H. Chase Stevens, s1107496

Joris Sebastijonas Urbaitis, s1104617

Introduction to Vision and Robotics Practical 1 Report

Introduction

In order to detect the objects found in the videos, our Matlab program processed each frame and detected moving objects. The information from different frames was linked together to form a coherent picture of the movement of the objects being thrown. We then detected whether these objects were balls and used basic kinematics to detect where the apex of their trajectory would be.

Methods

Thresholding

In order to detect the objects, we first have to preprocess each frame. First, the image is converted to grayscale. We then take the absolute difference between the frame and the reference background image. We use the result to create a histogram of greyscale values and use the algorithm provided during labs (`findthresh.m`) to find a threshold. The difference image is then filtered and converted to black and white using this threshold and, as a final step, `bwmorph`'s clean method is used to remove lone pixels. The remaining blobs in the image are then expanded, with pixels within 5 pixels of a white pixel being made white. This means that nearby blobs become connected. The downside is that they become more circular and make ball detection trickier.

Initially, the first frame of the video is selected to be the reference background image and converted to greyscale. Each frame, the current reference background image is split into a set number of horizontal slices. The total difference between the pixels of a background slice and the corresponding slice in a greyscaled version of the current frame is checked. If this distance is under a set threshold, that slice is updated by setting it to be the current frame. This method was chosen to avoid cases in which parts of the background had been reliably altered, but a moving object elsewhere in the frame was blocking this part of the background from being updated. The total effect of these efforts was that if the background reference is no longer providing sufficient information about the moving objects in the frame, it is updated. This allows us to quickly get rid of noise appearing in various parts of the image that have nothing to do with the objects being thrown up, for example, when the background cloth moves.

Detecting connected components

To detect the connected components in the thresholded black and white image we used the Matlab function `bwconncomp,` with the default connectivity of 8. This checks a 3x3 region around every non-zero pixel for non-zero values. It then uses the number of these to classify the pixel either as

part of a larger component or, if it is not connected to any known component, as a novel component. The function ultimately returns a list of pixel indices for every component that we then use for object and movement detection.

Linking data from different frames

In order to link data between different frames we first filter the connected component data in order to remove objects that are very small or very large and likely to be noise. We then find the center of each object and calculate a pseudo-Euclidean distance in three-dimensional space (x coordinate, y coordinate, and time) between this object and every object detected in the last few frames, using the following formula:

$$dist = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2 + (|t_0 - t_1|)^3}$$

The closest of these within a specified threshold is deemed to be the same object; if no past object falls within this threshold we create a new object with a unique identifier. It should be noted that, in our implementation, it is possible for multiple connected components to sometimes be classified as the same object, allowing for error in their detection.

Apex of the trajectory

Kinematics is used to detect if an object has reached the highest point on the Y-axis of its trajectory. After linking data from different frames we can calculate the Y-axis velocity of the object. We know that the velocity at the apex of the trajectory of the ball will be zero. We also know that velocity can be calculated using the following physics formula:

$$v = v_0 + at$$

Since the relevant acceleration force in this case is gravity and v_0 can be assumed to be 0 (as we are calculating relative to the current moment in time), we can derive the time t at which the velocity will be 0 by calculating:

$$t = \frac{v}{g}$$

Here g is the gravitational constant. It should be noted that although we use the accepted value of 9.8 meters per second squared for the gravitational constant, the relationship this bears to our final time calculation is tenuous at best.

After calculating the time t at which the velocity will be zero, we can use it to decide whether the object is close to its apex by considering the size of t. If t is under a certain threshold, we mark the object as being at its apex. This can have the result of marking the apex very slightly before the actual highest point, but this is mostly invisible to the human eye in our experiments. The upside is that we do not consider future frames or delay displayed frames to calculate this.

Ball detection

We take the area of the connected component we are trying to classify. We 'overlay' a circle of the same size on top of it and check what percentage of the object falls within the circle, giving us a compactness metric. Using a pre-set threshold we can then decide whether the object is a ball or not.

Results

We didn't use one of the videos for testing, saving it to be run only with the final version of our code. The results of object classification in that run are as follows:

	Positives	Negatives
True	6	2
False	2	0

In all but one of the true positive cases, the apex was detected at the correct time – in the outlier case, the apex was detected earlier than was appropriate.

Image pipeline

Here, we outline the steps images go through when being processed. This process begins with a raw, full-color frame (see Figure 1) and a background image (see Figure 2), and returns a mask, which is then in turn used (in conjunction with previous data on objects) to infer current objects. We take the full-color frame and translate it into greyscale, then take the absolute difference of this and the background (see Figure 3). We then threshold this resultant image to create the binary image seen in Figure 5 which, after applying `bwdist`, becomes Figure 6. After linking the connected components in this binary image to previously seen objects, we can draw traces of these objects' previous histories while using the binary image as a mask for the frame, resulting in Figure 8.

Figure 1. Raw frame 768 from video GOPR0008.

Figure 2. Background image used at time of frame 768.

