XVISION 2 - A FRAMEWORK FOR DYNAMIC VISION

by

Samuel J. Lang

A thesis submitted to The Johns Hopkins University in conformity with the requirements for the degree of Master of Science in Engineering.

Baltimore, Maryland

October, 2001

Abstract

XVision2 is a redesign and implementation of the XVision framework, a standard tool for dynamic vision and visual tracking. The idea behind XVision2 is to provide a platform for implementing vision applications, on standard desktop hardware, in a dynamic and real-time fashion. The XVision2 library combines many of the features of the old library, while taking further advantage of the features of the C++ language and providing support for many of the new technologies available today. This document details the design and motivation of XVision2 , the new features over the old XVision, and some of the implementation issues involved.

Acknowledgements

The author wishes to acknowledge those who have provided counsel and support throughout this project. My thesis advisor, Dr. Greg Hager has provided guidance, motivation, excellent advice, and financial support. He has remained patient with me through my inexperience, obtuseness, and general quirks. I wish to thank Darius Burschka, for his good advice, much needed technical support, and the many quality discussions over a Bass ale. Thanks to Xiantian Dai (Donald) and Jason Corso for their help with the XVision2 library, and the C++ language. Linda Rorke has provided information and advice necessary to finish this Thesis. Finally, but in no way least important, the role my parents have played as counselors and role models in bringing me to this point.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Research in computer vision, especially in the area of tracking has largely focused on finding robust algorithms that perform well independent of their computational complexity. Testing and verification of such algorithms is often done on pre-stored image sequences, over an unspecified period of time. This research is motivated by the idea that sooner or later the computational power needed to run these algorithms in real-time will be developed [5].

Unfortunately, research of this kind overlooks the many applications of visual tracking that exist today, applications which provide motivation for a real-time visual tracking software library that supports them. Approaches that attempt to provide robustness for a wide variety of tracking applications (i.e. globally), such as the popular Canny algorithm, are impractical because computational complexity prevents using them in a real-time vision application. Instead, these algorithms can be replaced by a set of simple but fast feature detection algorithms. With such a basis set, a bottom-up approach can be taken to tracking, where sets of simple features are combined in a constrained manner to form solutions to specific tracking applications.

XVision has been a standard tool used for real-time vision and tracking applications for many computer vision and robotics researchers within the past decade. Designed by Greg Hager and Kentaro Toyama [5, 4], the software library not only provides the end user with a rich set of features to build on for tracking and vision, but it also provides a framework for handling image sequences sent from a video device (a.ka. frame grabber) and displaying images in a window. Although still widely used, as computing power has progressed, and the standards for video devices have changed, much of the support that XVision provides has become outdated. Such features as digital cameras over analog, and support for color images over simply grayscale have become highly demanded and often required by the vision community. XVision2 attempts to solve many of these problems of the older version of XVision, while still providing the real-time capabilities and modular design of the previous library.

One of the most important changes in the new XVision2 over the old XVision, is the design of images. XVision previously supported only single banded images, or grayscale. While this design provided enough support for most applications in the past, it neglected the support of many of the simple tracking algorithms and applications that use color as their primary parameter for localization. The design of the XVision2 image classes includes support for just about any possible image type, from RGB to YUV, as well as supporting standard grayscale.

The design of images with color in mind provides another advantage: support for color video devices (framegrabbers) and window displays was easily added to the library. Although the old XVision had provided support for video devices and window displays, support remained in the grayscale domain, so that few of the features of currently available hardware were used. By providing a color framework to build on, these components of the library were able to include support for color, including color display using the X-Windows toolkit, and support for the new IEEE1394 (a.k.a. Firewire) video device. The resulting advantages to the end user are substantial. The start-up time for developing vision algorithms or applications based on color is greatly reduced, as the end-user is provided with a simple API that provides a mechanism for these available tools. The added advantage is reduced startup time WITH color, over the features already available in the original XVision library [5].

