Summer 8-2-2022

Charger Robotic Air hockey-Modified Air Hockey Table Featuring Automated Play Against a Human Opponent

Robert Geren
University of Alabama in Huntsville

Charger Robotic Air Hockey - Modified Air Hockey Table Featuring Automated Play Against a Human Opponent

by

Robert Geren

Additional Authors: John Blankenship, Jackson Cobble, Kyle Grantland, and Zander Winston

An Honors Capstone

submitted in partial fulfillment of the requirements

for the Honors Diploma

to

The Honors College

of

The University of Alabama in Huntsville

August 2, 2022

Honors Capstone Director: Dr. Earl Wells

Professor of Computer Engineering

Robert Geren	August 2, 2022
Student	Date
B. Earl Wells	August 9, 2022
Director	Date
Tommy Morris	August, 10, 2022
Department Chair	Date
William S Wilkerson Digitally signed by William S Wilkerson Date: 2022.08.11 15:08:51 -05'00'	
Honors College Dean	Date

HONORS COLLEGE
THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Honors College
Frank Franz Hall
+1 (256) 824-6450 (voice)
+1 (256) 824-7339 (fax)
honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis.

In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department, Director of the Program, or the Dean of the Honors College. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

Robert Geren

Student Name (printed)

Robert Geren

Student Signature

August 2, 2022

Date

TABLE OF CONTENTS

Acknowledgement:

We would like to give a special thanks to Dr. Coe for his sponsorship and to Dr. Wells for his contribution as our professor.

ABSTRACT

Our project features an air hockey table that has been modified to feature fully automated play against a human. The project is focused on showcasing the skills taught throughout our degree program, specifically on the fields of robotics and artificial intelligence. It does this by mating a robotic paddle to a Nvidia Jetson Nano 2GB running a computer vision system tracking the puck on the table. This provides a well-defined system to showcase our abilities in hardware/software integration in addition to providing experimentation with neural networks as applied to computer vision.

Initially, there were substantial difficulties during the process of getting the entire system running well enough to play against a human. However, integrating the hardware robotics with the software computer vision gave a great insight into the complexities of real-world systems. Similarly, the act of training a neural network to track the puck taught us a great deal about how these systems are used and what considerations to keep in mind when using them. Finally, the process of optimizing the system so that it could run with reasonable latency taught us the importance of troubleshooting when customizing the system's software build, especially when dealing with specialized hardware. In all, the major accomplishments of this project were redesigning the motors and robotics system, training a neural network for computer vision using a custom training set, enabling parallelization and thus optimization of the system's latency, and tying these parts together into a unified system that took in image data and output motor drive signals.

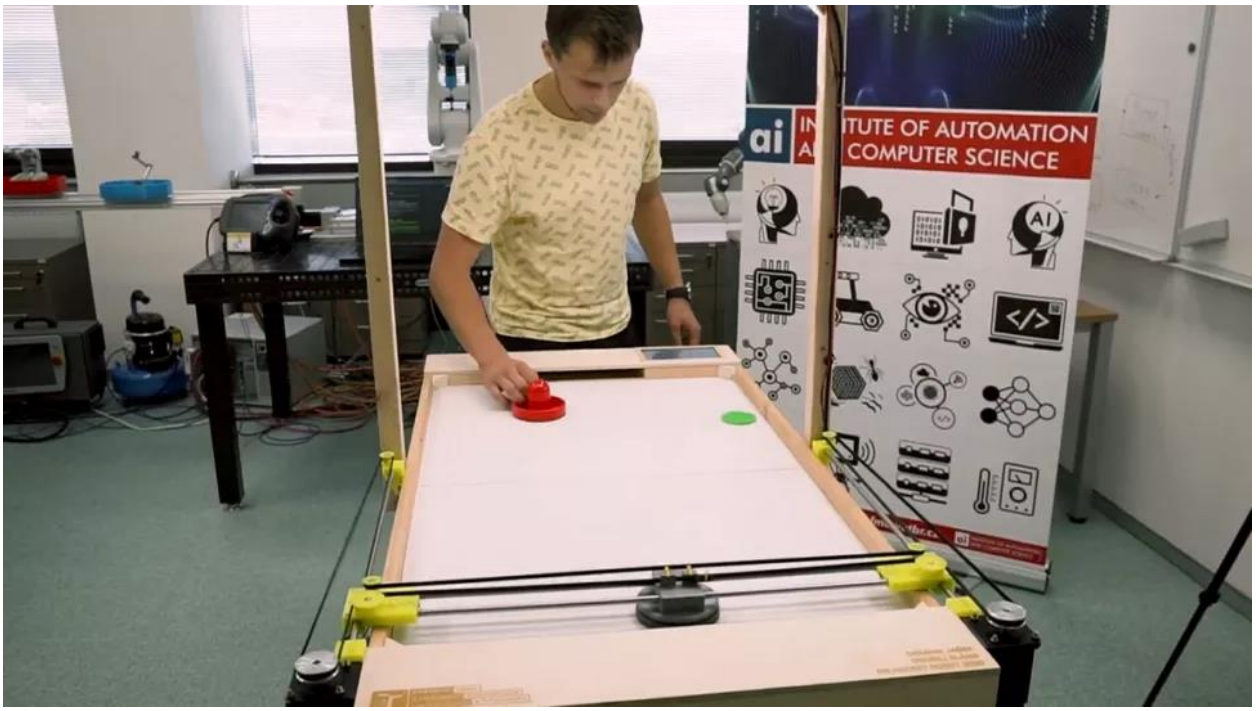
I. INTRODUCTION

A. Market Research

While this project is incredibly niche, we discovered we are not first to attempt this engineering endeavor. The earliest documentation of such a project dates back to 2008. This project was completed by the Nuvation engineering team. The most unique aspect of this air hockey playing robot is that AI controls a robotic arm that sits in front of the table like a player would. While this arm looks impressive, it is needlessly complicated and certainly more expensive than a motorized belt system.

The second project we examined was much more in line with the goals for our own project. In 2020 a group of graduate students at the Barkhausen Institute in Dresden, Germany, created an AI controlled air hockey table as a part of their master's thesis. This project featured an overhead camera that used computer vision to track the puck. Additionally, the students made

use of a belt system driven by stepper motors to move the paddle around the board and hit the puck. This project was well-executed and served as major inspiration for our own project.



B. Engineering Requirements

We came up with the following four major requirements for this project. First, the system shall be able to independently control the swing and position of the paddle on the board. Second, the system shall use computer vision software in combination with a neural network to track the location of the paddle and the puck on the board. Third, the system shall use a physics simulation to predict the future path of the puck. Finally, the system shall be able to facilitate human vs AI air hockey gameplay by combining all previous requirements.

C. Marketing Requirements

With the goal of being able to facilitate human vs robot air hockey play, our design requirements were built around one primary question, “What would our system need to do for someone to play air hockey without a partner?” This led us to forming our four engineering requirements. The procedures that a human player uses to play air hockey consist of the following: visually tracking the location of the puck, using the paddle to strike the puck towards the opposing goal, and understanding of the laws of physics to deflect the puck off the walls of the air hockey table. If we could emulate these three components within our project, then we would have all the pieces we need to facilitate meaningful gameplay.

II. Design Process

A. Image Processing

Our image processing module uses a convolutional neural network to detect the puck and the motor. Instead of creating a neural net from scratch, we implemented a pre-made architecture named YOLO (You Only Look Once) to detect our objects. The network was trained using images pulled from a live video feed which had puck and motor objects labeled with their corresponding location using the Computer Vision Annotation Tool (CVAT).

Once we obtained the training set for the puck and motor, we set up our configuration file to finetune the way the neural net would be trained and then ran the training program. Training was attempted with three different sets of parameters, each with better results than the last. The parameters we ended up with were: 453 total images of the puck, 200 total images of the motor, 64 input batch size (total numbers of inputs to feed through before updating internal CNN filters), 4000 max batches, images downscaled to 608x608 resolution, and the implementation of a learning rate scheduler.

For the CV implementation, there was the option of doing contour detection instead of running the images through a neural net. This method would be filtering the image to try and make the puck stand out with intense colors on a muted background. Then, high intensity lines would be constructed to try to find polygons, and polygons with 15+ edges could be considered a circle. This would have been potentially faster than running the images through the neural net, but the idea was scrapped due to the difficulty of finding the proper way to filter the image to remove noise and glare caused by the overhead lights on the surface of the table.

B. Air Hockey Software Simulation

To accelerate the AI training for use on our air hockey table, we created a 1:1 scale C++ simulation of said table. By implementing the physics calculations of an air hockey table in a computer program, we can simulate thousands of games of air hockey in only a few minutes. The general idea of the simulation is to have one side of the table controlled by a deterministic AI and the other side controlled by a custom-trained AI. By training our custom AI against an AI that will always move to the best possible position, we are able to maximize training efficiency and generate an effective AI that can be ported to compete against a human opponent on our physical air hockey table.

To create a virtual air hockey table, we began by creating a table class. The table class contains two paddle objects and a puck object. On initialization, this class creates an instance of each of the contained objects and assigns a controller to both paddle objects. The controller assigned to each paddle can be either a deterministic AI or a custom AI model to be trained through simulation.

The puck class was designed to handle the main physics calculations of the simulation. The puck contains a two-dimensional velocity vector, a two-dimensional position vector, a radius, table dimensions, and a reference to the left and right paddles. The puck also contains a tick function synchronized with the tick functions of both paddles' controllers. The tick functions in the program are designed to simulate a change in time called *dt* (delta time) between each function call. Because *dt* is simply a variable used in physics calculations, the application does not need to wait for *dt* seconds to elapse.

On each tick of the puck, the puck's velocity is recalculated using the formula

$$velocity = velocity * e^{-K*dt/m}$$

where *K* is the friction coefficient of the puck, *dt* is the time elapsed between ticks, and *m* is the mass of the puck. The position is then recalculated using the formula

