BocceVision Status 04

This report has 3 parts

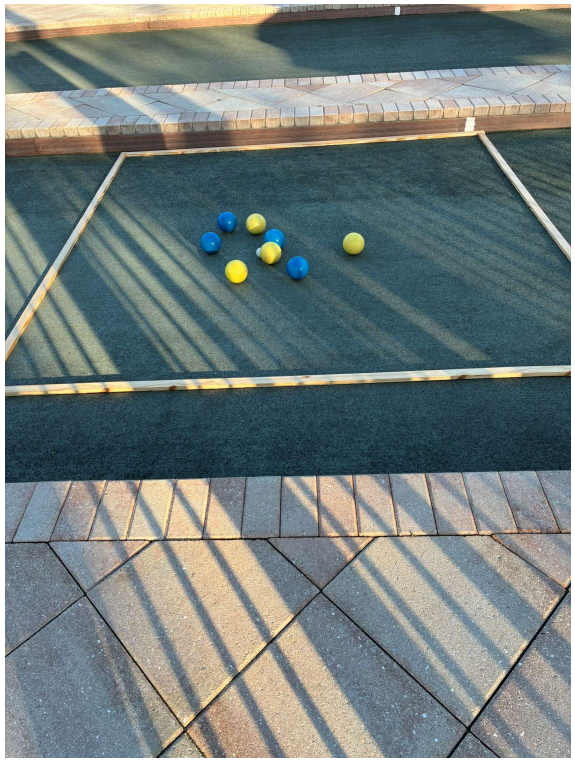
1. Camera Trials at Bocce Court and with Hotspot
2. BocceGame for Vision Logic
3. Simulated BocceVision scoring

Summary

1. Camera Trials at Bocce Court and with Hotspot - unproductive
2. BocceGame for Vision Logic - successful
3. Simulated BocceVision – successful

Future work will look at images either real or simulated of spheres on a plane, to learn what is needed to get a good homographic image for making measurements.

Camera Trials at Bocce Court – and Hotspot Test



I made a test area at the Pelican Sound Bocce Courts. An 8 foot square was used to provide known dimensions.

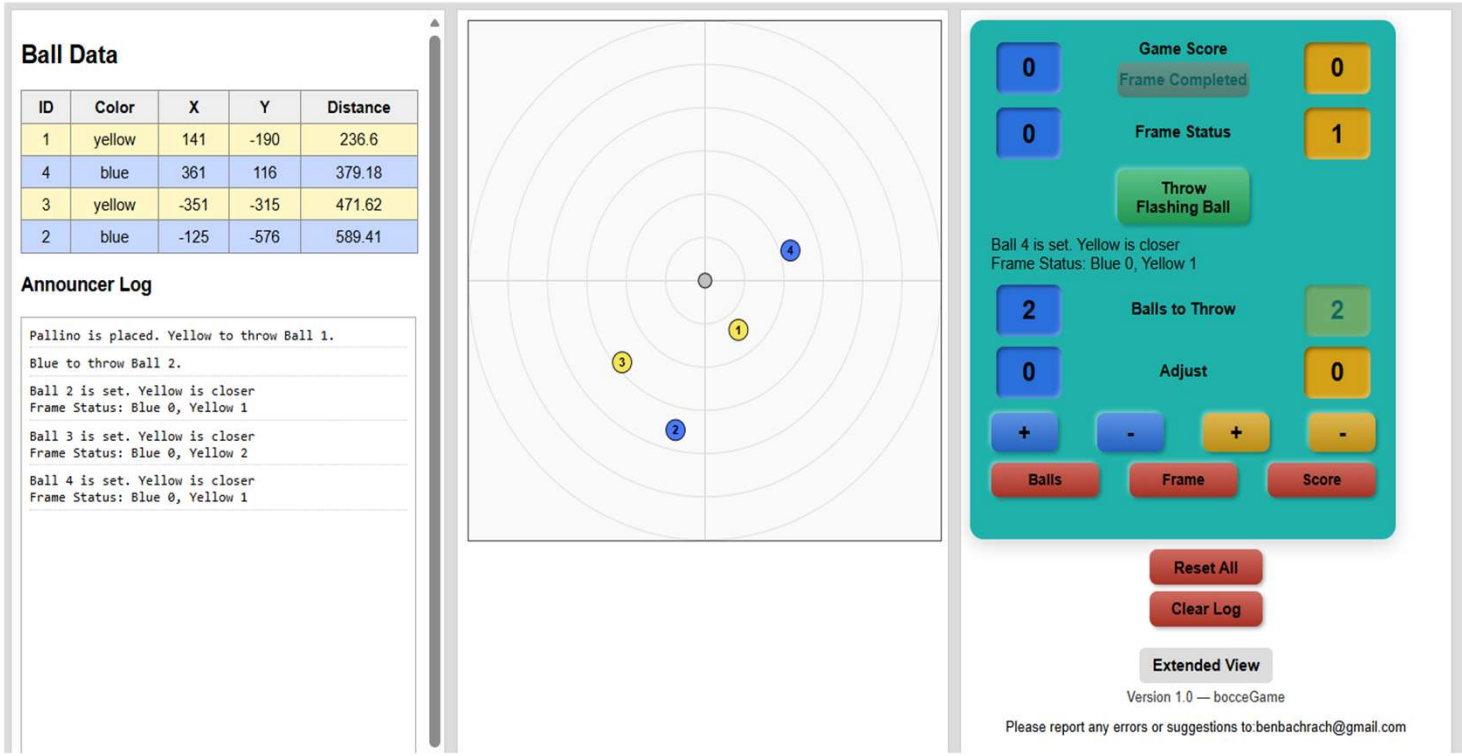
After the setup, with the Braload Mini Camera set on a 12 foot pole, I tried to connect the camera to the Pelican Sound Public Wi-Fi. I had previously connected the camera to that Wi-Fi network, and found it worked. However, the bocce courts were too far from the nearest repeater to get enough signal bars needed for the camera.