For the rest of this document, the notion of a *feature* will refer to a specific local part of an image with special properties. A feature may be an edge or a line, or it may be an object marked by its color. Both what are referred to as feature-based and region-based identifiers in Figure 1.1 can be considered *features* for this discussion.

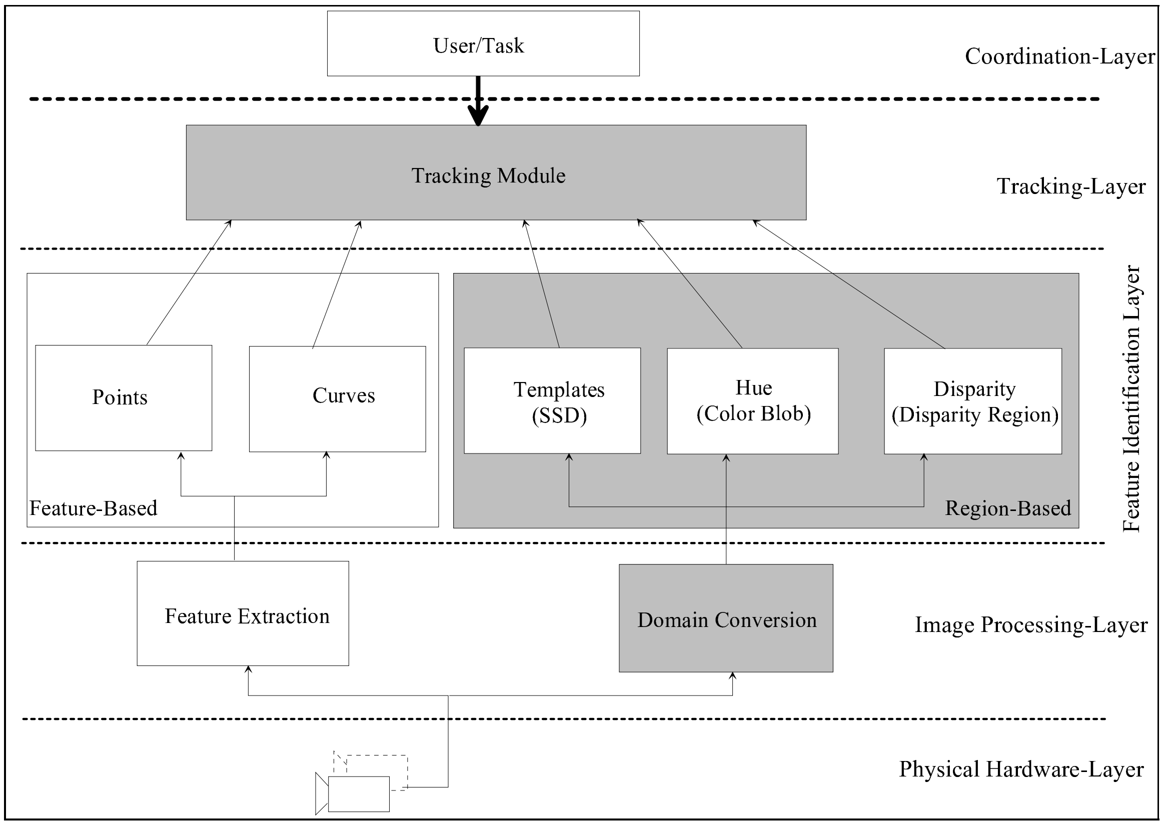

Figure 1.1: Tracking Hierarchy

As stated previously, combining simple features together allows tracking to be done at real-time, and have a robustness competitive to other top-down approaches. Two important aspects of feature tracking, are the performance of individual features, and the combination of features for a specific tracked object which provide the most optimal results. Both aspects hinge on the notion of a feature's state. A feature's *state* consists of the parameters of that feature (location, size, etc.) at a given point in time. By generalizing the idea of state, we provide a standard mechanism for measuring the performance of an individual feature (compared to both its own previous performance and performance of other features). With a standard state for each feature, we can also build complex features and measure the performance of such features, using the state values of the basic features as input.