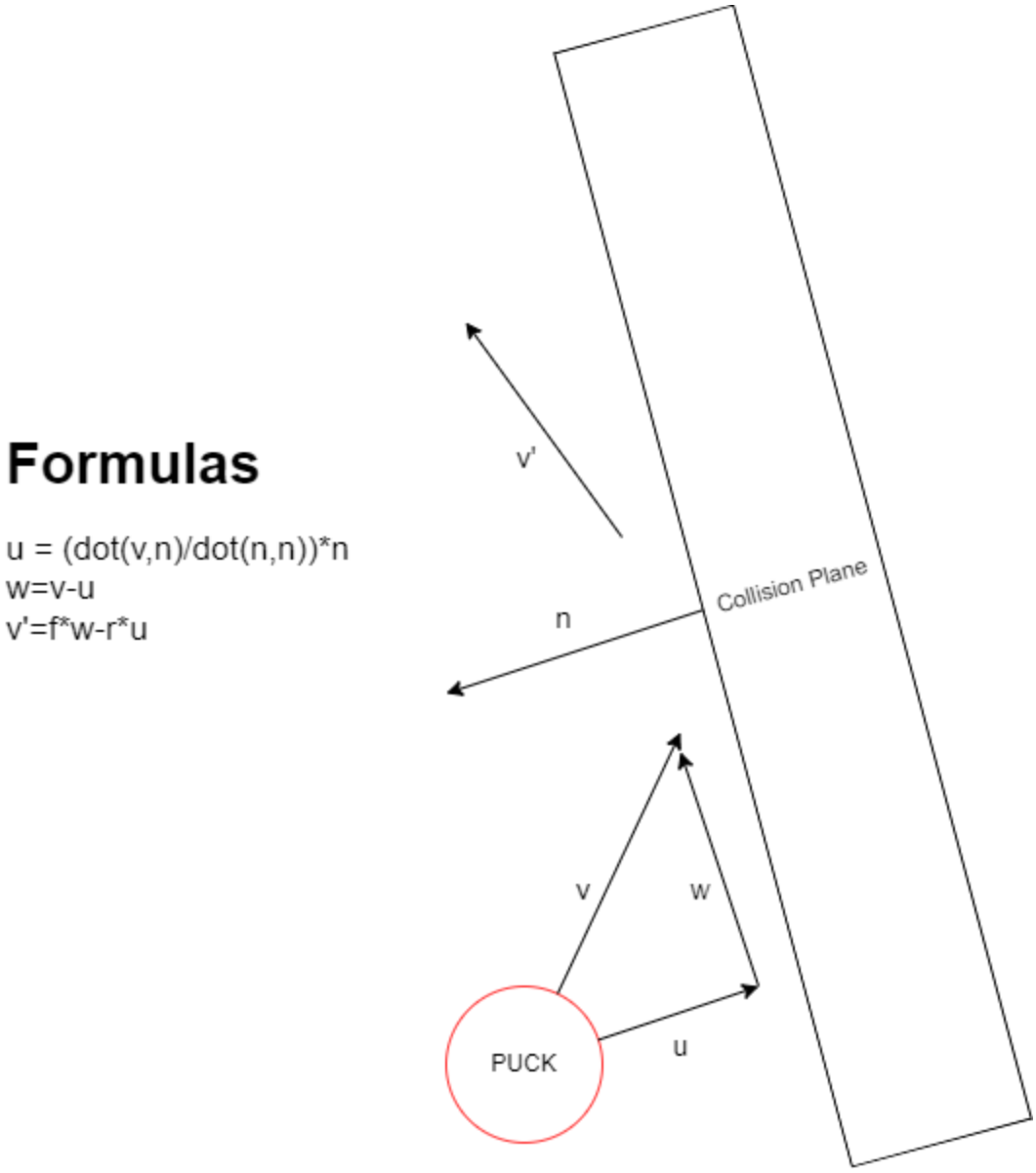
$$position = position + velocity * dt$$

and compared with the table dimensions and paddle locations to check for collisions. If a collision has occurred, the plane of the colliding object is passed into the following equations where *v* is the initial velocity of the puck, *v'* is the final velocity of the puck, *n* is the normal vector to the colliding plane, *f* is the coefficient of friction, and *r* is the coefficient of restitution.

$$u = \frac{dot(v,n)}{dot(n,n)} * n$$

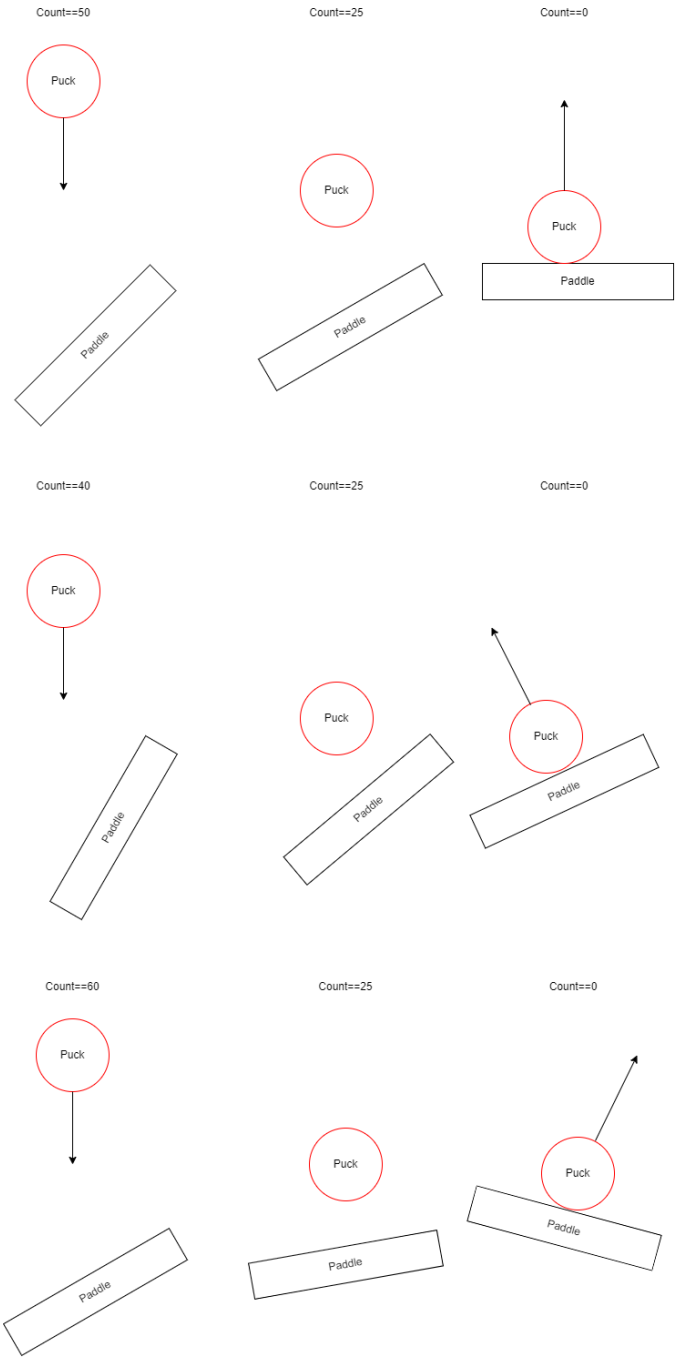
$$w = v - u$$

$$v' = f * w - r * u$$

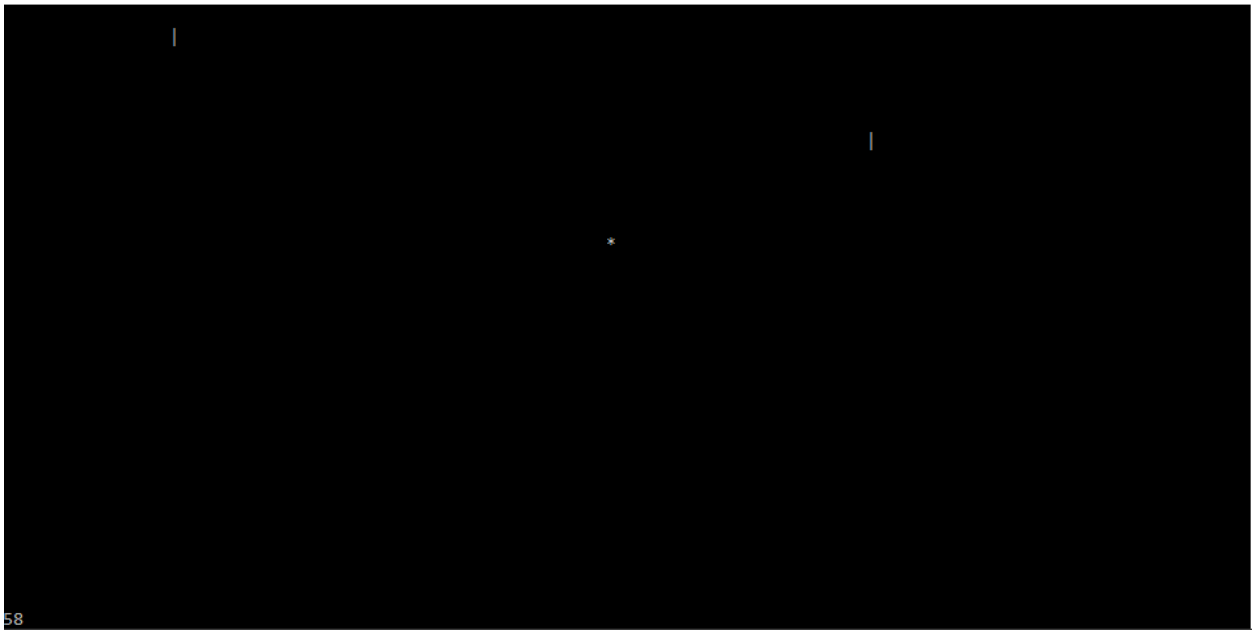


On each tick of the deterministic AI of the paddle, a copy of the puck is created. This new puck then runs its own tick function asynchronously to the actual puck. When the X location of the copied puck intersects with the X location of the paddle, the copied puck's Y location is set as the target Y location for the paddle. The paddle's tick function then moves the paddle toward the target Y location on each tick. Additionally, a count is saved for how many iterations are remaining until the puck reaches this X location. Because this saved count will be zero when

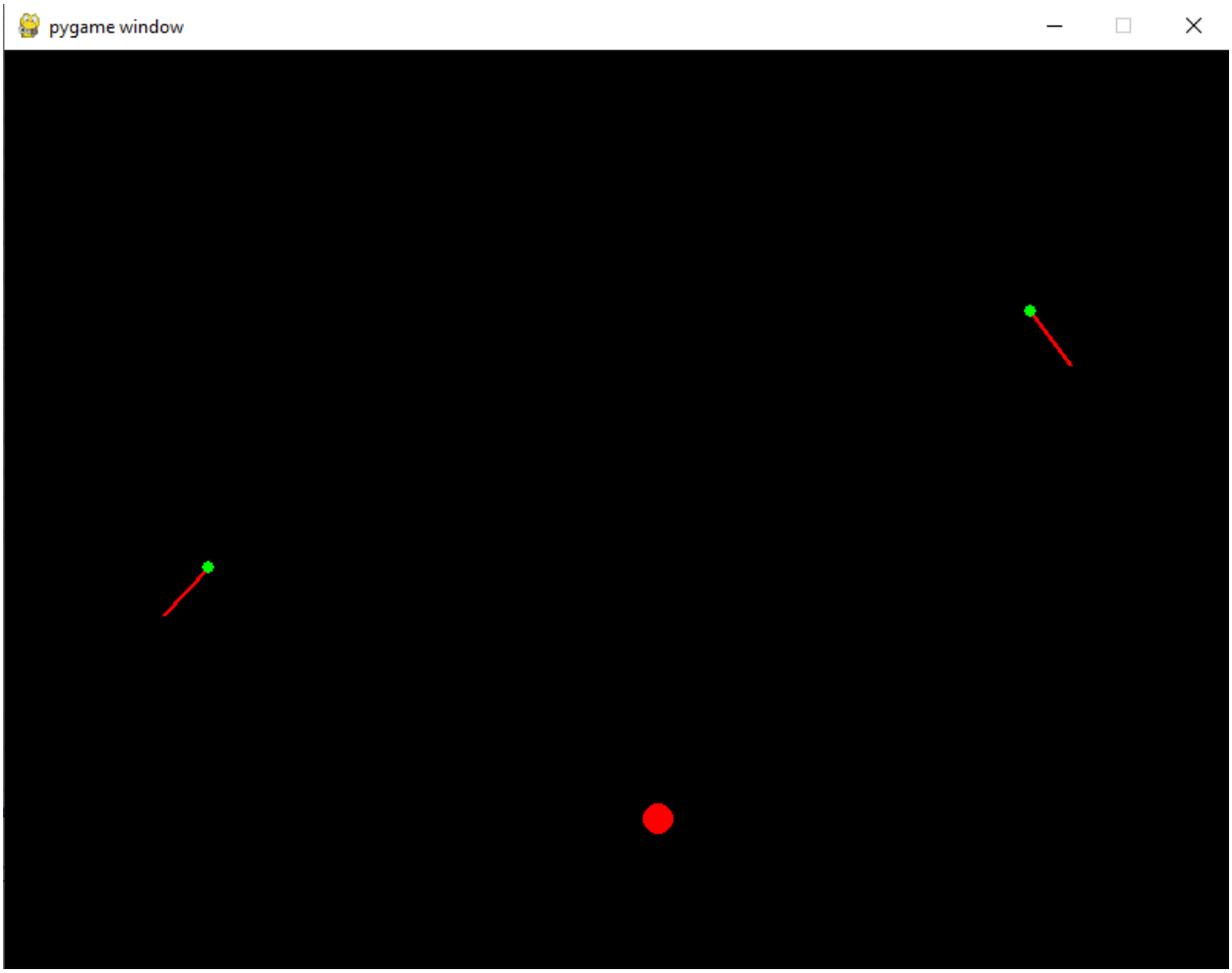
the puck reaches the paddle, the paddle begins to rotate toward the puck as the count approaches zero. To guide the direction that the puck is hit, the paddle can offset when it begins rotating relative to the count. For example, if the paddle needed to tick 50 times to rotate to zero degrees, it could begin rotating when the count equals 40 to hit the puck to the left or when the count equals 60 to hit the puck to the right.