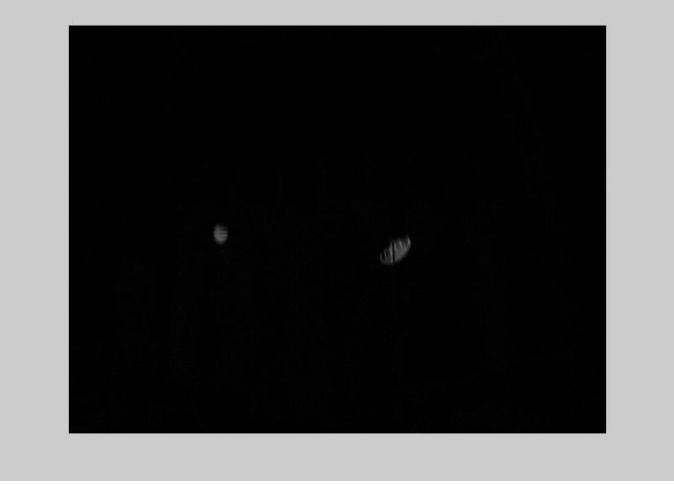

Figure 3. Absolute background-frame difference.

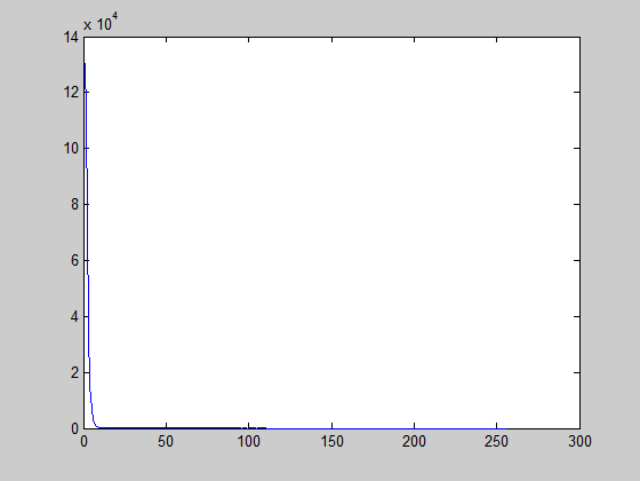

Figure 4. Histogram of Figure 3.

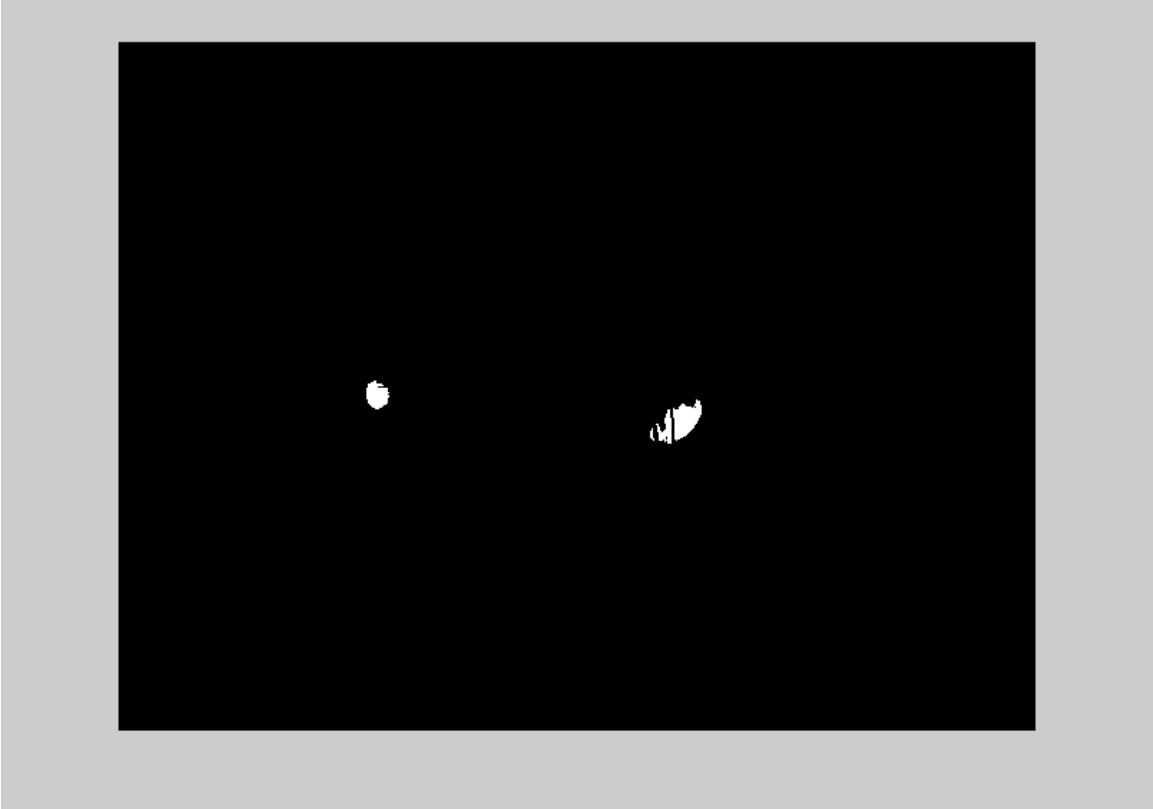

Figure 5. Thresholded background-frame difference.

