Catch 'em Bot

A robot that catches things, 'cause you gotta catch 'em all!

(PAPER REVISION 0)

An ambitious robot attempt made by this weird guy:

Miles Mulet

Made inside the wonderful environment of:

EEL4665, Intelligent Machine Design Laboratory

Under of the wisdom of awesome instructors:

A. Antonio Arroyo, PhD

Eric M. Schwartz, PhD

With a ton of help in and out of labs by Grade “A” TAs:

Andy Gray

Jake Easterling

Ralph F. Leyva

Table of Contents:

Abstract:

Hey! Catch 'em Bot is what the kids call “neat” because it catches thrown objects! When someone throws a specially colored ball, Catch 'em Bot moves around while always keeping the ball in its sight. It does this by utilizing a super cool wheel system known as “omni wheels” which, as the name suggest, allow for omni-directional movement! Woah! As the ball gets closer and closer, Catch 'em Bot moves in to catch it, acting as if it was predicting where the ball was gonna land! Fun for the whole family!

Wanna make sure the ball goes somewhere where it will be caught in style by Catch 'em Bot? Not a problem! Catch 'em Bot also communicates wirelessly with a *throwing station* that tells Catch 'em Bot about where the ball will end up so Catch 'em Bot is ready to go! Cool! Go teamwork!

Introduction:

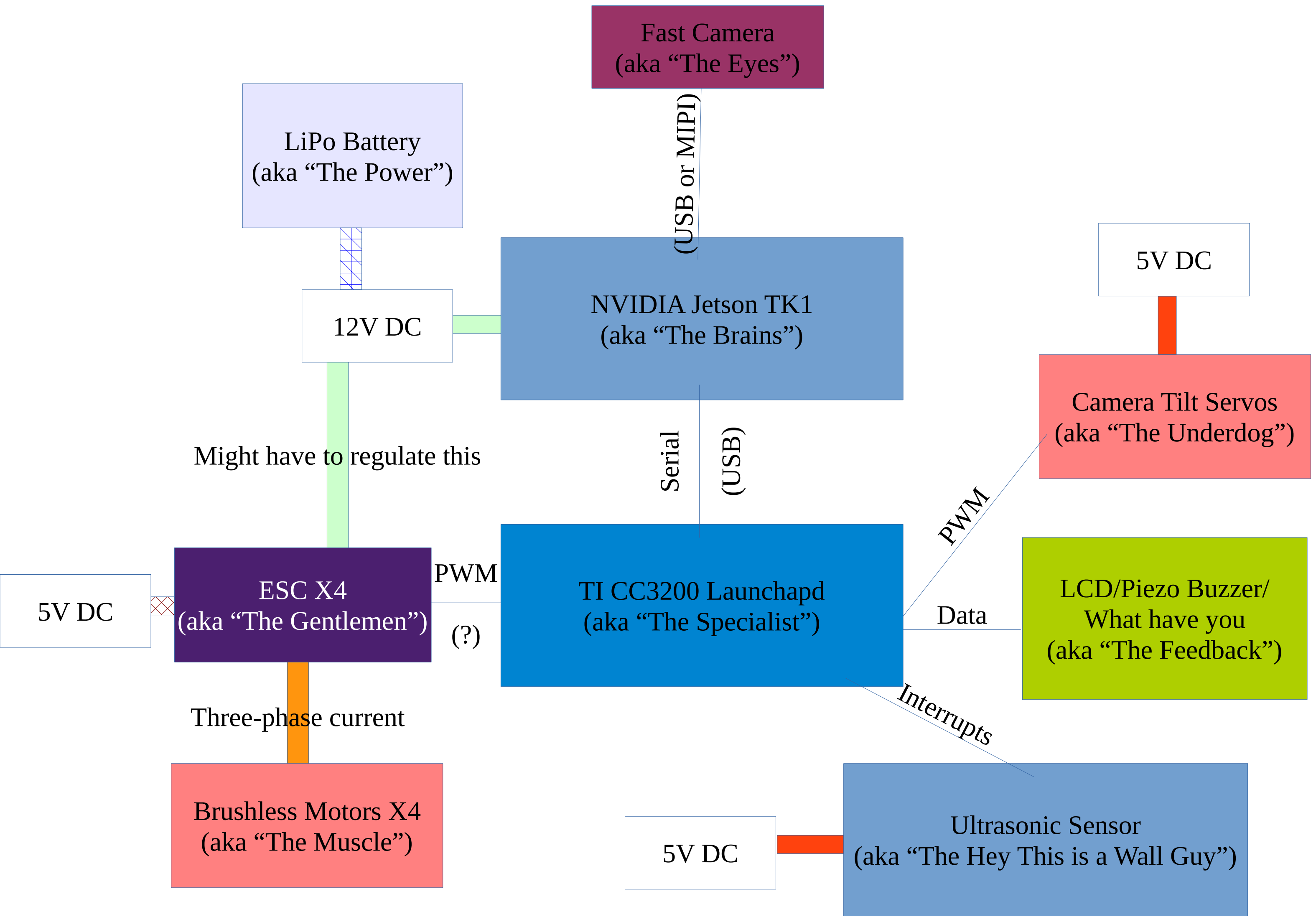
Robots are cool! However - while Hollywood seems to disagree – making robots perform tasks that humans can naturally is a huuuuge challenge. Take, for example, the game of catch played by family members and friends.

