Automatic Connect 4 Playing Robot

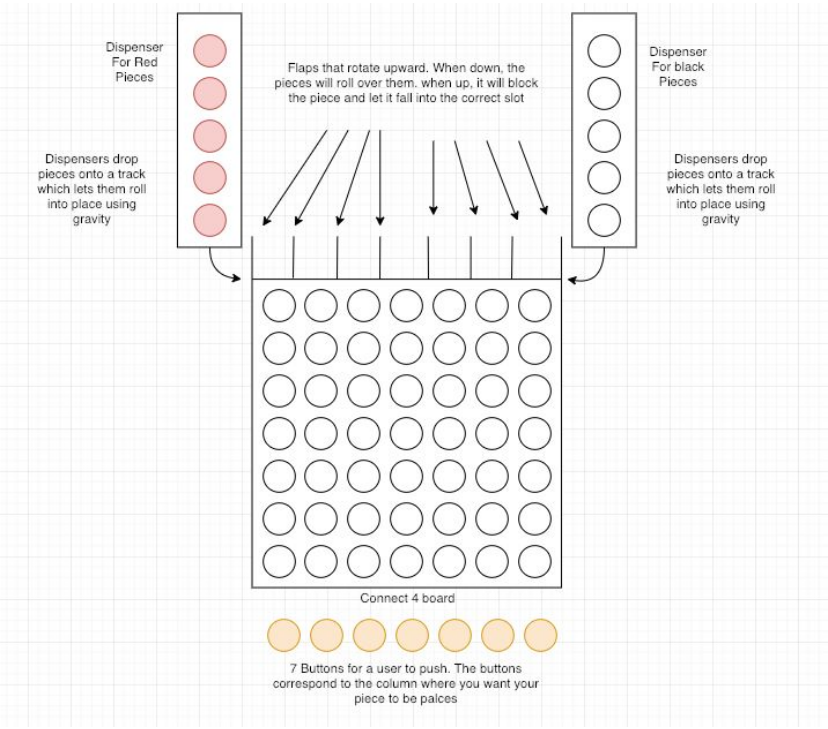
Dane Zieman
(dziema2@uic.edu)
Liyuan Chen
(lchen79@uic.edu)
Bharat Middha
(bmiddh2@uic.edu)

Abstract
This project is a robot that plays the game of connect four against a human automatically. It takes the users move from a button, one for each of the seven columns a move can be made in. After the player selects their move, the robot will use AI to decide which column to place its move in. All moves will be made with a series of game piece dispensers and mechanical tracks/slots to dictate which column the dispensed piece falls into.

Introduction
Connect Four is a popular game often played by children and adults throughout the United States. First created in 1974, it pits 2 players who each try to make a contiguous sequence of 4 of their own pieces while blocking the other player's effort to do so. For this project, we are aiming to create a robot that can not only mechanically place pieces onto the board, but also serve as an opponent, utilizing AI to contest challengers for victory.

To interact with this machine, players will press buttons located at the bottom of each column of the game board to indicate where they want to play their pieces, prompting the robot to deposit a game piece into the specified column. Subsequently, the AI will decide on a move based on the current game state, and after conducting its move, will return control back to the player. Because all moves are conducted by the robot, it maintains complete information over the game state, and can appropriately pick an appropriate move at all times.

Project Design Procedure
The design of the robot will consist of 3 main parts: the game piece dispenser, the connect 4 board, and a button array. Moves are selected via the button array on the bottom; when a move is conducted, a signal will be sent to the dispensers, which will release a game piece of the appropriate color from one of the 2 containers. At the same time, seven trap door-like slots, one placed over each column, will control where the dispensed piece falls. The slots will have two states: open and closed.When closed, the slots will serve as a track for the piece to roll over. When opened, the slot will serve as an open hole and wall, blocking the piece and forcing it to fall to the desired column. An initial conceptual drawing of this machine is show to the right, and is also available in the appendix.



In total, this will require a minimum of 18 input/outputs: 7 stepper motors for the slot covers/tracks, 2 for the 2 dispensers, and 7 for the 7 buttons. The dispensers and the slots will be custom built for this project. To accomplish this, measurements of the Connect 4 board and pieces will be taken, and appropriately sized parts will be designed using 3D drawing tools, such as solidworks.

Design subsections

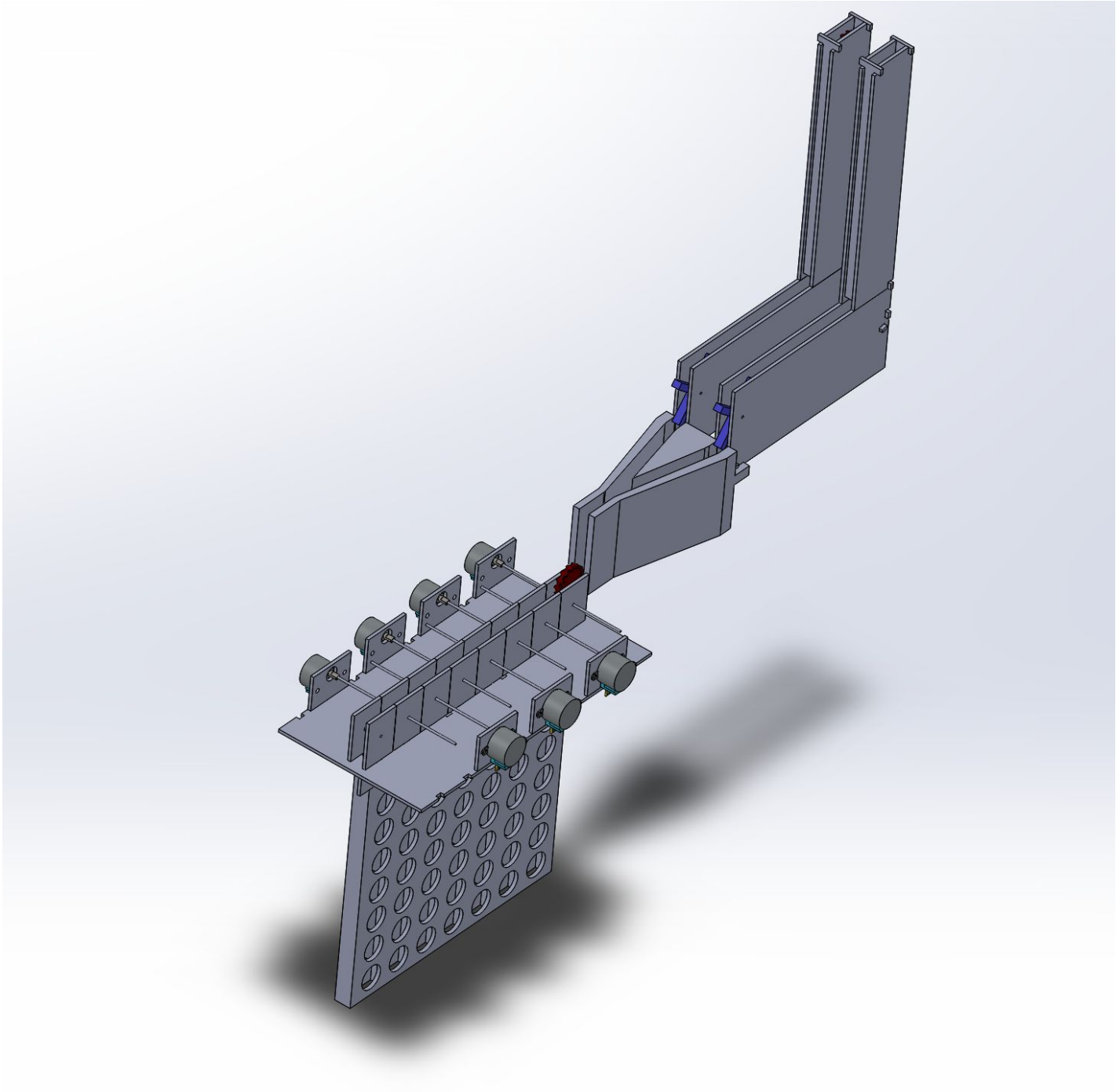
1) Control arduino: This subsection consists of an arduino and nine buttons. These buttons will all be input to the arduino, and will decide which of the flaps to open and make a move. It will be connected to the logic arduino via wireless communication. Diagrams are included for this arduino below

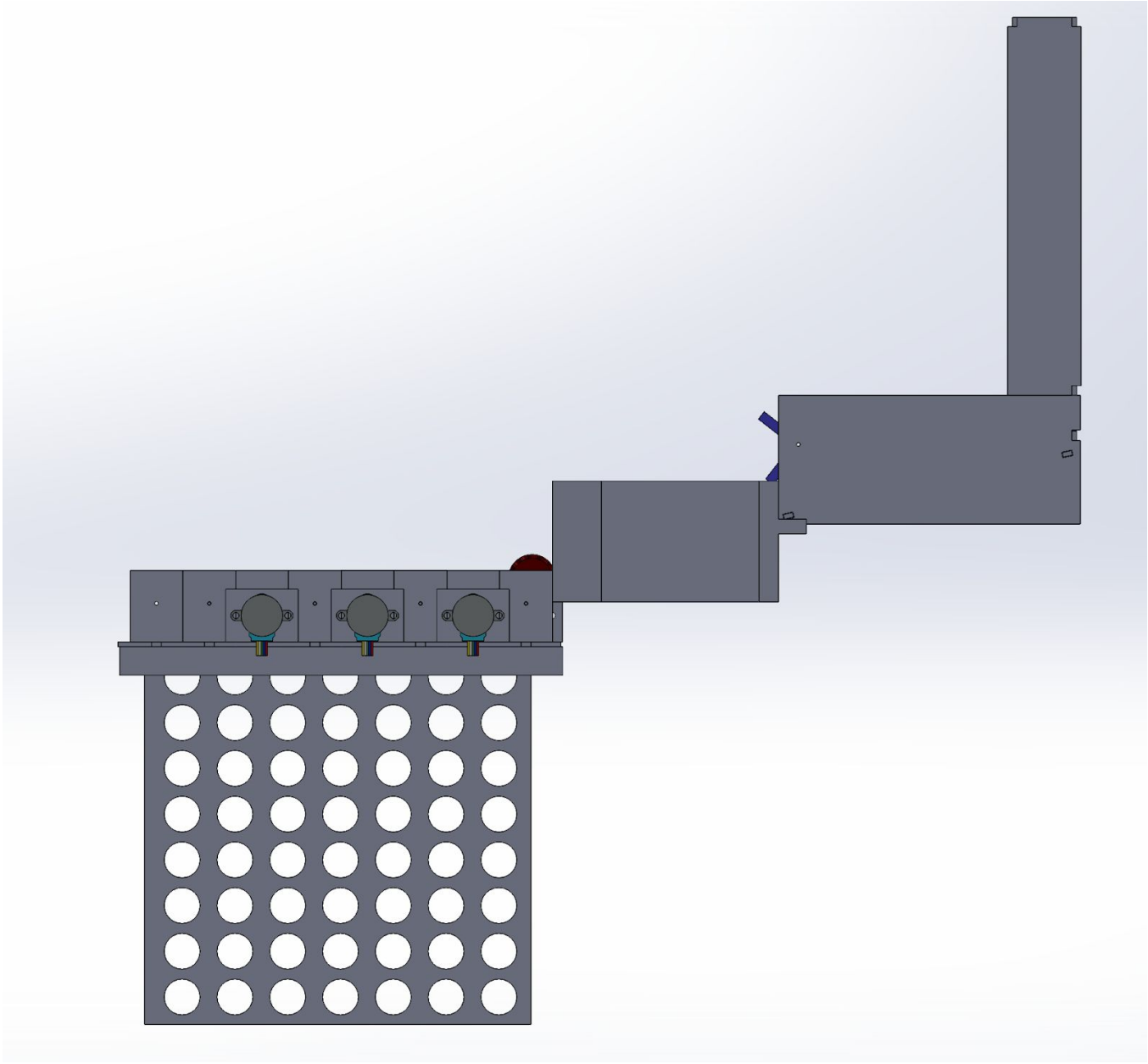
2) Logic arduino: This arduino keeps track of the game state, and decides upon moves for the AI as well as relays the player moves to the motor arduino.

3) Motor arduino: This arduino controls all 9 of the stepper motors used for this project. It is connected to the logic arduino via serial, and it makes moves based off the input given to it from the logic arduino.