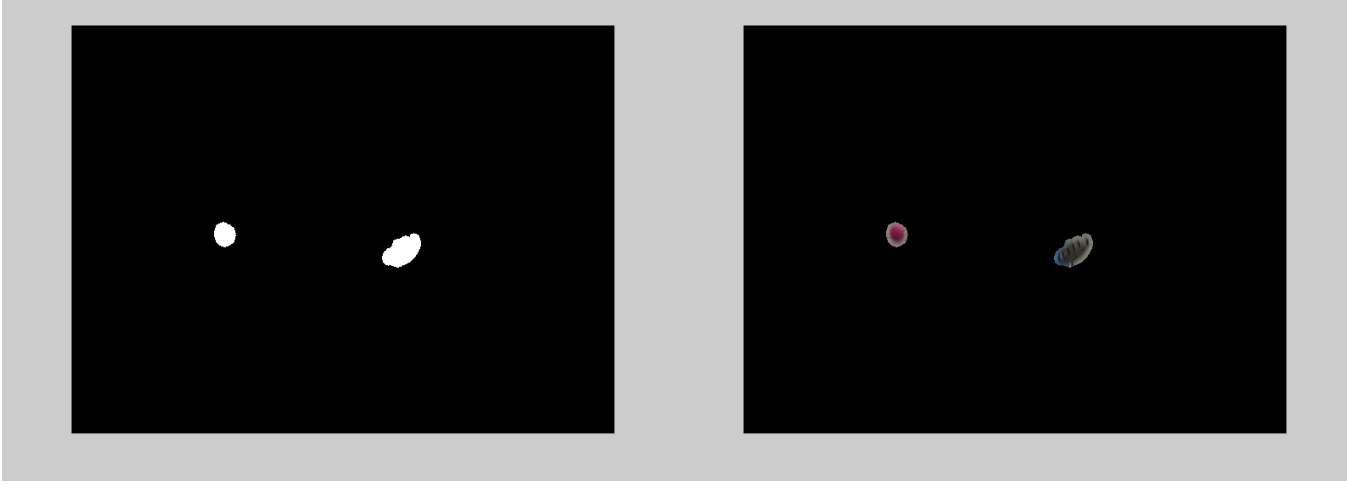

Figure 6. "Thickened" thresholded image.

Figure 7. Figure 6 used as a mask for frame 768.

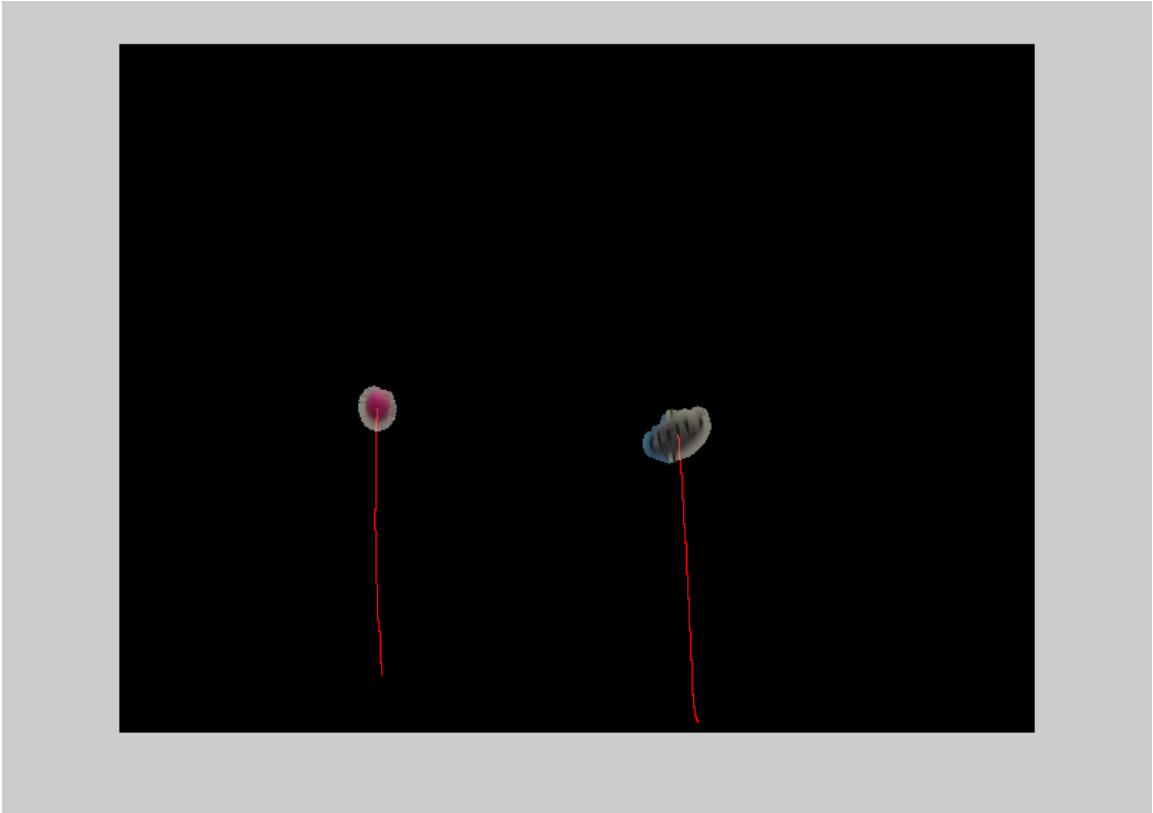

Figure 8. Masked frame with object traces.