The remainder of this document is divided into three separate chapters. The first discussed the design and implementation of images in XVision2 , along with some of their characteristics and features. The second chapter details the tracking framework and its design. The final chapter gives a general overview of a few applications using the XVision2 library, demonstrating the ease of the library to construct complex tracking environments while maintaining performance. An appendix concerning video devices and windowing displays, their design and some the features follows this document.

Chapter 2

Images

2.1 Introduction

Images in XVision2 are the basic building block of the library. They are the components pulled from frame grabbing devices, pushed through many different manipulations (such as warping or segmentation), and finally sent to a display device. As the main component, Images were designed to support a wide variety of formats, and allow other formats to be added in a simple and straightforward manner. This chapter describes the design of Images, some of their chief characteristics, and how they are implemented.

2.2 Image Formats

In previous versions of the XVision library (written in 1993), images were mainly only of one type: gray-scale [4]. This was due primarily to the limiting capabilities of the hardware. Frame grabbers and displays (without getting extremely costly) only produced grayscale images. But as color frame grabbers and displays became cheap and commonplace, the need for a vision library that supported all the possible image formats was necessary. Thus, one of the primary design goals of XVision2 is to support a wide variety of image formats for video cards, displays, and image processing algorithms. For such a design requirement, not only was it important to support many of the current standard pixel representations (such as RGB, YUV, HSV, etc.), but to provide a design mechanism for easily and conveniently adding other pixel formats. This suggests a parametric design, where the behavior of the image does not depend upon the type of which it is instantiated [12]. One can imagine a combination of inheritance and parametric polymorphism, where an image is represented by a class (we specify the design in terms of the C++ language), which is templated on the pixel format. Further specification and functionality of image types, relating to grayscale or color versions, can be performed through inheritance of the abstract image type. Figure 2.1 shows a diagram of the image framework.

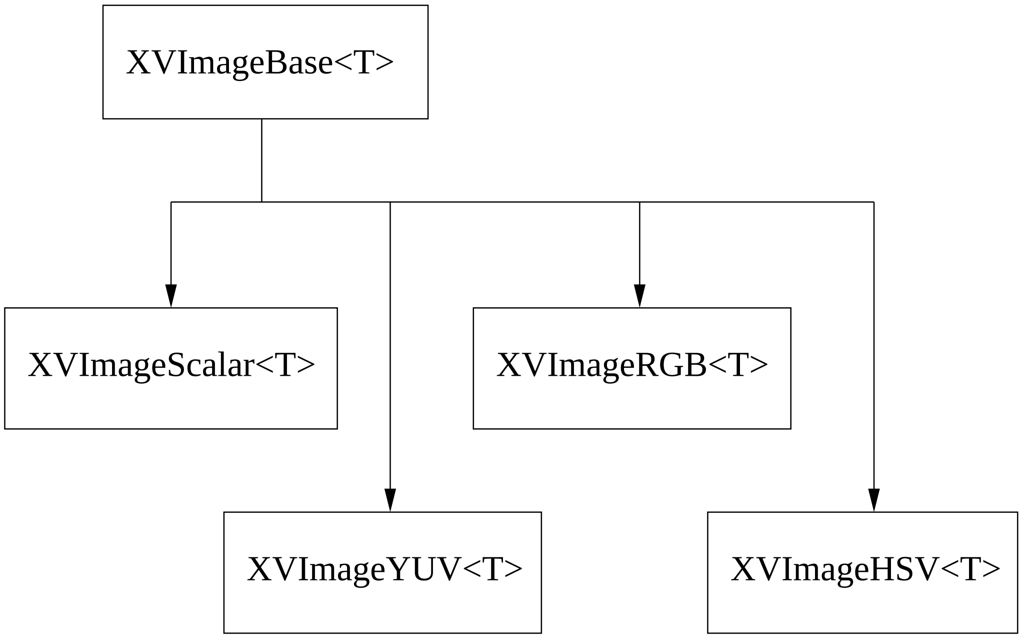

Figure 2.1: Diagram showing combination of inheritance and templating in image design. The base class: XVImageBase<T>, is templated on the pixel type, as are the inherited classes.