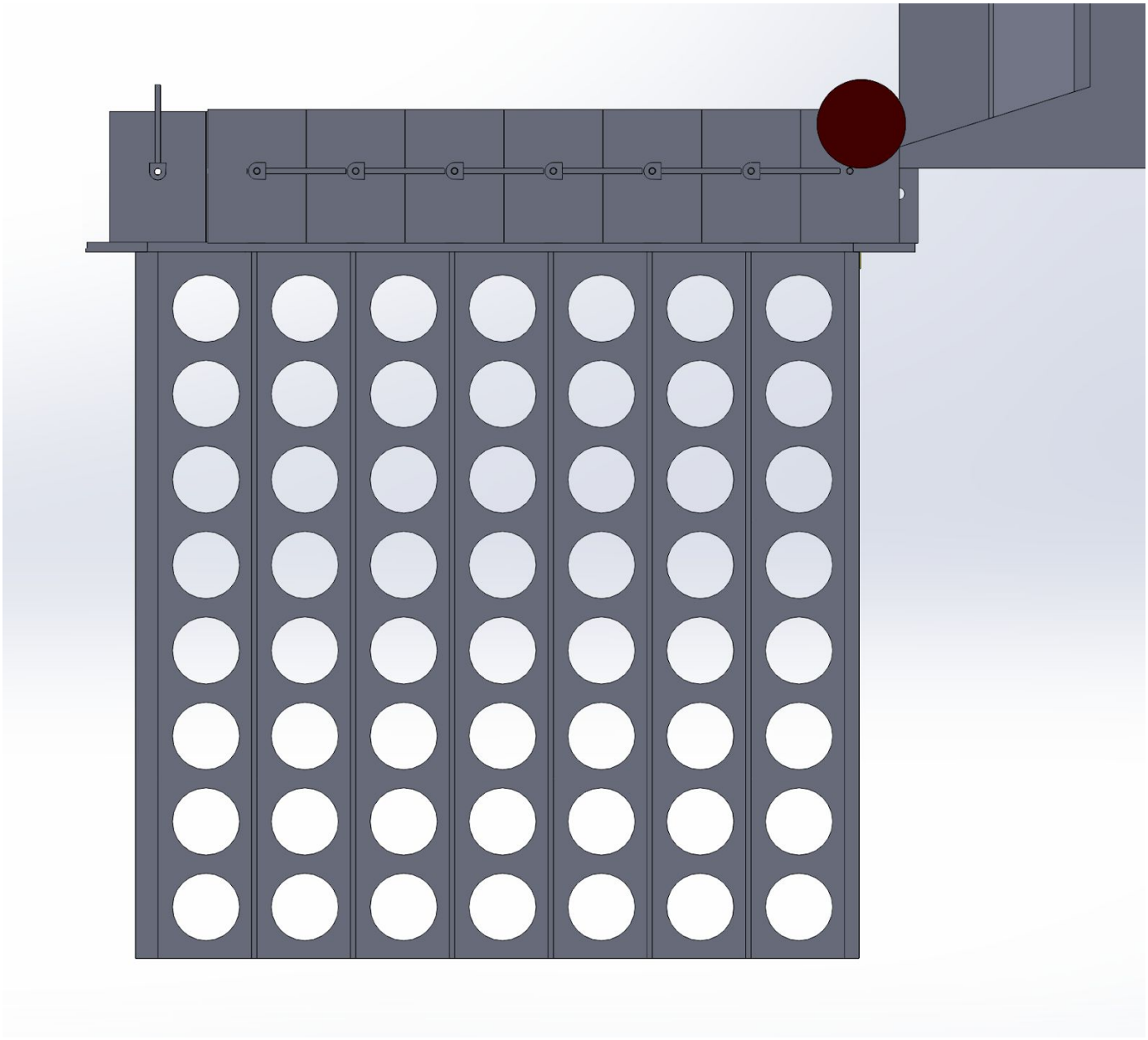
Detailed Drawings

Full assembly:

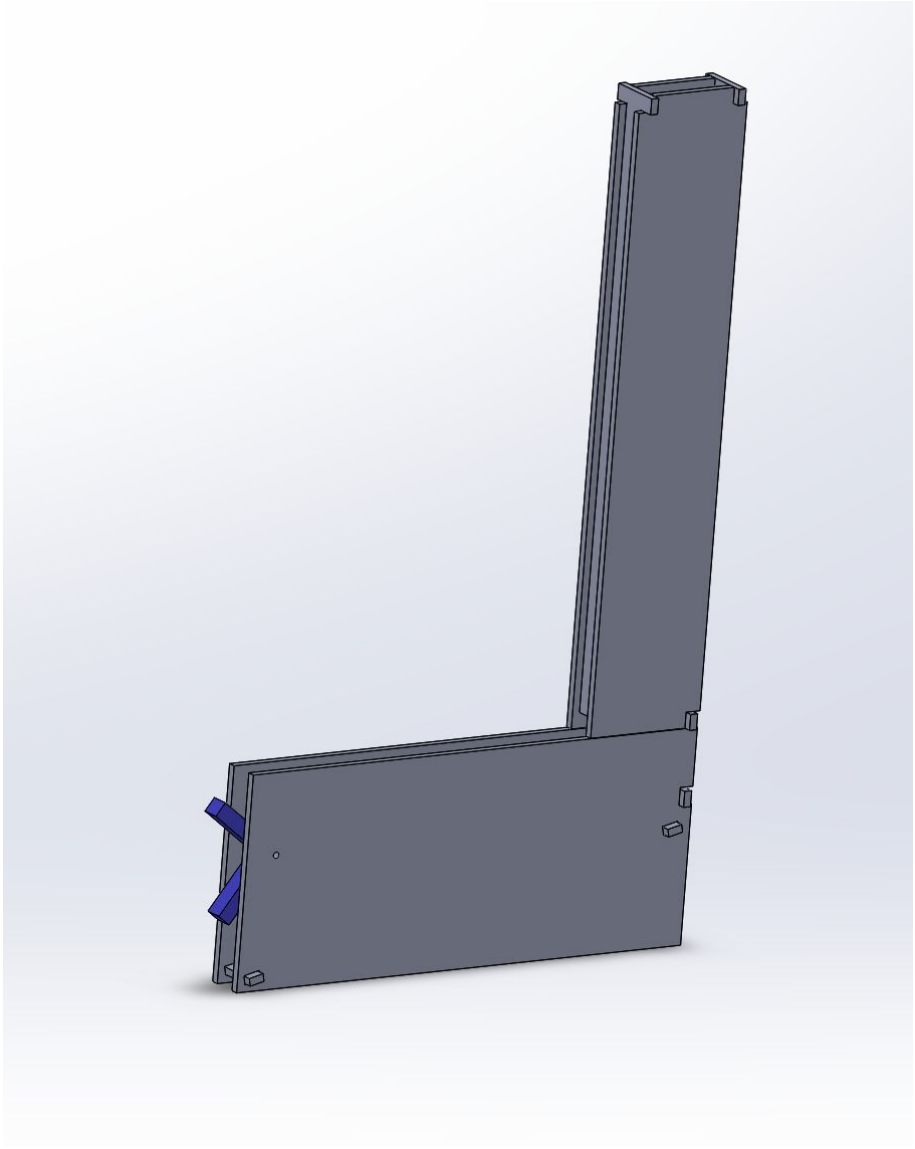




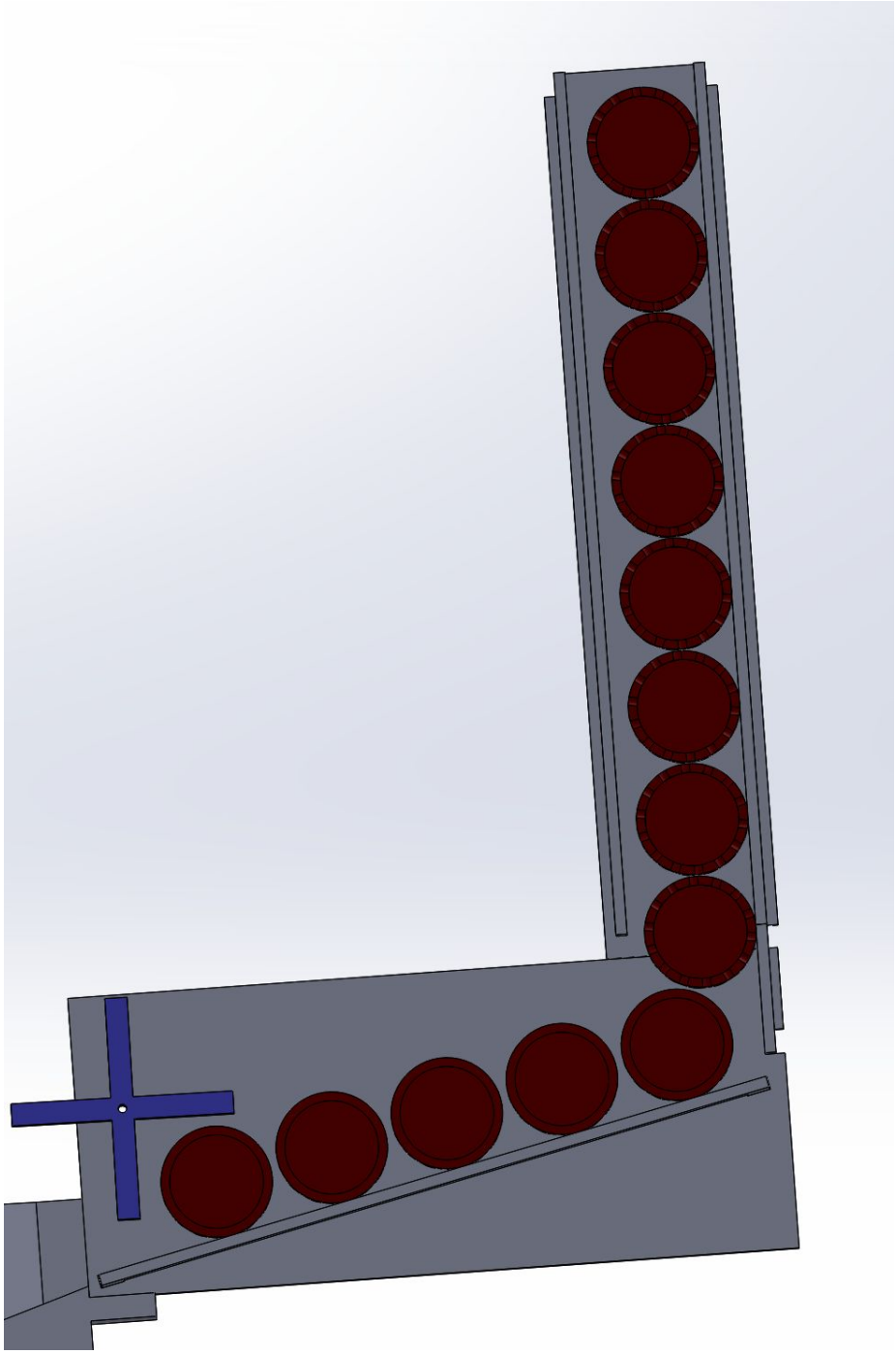
Flaps above the columns (showing a desired move into the seventh column):



Dispenser drawings



:



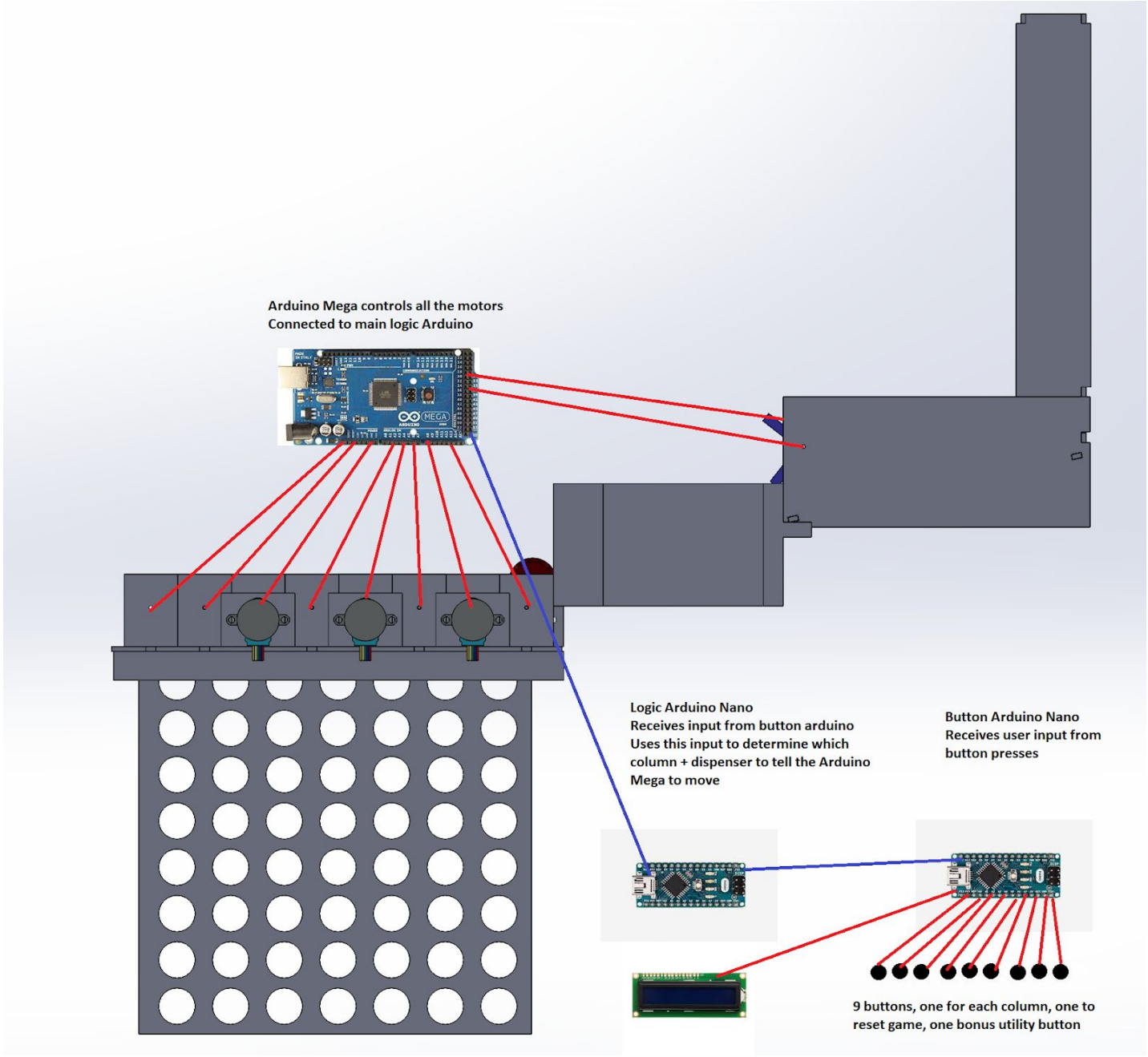
Arduino Usage

We will be using 3 arduinos to create this robot. One unit will take user input from a player, another will control the two dispensers and the seven latches over each column and the third will control the game logic. For communication between arduinos, the logic arduino will communicate with the motor arduino via serial, and the logic arduino will communicate with the controller arduino wirelessly Each of these arduinos will be in charge of their own input/output device and communicating their results to other arduinos. A sample breakdown is as follows: Code for the arduinos is shown below in the appendix

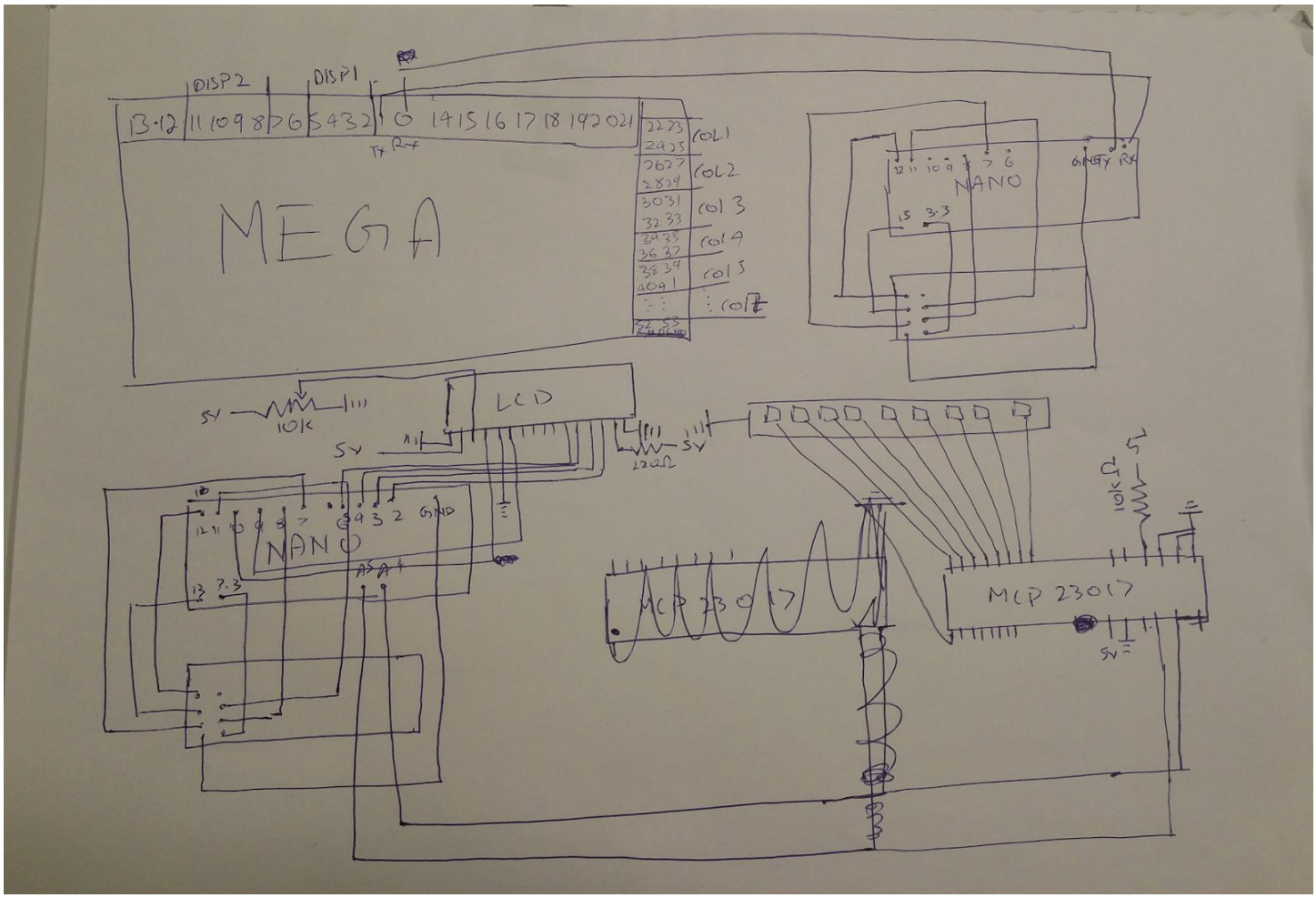
- Button Arduino: 9x Buttons
- Motor Arduino: 9x stepper motors
- Logic Arduino: Communications to the other two arduinos

ARDUINO DRAWINGS

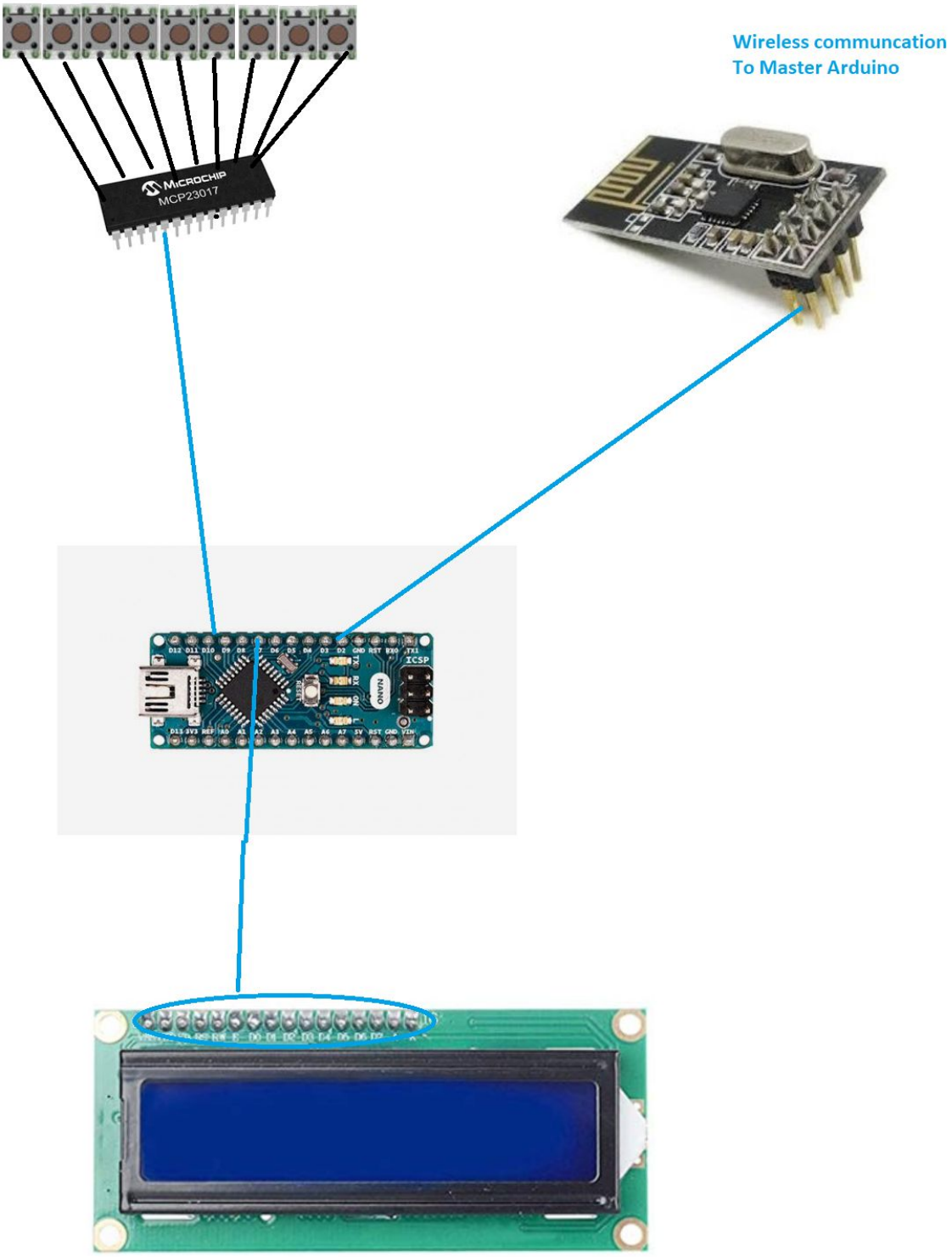
High level arduino drawing showing location/communication



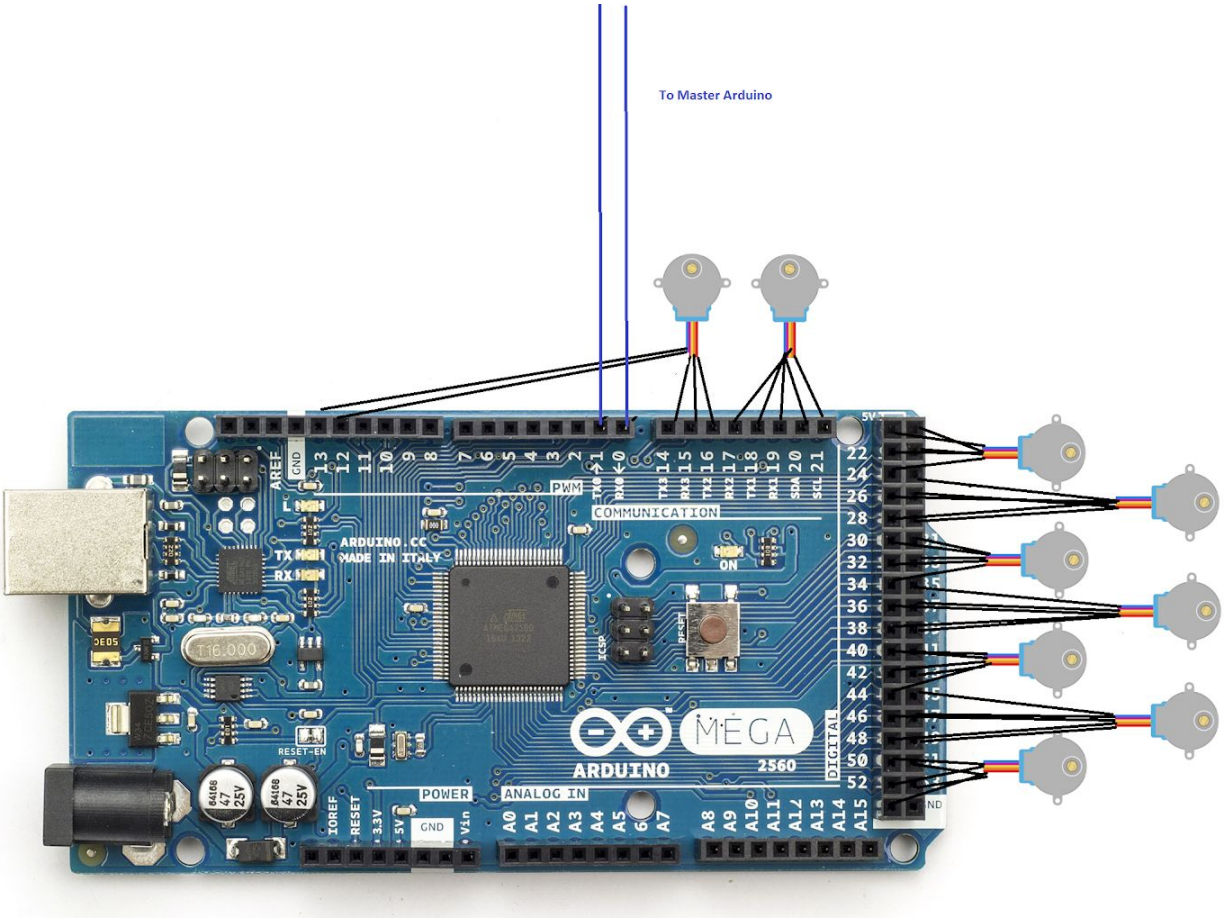
Detailed sketch of total assembly showing all inputs and outputs and connections