I am now inquiring about adding a repeater station close to the bocce courts. The club assistant manager, suggested that a Wi-Fi Hotspot on this phone might be an alternative. Several attempts at getting his Hotspot to connect to the camera failed. Working to get camera images at the bocce courts is now postponed until a method for connecting the camera is found.

BocceGame for Demonstrating Controller and Scoring Logic

While waiting for a good Wi-Fi connection for the mini-camera. I decided to make a computer game using the proposed Referee Controller for BocceVision. The first version of the game was implemented using PHP on my internet site: https://benbachrach.com/bv5/start.php. Running the game confirmed the logic that a BocceVision system would use to identify who throws next, based on the frame status.

The application is available for anyone to use. Users are encouraged to report errors or suggestions to me.



BocceGame: Python Version with Square View of scoring area.

To help learn to use python with Tkinter, a gui extension, Copilot was used to help turn the php version of BocceGame to a python version.
The initial set of code from python was almost correct, but it took more than 15 hours to debug because having Tkinter with a flashing display is difficult.
A second display for the bocce court is shown on the next page

The Balls to Throw flashes to indicate which color throws next. Easy with PHP, difficult with python using Tkinter.

BocceGame for Python Version – extended view

The second view shows a longer court and a ball shelf showing balls available to throw. My experience with Tkinter which Copilot says is the easiest gui for python, suggests that python should be used for analyzing images and supporting scoring but not for generating the courtside TV display.

Simulated BocceVision Scoring - background

While waiting for a good Wi-Fi connection for the mini-camera. I decided to use python to develop a first step toward the image analysis BocceVision may use. Most of this work was done before I worked on BocceGame. Since I have little python coding experience, I used Microsoft Copilot and VS Code with OpenCV to help.

Originally, I used an image taken by the Braload Mini Camera as the test subject. Using OpenCV I was able to make a homographic image replicating a bird's-eye view of a group of bocce balls with the pallino at the center. This process used GIMP to crop the image and then post processed that image using python. It worked but was cumbersome making test cases. As an alternative, I developed a process that used just python to make an artificial image and then processed that image. To simplify the debugging, each step of the process was a stand-alone app. When completed, a master app was made to link each step, step0run_pipeline.py.

The imaginary camera has a field of view 2000 px by 2000 px with the pallino at the center, where each pixel represents 1 mm. The camera location was defined using a coordinate system centered on the pallino with Camera position in mm
(x horizontal, y vertical, z height) = 2050.0, 2000.0, 4000.0. That is 4 meters high, along the centerline of the pallino, and 50 mm outside the field of view. That means the camera was looking down at a 45 deg angle.