2.3 The Pixmap

Focusing on a region of interest (ROI) of an image or acquiring a sub-image are one of the most frequent operations in computer vision algorithms. Segmentation and tracking are two examples where focusing on a specific part of an image is necessary. In many applications (such as edge finding), there are a large number of small regions in an image that are processed. For these reasons, the structure of images in the XVision2 library needed a simple and fast mechanism for getting a specific region of an image. The idea of a Pixmap, which maintains the pixel data separate from the image itself (and which the image refers to), allows for such a mechanism.

The design of the Pixmap, consists of a simple container that holds the pixel data, and other initial image information (such as image dimensions). Such an abstraction allows for multiple images to access the same pixel data maintained by the pixmap, or even subsets of the pixel data (regions of interest) without performing any pixel duplication. The previous class diagram is now shown in Figure 2.2 updated with the pixmap framework. A simple locking mechanism allows images to modify their

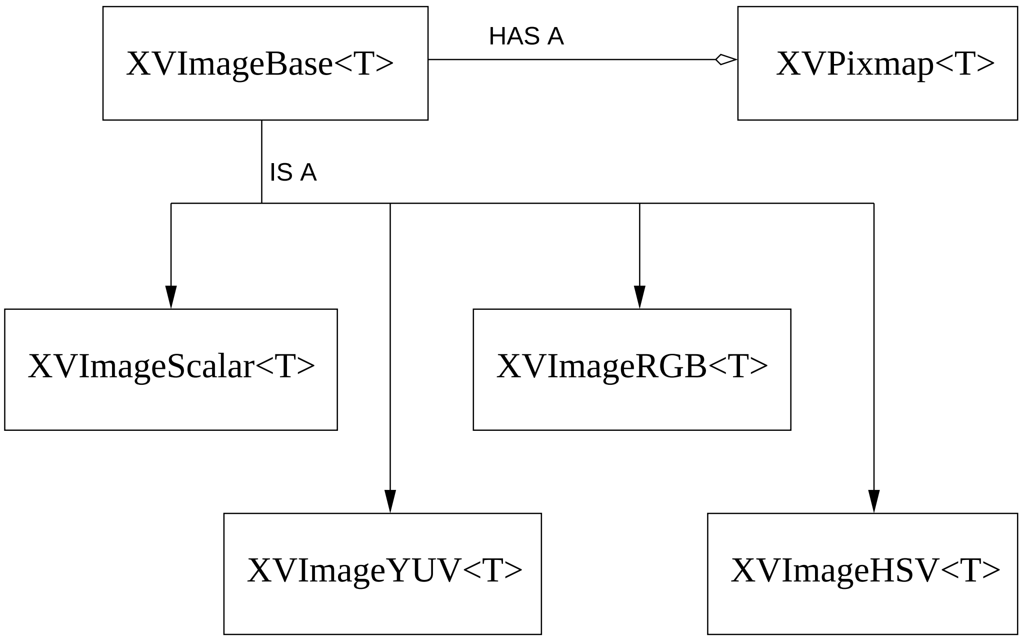

Figure 2.2: Diagram showing Image structure with Pixmap. Each image references a pixmap which may be unique or may be referenced by other images.

pixel data (i.e. the pixel data of its referring Pixmap). Preventing multiple images referring to the same Pixmap from modifying the same data simultaneously. Once an image has a lock or handle to the Pixmap, other images referring to that Pixmap cannot modify it until the first image unlocks the Pixmap. Refer to Figure 2.3 for further illustration.

