5-2013

3D Object Tracking and Motion Profiling

Corey A. Spitzer
University of Nebraska at Omaha

3D Object Tracking and Motion Profiling

A Thesis

Presented to the
Department of Computer Science
and the
Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

University of Nebraska at Omaha
by
Corey A. Spitzer

May, 2013

Supervisory Committee:
Quiming Zhu
Zhengxin Chen
Hamid Sharif

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Abstract

3D Object Tracking and Motion Profiling

Corey A. Spitzer, MS

University of Nebraska, 2013

Advisor: Quiming Zhu

In order to advance the field of computer vision in the direction of "strong AI", it's necessary to address the subproblems of creating a system that can "see" in a way comparable to a human or animal. Due to very recent advances in depth-sensing imaging technology, it is now possible to generate accurate and detailed depth maps that can be used for image segmentation, mapping, and other higher-level processing functions needed for these subproblems. Using this technology, I describe a method for identifying a moving object in video and segmenting the image of the object based on its motion. This creates a coarse vector field where each segment denotes a region of the object that is moving in the same general direction, rounded to the nearest 45 degrees. The approach described combines a conventional background subtraction algorithm, depth sensor data, and a biologically-inspired artificial neural circuit. In most cases the entire process can execute in near real time as the video is captured and is reasonably accurate.

Table of Contents

List of Figures

I. Introduction

One of the long-term goals of artificial intelligence research in the area of computer vision is to create a system which can "see" in a way comparable to a human or animal. With such a technology, we could develop systems that could automatically interact with the world through specialized hardware in a meaningful and useful way.

Accomplishing this goal requires addressing all of the difficult subproblems, one of which is how to identify, track, and model a moving object. This is something that even three-month-old infants can do without any conscious effort [Hamer 1990].

Specifically, what I've attempted to do is identify a single moving object in a video and segment said object according to the individual directions of movement of each portion of the object. This is a fundamental prerequisite toward the animal capabilities of predicting future locations and paths of objects in the physical world. This technology can be used in applications where movement needs analyzed as in sports, defense, robotics, and other industries.

In so doing, I have also attempted to incorporate knowledge and models of biological circuits and systems as much as possible in an effort to advance the field of computer vision and biomimicry.

The major challenges of this research lied first in the question, "What is an object?" For

example, if a man was filmed standing still except only his arm was moving, would the moving object only be constrained to the arm, the entire person, or some other region? Second, whatever is logically considered the moving object, a vector field would have to be created to assign motion vectors to specific regions. Finally, how would such a vector field be segmented into regions of (near-)homogenous movement? How many segments should there be and what threshold(s) should be applied?

My hypothesis was that the depth information provided by the Kinect sensor would provide an affordance that would greatly simplify the tracking problem and/or substantially increase the accuracy of a tracking algorithm based on previous work.

II. Overview

II.1. History of the problem

The problem of tracking objects through a sequence of images has been studied in one form or another for decades. Researchers have analyzed image sequences, looking at a variety of aspects and using several different apparatus: 2D and 3D analysis, point and line-based features, one or multiple cameras, etc., but only recently has the relevant science and technology advanced far enough to properly address the complexities of the real world.

One of the earliest publications addressing object tracking was [Smith and Buechler 1975]. In it, Smith and Buechler used the Kalman filter, a special kind of temporal Bayesian network which can be used to estimate a state given noisy measurements of past states. They presented an algorithm for mapping the trajectory of moving objects against a stationary background.

In [Falconer 1977], the Hough transform, a technique most commonly associated with edge detection in still images, was repurposed to track multiple targets moving in straight lines. Falconer imposed limiting constraints on his approach; the movement of the targets must be linear and the images with which he worked were amenable: simplistic, if not binary, scenes where segmentation was assumed to be perfect.

In [Aggarwal and Wang 1987] a simple pre-defined set of four points and one line that

were assumed to lie on the same rigid moving object were used for tracking across frames of video. Given the constraints and a priori knowledge, the motion of the camera could be deduced.

[Zhengyou and Faugeras, 1992] expanded on this work by using stereo cameras to correspond and track 3D line segments. Using a Kalman filter, they addressed the problem of an autonomous vehicle navigating in an unknown environment where other objects would also be moving.

II.2. State of the art

The state of the art of motion tracking and mapping has been led by the Microsoft Research team which has recently addressed real-time dense mapping and tracking in 3D.

In [Newcombe et. al. 2011], streaming data from a Microsoft Kinect depth sensor is fused in real time to create a 3D model of scenes with high accuracy. The approach is based on extensions to a combination of the Parallel Tracking and Mapping (PTAM) system as well as an iterative closest point (ICP) algorithm for estimating the relative motion of the camera between consecutive poses (or frames).

The system first reads raw depth data from the sensor and constructs vertex and normal vector maps of the scene (surface measurement). Then an ICP algorithm tracks the depth data from consecutive frames to estimate the movement of the camera. The new surface

measurement is then integrated into the world model (a volumetric, truncated signed distance function representation). A surface prediction is made by raycasting the signed distance function into the estimated frame. Finally, a multi-scale ICP alignment between the predicted surface and current sensor measurement improves the sensor pose estimation.

Using this technology, the Microsoft Research team is able to fully map a scene and create an accurate 3D model with a high level of detail in real time by moving the camera around the space. In addition, experiments have been performed showing robust tracking of moving objects in the modeled scene [Izadi et al. 2011].

III. Principles and Concepts

Humans have evolved the ability to effortlessly and unconsciously process a large amount of information from visual input between one moment and the next. We can recognize the distances and relationships between an object and its surroundings as well as the direction and speed of movement while concurrently predicting its future path and position. Our brains are hardwired from birth to eventually perform these essential tasks.

Given the equivalent information as the human visual system gathers -- a 2D projection of colors and intensities -- and even with (in a sense) superior depth information, our task is ultimately to create a technology to give the machine the same capability.

The human retina consists of multiple layers of cells including a layer of photoreceptors which absorb light and turn it into electrochemical signals. These signals are proportional to the intensity and wavelength of the light absorbed so that is the information that is sent on to neural circuits. Likewise, the sensor used in this thesis absorbs light and sends electrical signals to a computer. Just as the brain must take inputs from two eyes and compute the depth information, the sensor uses an infrared grid projected onto the scene which is detected by an infrared camera to create a depth map.

Both the brain and the computer receive “frames” from the scene being observed. The brain can process two subsequent frames and recognize 1. that an object is moving, 2. the boundaries of the moving object against the background and occluding objects, 3. which

regions of each frame image belong to the moving object, and 4. the direction and speed of the motion. Since we don't have a full understanding of how the brain does this, it is a challenge to create a system that will allow a machine to perform the same tasks.

IV. Techniques

IV.1. Feature Selection Techniques

Researchers have segmented images for tracking based on regional features such as color, texture, and optical flow. Color segmentation is relatively trivial, but for identifying regions for a given texture, common techniques include using Gray-Level Cooccurrence Matrices (a 2D histogram which shows the cooccurrences of intensities in a specified direction and distance), Law's texture measures (twenty-five 2D filters generated from five 1D filters corresponding to level, edge, spot, wave, and ripple), wavelets (orthogonal bank of filters), and steerable pyramids [Yilmaz et al. 2006].

Optical flow is described by [Yilmaz et al. 2006] as "a dense field of displacement vectors which defines the translation of each pixel in a region. It is computed using the brightness constraint, which assumes brightness constancy of corresponding pixels in consecutive frames."

Additionally, edge detection methods are often used to track objects based on portions of their boundaries, of which the Canny Edge Detector algorithm is one of the most popular.

IV.2. Background Subtraction

Background subtraction is a class of algorithms that classify a given pixel as being members of either the foreground or background based on its evolution over time. This generally consists of creating a background model over a sequence of frames and then

mark pixels that break the model i.e. exhibit properties beyond some threshold.

One of the most commonly used techniques to do create a background model is to compare the median properties (e.g. intensity) of a pixel over a set of frames, the implicit assumption being that any given pixel will display the background for more than half the frames in the buffer [Cheung and Kamath].

To find foreground candidate pixels, the most commonly-used approach is to compare the difference between actual intensity and background model intensity for a given pixel and if this exceeds a (typically experimentally-derived) threshold, the pixel is a good candidate [Cheung and Kamath].

IV.3. Interest Point Detection

An interest point is a small area or pixel which has some expressive texture in its locality which can be used for tracking. Common techniques for classifying points as interesting are Moravec's detector, Harris detector, Scale Invariant Feature Transform, and Affine Invariant Point Detector [Yilmaz et al. 2006].

IV.4. Point Tracking

In point tracking, objects detected in consecutive frames are represented by points, and the association of the points is based on the previous object state which can include object position and motion. [Yilmaz et al. 2006].

Point tracking techniques can be categorized into deterministic and probabilistic methods. In deterministic point tracking, a set of points are detected in both of two consecutive frames. Any point detected in the earlier frame can theoretically correspond to any point detected in the latter frame and vice-versa, but only one correspondence mapping for all the points is correct. To determine which mapping will be used, the problem treated as an optimization problem with constraints such as a limit on how fast points can travel between consecutive frames, preferences for small changes in position, preferences for small changes in velocity (i.e. smooth motion among more than two frames e.g. uniform acceleration), preferences for similar velocities for points close to each other, etc. [Yilmaz et al. 2006].