People do this easy! What is there to it, right? You just look at where a ball is going and you... uh... just “catch it!”

Well, part of what makes robotics so interesting is discovering that sometimes these simple actions like “catching” things are not at all trivial to implement via hardware and require complex computational algorithms. I aim to (as much as possible) devise an algorithm that allows Catch 'em Bot to approximate what we might see as “expected” animal behavior. In doing so, I hope to learn a little bit more about artificial intelligence, controls, and how these things can be made better so robots of the future can perform more tasks for humans. Plus catch! That's fun!

How Catch 'em Bot is Hooked Up:

Below is a real dandy flow-chart that shows how Catch 'em Bot “thinks”, “sees,” and otherwise makes its way through the world. (Inspiration drawn heavily from *Nicolas Lavelaine*. Thanks!)


Fast Camera (aka “The Eyes”)
(USB or MIPI)
LiPo Battery (aka “The Power”)
12V DC
NVIDIA Jetson TK1 (aka “The Brains”)
5V DC
Camera Tilt Servos (aka “The Underdog”)
Might have to regulate this
Serial
(USB)
PWM
5V DC
ESC X4 (aka “The Gentlemen”)
PWM
(?)
TI CC3200 Launchapd (aka “The Specialist”)
Data
LCD/Piezo Buzzer/ What have you (aka “The Feedback”)
Three-phase current
Interrupts
Brushless Motors X4 (aka “The Muscle”)
5V DC
Ultrasonic Sensor (aka “The Hey This is a Wall Guy”)

The gist of the above graph:

Basically, the *Jetson TK1* runs all the complicated OpenCV on data that it is reading from a *fast camera* (preferably hooked up via MIPI or at least USB 3.0 in order to get lots of speed). The *Jetson*, while awesome and a valuable member of this team, soaks up a whopping 12V; not to fear! Our good 'ol *LiPo Battery* feeds it and the ESCs as well (incidentally, the ESCs supply a “dirty” 5 V output for the less sensitive components)

Once the Jetson has crunched the data, it sends it to a *TI CC3200 Launchpad*, which uses very basic algorithms to assign the corresponding motion to the ESCs. If need be, it also moves the camera (via a servo) or overrides the Jetson's commands because it is getting object detection from the Ultrasound sensor. Regardless of what it is doing, it sends out a signal to the *LCD Screen* (and maybe a *piezo buzzer* or other fun toys) to let the user know what Catch 'em Bot is thinking.

The PWM that gets sent to the *ESCs* is translated by them to the brushless DC motors. This is a part of the project that, admittedly, I'm not too sure about right now; I need the motors to change direction very fast, and I'm not quite sure that the ESCs can translate the instructions from the TI Launchpad fast enough, let alone get the motors to change directions fast enough. (Should this setup fail, I'm thinking of going with Brushed DC motors if they're faster, or even hacked servos. Not ideal by any means, I know, but better than nothing!)

How Catch 'em Bot is set up to move:

The chassis has yet to be built, but I'm thinking of something along the lines of an *octagon design:* this way, should I choose to change wheel types, I will not have to mount the motors on a different location.

