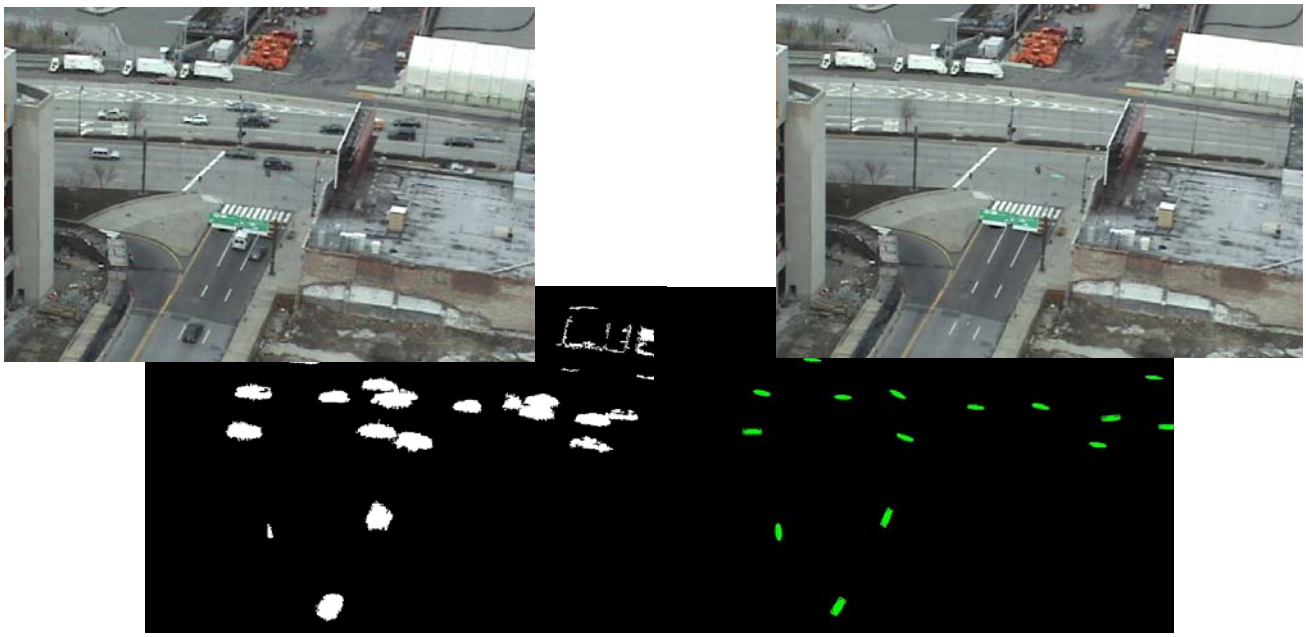


Synthesizing Virtual Satellite Views of Highway Traffic

David Lariviere (dal2103@columbia.edu)
Final Project for Automatic Visual Surveillance, Spring 2008
Taught by Drs Andrew Senior, Rogerio Feris, and Ying-Li Tian.

(Top Left) Input Frame. (Top Right) Background Model. (Middle Left) Foreground. (Middle Right) Estimated Vehicle Locations and Orientations. (Bottom Center) Synthesized Virtual Satellite View of Highway Traffic.

Table of Contents

Introduction

The goal of my final project was to create a software program for creating synthetic overhead views, as would be seen from a satellite, of traffic moving on a highway, given surveillance footage taken from a nearby building.

Visually speaking, given a video camera with a view like Figure 1a**,** synthesize a virtual "overhead" view of the same scene, as it would appear from Figure 1b.

(a)

(b)

Figure 1: (a) Input "Side" View, (b) Overhead View

Notation

C_1: Used to describe side view, including the focal plane upon which the image is projected.
C_2: Used to describe the overhead view, including the focal plane upon which the image is projected.
x_s: a 2D point expressed in pixel coordinates of C_1.
x_o: a 2D point expressed in pixel coordinates of C_2.
G_P: The Ground Plane, in 3D space, of which C_1 and C_2 share at least partially overlapping views.
H: The homography (projective transformation) that maps points from C_1 to C_2, for all points on G_P.
B_i: The i^{th} 3D box which rests upon G_P.
View: Planar projection of either 2D or 3D points onto a 2D surface.

Mathematical Background:

General Case: Plane to Plane Mapping via Homography

As explained in the introduction, the intent is to map pixels in one image to pixels in another. More specifically, we wish to map between two distinct views of a common ground plane, G_P, as observed and projected onto the focal planes of two separate cameras, C_1, and C_2.

For the specific case where all points are known to be coplanar in 3D space, it is possible to calculate a warping which maps the location of where each point is seen in one image to where it would be seen in another. Note that this is only possible in the case where all 3D points are coplanar. For example, given Figure 2, there exists a homography, H, which maps coplanar points between two Planes. Note, however, that if H maps on a plane G_P, then the same H cannot, in general, also correctly map points not on G_P.

Assume one is given a set of at least four corresponding points, x_s in the side view C_1, and x_o in the overhead view C_2, such that each pair of points are of the same physical point in 3D space, X. If no exists at least four pairs, such that no three are collinear, then the relationship between any pair of points between the two views is determined, up to scale, by a homography:

$$x_o^T \times H \; x_s = 0$$

(where '×' is the vector cross product).

In addition, given H and either x_o or x_s, it is possible to calculate the other point:

$$x_o = H \, x_s$$

$$H^{-1} \, x_o = x_s$$

The above pair of equations provides the means to calculate where a point in the second image will appear, given the location of the same 3D point in the first image.

[1] Multiple View Geometry. Hartley and Zisserman.