In probablistic point tracking, sensor noise and random perturbations of the objects being tracked are addressed by using a temporal Bayesian technique called filtering which is used to predict a future state given the measured current and past states of a system. One very popular model is the Kalman filter which assumes a linear system with a Gaussian distribution of noise. If the noise does not follow a Gaussian distribution, a particle filter is often used [Yilmaz et al. 2006].

Another probablistic point tracking method is the Multiple Hypothesis Tracking (MHT) algorithm which, as the name suggests, maintains a list of possible correspondences for each object being tracked. Each hypothesized correspondence has a probability of correctness assigned which is refined as subsequent frames are processed [Yilmaz et al. 2006].

IV.5. Kernel Tracking

Kernel tracking is a region-based tracking approach and also comes in two flavors: template based and multi-view based tracking.

In template based kernel tracking, when tracking a single object, the most common approach is template matching, a brute force method of searching a given frame for a region similar to the search object template defined in the previous frame [Yilmaz et al. 2006]. A template can be thought of as a window that slides across the image and comparing itself to the portion of the image in which it covers to find a matching object. The comparison is performed using a similarity measure which typically takes into account intensity and color features.

In multi-view based tracking, multiple cameras are used to overcome the problem of tracking an object which appears to having (potentially wildly) different shapes depending on the angle at which it is viewed. For instance, a typical coffee mug appears as a circle with a rectangular handle protruding from one side when viewed from above, but then morphs into a cylinder with a semicircular handle as the mug is turned 90 degrees. Using Principal Component Analysis, a subspace representation of an object is built and tracking is performed by estimating the affine parameters iteratively until the difference between the input image and the projected image is minimized [Yilmaz et al. 2006].

IV.6. Silhouette Tracking

Silhouette tracking is also split into two categories: shape matching and contour tracking.

Shape matching is very similar to template matching in that it searches the current frame for an object boundary that closely matches a pre-specified shape. Sillhouette detection is usually performed via background subtraction. The problem with this approach is that it can faulter when searching for nonrigid objects or, as in the coffee mug example above, a shape morphs as an object or camera rotates [Yilmaz et al 2006].

Contour tracking attempts to solve the problem of tracking nonrigid objects whose shapes change as they rotate by iteratively morphing (i.e. evolving) a silhouette from that of an object in the current frame to that of the corresponding object in the next frame. Contour tracking uses two main approaches: the first is to use state space models to model the contour shape and motion, the second directly evolves the contour by minimizing the a contour energy functional [Yilmaz et al 2006].

When contour tracking using state space models, the tracked object's state is defined in terms of the shape and the motion parameters of the contour. The state is updated iteratively to probabilistically match around a newly observed boundary [Yilmaz et al 2006].

The second approach to contour tracking uses an energy functional computed from optical flow and/or other statistics observed in the image. This approach evolves the

contour by minimizing the energy functional which describes changes in observations between frames such as intensity [Yilmaz et al 2006].

IV.7. Support Vector Machines

A support vector machine is a mechanism used to classify a feature space into two different categories. Typically, input values from some training set are sent through some mapping function (from input space) to a feature space where the values are linearly separable and can therefore segregated into two classifications by some line or (hyper)plane. Then actual input values are mapped to feature space and classified by which side of the separating hyperplane they fall on. Support vector machines have been used to classify an object to be tracked as opposed to an object in the background [Yilmaz et al 2006].

V. Initial Approaches

During the course of this research and experimentation, a number of failed approaches were attempted before the final algorithm described in Section VI was achieved.

V.1. First Attempt at Vector Field Generation

In the early stages of experimentation, it was reasoned that if a vector field could be generated from affordances in local regions of the image, large regions which had similar vectors could be smoothed and treated as single segments. For example, consider the simple case of a 4x4 pixel image as seen in Figure V.1.

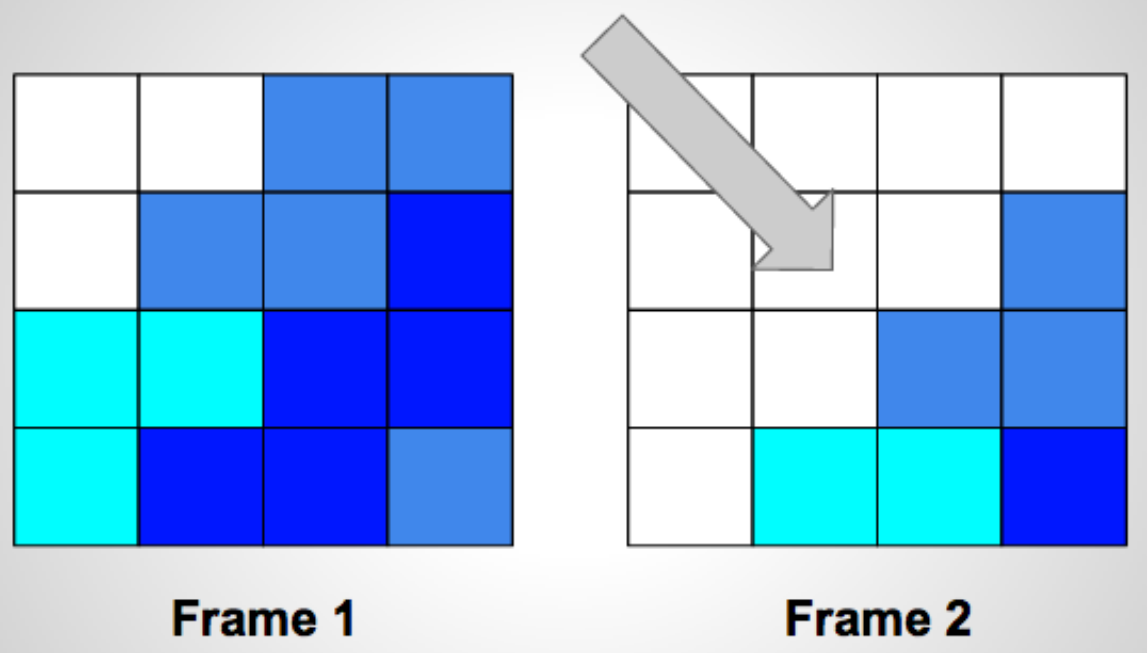

Figure V.1: A simple example of an object moving down and to the right over the course of two frames.

For each pixel in the current frame (Frame 2 in the figure above), its color would be compared to its (eight) neighboring pixels in the previous frame and the direction of motion would be then be indicated. For instance, if a pixel at a given location in the

current frame matched the color (within some threshold) of the pixel to the left in the previous frame, this would imply motion in the rightward direction. In the case of multiple vectors indicated for the same pixel, all the vectors for said pixel would be averaged. Iterating over all of the relevant pixels in this fashion would generate a vector field where, in theory and in this example, there might be local inaccuracies, but in global regions the outliers would be smoothed and average out.

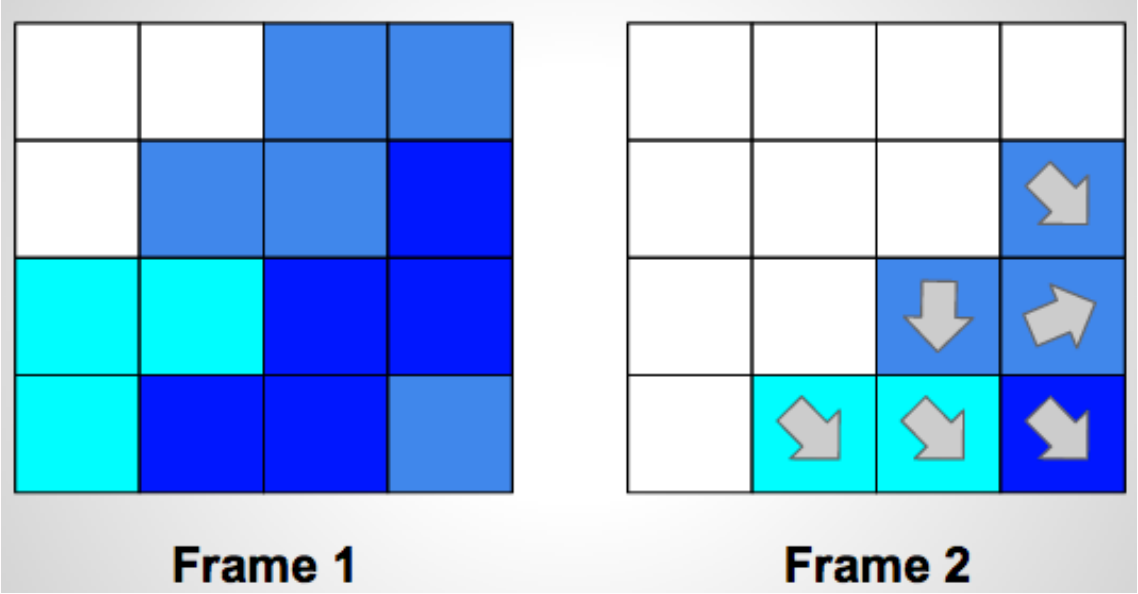

Figure V.2: A vector field generated from comparing pixels in the current frame to neighboring pixels in the previous frame.

