Development of a Software-based Real-time Computer Vision System for Autonomous Terrestrial Navigation

Zach Rhoads & Dean Wyatte

April 2006
Abstract

This document describes a real-time computer vision system capable of tracking two-dimensional objects within a scene. The system is intended for use in an autonomous terrestrial navigation system. We discuss the requirements, design, and implementation of such a system. On a Pentium 4 1.6 GHz PC with 512 MB RAM running Windows 2000, the system can reliably track features in a scene at 14 Hz.
### Table of Contents

1. Introduction .................................................. 1
2. Requirements .................................................. 1
3. Design ......................................................... 3
4. Implementation ............................................... 6
   4.1 StartCamera .............................................. 7
   4.2 CaptureGrayscale ........................................ 9
   4.3 OptimalThreshold ........................................ 10
   4.4 SelTargetFeature ......................................... 12
   4.5 Features ................................................. 12
      4.5.1 MakeEdgemap ....................................... 13
      4.5.2 BitmapConnect ..................................... 13
      4.5.3 FPointCentroid .................................... 15
      4.5.4 FPointInerTensor .................................. 15
      4.5.5 FindTargetFeature .................................. 16
5. Conclusions .................................................. 17
1. Introduction

The purpose of this project is to construct a real-time computer vision system. The potential applications for such a system are numerous, but its main utility lies in autonomous navigation. Previous efforts have been made in developing real-time computer vision systems for use in autonomous navigation. Of particular interest is the Skeyeball system\textsuperscript{1}.

The Skeyeball system consists of a radio-controlled airplane equipped with a camera, which is used to track objects in real-time. The system is largely implemented in hardware due to the necessity for a lightweight, yet efficient system.

Our system is based largely on the Skeyeball system, but makes several key changes. First, whereas the Skeyeball system was designed for aerial navigation, our system is designed for terrestrial navigation. Therefore, we are not limited by weight constraints and are able to implement our system in software. Being able to implement the system in software grants us more processing power, allowing for a more robust image thresholding algorithm. Robust image thresholding algorithms are of particular interest in a terrestrial system due to contrast concerns created by uneven lighting and shadows. Finally, we will not be implementing any sort of control scheme. However, the information necessary to do so is output by and is easily accessible within our system.

2. Requirements

The development of a real-time computer vision system has several inherent requirements. First and most importantly is the notion of real-time computing. A real-time system must be efficient enough to be able to process inputs and compute a
meaningful output reaction within an imposed time frame. In this case, the vision system must do the following within the time frame:

1. Capture the current scene.
2. Extract features from the scene.
3. Locate and track the target feature(s) in the scene.
4. Output the location of each feature.

Because the system is to be used in autonomous navigation, the time frame that this loop must be completed within should be realistic enough to enable the vehicle to react properly to obstacles. The suggested time frame for completing such a loop is at least 1-10 Hz. The operating frequency of this event loop can be influenced by a multitude of factors. One such factor is the composition of the scene to be captured.

Scene composition is a more influential factor in terrestrial systems than in aerial systems like our reference system. The scenes captured by a terrestrial system can be significantly more complicated than the scenes captured by an aerial system. An aerial system captures scenes from an optimal perspective. The system always has a clear line of sight to its target. This lack of obstruction makes for uniform lighting conditions, causing the distinction between features and background to be readily apparent. Conversely, a terrestrial system is generally restricted to a less-than-optimal perspective. The line of sight between the system and its target is not necessarily clear. Obstacles may obstruct the systems view of the target. Additionally, there is no guarantee that the lighting will be uniform within a scene when on the ground. Structures such as mountains and canopy can partially or completely block light, rendering shadows across important features of objects. In extreme cases, this can cause certain features to be partially or
completely undetectable to the system.

Due to the inherent lighting issues in terrestrial computer vision systems, we utilize an image thresholding algorithm that is robust enough to produce meaningful results in all situations. Therefore, we require significantly higher processing power than our reference system. Aerial systems, like our reference system, are bounded by space and weight constraints and therefore are forced to use low-powered microprocessors and creative algorithms. Fortunately, terrestrial systems are not limited by any of these concerns, and a heavier and more powerful setup can be used alongside more computationally powerful algorithms.

3. Design

One of the most fundamental tasks of a computer vision system is to capture visual information from the real world. To accomplish this, we use a Dragonfly series digital camera manufactured by Point Grey Research. Point Grey Research was chosen as a supplier because they include a robust API for their cameras. The camera connects to the host PC via FireWire and can be interfaced in C/C++.