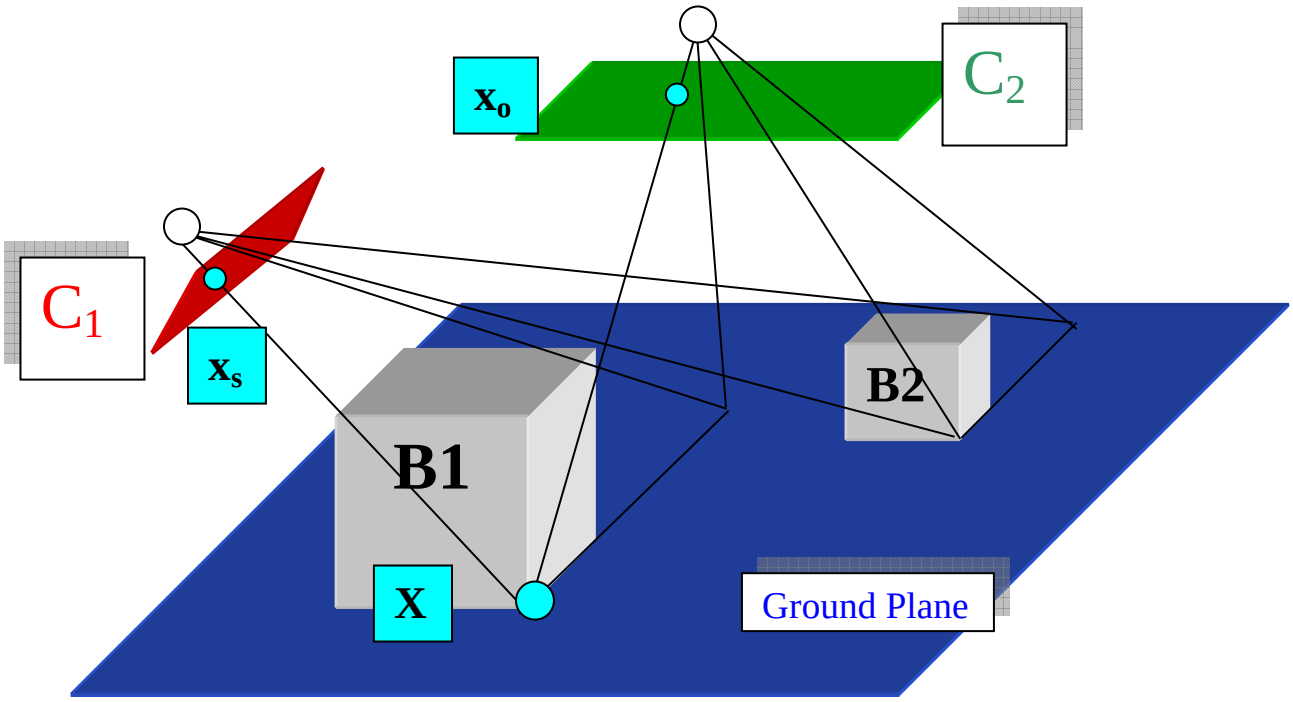


Figure 2 View of a 3D scene which lies upon a Groundplane, as seen from both the side and overhead views, C_1 and C_2, respectively.

Computation of Homography:

There are many methods for calculating the homography, depending on the set of limitations imposed upon the relationship between the two views. In the case considered for the side to overhead view mapping of objects on the ground plane, at least 4 pairs of point correspondences between the side and overhead views are required, subject to the non-collinearity constraint mentioned.

Note that the original intent of the project was to automatically compute H given a single view. It was eventually suggested to simply manually calculate the Homography, instead.

This was a logical modification given the desire to create not an arbitrary bird's eye of a ground plane, but instead warp all data to existing satellite photographs of the area under observation. In addition, the usage of satellite photos allows one to much better examine the quality of the mapping and vehicle localization.

Homography Results:

Mapping Results between Side and Overhead Views:

View Figure 3 shows a typical input frame, along with the corresponding points which were used in estimating the homography. Note that 4 rather than the minimum 5 points were used, leading to an overconstrained problem.

View Figure 3 Example side view with point correspondences (shown as Bullseyes) mapped to overhead

Figure 4 (Top) Overhead View with point correspondences. (Bottom) Side View projectively warped to overhead view, via computed Homography. Vertical lines are used to show accuracy of the mapping.