Unfortunately, applying the method to real images didn't produce good results. The vector field was far too noisy. Whereas in the example above for the small region that was computed there was a general consensus from 2/3 of the pixels, in real images there was no clear consensus in the majority of regions in the field.

Figure V.3: A vector field generated with an image of an arm.

V.2. Second Attempt at Vector Field Generation

In another attempt at generating a vector field, the image was treated as a grid of cells where in each cell the "center of mass" of the pixels that make up the moving object is calculated. In theory, by measuring the change in position of the center of mass across consecutive frames for a given cell, this would indicate the net direction for part of the object that occupied that cell.

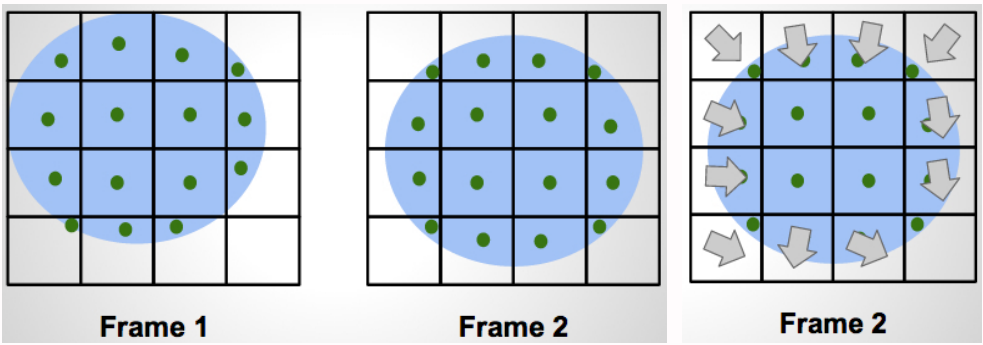

Figure V.4: A vector field generated from the change in center of mass for each cell across two consecutive frames.

This too produced inadequate results. The major problem was because of curved and irregularly-shaped borders of objects which may be moving in one direction, the vector generated for a given cell was only weakly correlated; it was more correlated to the number of pixels entering and exiting the cell as the object moved. Also, regions of the moving object that were convex in shape would show the inherent weakness in using the

center of mass calculation.

Figure V.5: The calculated centers of mass (shown as green squares) in a real image.

V.3. Fast Optical Flow Estimation

As I was having little success in generating an accurate vector field, I approached the problem from a different angle: what was needed was direction selectivity, the characteristic of an algorithm which produces an output depending on the direction the object is moving in. Thus instead of trying to assign a direction, it might be beneficial to have an algorithm react to a specific direction. Based on the fast optical flow algorithm presented in [Catalano 2009], I wrote an algorithm which behaves as follows. If a given pixel at coordinate (x, y) belongs to the moving object in the current frame, but the pixel at the same coordinate in the previous frame does not cover the moving object, then the directions indicated by the neighboring pixels that belong to the moving object in the current frame are averaged and the resulting vector is associated with that pixel. If the converse is true i.e. the pixel at (x, y) belongs to the moving object in the previous frame, but not in the current frame, the vector is calculated the exact same way, but flipped 180 degrees.

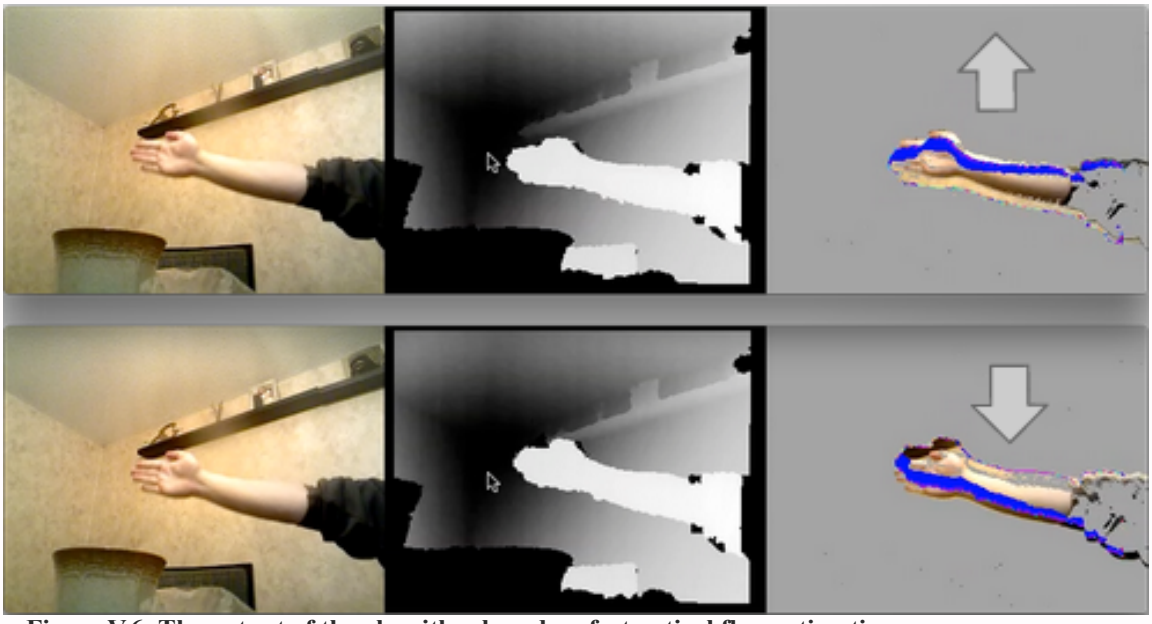

Figure V.6: The output of the algorithm based on fast optical flow estimation as an arm moves up (top image) and down (bottom image).

The result was not direction selectivity per se, but it was promising in that the leading edge of the object relative to its direction of motion was clearly identified no matter which direction the object was moving in.

VI. Final Algorithm

Given a video sequence of a moving object, the tasks that needed to be completed for each frame were as follows:

1. If an object is moving in the frame, identify all the pixels that make up the object.

2. Segment the moving object into regions where each region is moving in the same direction.

A background subtraction algorithm is used: a comparison is made between each pixel in the RGB output and the pixel at the same location in the previous frame. If the colors do not match within some threshold then we know that the pixel being examined part of the moving object in one of the two frames and it is considered "significant".

```
function colorDifference(color1, color2)
{
  brightness1 = ((color1.red * 299) + (color1.green * 587) +
                (color1.blue * 114)) / 1000;
  brightness2 = ((color2.red * 299) + (color2.green * 587) +
                (color2.blue * 114)) / 1000;

  return Math.abs(brightness1 - brightness2);
}

function backgroundSubtraction()
{
  for each pixelPosition in currentFrameData
  {
    if(colorDifference(previousFrameData[pixelPosition],
        currentFrameData[pixelPosition]) > BGS_THRESHOLD)
    {
      currentFrameData[pixelPosition] = INSIGNIFICANT_PIXEL_COLOR;
    }
  }
}
```

Figure VI.1: The background subtraction algorithm.

The background subtracted image is scanned to find clusters of significant pixels of a

given minimum size (so as to disregard noise). As this is done, a tracking square is created to denote the smallest area of the whole image where these clusters (and thus the moving object) is contained.

Figure VI.2: From left to right, the RGB image, the depth map (lighter pixels are closer to the sensor), and the background subtracted image with the tracking square.

```
for each pixelPosition in currentFrameData
{
  if(currentFrameData[pixelPosition] != INSIGNIFICANT_PIXEL_COLOR)
  {
    significantPixelCount = 0

    for each neighborPosition within radius R of pixelPosition
    {
      if(currentFrameData[neighborPosition] != INSIGNIFICANT_PIXEL_COLOR)
      {
        significantPixelCount++;
      }
    }

    if(significantPixelCount >= TS_THRESHOLD)
    {
      if(currentFrame.trackingSquare == null)
      {
        currentFrame.trackingSquare = new TrackingSquare(origin = pixelPosition,
                                          width = 1, height = 1)
      }
      else
      {
        currentFrame.trackingSquare.expandToInclude(pixelPosition);
      }
    }
  }
}
```

Figure VI.3: The algorithm for generating the tracking square.

Within this tracking square, the significant pixel that has the minimum reading from the depth map (i.e. the pixel closest to the depth sensor) is then selected as a starting point

for a paint fill algorithm. The paint fill expands in all directions from this point to cover the pixels which are part of the same object as the starting point. It does this by testing adjacent pixels to see if there is a large change in depth; as long as the depth changes within some threshold in any given direction, the adjacent pixel in the direction will be considered as part of the same object. If there is a large depth increase or decrease (e.g. as you would expect on the edge of a foreground object against a background), this is treated as a border to the object. This creates the tracking mask, the set of pixels that make up the moving object.