The camera has parameters of an iPhone 17:
FOCAL_LENGTH_MM = 4.25
SENSOR_WIDTH_MM = 7.6
SENSOR_HEIGHT_MM = 5.7
Giving a synthetic camera pixel density:
SENSOR_PIXELS_WIDTH = 2000.0
FINAL_SIZE = 2000 # final output resolution

Simulated BocceVision Scoring - procedure

The starting point is to prepare a json file defining the locations of each ball in the view of the imaginary camera. For this report, I will present the results using ballLocation02.json with:


```
{  "yellowBalls": [
    {"id": 1, "x": -900, "y": 900},
    {"id": 2, "x": 900, "y": 900},
    {"id": 3, "x": 900, "y": -900},
    {"id": 4, "x": -900, "y": -900}  ],
  "blueBalls": [
    {"id": 5, "x": -100, "y": 100},
    {"id": 6, "x": 100, "y": 100},
    {"id": 7, "x":  100, "y": -100},
    {"id": 8, "x": -100, "y": -100}   ]
}
```


The balls have a radius of 54 mm, making them 1 mm more than the 107 mm regulation diameter of a bocce ball. The program requires integer ball sizes. Using 53 mm did not make any difference in the results.

Procedure:

To start the application you use the base name of the file to be used. At each step, the app pauses until the user presses enter on the keyboard:

python step0run_pipeLine.py ballLocation02

It calls step1photoFromJson.py and displays: ballLocation02.png.

Pressing enter runs: step2perspective_from_ortho.py giving ballLocation02_perspective.png. This is the simulated image from the camera.

Pressing enter runs: step3ortho_from_perspective.py giving ballLocation02_ortho.png. This is the homographic image derived from the simulated image.

Pressing enter runs: step4analyzeOrtho.py. It measures and reports the distances to the pallino in a json file.

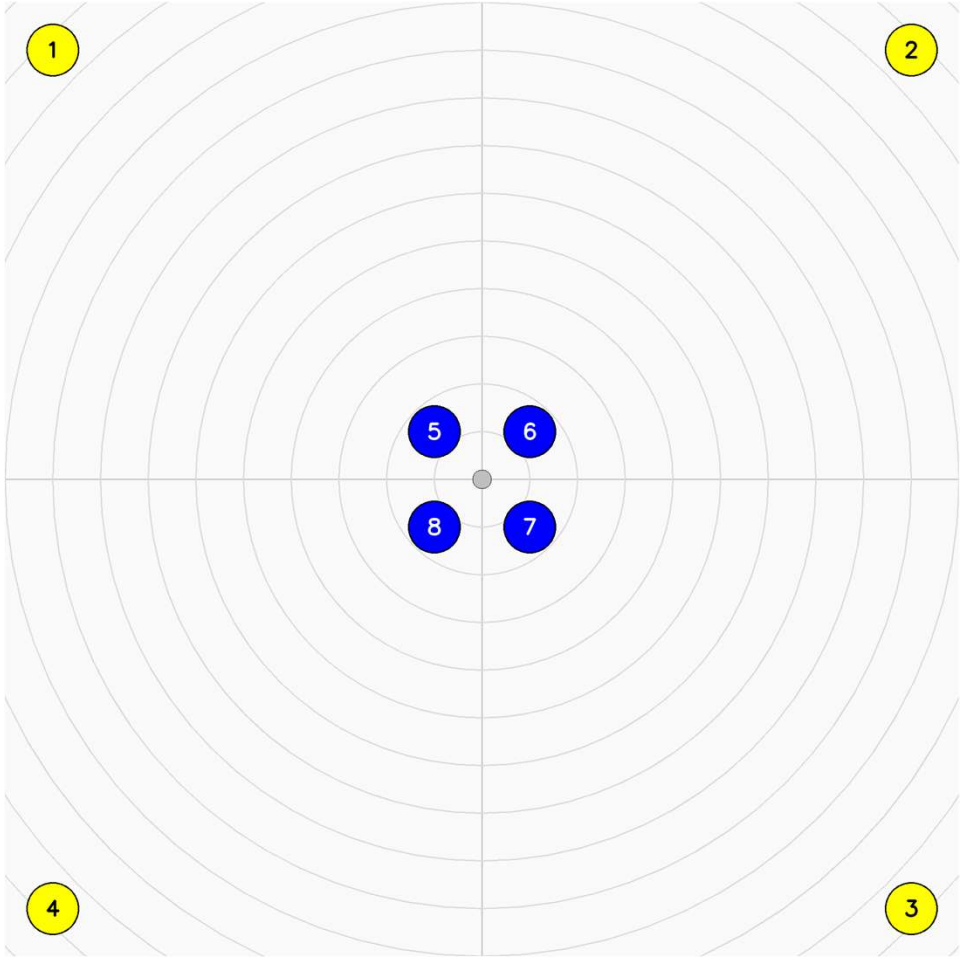
Pressing enter runs: step5reportResults.py which gives a table comparing the input ballLocation02.json with the ballLocation02.measured.