Mathematical Requirements for View Synthesis

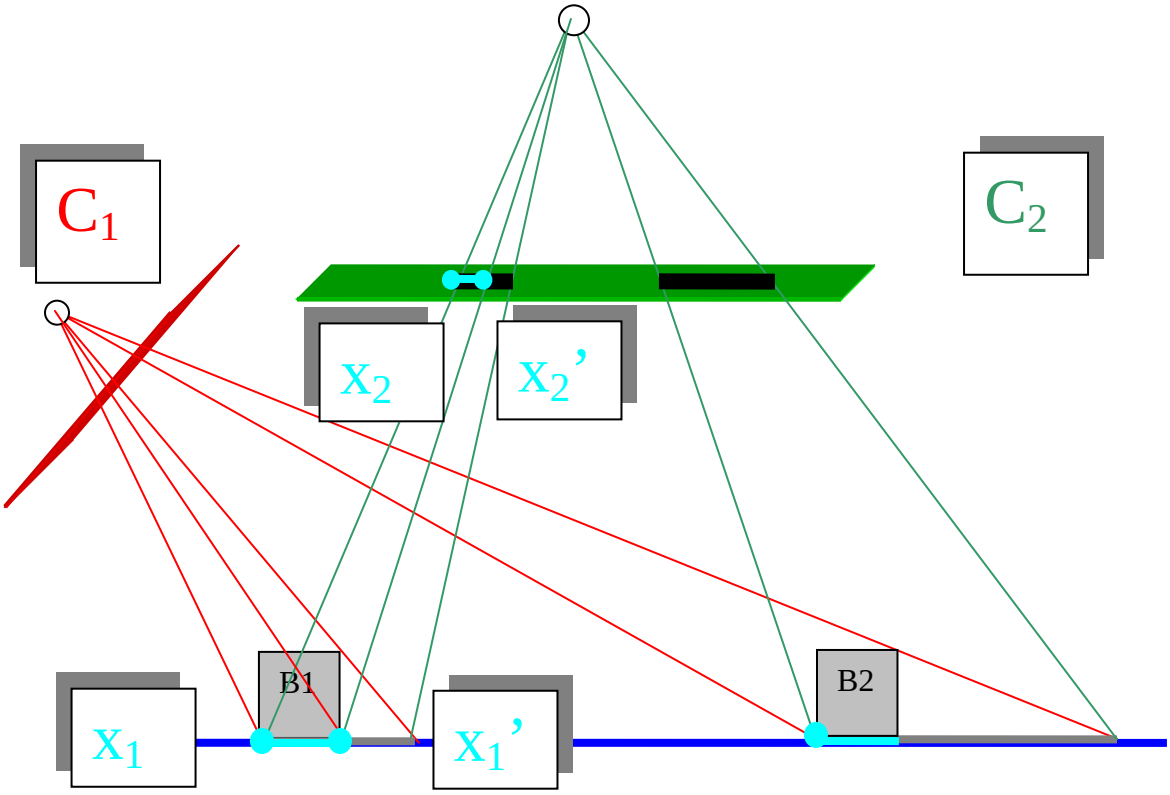


Figure 5 Another view (seen along X axis) of the 3D scene depected in Figure 2. Note how above, the projection of B_2 from C_1 back onto G_P is much larger than B_1, because B_2 is further away form C_1.

Required Information from C_1:

The primary motivation in processing C_1 in order to segment and extract objects is to collect sufficient information to *correctly* render each object as it *would appear from above, given a view from the side*.

Figure 2 and Figure 5 depict a relatively simple 3D scene in which two cameras share overlapping views of two coplanar cubes of equal size. Both cubes are based upon the ground plane. As is depicted in figures, our intent, given the view from C_1, is to synthesize an image approximating the view of C_2.

The fundamental problem, however, is that while a homography, H, is being used to map corresponding points on the ground plane, G_p, between C_1 and C_2, the objects in the scene are not actually completely within the ground plane (with z=0).

Simple View Synthesis

A naïve method of synthesizing a view from C_2 is as follows:

1. project the 3D scene onto C_1, thereby creating the perspective view of the scene, as viewed by C_1, under perspective distortion.
2. project the image of C_1 onto G_p, ignoring the 3D structure of the scene.
3. Project the image of G_P onto C_2, forming the virtual view.

One problem, as can be seen in the black lines projected onto C_2 in Figure 5, is that while both squares are of equal size, the resulting synthesized view contains errors that are both significant and non-linear, The further from C_1 an object is, the larger the distortion in its synthesized view projected onto C_2. This effect can also be clearly seen the homography warping example in Figure 4.

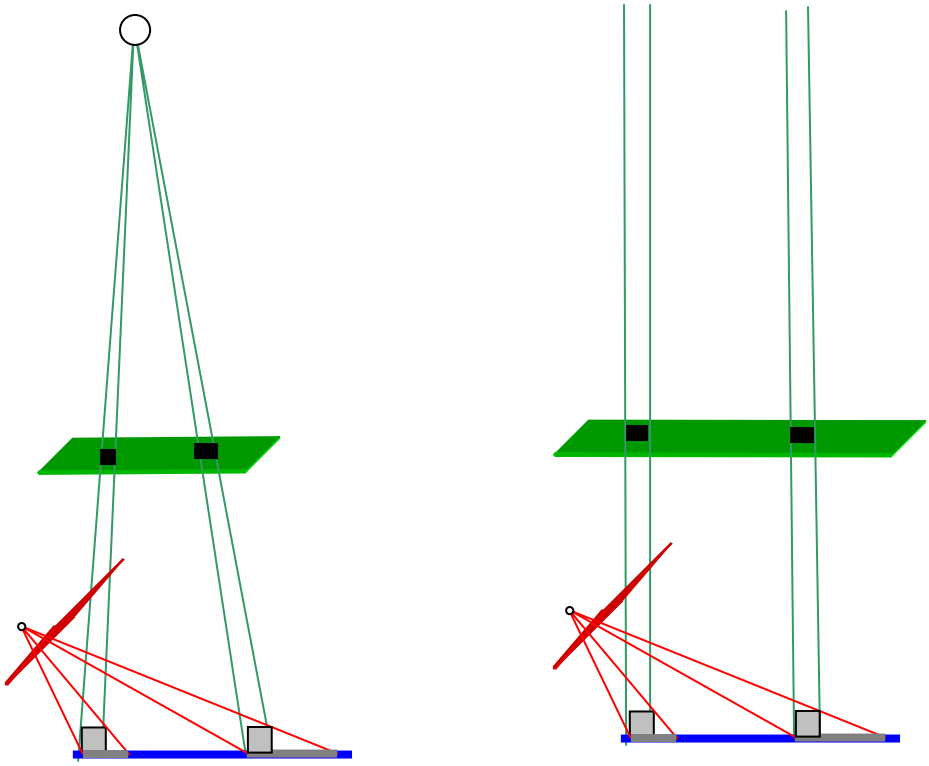
Object Constraints:

As so far presented, the problem of generating correct views for C_2 given objects projected onto C_1 is an under constrained problem that cannot be solved. It is under constrained because while H may map points upon G_p from C_1 to C_2, the mapping is known only up to an arbitrary scale factor, which in general lacks absolute measurements of distance across the plane. In addition, without further constraints, it is impossible to know the 3D structure of B1 given only a single view.

To constrain the problem sufficiently so that it may be solved, we make the assumption that the dimensions of B1, as observed from C_2, is known a priori. Given such information, it is then possible to reverse the process previously described. For example, consider the 2D view presented in Figure 5 of the image of B_1 projected onto C_1. Without loss of generality, we will restrict the example to the case of 1D projection of 2D scene, where the same technique should hold in the 2D → 3D space as well.