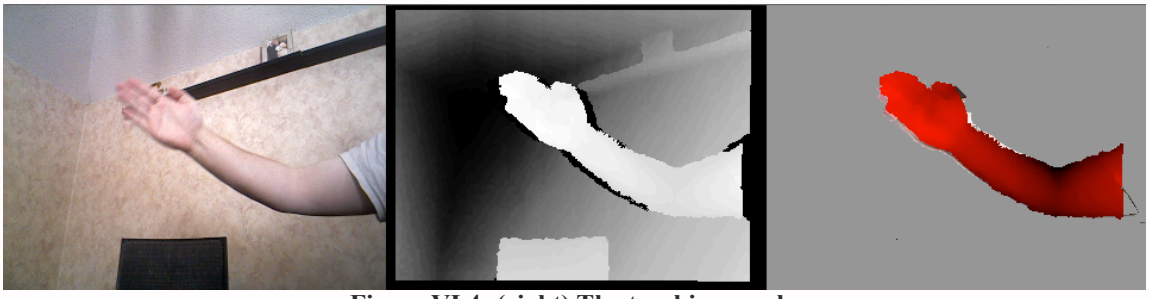

Figure VI.4: (right) The tracking mask.

```
// Determines if a given pixel that was not subtracted during the
// background subtraction algorithm 1. is not noise and
// 2. is part of a region of similar pixels that has a certain
// minimum area belonging to the same physical object.
function isInSignificantPixelCluster(pixelPosition)
{
  if(currentFrameData[pixelPosition] != INSIGNIFICANT_PIXEL_COLOR)
  {
    significantPixelCount = 0;
    referenceDepth = currentFrameDepthMap[pixelPosition];

    for each neighborPosition within radius R of pixelPosition
    {
      neighborDepth = currentFrameDepthMap[neighborPosition];

      if(currentFrameData[neighborPosition] != INSIGNIFICANT_PIXEL_COLOR &&
          Math.abs(referenceDepth - neighborDepth) <= SPCD_THRESHOLD)
      {
        significantPixelCount++;
      }

      return (significantPixelCount >= SIGNIFICANT_PIXEL_CLUSTER_MIN_COUNT);
    }
  }

  return false;
}

function chooseStartingPixelPosition()
{
  trackingSquare = currentFrame.trackingSquare;
  closestPixelPosition = trackingSquare.center
  closestPixelDepth = 99999999;

  for each pixelPosition in trackingSquare
  {
    if(currentFrameData[pixelPosition] != INSIGNIFICANT_PIXEL_COLOR &&
      depthMapData[pixelPosition] < closestPixelDepth &&
      isInSignificantPixelCluster(pixelPosition))
    {
      closestPixelPosition = pixelPosition;
      closestPixelDepth = depthMapData[pixelPosition];
    }
  }

  return closestPixelPosition;
}
```

```
function generateTrackingMask()
{
  startingPixelPosition = chooseStartingPixelPosition();

  trackingMaskPixels = new Array();
  trackingMaskPixels.add(startingPixelPosition);

  // Do a paint fill algorithm
  foreach currentPixelPosition in trackingMaskPixels
  {
    int currentPixelDepth = depthData[currentPixelPosition];

    // array of the directions of pixel index neighbor traversal
    // (i.e. left, right, up, down)
    directionIncrements = {-1, 1, -IMAGE_WIDTH, IMAGE_WIDTH};

    for each directionIncrement in directionIncrements
    {
      neighborPixelPosition = currentPixelPosition + directionIncrement;
      previousPixelDepth = currentPixelDepth;

      while(true)
      {
        if(neighborPixelPosition < 0 ||
          neighborPixelPosition >= IMAGE_WIDTH * IMAGE_HEIGHT)
        {
          break;
        }

        neighborPixelDepth = currentFrameDepthMap[neighborPixelPosition];

        if(!trackingMaskPixels.containsKey(neighborPixelPosition) &&
           Math.abs(previousPixelDepth - neighborPixelDepth)
           < SAME_SURFACE_DEPTH_THRESHOLD)
        {
          trackingMaskPixels.add(neighborPixelPosition);
          previousPixelDepth = neighborPixelDepth;
          neighborPixelPosition += directionIncrement;
        }
        else
        {
          break;
        }
      }
    }
  }

  return trackingMaskPixels;
}
```

Figure VI.5: The algorithm for generating the tracking mask.

Each pixel in this tracking mask is sent as input to a set of Reichardt detectors which assign a direction of movement to edge pixels. A Reichardt detector is a simple circuit

found in the human retina that reads the signal from a photoreceptor (e.g. a pixel in our case) and multiplies the result with the signal from a neighboring (spatially) signal. The two signals are then subtracted which results in a circuit which responds to sensed movement in a given direction (the direction of the subtraction) [Borst 2011].

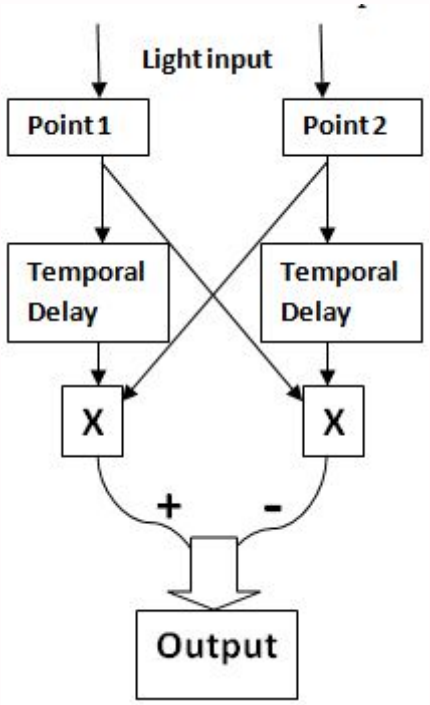

Figure VI.6: A Reichardt detector (source: http://en.wikipedia.org/wiki/File:Reichardt_model.jpg)

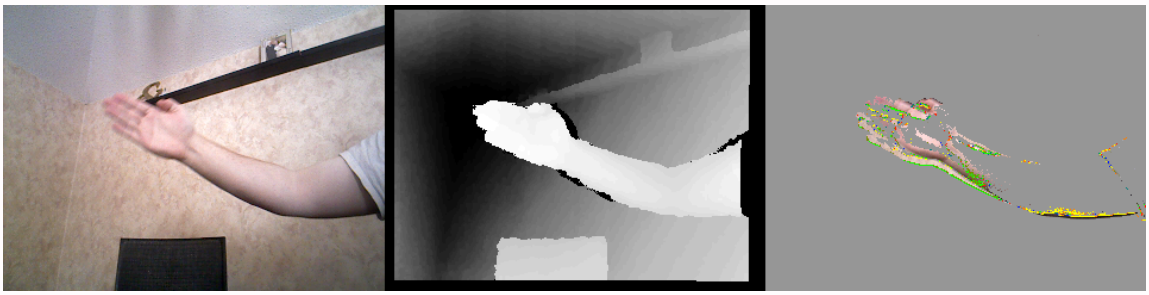

Figure VI.7: (right) Edges are assigned color-coded directions of motion.

```
function reichardtDetection()
{
  for each pixelPosition in currentFrame.trackingMask
  {
    vectors = new Array();

    for each neighborPixelPosition within radius = 1 of pixelPosition
    {
      if(currentFrameData[pixelPosition] != INSIGNIFICANT_PIXEL_COLOR &&
        previousFrameData[neighborPixelPosition] != INSIGNIFICANT_PIXEL_COLOR &&
        colorDifference(currentFrameData[pixelPosition],
           previousFrameData[neighborPixelPosition]) < COLOR_DIFFERENCE_THRESHOLD))
      {
        vectors.add(getNeighborVector(pixelPosition, neighborPixelPosition));
      }
    }
  }

  currentFrame.setVectorColor(Math.roundToNearest45Degrees(average(vectors)));
}
```

Figure VI.8: The algorithm for the Reichardt Detector.

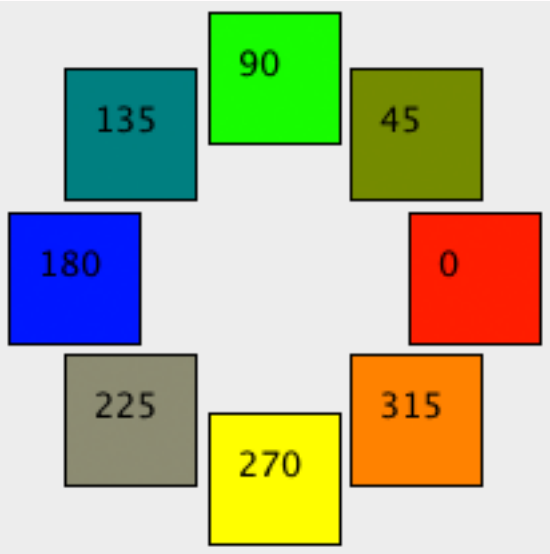

Figure VI.9: The colors associated with the directions of motion, rounded to the nearest 45 degrees.

With the boundary and texture edges assigned to angles of motion, each non-edge pixel is traversed and assigned the vector of the nearest edge pixel within some radius (regardless of whether the edge is part of the texture or part of the object boundary).