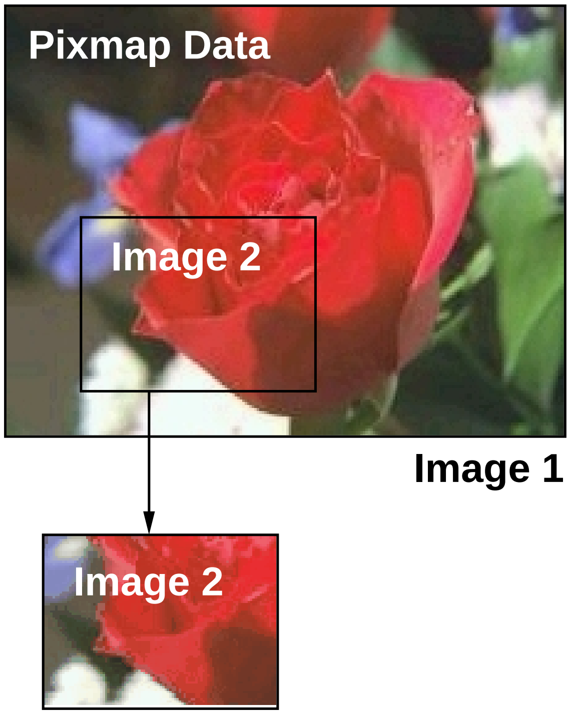

Figure 2.3: Diagram showing Pixmap and Image design. Image 1 and Image 2 have the same underlying Pixmap

2.4 Image Iterators

Another important aspect to the design of images in a vision library is the ability to access the contents of the image (pixel data) quickly and easily. Vision algorithms often need to iterate through an image a pixel at a time, or a row at a time. Due to the underlying pixmap design of XVision2 , we found it important to include a framework for stepping through an image pixel by pixel, where position information (column and row position) is maintained internally. Also, due to the underlying pixmap structure of images, it was necessary that iterators have both writable (allow modification of the image contents) and read-only properties.

Two types of iterators have been implemented as a part of the XVision2 framework:

Basic Iterator A basic iterator is templated on the pixel type (just as the image) and has a similar interface to the STL iterators in C++. The `* operator` provides access to the pixel value, while the prefix and postfix `++ operator`s increment the iterator to the next pixel in the image (wrapping the iterator to the beginning of the next row if at the end). Determining whether the iterator has reached the end of a row and must wrap to the next one requires a check at each increment, but this is done implicitly inside the iterator framework. The `end()` function provides a boolean value of true if the iterator has reached the end of the image.

Row Iterator For many applications, iterating through the image can be done one row at a time. A row iterator provides an interface to this type of looping structure. The row iterator checks for the end of a row using the `end()` function, and a `reset()` function performs the wrapping of the iterator to the next row in the image. This provides an implementation that allows for iteration from an arbitrary point in an image (not always the same column at each row) to the end of the row.

2.5 Image Operations

Included in the XVision2 library, exist some of the basic operations that are frequently performed on images. With the design goals of the library, namely to perform tracking in real-time, the image operations described here have each been implemented with efficiency and performance in mind. The operations on images have been grouped into four types: Filtering, Algebra, Warping, and Sampling. The first two: Filtering and Algebra, are available only for scalar images (of the `XVImageScalar<T>` type), as mathematical operations (such as addition) performed on multiband pixels is undefined. The last two operations: Warping and Sampling, do not modify the pixels themselves, and thus can be performed on any image type (both scalar and multiband). Descriptions of the operations with examples of their use are provided below.

2.5.1 Filtering

Filtering of images is a common step in any vision application, especially before further image processing and feature finding is performed. The XVision2 library implements a set of filtering functions, all of which are considered fast filters (and thus relatively simple) in an effort to maintain the real-time vision and tracking capabilities of the library. A generic `convolve` function also provides the end user with the capability of performing filtering with any mask necessary. The list of convolutions available in the XVision2 library with efficient implementations are as follows:

Box Filters - perform addition of all the pixels within the mask (including optional normalization). The box filter is often used for fast and simple smoothing. Since all the values within the mask of a box filter are 1, A separate implementation of the box filter improves performance, because all the values within the mask are 1, so no multiplications are required. The mask is also separable, so filtering in the x and y directions improves performance over one multidimensional filter. An Example of the Box Filter being used is shown in Figure 2.4. For efficiency, the functions provided allow the algebraic form to be used, where a new image is created implicitly. The in-place form for filters also exists, where the resulting image is created externally, and passed in as a parameter to the static filter function. Both forms are shown in 2.4.