Because the air hockey simulation was designed to quickly simulate thousands of iterations of an AI, the application was built without any attached module to display the games. For the purposes of debugging and visualization, there are two detached modules that the application can employ. The first method, which is primarily used for debugging, prints an ASCII matrix to the console which shows the location of each paddle and the puck in real time.



The second method, which is primarily used for visualization of the system's physics, logs the location of the puck as well as the location and rotation of the paddle to a comma separated variable (CSV) file. The CSV is then interpreted by a separate program to draw each simulated frame.



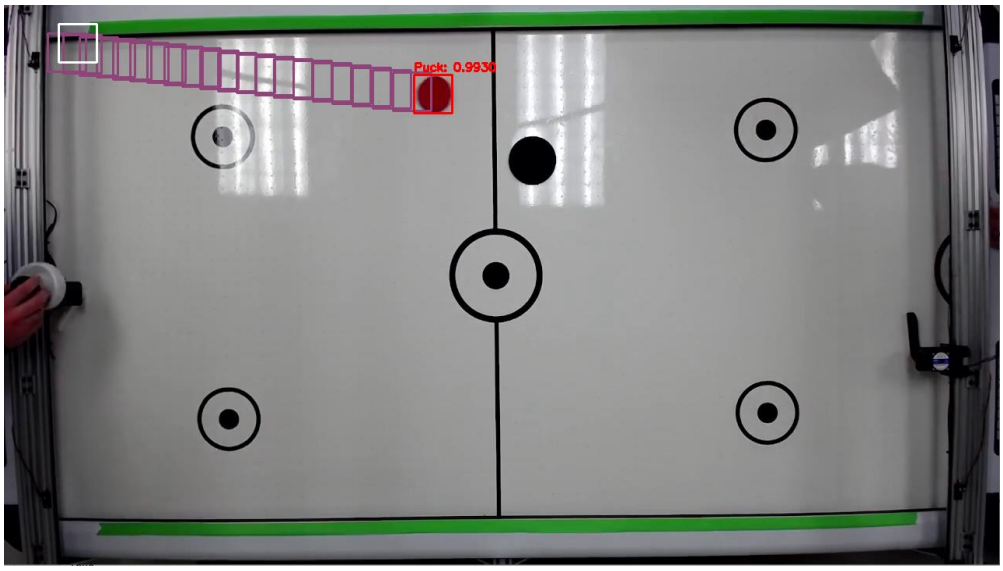
C. Air Hockey Physical Implementation

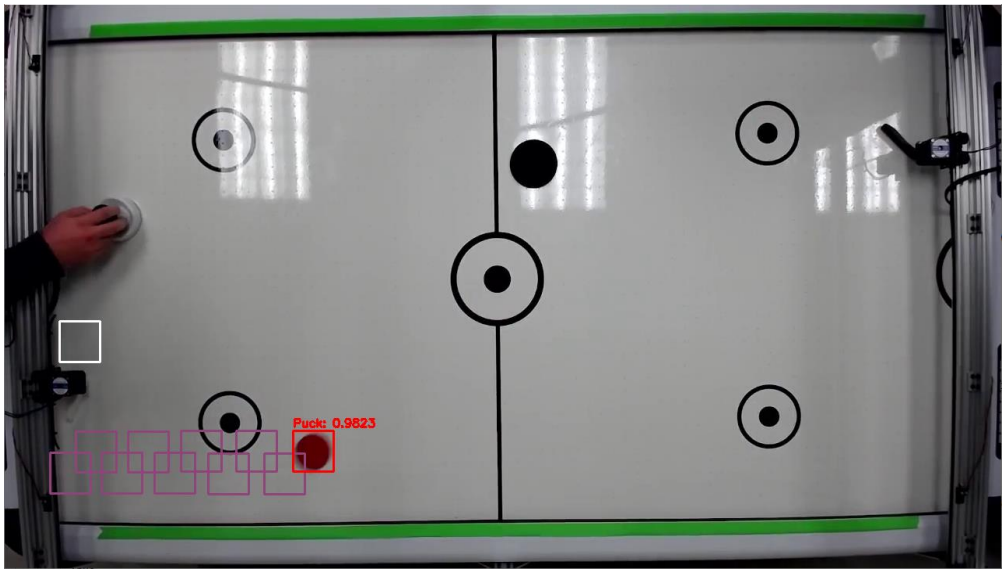
Because the air hockey simulation was designed to scale with the physical air hockey table, significant portions of the simulation code were able to be ported into Python to control the motors attached to the paddle. However, due to hardware limitations of the Jetson Nano 2GB, we were forced to simplify the physics calculations of the puck.

First, we removed the calculations for the puck colliding with the left and right boundaries of the table. These calculations were unnecessary because the real puck would already be beyond the reach of the paddle if this occurred. Second, we simplified the puck collision with the upper and lower boundary of the table to simply reflect the Y velocity of the virtual puck and multiply by the restitution coefficient, i.e., the value that defines the ratio of the initial velocity to the final velocity after a collision. This could be done without violating the physics of the system because the normal vector, or perpendicular vector, of the upper and lower boundaries of the table is close to vertical. Last, we adjusted the simulated delta time and set a hard limit on how many iterations the puck could be simulated. While this reduced the accuracy of the AI, it significantly reduced the number of operations required for every frame captured by our camera.

In addition to the above modifications, we needed to calculate the velocity at which the puck was moving. This was done by saving the position of the puck on each frame then calculating the change in position and multiplying it by the framerate of the camera. For example, if the position of the puck was [50,10] in one frame and [55,15] in the next frame with the camera running at 30 frames per second, the calculated velocity would be [(55-50)*30, (15-10)*30] or [150, 150] pixels per second.

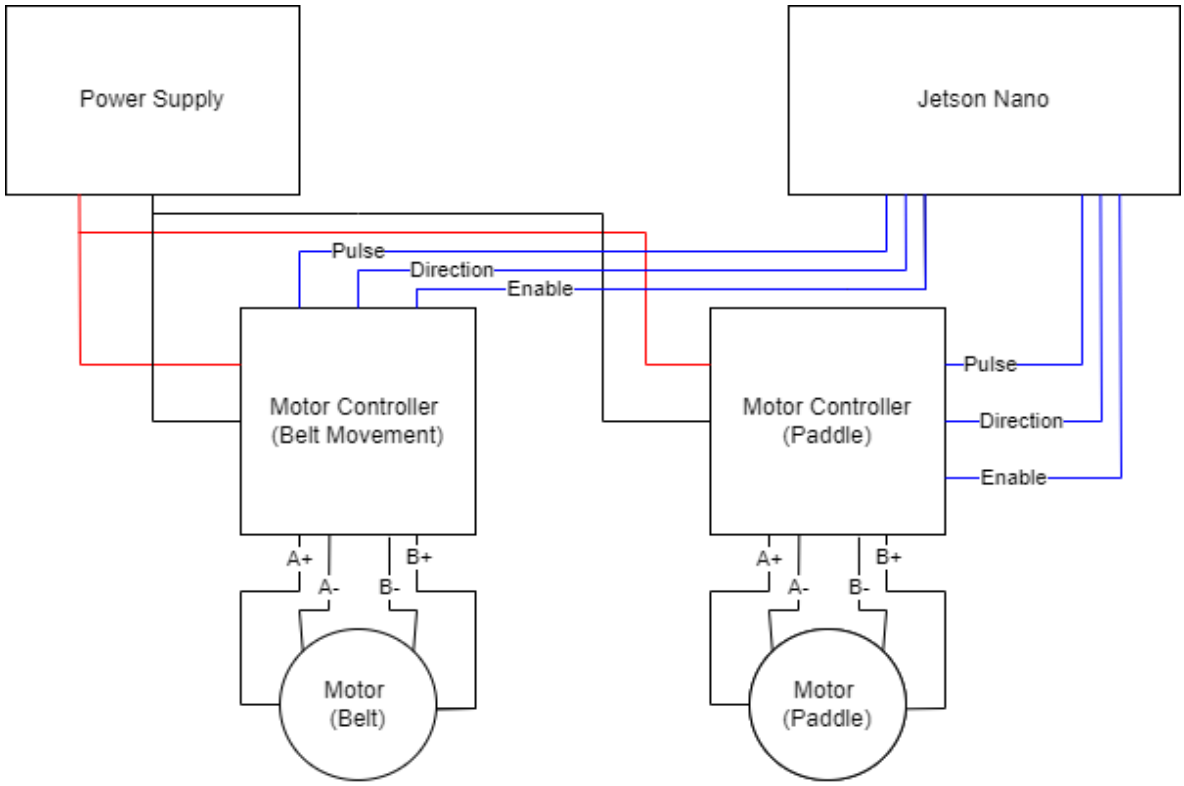
We also needed to find the upper and lower boundaries of the table. Rather than hard-coding this into our application, we saved four variables representing each of the four corners of the table. These variables are calibrated at runtime as the puck approaches these corners, and the boundaries are calculated as the line between their corresponding points. For example, the upper boundary would be any position above the line between the upper left and upper right corners of the table.