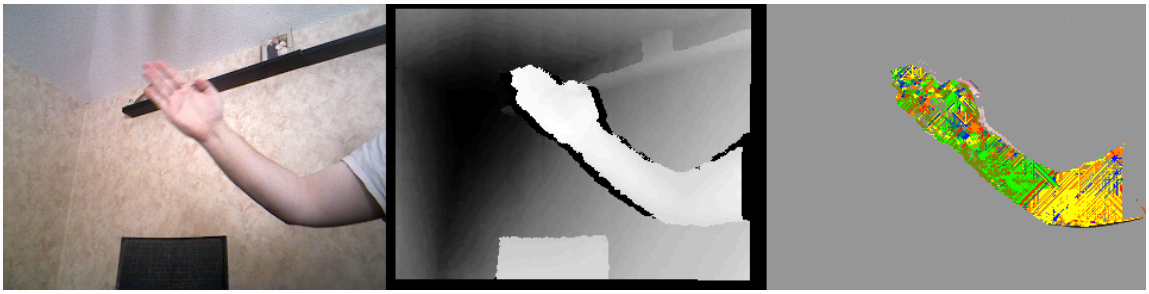

Figure VI.10: (right) The non-edge pixels adopt the the motion vector (and color) of the nearest edge pixel within a given radius.

```
function assignInsignificantPixelsToClosestEdgePixel()
{
  coloredEdgePixels = currentFrame.coloredTrackingMaskPixels;

  for each pixelPosition in currentFrame.trackingMask
  {
    if(!coloredEdgePixels.contains(pixelPosition))
    {
      // span out in 8 directions to find the closest member of coloredEdgePixels

      boolean found = false;

      for(int radius = 1; !found && radius < 100; radius++)
      {
        for each direction
        {
          distantPixelPosition = pixelPosition + direction * radius;

          if(!found && coloredEdgePixels.contains(distantPixelPosition))
          {
            currentFrame.coloredTrackingMaskPixels[pixelPosition] =
               coloredEdgePixels[distantPixelPosition];

            found = true;
          }
        }
      }
    }
  }
}
```

Figure VI.11: The algorithm for assigning vectors to non-edge pixels.

When we are restricted to assigning vectors to pixels based only on the nearest edge pixel that is on the boundary of the moving object within some radius, ignoring edge pixels that are part of the texture, some pixels are not assigned any vector.

Figure VI.12: (right) The non-edge pixels adopt the the motion vector (and color) of the nearest edge pixel which is on the object boundary within a given radius.

Finally, the motion vectors for the tracking mask pixels are smoothed so the object is segmented into more homogeneous regions. This is done by traversing all the pixels in the tracking mask and for each pixel, a survey of its neighbors is conducted and if at least Sp = 75% of the neighbors within a radius of Sr = 4 pixels have the same associated vector, the current pixel then adopts that same vector. This process repeats until there are no pixels whose motion vectors change.

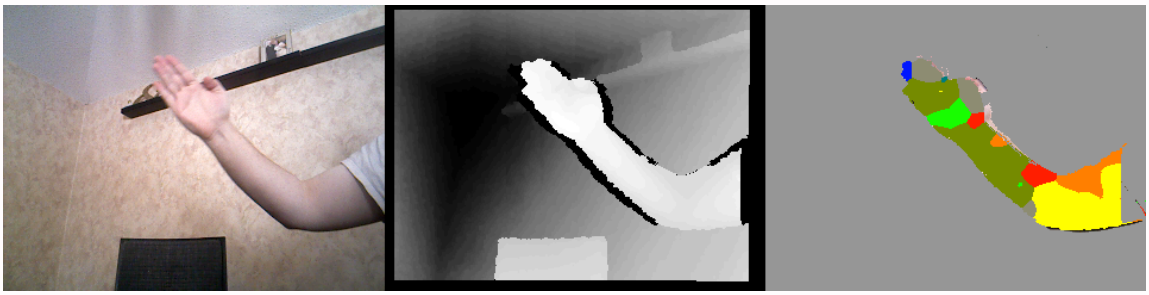

Figure VI.13: (right) The "any edge" algorithm (see Figure V.6) after smoothing.

```
function smoothSegments()
{
  changedPixels = new Array();
  trackingMaskPixels = currentFrame.trackingMaskPixels;
  newPixelColors = new Array();
  pixelColors = currentFrame.pixelColors;
  radius = 4; // this is Sr
  neighborPollThreshold = 0.75; // this is Sp

  while(true)
  {
    colorChange = false;

    // recolor all pixels that are at least 75% surrounded by
    // other pixels of a different color
    for each pixelPosition in currentFrame.pixelColors
    {
      // these are the pixels to change IF a color (i.e. vector)
      // change is necessary
      pixelsToChange = new Array();
      pixelsToChange.add(pixelPosition);

      if(!changedPixels.contains(pixelPosition))
      {
        colorCounts = new HashMap();

        for each neighborPixelPosition within radius distance from pixelPosition
        {
          if(pixelColors.contains(neighborPixelPosition))
          {
            colorCounts[pixelColors[neighborPixelPosition]]++;
          }
          else if(trackingMaskPixels.contains(neighborPixelPosition))
          {
            // add the neighbor pixel to pixels to change because it's in the
            // tracking mask, but it has no color (i.e. vector) yet
            pixelsToChange.add(neighborPixelPosition);
          }
        }
```

```
        // now tally up all the neighboring colors

        colorCountSum = 0;
        highestColorCount = 0;

        // default behavior: leave the color as is
        newColor = pixelColors[pixelPosition];

        // find the color that is most abundant around pixelPosition
        for each color in colorCounts
        {
          colorCount = colorCounts[color];

          if(colorCount > highestColorCount)
          {
            highestColorCount = colorCount;
            newColor = color;
          }

          colorCountSum += colorCount;
        }

        for each pixelToChange in pixelsToChange
        {
          // if we're over the threshold to actually change the color
          if(highestColorCount / colorCountSum > neighborPollThreshold)
          {
            newPixelColors[pixelToChange] = newColor;
            changedPixel.add(pixelToChange);
            colorChange = true;
          }
        }
      }
    }

    currentFrame.saveNewPixelColors(newPixelColors);

    if(!colorChange)
    {
      return;
    }
  }
}
```

Figure VI.14: The smoothing algorithm for generating the final segments.

Figure VI.15: (right) The "object boundary edge only" algorithm (see Figure V.7) after smoothing.

VII. Results and Analysis

Testing the program relied on the following constraints:

1. Only one moving object is in the scene at any given time.

2. The moving object is not occluded in any way, but it doesn't have to be the object closest to the sensor.

3. The moving object isn't near enough to another object or surface in the scene that the program mistakes the combination as a single object.

4. The object is moving relatively slowly.

Experiments were conducted with different movements of a human hand and arm and the resulting motion segments were evaluated for accuracy against actual motion. The final algorithm as described in Section VI provided the most accurate model that identifies the motion of the object.

Having said that, the accuracy of any algorithm that attempts to segment the tracking mask based on the 8 cardinal and ordinal directions of movement will be subjective; the number, size, and shape of the segments are an inherently fuzzy proposition. In addition, the rules that may be used for rounding and the influence of neighboring pixels in reassigning motion vectors generate several "good" models for the moving object. Because of these factors, the problem is ill-posed.

Changing the smoothing parameters (Sp = the percentage of neighboring vectors required for a pixel to adopt said vector; and Sr = the radius of neighbor pixels to consider) will give different results.

The number of segments is inversely proportional to Sr. This is because as Sr increases, larger areas must become homogeneous for the halting condition of the smoothing function to be reached. Also as Sr increases, more passes through the smoothing function are needed before an equilibrium is reached and no pixels are reassigned neighboring vectors and performance slows.

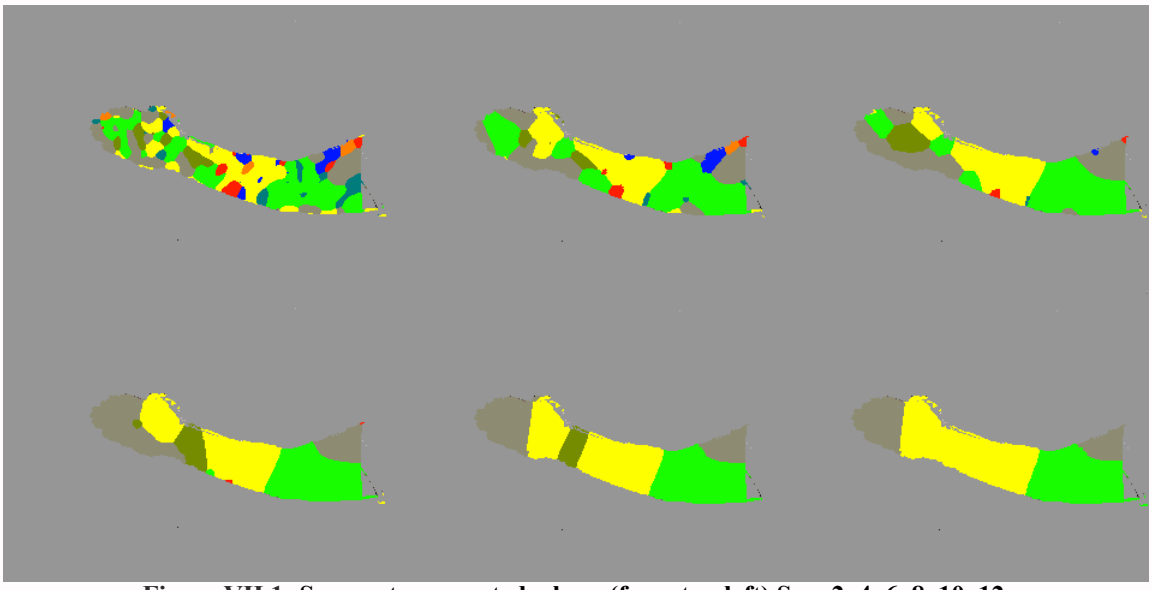

Figure VII.1: Segments generated where (from top left) Sr = 2, 4, 6, 8, 10, 12.