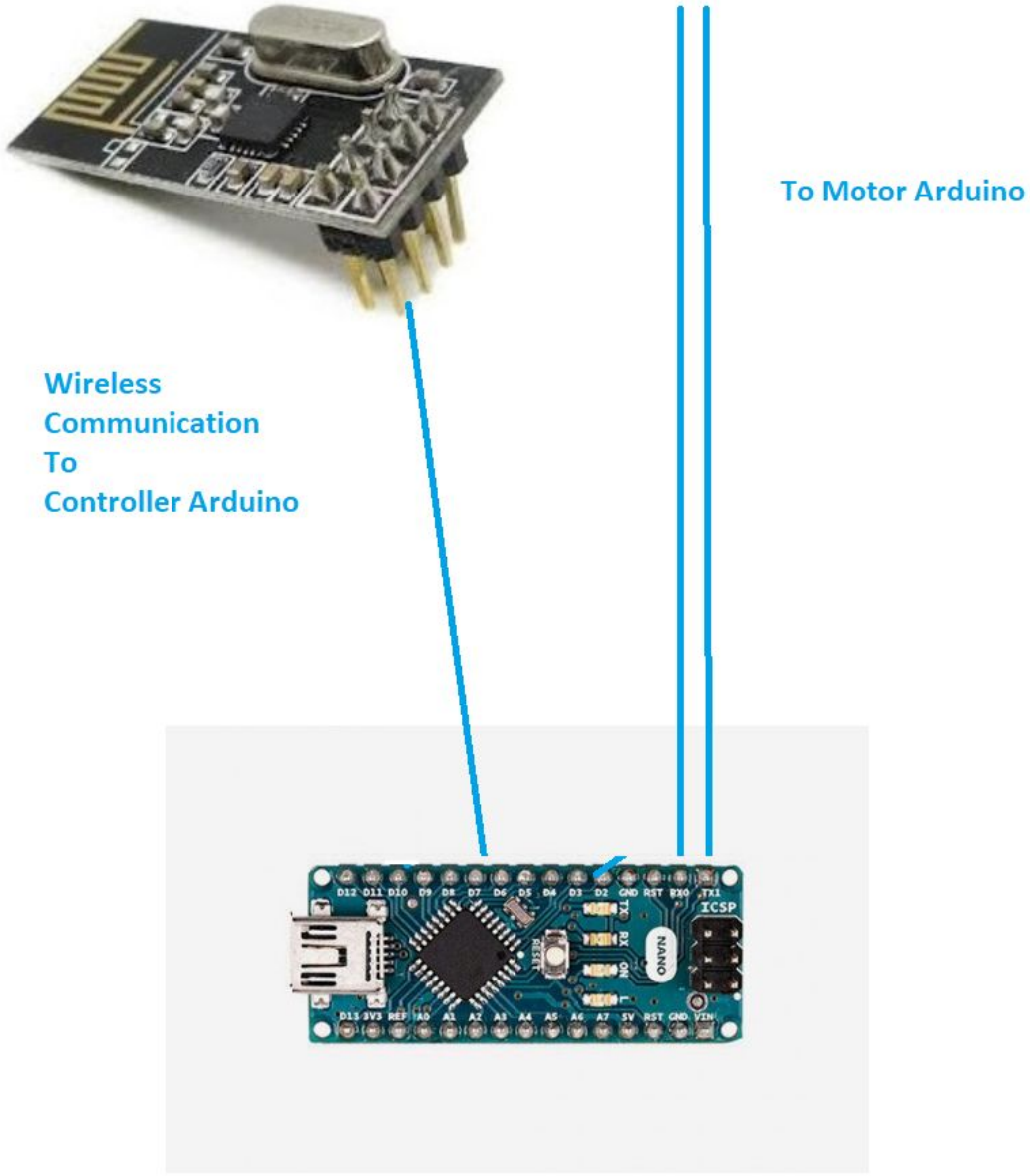
Less detailed, but easier to read drawing of arduino that controls the nine buttons



Less detailed, but easier to read drawing of Arduino that controls the seven flaps and two dispensers
Each of the stepper motors will be connected to a flap or dispenser



Less detailed, but easier to read drawing of Master Arduino



AI Implementation

Our AI program was heavily borrowed from Ryan Marcus's implementation (link can be found in Appendix C). The AI uses an expectimax search algorithm with alpha-beta pruning and implements a transposition table for fast caching of starting optimal moves. Initially we ran the program with a search tree depth of 7 moves, but found that our arduino was unable to handle the computation complexity of such a large look ahead, and so we cut down to a 4 move look ahead for our final implementation. Unfortunately, we also ran into some issues with memory allocation and just executing the AI on the arduino, and so for our actual demo we used a scaled back version of an AI that was more suitable for demonstrations

Timeline

Friday March 1st: Finalization of project

Week of March 4th: Purchase connect 4 game, take measurements needed to design custom parts, begin 3D drawings of custom parts.

Week of March 11th: Finalize initial prototypes for custom parts, submit prototypes for a dispenser and a slot. Explore ideas for arduino communication and look into Connect 4 AI

Week of March 18th: Test functionality of prototypes, make any changes necessary to get them to be operational. Continue working on ideas for communication and AI

Week of March 25th-Week of April 8: Finish the final designs for all custom parts, and submit them. Plan out wiring and solder what we can before parts are ready

Week of April 15th: Assemble all parts together, begin wiring

Week of April 22nd: Present our project at Engineering Expo 2019

Week of April 27th: Demo our project in the final lab session

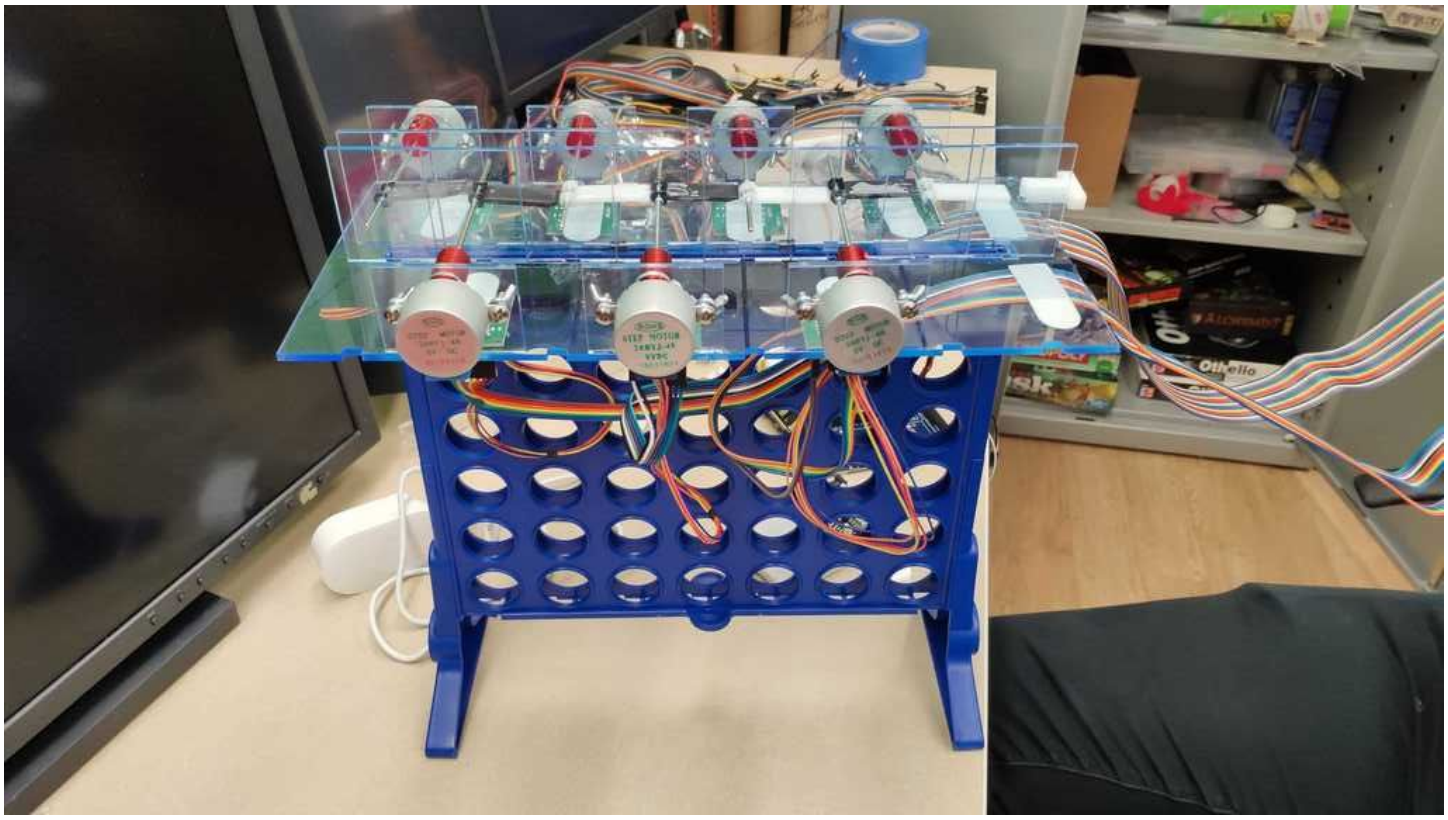
Results

The overall design intent was fulfilled in this project. We were able to demonstrate 10 consecutive correct moves for our demonstration (and probably could have done more, the machine was still running well when the demonstration ended). Some difficulties/areas of improvement for the project are as follows:

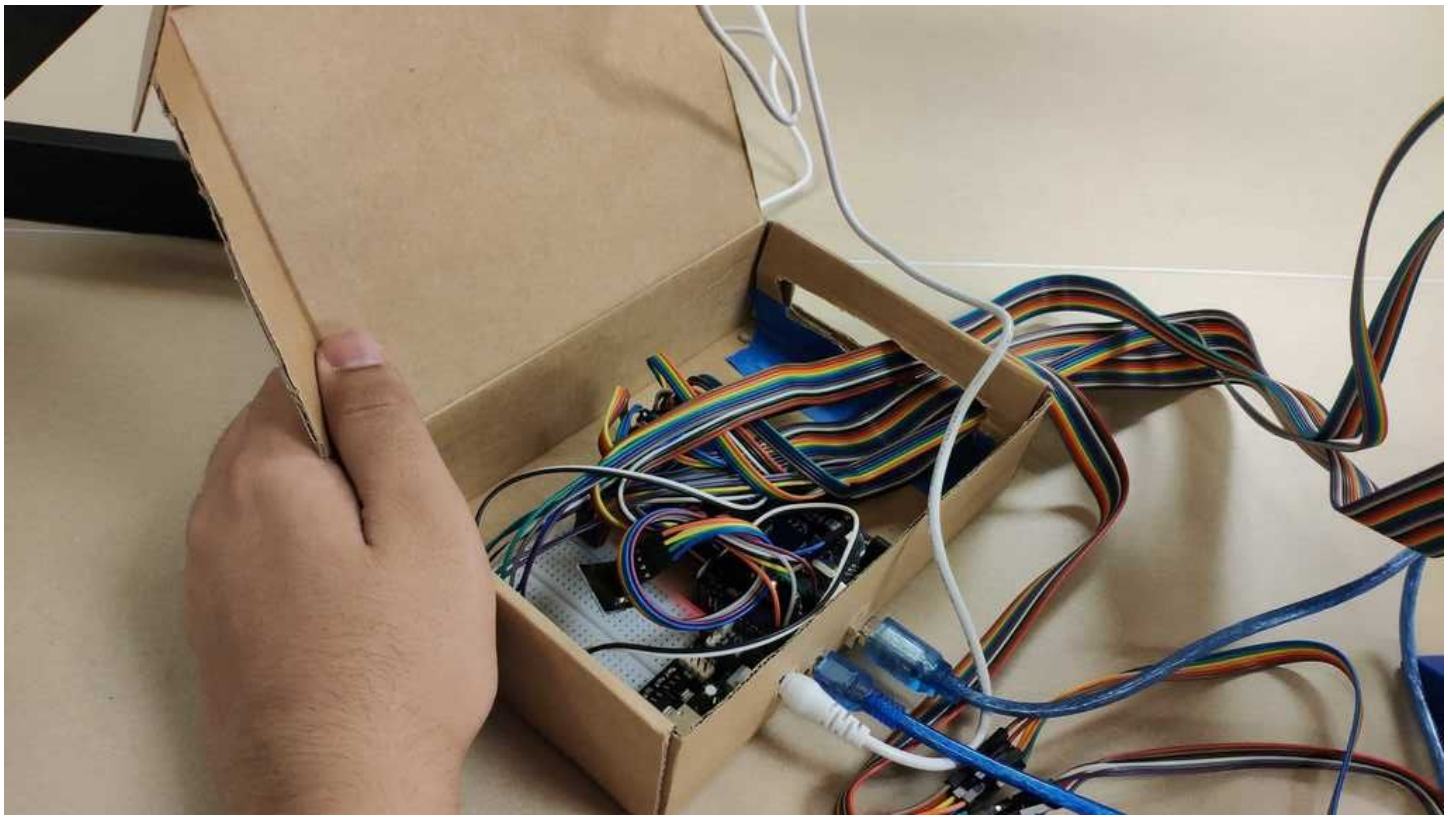
- Some of the pieces, such as the flaps, were super glued on. This super glue was not reliable, and sometimes the flaps would become loose and either sag or get stuck. It would be better to design a part that uses set screws to lock rotation, so that the connection to the axles is more robust
- The gravity feed could be fine tuned. The original design did not provide enough gravity, so we augmented the final design with an extra sloped piece to give it a bit more momentum. This gave it more consistent momentum, but due to the quick nature of the implementation of this piece it wasn't extremely reliable, sometimes the pieces would bounce or not come in straight on and thus lose momentum
- The table holding the dispenser could be better designed. We used a commercially available laptop desk with adjustable heights. However, this table was wobbly and not level. Designing a custom mount for the dispenser, at an optimal height, would improve the design
- Our full AI was too computationally intensive for an arduino Nano to do in a reasonable amount of time. We could either remedy this by offloading the actual calculation to a more powerful computer (e.g. a laptop) or we could use a simpler AI.
- Some of the parts were more difficult to assemble than they needed to be. This could be remedied by making some of the gaps between mating parts a little tighter, and designing the parts to be more friendly to a laser cut and assemble design
- Some of the parts had small functionality problems that were fixed using hotfixes. These problems could be fixed more robustly by slightly modifying the design
- Sometimes, due to the large number of motors connected to the Arduino Mega, it would reset due to insufficient power. We remedied this by connecting more power from a bunch of different sources. Powering the motors with a powerful external supply would be a better long term solution
- The connecting structure between our different components (dispenser, ramp, flaps, connect 4 board, etc) could be improved; a lot of our different parts were just left free standing, and we had to duct tape them together after we found out that a lot of them were shifting around during setup and operation. Adding more super glue/acrylic cement between pieces could have potentially solved this problem, but it also would have greatly increased the amount of space needed to store and transport the project

Pictures of final assembly

Top flap piece with motors



Box containing master + motor arduinos



First assembly (no motors)

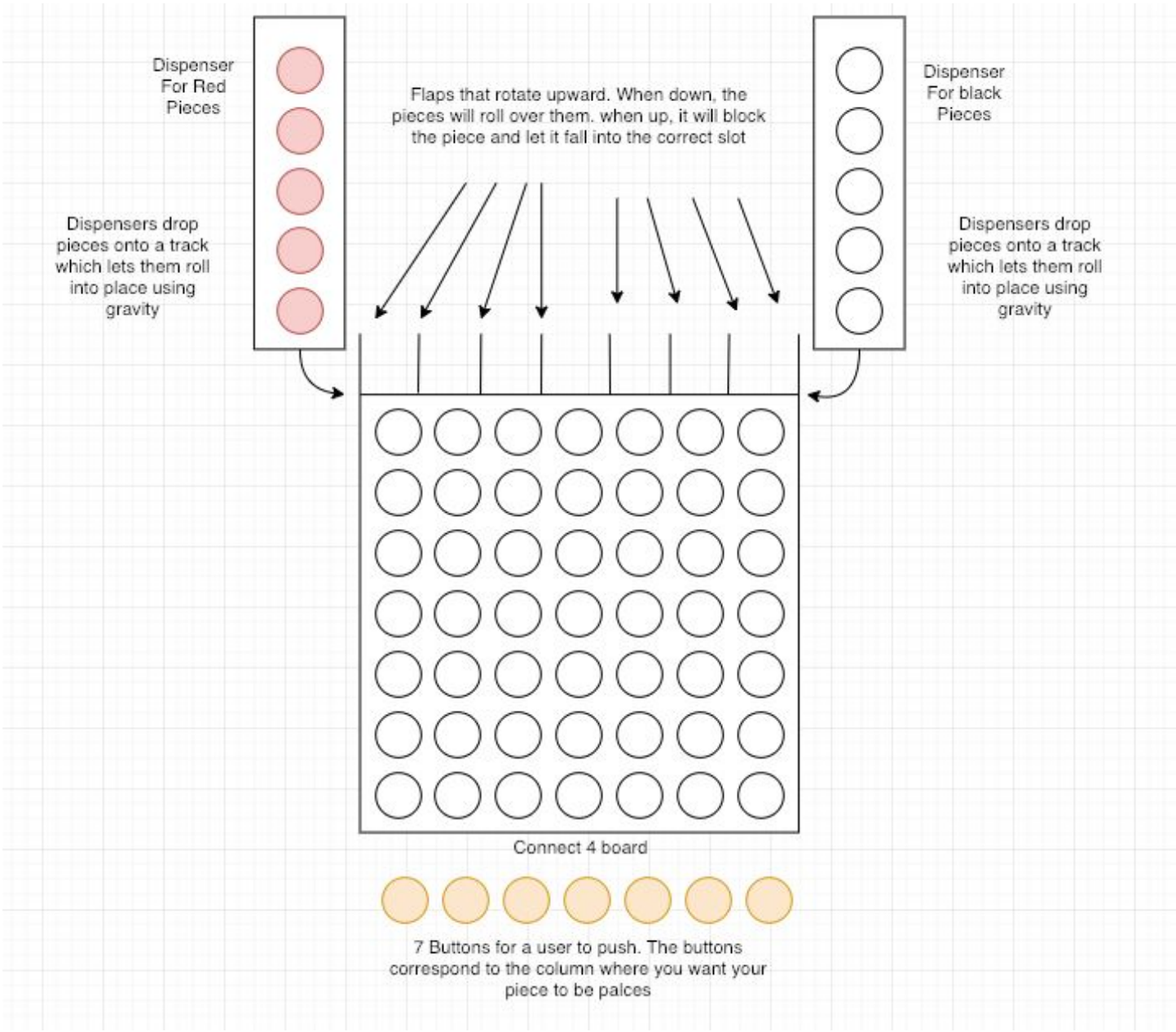
Final assembly + controller arduino

SUPPORTING MATERIALS:

Appendix A: Bill of Materials

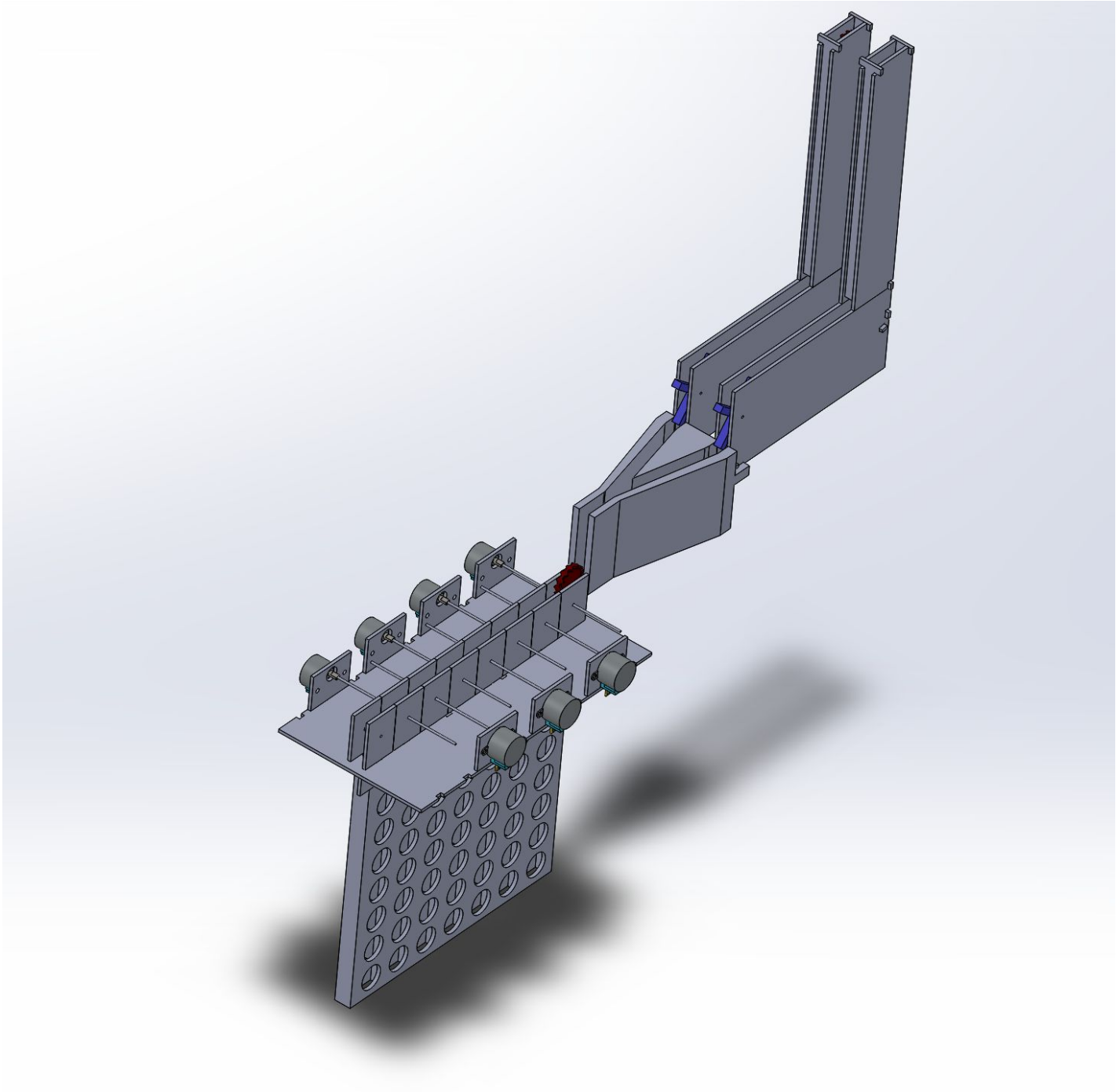
- 9x stepper motor
- 1x arduino mega
- 2x arduino nano
- 7x slot covers/tracks (one for each column, custom built, 3D printed)
- 2x dispensers (custom built, Laser cut and assembled)
- 9x buttons
- 1x Connect Four game
- 9x 2mm - 5mm couplings (to connect motors)
- 9x 2mm axles
- 1x MCP23017 GPIO Expander
- 2x nRF24 2.4 GHz Transceivers
- 1x channel to join two dispensers (custom built, 3D printed)
- 1x top flap holder piece (custom built, Laser cut and assembled)
- 1x platform to support dispensers
- 1x sloped track piece to give extra gravity speed to pieces