1. Locate the point, x_1, at which B_1 intersects G_p, such that the point is transferred from C_1 to a point x_2 in C_2, via H, (it will be correct, given x_1 is within G_p).
2. Given x_2 and known object dimensions (in 1D, only a known width, w), compute x_2', where x_2' is equal to $x_2 + w$.
3. Now compute the corresponding point in C_1, mapped from x_2', via H^{-1}.
4. Draw in C_1 the line through x_1 and x_1', which in the 1D case, draws the "object" such that when the image is warped to C_2, the line will be of length w.

Orthographic Projection:



(a) (b)

Figure 6 (a) depicts perspective projection in which the focal length is very large relative to the depth differences within the scene. (b) depicts orthographic projection in which the focal length is taken to infinity and rays become parallel.

All previous discussions of projection have assumed strictly perspective projection. If one assumes a slightly weaker model under certain conditions of the relative view and scene, then it is possible to simplify the mapping. Note Figure 6(a) in which the focal length (distance between the camera image plane C_2 and the focus point) is even larger than the distance between C_2 and G_p. Further note that focal length is significantly larger than any depth variations within the objects on G_p. In the hypothetical case where the focal length is increased to infinity, the resulting projection is known is *orthographic projection.* So long as the relative distance between the top and bottom of an object (the relative "depth" of the object, relative to C_2) is small, then the orthographic projection well models the perspective projection.

Equivalence of G_p and C_2 under Orthography:

Modeling the projection of the scene from the ground plane to C_2 under orthographic projection (orthography) has one additional *very* useful property for the specific situation being considered:

If C_2 is assumed to be parallel to G_P, and if the projection of G_P onto C_2 is well approximated by orthographic projection, then ***the projection of the scene onto C_2 is equivalent to the projection of the scene on G_P.*** If C_2 is equivalent to G_P, then an object located anywhere on G_P will appear the same in its projection in C_2!

The result is in stark contrast to the situation involving C_1 which is not parallel to G_P, in that the same object, depending on its location, can very dramatically in its projected size.

The further implication is that given:

1. a corner point within C_1 of an object on G_P
2. the object's orientation relative to G_P
3. the object's dimensions in C_2

it is possible to accurately reconstruct the scene viewed from C_2, given a single view from C_1 (see Figure 7).

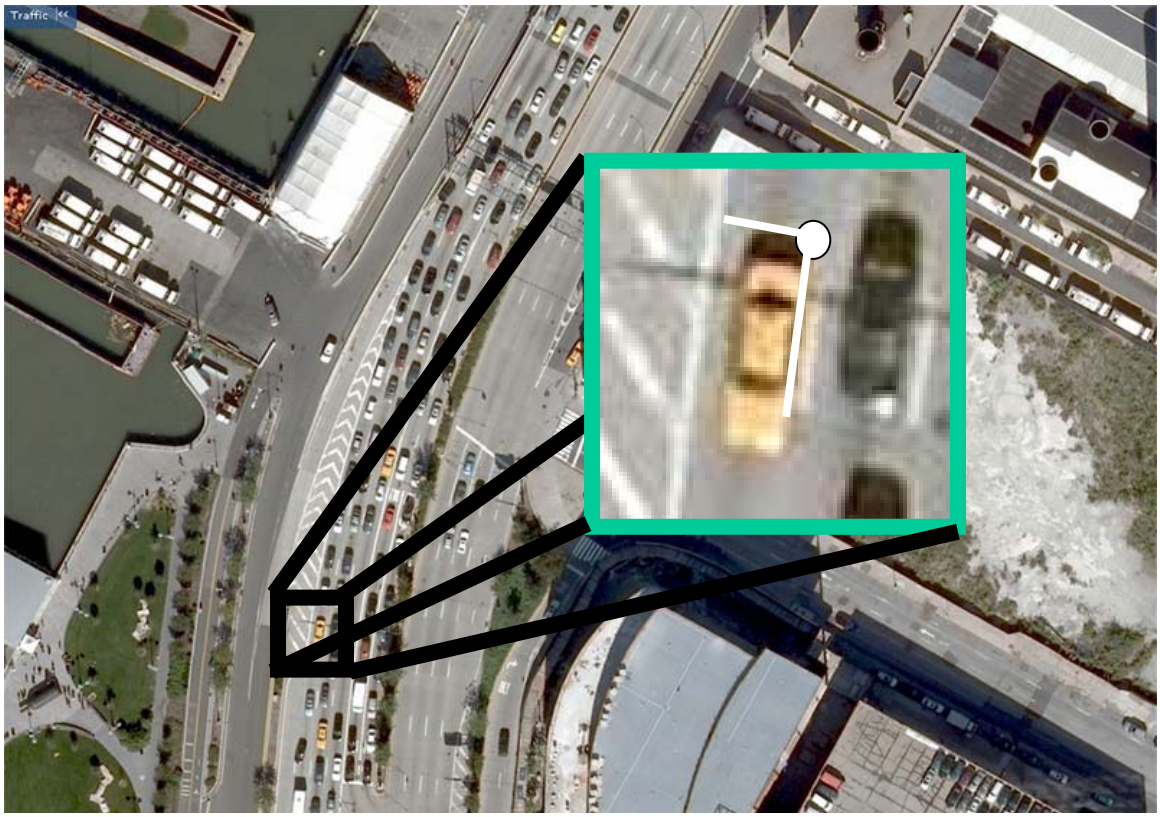


Figure 7 An overhead view well modeled by orthographic projection. Note how all cars throughout the image appear the same size. Given a single point on the ground plane, combined with known object orientation and width, it is possible to determine its projection within the view.

Overview of Video Processing Algorithms:

As outlined in Figure 8, the system consists of a chain of algorithms responsible for separating the image into foreground and background, grouping pixels of the foreground into objects, processing the properties of the objects, and then rendering each object as it would be seen from the overhead view, via the homography.

