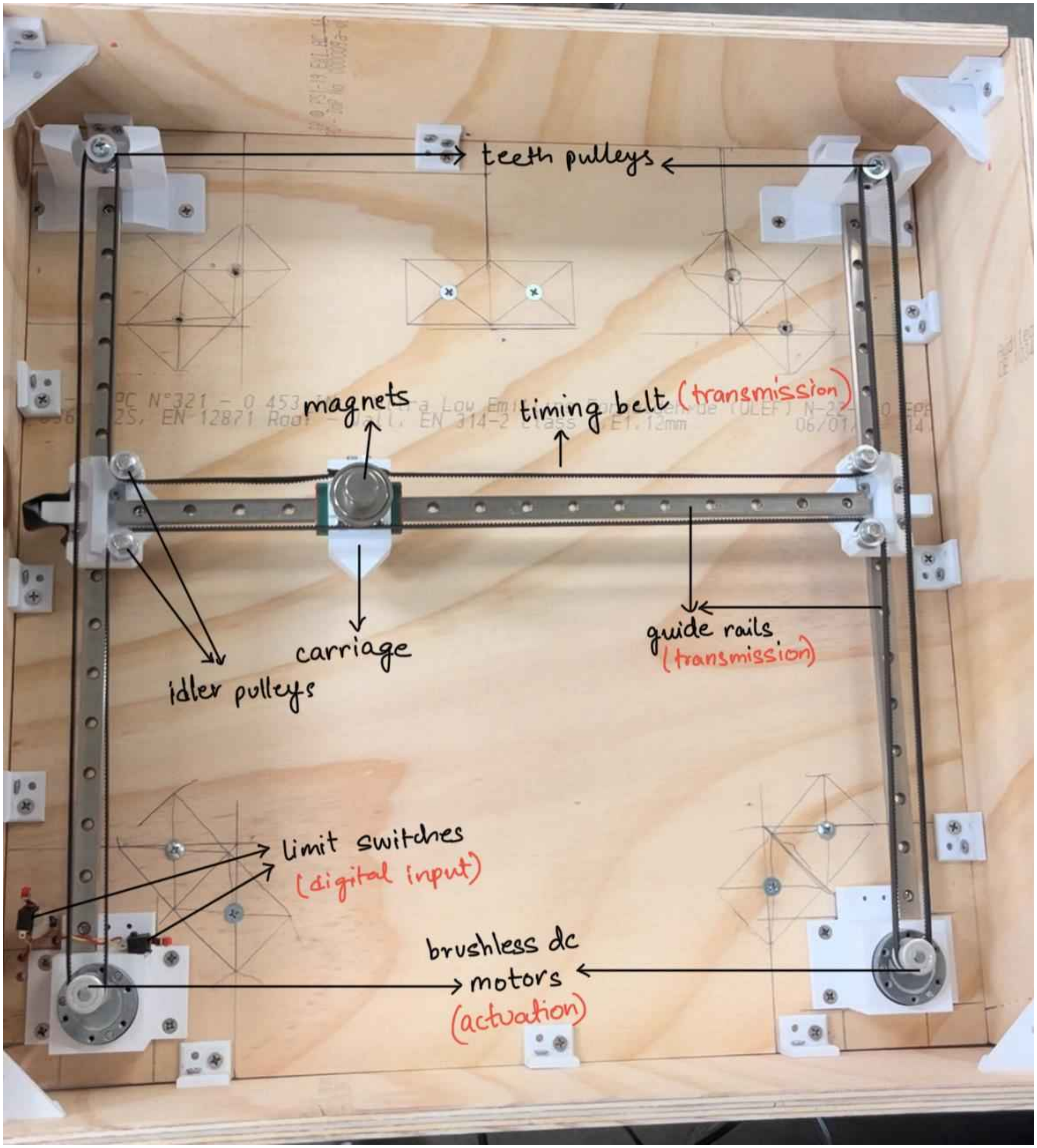

Figure 7: Full scale image of the mechanics of the sand table

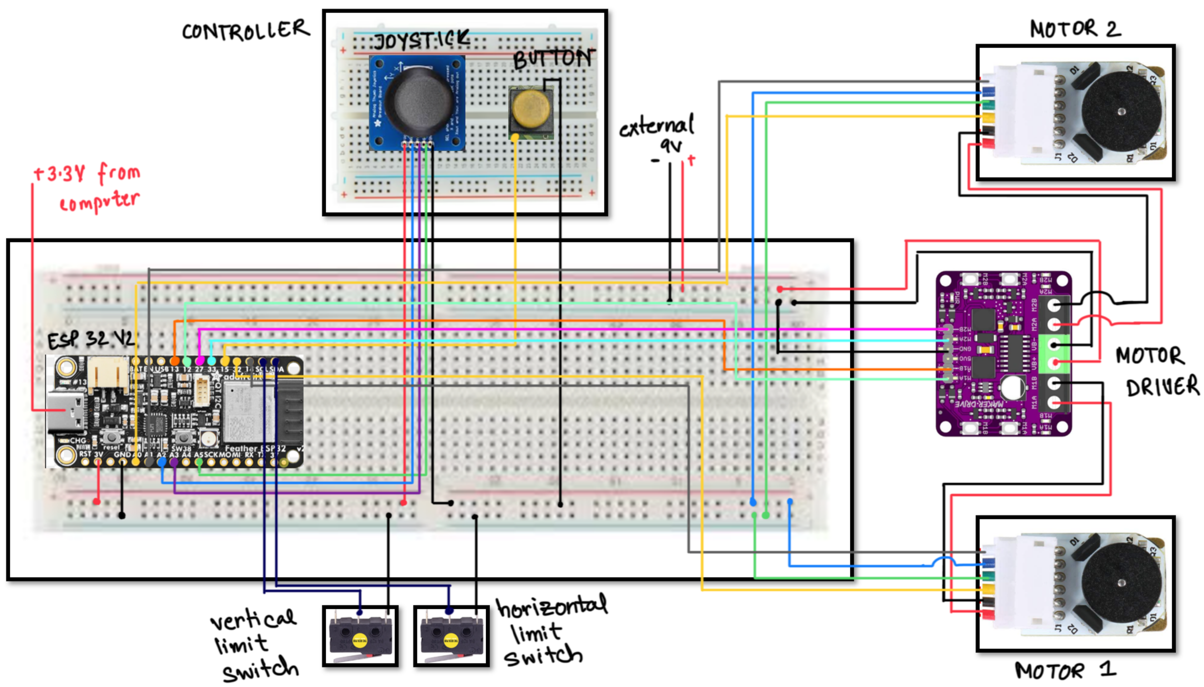