As Sp increases, the larger segments get larger and the smaller segments get smaller. This is because a smaller segment will have proportionally fewer pixels where the percentage of neighboring pixels that share the same motion vector is greater than Sp. If we think of each pixel's neighbors as casting a vote to elect the best motion vector, the larger segments -- specifically the segments with the most border area that has a width greater than Sr -- will have more pixels voting for their current associated motion vector thus expanding the segment outward. Conversely, the smaller segments will adopt more

neighboring motion vectors and their borders will shrink.

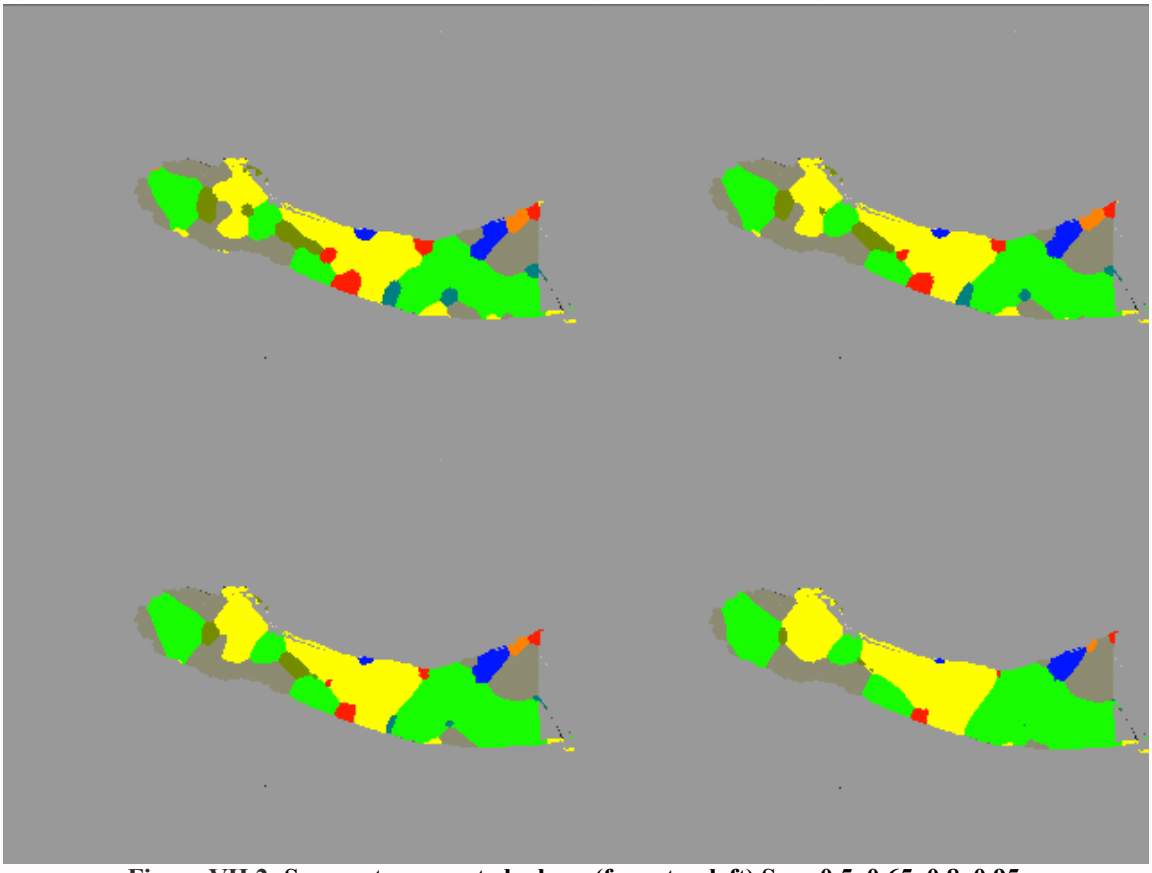

Figure VII.2: Segments generated where (from top left) Sp = 0.5, 0.65, 0.8, 0.95.

Figure VII.3 shows nine consecutive frames of a fast-moving arm bending.

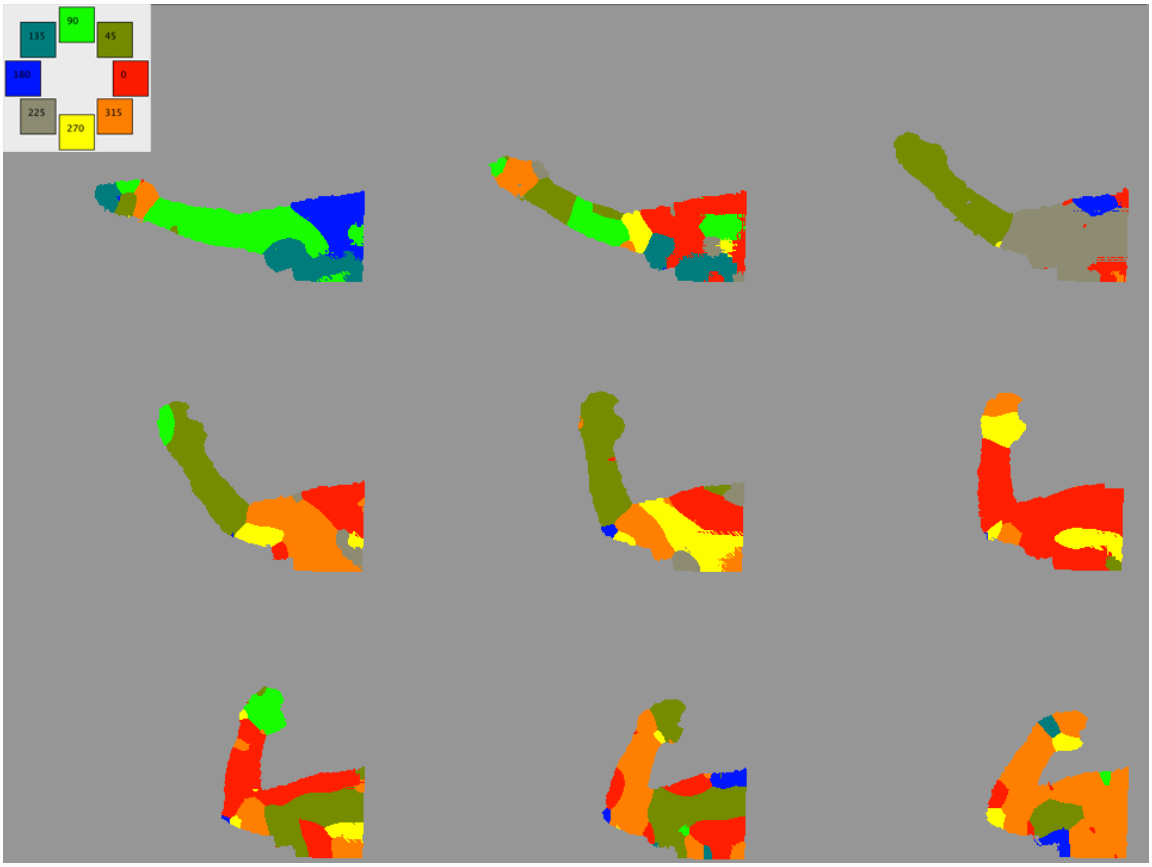

Figure VII.3: Segments generated for a fast-moving arm bending (Sr = 4, Sp = 0.75).

Figure VII.4 shows fifteen representative frames (some insignificant frames were omitted for clarity) of a rigid cardboard tube lowering then raising along an arc. Note that the segmentation becomes more nuanced as the speed slows just before and just after the change in direction.