Errors

Our software has a number of quirks that prevent perfect detection and classification of objects, as outlined below.

Misdetection of objects

Stemming largely from flaws in our keying out process (see Figure 7 for a refresher), we repeatedly included some parts of the background noise in our mask. Although smaller noise blobs were ignored, larger ones were incorrectly identified as being objects, which lead at very least to their being tracked and, in some rare cases (see Figure 9), triggering an apex event. This complication especially became an issue in videos with large and relatively sudden changes in the background, such as videos in which jettisoned objects collide with objects in the background. The potential for this happening with a great deal of noise otherwise was mitigated by our use of the `bwmorph` clean function.

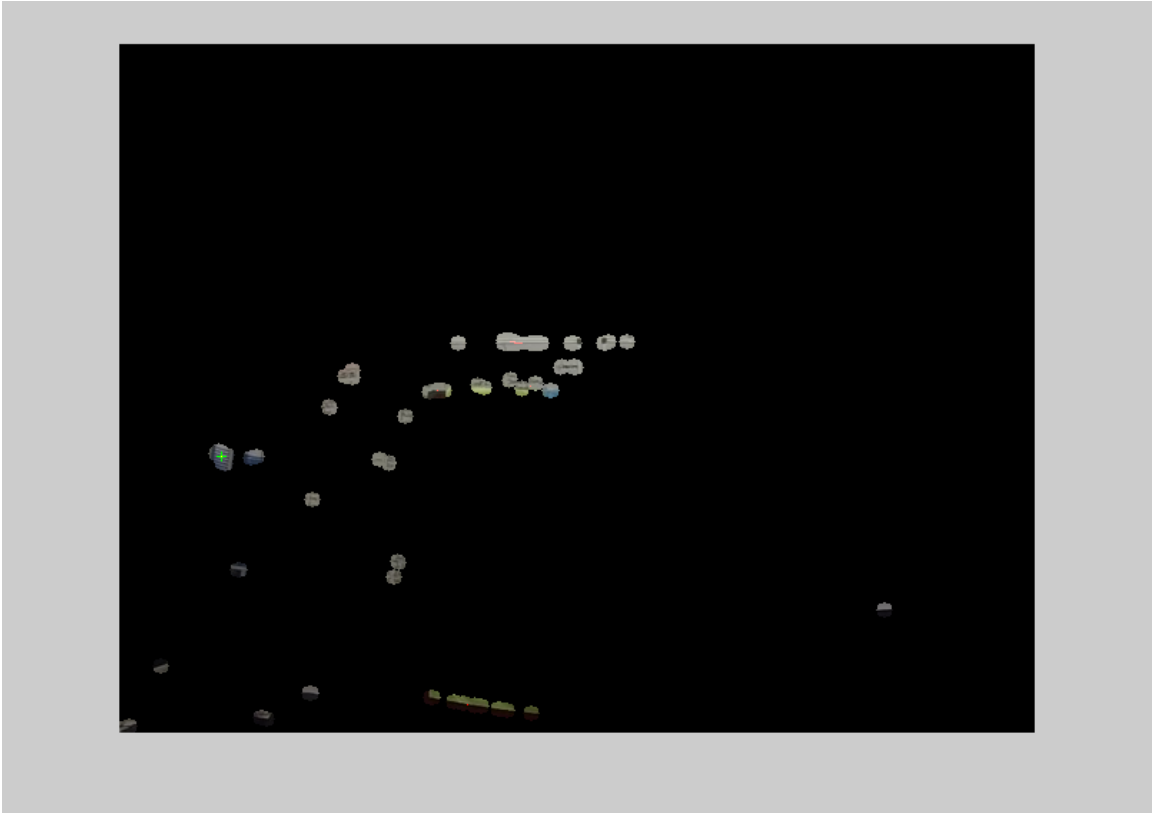

Figure 9. GOPR0008 frame 347. Background noise triggers an apex event as demarcated by the green cross.

Misclassification of objects

As mentioned in the previous subsection, our program is prone to misclassification, in large part due to our reticence to allow false negatives. In particular, our efforts to consistently identify the yellow ball found in many videos as being a ball object (and subsequently tagging its apex) allowed other objects, most notably rectangular prisms and rugby balls, to be classified as balls. The challenge presented by the yellow ball was a direct result of its extreme similarity in color to the background. This in turn required us not only to lower the threshold found by `findthresh.m` and "thicken" the mask generated by `generate_keying_mask.m`, but to also be more lenient in our threshold used to identify balls by object compactness. An example of this poor behavior can be seen in Figure 10.