Results from this process are shown on the following pages:

Simulated BocceVision Scoring – results from step 1

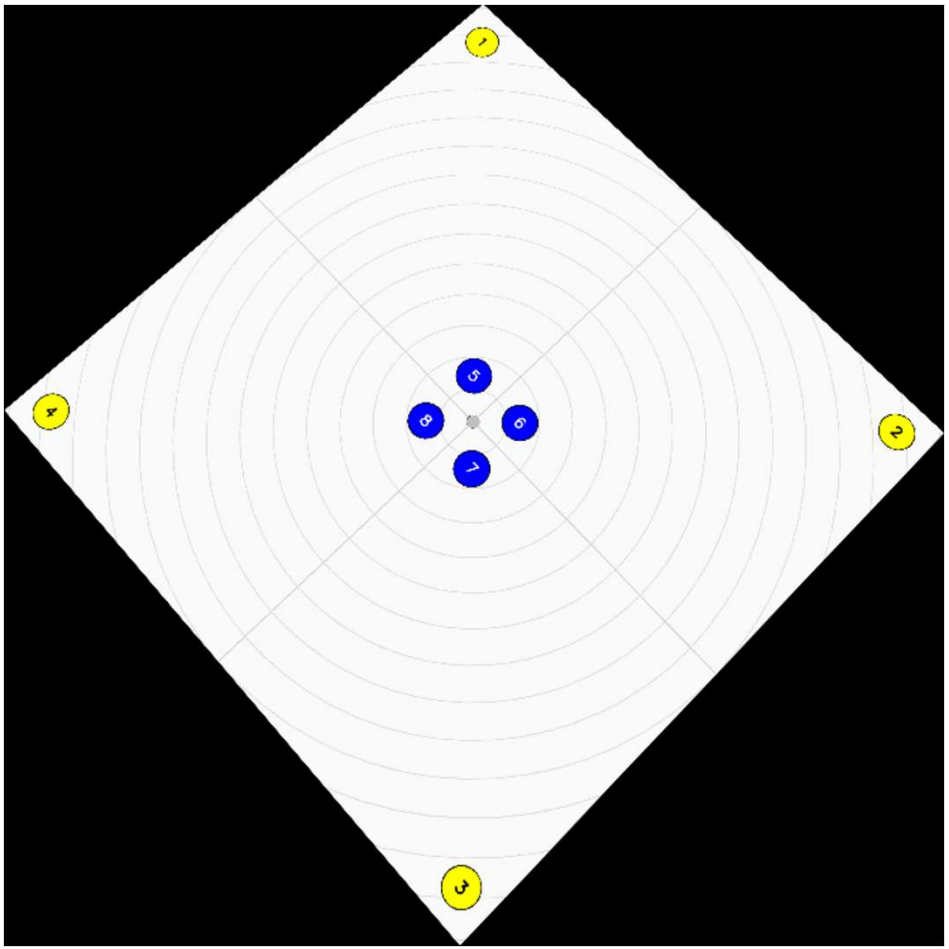
Step1 gives a png file 2000 px x 2000 px
with 4 yellow balls located near the corners of the field of view, 4 blue balls close to the pallino. Cross hairs and concentric circles help visualize the locations of the balls.