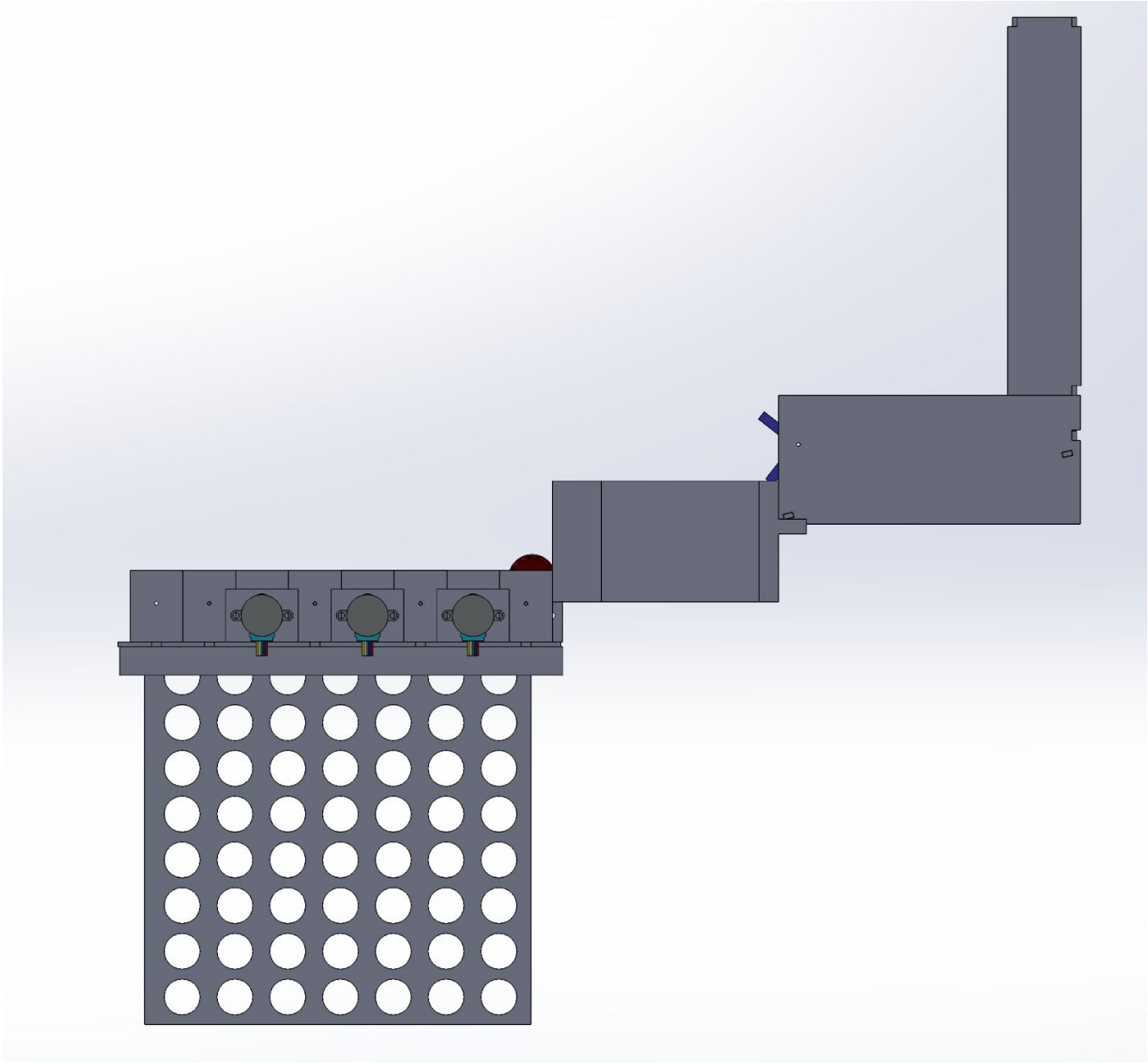
Appendix B: Conceptual drawing of the machine



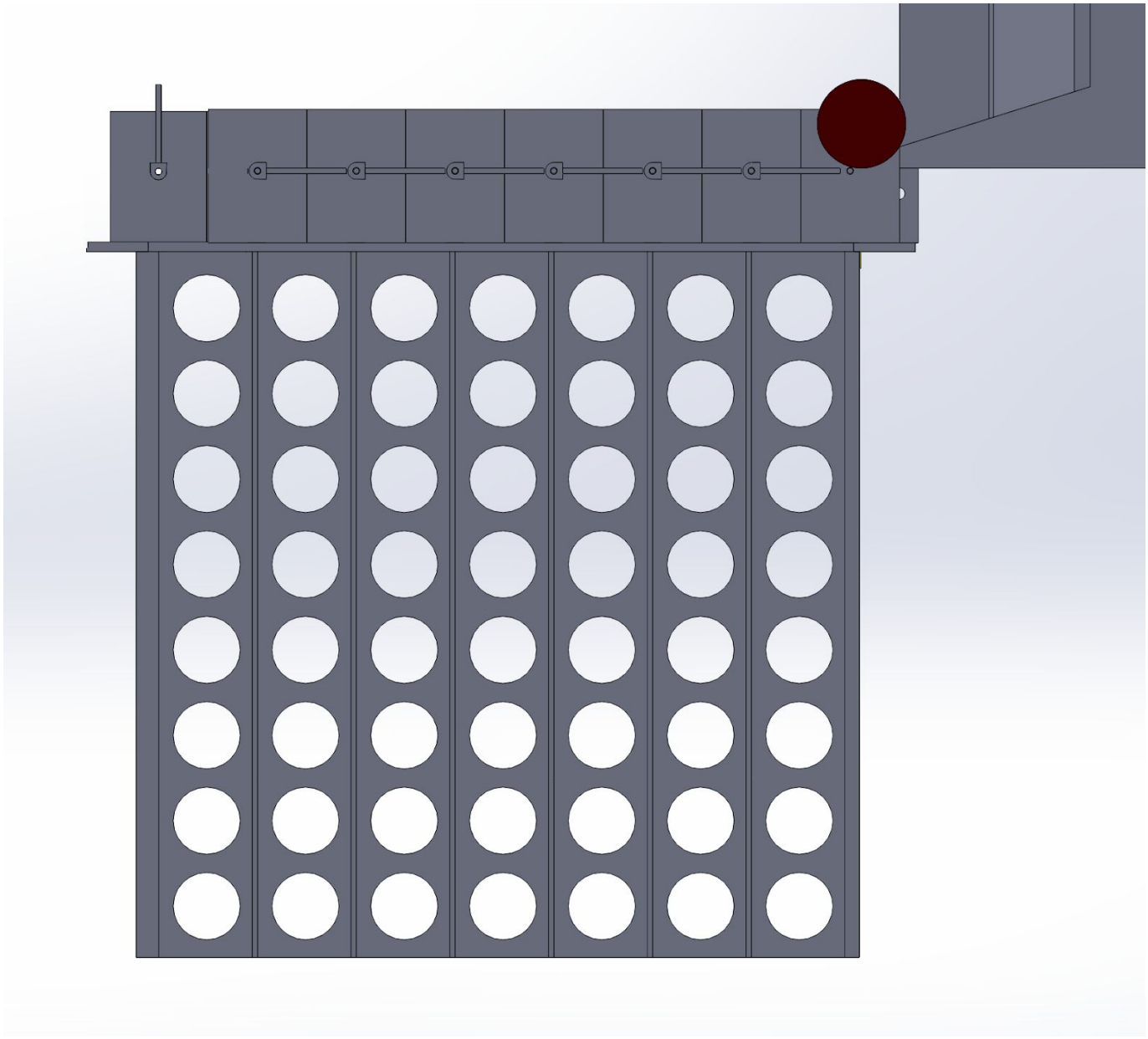
Detailed Drawings

Full assembly:

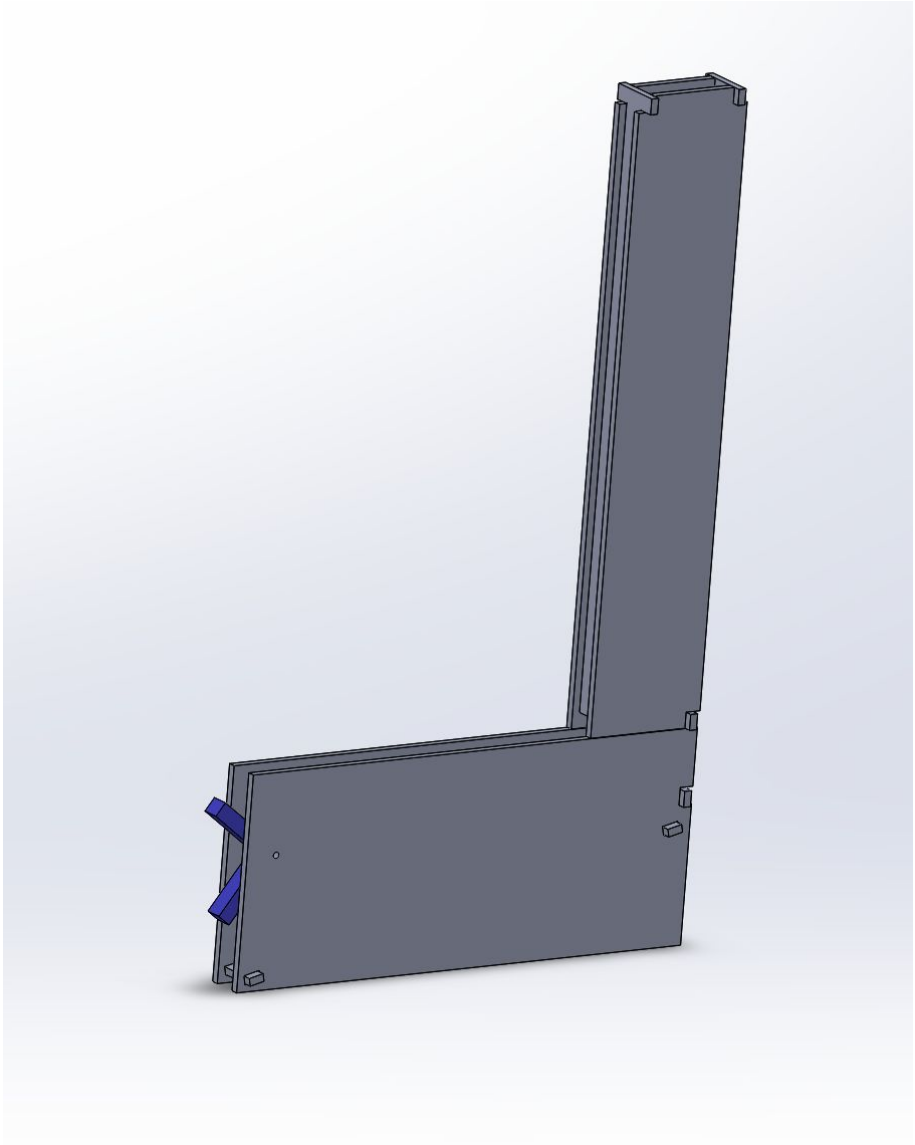




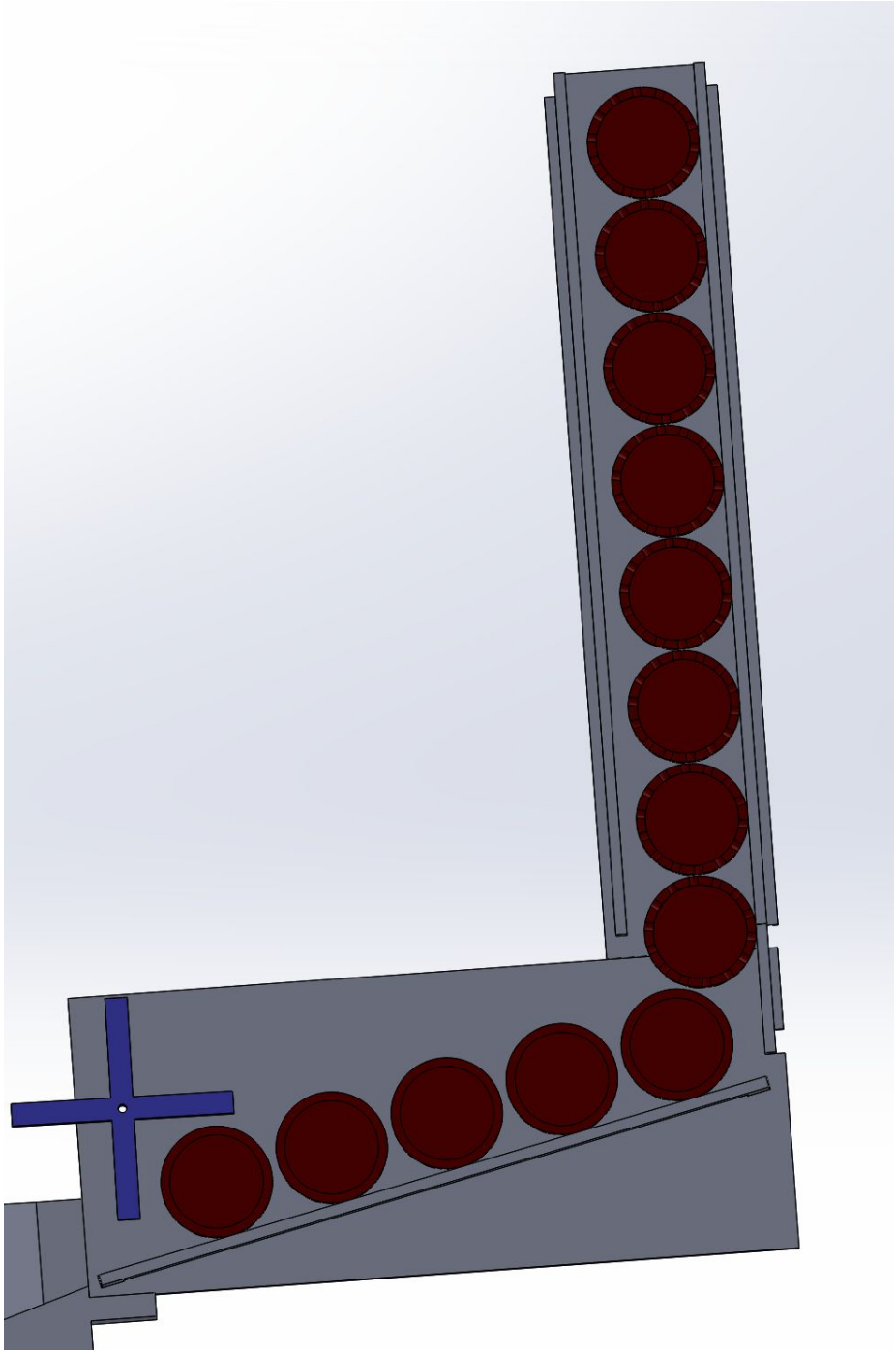
Flaps above the columns (showing a desired move into the seventh column):



Dispenser drawings



:



Appendix C: References

- Arduino Push button reference
    - http://forum.arduino.cc/index.php?topic=178999.0 multi stepper motor control diagram
- AI Code (Expectimax search with alpha-beta pruning) borrowed from Ryan Marcus
    - https://github.com/RyanMarcus/connect4

Appendix D: Description of original work for this project

- As previously stated, this idea was not created using any references, we wanted to make a robot that plays a game, and connect 4 was chosen due to its speed in demonstration (a full game of connect 4 take only a few minutes), and low complexity for AI development.
- The two dispensers, and the seven cover slots/tracks will be custom designed by us. Most pieces were laser cut and assembled by us. The flaps are also custom designed, and 3d printed by us in the makerspace
- The arduino configuration between the motors, as well as the serial and bluetooth wireless code used to communicate between different boards were written by us
- The wireless controller, which consists of an LCD screen, arduino uno, and a bluetooth module, was custom wired and programmed by us

Appendix E: Code for Arduinos

Code for motor arduino

```
/*
Team 43
Team Members:
Li Chen Lchen79
Bharat Middha bmiddh2
Dane Zieman dziema2
Name: Automated Connect 4 Playing Robot
must have the following libraries
<AccelStepper.h>

for this code to compile

Abstract:
This project is a robot that plays the game of connect
four against a human automatically. It takes the users move
from a button, one for each of the seven columns a move
can be made in. After the player selects their move,
the robot will use AI to decide which column to place its
move in. All moves will be made with a series of game piece
dispensers and mechanical tracks/slots to dictate which
column the dispensed piece falls into.
*/
#include <AccelStepper.h>
#define HALFSTEP 4

#define NUM_COLUMNS 7
#define NUM_DISPENSERS 2
#define MAX_SPEED 10000
#define ACCELARATION 300
#define SPEED 300

AccelStepper column1(HALFSTEP, 22, 24, 23, 25);
AccelStepper column2(HALFSTEP, 26, 28, 27, 29);
AccelStepper column3(HALFSTEP, 30, 32, 31, 33);
AccelStepper column4(HALFSTEP, 34, 36, 35, 37);
AccelStepper column5(HALFSTEP, 38, 40, 39, 41);
AccelStepper column6(HALFSTEP, 42, 44, 43, 45);
AccelStepper column7(HALFSTEP, 46, 48, 47, 49);
AccelStepper dispenser1(HALFSTEP, 2, 4, 3, 5);
AccelStepper dispenser2(HALFSTEP, 8, 10, 9, 11);

byte direction[7] = {1, 1, -1, -1, 1, 1, -1};
```

```
AccelStepper *columns[NUM_COLUMNS] = {
  &column1,
  &column2,
  &column3,
  &column4,
  &column5,
  &column6,
  &column7
};

AccelStepper *dispensers[NUM_DISPENSERS] = {
  &dispenser1,
  &dispenser2
};


void initSteppers() {
  for (byte col = 0; col < NUM_COLUMNS; col++) {
    columns[col]->setSpeed(SPEED);
    columns[col]->setMaxSpeed(MAX_SPEED);
    columns[col]->setAcceleration(ACCELARATION);
  }
  for (byte disp = 0; disp < NUM_DISPENSERS; disp++) {
    dispensers[disp]->setSpeed(SPEED);
    dispensers[disp]->setMaxSpeed(MAX_SPEED);
    dispensers[disp]->setAcceleration(ACCELARATION);
  }
}

void openColumn(AccelStepper *s, int dir, char disp) {
  s->move((dir) * (512));
  s->runToPosition();
  if (disp >= 'x' && disp <= 'y') openDispenser(dispensers[(int)(disp - 'x')]);
  delay(5000);
  s->move((dir) * (-512));
  s->runToPosition();
  Serial.println("Z");
}

void openDispenser(AccelStepper *s) {
  s->move(-512);
  s->runToPosition();
}

void setup() {
  Serial.begin(115200);
  Serial.begin(115200);
```

```
  initSteppers();
}

void loop() {
  String input;
  input = Serial.readString();
  char col = input[0];
  char disp = input[2];

  if (col >= 'a' && col <= 'g')
    openColumn(columns[col - 'a'], direction[col - 'a'], disp);
}
```

Code for controller arduino

```
/*
Team 43
Team Members:
Li Chen Lchen79
Bharat Middha bmiddh2
Dane Zieman dziema2
Name: Automated Connect 4 Playing Robot
must have the following libraries
<RF24.h>
"Adafruit_MCP23017.h"
<LiquidCrystal.h>

for this code to compile

Abstract:
This project is a robot that plays the game of connect
four against a human automatically. It takes the users move
from a button, one for each of the seven columns a move
can be made in. After the player selects their move,
the robot will use AI to decide which column to place its
move in. All moves will be made with a series of game piece
dispensers and mechanical tracks/slots to dictate which
column the dispensed piece falls into.
*/

#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>
#include "Adafruit_MCP23017.h"
#include <LiquidCrystal.h>
```

```
const byte RS = 10, E  = 9, D4 = 5, D5 = 4, D6 = 3, D7 = 2;
const byte columnButtons[7] = { 2, 3, 4, 5, 6, 7, 8 };
const byte btnA = 0, btnB = 1;
const byte address[6] = "00001";

RF24 radio(7, 8);
LiquidCrystal lcd(RS, E, D4, D5, D6, D7);
Adafruit_MCP23017 mcp;

void sendRF(int btn) {
  int data;
  if (btn <= 8 && btn >= 2) data = ('a' + (8 - btn));
  if (btn <= 1 && btn >= 0) data = ('x' + (1 - btn));
  Serial.println(data);
  radio.write(&data, sizeof(data));
}

void setup() {
  Serial.begin(115200);

  lcd.begin(16, 2);
  lcd.print(" Shin Ramyun Ai ");
  lcd.setCursor(0,1);
  lcd.print("=== Connect4 ===");

  mcp.begin();

  for (int i = 0; i <= 9; i++) {
    mcp.pinMode(i, INPUT);
    mcp.pullUp(i, HIGH);
  }

  radio.begin();
  radio.openWritingPipe(address);
  radio.openReadingPipe(0, address);
  radio.setPALevel(RF24_PA_MAX);
  radio.stopListening();
}

void loop() {
  for (int i = 0; i <= 9; i++) {
    if (mcp.digitalRead(i) == LOW) sendRF(i);
  }
}
```