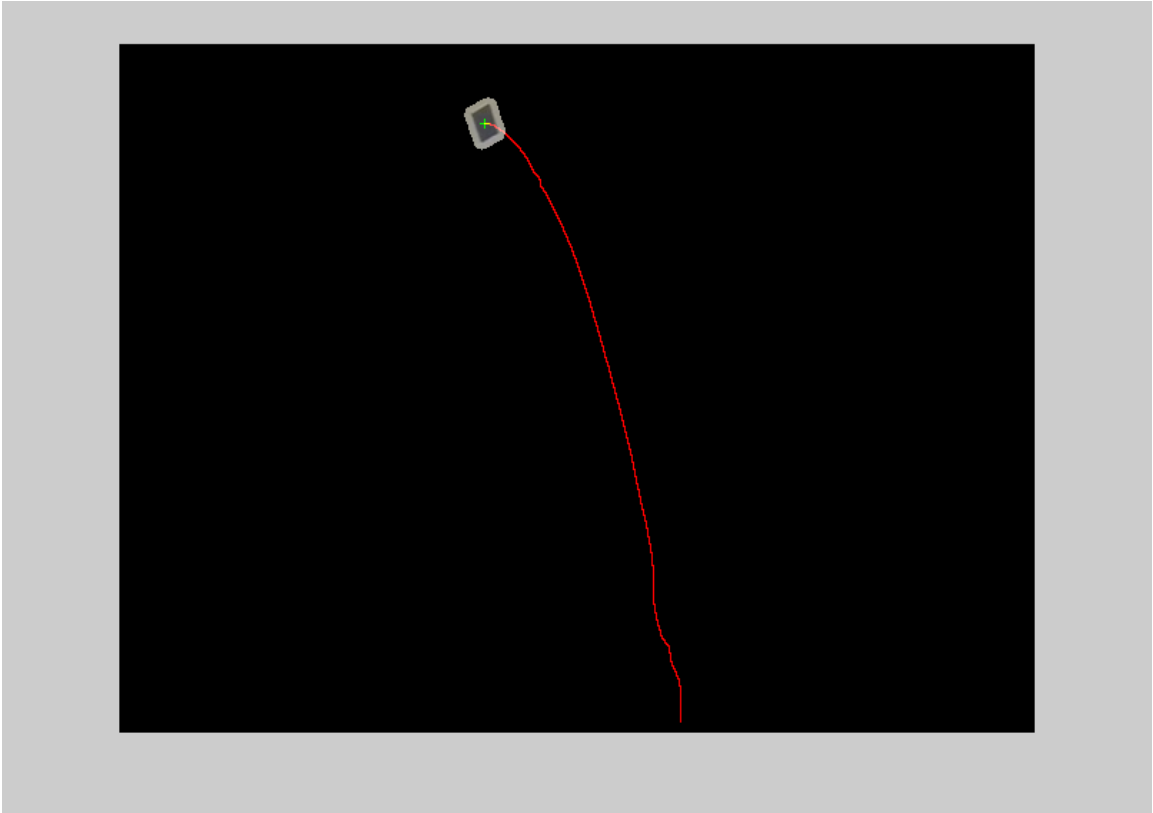

Figure 10. GOPR0008 frame 1082. A cube at its apex is misclassified as a ball.

Incorrectly timed apexes

In a singular circumstance, an object was identified as being at its apex (Figure 11) well before it actually was (Figure 12). Though we were unable to identify any one step in the process responsible for this failure, it was likely a combination of our naïve kinematics calculations as well as the object in question (the yellow ball) being only partially keyed out.

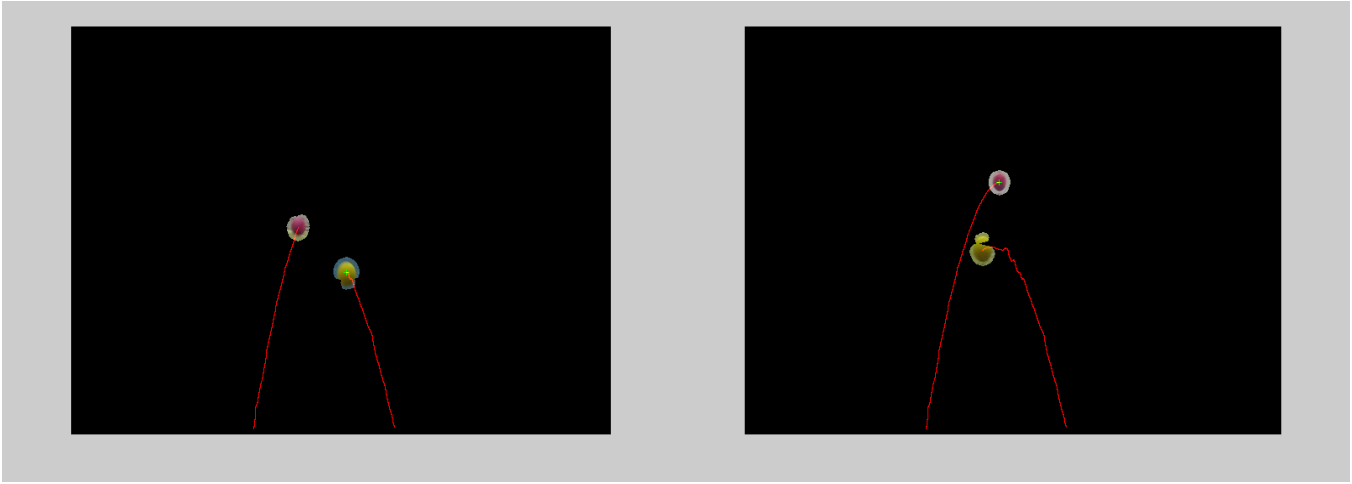

Figure 11. GOPR0005 frame 607. The infamous yellow ball has its apex tagged too early in its trajectory.