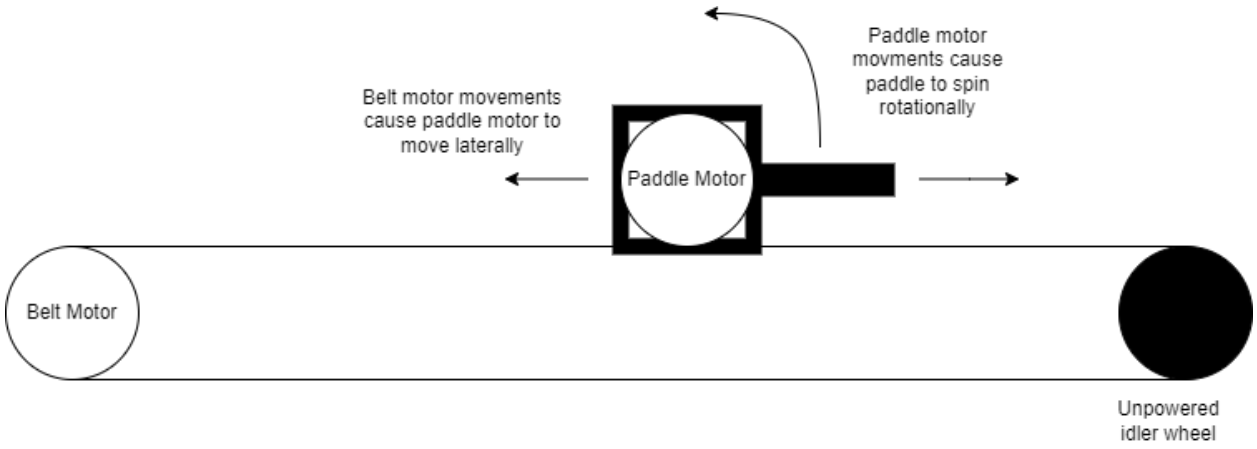
D. Robotics Implementation

The skeleton of our robotics system was already constructed when we arrived on this project. An external power supply feeds up to four amps of power to the motor controller chips which in turn take their input signals from Jetson Nano 2GB board to drive the motors. Initially, both sides of the table had two motors installed: one to swing the paddle used to strike the puck and one to move the first motor laterally across the table. However, over the course of working on the project we removed the motors on one side of the table to allow for human-AI play. The paddle is attached directly to its motor's drive shaft while the lateral motion motor spins a gear which drives a toothed rubber belt. This belt runs along a channeled beam spanning the breadth of the table with the paddle motor's bracket tied into its length.



The flaws in this system were readily apparent after we began robotics testing. The motor control chips had extremely poor thermal dissipation solutions which often resulted in overheating. The controllers were themselves installed into an unlabeled perfboard structure which came with little to no documentation as to functionality. The motors themselves even failed to respond to direct command pulses to their coils. These factors combined led us to fully replace the motors and control system with new motors, ones clearly labeled and with robust cooling solutions.

The new system runs both motors at a 2.5A power draw with a step division of 1600 steps per revolution. With these settings, the paddle can spin at a maximum speed of over 900 revolutions per minute, and the lateral motion belt can move the paddle assembly across the table in less than two seconds. However, another fault of the robotics assembly quickly became readily apparent. The motors themselves were not the limiting factor to the belt's movement speed, but rather the belt itself was. Due to either the materials involved or the age of the belt, under any substantial strain the belt would begin to stretch or slip. This caused the motor to "judder" as it crossed the table. Due to this, we were forced to slow the motor's speed to ensure stable movement as best we could.



Had we been able to design a new system from scratch, we would have instead had the motor disconnected from the belt by a gear system. This would have reduced the amount of friction applied directly to the motor shaft allowing it to move smoothly at higher speeds. We would also have added a spring-set tensioning wheel to the drive belt ensuring that any slack that developed was removed and further reducing the strain on the motor gear. Additionally, the toothed rubber belt could have been swapped for a nylon cord or a similar material resistant to stretching and that also had reduced friction compared to the rubber belt.

Another improvement would have been to add a timer chip into the hardware. This chip could have generated the motor pulses for the controllers. This solution would then have had the software controlling the table use the direction and enable pins to direct the motors around the table. In all, this would have been the better solution as it would have reduced both the processing overhead on the CPU as well as power draw and therefore heat generation on the

motors themselves. Had we been considering those factors early in the project, we would have likely gone with this solution.

D. Software Optimization

The essential task in increasing the speed of the system's reaction time was optimizing the system's software to utilize NVIDIA's CUDA parallel computing architecture. At first, the motor control and air hockey simulation code was revamped to use PyTorch, an alternative to OpenCV. The advantage to working with PyTorch instead of OpenCV was that PyTorch enables CUDA acceleration by default. However, this restructuring of the code became much too time- and labor-intensive and was ultimately abandoned in the end.

Next, because the version of OpenCV precompiled for the Jetson Nano does not have CUDA support enabled for Python 3.9, the version used for the project, an effort was made to compile OpenCV natively on the Jetson Nano itself. However, due to unfamiliarity with the CMake software and the environment variables used to package OpenCV, a successful build was unable to be achieved as the system would continually run out of RAM during compilation. Due to each compilation lasting 3 hours at a minimum before failing, this process quickly became a large time sink. Thus, cross-compilation was attempted next. While a successful compilation was achieved on a laptop with 8GB of RAM, installation of the compiled CUDA-enabled OpenCV package was never successful on the Nano itself.

After further researching CMake's compilation options, multiple attempts were made at compiling OpenCV natively once more. Eventually, the proper build options were determined and OpenCV was installed with CUDA support enabled. In doing so, the system's image processing speed increased from 1FPS to 4FPS. However, even considering this fourfold

improvement, a more capable device than the Nano 2GB would be needed to support near-instantaneous play against a human opponent.

III. Project Management Information

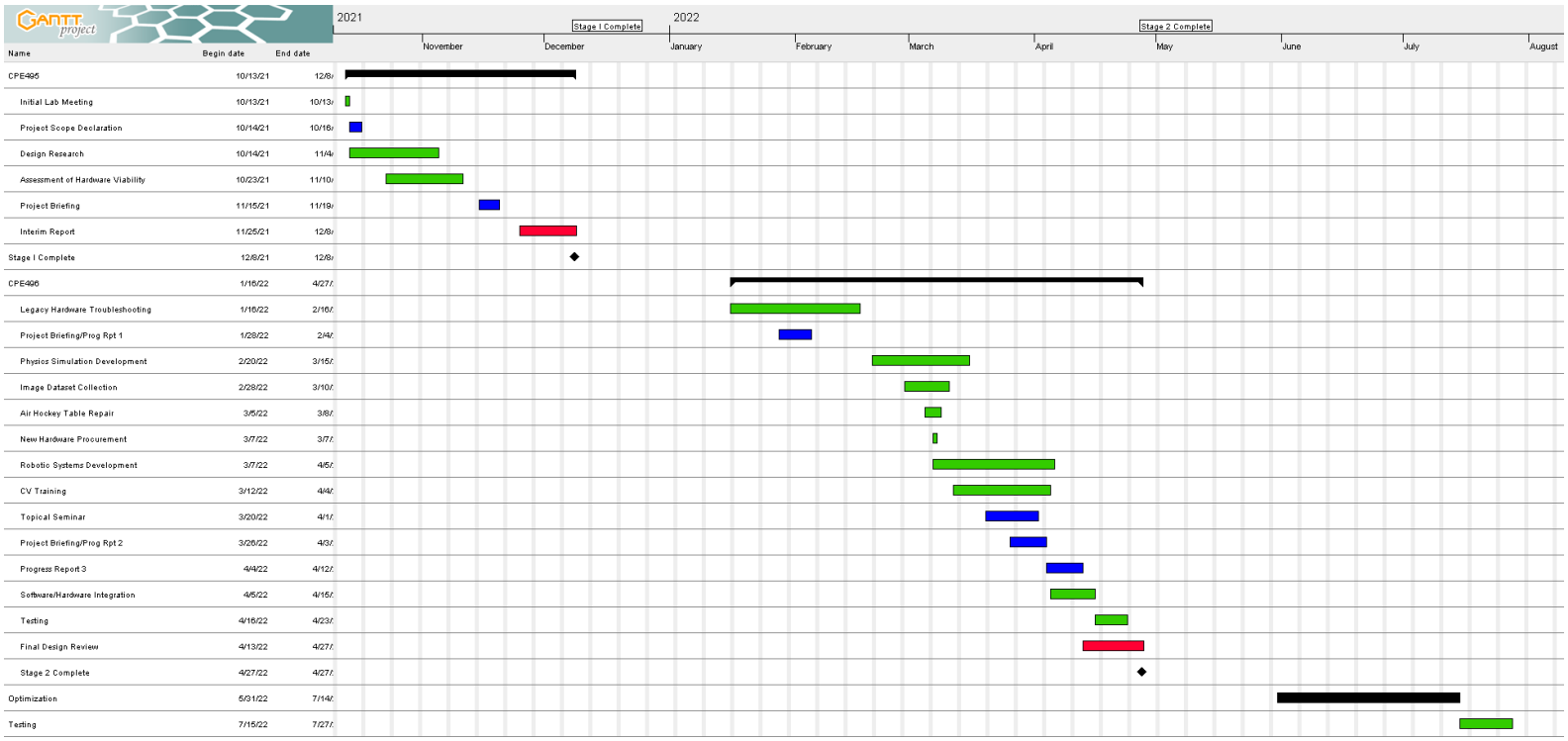
The project was managed in a mostly waterfall style. A standard methodology of developing, testing, re-developing, and so-on was employed. This presented difficulties as issues revealed themselves during testing. Our project milestone definition was broken down into several different stages, all of which carried significance.

A. Costs

Component	Units	Cost Per Part
Stepper Motor	2	$78
64GB MicroSD Card	1	$8
USB Cable	1	$15
USB Extension Cable	1	$15
	Total	$116

Regarding costs, the costliest items were the replacement stepper motors costing a combined 78 dollars. Otherwise, costs were low as much project hardware was inherited from years past. The only other costs were a 64GB MicroSD Card, a USB Cable, and a USB Extension Cable all costing 38 dollars in sum, which brought the total cost of the project to 116 dollars.

B. Project Management Timeline



As far as our project schedule went, most of our time in the Fall semester went toward researching the design specifications and documentation left behind by groups who worked on the project in previous years and toward assessing the viability of the hardware left behind by said groups.

In the Spring semester, we spent a great deal of time initially troubleshooting the legacy hardware that we had inherited before deciding that we would need to procure new hardware at least in part. After this decision, we began working on developing our physics simulation which ended up taking a significant portion of our time. During the same time window, we were able to perform image dataset collection, repair a section of the air hockey table as its surface had been left marred and unusable, and procure necessary hardware to continue developing the robotics

portion of the project. Upon acquiring this hardware, robotic systems development began and lasted approximately one month.

During this time, we were also able to train our CV model using the image dataset we had collected. With both the robotics systems and CV model developed, we were then able to move on to the software/hardware integration phase in order to develop a single functional product. Once the table was finally at the functional stage, we were able to begin testing until the Spring semester came to its close.

After the Spring semester, while the table was technically functional and could be played against at slow speeds, there was still room for improvement when it came to the system's latency in reacting to a puck. Thus, during the Summer semester, the objective was to optimize the system with speed of processing in mind. Initially, an attempt was made at revamping the motor control and air hockey simulation code to use the PyTorch framework which has CUDA support enabled by default. However, as the semester progressed it became clear that this restructuring of the code was becoming too time- and labor-intensive. Thus, the next step was to attempt enabling CUDA within OpenCV. While this endeavor was fraught with a multitude of missteps due to trial-and-error, it ultimately provided modest gains, boosting the FPS processing speed during an additional round of testing from approximately 1FPS to 4FPS.

C. Testing

Knowing the limited performance that we would be able to achieve with the Jetson Nano 2GB, our testing was very simple. We simply launched the puck towards the AI defended goal at a slow to moderate pace to give the system ample time to react, recording the results after each trial and making iterative modifications to the system as necessary.

IV. Societal Impact Analysis

To explain the context of our project, our design must be considered in terms of commercial and research purposes. In this regard, widespread adoption of our project would not have serious implications as it is more useful as a proof of concept in terms of what AI is capable of. As such, it should be considered more in the manner of AI adoption for particular use cases at the societal scale.