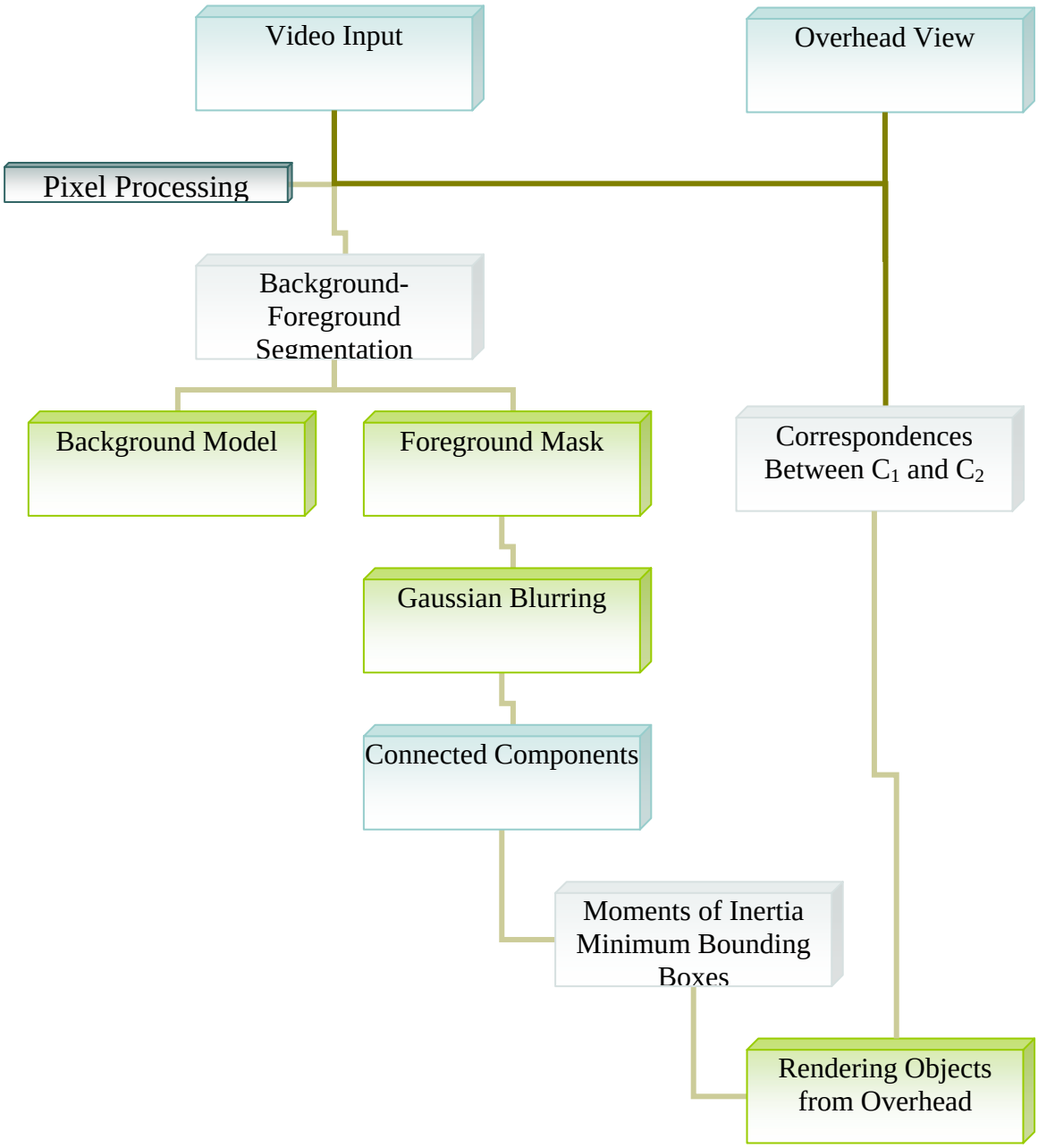


Figure 8: System Overview

Video Capture:

In the first step, video frames are individually captured, either from a pre-recorded movie file or captured live from an attached video camera over firewire. All pre-recorded video was captured utilizing a Sony DCR-HC21, optionally f to tape, and then capturing to a PC using either VLC or Adobe Premiere. In order to maintain maximum quality, all videos are initially captured and stored in the native DV format of the camera: 720x480 with 4:2:0 YUV sampling at roughly 28 Mbits/sec. A .5x wide-angle lens was also acquired for the project in order to maximize the field of view.

While many other methods were experimented with, it was found that in order to allow OpenCV's HighGUI video input library to successfully read a movie, it was unfortunately necessary to save all videos as RAW interlaced 4:2:0 wrapped in an AVI container. Initially ffmpeg was used for all video conversion, but later mencoder was also required in order to rotate one test video sequence which was required sideways. In addition to repackaging, ffmpeg was also utilized to deinterlace some video test sequences, which led to a clear increase in quality, removing interlacing artifacts that were extremely noticeable both in input video and the resulting foreground mask.

All recorded video was saved at either 720x480 or down sampled to 320x240, in order to increase performance, especially related to disk I/O.

In addition to resizing recorded video files beforehand, input frames can also be resized after being read into the program, in order to reduce the computational load on all subsequent algorithms. Note that the homography and associated point correspondences are automatically updated with respect to rescaling factor.

Volume of Recorded Video

During the life of the entire project countless hours of video footage were recorded under varying times of day, traffic patterns, illuminations, zooms, lenses, and weather. In total more than 150 GB of video were digitized. That said, only about 5 GB were used for the final stages of development.

Background-Foreground Segmentation:

Background

Once a video frame is read into memory and optionally resized, the frame is then passed on to the main video processing loop. The first step in processing the actual contents of the video is to segment each frame between the background, the static elements of the scene, and the foreground, possibly moving or at least relatively new objects which should be rendered in the overhead view. An implementation of the Mixture of Gaussians BGS algorithm provided with OpenCV is used [2].