The current design for the wheels is to use *omni-directional* wheels (not to be confused with Mecanum wheels):

This allows our fair robot to be able to *strafe*, i.e. move left without rotating. This is really great because we catch the ball by making lots of little adjustments, and the omni wheel design lets us do these without having to come up with some complicated path.

The motors that are used to drive the omni-directional wheels are 4 *brushless DC* motors, exact specifications to be determined. I decided to use brushless DC motors because of their high speeds, but I'm a little skeptical that it will work as intended; as noted above the design may change and the brushless motors might be replaced with brushed ones or even hacked servos.

How Catch 'em Bot senses the World:

Catch 'em Bot will be equipped with a plethora of sensors:

Ultrasonic Sensors

These are placed around the robot in order to implement object avoidance; the microcontroller looks at them first before assigning any other directions to the motors. In this way, Catch 'em Bot prioritizes object avoidance above anything else. We won't be running into walls anytime soon!

The Camera

This is the most important sensor of all! Catch 'em Bot will be using a camera to capture an image several times per second. The speed of the camera is crucial: if the Jetson is unable to receive and process frames fast enough, then the poor little robot will have a very tough time catching anything. The camera image itself is processed on the Jetson TK1 using the *OpenCV* library. Pretty nifty, huh?

How Catch 'em Bot figures out how to catch the ball

As I was examining past work in catching objects and the challenges that it brought up, one of the concerns that frequently came up was that a camera only gives a 2D image – that is, it's “impossible” to predict where the ball will land as we are missing the depth axis.

The idea driving this project is twofold:

1) the above idea is bologna: why don't we just use the scale of the target ball as a way to calculate the distance from the ball to the camera?

2) people are overcomplicating things: we don't need to predict the trajectory of the ball at all; we can algorithmically move the robot in such a way that it ends up where the ball will land without having to waste precious CPU time constructing a 3D model of the ball and the surrounding space.

I'm still in the process of figuring everything out, but the current movement algorithm is rather simple (and only yields good results if the ball has been launched in a small range of acceptable values). All we do is move the robot such that it attempts to keep the ball centered in the camera at all times. We don't even take into account the ball's change in movement (though we do have the capacity to) and we only lightly change the behavior based on the area of the detected ball.

As I said, results in simulation have been... less than ideal, but still workable. I will continue to bash out more workable algorithms as fast as I can.

Conclusion

Of course, there remains much to be done with the project: the exact specifications for all of the items need to be determined – not to mention actually ordering and assembling all of the parts! The reason that it has taken me so long to do this is that I believe in the power of simulation and will only spend the money to order the parts when I am fairly confident in their success.

However, I believe in the project and hope whole-heartedly that I am able to develop something that impresses. “Slow and steady wins the race,” as they say, and I shall very cautiously make my way towards the finish line.

Thanks! I know that this report is pretty much a skeleton right now, but I'm excited to show you how it will flesh out in the weeks to come. I'll be updating this report every week (along with adding the weekly reports that summarize the changes made here), so check back then to see this idea come to life!

Appendix A.