Figure 12. GOPR0005 frame 628. The yellow ball as positioned closer to its true acme.

Objects keyed into multiple non-contiguous connected components

Although our technique of allowing many-to-many mappings of past object identifications to current-frame objects helped limit the impact of this phenomenon, single objects becoming multiple disjoint connected components caused errors in trajectory tracking and object classification. As previously mentioned, this was most pertinent in the case of the yellow ball, but this effect also impacted other objects, primarily, the kangaroo (as seen in Figure 13) and the longer rectangular prism. We surmise that this was primarily caused by some regions of the objects having different lighting, making them too similar to the background image.

Figure 13. GOPR0005 frame 1470. The kangaroo is broken into multiple connected components.

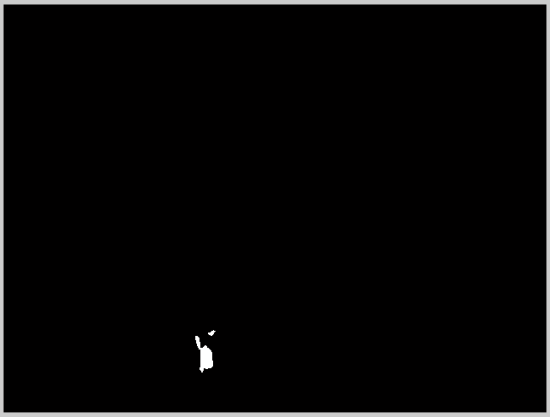

Figure 14. The binary mask for GOPR0005 frame 1470, before thickening.

Discussion

What works

Overall, our program performed amicably. Path marking of thrown objects generally works to a sufficiently acceptable degree, and object detection, while occasionally thrown off by background noise, very reliably is able to capture objects in-frame, in large part due to our robust keying techniques. Object linking also works with a very high degree of reliability, and never caused visible issues. In our training set there was only one occurrence of a false negative, and there were zero occurrences in the test video. We were generally very pleased with the speed of our program.

What doesn't

Many thrown objects are mistakenly classified as balls, leading to high levels of false positives both in the training data and the test video. Keying out of objects was in some circumstances too lenient (including parts of the background in our mask) and in other circumstances too rigid (not including parts of thrown objects).

Both trajectory marking and apex detection were affected by the shifting positions of objects due to poor keying. Additionally, apex detection was precise only to a limited degree; apex detection was also affected by our naïve approach to kinematics which only used the two most recent known object positions to calculate velocity. Because object classification was done nearer to the apex to

reduce motion blur, there is a noticeable lag in frame playback as this processing-intensive operation is carried out.

Next time

A more sophisticated approach to object classification would be highly preferable. This could take many forms. The use of edge detection or the incorporation of color data might be advantageous. A less heavy-handed approach to classification might also be found in training a Bayesian classifier on our available data; this classifier could be used to effectively capitalize on moment calculations.

Our kinematics calculations were very basic. A more advanced approach to this issue would be attempting to fit a second-order polynomial to the observed positions of our objects, or even simply smoothing observed velocities over multiple observations.

A keying approach informed by known object motion could plausibly have intelligently filtered out many parts of the background.

We failed to make full use of our linking system: although we ended up with multiple connected components being classified as parts of the same object in many frames, we didn't use this to inform object classification or even object trajectory.

Work Distribution

The majority of coding was done using the “extreme” programming paradigm, with both software engineers sitting at the same workstation. There were few exceptions to this during which one of us coded independently while our partner was otherwise indisposed. This report was written in a similar fashion. We recommend a 50:50 split of the mark.

Code

For convenience, all code is also available on github upon request. Email chase@chasestevens.com with subject "IVR github access request".

