Week 13

This Week:

Bounce Location Tracking:

This week I wanted to pair the position tracking with UART communication to get an accurate estimate for the location of a bounce. The trickiest part of pairing the lower speed sampling of the UART message with the higher speed sampling of the camera. With the naïve testing I did at the end of last week, sampling the ball position as soon as we receive a message, caused problems: the position of the bounce would always be inaccurate as the position was always delayed. To solve this I set out to collect all position points, and average the position of the ball between consecutive UART messages. The first order of action to accomplish this was to upgrade the UART decoder to store the time of the current message and the last message:

```
// Lines added to UartDecoder.readSerial()
decoder.last_time = decoder.curr_time;
decoder.curr_time = std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::system_clock::now().time_since_epoch()).count();
```

Now with the UART decoder upgraded, I upgraded the Table class. The first change was creating functions to start a sampling thread. With C++ multithreading we can keep a thread live that continuously samples ball position data at a max sampling speed:

```
void Table::detectionThread(){
    Table &table = *this;
    std::chrono::milliseconds time_offset = std::chrono::milliseconds(1000/table.sampleFreq);
    std::chrono::milliseconds lastSample;
    lastSample = std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::system_clock::now().time_since_epoch());
    std::chrono::milliseconds currSample;
    for(;;){
        currSample = std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::system_clock::now().time_since_epoch());
  if(stopSample)
      return;
        if(currSample - lastSample >= time_offset){
            lastSample = currSample;
            BounceStore currBallPos = BounceStore(table.getNormalizedCoords(),currSample.count());
      if(currBallPos.loc.x < 0){
    continue;
      }
      table.bounceListI++;
      table.bounceList.push_front(currBallPos);
        }
    }
}
```

As seen in this code segment, the ball position data is now stored in a doubly linked list, using an new object to store both the position and the time of the sample:

```
BounceStore(cv::Point2f,int);
        cv::Point2f loc;
        unsigned int time;
};
```

The list stores the most recent value at the front of the list. I thought for a while about creating a circular buffer to store a fixed amount of bounces and remove stale data. After discussing with James, I opted to just store all data that will be removed when the thread stops. This is not ideal as our list will grow in size until the game is stopped. However, the structure of the bounce store is not very large in size and the game time will be at most 5

minutes, so for now, we will accept this fault, and optimize in the future if we run into issues.

When getting the average position in a time frame we start at the beginning of the list (the most recent data point) and loop until the end of the time frame:

```
cv::Point2f Table::getAveragedPos(int startTime, int stopTime){
    Table &table = *this;
    float xSum = 0;
    float ySum = 0;
    int nSamples = 0;
    std::list<BounceStore>::iterator it;
    std::cout << startTime << " to " << stopTime << std::endl;
    for (it = table.bounceList.begin(); it != table.bounceList.end(); it++){
  //std::cout << it->time << std::endl;
  if(it->time > stopTime)
      continue;
  if(it->time < startTime)
      break;
  //std::cout << it->loc.x << std::endl;
  xSum += it->loc.x;
  ySum += it->loc.y;
  nSamples++;
    }
    if(nSamples == 0){
  return cv::Point2f(-1,-1);
    }
    return cv::Point2f(xSum/nSamples, ySum/nSamples);
}
```