Before being used for capturing images, the camera must be initialized. After initialization, the camera can be interfaced in our main event loop, which is as follows:

1. Retrieve the most recent image from the camera’s buffer.
2. Down-sample and convert the image to grayscale.
3. Threshold the image.
4. Detect the edges of all prominent features.
5. Group the edges into coherent feature(s).
6. Classify each unique feature.

7. Locate and track the target feature(s).

8. Output the location of each feature.

As soon as the camera is initialized, it begins filling its buffer with captured images. When presented with an image request, the camera returns the most recent image from its buffer.

The resulting image is 640 x 480 pixels. Processing an image of this size would hardly be feasible in a real-time system, so the image is down-sampled to 200 x 200 pixels. To further simplify things, all color information is removed and the image is converted to 8-bit grayscale values. After simplification the system begins the incremental process of feature extraction. At each stage of feature extraction the image is further reduced in complexity until we are left with the coordinates of the center of mass for each unique feature. The first stage of feature extraction is image thresholding.

Image thresholding, in its simplest incarnation, will produce a binary image representing the segmentation of features and background by a single grayscale value. To calculate this value, a histogram is computed. The histogram corresponds to a bi-modal distribution with the smaller mean representing the features in the image and the larger mean representing the background in the image. This segmentation value will typically be placed between the two distributions.
After the image is thresholded the edges of each feature in the image can be easily detected. Edge detection is accomplished by inspecting the neighbors of each pixel. If a pixel has between 2 and 7 black neighbors, it is marked as an edge point. Edge detection does not alter the image. Instead, it produces a new image which solely contains the edges of each feature traced in black. The edge detected image is analyzed by an algorithm that logically groups connected edge points. Each resulting connected component represents a unique feature in the image. Finally, each feature is classified by registering it with a unique signature that is invariant across space and time. For this process, we use the second moments of inertia about the feature’s principal axes of rotation. Because this property is invariant, it saves us from having to reclassify the feature when it changes location.

*Figure 1: Thresholding via histogram analysis*
4. Implementation

The computer vision system is implemented in C/C++. Because the API provided by Point Grey Research is intended for use under the Windows operating system, we have developed the system using Microsoft Visual Studio 6.0. The contents of the relevant files in the project are as follows:

- **Camera.cpp**: Functions for interfacing with the camera.
- **ConnectedComponents.cpp**: The algorithm for calculation of connected components.
- **FeatureDetector.cpp**: Contains functions for feature classification.
- **Features.cpp**: Main function for feature extraction.
- **SimpleEdge.cpp**: Edge detection.
- **Threshold.cpp**: Various image thresholding algorithms.

Additionally, the execution of events in our main control module (**Main.cpp**) is as follows:

```c
extern FlyCaptureImage imageGray;
unsigned char* IMAGE_FRAME;

int main() {
    StartCamera();
```
while(1) {
    CaptureGrayscale();
    OptimalThreshold(imageGray);

    IMAGE_FRAME = imageGray.pData;
    SelTargetFeature();
    Features();
}
...

4.1 StartCamera

This function handles the initialization of the Point Grey Dragonfly camera. It is heavily dependent on the Point Grey API and many of the calling conventions reflect this fact. Additionally, all Point Grey function calls in the source code return an error (both successful and failed calls) which is handled by an error handling function, _HANDLE_ERROR(FlyCaptureError error, char* function). However, this convention has been omitted from this document in order to aid in readability. The basic execution of events in StartCamera is as follows:

FlyCaptureContext context;
FlyCaptureImage rawImage;
FlyCaptureImage imageConverted;
FlyCaptureImage imageGray;

int outRows = 200;
int outCols = 200;

int StartCamera() {
...
    flycaptureCreateContext(&context);
    flycaptureInitialize(context, _CAMERA_INDEX);

    flycaptureStart(context,
        FLYCAPTURE_VIDEOMODE_ANY,
        FLYCAPTURE_FRAMERATE_ANY);
...

    flycaptureGrabImage2(context, &rawImage);
    imageConverted.pData = new unsigned char[rawImage.iCols * rawImage.iRows * 4];
    imageConverted.pixelFormat = FLYCAPTURE_BGRU;
imageGray.iCols = outCols;
imageGray.iRows = outRows;
imageGray.iRowInc = outCols;
imageGray.pData = new unsigned char[imageGray.iCols * 
    imageGray.iRows];
imageGray.pixelFormat = FLYCAPTURE_MONO8;
...