The background model is iteratively updated as each new frame is processed. In order to first build a relatively stable background model before attempting to detect objects, at startup, the first several frames are processed by the BGFG module without being passed to subsequent models. With the proper framerate and learning update parameters, a proper background is quickly learnt (see Figure 9).

Foreground

Once an initial estimate of the background is calculated, it is then possible to calculate a foreground mask, marking areas of the current input frame whose pixel values differ from the background model above a preset threshold. The foreground mask, while often quite noisy, is essential in isolating potential objects of interest. In the mixture of Gaussian implementation provided in OpenCV, all pixels in the foreground are strictly segmented via the provided threshold into either foreground (1.0) or background (0.0), with no probability values in between.

Figure 9 (Top) Example Input Frame; (Bottom Left) Background Model. (Bottom Right) Foreground Mask for current frame given background model.

Connected Components Object Segmentation:

The foreground mask is next fed into another algorithm responsible for grouping connected regions of foreground pixels into sets. OpenCV surprisingly does not directly include a connected components segmentation implementation. The program therefore used cvBlobsLib, an opensource implementation compatible with OpenCV. [3]

cvBlobsLib is intended to operate on grayscale images, with a threshold parameter provided by the user to first apply binary segmentation, after which object segmentation is run on pixels above the threshold. The author of the implement mentions that the algorithm implemented (may) have been based upon [4]

Foreground Filtering Enhancement

It was often the case that the foreground masks were extremely noisy, especially around edges. One simple technique introduced during development to mitigate such artifacts was to apply a Gaussian blurring filter to the foreground mask before running connected components. The size of the blurring kernel and the threshold value for binary segmentation can be used to control the extent to which objects are shrunk or grown. This method was especially helpful in preventing separate objects that were barely touching from being segmented into a single connected component.

A strictly horizontal blur kernel was also manually constructed and tested in an attempt to mitigate a specific artifact. Note first that the camera was always positioned such that its Y-axis was perpendicular to the surface of the earth, meaning that thin cylindrical rods normal to the ground plane would appear as vertical lines in the side views. It was often the case that such thin vertical lines from poles, street signs, and lights would obstruct views of objects, splitting vehicles in two with their two connected components separated by only a few pixels. Experiments using linear scan-line blur kernels were also carried out. The method, while sometimes useful, did not demonstrate a noticeable improvement in quality (avoiding vertical blurring) as compared to the Gaussian blur kernel.

Vehicle Localization

Once the foreground has been segmented into separate objects, each object, containing a set of pixels, is then processed individually in order to estimate various properties: centroid (center of mass), orientation (axis of largest momentum), minimum oriented bounding box, average color, location on ground plane of side view, corresponding location in overhead view. The end result of vehicle localization within C_1 is shown in Figure 10.

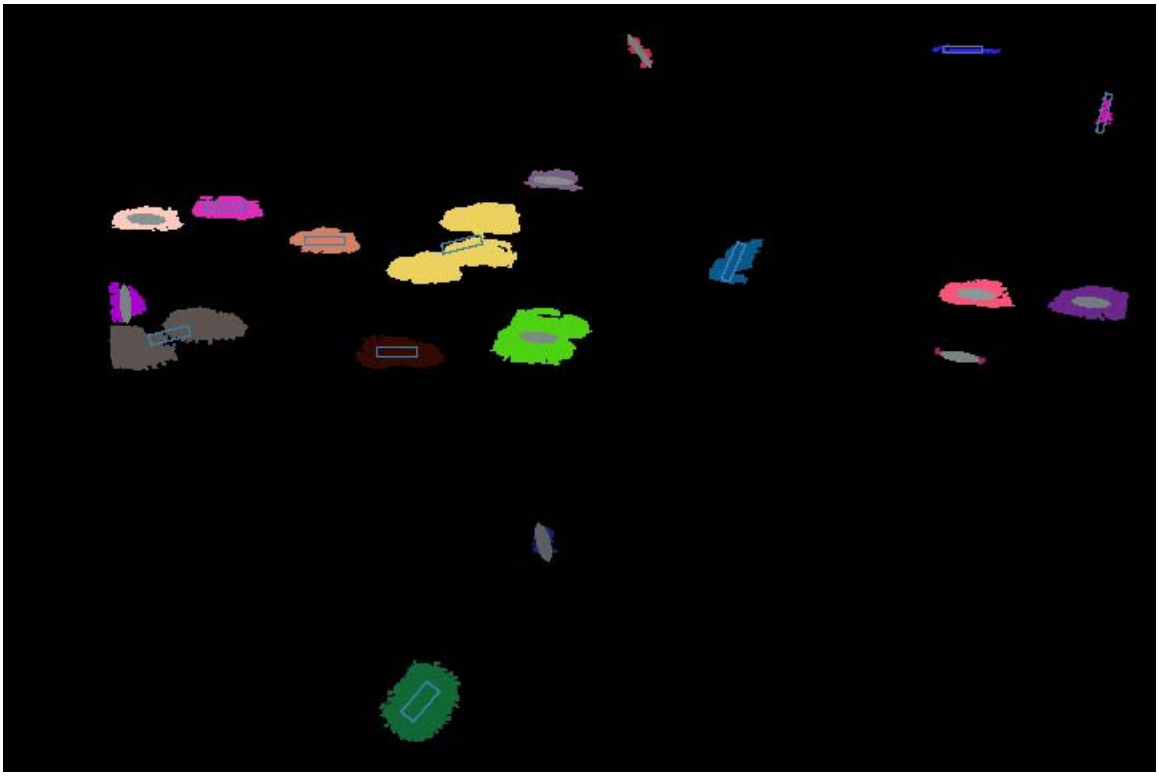


Figure 10 Object Segmented Image with Car Locations. To generate the image above, first the original foreground mask of each object is filled with a random color. Next one of the techniques is used to estimate the most likely location and orientation of a car. Lastly the car is drawn, either as an oriented ellipsoid, or oriented rectangle at the proper size such that when the full image is warped to C_2, cars are of the proper size.