Current Code used to Control Robot in Simulation (written in C#)

```
using UnityEngine;
using System.Collections;

public class CatchBall : MonoBehaviour {

    public Camera openCVCamera;

    public int lastXPos, lastYPos, lastArea = -50;
    public int currentXPos, currentYPos, currentArea = -50;
    public int detected = 0;

    public int shouldStart = 0;

    public int xThreshold = 0; //Used in algorithm 3 to see how close to the "center" of the screen we want the ball
    public int yThreshold = 0; //ditto
    public int areaThreshold = 9000; //Used in algorithm 3 to see when we want to stay put (because the ball is within
our grasp)

    public int currentState = 0;    //For Debugging purposes

    private TestScript openCVData;  //TestScript interfaces with OpenCV to get the X,Y, and Area values
    private RobotLocomotion locomotion; //RobotLocomotion moves the robot in different ways

    private Vector3 ballMovement = Vector3.zero; //Used to track the last frame's ball movement (we keep track of it
so we can compensate how we move in the movement algorithm)

        // Use this for initialization
        void Start () {
        openCVData = openCVCamera.GetComponent<TestScript>();
        locomotion = GetComponent<RobotLocomotion>();
        locomotion.SetMotionActivated(true);
    }

    // Update is called once per frame
    void Update() {
        //Shift in new data, accounting for the shift in position that we just had
        lastXPos = currentXPos; //+ (int) ballMovement.x*10; //<---- I was using this line in Algorithm 3. I don't
know why, though...
        lastYPos = currentYPos;
        lastArea = currentArea; //+ (int) ballMovement.z * 10;

        //-----
        currentXPos = openCVData.XPos;
        currentYPos = openCVData.YPos;
        currentArea = openCVData.Area;

        detected = openCVData.detected;

        //Perform the actual logic:


        //- -   -   -  -    -   -   -   -   -   -   -   -   -   -
        //=====================================-------------------------------------------------------

//========================================================-------------------------------------------------------
        //--==== ITERATION 1:         Just follows the ball (Calculates the ball's move vector and does the same
thing. Ignores balls Y movement)
        //                      Tries to directly convert camera XYA coordinates into Unity's XYZ coordinates
        //                          Using a linear function.
        //
        //           Caveats:
        //                  + Assumes that the robot is always facing forward (i.e., rotation is 0,0,0)
        //                  + Does not currently support any logic to handle the tilting camera.
        //
        //
        //          ++Results++
        //                  - I mean, does its job.
        //                  - As the ball gets very close to the camera, the linear mapping of AREA to Z breaks down
        //                  - Might be a good backup project?

//=========================================================-------------------------------------------------------
        //=====================================--------------------------------------------------------
        //- -   -   -   -   -   -   -   -   -   -   -   -   -   -
        /*
        Vector3 ballMovement = new Vector3(currentXPos - lastXPos, currentYPos - lastYPos, currentArea - lastArea);

        ballMovement.Scale(new Vector3(.01f, 0f, -.0001f));

        transform.localPosition += ballMovement;
        */

        //- -   -   -  -    -   -   -   -   -   -   -   -   -   -
        //=====================================--------------------------------------------------------

//=========================================================-------------------------------------------------------
        //--==== ITERATION 2:         Follows only the X motion of the ball as it approaches. When it starts falling,
it
        //                                  adjusts its Z position to try and Get the Area in the sweet spot
        //                      Tries to directly convert camera XYA coordinates into Unity's XYZ coordinates
        //                          Using a linear function.
        //
        //           Caveats:
        //                  + Assumes that the robot is always facing forward (i.e., rotation is 0,0,0)
        //                  + Does not currently support any logic to handle the tilting camera.
        //                  + Assumes Parabolic Motion of the ball (and that the ball is heading towards the robot)
        //
        //
        //          ++Results++
        //                  - Eh...
        //                  - Having trouble getting it to move correctly in the x direction, likely because I think
the compensation for bot movement is not good enough
        //                  - Suuuuuper inconsistent. Annoyingly so. Not really sure why in simulation; I guess
Unity's physics engine adds in some psuedo randomness
        //                  - Not ready to move on to the next version yet as this sometimes almost gets the right
answer. I'm gonna refine it for a little while and see if I can't make something of it
        //                  - Okay, I tried it and I don't like it. It's great for mid-range catching, but if we start
too close or too far from the ball then everything's a disaster
        //                          (when in the sweet range, handles adjusting its X-Position relatively well, I was
surprised)