In this vein, many of the legal and political issues pertaining to global and societal impact are negligible in the scope of this project seeing as the primary social context of this project is its use for promotion of the ECE department and how it may affect the users and bystanders. As such, the main concern is the health and safety of said users and bystanders. If marketed as an economically viable commercial product, precautions would be discussed to ensure that bystanders will be safe should projectiles from the board fly off unexpectedly. These precautions, e.g. limiting the power of the motors, would be regarded much in the same manner as if the product were being sold like any other air hockey table to play standard air hockey with a regular human player. Any precautions for what would describe improper use would be standard safety measures companies recite regarding safety and use such as not swallowing small items and not lifting heavy objects by oneself.

Many of the global and societal impacts of this project rely on the ethicality of building a fully functioning learning AI. This is cause for concern for many due to the possibility of the AI becoming as intelligent as humans or even more so. As far as environmental effects go, this project at the commercial scale would bring with it the same or similar concerns that other consumer entertainment and electronics products elicit. The same goes for issues regarding privacy and security concerns especially seeing as the product would not be network connected.

Regarding manufacturability, this product in its current state is not practical to manufacture as a pre-built product in its entirety much like a standard air hockey table. Instead, it would best be left in pieces to be assembled. In terms of software maintainability, the code should be easily modifiable as it is accessible via the NVIDIA Jetson Nano 2GB. In terms of robustness, the table itself is fairly durable, and the overhead support structure for the additional machinery and webcam is made of sturdy metal. Items of concern are the sensitive pieces of hardware such as the camera, NVIDIA Jetson Nano 2GB, the motors, the paddles, and the surface of the air hockey table as wear and tear play a factor over the lifetime of the product.

V. Conclusion

A. Lessons Learned

We learned many lessons along the way that will certainly be useful in our future engineering endeavors. While we were plagued by the traditional unpredictable problems such as worn-out motors, one of the biggest headaches was compatibility issues between the various software and hardware components as we tried to tie multiple components of the project together. OpenCV required a different version of Python than what the Jetson uses by default, so we attempted to remove the current version of Python and install a newer version. Little did we know, Python 3.6 is critical to the Jetson OS so we unintentionally broke the board and needed to reflash it numerous times. We had many setbacks in this manner. Doing proper research in terms of software compatibility would have saved us countless hours that could have been spent polishing our project as we had intended. If we were to do this project again, we would certainly not use the Nvidia Jetson Nano 2GB development board.

B. Results

Overall, we were able to successfully implement all but our final functional requirement. Ultimately our system was able to track the motion of the puck and move the motor in the correct direction to intercept the puck. After the initial round of testing, the OpenCV code in our program only captured information at 1FPS. This means that the system had an excruciatingly slow reaction time. During the Spring testing phase, the system would move to intercept with a delay after the puck had passed the goal line the AI was defending. Fortunately, after optimization and an additional round of testing during the Summer semester, the OpenCV code was able to utilize the onboard GPU's parallelization capabilities which allowed for a modest improvement in the system's reaction time from 1FPS to 4FPS. However, even with a fourfold improvement in processing speed, it was clear that a faster device would be necessary to drive AI vs. human play in real-time.

VI. APPENDIX

A. *YOLOTEST.py*

```
import cv2
import numpy as np
import time
import sys
import threading
from math import exp
import asyncio
import Jetson.GPIO as GPIO

motorY=500
motorTarg=500


def dot(vector_a, vector_b):
    product = vector_a[0] * vector_b[0];
    product = product + vector_a[1] * vector_b[1];
    return product

def motor_control():
    # motor 1
    GPIO.setmode(GPIO.BOARD)
    GPIO.setup(36, GPIO.OUT)  # pulse
    GPIO.setup(13, GPIO.OUT)  # dir
    GPIO.setup(40, GPIO.OUT)  # enable

    # motor 2
    GPIO.setup(29, GPIO.OUT)  # pulse
    GPIO.setup(35, GPIO.OUT)  # dir
    GPIO.setup(26, GPIO.OUT)  # enable
    global motorY
    global motorTarg

    while(True):
        print("DEBUG==")
        print(motorY)
        print(motorTarg)
        GPIO.output(36, 0)  # reset pulse pins to 0
        GPIO.output(29, 0)  # reset pulse pins to 0
        time.sleep(0.0001)  # let motor move
        if(motorY < motorTarg): # motor higher than target
            GPIO.output(13, 0)  # set arm motor to direction 0
            GPIO.output(36, 1)  # pulse arm motor
        elif(motorY > motorTarg):   # motor lower than target
            GPIO.output(13, 1)  # set arm motor to direction 1
            GPIO.output(36, 1)  # pulse arm motor
        elif(abs(motorY-motorTarg) < 10):
            pass
        GPIO.output(29, 1) # spin paddle motor
        time.sleep(0.0001)
```

```
if __name__ == "__main__":
    x=threading.Thread(target=motor_control,args=(),daemon=True)
    x.start()

    ##set up video capturing
    cam = cv2.VideoCapture(0)
    cam.set(cv2.CAP_PROP_FRAME_WIDTH, 1280)
    cam.set(cv2.CAP_PROP_FRAME_HEIGHT, 720)
    print(cam.isOpened())
    cv2.namedWindow("test")

    K = .01
    MASS = .025
    FREQ = 100  # Simulation ticks per second
    CAMERAFPS = 30

    dt = (1 / FREQ)
    topLeft = []
    topRight = []
    bottomLeft = []
    bottomRight = []
    motorYPos = []


    def calcBoundary(point1, point2, x):
        if point2[0] - point1[0] == 0:
            return 1
        return (point2[1] - point1[1]) / (point2[0] - point1[0]) * (x -
point1[0]) + point1[1]


    ##files from darknet
    LABELS_FILE = '/home/crah/PycharmProjects/default/data/obj.names'
    CONFIG_FILE = '/home/crah/PycharmProjects/default/data/yolov4-tiny-
airhockey.cfg'
    WEIGHTS_FILE = '/home/crah/PycharmProjects/default/data/yolov4-tiny-
airhockey_8000.weights'
    CONFIDENCE_THRESHOLD = 0.01
    LABELS = open(LABELS_FILE).read().strip().split("\n")

    ##boxes for labels
    np.random.seed(4)
    COLORS = np.random.randint(0, 255, size=(len(LABELS), 3),
                               dtype="uint8")

    ##load net
    net = cv2.dnn.readNetFromDarknet(CONFIG_FILE, WEIGHTS_FILE)

    # determine only the *output* layer names that we need from YOLO
    ln = net.getLayerNames()
    ln = [ln[i - 1] for i in net.getUnconnectedOutLayers()]

    ##loop for detection
    lastX = -1
    lastY = -1
    while True:
```

```
        ret, image = cam.read()
        print("frame")
        if not ret:
            print("failed to grab frame")
            break

        (H, W) = image.shape[:2]

        blob = cv2.dnn.blobFromImage(image, 1 / 255.0, (416, 416),
                                     swapRB=True, crop=False)
        net.setInput(blob)
        layerOutputs = net.forward(ln)

        # initialize our lists of detected bounding boxes, confidences, and
        # class IDs, respectively
        boxes = []
        confidences = []
        classIDs = []

        # loop over each of the layer outputs
        for output in layerOutputs:
            # loop over each of the detections
            for detection in output:
                # extract the class ID and confidence (i.e., probability) of
                # the current object detection
                scores = detection[5:]
                classID = np.argmax(scores)
                confidence = scores[classID]

                # filter out weak predictions by ensuring the detected
                # probability is greater than the minimum probability
                if confidence > CONFIDENCE_THRESHOLD:
                    # scale the bounding box coordinates back relative to the
                    # size of the image, keeping in mind that YOLO actually
                    # returns the center (x, y)-coordinates of the bounding
                    # box followed by the boxes' width and height
                    box = detection[0:4] * np.array([W, H, W, H])
                    (centerX, centerY, width, height) = box.astype("int")

                    # use the center (x, y)-coordinates to derive the top and
                    # and left corner of the bounding box
                    x = int(centerX - (width / 2))
                    y = int(centerY - (height / 2))

                    # update our list of bounding box coordinates,
confidences,
                    # and class IDs
                    boxes.append([x, y, int(width), int(height)])
                    confidences.append(float(confidence))
                    classIDs.append(classID)

        # apply non-maxima suppression to suppress weak, overlapping bounding
        # boxes
        idxs = cv2.dnn.NMSBoxes(boxes, confidences, CONFIDENCE_THRESHOLD,
                                CONFIDENCE_THRESHOLD)

        # ensure at least one detection exists
```

```
        if len(idxs) > 0:
            # loop over the indexes we are keeping
            for i in idxs.flatten():
                # extract the bounding box coordinates
                (x, y) = (boxes[i][0], boxes[i][1])
                (w, h) = (boxes[i][2], boxes[i][3])
                globalx = x
                # print(classIDs[i])
                # print(x)
                if len(bottomLeft) == 0:
                    bottomLeft = [x, y]
                    bottomRight = [x, y]
                    topLeft = [x, y]
                    topRight = [x, y]
                if classIDs[i] == 0:
                    print("Detected Puck")
                    if x < 500:
                        if y > bottomLeft[1]:
                            bottomLeft = [x, y]
                            # print("bottom left =")
                            # print(bottomLeft)
                        elif y < topLeft[1]:
                            topLeft = [x, y]
                    else:
                        if y > bottomRight[1]:
                            bottomRight = [x, y]
                        elif y < topRight[1]:
                            topRight = [x, y]
                    if lastX >= 0:
                        puckVelocity = [(x - lastX) * CAMERAFPS, (y - lastY)
* CAMERAFPS]
                        puckPosition = [x, y]
                        yUpperBoundary = 40
                        yLowerBoundary = 650
                        motorXPosition = 70
                        collisionPlane = [0, 0]
                        ticksToPaddle = 0
                        while puckPosition[0] > motorXPosition and
ticksToPaddle < 1000 and puckVelocity[0] < -1:
                            ticksToPaddle += 1
                            puckVelocity[0] = puckVelocity[0] * exp(-K * dt /
MASS)
                            puckVelocity[1] = puckVelocity[1] * exp(-K * dt /
MASS)
                            lastPosition = puckPosition
                            puckPosition[0] = puckPosition[0] +
puckVelocity[0] * dt
                            puckPosition[1] = puckPosition[1] +
puckVelocity[1] * dt
                            # print(puckVelocity[0])
                            # print(puckPosition[0])
                            if (puckPosition[1] > calcBoundary(bottomLeft,
bottomRight, puckPosition[0]) or
                                    puckPosition[1] < calcBoundary(topLeft,
topRight, puckPosition[0])):
                                puckVelocity[1] = -1 * puckVelocity[1]
                                # puckPosition[1]=yLowerBoundary
```

```
                            puckPositionTest = [int(puckPosition[0]),
int(puckPosition[1])]

                            """color = [int(c) for c in COLORS[classIDs[i]]]

                            cv2.rectangle(image, (puckPositionTest[0],
puckPositionTest[1]), (puckPositionTest[0] + w, puckPositionTest[1] + h),
color, 2)
                            text = "{}: {:.4f}".format(LABELS[classIDs[i]],
confidences[i])
                            cv2.putText(image, text, (x, y - 5),
cv2.FONT_HERSHEY_SIMPLEX,
                                        0.5, color, 2)
                            #time.sleep(dt)"""
                        if puckPosition[0] < motorXPosition:
                            # rint("estimated Position=")
                            # print(puckPosition)
                            if len(motorYPos) > 7:
                                motorYPos.pop(0)
                            motorYPos.append(puckPosition[1])
                        if x < motorXPosition:
                            # print("Actual Position=")
                            # print([x,y])
                            pass
                    if lastX != x:
                        lastX = x
                        lastY = y
                    color = [255, 255, 255]
                    if len(motorYPos) > 0:
                        estimPos = 0
                        for guess in motorYPos:
                            estimPos += guess
                        estimPos = int(estimPos / len(motorYPos))
                        motorTarg=estimPos
                        print(estimPos)
                        cv2.rectangle(image, (motorXPosition, estimPos),
(motorXPosition + w, estimPos + h), color, 2)
                        text = "{}: {:.4f}".format(LABELS[classIDs[i]],
confidences[i])
                        cv2.putText(image, text, (x, y - 5),
cv2.FONT_HERSHEY_SIMPLEX,
                                    0.5, color, 2)
                elif classIDs[i] == 1:
                        print("detected motor")
                        motorY=y

                color = [0,0,255]

                cv2.rectangle(image, (x, y), (x + w, y + h), color, 2)
                text = "{}: {:.4f}".format(LABELS[classIDs[i]],
confidences[i])
                cv2.putText(image, text, (x, y - 5),
cv2.FONT_HERSHEY_SIMPLEX,
                            0.5, color, 2)
```

```
    # show the output image
    cv2.imshow('img', image)

    k = cv2.waitKey(1)
    # time.sleep(.1)

cam.release()

cv2.destroyAllWindows()
```