Prewitt Filters - perform derivative filtering over the image by subtracting adjacent pixels, or pixels within a certain width. Prewitt Filters are also separable, and so have a similar implementation design as Box Filters. For a filter with a width greater than 2, many of the values in the filter's mask are zero, so a separate implementation which does not perform useless multiply by zero operations improves efficiency. Since

```
XVImageScalar<int> exIMG;
XVImageScalar<int> resultIMG;
...

// form where result image is passed in
XVBoxFilterX(exIMG, 3, resultIMG);
XVBoxFilterY(exIMG, 2, resultIMG);

// form where result image is create implicitly
resultIMG = XVBoxFilterY(XVBoxFilterX(exIMG, 3), 2);
```

Figure 2.4: A Box Filter Example. The conglomerate box filter in both the x and y directions is equivalent to using a box filter with 3 rows and 2 columns.

Prewitt Filtering is frequently used for edge detection in either the x or y directions, the example shown in Figure 2.5 is done only in the x direction.

```
XVImageScalar<int> exIMG;
...
XVImageScalar<int> resultIMG;

resultIMG = XVPrewittFilterX(exIMG, 2, 4);
```

Figure 2.5: A Prewitt Filter Example. This example filters the image in the x direction with a mask width of 4. The second parameter is a normalization value.

Roberts Filters - another variation of computing image gradients. Although the Roberts Filter isn't separable, only half of the values in the mask are non-zero, so performance can be increased with a separate implementation. An example of the Roberts filter being used is shown in Figure 2.6.

```
XVImageScalar<int> exIMG;
XVImageScalar<int> resultIMG;
...
XVRobertsFilter(exIMG, resultIMG, 0, ROBERTS_X);
```

Figure 2.6: A Roberts Filter Example. The parameters are the image to be filtered, the resulting image, a normalization factor, and the direction

2.5.2 Algebra

In a tracking library, when one is dealing with sequences of frames in the temporal domain, comparison of adjacent frames (or frames a few steps away) is often essential. The implementation of simple mathematical operations on images, such as addition and subtraction, and comparison operators, such as greater than and less than, provide a framework for these comparisons. With images represented as objects in XVision2 , the features of operator overloading in C++ allow for these operations to be used simply and easily. Examples of the different types of operations that can be performed on a pair of scalar images (of equal size) is shown in the example (Figure 2.7).

```
XVImageScalar<int> imgA, imgB;
...
XVImageScalar<int> imgSUM = imgA + imgB;
XVImageScalar<int> imgDIF = imgA - imgB;
XVImageScalar<int> imgGT  = imgA > imgB;
XVImageScalar<int> imgLT  = imgA < imgB;
```

Figure 2.7: An Example of the operations that can be performed on scalar images.

2.5.3 Affine Warping

Many algorithms in vision applications requiring warping of acquired images either by rotating, scaling, or sheer (affine transformations). XVision2 provides a simple mechanism for doing image warping efficiently and easily, using an algorithm similar to the Bresenham algorithms used for fast line rendering [6, 5].

To solve boundary issues involved with warping, XVision2 performs warping from the target to the source. As in Figure 2.8, warping consists of stepping through the target image, computing the nearest neighbor in the source image for each pixel.

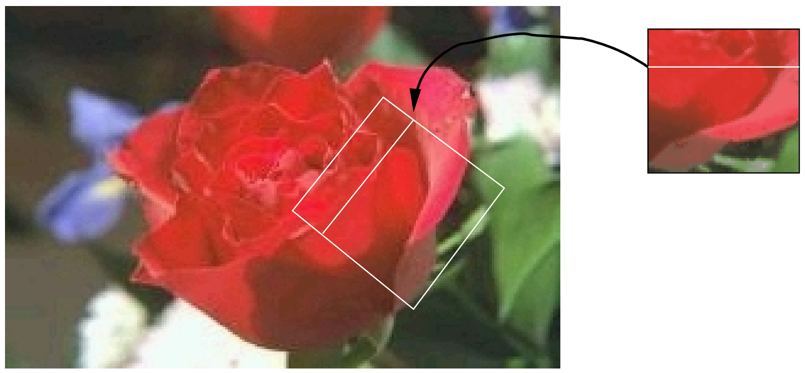

Figure 2.8: Figure of warping taking place. The image on the right is the warped image, and the source is the outlined rectangle on the left.