//=========================================================-------------------------------------------------------
        //=====================================--------------------------------------------------------
        //- -   -   -   -   -   -   -   -   -   -   -   -   -   -
        //ballMovement = new Vector3(currentXPos - lastXPos, currentYPos - lastYPos, currentArea - lastArea);
        /*
        ballMovement = new Vector3(currentXPos - lastXPos, currentYPos - lastYPos, currentArea - lastArea);

        if (currentYPos > 10 && lastYPos > 10)
            {
                shouldStart = 1;         //We don't want junk data at the beginning
                //Debug.Log("Heeeeeeellloooooo!");
            }
            else
            {
                locomotion.SetDirection(Vector3.zero);
                ballMovement = Vector3.zero;
                //locomotion.SetMotionActivated(false);
                shouldStart = 0;
            }

        if (shouldStart == 1)
        {
            if (ballMovement.y > 0 && currentArea < 15000) //The ball is currently still moving up; it might be best
to only move in the X direction (unless the ball is approaching us fast!)
            {
                ballMovement.Scale(new Vector3(.01f, 0f, 0f));
                locomotion.SetDirection(ballMovement);
            }
            else  //Either the ball has begun its decent or we are getting really close to the ball; either way, we
need to start moving backwards as well as with the x position of the ball
            {
                if (currentArea < 15000) //We need to get closer
                {
                    ballMovement.Scale(new Vector3(.01f, 0f, .001f));
                }
                else //Too close!
                {
                    ballMovement.Scale(new Vector3(.01f, 0f, -.001f));
                }

                locomotion.SetDirection(ballMovement);

            }
        }
        */


        //- -   -   -  -    -   -   -   -   -   -   -   -   -   -
        //=====================================------------------------------------------------------

//==========================================================------------------------------------------------------
        //--==== ITERATION 3:         The idea behind this one is that we need to keep the ball centered in the camera
at all times. It adjusts its X position like normal, but
        //                                  uses both the Y position and Area to move to keep the ball in the center
of the screen and at as large an area as possible, in that priority.
        //                                  That part was important! IT TRIES TO KEEP THE BALL IN THE CENTER OF THE
CAMERA FIRST, AND WORRIES ABOUT HOW BIG THE BALL IS LATER
        //                      Tries to directly convert camera XYA coordinates into Unity's XYZ coordinates
        //                          Using a linear function.
        //
        //           Caveats:
        //                  + Assumes that the robot is always facing forward (i.e., rotation is 0,0,0)
        //                  + Does not currently support any logic to handle the tilting camera.
        //                  + Assumes Parabolic Motion of the ball (and that the ball is heading towards the robot)
        //
        //
        //          ++Results++
        //                  - Best so far!
        //                  - Still wildly inconsistent... sometimes it moves in the completely wrong direction! Other
times, everything as equal as I can make it, it runs perfectly.
        //                  - Actually works best with a relatively high motor delay (settings I were using were
around the 0.05, 0.1, and 0.1 for Order Delay, Motor Delay, and Max Speed, respectively)
        //                  - Suffers from the "end jutter problem," where it freaks when it is about to catch the
ball. Troublesome...
        //                  - Maybe this one overcomplicated things a bit. Next Algorithm's gonna be a bit simpler.

//==========================================================------------------------------------------------------
        //=====================================------------------------------------------------------
        //- -   -   -   -   -   -   -   -   -   -   -   -   -   -
        /*
        ballMovement = new Vector3(currentXPos - lastXPos, currentYPos - lastYPos, currentArea - lastArea);

        if (currentYPos > 10 && lastYPos > 10)
        {
            shouldStart = 1;         //We don't want junk data at the beginning

        }
        else
        {
            locomotion.SetDirection(Vector3.zero);
            ballMovement = Vector3.zero;
            //locomotion.SetMotionActivated(false);
            shouldStart = 0;
        }

        if (shouldStart == 1)
        {
           ballMovement.Scale(new Vector3(-.05f, 1f, 1f));

            if ((currentXPos < 320 - xThreshold || currentXPos > 320 + xThreshold) || (currentYPos < 240 - yThreshold
|| currentYPos > 240 - yThreshold)) { //&& currentArea < 14000) {
                //ballMovement.Scale(new Vector3(currentXPos - 320f, 1f, 1f));
                if (currentYPos < 240 - yThreshold)
                {
                    ballMovement.z = ballMovement.y;
                    ballMovement.Scale(new Vector3(.01f, 0f, .01f));
                }
                else if(currentYPos > 240 + yThreshold)
                {
                    ballMovement.z = ballMovement.y;
                    ballMovement.Scale(new Vector3(.01f, 0f, -.01f));
                }
                else
                {