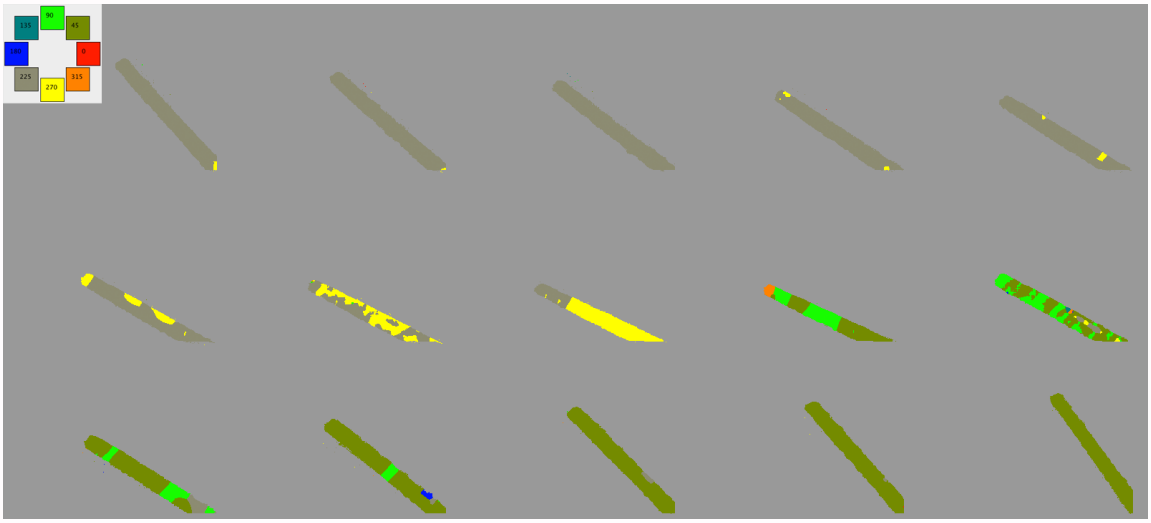

Figure VII.4: Segments generated for a cardboard tube lowering then raising (Sr = 4, Sp = 0.75).

This implies an effect that was also observed in other experiments: the simplicity of the segmentation decreases when the object is moving slower than some lower bound. This is most likely because during the Reichardt detection phase of the algorithm, when the object is moving slowly, a significant number of the edge pixels do not move far enough between two consecutive frames for the color difference threshold to be met which is used to detect motion. Therefore in these cases edges are less well defined and subsequently more pixels adopt their neighbors' motion vectors. The overall consequence is increased fuzziness.

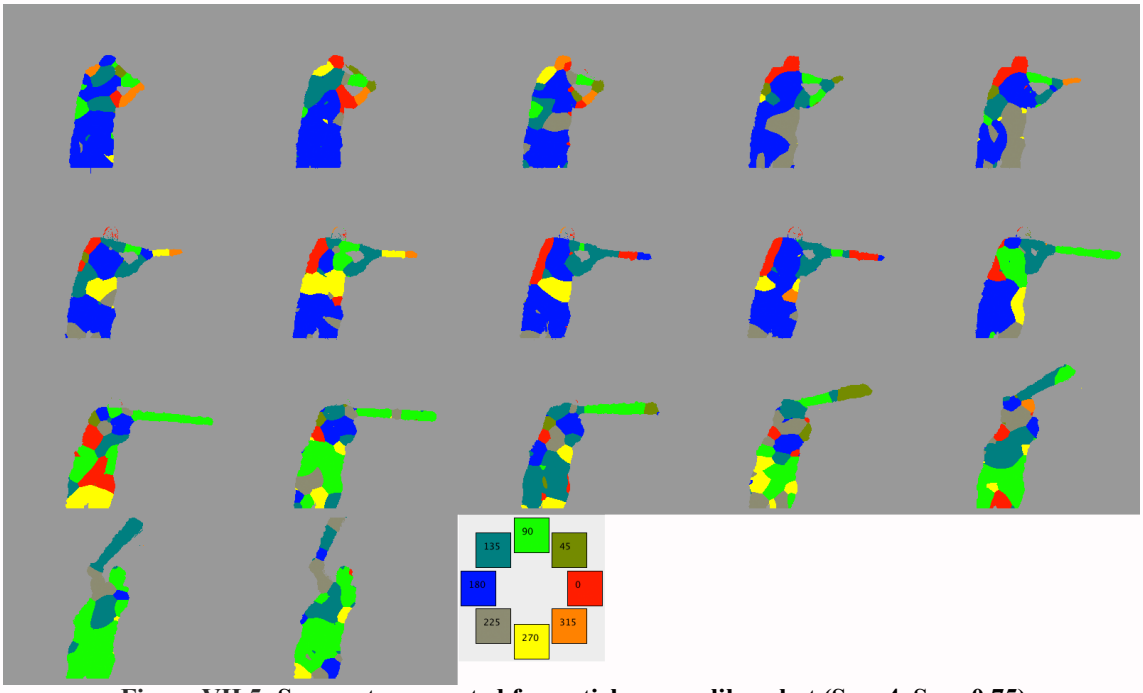

Figure VII.5: Segments generated for a stick swung like a bat (Sr = 4, Sp = 0.75).

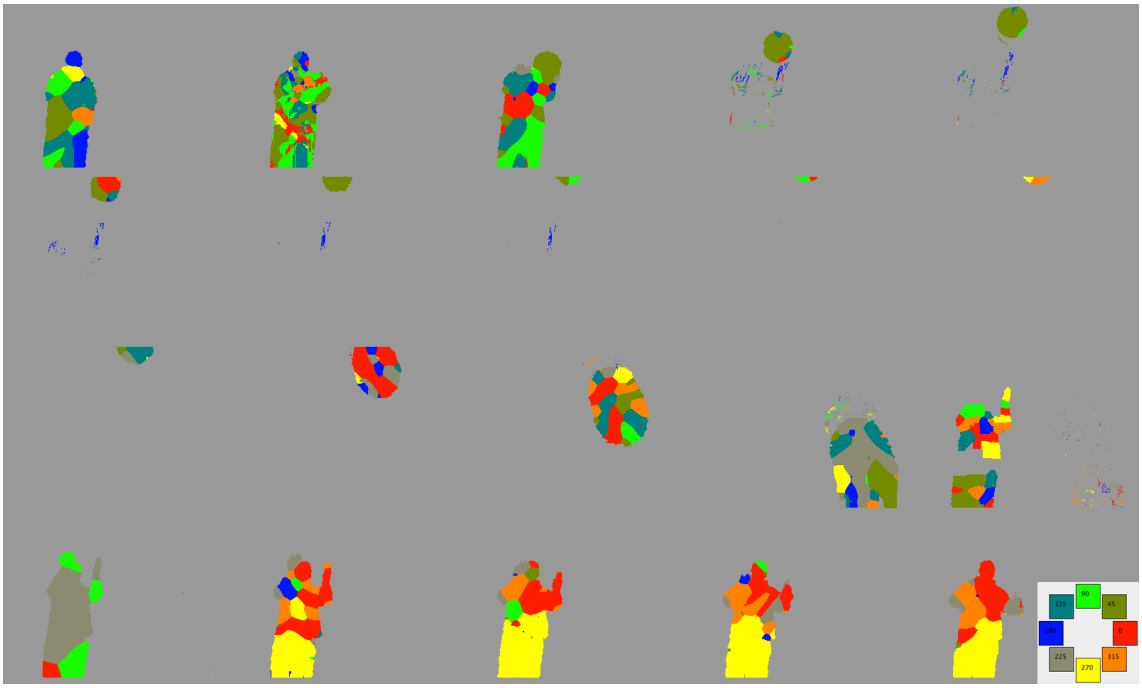

Figure VII.6: Segments generated for a ball thrown near the camera (Sr = 4, Sp = 0.75).

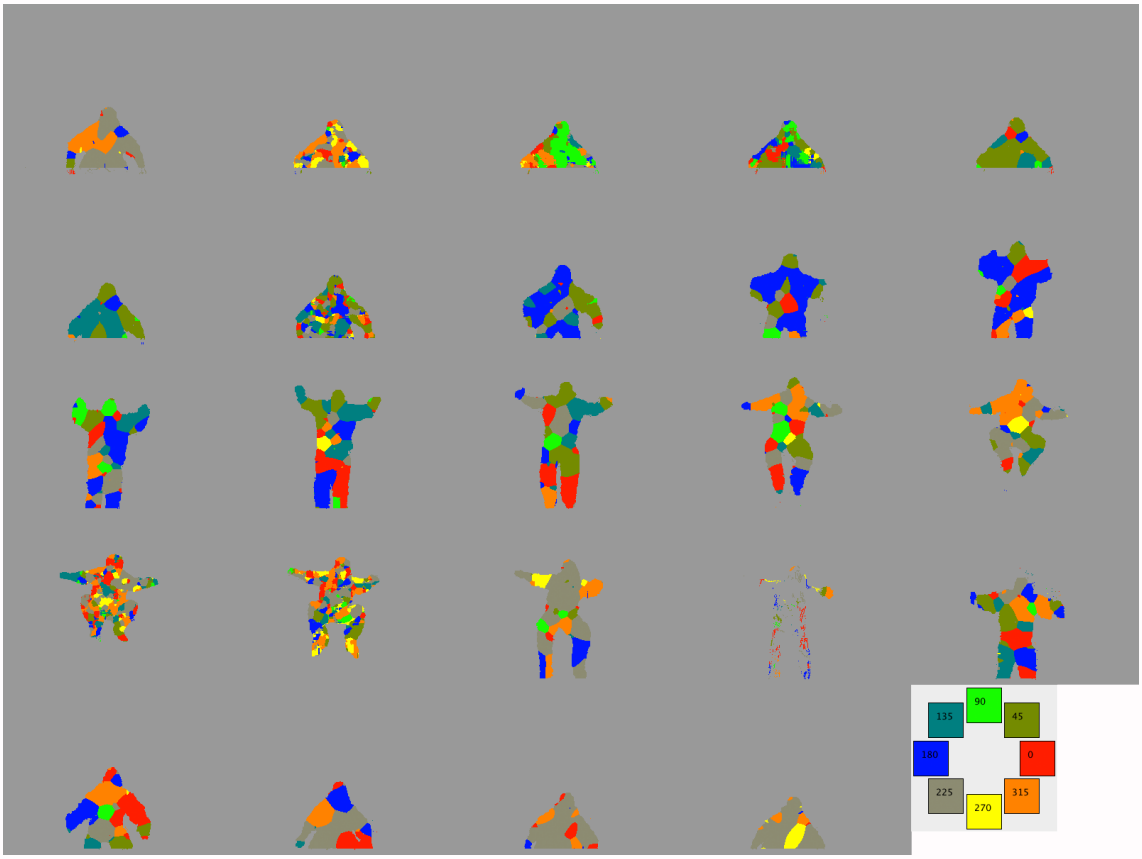

Figure VII.7: Segments generated for a person jumping vertically (Sr = 4, Sp = 0.75).