Identifying Cars in the Side View:

In order to acquire the constraints related to object sizes in G_P and therefore C_2, it is possible to rely on satellite photographs which are relatively well modeled under orthography. Then, one can measure a vehicle in pixel coordinates to obtain the relative size of an average sedan-category car when viewed from C_2 upon G_P.

Obtaining Constraints from Objects in C_1:

As previously described in the mathematical section on, the final step in generating a synthetic view of C_2 given C_1 is to derive the object location (on G_P) and orientation. In the implementation of the software, it is assumed that all vehicles have roughly the same area as viewed from above. Therefore the problem becomes one of locating a known point on the object in C_1 that is on G_P and on calculating the object's orientation.

Several methods were experimented for estimating a known point of the object that is known to be on G_P, a description of two of which follows.

Moments of Inertia

One method of estimating both a known point on G_P and the object's orientation was to utilize the moments of inertia.

Center of Mass:

To locate a known point, one can utilize the first moment of inertia, also known as the center of mass of the object. Note that in the case of equally weighted points, the center of mass is equivalent to the average location among all of the object's points. If the Gaussian blurring technique is being applied, however, such that the foreground mask is not binary, with values in between (0,1), then the center of mass becomes distinct and a potentially more useful measure of the center point (as viewed from C_1).

The problem with using the center of mass, however, is that it is highly dependent on the relative orientation of C_1 with respect to C_2. Note that as C_1 approaches convergence with C_2, the centroid converges to that desired centroid on G_P/C_2. On the other hand, as C_1 approaches the horizon, the centroid converges to infinity.

Axis of Least Momentum:

Just as the first order moment was used to calculate the centroid, the second order moments of inertia can be utilized to estimate the orientation of the object, given its point set. Here again, however, in practice the measurement is extremely noisy. While it

sometimes produced acceptable results in the image, it was often off, and almost always vacillated about the true value on every successive frame.

Minimum Oriented Bounding Box:

An even simpler method of estimating properties of an object involves calculating the minimum oriented bounding box. The simplest bounding box is when such that its axis are aligned with the image axis, meaning that the corners of the bounding box are simply:

Corner1 = (minX, minY), corner2 = (minX, maxY), corner3 = (maxX, minY), corner 4= (maxX, maxY),
Where max/min(x/y) represent the largest or smallest value amongst all the object's points.

OpenCV provides a slightly more advanced algorithm which calculates the angle of orientation yielding the bounding box of smallest area for the object's point set.

Rendering Overhead View

Figure 11: Example Overhead View depicting both the simple warping scenario where pixels C_1 are directly mapped to C_2 (colored blobs) and also the refined estimates of vehicle localization/orientation (grey ellipsoids/rectangles).

Given the constraints previously discussed, it is at last possible to generate a synthetic overhead view. One can either utilize the point and its orientation to draw the object directly to the overhead view of C_2, or one utilize the inverse homography to first draw

the object in C_1 such that when the entire image is warped from C_1 to C_2 via H, it is rendered correctly, given the assumptions on object dimensions.

The first case of drawing the object directly in C_2 is trivial, by applying H to bring the point from $C_1$to C_2 and then drawing the object relative to the known point using the assumed dimensions.

The more complicated case of first drawing the object in C_1 and then warping it to C_2 requires a bit more effort. To draw the object "correctly" in C_1, it is necessary to essentially figure out how to draw C_1 relative to the point, such that after warping, it is rendered with the required dimensions in C_2. As an example, consider the simple case of an image-axis-aligned object observed in C_1 with its bottom-right point, known to be on G_P. Given the point C_1, one can compute the corresponding point in C_2, along with the point C_2 – (objectWidth, objectHeight). Where (objectWidth, objectHeight) are the expected dimensions of a car as measured in pixels throughout C_2. If the car is to be displayed properly after warping, then it should be drawn in C_1 utilizing the points provided by warping (C_2 – (objectWidth, objectHeight)) to C_1 with H^{-1} . Thus, when the complete view of C_1 is rendered in C_2, the object will be appear the proper dimensions (avoid integer quantization issues).

Drawing Objects In the Image

Filling Object's Foreground Mask

Throughout the life of the project several iterations of rendering methods were developed. The simplest and first method of displaying identified objects' locations was to simply fill in the foreground mask of each object with a random color. This provided a very rough approximate, but was heavily subject to the distortive problems discussed whereby the non-planar object's rays were being projected onto the ground plane before being warped into C_2 creating grossly large errors in the size of vehicles which varied inversely to the distance between the object and C_1.

Ellipse of Oriented Bounding Box

The next drawing method implemented was to utilize OpenCV's cvEllipseBox to draw an oriented ellipse with axis corresponding to the oriented width and height of the object. This method provided slightly better results.

Oriented Box:

The final drawing method was a simple drawing function to draw the outline of an oriented bounding box. Unfortunately the code does not yet fill the polygon, so when drawn it is difficult to see (although visible in Figure 11 within the orange blob in the center, for example).

Estimating Object Color:

Attempts were made to estimate the color of the object given all of the pixels considered part of its connected components set. The initial implementation simply calculated the average (R,G,B) value amongst all pixels in the set. Unfortunately, the result was always a shade of grey.

The problem is several fold. First the foreground mask was far from an exact, and often included considerable pixels outside the actual car, which most often meant the inclusion of pixels corresponding to the grey pavement. Second, the car itself often contained considerable dark patches, especially over the windows, tires, and depending on time of day, from the shadow.

Next, I tried a subsequent technique to both filter out extremes and "boost" the relative color by boosting relative differences between the (R,G,B) channels and the average grey examples, using a "boosting coefficient." To filter out extremes, a pixel's value was not contributed to the average unless it was within a certain threshold, in between black and white, in attempting to filter out both shadows and saturating highlights.