                    ballMovement.Scale(new Vector3(1f, 0f, 0f));
                }
            }
            else{ //The ball is relatively in the center
                if (currentArea < areaThreshold) //We need to get closer
                {
                    ballMovement.Scale(new Vector3(1f, 0f, .001f));
                    //ballMovement.Scale(Vector3.zero);
                }
                else //Wait for it to come on in
                {
                    ballMovement.Scale(Vector3.zero);
                    locomotion.SetMotionActivated(false);
                }

             }

             locomotion.SetDirection(ballMovement);
        }
        */


         //- -   -   -  -    -   -   -   -   -   -   -   -   -   -
        //=====================================-------------------------------------------------------

//=========================================================-------------------------------------------------------
        //--==== ITERATION 4:         The idea behind this one is that we need to keep the ball centered in the camera
at all times. WE JUST DO THIS, though, not even taking into account
        //                                  the ball's change in direction. The idea behind this algorithm is "less is
more," you know?
        //
        //                      Tries to directly convert camera XYA coordinates into Unity's XYZ coordinates
        //                          Using a linear function.
        //
        //           Caveats:
        //                  + Assumes that the robot is always facing forward (i.e., rotation is 0,0,0)
        //                  + Does not currently support any logic to handle the tilting camera.
        //                  + Assumes Parabolic motion of the ball
        //
        //
        //          ++Results++
        //                  - Reeeeeallly suffers with larger motor delay time
        //                  - But so good with smaller delay times... it's a shame that it's likely this will not
occur.
        //                  - After some tuning this is actually looking pretty darn good!
        //                        (Still fast motor times, though: 0.05, 0.05, and 0.05 for Order Delay, Motor delay,
and Maximum Speed)
        //                  - Much more stable than the last 3 iterations.
        //                  - Of course, the slow speed means it's incapable of catching things significantly out of
its range.
        //                          (Maybe it would be possible to increase the cap on the speed based on how fast the
ball is moving? Next iteration?)
        //                  - "Loop-dee-loop" problem occurs when the ball is thrown at high Y component Force and low
Z component force (i.e., (0, 70, -5))
        //                          (In other words, it's impossible to catch a ball that's thrown straight into the
air)
        //                          (Maybe fix it by not going backwards if it makes the area smaller? IDK, something
for next iteration if anything)
        //                  - Similarly, it has trouble catching balls that are thrown so that they'll land far behind
the robot
        //
        //                  - To sum up the three above results, it works great when you're within the sweet-spot
range, but terribly otherwise. At least it's consistent (looking at you iterations 1-3)
        //
        //                  - Definitely think variable maximum speed needs to be implemented in the next iteration if
possible
        //

//=========================================================-------------------------------------------------------
        //=====================================-------------------------------------------------------
        //- -   -   -   -   -   -   -   -   -   -   -   -   -   -
        if (detected == 1 && currentYPos > 10 && lastYPos > 10)
        {
            shouldStart = 1;         //We don't want junk data at the beginning

        }
        else
        {
            locomotion.SetDirection(Vector3.zero);

            shouldStart = 0;
        }

        if (shouldStart == 1)
        {
            currentState = 1;
            ballMovement = new Vector3(1, 0, 1);
            if ((currentXPos < 320 - xThreshold || currentXPos > 320 + xThreshold) || (currentYPos < 240 - yThreshold
|| currentYPos > 240 - yThreshold)) { //&& currentArea < 14000) {
                ballMovement.Scale(new Vector3(currentXPos - 320f, 0f, 300f - currentYPos));
                ballMovement.Scale(new Vector3(.2f, 1f, .3f));

                currentState = 2;
                //ballMovement.Scale(new Vector3(1f, 1f, currentArea));

                //if (currentArea < 1500) //Special case if the ball has been thrown high but not far
                //{
                //    ballMovement.Scale(new Vector3(1f, 0f, .1f));
                //}
            }
            else
            {
                ballMovement.Scale(Vector3.zero);
                currentState = 3;
            }

            if (currentYPos > 465) //It's out of range, we'd better slowly look back to try and get it.
            {
                ballMovement.Scale(new Vector3(0f, 0f, .0005f));
            }


            if (currentArea > areaThreshold || currentYPos <= 9) //We got it
            {
                currentState = 4;
                ballMovement.Scale(Vector3.zero);
                //locomotion.SetMotionActivated(false);
            }
            else
            {
                //currentState = 5;
                locomotion.SetDirection(ballMovement);
            }
        }

    }
}
```