B. YOLOTESTv2.py

```
import cv2
import numpy as np
import time
import sys
from math import exp

##set up video capturing
MP_FILE='001.mp4'
cam = cv2.VideoCapture(MP_FILE)
print(cam.isOpened())
cv2.namedWindow("test")



K=.01
MASS=.025
FREQ=100 #Simulation ticks per second
CAMERAFPS=30

def dot(vector_a,vector_b):
    product = vector_a[0] * vector_b[0];
    product = product + vector_a[1] * vector_b[1];
    return product

dt=(1/FREQ)
topLeft=[]
topRight=[]
bottomLeft=[]
bottomRight=[]
motorYPos=[]

def calcBoundary(point1,point2,x):
    if point2[0]-point1[0]==0:
        return 1
    return (point2[1]-point1[1])/(point2[0]-point1[0])*(x-
point1[0])+point1[1]

##files from darknet
LABELS_FILE='data\obj.names'
CONFIG_FILE='data\yolov4-tiny-airhockey.cfg'
WEIGHTS_FILE='data\yolov4-tiny-airhockey_last.weights'
CONFIDENCE_THRESHOLD=0.3
LABELS = open(LABELS_FILE).read().strip().split("\n")

##boxes for labels
np.random.seed(4)
COLORS = np.random.randint(0, 255, size=(len(LABELS), 3),
      dtype="uint8")

##load net
net = cv2.dnn.readNetFromDarknet(CONFIG_FILE, WEIGHTS_FILE)

# determine only the *output* layer names that we need from YOLO
ln = net.getLayerNames()
```

```
ln = [ln[i - 1] for i in net.getUnconnectedOutLayers()]

##loop for detection
lastX=-1
lastY=-1
while True:
    ret, image = cam.read()
    if not ret:
        print("failed to grab frame")
        break

    (H, W) = image.shape[:2]

    blob = cv2.dnn.blobFromImage(image, 1 / 255.0, (416, 416),
                                 swapRB=True, crop=False)
    net.setInput(blob)
    layerOutputs = net.forward(ln)

    # initialize our lists of detected bounding boxes, confidences, and
    # class IDs, respectively
    boxes = []
    confidences = []
    classIDs = []

    # loop over each of the layer outputs
    for output in layerOutputs:
        # loop over each of the detections
        for detection in output:
            # extract the class ID and confidence (i.e., probability) of
            # the current object detection
            scores = detection[5:]
            classID = np.argmax(scores)
            confidence = scores[classID]

            # filter out weak predictions by ensuring the detected
            # probability is greater than the minimum probability
            if confidence > CONFIDENCE_THRESHOLD:
                # scale the bounding box coordinates back relative to the
                # size of the image, keeping in mind that YOLO actually
                # returns the center (x, y)-coordinates of the bounding
                # box followed by the boxes' width and height
                box = detection[0:4] * np.array([W, H, W, H])
                (centerX, centerY, width, height) = box.astype("int")

                # use the center (x, y)-coordinates to derive the top and
                # and left corner of the bounding box
                x = int(centerX - (width / 2))
                y = int(centerY - (height / 2))
                # update our list of bounding box coordinates, confidences,
                # and class IDs
                boxes.append([x, y, int(width), int(height)])
                confidences.append(float(confidence))
                classIDs.append(classID)

    # apply non-maxima suppression to suppress weak, overlapping bounding
    # boxes
    idxs = cv2.dnn.NMSBoxes(boxes, confidences, CONFIDENCE_THRESHOLD,
```

```
                            CONFIDENCE_THRESHOLD)

    # ensure at least one detection exists
    if len(idxs) > 0:
        # loop over the indexes we are keeping
        for i in idxs.flatten():
            # extract the bounding box coordinates
            (x, y) = (boxes[i][0], boxes[i][1])
            (w, h) = (boxes[i][2], boxes[i][3])
            #print(classIDs[i])
            #print(x)
            if len(bottomLeft)==0:
                bottomLeft=[x,y]
                bottomRight=[x,y]
                topLeft=[x,y]
                topRight=[x,y]
            if classIDs[i]==0:
                if x<500:
                    if y>bottomLeft[1]:
                        bottomLeft=[x,y]
                        #print("bottom left =")
                        #print(bottomLeft)
                    elif y<topLeft[1]:
                        topLeft=[x,y]
                else:
                    if y>bottomRight[1]:
                        bottomRight=[x,y]
                    elif y<topRight[1]:
                        topRight=[x,y]
                if lastX>=0:
                    puckVelocity=[(x-lastX)*CAMERAFPS,(y-lastY)*CAMERAFPS]
                    puckPosition=[x,y]
                    yUpperBoundary=40
                    yLowerBoundary=650
                    motorXPosition=70
                    collisionPlane=[0,0]
                    ticksToPaddle=0
                    while puckPosition[0]>motorXPosition and
ticksToPaddle<1000 and puckVelocity[0]<-1:
                        ticksToPaddle+=1
                        puckVelocity[0] = puckVelocity[0] * exp(-K * dt /
MASS)
                        puckVelocity[1] = puckVelocity[1] * exp(-K * dt /
MASS)
                        lastPosition=puckPosition
                        puckPosition[0] = puckPosition[0] + puckVelocity[0] *
dt
                        puckPosition[1] = puckPosition[1] + puckVelocity[1] *
dt
                        #print(puckVelocity[0])
                        #print(puckPosition[0])
                        if
(puckPosition[1]>calcBoundary(bottomLeft,bottomRight,puckPosition[0]) or
puckPosition[1]<calcBoundary(topLeft,topRight,puckPosition[0])):
                            puckVelocity[1]=-1*puckVelocity[1]
                            #puckPosition[1]=yLowerBoundary
```

```
                        puckPositionTest = [int(puckPosition[0]),
int(puckPosition[1])]
                        """color = [int(c) for c in COLORS[classIDs[i]]]

                        cv2.rectangle(image, (puckPositionTest[0],
puckPositionTest[1]), (puckPositionTest[0] + w, puckPositionTest[1] + h),
color, 2)
                        text = "{}: {:.4f}".format(LABELS[classIDs[i]],
confidences[i])
                        cv2.putText(image, text, (x, y - 5),
cv2.FONT_HERSHEY_SIMPLEX,
                                    0.5, color, 2)
                        #time.sleep(dt)"""
                    if puckPosition[0]<motorXPosition:
                        #rint("estimated Position=")
                        #print(puckPosition)
                        if len(motorYPos)>7:
                            motorYPos.pop(0)
                        motorYPos.append(puckPosition[1])
                    if x<motorXPosition:
                        #print("Actual Position=")
                        #print([x,y])
                        pass
                if lastX != x:
                    lastX=x
                    lastY=y
            color = [255,255,2555]
            if len(motorYPos)>0:
                estimPos=0
                for guess in motorYPos:
                    estimPos+=guess
                estimPos=int(estimPos/len(motorYPos))
                cv2.rectangle(image, (motorXPosition, estimPos),
(motorXPosition + w, estimPos + h), color, 2)
                text = "{}: {:.4f}".format(LABELS[classIDs[i]],
confidences[i])
                cv2.putText(image, text, (x, y - 5),
cv2.FONT_HERSHEY_SIMPLEX,
                            0.5, color, 2)

            color = [0,0,255]

            cv2.rectangle(image, (x, y), (x + w, y + h), color, 2)
            text = "{}: {:.4f}".format(LABELS[classIDs[i]], confidences[i])
            cv2.putText(image, text, (x, y - 5), cv2.FONT_HERSHEY_SIMPLEX,
                        0.5, color, 2)

    # show the output image
    cv2.imshow('img', image)

    k = cv2.waitKey(1)
    #time.sleep(.1)

cam.release()

cv2.destroyAllWindows()
```

C. simulationViewer.py

```
import pygame
import csv
import time

TABLEWIDTH=2.1336
TABLEHEIGHT=1.2192
WIDTH=800
HEIGHT=600
pygame.init()
screen=pygame.display.set_mode((800,600))
green= (0,255,0)
red = (255,0,0)
black = (0,0,0)
waitforfirst=True
with open('airHockey.csv') as csvfile:
   csvreader=csv.reader(csvfile,delimiter=',')
   for row in csvreader:
       pygame.event.get()
       screen.fill(black)
       temp = []
       idx=0
       for item in row:
           if idx%2==1:
               temp.append(HEIGHT * float(row[idx]) / TABLEHEIGHT)
           else:
               temp.append(WIDTH * float(row[idx]) / TABLEWIDTH)
           idx+=1
       x=pygame.draw.circle(screen, red, (temp[0],temp[1]),10,0)
       y=pygame.draw.lines(screen, red, False,
[(temp[2],temp[3]),(temp[4],temp[5])], 3)
       y2 = pygame.draw.circle(screen, green, (temp[2],temp[3]),4,0)
       z=pygame.draw.lines(screen, red, False, [(temp[6], temp[7]), (temp[8],
temp[9])], 3)
       y2 = pygame.draw.circle(screen, green, (temp[6], temp[7]), 4, 0)
       pygame.display.update()
       if waitforfirst:
           time.sleep(2)
           waitforfirst=False
       time.sleep(.002)
   time.sleep(10)
```

D. HockeySim.cpp

```
// HockeySim.cpp : This file contains the 'main' function. Program execution
begins and ends there.
//

#include <iostream>
#include "Table.h"
#include "Display.h"

#include <sys/types.h>     // needed for the _ftime() function
#include <sys/timeb.h>     // contains definition of _timeb struct
#include <time.h>          // needed for other time functions and structures
#include <iostream>

struct _timeb   tStruct;
double          thisTime;
double          outputTime;
double lastTime;
bool            done = false;  // while loop flag


int main()
{
      Table* table = new Table(2.1336, 1.2192);
    Display* display = new Display(table);


    _ftime_s(&tStruct); // Get start time
    thisTime = tStruct.time + (((double)(tStruct.millitm)) / 1000.0); //
Convert to double
    outputTime = thisTime; // Set next 5 second interval time
    int count = 0;
    while (true) {
        double dt = .001;
        table->getPuck()->onTick(dt);
        table->getPaddle1()->onTick(dt);
        table->getPaddle2()->onTick(dt);
        //if(count++>100)
        display->update();
        //display->writeFile();
    }
}
```