videoread.m

```
file_dir = './GOPR0005/'; %put here one of the folder locations with images;
filenames = dir([file_dir '*.jpg']);

frame = imread([file_dir filenames(1).name]);
grey_back = rgb2gray(frame);

figure(1); h1 = imshow(apply_mask(generate_keying_mask(grey_back, grey_back),
grey_back));

READJUSTMENT_THRESH = 1e-4;
BACKGROUND_LOOKBACK = 3;

OBJECT_SIZE_THRESH = [300, 5000];
OBJECT_LOOKBACK = 3;

% sqrt(10^2 + 10^2 + 2^3), based on distance calculation in update_objects
OBJECT_LINKING_DIST_THRESH = 18;

INITIAL_FRAME = 607;

[X, Y] = size(grey_back);
global IMG_WIDTH
IMG_WIDTH = X;

[frame_count, x] = size(filenames);
prev_frames = {};
prev_frames{frame_count} = []; % preallocation
object_history = {};
object_history{frame_count} = {};

% Read one frame at a time.
for k = INITIAL_FRAME : frame_count

    % Read in the next frame and convert it to grayscale
    frame = imread([file_dir filenames(k).name]);
    scene = rgb2gray(frame);

    % Generate the mask and expand the blobs in it
    mask = generate_keying_mask(scene, grey_back);
    mask = bwdist(mask) <= 5;

    % Apply the mask to the frame for later display
    masked_frame = apply_mask(mask, frame);

    % Find the connected components in the mask
    conn_comp = bwconncomp(mask);

    % Find objects from the connected components and check if they are at
```

```
    % their apex
    [object_history, new_objs] = update_objects(k, object_history, conn_comp,
OBJECT_LOOKBACK, OBJECT_LINKING_DIST_THRESH, OBJECT_SIZE_THRESH);
    object_history = assign_apexes(k, object_history);

    X = max(size(object_history{k}));
    obj_at_apex = 0;

    for i=1:X,
        obj = object_history{k}{i};

        if obj.apex_found == 1
            % Display a cross on the frame
            masked_frame = insertMarker(masked_frame, [obj.x, obj.y]);

            % Mark that the apex of this object has been processed and
            % should no longer be checked
            object_history{k}{i}.apex_found = 2;

            obj_at_apex = 1;
        end
    end

    % Display the trajectories of detected objects
    if k > INITIAL_FRAME
        masked_frame = show_paths(masked_frame, object_history, k, INITIAL_FRAME);
    end

    % Display the frame with the mask applied
    set(h1, 'CData', masked_frame);
    drawnow('expose');

    % Check if we need to pause for the apex frame
    if obj_at_apex == 1
        pause(3);
    end

    % Update the background reference image
    [grey_back, prev_frames] = update_background(k, grey_back, prev_frames, mask,
scene, BACKGROUND_LOOKBACK, READJUSTMENT_THRESH, INITIAL_FRAME);

    % Print processing info
    disp(['showing frame ' num2str(k) ', current objects: '
num2str(conn_comp.NumObjects) ' (' num2str(new_objs) ' new)']);
end
```

update_objects.m

```
function [ history, new_objs ] = update_objects( time, history, conn_comp, lookback,
dist_threshold, size_threshold )

    [~, Z] = size(conn_comp.PixelIdxList);
    lower_size_thresh = size_threshold(1);
    upper_size_thresh = size_threshold(2);

    % Get the centers of the connected components
    props = regionprops(conn_comp, 'centroid');
    cc_centers = cat(1, props.Centroid);

    new_objs = 0;
    history{time} = {};

    obj_index = 0;

    for z=1:Z,

        % Check if the component is too large or too small
        % to be considered an object
        area = max(size(conn_comp.PixelIdxList{z}));
        if area < lower_size_thresh || area > upper_size_thresh
            continue % 2small2careabout
        end

        obj_index = obj_index + 1;

        comp_x = cc_centers(z, 1);
        comp_y = cc_centers(z, 2);
        lowest_dist = 1000000;
        no_obj_match = 1;
        lowest_dist_x = -1;
        lowest_dist_y = -1;
        lowest_dist_t = -1;

        % Check for nearby objects in multiple timeslices
        for time_slice=max(1, time - lookback - 1):time-1
            num_objs = max(size(history{time_slice}));

            for i=1:num_objs,
                past_obj_x = history{time_slice}{i}.x;
                past_obj_y = history{time_slice}{i}.y;

                % Calculate distance to object. Using time as well since we
                % want older observations to have less weight
                dist_ = sqrt((comp_x - past_obj_x)^2 + (comp_y - past_obj_y)^2 +
abs(time - time_slice)^3);

                % found new closest obj from our sordid past
                if (dist_ < lowest_dist) && (dist_ < dist_threshold)
                    lowest_dist_id = history{time_slice}{i}.id;
                    lowest_dist_x = history{time_slice}{i}.x;
```

```
                    lowest_dist_y = history{time_slice}{i}.y;
                    lowest_dist_t = time_slice;
                    lowest_dist_apexfound = history{time_slice}{i}.apex_found;
                    lowest_dist = dist_;
                    no_obj_match = 0;
                end
            end
        end

        new_objs = new_objs + no_obj_match;
        if no_obj_match == 1
            apex_found = 0;
            lowest_dist_x = comp_x;
            lowest_dist_y = comp_y;
            lowest_dist_t = time;
            lowest_dist_id = strcat(num2str(time), '_', num2str(z));
        else
            apex_found = lowest_dist_apexfound;
        end

        % Store the object in the history
        history{time}{obj_index} = struct('x', comp_x, 'y', comp_y, ...
            'id', lowest_dist_id, ...
            'y_apex', -1, 'apex_found', apex_found, ...
            'idxlist', conn_comp.PixelIdxList{z}, ...
            'prev_x', lowest_dist_x, 'prev_y', lowest_dist_y, 'prev_t',
lowest_dist_t);
    end

end
```