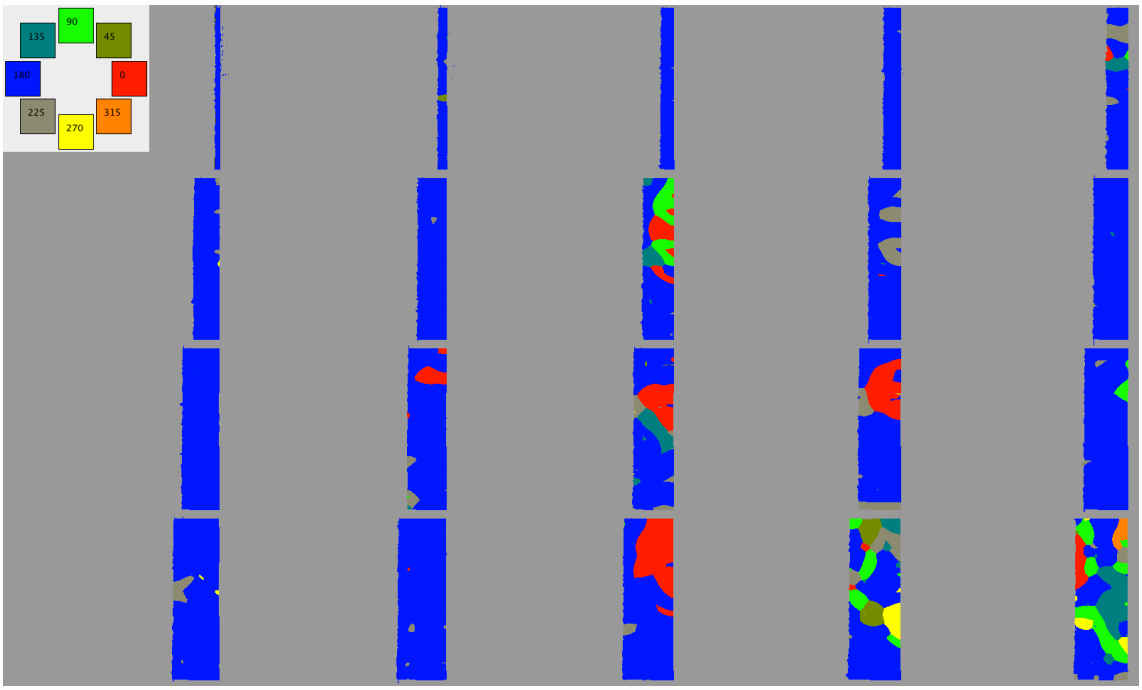

Figure VII.8: Segments generated for a door swinging open toward the camera (Sr = 4, Sp = 0.75).

Figures VII.5 - VII.8 show further examples. Again we see, most prominently in the jump video, slow moving objects being "over-segmented." With fast moving objects

such as the stick being swung and the falling ball, an effect can be observed where the combined region the object occupies in the two frames involved in the background subtraction are used. This is a limitation of the algorithm and appears to the be primarily the result of a delay between capturing the RGB input and the depth sensor input, but also there is blurring in the RGB image.

VIII. Summary

In this research, I addressed the problem of profiling the movement of an object from a video. My approach combined a conventional background subtraction algorithm with depth sensor input to perfectly isolate the moving object within the constraints of the experiments. Then an artificial Reichardt detector, a neural circuit that in the human visual system that selectively detects motion in a given direction, is used to assign directions of motion to the object. With some further processing and smoothing, the software produces a reasonably accurate segmented profile of motion for the object being observed.

This research contributes to the science of computer vision by successfully demonstrating a method that combines advances in consumer-grade vision hardware with biologically-inspired techniques to generate a vector field for a moving object in a video.

IX. Directions for Future Research

This research can be further expanded in several ways. First, one could work towards eliminating some of the constraints by handling occlusions, multiple simultaneous moving objects, a moving camera, etc. The major difficulty with any of these tasks would be finding the appropriate starting point(s) for the paint fill algorithm that creates the tracking mask. In my approach, the pixel that is closest to the depth sensor is the starting point, but a new scheme would have to be created in order to determine potentially multiple points, any of which may not be the closest to the sensor in the tracking square (e.g. in the case of an occlusion).

In keeping the constraints, the next logical step would be to address the correspondence problem wherein a segment in one frame must be associated with a segment in the next consecutive frame in order to track the path of that segment in time. In other words, a given area of the physical object must be recognized as the same area of the same object in the next frame. The major challenge in this is that some objects appear amorphous and of differing sizes across time; the arm examples above clearly illustrate this. As a consequence, segments are also change in size and shape across frames. Plus, in the algorithm presented here, the number of segments can change so a segment could completely disappear between one frame and the next. However, the correspondence problem could be tackled considering only the object taken as a whole.

In any case, there is a lot of room for experimenting with the current algorithm. One could attempt to generate a predefined constant number of segments for an object. This could work well, especially if employed strictly for a single domain such as only tracking humans or only cars.

References

J. K. Aggarwal and Y. F. Wang, "Analysis of a Sequence of Images Using Point and Line Correspondences", Proceedings of *IEEE International Conference on Robotics and Automation*, 1987, pp. 1275-1280

Dr. Alexander Borst, "In Search of the Holy Grail of Fly Motion Vision (2011)," http://videocast.nih.gov/summary.asp?Live=10065 (as of March 5, 2012)

Giuseppe Catalano et al., "Optical Flow," http://www.cvmt.dk/education/teaching/f09/VGIS8/AIP/opticalFlow.pdf (as of May 1, 2013)

Sen-Ching S. Cheung and Chandrika Kamath, "Robust techniques for background subtraction in urban traffic video," http://www.vis.uky.edu/~cheung/doc/UCRL-CONF-200706.pdf (as of May 29, 2011)

D.G. Falconer, "Target Tracking with the Hough Transform," *Circuits, Systems and Computers, 1977 Conference Record*, 1977 pp. 249-252.

Russel D. Hamer, "What Can my Baby See?" *Parents' Press* Vol. XI, No. II, 1990.http://www.ski.org/Vision/babyvision.html (as of July 3, 2011).

Jeff Hawkins and Dileep George, "Hierarchical Temporal Memory Concepts, Theory, and Terminology" 2007. http://www.numenta.com/htm-overview/education/Numenta_HTM_Concepts.pdf (as of February 12, 2012).

Izadi, Shahram et al., "KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera," *IEEE ISMAR*, IEEE, October 2011

Kloihofer, W. and Kampel, M., "Interest Point Based Tracking," *Pattern Recognition (ICPR), 2010 20th International Conference on*, vol., no., pp.3549-3552, 23-26 Aug. 2010

Newcombe, Richard A. et al., "KinectFusion: Real-Time Dense Surface Mapping and Tracking," ACM Symposium on User Interface Software and Technology, October 2011

Massimo Piccardi, "Background subtraction techniques: a review," http://profs.sci.univr.it/~cristanm/teaching/sar_files/lezione4/Piccardi.pdf (as of May 29, 2011).

Stuart Russell and Peter Norvig, "Artificial Intelligence: A Modern Approach", 3rd ed., Prentice-Hall. 2010.

P. Smith and G. Buechler, "A Branching Algorithm for Discriminating and Tracking Multiple Objects", *IEEE Transactions on Automatic Control*. volume 20, issue 1, 1975, pp. 101-104

William B. Thompson and Ting-Chuen Pong, “Detecting Moving Objects,” *International Journal of Computer Vision* Issue 4, 1990, pp. 39-57.

A. Yilmaz, O. Javed, and M. Shah, “Object tracking: A survey,” *ACM Computing Surveys,* vol. 38, no. 4, pp. 1–45, 2006.

Zhengyou Zhang and Olivier D. Faugeras, "Three-dimensional motion computation and
object segmentation in a long sequence of stereo frames," *International Journal of Computer Vision,* vol. 7, issue 3, pp. 211-241, 1992.