ballLocation02.png



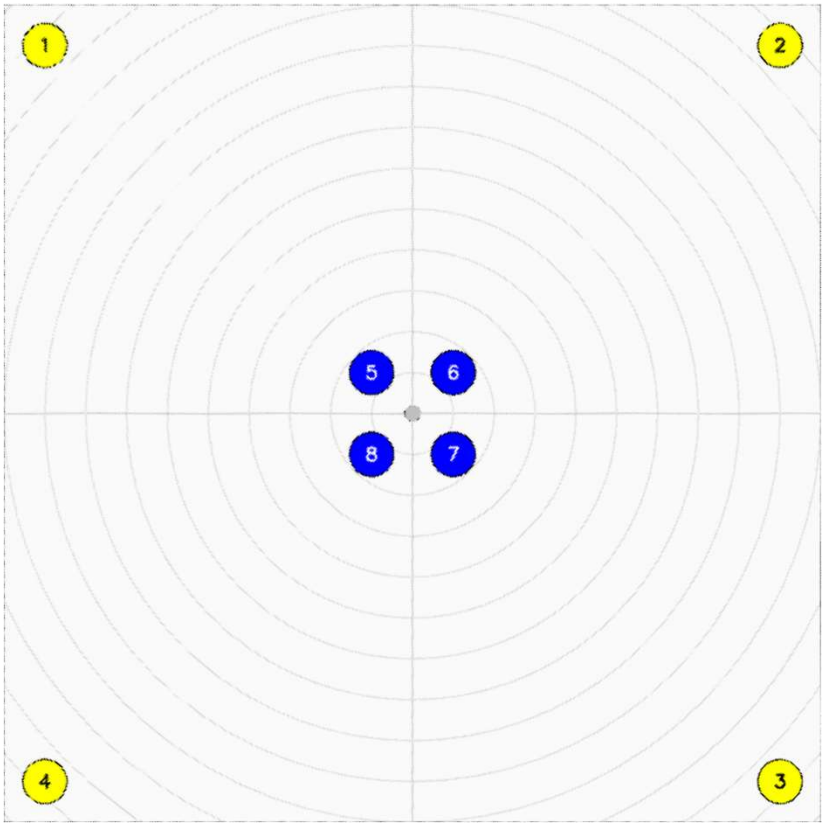
Simulated BocceVision Scoring – results from step 2

Step2 gives a png file 2000 px x 2000 px ballLocation02_perspective.png showing the view from the camera with the camera in the lower right corner area



Simulated BocceVision Scoring – results from step 3

Step3 gives a png file 2000 px x 2000 px ballLocation02_ortho.png showing the view the reconstructed homographic image.



Simulated BocceVision Scoring – results from step 4

Step4 gives ballLocation02_measured.json
giving the dimension calculated by OpenCV from homographic image given by step 3.


```
{
   "yellowBalls": [
      {   "id": 1,  "x": -901.0,  "y": 901.0  },
       {   "id": 2,  "x": 900.0,    "y": 900.0  },
       {    "id": 3,  "x": 899.0,   "y": -899.0  },
       {    "id": 4,   "x": -900.0,  "y": -900.0  }    ],
   "blueBalls": [
        {    "id": 5,  "x": -100.0,   "y": 101.0   },
        {     "id": 6,  "x": 100.0,     "y": 100.0   },
        {      "id": 7,  "x": 100.0,     "y": -100.0  },
        {       "id": 8,  "x": -101.0,    "y": -99.0  } ]
}
```

Simulated BocceVision Scoring – results from step 5

Step5 gives a table showing difference between the dimensions used to make step1 and the dimensions measured in step4.

ID	Color	x0	y0	x1	y1	d0	d1	error
1	yellow	-900.00	900.00	-901.00	901.00	1272.79	1274.21	1.41
2	yellow	900.00	900.00	900.00	900.00	1272.79	1272.79	0.00
3	yellow	900.00	-900.00	899.00	-899.00	1272.79	1271.38	-1.41
4	yellow	-900.00	-900.00	-900.00	-900.00	1272.79	1272.79	0.00
5	blue	-100.00	100.00	-100.00	101.00	141.42	142.13	0.71
6	blue	100.00	100.00	100.00	100.00	141.42	141.42	0.00
7	blue	100.00	-100.00	100.00	-100.00	141.42	141.42	0.00
8	blue	-100.00	-100.00	-101.00	-99.00	141.42	141.43	0.01

The maximum error was found to be 1.41 mm. That is the hypotenuse of a right triangle with sides 1 mm. It is caused by the algorithm rounding to the nearest pixel, where each pixel represented 1 square mm. A higher resolution image or using multiple images of the same scene would be needed to reduce the error further. Future work will look at images either real or simulated of spheres on a plane, to learn what is needed to get a good homographic image for making measurements.