update_background.m

```
function [ grey_back, prev_frames ] = update_background( frame_num, grey_back,
prev_frames, mask, current_scene, background_lookback, readjustment_threshold,
initial_frame )

    [X, Y] = size(mask);
    CHUNKS = 1; % number of horizontal strips to split image into
    increment = X * Y / CHUNKS;

    % Convert images to 1D vectors
    prev_frames{frame_num} = reshape(mask, 1, X*Y);
    temp_grey = reshape(grey_back, 1, X*Y);
    temp_scene = reshape(current_scene, 1, X*Y);

    % Only start updating the background after we have enough frames to
    % work with
    if frame_num > background_lookback + initial_frame

        % Update the background reference in chunks
        for chunk=1:CHUNKS,
            % Compute coordinates of the chunk in the 1D image vector
            i = 1+((chunk - 1)*increment);
            j = chunk*increment;

            % Extract the chunks from the previous and current frame
            prev = prev_frames{frame_num - background_lookback}(i:j);
            cur = prev_frames{frame_num}(i:j);

            % If the difference between the two chunks is sufficient,
            % update the background
            diff = double(sum(abs(prev - cur))) / double((X*Y));
            if diff <= (readjustment_threshold / sqrt(CHUNKS))
                temp_grey(i:j) = temp_scene(i:j);
            end
        end
    end

    % Convert the background reference image to a 2D matrix before return
    grey_back = reshape(temp_grey, X, Y);

end
```

show_paths.m

```
function [ img ] = show_paths( img, object_history, time, initial_time )

    for t=time:-1:(initial_time + 1), % for every time t
        object_count = max(size(object_history{t}));

        for obj_i=1:object_count, % for every old object at t
            obj = object_history{t}{obj_i};

            cur_object_count = max(size(object_history{time}));
            for obj_j=1:cur_object_count % for every current object
                cur_obj = object_history{time}{obj_j};

                if strcmp(obj.id, cur_obj.id) % if the old object is the current
object
                    % draw the old object's line to its previous position
                    img = draw_line(img, obj.x, obj.y, obj.prev_x, obj.prev_y);
                    break
                end
            end
        end
    end
end
```

isball.m

```
function [ is_ball ] = isball(center_x, center_y, idxlist)
    % Checks if an object is a ball by checking how many of its pixels fall
    % within an imaginary circle of the same area

    THRESHOLD = .905;

    is_ball = false;

    % Find the radius of a circle of the same area as the object
    area = max(size(idxlist));
    radius = area / pi;

    pixels_inside = 0;

    % Find how many pixels fall within the circle
    for i=1:area,
        [X, Y] = getxy(idxlist(i));
        dx = abs(center_x - X);
        dy = abs(center_y - Y);

        % Check if the pixel is within the circle
        if double(dx^2+dy^2) <= radius,
            pixels_inside = pixels_inside + 1;
        end

    end

    if double(pixels_inside)/double(area) > double(THRESHOLD)
        is_ball = true;
        disp(double(pixels_inside)/double(area))
    end
end
```

getxy.m

```
function [ x, y ] = getxy( coord )
    % Convert coordinate indexing into a 1D image into [X,Y] for a 2D one
    global IMG_WIDTH
    y = mod(coord, IMG_WIDTH);
    x = idivide(cast(coord, 'uint32'), cast(IMG_WIDTH, 'uint32'));
end
```

generate_keying_mask.m

```
function [ thresholded_diff ] = generate_keying_mask( scene, background )

    %params
    MIN_THRESHOLD = 16;
    THRESHOLD_MULT = 0.6;

    %get differences between background, scene:
    diff = (max(background, scene) - min(background, scene));

    %create histogram:
    edges = linspace(0, 255, 256);

    [X, Y] = size(scene);
    vec = reshape(diff, 1, X*Y);      % turn image into long array
    histogram = histc(vec, edges)';         % do histogram

    %smoothing hist:
    threshold = findthresh(histogram, 6, 0); % params taken from lab
    threshold = THRESHOLD_MULT * threshold;
    threshold = max(threshold, MIN_THRESHOLD);

    thresholded_diff = zeros(X, Y);

    threshold_indices = diff > threshold;
    thresholded_diff(threshold_indices) = 255;
    thresholded_diff = bwmorph(thresholded_diff, 'clean', 1);
end
```

draw_line.m

```
function [ img ] = draw_line( img, x1, y1, x2, y2 )
    % Compute a set of points for the x and y axes
    cpts = linspace(x1,x2,500);
    rpts = linspace(y1,y2,500);

    % Get the linear indices corresponding to these points
    index = sub2ind(size(img),round(rpts),round(cpts));

    % Set the color of the points on the line to red
    img(index) = 1.0;
end
```

assign_apexes.m

```
function [ object_history_new ] = assign_apexes( time, object_history)

    object_history_new = object_history;

    if time <= 1
        % We don't have the necessary history of the objects to evaluate
        return
    end

    Z = max(size(object_history{time}));

    GRAVITY = 9.8;
    TIME_DIFF = 0.04;

    for obj_i=1:Z,
        obj = object_history{time}{obj_i};

        % If we already found the apex for this object, skip it
        if obj.apex_found ~= 0
            continue
        end

        vel = (obj.y - obj.prev_y) / double(time - obj.prev_t);
        t = vel / GRAVITY;

        if abs(t) < TIME_DIFF
            % We only care about apexes of balls
            if isball(obj.x, obj.y, obj.idxlist)
                object_history_new{time}{obj_i}.apex_found = 1;
            end
        end
    end
end
```

apply_mask.m

```
function [ output ] = apply_mask( mask, image )
    [X, Y, Z] = size(image);
    output = zeros(X, Y, Z);

    for i=1:X,
        for j=1:Y,
            if mask(i, j) >= 1
                % Convert to RGB with values ranging from 0 to 1
                output(i, j, :) = double(image(i, j, :)) / double(256);
            end
        end
    end
end
```