Each Point Grey camera must be associated with a FlyCaptureContext (context). The context is created with flycaptureCreateContext(&context). The actual association of the context and the camera is performed in flycaptureInitialize(context, _CAMERA_INDEX). Here, context is bound to the camera located on the FireWire bus at _CAMERA_INDEX which is defined to be 0.

Once the camera is associated with a context, it can begin capturing images. flycaptureStart(context, FLYCAPTURE_VIDEOMODE_ANY, FLYCAPTURE_FRAMERATE_ANY) begins this process. Arguments 2 and 3 of flycaptureStart set the video mode and the frame rate of the camera. In this case, they are set to “any usable”, which passes the decision to the camera’s driver.

In order to allocate memory for our images, we must capture one image for reference. flycaptureGrabImage2(context, &rawImage) grabs the initial image from the camera’s buffer and places it in rawImage. rawImage is an instance of the FlyCaptureImage data structure. A FlyCaptureImage is defined as follows:

```c
struct FlyCaptureImage {
    int iRows;
    int iCols;
    int iRowInc;
    FlyCaptureVideoMode videoMode;
    FlyCaptureTimestamp timestamp;
    unsigned char* pData;
    bool bStippled;
    FlyCapturePixelFormat pixelFormat;
    unsigned long ulReserved[6];
}
```
The members that we are interested in are iRows, iCols, iRowInc, pData, and pixelFormat. iRows and iCols are number of rows and columns (in pixels) in the image. iRowInc is the number of bytes per row in the image. pData is a pointer to the actual image data and pixelFormat is the format of this data.

To declare the FlyCaptureImages that will be used to hold our images at various stages in processing, we must initialize these members.

   imageConverted holds a BGRU (black, green, red, unused) image. Therefore we must allocate 4x as much memory as rawImage. Additionally, we must explicitly set the pixel format to BGRU. This accomplished by setting imageConverted.pixelFormat to FLYCAPTURE_BGRU. Similarly, imageGray holds an 8-bit grayscale image. However, imageGray will be down-sampled to 200 x 200 pixels. Therefore, we explicitly set imageGray.iRows, imageGray.iCols, and imageGray.iRowInc to the appropriate values (outRows and outCols). Additionally, we must again explicitly set the pixel format, this time, to 8-bit grayscale. This is accomplished by setting imageGray.pixelFormat to FLYCAPTURE_MONO8.

4.2 CaptureGrayscale

This function is called every time an image is to be captured. Like StartCamera, it is fairly dependent on the Point Grey API. First, an image is captured and stored in rawImage. As is the case in StartCamera, this action is performed by calling flycaptureGrabImage2(context, &rawImage). The resulting image is a Bayer tiled image. Each pixel in a Bayer tiled image is filtered to record only one of three colors (Red, Green, or Blue), so the image is meaningless. However, we are able to interpolate
the complete image by converting it with `flycaptureConvertImage(context, &rawImage, &imageConverted)`. The resulting image is stored in `imageConverted`, and contains a meaningful pixel array of RGB values.

`imageConverted` is then down-sampled to 200 x 200 pixels and converted to 8-bit grayscale for the sake of efficiency. The down-sampling algorithm averages the pixel values over three adjacent pixels:

```plaintext
for each p1 in imageConverted
  for each p2 in imageGray
  end
end
```

The result is stored in `imageGray`. The conversion to 8-bit grayscale is taken care of for us by the `PixelFormat` of `imageGray`, which is set to `FLYCAPTURE_MONO8`.

### 4.3 OptimalThreshold

When performing an image threshold, one would typically want to set the segmentation value at the lowest local minima of the histogram function. Unfortunately, not all images produce a histogram with a distinct bi-modal distribution; only simple, high contrast images. This is simply not the case in a system that receives real-world visual input. Complications such as differences in lighting as well as the object-to-background ratio can affect the histogram distributions for any given scene. Therefore, the ideal image thresholding algorithm would be efficient enough to converge at the lowest local minima in a bi-modal distribution yet robust enough to arrive at a meaningful segmentation value in a histogram that reflects typical real-world scenarios.
Figure 3: Segmentation at lowest local minima in images of varying complexity.

The algorithm implemented is the iterative (optimal) image thresholding algorithm described by Sonka et al in *Image Processing, Analysis, and Machine Vision*:  

1. Choose an arbitrary starting segmentation value $T$.
2. At step $t$, compute $\mu'_B$ and $\mu'_O$ as the mean background grayscale intensity and mean object grayscale intensity respectively ($f(i,j)$ is the grayscale value of the pixel located in the $i^{th}$ column, $j^{th}$ row).

$$\mu'_B = \frac{\sum_{f(i,j) \in background} f(i,j)}{\# background \ pixels}$$

$$\mu'_O = \frac{\sum_{f(i,j) \in object} f(i,j)}{\# object \ pixels}$$

3. Set $T^{(t+1)} = \frac{\mu'_B + \mu'_O}{2}$
4. $T^{(t+1)}$ now provides an updated segmentation value.
5. If $T^{(t+1)} = T^t$, halt; else goto (2).