An affine matrix is computed initially, before warping is done. This matrix is then used in computing the positions of the pixels in the source image.

$$\begin{bmatrix} sx \\ sy \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} tx \\ ty \end{bmatrix} \tag{2.1}$$

Since warping is done from the target image, one row at a time, a full matrix multiply for each pixel is unnecessary. Instead, an offset is precomputed from equation (2.1) at the beginning of each row of the target image. Then for each pixel in that row, the offset is updated (for which only two additions are required per pixel) [5].

Warping is implemented as a warping class (`XVAffineWarp`), which maintains the warping matrix. For warping of many images equally (using the same affine matrix), this class should be used directly. Alternatively, a set of static functions exist for ease of use, which can be called on any image type. An example of full affine warping (rotation, scale, and sheer) is given in Figure 2.9.

```
XVImageRGB<XV_RGB> exIMG;
...
XVImageRGB<XV_RGB> resultIMG;

resultIMG = warpRect(exIMG,
             XVPosition(exIMG.Width() / 2, exIMG.Height() / 2),
             XVSize(exIMG.Width() / 4, exIMG.Height() / 4),
             30, 2, 2, 1);
```

Figure 2.9: Example of Image Warping. The parameters to the warping function are the image to be warped, the center of the warping region, the size of the warping region, the angle (30^o), scale factors x and y (both 2), and a sheer of 1.

2.5.4 Sampling

For real-time tracking, the standard frame size (640×480) is often larger than necessary, and decreasing the size of the image by a factor, may not decrease accuracy. Tracking of straight edges for example, does not decrease the accuracy of the tracker by down-sampling in the direction of the edge (for a vertical edge in the image, this would be down-sampling the rows), unless the edge has a length of only a few pixels. Down-sampling also decreases the size of the image, thus down-sampling before performing feature detection improves performance. A set of static functions provide fast down-sampling of images in both the x and y directions. An example of down-sampling is shown in Figure 2.10.

```
XVImageRGB<XV_RGB> exIMG, resultIMG;
...
resample(0.5, 0.25, exIMG, resultIMG);
```

Figure 2.10: A Down Sampling Example. The image is being down-sampled in the x direction by a factor of 2 (every other row), and in the y direction by a factor of 4.

2.6 Image Conversion

2.6.1 Implicit Conversion

Due to the large number of different image formats available, converting between different formats is unavoidable. For a specific application, the video source may return YUV, from which conversion to grayscale is necessary for image processing and any computer vision algorithms, and finally to display the images, they are converted to RGB. The many image format conversions that are often required add clutter to the algorithm's code, making them difficult to write and debug. In order to prevent this clutter, the image classes were designed with the ability to convert automatically from one format to the other using the polymorphic properties of the C++ class structure. Image format types are grouped first by color and grayscale. An `XVColorBase` class represents all the different color images. `XVColorImage` acts as an un-templated interface to the color classes (including `XVColorBase`) and provides conversion functions for each of the image formats available. This allows for implicit conversion of one format to the other using the features of the C++ language, as shown in Figure 2.11.

2.6.2 Explicit Conversion

Implicit conversion of images from one type to another moves the demands of knowing when and how to convert image types from the end user of the library, to the library itself. Similar to programming languages which include the features of

```
// define the image objects

XVImageRGB<XV_RGBA32> rgbIM;
XVImageYUV<XV_YUV24>  yuvIM;

//  initialize the rgb image here
//  ...

// do the conversion!

yuvIM = rgbIM;
```

Figure 2.11: Code required to convert from an RGB image to a YUV image

an automatic garbage collector, there are disadvantages to the implicit conversion design, specifically: images may be converted from one type to another without the end user's knowledge, or even worse, against their expectations. Although the design of the conversion feature attempts to avoid those unwanted circumstances, as with most features that provide automatic functioning of expected behavior, not every scenario can be handled.