E. Table.h

```
#pragma once
class Controller;
#include "Puck.h"
#include "Paddle.h"
#include "Controller.h"
class Table
{
private:
	double width;
	double height;
	Puck* puck;
	Paddle* paddle1;
	Paddle* paddle2;
public:
	Table(double width, double height);
	Puck* getPuck();
	Paddle* getPaddle1();
	Paddle* getPaddle2();
	double getWidth();
	double getHeight();
};
```

F. Table.cpp

```
#include "Table.h"

Table::Table(double width, double height) {
	this->width = width;
	this->height = height;
	paddle1 = new Paddle(.3556);
	paddle2 = new Paddle(1.788);
	puck = new Puck(width, height, paddle1, paddle2);
	//assign controller to paddles
	paddle1->assignController(new Controller(paddle1,puck,true,
.001,width,height));
	paddle2->assignController(new Controller(paddle2, puck, false, .001,
width, height));
}

Puck* Table::getPuck() {
	return puck;
}


Paddle* Table::getPaddle1() {
	return paddle1;
}

Paddle* Table::getPaddle2() {
	return paddle2;
}

double Table::getWidth() {
	return width;
}

double Table::getHeight() {
	return height;
}
```

G. Display.h

```
#pragma once
#include <stdio.h>
#include <iostream>
#include "Paddle.h"
#include "Puck.h"
#include "Table.h"
#include <windows.h>
#include <fstream>

class Display
{
private:
	int xMax = 100; int yMax = 50;
	double width;
	double height;
	Puck* puck;
	Paddle* paddle1;
	Paddle* paddle2;
	std::ofstream myFile;
public:
	Display(Table* table);
	void update();
	void writeFile();
};
```

H. Display.cpp

```
#include "Display.h"
Display::Display(Table* table) {
	width = table->getWidth();
	height = table->getHeight();
	puck = table->getPuck();
	paddle1 = table->getPaddle1();
	paddle2 = table->getPaddle2();

	HANDLE out = GetStdHandle(STD_OUTPUT_HANDLE);

	CONSOLE_CURSOR_INFO     cursorInfo;

	GetConsoleCursorInfo(out, &cursorInfo);
	cursorInfo.bVisible = false; // set the cursor visibility
	SetConsoleCursorInfo(out, &cursorInfo);
	myFile.open("airHockey.csv", std::ofstream::out |
std::ofstream::trunc);
}

void Display::update() {
	system("cls");
	double puckPosition[2],paddle1Position[2],paddle2Position[2];
	puckPosition[0] = puck->getPosition()[0];
	puckPosition[1] = puck->getPosition()[1];
	paddle1Position[0] = paddle1->getPosition()[0];
	paddle1Position[1] = paddle1->getPosition()[1];
	paddle2Position[0] = paddle2->getPosition()[0];
	paddle2Position[1] = paddle2->getPosition()[1];

	int scaledPuckPosition[2], scaledPaddle1Position[2],
scaledPaddle2Position[2];

	scaledPuckPosition[0] = xMax * puckPosition[0] / width;
	scaledPuckPosition[1] = yMax * puckPosition[1] / height;
	scaledPaddle1Position[0] = xMax * paddle1Position[0] / width;
	scaledPaddle1Position[1] = yMax * paddle1Position[1] / height;
	scaledPaddle2Position[0] = xMax * paddle2Position[0] / width;
	scaledPaddle2Position[1] = yMax * paddle2Position[1] / height;

	for (int y = 0; y < yMax; y++) {
		for (int x = 0; x < xMax; x++) {
			if ( scaledPuckPosition[0] == x && scaledPuckPosition[1] ==
y) {
				std::cout << "*";
			}
			else if (scaledPaddle1Position[0] == x &&
scaledPaddle1Position[1] == y || scaledPaddle2Position[0]==x &&
scaledPaddle2Position[1] == y) {
				std::cout << "|";
			}
			else {
				std::cout << " ";
			}
		}
```

```
            std::cout << "\n";
      }

      std::cout << scaledPuckPosition[0];
}

void Display::writeFile(){
      double puckPosition[2], paddle1Position[2], paddle2Position[2],
paddle1End[2], paddle2End[2];
      puckPosition[0] = puck->getPosition()[0];
      puckPosition[1] = puck->getPosition()[1];
      paddle1Position[0] = paddle1->getPosition()[0];
      paddle1Position[1] = paddle1->getPosition()[1];
      paddle2Position[0] = paddle2->getPosition()[0];
      paddle2Position[1] = paddle2->getPosition()[1];
      paddle1End[0] = paddle1->getEndPosition()[0];
      paddle1End[1]= paddle1->getEndPosition()[1];
      paddle2End[0] = paddle2->getEndPosition()[0];
      paddle2End[1] = paddle2->getEndPosition()[1];
      myFile << puckPosition[0] << "," << puckPosition[1] << ",";
      myFile << paddle1Position[0] << "," << paddle1Position[1] << ",";
      myFile << paddle1End[0] << "," << paddle1End[1] << ",";
      myFile << paddle2Position[0] << "," << paddle2Position[1] << ",";
      myFile << paddle2End[0] << "," << paddle2End[1] << "\n";
}
```

I. Puck.h

```
#pragma once
#include <cmath>
#include <vector>
//#include "Table.h"
#include "Paddle.h"

#define K .01
#define MASS .025

class Puck
{
private:
	double velocity[2] = {10,10};
	double position[2];
	double lastPosition[2];
	double tableWidth;
	double tableHeight;
	double radius=.1;
	double dot(double vector_a[2], double vector_b[2]);
	Paddle *paddle1; Paddle *paddle2;
	bool detectPaddleCollision(Paddle* paddle1);
	double magnitude(double pointA[2], double pointB[2]);
	bool collideLast = false;
public:
	void onTick(double dt);
	void collide(int force,double plane[2]);
	Puck(double tableWidth,double tableHeight, Paddle *paddle1, Paddle
*paddle2);
	double* getPosition();
	double* getVelocity();
	void setFields(double* position, double* velocity);
	double getRadius();
};
```

J. Puck.cpp

```
#include "Puck.h"
#include <iostream>
Puck::Puck(double tableWidth, double tableHeight,Paddle *paddle1,Paddle
*paddle2) {
	this->tableWidth = tableWidth;
	this->tableHeight = tableHeight;
	this->paddle1 = paddle1;
	this->paddle2 = paddle2;

	position[0] = tableWidth / 2;
	position[1] = tableHeight / 2;
	lastPosition[0] = position[0];
	lastPosition[1] = position[1];
}

void Puck::onTick(double dt) {
	velocity[0] = velocity[0] * exp(-K * dt / MASS);
	velocity[1] = velocity[1] * exp(-K * dt / MASS); // add air resistance

	position[0] = position[0] + velocity[0] * dt;
	position[1] = position[1] + velocity[1] * dt; //update puck position
based on velocity

	if ((position[1] > 4 * tableHeight / 10 && position[1] < 6 *
tableHeight / 10 && (position[0] - radius < 0 || position[0] + radius >
tableWidth))) {
		position[0] = tableWidth / 2;
		position[1] = tableHeight / 2;
		lastPosition[0] = position[0];
		lastPosition[1] = position[1];
		velocity[0] = 5+rand() % 10;
		velocity[1] = 5+rand() % 10;
		paddle1->reset();
		paddle2->reset();
	}
	//detect collisions with table boundaries
	double plane[2] = { 0,0 };
	if (position[0]-radius < 0) {
		plane[1] = 1;
		collide(0, plane);
		//position[0] = radius;
	}
	else if (position[0]+radius > tableWidth) {
		plane[1] = 1;
		collide(0, plane);
		//position[0] = tableWidth - radius;
	}
	if (position[1] - radius < 0) {
		plane[0] = 1;
		collide(0, plane);
		//position[1] = radius;
	}
	else if (position[1] + radius > tableHeight) {
		plane[0] = 1;
```

```
            collide(0, plane);
            //position[1] = tableHeight-radius;
            //position[0] = lastPosition[0];
            //position[1] = lastPosition[1];
      }
      //detect collision with paddles
      if (!collideLast) {
            if (paddle1 != nullptr && detectPaddleCollision(paddle1)) {
                  collide(1, paddle1->getArm());
                  collideLast = true;
            }
            if (paddle2 != nullptr && detectPaddleCollision(paddle2)) {
                  collide(1, paddle2->getArm());
                  collideLast = true;
            }
      }
      else {
            if (paddle1 != nullptr && !detectPaddleCollision(paddle1) &&
(paddle2 != nullptr && !detectPaddleCollision(paddle2))) {
                  collideLast = false;
            }
      }
      //set last position to current position
      lastPosition[0] = position[0];
      lastPosition[1] = position[1];
}

bool Puck::detectPaddleCollision(Paddle *paddle1) {
      //Detect Collisions with paddles
      double* pointA = paddle1->getPosition();
      double* pointB = paddle1->getEndPosition();
      //find slope
      double m = (pointA[1] - pointB[1]) / (pointA[0] - pointB[0]);
      //Ax+By+C
      //y-pointA[1]  == m(x-pointA[0])
      //y/m - pointA[1]/m == x- pointA[0]
      // - x + y/m - pointA[1]/m + pointA[0]
      double A = -1;
      double B = 1 / m;
      double C = -pointA[1] / m + pointA[0];
      double distance = abs(A * position[0] + B * position[1] + C) /
sqrt(pow(A, 2) + pow(B, 2));

      //find distance to paddle

      return (distance <= radius && magnitude(pointA, position) < paddle1-
>getArmLength() &&
            (paddle1->getArm()[1] > 0 && paddle1->getPosition()[1] -
position[1] < 0 ||         //Should ensure puck is on arm side of paddle
                  paddle1->getArm()[1] < 0 && paddle1->getPosition()[1] -
position[1] > 0)); //Should ensure puck is on arm side of paddle
            //return true;
            //collide(1, paddle1->getArm());

      //}
}
```

```
void Puck::collide(int force, double plane[2]) {
	//calculate reflection velocity
	collideLast = true;
	double unitMultiplier = 1 / sqrt(pow(plane[0], 2) + pow(plane[1], 2));
	double n[2] = {-plane[1]*unitMultiplier,plane[0]*unitMultiplier};
	double u[2];
	u[0] = (dot(velocity, n) / dot(n, n))*n[0];
	u[1] = (dot(velocity, n) / dot(n, n))*n[1];
	double w[2] = { velocity[0] - u[0],velocity[1] - u[1] };
	double zeros[2] = {0,0};
	double mag = magnitude(zeros, velocity);
	if (force > 0 && mag > 25) {
		velocity[0] = w[0] - .9 * u[0];
		velocity[1] = w[1] - .9 * u[1];
	}
	else if (force > 0 && mag > 17) {
		velocity[0] = w[0] - 1 * u[0];
		velocity[1] = w[1] - 1 * u[1];
	}
	else if (force > 0 && mag>10) {
		velocity[0] = w[0] - 2 * u[0];
		velocity[1] = w[1] - 2 * u[1];
	}
	else if (force > 0 && mag > 5) {
		velocity[0] = w[0] - 3 * u[0];
		velocity[1] = w[1] - 3 * u[1];
	}
	else if (force > 0) {
		velocity[0] = w[0] - 4 * u[0];
		velocity[1] = w[1] - 4 * u[1];
	}
	else{
		velocity[0] = w[0] - .5*u[0];
		velocity[1] = w[1] - .5*u[1];
	}
	//maybe fix position if something is bad here.
	position[0] = lastPosition[0];
	position[1] = lastPosition[1];

	//TESTING
	//velocity[0] *= 1.2;
	//velocity[1] *= 1.2;
}

double Puck::dot(double vector_a[2], double vector_b[2]) {
	double product = 0;
	for (int i = 0; i < 2; i++)
		product = product + vector_a[i] * vector_b[i];
	return product;
}

double Puck::magnitude(double pointA[2], double pointB[2]) {
	return sqrt(pow(pointA[0] - pointB[0], 2) + pow(pointA[1] - pointB[1],
2));
}

double* Puck::getPosition() {
```

```
	return position;
}

double* Puck::getVelocity() {
	return velocity;
}

void Puck::setFields(double* position, double* velocity) {
	this->position[0] = position[0];
	this->position[1] = position[1];
	this->velocity[0] = velocity[0];
	this->velocity[1] = velocity[1];
}

double Puck::getRadius() {
	return radius;
}
```

