# Motion Capture and Tracking System with Raspberry Pi

Paul Canada, George Ventura, Christopher Iossa, Orquidia Moreno
Western Connecticut State University in Danbury, CT

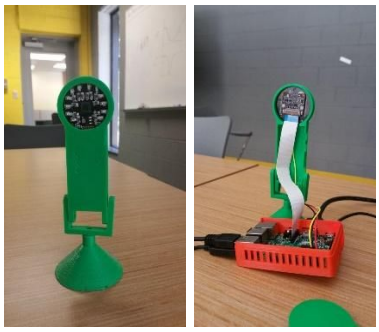

(a) (b) (c)

Figure 1:(a) Original image captured with marker highlighted. (b) Original image converted to grayscale. (c) Original image posterized with binary thresholding.

## Introduction

MoCap (or Motion Capture) is when a device is used to capture patterns and motion of an actor and transmit that data to a host where a simulation software displays and applies it to a virtual actor. MoCap has been one of the leading and most useful tools within animation in order to capture fluid and detailed motion. However, it is a tool that can be quite expensive for animators and game developers on tight budgets.

Professional MoCap camera setups can have cameras that are at least a thousand dollars each. By implementing a MoCap system using a Raspberry Pi Zero and IR camera, we hope to create a camera at a cap of $100 USD. This easy to use and inexpensive system can provide great advantages to those who are studying animation and using motion capture to create character motions, as well as a good introductory system to play around and get a feel for MoCap without committing to a high tech setup.



(a) (b)

Figure 2: (a) Prototype case holding Raspberry Pi Zero, NoIR camera, and IR light ring. (b) NoIR camera, light ring, Raspberry Pi 3 hooked up.

With the Raspberry Pi, we could create cameras that would be lightweight and easy to move around. We also planned to research how much processing the Pi's could do in order to lessen some of the calculations the host computer would need to do in order to keep track of markers.

## Our Approach

When developing and planning out this project, we decided to prototype with Python 3.6 and OpenCV 3.2 with the Raspberry Pi Zero. If we were to run out of resources, we would make a port to Processing with Java and the OpenCV library. Utilizing OpenCV, we could take in an image frame from the camera while it was capturing, convert it to grayscale, and then apply a binary threshold to the image. This would give us an image consisting of just black and white, where lighter shades of gray were converted to white and darker shades were converted to black, depending on the level of the threshold.

When the image is properly "thresholded", we are able to draw a bounding box around the areas with white space. To make these white spaces more apparent in the 2D plane, we used white reflective tape that would be able to reflect the IR light given off by the IR light ring. This way we can set our threshold to about 180-220, where 255 is the max. This ensures that the whitespace being captured in frame is actually the reflective marker being presented and any extra bits are filtered out.

Once we have the markers being found by the camera in each frame, we are able to create an object for each marker and extract the center point coordinates and timestamp and store them in the object.

$$centerX = \left(x1 + \left(\frac{x2}{2}\right)\right)$$
$$centerY = \left(y1 + \left(\frac{y2}{2}\right)\right)$$

(a)

Figure 3: (a) Equation for finding center point of marker given OpenCV bounding box rules.

The next step is to set up a host computer solution where the Pi can connect and transmit the marker data back to the host. This is further explained in the **Future Work** section.

## Implementation

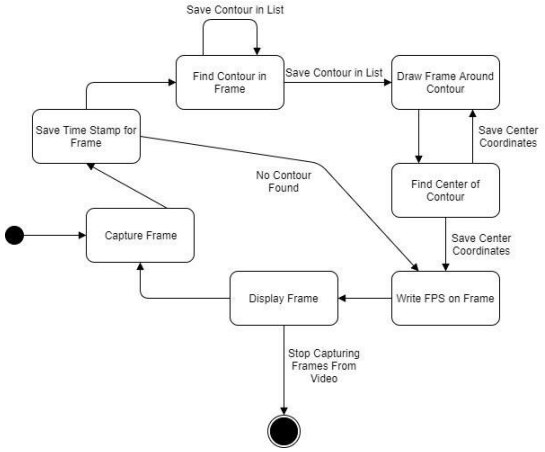
A Raspberry Pi 3, with a 64 bit quad core ARM processor equipped with a Sony IMX219 CMOS image sensor (NoIR camera), was used as the main hardware configuration. Installed on the camera unit was an infrared LED ring to emit IR light. Using the Raspbian Jessie Linux distro as the device operating system and by installing the Video4Linux (V4L) hardware driver, we were able to interface directly with the Camera Serial Interface (CSI) through the OpenCV library.

By using the newest release of the Raspberry Pi firmware and the V4L we were able to obtain frame rates of up to 90 fps at 320x240 and 40 fps at 640x480. It also allowed us to configure the camera's shutter speed, exposure, gain and ISO levels.

As the camera is initialized, each frame is sent to a subroutine which will process the frames into grayscale format. With the grayscale format, the image is posterized using a binary threshold mask using a fixed constant 8-bit integer T. All values less than the threshold value will become a black pixel and all greater a white pixel. Using this method, we were able to isolate our target markers.

Using the threshold mask, all contours in the image were mapped, and a bounding rectangle was drawn around all objects that were found. The center point of the bounding rectangle was calculated and stored in an object that would be displayed in console. For the future, this information, along with a timestamp of the frame that the marker is in, will be sent to a host computer.

We set a hard cap as to how many markers were able to be drawn, so as not to increase the number of calculations being done when there are any blobs on screen. When testing in a non-configured environment, the capture will sometimes pick up blobs that are not intended; e.g. a watch face, or any similar reflective surface. However, when in an appropriate environment such as a dark room without natural or fluorescent lighting, little to no miscellaneous blobs are detected.




(a)

Figure 4: (a) State diagram showing workflow for marker detection system.

## Future Work

Future work involves resolving some of the issues we encountered during this first summer. Namely, the frame rate issues we encountered when running the software on the Pi Zero. While we were able to get into the 25-40 FPS range on a Pi 3 depending on the number of markers present, the Pi Zero averages out at around 15 FPS. This presents a significant hurdle.

Some preliminary solutions we devised are to lower the capture resolution or move to the Pi 3. Both have drawbacks, with a lower capture resolution, the blobs have a greater chance of colliding. Moving to the Pi 3 on the other hand would reduce our battery life when the design is finalized.

Another issue we encountered is consistent labeling of blobs. The blobs are currently labeled in order of their detection in the raster (bottom to top, left to right). This is not ideal, as we would like to track each blob based on their historical labels, not on a frame by frame basis. Similarly, we need to decide how to handle markers that go out of frame and come back into frame, should they be labeled the same as they were before they went out of frame (reserved label) or given the first available label (dynamic labeling)?

In time, we will need to calculate the camera's position relative to other cameras, add a gyroscope to the camera in order to track (yaw, pitch, roll), an on off switch for the cameras, modify the software so that it launches on boot and requires little or no user interaction, make the camera battery powered, and modify the casing to accommodate all these components. For the host computer, we need to implement point triangulation and devise a means of communication between the host computer and each individual camera. We must also decide if a port to a compiled language will yield performance gains with calculations and video capture.