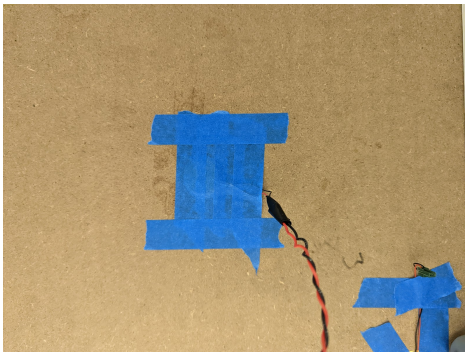
The `getAveragePos()` function is called with the time range collected from UartDecoder. The test main I wrote for the table is as follows.:

```
#ifdef TESTTABLE
int main(int argc, char ** argv){
    Projector proj = Projector(1536,768);
    CameraInterface cam = CameraInterface();
    ColorTracker colTrack = ColorTracker();
    ContourTracker conTrack = ContourTracker();
    Table table = Table(cam, colTrack, conTrack, 10);
    std::string deviceStr = "/dev/ttyUSB0";
    UartDecoder uart = UartDecoder(deviceStr);
    if(uart.serial_port == 0){
        std::cout << "Problem Setting Up Serial Port" << std::endl;
        return 1;
    }
    table.setTableBorder();
    table.startDetection();
    cv::Point2f currBounce = cv::Point2f(0,0);
    cv::Point2f projPos;
    cv::Point2f prevProjPos(-1, -1);
    for(;;){
        uart.readSerial();
        if (cv::waitKey(33) == 27){
      table.stopDetection();
      break;
  }
  if(uart.getBounce() == RED || uart.getBounce() == BLUE){
      std::cout << "Bounce" << std::endl;
      currBounce = table.getAveragedPos(uart.last_time, uart.curr_time);
  }
  projPos.x = currBounce.x * proj.w;
  projPos.y = currBounce.y * proj.h;
  cv::circle(proj.display,projPos, 20, cv::Scalar(225,225,225), 4);
  proj.drawLine(prevProjPos, projPos, 5, cv::Scalar(255, 127, 255));
  prevProjPos = projPos;
  if(proj.refresh()) break;
    }
    return EXIT_SUCCESS;
}
#endif
```

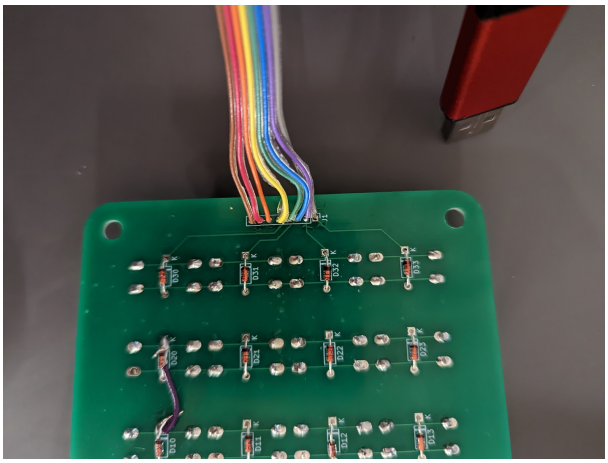
`startDetection()` and `stopDetection()` are used to initialize and end `detectionThread()`.

Hardware:

Our table tennis table and attached microphone array as undergone a fair amount of stress with the frequent packing and unpacking for testing. This week, I soldered two new piezo microphones to the table to replace broken ones and re-screwed the supporting edge of the table to the table surface. I also soldered the keypad matrix permanently to the wire bus that connects to the PCB.

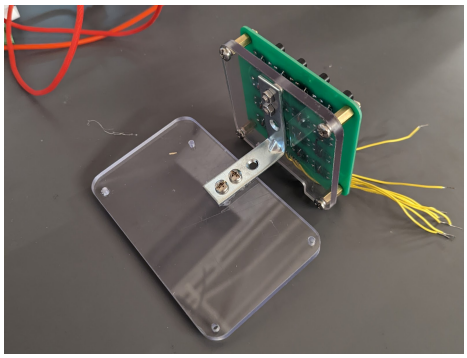


Attached Piezo Microphone



Soldered Wire Bus

Additionally here is the partially assembled housing for our PCBs



Assembled Housing minus PCB

Next Week:

Hardware:

This next week we need to construct, or start to construct the support structure to hold the projector when we present. With some advice from the lab staff we decided to construct this frame from aluminum T-track. This option will give us the modularity we need and the rigidness to properly support our camera and projector.

I also want to look into a better way of securing our microphones to the table. The current method of painters tape has worked, but is not ideal. I aim to use an adhesive to secure it, or a mechanical piece to support.

Software:

A large part of the development process at this time is integration. Bartosz has started the code for dropshot, and this coming week I will help to integrate the bounce position code I wrote this week into the game. James is working on rendering polygons to the table and I will most likely help with the logic that determines which polygon that the ball bounced in. This will be implemented with a 2D matrix of bools that store the state of the cell in the current game.