For an example of color boosting, if the average grey value were 195, but the average values for (R,G,B) were (185, 195, 205), and the "boosting coefficient" equal to 2.0, then the (R,G,B) value of the object would considered (175, 195, 215), such that the difference between any channel's value and the average grey value were multiplied by a factor equal to the boosting coefficient. Note that normalization was also applied so that average grey value would remain the same.

Unfortunately, the color boosting method also failed. At best some objects appeared various shades of yellow, pink, or brown.

Another method not yet implemented which may yield better results would be to utilize a colorspace other than RGB (YUV, for example), which in theory should separate the intensity of light from its wavelength. Depending on the precision of the foreground mask and amount of shadows, however, even this method may still fail.

Implementation Details:

Software Design

An initial attempt was made to design a re-usable well structured library. To this end, most externally used libraries for video input and connected components segmentation were abstracted with a custom abstraction layer, so that in the future alternative libraries can be swapped in. For example, there is a abstract GPEVideoInput class, from which both DSCamVideoInput (a wrapper for the dsCam video library), and HighGUIFileVideoInput (a wrapper for HighGUI's file-based video mechanisms).

Development Environment:

In the initial weeks of the project starting in mid-March, a concerted effort was made to utilize Eclipse IDE + CDT with MingW/GCC as the development platform of choice. Unfortunately, I was simultaneously also determined to find a way to capture live video over firewire during program execution, which required the use of an external library not included with OpenCV. Unfortunately the library, dsCam, utilized C++ and DirectX/DirectShow and was compiled with MSVC. Despite many attempts, it became clear that it would be impossible to utilize the library with GCC, and so it was required to abandon Eclipse/MingW and revert to Microsoft Visual Studio C++ 2005 Express Edition.

Software Libraries:

As mentioned throughout the algorithm sections, several third party libraries were utilized in the project, and many more evaluated during development, including OpenCV[5], cvBlobsLib, and dsCam.

By far the most important library used was OpenCV. OpenCV provided many of the fundamental algorithms, including the calculation of the homography and the perspective warping of images.

While OpenCV is an extremely powerful and often highly optimized image processing and computer vision library, it unfortunately suffers a few limitations which significantly affected both this project, and many others I have worked on in the past involving video.

Video Input

OpenCV contains a hodge-podge of methods for reading in video, which vary both across and within Operating Systems. For Windows, there are two separate libraries: HighGUI and CvCAM. While it was possible to stream video over USB, neither library was able to access my video camera over firewire, presumably because both rely on the now ancient VfW (Video for Windows) API which hasn't been updated with firewire video support.

The quality difference between streaming the output of the same video camera over firewire and USB, both with respect to resolution and level of compression artifacts, was dramatic enough to warrant an extra investigation into video input.

In addition to live video input, HighGUI is also the OpenCV recommended method of inputing recorded video files into OpenCV on Windows. Unfortunately, most likely again due to VfW, HighGUI's video input library was extremely limited in its support. Virtually the only file format that worked was RAW interlaced I420 wrapped in AVI. There are also unconfirmed reports on many forums that some versions of AVI are limited to 1GB in size. When creating AVI files larger than 1 GB, it seems that some video encoding programs (ffmpeg) would package multiple video streams, splitting on 1 GB boundaries. OpenCV could play the first 1 GB segment of the video, but not subsequent streams within the file, placing a limit on the size of pre-recorded video that could be used during testing. The 1 GB input limit was another reason for transcoding video to 320x240, such that video clips four times as long could be played.

dsCam Library

There are several third party libraries mentioned on the OpenCV Wiki to input video from cameras, including dsCam, which provides support for synchronizing and simultaneously reading from several cameras. Unfortunately, after several days of work, it was determined impossible to use the Visual Studio-compiled library with Eclipse+MingW. Much more important, however, about half way through the project in April, I realized that statically linking the library was somehow corrupting the stack, *even when none of its functions were actually being called!* It took a few weeks to figure out, since the program was continuously crashing while within specific functions in the connected components library, falsely leading me to believe that the problem was in fact there.

Conclusion:

In the end, while I did not create a program for automatically constructing Homographies from video, the primary goal and ambition for the project was achieved: the accurate generation of virtual bird's eye views of traffic on the West Side Highway.

I learned a lot through this project. Due to time constraints, I was somewhat limited in the number and complexity of features that I could pursue. The good news is that I now have a solid code base upon which to build more advanced traffic surveillance and processing software.

Going forward, the first feature I want to add is template based matching and tracking. A simple tracker, comparing segmented object statistics (centroid, area, color, etc) was implemented and performed quite well at matching the same blob across frames, so long as the blob did not merge with blobs of other vehicles. During testing, however, it was found that occlusions from clumps of adjacent vehicles posed significant problems, and would require a more advanced tracker with ability to handle partial occlusions.

Once I have a module capable of efficiently extracting templates and matching, my hope is to utilize the many hours of input footage to train Viola and Jones style feature part detectors for both different types and sub features of cars, which going forward can be used to handle occlusions, and improve the background/foreground segmentation modules. Lastly, given the extremely predictable linear motion that governs most cars, applying Kalman filtering to predict objects in subsequent frames is expected to yield worthwhile results.

Bibliography

[1] Multiple View Geometry, 2nd Edition. Richard Hartely and Andrew Zisserman. Cambridge press.
[2] KaewTraKulPong and Bowden. “"An Improved Adaptive Background Mixture Model for Real-time Tracking and Shadow Detection." Provided with OpenCV v1.0 in cvbgfg_gaussmix.cpp
[3] cvBlobsLib. http://opencvlibrary.sourceforge.net/cvBlobsLib
[4] Snyder, W. Cowart, A. “An Iterative Approach to Region Growing.” IEEE Transactions on Pattern Analysis and Machine Intelligence, 1983.
[5] OpenCV v1.0. Intel Corporation. http://sourceforge.net/projects/opencvlibrary/