Code for Master Arduino (With AI)

```
/*
Team 43
Team Members:
Li Chen Lchen79
Bharat Middha bmiddh2
Dane Zieman dziema2
Name: Automated Connect 4 Playing Robot
must have the following libraries
<SPI.h>
<nRF24L01.h>
<RF24.h>
<limits.h>
<stdlib.h>
<stdio.h>
<string.h>
<stdbool.h>


for this code to compile

Abstract:
This project is a robot that plays the game of connect
four against a human automatically. It takes the users move
from a button, one for each of the seven columns a move
can be made in. After the player selects their move,
the robot will use AI to decide which column to place its
move in. All moves will be made with a series of game piece
dispensers and mechanical tracks/slots to dictate which
column the dispensed piece falls into.
*/
#include <limits.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

#define OFF_BOARD -2
#define EMPTY -1
#define LOOK_AHEAD 5
#define TABLE_SIZE 32000
#define TABLE_BIN_SIZE 10

#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>
```

```
RF24 radio(7, 8);
const byte address[6] = "00001";
int currentDispenser = 0;

typedef struct {
  int width;
  int height;
  int* board;
  int last_move;
  int weight;

  int refs;
} GameState;

typedef struct {
  GameState*** bins;
} TranspositionTable;

typedef struct {
  GameState* gs;
  int player;
  int other_player;
  int turn;

  int alpha;
  int beta;

  int best_move;

  TranspositionTable* ht;
} GameTreeNode;

GameState* globalState;
GameTreeNode* g_node = NULL;

void setup() {
 Serial.begin(115200);
 radio.begin();
 radio.openReadingPipe(0, address);
 radio.openWritingPipe(address);
 radio.startListening();
 //intiate game board
 startNewGame();
}


void moveHandler(int data) {
 Serial.println(String((char)data) + "," + String((char)(currentDispenser + 'x')));
```

```
 while (true) {
  while (!Serial.available()) {}
  while (radio.available()) {
   int garbage = 0;
   radio.read(&garbage, sizeof(garbage));
  }
  String resp = Serial.readString();
  if (resp[0] == 'Z') {
   break;
  }
 }

 currentDispenser = (currentDispenser + 1) % 2;
}

void loop() {
 if (radio.available()) {
  int data;

  radio.read(&data, sizeof(data));
  if (data >= 'a' && data <= 'g') {
   moveHandler(data);
   playerMove(data);
   checkWin(globalState);

    int ai_move = computerMove(7) + 'a';
   moveHandler(ai_move);
    checkWin(globalState);
  }
  if (data == 'x')
   resetGame();
 }
}

GameState* newGameState(int width, int height) {
  int i;
  GameState* toR = (GameState*) malloc(sizeof(GameState));

  if (toR == NULL)
    return NULL;


  toR->width = width;
  toR->height = height;

  toR->weight = 0;
  toR->refs = 1;
  toR->last_move = 0;
```

```
  toR->board = (int*) malloc(sizeof(int) * width * height);
  if (toR->board == NULL) {
    free(toR);
    return NULL;
  }


  /*@+forloopexec@*/
  for (i = 0; i < width * height; i++) {
    toR->board[i] = EMPTY;
  }
  /*@=forloopexec@*/

  return toR;
}

void freeGameState(/*@owned@*/ GameState* gs) {
  gs->refs--;
  if (gs->refs <= 0) {
    free(gs->board);
    free(gs);
  }
}

void retainGameState(GameState* gs) {
  gs->refs++;
}

int at(GameState* gs, int x, int y) {
  if (x < 0 || y < 0)
    return OFF_BOARD;

  if (x >= gs->width || y >= gs->height)
    return OFF_BOARD;

  return gs->board[x * gs->height + y];
}

void drop(GameState* gs, int column, int player) {
  int i;
  for (i = 0; i < gs->height; i++) {
    if (at(gs, column, i) == EMPTY) {
      gs->board[column * gs->height + i] = player;
      gs->last_move = column;
      return;
    }
```

```
  }


}

int checkAt(GameState* gs, int x, int y) {
  // check across
  bool found = true;
  int curr = at(gs, x, y);
  int i;
  for (i = 0; i < 4; i++) {
    if (at(gs, x + i, y) != curr) {
      found = false;
      break;
    }
  }

  if (found && (curr != EMPTY && curr != OFF_BOARD))
    return curr;

  // check down
  found = true;
  for (i = 0; i < 4; i++) {
    if (at(gs, x, y + i) != curr) {
      found = false;
      break;
    }
  }

  if (found && (curr != EMPTY && curr != OFF_BOARD))
    return curr;

  // check diag +/+
  found = true;
  for (i = 0; i < 4; i++) {
    if (at(gs, x + i, y + i) != curr) {
      found = false;
      break;
    }
  }

  if (found && (curr != EMPTY && curr != OFF_BOARD))
    return curr;

  // check diag -/+
  found = true;
  for (i = 0; i < 4; i++) {
    if (at(gs, x - i, y + i) != curr) {
```

```
      found = false;
      break;
    }
  }

  if (found && (curr != EMPTY && curr != OFF_BOARD))
    return curr;

  return 0;
}

int getIncrementForArray(int* arr, int player) {
  int toR = 0;
  int i;
  for (i = 0; i < 4; i++) {
    if (arr[i] == player) {
      toR = 1;
      continue;
    }

    if (arr[i] != player && arr[i] != EMPTY) {
      return 0;
    }
  }

  return toR;
}

int countAt(GameState* gs, int x, int y, int player) {

  // check across
  int found = 0;
  int buf[4];
  int i;

  for (i = 0; i < 4; i++) {
    buf[i] = at(gs, x + i, y);
  }

  found += getIncrementForArray(buf, player);

  // check down
  for (i = 0; i < 4; i++) {
    buf[i] = at(gs, x, y + i);
  }
  found += getIncrementForArray(buf, player);

  // check diag +/+
```

```
  for (i = 0; i < 4; i++) {
    buf[i] = at(gs, x + i, y + i);
  }
  found += getIncrementForArray(buf, player);

  // check diag -/+
  for (i = 0; i < 4; i++) {
    buf[i] = at(gs, x - i, y + i);
  }
  found += getIncrementForArray(buf, player);

  return found;
}

bool getWinner(GameState* gs) {
  int x, y;
  bool res;
  for (x = 0; x < gs->width; x++) {
    for (y = 0; y < gs->height; y++) {
      res = (bool) checkAt(gs, x, y);
      if (res)
        return res;
    }
  }

  return 0;
}

int isDraw(GameState* gs) {
  int x, y;
  for (x = 0; x < gs->width; x++) {
    for (y = 0; y < gs->height; y++) {
      if (at(gs, x, y) == EMPTY)
        return 0;
    }
  }

  return 1;
}

int getHeuristic(GameState* gs, int player, int other_player) {
  int toR = 0;
  int x, y;
  for (x = 0; x < gs->width; x++) {
    for (y = 0; y < gs->height; y++) {
      toR += countAt(gs, x, y, player);
      toR -= countAt(gs, x, y, other_player);
    }
```

```
  }

  return toR;
}

int canMove(GameState* gs, int column) {
  int y;
  for (y = 0; y < gs->height; y++) {
    if (at(gs, column, y) == EMPTY)
      return 1;
  }

  return 0;
}

/*@null@*/
GameState* stateForMove(GameState* orig, int column, int player) {
  GameState* toR;
  if (orig == NULL || orig->board == NULL)
    return NULL;



  toR = newGameState(orig->width, orig->height);
  if (toR == NULL)
    return NULL;



  memcpy(toR->board, orig->board, sizeof(int) * orig->width * orig->height);
  drop(toR, column, player);
  return toR;
}

void printGameState(GameState* gs) {
  int i, x, y, toP;
  for (i = 0; i < gs->width; i++) {
    printf("%d ", i);
  }

  printf("\n");

  for (y = gs->height - 1; y >= 0 ; y--) {
    for (x = 0; x < gs->width; x++) {
      toP = at(gs, x, y);
      if (toP == EMPTY) {
        printf("  ");
      } else {
        printf("%d ", toP);
```

```
      }
    }
    printf("\n");
  }

  for (i = 0; i < gs->width; i++) {
    printf("%d ", i);
  }

  printf("\n\n");
}

unsigned long long hashGameState(GameState* gs) {
  unsigned long long hash = 14695981039346656037Lu;
  int i;
  for (i = 0; i < gs->width * gs->height; i++) {
    hash ^= gs->board[i];
    hash *= 1099511628211Lu;
  }
  return hash;
}

int isGameStateEqual(GameState* gs1, GameState* gs2) {
  int i;
  if (gs1->width != gs2->width || gs1->height != gs2->height)
    return 0;

  for (i = 0; i < gs1->width * gs1->height; i++) {
    if (gs1->board[i] == gs2->board[i])
      continue;
    return 0;
  }

  return 1;
}

TranspositionTable* newTable() {
  int i, j;
  TranspositionTable* toR = (TranspositionTable*) malloc(sizeof(TranspositionTable));
  toR->bins = (GameState***) malloc(sizeof(GameState**) * TABLE_SIZE);
  for (i = 0; i < TABLE_SIZE; i++) {
    toR->bins[i] = (GameState**) malloc(sizeof(GameState*) * TABLE_BIN_SIZE);
    for (j = 0; j < TABLE_BIN_SIZE; j++) {
      toR->bins[i][j] = NULL;
    }
  }

  return toR;
```

```
}

GameState* lookupInTable(TranspositionTable* t, GameState* k) {
  int hv = hashGameState(k) % TABLE_SIZE;
  int i;
  GameState** bin = t->bins[hv];
  for (i = 0; i < TABLE_BIN_SIZE; i++) {
    if (bin[i] == NULL) {
      return NULL;
    }

    if (isGameStateEqual(k, bin[i])) {
      return bin[i];
    }
  }
  return NULL;
}

void addToTable(TranspositionTable* t, GameState* k) {
  int hv = hashGameState(k) % TABLE_SIZE;
  int i;
  GameState** bin = t->bins[hv];
  for (i = 0; i < TABLE_BIN_SIZE; i++) {
    if (bin[i] == NULL) {
      bin[i] = k;
      retainGameState(k);
      return;
    }
  }

  fprintf(stderr, "Overflow in hash bin %d, won't store GameState\n", hv);
}

void freeTranspositionTable(TranspositionTable* t) {
  int i, j;
  for (i = 0; i < TABLE_SIZE; i++) {
    for (j = 0; j < TABLE_BIN_SIZE; j++) {
      if (t->bins[i][j] != NULL)
        freeGameState(t->bins[i][j]);
    }
    free(t->bins[i]);
  }
  free(t->bins);

  free(t);
}
```

```
GameTreeNode* newGameTreeNode(GameState* gs, int player, int other, int turn, int alpha, int beta,
TranspositionTable* ht) {
  GameTreeNode* toR = (GameTreeNode*) malloc(sizeof(GameTreeNode));
  toR->gs = gs;
  toR->player = player;
  toR->other_player = other;
  toR->turn = turn;
  toR->alpha = alpha;
  toR->beta = beta;
  toR->best_move = -1;
  toR->ht = ht;
  return toR;
}

int heuristicForState(GameState* gs, int player, int other) {
  if (isDraw(gs))
    return 0;

  int term_stat = getWinner(gs);
  if (term_stat == player)
    return 1000;

  if (term_stat)
    return -1000;


  return getHeuristic(gs, player, other);

}


// using a global instead of qsort_r because of emscripten support

// GameTreeNode* g_node = NULL;


int ascComp(const void* a, const void* b) {
  GameTreeNode* node = g_node;
  return heuristicForState(*(GameState**) a, node->player, node->other_player) -
    heuristicForState(*(GameState**) b, node->player, node->other_player);

}

int desComp(const void* a, const void* b) {
  GameTreeNode* node = g_node;
  return heuristicForState(*(GameState**) b, node->player, node->other_player) -
    heuristicForState(*(GameState**) a, node->player, node->other_player);
```

```
}

int getWeight(GameTreeNode* node, int movesLeft) {
  int toR, move, best_weight;
  if (getWinner(node->gs) || isDraw(node->gs) || movesLeft == 0)
    return heuristicForState(node->gs, node->player, node->other_player);

  GameState** possibleMoves = (GameState**) malloc(sizeof(GameState*) * node->gs->width);
  int validMoves = 0;
  for (int possibleMove = 0; possibleMove < node->gs->width; possibleMove++) {
    if (!canMove(node->gs, possibleMove)) {
      continue;
    }

    possibleMoves[validMoves] = stateForMove(node->gs, possibleMove, (node->turn ? node->player :
node->other_player));
    validMoves++;
  }

  // order possibleMoves by the heuristic
  g_node = node;
  if (node->turn) {
    // qsort_r is apparently non-standard, and won't work with emscripten. So we'll need to use a global.
    qsort(possibleMoves, validMoves, sizeof(GameState*), ascComp);
  } else {
    qsort(possibleMoves, validMoves, sizeof(GameState*), desComp);
  }

  best_weight = (node->turn ? INT_MIN : INT_MAX);

  for (move = 0; move < validMoves; move++) {
    // see if the game state is already in the hash table
    GameState* inTable = lookupInTable(node->ht, possibleMoves[move]);
    int child_weight;
    int child_last_move;
    if (inTable != NULL) {
      child_weight = inTable->weight;
      child_last_move = possibleMoves[move]->last_move;

    } else {
      GameTreeNode* child = newGameTreeNode(possibleMoves[move], node->player, node->other_player,
!(node->turn),
                node->alpha, node->beta, node->ht);
      child_weight = getWeight(child, movesLeft - 1);
      child_last_move = child->gs->last_move;
      free(child);
    }
```

```
    possibleMoves[move]->weight = child_weight;
    addToTable(node->ht, possibleMoves[move]);

    if (movesLeft == LOOK_AHEAD)
      printf("Move %d has weight %d\n", child_last_move, child_weight);

    // alpha-beta pruning
    if (!node->turn) {
      // min node
      if (child_weight <= node->alpha) {
        // MAX ensures we will never go here
        toR = child_weight;
        goto done;
      }
      node->beta = (node->beta < child_weight ? node->beta : child_weight);
    } else {
      // max node
      if (child_weight >= node->beta) {
        // MIN ensures we will never go here
        toR = child_weight;
        goto done;
      }
      node->alpha = (node->alpha > child_weight ? node->alpha : child_weight);
    }

    if (!(node->turn)) {
      // min node
      if (best_weight > child_weight) {
        best_weight = child_weight;
        node->best_move = child_last_move;
      }
    } else {
      // max node
      if (best_weight < child_weight) {
        best_weight = child_weight;
        node->best_move = child_last_move;
      }
    }


  }
  toR = best_weight;
done:
  for (int i = 0; i < validMoves; i++) {
    freeGameState(possibleMoves[i]);
  }
```

```
  free(possibleMoves);
  return toR;
}

int getBestMove(GameTreeNode* node, int movesLeft) {
  getWeight(node, movesLeft);
  return node->best_move;
}


// END OF API

void checkWin(GameState* gs) {
  int win = getWinner(gs);

  if (win) {
    printf("Game over! %d wins!\n", win);
    printGameState(gs);
    exit(0);
  }

  if (isDraw(gs)) {
    printf("Game over! Draw!\n");
    printGameState(gs);
    exit(0);
  }

}

int bestMoveForState(GameState* gs, int player, int other_player, int look_ahead) {
  TranspositionTable* t1 = newTable();
  GameTreeNode* n = newGameTreeNode(gs, player, other_player, 1, INT_MIN, INT_MAX, t1);
  int move = getBestMove(n, look_ahead);
  free(n);
  freeTranspositionTable(t1);
  return move;
}

// a couple of ease-of-use functions that will run a game in global state

void startNewGame() {
  globalState = newGameState(7, 6);
}

void playerMove(int move) {
  drop(globalState, move, 1);
}
```

```
int computerMove(int look_ahead) {
  int move = bestMoveForState(globalState, 2, 1, look_ahead);
  drop(globalState, move, 2);
 return move;
}

int isGameWon() {
  return getWinner(globalState);
}

int isGameDraw() {
  return isDraw(globalState);
}

int isEmpty(int x, int y) {
  return at(globalState, x, y) == EMPTY;
}

int pieceAt(int x, int y) {
  return at(globalState, x, y);
}

void resetGame(){
 freeGameState(globalState);
 startNewGame();
}
```