The algorithm proves to be particular effective for several reasons. For one, very little needs to be known about the image for the algorithm to converge on a meaningful segmentation value. It is arbitrary whether the object pixels are black or white. It only
matters that they are a different color than the background. Additionally, the algorithm operates equally well on noisy images. Because noise only occupies a relatively small area of an object or background, it is contained in the respective set of pixels and therefore has only minimal impact on the overall grayscale intensity of its set.

4.4 SelTargetFeature

This function marks the address of the target feature in memory, and flags a global variable TARGET_AVAIL as true. If a target already exists the function returns immediately. Currently the target is defined as the largest feature in the scene.

4.5 Features

In order to maintain maximal compatibility with our reference system, we have adopted the same naming conventions. Because the current image at any given time in the reference system is referred to as IMAGE_FRAME, we set IMAGE_FRAME equal imageGray.pData, which currently contains a thresholded image. Within Features the basic execution of events is as follows:

```c
extern unsigned char* IMAGE_FRAME;
char edgeMap[40000];
extern CompListPtr featureList;

void Features(void) {
  ...  
  MakeEdgemap(IMAGE_FRAME, edgeMap);
  ... 
  BitmapConnect(edgeMap, featureList);
  ... 
  for (int i = 0; i < featureList->componentCount; i++) {
    FPointCentroid(featureList->comp[i]);
    FPointInerTensor(featureList->comp[i]);
  }
  ... 
  FindTargetFeature(featureList);
}
```
4.5.1 MakeEdgemap

This function implements the edge detection algorithm utilized by Antolovic in the *Skeyeball* system\(^1\). Edge detection is accomplished by moving a 3 x 3 mask throughout the image and examining the pixel at the center. If the pixel is black, its neighbors are examined for contiguity. If the pixel has between 2 and 7 black neighbors, a ‘1’ is written to its corresponding index in `edgeMap`\(^1\).

![Figure 4: Edges of varying complexity](image)

4.5.2 BitmapConnect

This function implements the connected components algorithm described by Lumia et al. in *A New Connected Components Algorithm for Virtual Memory Computers*\(^7\). The algorithm takes as input a binary image (in our case, `edgeMap`), and builds `featureList`, a structure that holds each unique feature. The algorithm makes two passes. In the first pass (top down), each row in `edgeMap` is examined. Adjacent points within the current row and the previous row are labeled as belonging to the same component. In the second pass (bottom up), equivalences in these labels are resolved via a depth-first search and the component count is stored in `featureList`. 
Figure 5: Labeling equivalences are resolved when component a and component b are recognized as belonging to the same parent component

featureList is of the type CompListPtr, a pointer to a ComponentList. A ComponentList is a small data structure that contains the number of components in the image (componentCount) and a pointer to these components (comp[i]; 0 <= i < componentCount) which are of type FPoint*. An FPoint* is defined as follows:

typedef struct FPoint {
    int fpCount;
    int offset_X;
    int offset_Y;
    long Ixx;
    long Iyy;
    long Ixy;
    long lambdaP;
    long lambdaM;
    short v[500];
    short h[500];
} FPoint;

fpCount is the number of points that compose the component. offset_X and offset_Y are the coordinates of the component’s center of mass. Ixx, Iyy, and Ixy are components of the inertial tensor and lambdaP and lambdaM are its second moments of inertia which are used in the classification of each unique feature. v and h are pointers to each set of points that compose the component. v holds the y-coordinates and h holds the x-coordinates.
4.5.3 FPointCentroid

This function computes and stores the center of mass of the passed component FPoint*. The center of mass may be calculated by:

\[ \sum \frac{r_{(i,j)}m_{(i,j)}}{M} \]

\(r\) is the location and \(m\) is the mass of point \((i,j)\) and \(M\) is the total mass of the component. Because the component is a two-dimensional object of uniform density, the center of mass is actually the component’s centroid. Therefore \(m_{(i,j)}\) is always the same and \(M\) is merely the number of points in the component.