Figure 8: Full scale image of our circuit

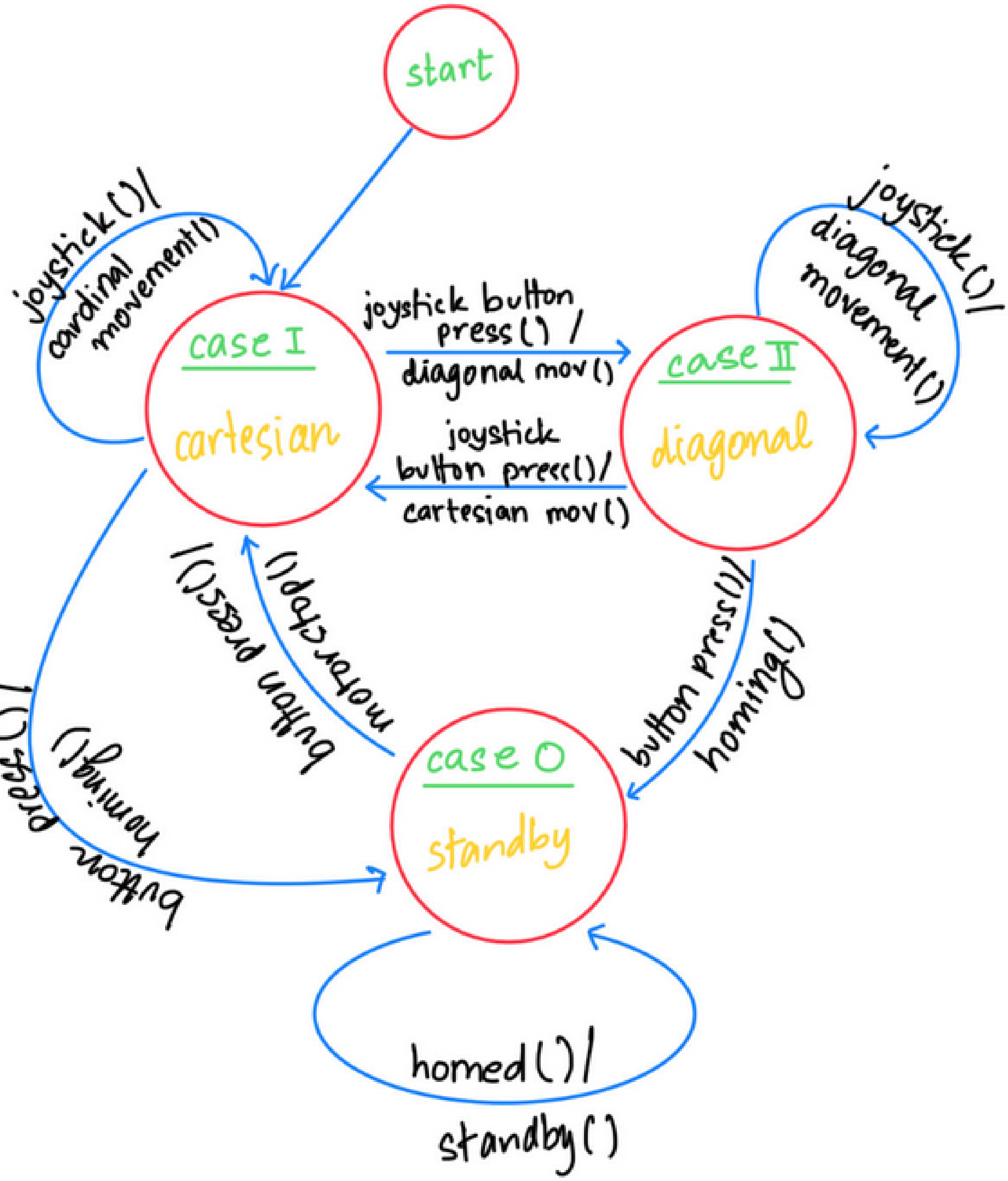

Figure 9: Full scale image of our state diagram