K. Paddle.h

```
#pragma once
class Controller;
#include <cmath>
#include "Controller.h"

class Paddle
{
private:
	double position[2];
	double arm[2]; //Extension from position
	double armLength = .1;  //Length of the paddle
	double armAngle;
	int movementAxis = 0;
	int rotateAxis = 0;
	double rotateRate = 3000; //degrees/s
	double movementRate = 3.5; //m/s
	Controller* controller = nullptr;
public:
	void onTick(double dt);
	Paddle(double xpos);
	double* getPosition();  //returns the rotation point of the arm
	double* getEndPosition(); //returns the end point of the arm
	double getArmLength();  //returns magnitude of arm vector
	double* getArm();
	void setArm(double angle);
	void setMovementAxis(int movementAxis);
	void setRotateAxis(int rotateAxis);
	void assignController(Controller* controller);
	double getMovementRate();
	double getRotateRate();
	double getRotation();
	void reset();
};
```

L. Paddle.cpp

```
#include "Paddle.h"

Paddle::Paddle(double xpos) {
	setArm(0);
	position[0] = xpos;
	position[1] = 1.2192/2;
}

void Paddle::onTick(double dt) {
	if (controller != nullptr) {
		controller->control();
	}
	if (movementAxis != 0) {
		position[1] += movementRate * movementAxis * dt;
	}
	if (rotateAxis != 0) {
		setArm(armAngle + rotateRate * rotateAxis * dt);
	}
}

double* Paddle::getPosition() {
	return position;
}

double* Paddle::getEndPosition() {
	double endPos[2] = {position[0]+arm[0],position[1] + arm[1] };
	return endPos;
}
double Paddle::getArmLength() {
	return armLength;
}

double* Paddle::getArm() {
	return arm;
}

void Paddle::setArm(double angle) {
	armAngle = angle;
	double angleRad = angle * 3.14159265359 / 180;
	arm[0] = armLength * sin(angleRad);
	arm[1] = armLength * cos(angleRad);
}

void Paddle::setMovementAxis(int movementAxis) {
	this->movementAxis = movementAxis;
}

void Paddle::setRotateAxis(int rotateAxis) {
	this->rotateAxis = rotateAxis;
}

void Paddle::assignController(Controller* controller) {
	this->controller = controller;
}
```

```
double Paddle::getMovementRate() {
	return movementRate;
}

double Paddle::getRotateRate() {
	return rotateRate;
}

double Paddle::getRotation() {
	return armAngle;
}

void Paddle::reset() {
	setArm(0);
	position[1] = 1.2192 / 2;
}
```

M. Controller.h

```
#pragma once
class Puck;
class Paddle;
#include "Puck.h"
#include "Paddle.h"

class Controller
{
private:
	Puck* realPuck;
	Paddle* paddle;
	bool left;
	bool calcThisTick=true;
	double desiredPosition=0;
	int swingCountdown = 0;
	double dt;
	double width;
	double height;
public:
	Controller(Paddle* paddle,Puck* puck, bool left,double dt, double
width, double height);
	void control();
};
```

N. Controller.cpp

```
#include "Controller.h"
#include <iostream>
Controller::Controller(Paddle* paddle, Puck* puck, bool left, double dt,
double width, double height) {
	this->paddle = paddle;
	this->realPuck = puck;
	this->left = left;
	this->dt = dt;
	this->width = width;
	this->height = height;
}

void Controller::control() {
	//Set Desired Position
	swingCountdown = 0;
	if (left) {
		if (realPuck->getVelocity()[0] < 0) {
			if (calcThisTick) {
				//calcThisTick = false;
				Puck* fakePuck = new Puck(width, height, nullptr,
nullptr);
				fakePuck->setFields(realPuck->getPosition(),
realPuck->getVelocity());
				while (swingCountdown < 10/dt && fakePuck-
>getPosition()[0] - fakePuck->getRadius() > paddle->getPosition()[0]) {
					fakePuck->onTick(dt);
					swingCountdown++;
				}
				desiredPosition = fakePuck->getPosition()[1] -paddle-
>getArmLength() / 2;
			}
		}
		else {
			calcThisTick = true;
		}
	}
	else {
		if (realPuck->getVelocity()[0] > 0) {
			if (calcThisTick) {
				//calcThisTick = false;
				Puck* fakePuck = new Puck(width, height, nullptr,
nullptr);
				fakePuck->setFields(realPuck->getPosition(),
realPuck->getVelocity());
				while (swingCountdown < 10 / dt && fakePuck-
>getPosition()[0] + fakePuck->getRadius() < paddle->getPosition()[0]) {
					fakePuck->onTick(dt);
					swingCountdown++;
				}
				desiredPosition = fakePuck->getPosition()[1] -
paddle->getArmLength() / 2;
			}
		}
		else {
```

```
                  calcThisTick = true;
            }
      }

      //Set Paddle Movement
      if (paddle->getPosition()[1] < desiredPosition - paddle-
>getMovementRate()*dt) {  //- paddle->getMovementRate()*dt fixes jitter
            paddle->setMovementAxis(1);
      }
      else if (paddle->getPosition()[1] > desiredPosition + paddle-
>getMovementRate() * dt) { //+ paddle->getMovementRate()*dt fixes jitter
            paddle->setMovementAxis(-1);
      }
      else {
            paddle->setMovementAxis(0);
      }

      //Set Paddle Rotation
      if (swingCountdown*dt*paddle->getRotateRate()<70+(rand()%41)-20 &&
swingCountdown * dt * paddle->getRotateRate() > 10) {
            if (left) {
                  paddle->setRotateAxis(1);
            }
            else {
                  paddle->setRotateAxis(-1);
            }
      }
      else {
            //reset
            if (left) {
                  if (paddle->getRotation()>-50) {
                        paddle->setRotateAxis(-1);
                  }
                  else {
                        paddle->setRotateAxis(0);
                  }
            }
            else {
                  if (paddle->getRotation() < 50) {
                        paddle->setRotateAxis(1);
                  }
                  else {
                        paddle->setRotateAxis(0);
                  }
            }
      }
      swingCountdown--;
}
```

O. InstallOpenCV.sh

```
#!/bin/bash
set -e
cd ~
sudo sh -c "echo '/usr/local/cuda/lib64' >> /etc/ld.so.conf.d/nvidia-
tegra.conf"
sudo ldconfig
cd ~/opencv
cd build

# run cmake
cmake -D CMAKE_BUILD_TYPE=RELEASE \
-D CMAKE_INSTALL_PREFIX=/usr \
-D OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib/modules \
-D EIGEN_INCLUDE_PATH=/usr/include/eigen3 \
-D WITH_OPENCL=OFF \
-D WITH_CUDA=ON \
-D CUDA_ARCH_BIN=5.3 \
-D CUDA_ARCH_PTX="" \
-D WITH_CUDNN=ON \
-D WITH_CUBLAS=ON \
-D ENABLE_FAST_MATH=ON \
-D CUDA_FAST_MATH=ON \
-D OPENCV_DNN_CUDA=ON \
-D ENABLE_NEON=ON \
-D WITH_QT=OFF \
-D WITH_OPENMP=ON \
-D BUILD_TIFF=ON \
-D WITH_FFMPEG=ON \
-D WITH_GSTREAMER=ON \
-D WITH_TBB=ON \
-D BUILD_TBB=ON \
-D BUILD_TESTS=OFF \
-D WITH_EIGEN=ON \
-D WITH_V4L=ON \
-D WITH_LIBV4L=ON \
-D OPENCV_ENABLE_NONFREE=ON \
-D INSTALL_C_EXAMPLES=OFF \
-D INSTALL_PYTHON_EXAMPLES=OFF \
-D OPENCV_GENERATE_PKGCONFIG=ON \
-D BUILD_EXAMPLES=OFF \
-D PYTHON3_EXECUTABLE:FILEPATH=/usr/bin/python3.9 \
-D PYTHON3_INCLUDE_DIR:PATH=/usr/include/python3.9 \
-D PYTHON3_INCLUDE_DIR2:PATH= \
-D PYTHON3_LIBRARY:FILEPATH=/usr/lib/aarch64-linux-gnu/libpython3.9.so \
-D PYTHON3_LIBRARY_DEBUG:FILEPATH= \
-D PYTHON3_NUMPY_INCLUDE_DIRS:PATH=/usr/local/lib/python3.9/dist-
packages/numpy/core/include \
-D PYTHON3_PACKAGES_PATH:PATH=/usr/lib/python3.9/dist-packages \
-D
FIND_PACKAGE_MESSAGE_DETAILS_PythonInterp:INTERNAL=[/usr/bin/python3.9][v3.9.
12()] \
-D FIND_PACKAGE_MESSAGE_DETAILS_PythonLibs:INTERNAL=[/usr/lib/aarch64-linux-
gnu/libpython3.9.so][/usr/include/python3.9][v3.9.12(3.9.12)] \
```

```
-D OPENCV_PYTHON_INSTALL_PATH_SETUPVARS:INTERNAL=/usr/lib/python3.9/dist-
packages \
-D PYTHON3LIBS_VERSION_STRING:INTERNAL=3.9.12 \
-D PYTHON3_DEBUG_LIBRARIES:INTERNAL= \
-D PYTHON3_INCLUDE_PATH:INTERNAL=/usr/include/python3.9 \
-D PYTHON3_LIBRARIES:INTERNAL=/usr/lib/aarch64-linux-gnu/libpython3.9.so \
-D PYTHON3_NUMPY_VERSION:INTERNAL=1.19.4 \
-D PYTHON3_VERSION_MAJOR:INTERNAL=3 \
-D PYTHON3_VERSION_MINOR:INTERNAL=9 \
-D PYTHON3_VERSION_STRING:INTERNAL=3.9.12 \
-D __INSTALL_PATH_PYTHON3:INTERNAL=/usr/lib/python3.9/dist-
packages/cv2/python-3.9 ..

# run make
FREE_MEM="$(free -m | awk '/^Swap/ {print $2}')"
# Use "-j 4" only swap space is larger than 5.5GB
if [[ "FREE_MEM" -gt "5500" ]]; then
  NO_JOB=4
else
  echo "Due to limited swap, make only uses 1 core"
  NO_JOB=1
fi
make -j ${NO_JOB}

sudo rm -r /usr/include/opencv4/opencv2
sudo make install
sudo ldconfig
make clean
sudo apt-get update
```