4.5.4 FPointInerTensor

This function computes the second moments of inertia of the passed component FPoint*. The second moments of inertia are invariant when the feature rotates or translates within the scene. This gives the feature a unique quality that can be used for tracking between frames. To find the second moments of inertia we compute the inertial tensor. The tensor is two-dimensional and is represented by a 2 x 2 matrix:

\[
\begin{bmatrix}
I_{xx} & I_{xy} \\
I_{yx} & I_{yy}
\end{bmatrix}
\]

where:

\[ I_{xx} = \int dM x^2 \]

\[ I_{xy} = \int dM xy \]

\[ I_{yy} = \int dM y^2 \]
\(dM\) is the infinitesimal mass (which in our case is uniform) of point \((x,y)\).

The second moments of inertia are the eigenvalues of the tensor. Recall that if a matrix is diagonizable its eigenvalues lay along the diagonal. Therefore, we take the discriminant of the tensor:

\[
D = \sqrt{(Ixx - Iyy)^2 + (4*Ixy^2)}
\]

The second moments of inertia (eigenvalues) are simply the roots of the discriminant:

\[
\frac{Ixx + Iyy \pm D}{2}
\]

### 4.5.5 FindTargetFeature

This function locates the address of the current target feature in memory. Because the position of features generally varies between frames we must examine the invariant properties of the features – their second moments of inertia. They should remain more or less unchanged across frames:

\[
|\lambda_{M_{\text{target}}} - \lambda_{M_{\text{candidate}}}| < 2000 \\
|\lambda_{P_{\text{target}}} - \lambda_{P_{\text{candidate}}}| < 2000
\]

(Note: a difference of 2000 is a relatively small difference on the scale of second moments of inertia). Additionally, the number of points that compose the component is examined, as it should hold fairly constant in only two dimensions:

\[
|f_{P_{\text{target}}} - f_{P_{\text{candidate}}}| < 15
\]

If these criteria are met, the candidate feature is identified as the target. Else, we record that we were unable to locate the target and try again during the next iteration. If we are unable to locate the target in three successive tries, a global variable TARGET_AVAIL is
flagged as false.

5. Conclusions

Using the combination of methods proposed above, we are able to reliably track features within a scene. The test setup uses a Pentium 1.6 GHz PC with 512 MB RAM running Windows 2000. Averaged over 1000 iterations, the system operates at 14 Hz.

Of course, there is always room for improvement. Because our system is designed for use in a terrestrial navigation system, a control scheme must be implemented. The control scheme can be as simple as a system of attractors and detractors such as that described by Braitenberg in *Vehicles: Experiments in Synthetic Psychology*. A neural network can be designed that examines the shape of features. If the feature matches the shape of the current path (parallel lines extending into a vanishing point), the vehicle calculates only minor adjustments weighted toward the center of the path. Conversely, if feature matches the shape of an obstacle a motion vector is computed to move the vehicle away from the particular feature.

Additionally, the system can be designed to closer mimic the design of the human visual cortex. The visual cortex is designed as a set of parallel specialized modules that integrate at later stages. The module in which motion is encoded (MT) is distinct from the module that color is encoded (V4). While our system is modularized in a similar fashion, the stages are advanced through sequentially.
While designing many highly specialized visual subsystems is hardly practical, there are ways to accommodate a more human-like visual system. One suggestion is to introduce many sources of input instead of many subsystems for processing. The sources of input do not necessarily need to be identical. Having identical sources of input does introduce interesting possibilities such as depth mapping and has been explored\textsuperscript{6}. However, there is also the possibility of having separate input sources for red, blue, and green color filtered images as well as low and high contrast filtered images. This setup mimics that of the cones and rods present in the human retina. Cones are sensitive to red, green, and blue light while rods are sensitive to the intensity of light.

Color filtered imaging has already been realized and is implemented as Bayer filtering on photosensor arrays such as those found in the Dragonfly series of cameras. Contrast filtered imaging has not been explored, at least to our knowledge. One way to
artificially implement contrast filtered imaging would be to equip the system with input
devices that can capture light in non-visible parts of the electromagnetic spectrum. We
envision that the ideal range to capture would be the infrared range, as it provides above
average resolving power in low contrast scenarios yet is biologically plausible. Humans
experience non-visual cues that aid in visual perception, such as sound or heat. Because
infrared radiation is intrinsically linked to the temperature of the object, we propose that
this gives an artificial system a secondary cue in visual perception. Additionally, the pit
viper is known to have infrared sensory organs on its head\(^4,5\). This is proof that the
construct exists elsewhere in nature.

Of course, there are an infinite number of secondary cues that can be added to an
artificial visual perception system. Infrared image capture is only one idea out of many
that can only be fully realized in time.
References