Also, the conversion operations performed by the implicit form are generic enough to allow for conversion from any image type to any other type. Thus, an efficient algorithm for converting between two specific types cannot be utilized during implicit image conversion.

Noticing the drawbacks of implicit conversion, we provided explicit conversion functions for many pairs of image types, which provide the most efficient implementations for converting between those pairs. An example of explicitly converting from an RGB image to a scalar image using the red band of the image (instead of costly addition of all three RGB bands and normalization) is shown in Figure 2.12.

```
XVImageRGB<XV_RGB> rgbIMG;
XVImageScalar<int> scIMG;
...
RGBtoScalar(rgbIMG, scIMG, XV_REDBAND);
```

Figure 2.12: An example of explicitly converting from RGB to scalar using the red band of the RGB image.

Chapter 3

Tracking

3.1 Introduction

There are many branches of computer vision. The main purpose of XVision2 is to be a framework for dynamic vision and real-time tracking, especially of complex image features, such as faces, or other objects such as obstacles in an autonomous mobile robots environment[1]. XVision2 is designed to handle tracking of features from images in a straightforward manner. In this chapter, we detail the design of the Tracking framework in XVision2.

3.2 Simple Features

The design of the tracking framework pivots around the notion of features. A *feature* in XVision2 is any locally detectable part of an image that can be tracked over a sequence of frames [13]. Each feature contains a *state*, which represents the feature's position at a given point in time. The state of a feature can be its position in an image, its size, an accuracy measure, or any possible combination of parameters. A *feature tracker* is designed to determine the next state (a *step*) of a feature, given as inputs its previous state, and the next image frame. Refer for Figure 3.1 for a

[1]XVision2 was used as the vision framework for the Johns Hopkins Robocup Team, which participated in a mobile robots soccer competition in August of 2001

diagram of the `feature` design.

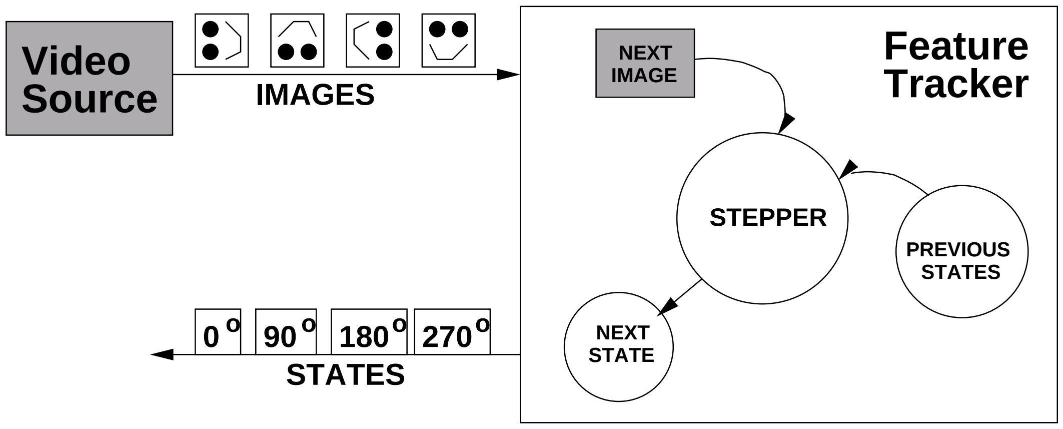

Figure 3.1: A diagram of the design of a *Feature Tracker*. The *STEPPER* in the diagram represents the computation required to determine the *next state* from *previous states* and the *next image*.

As images are fed to the feature tracker (also referred to as the `XVFeature<T>` class), and states are accumulated, previous states can be used with greater or lesser weighting to determine future states using a simple interpolation model, or a more sophisticated optimal estimation algorithm (also known as Kalman Filtering or Tracking) [13].

Some of the more important design goals of the feature framework are as follows:

- A feature always updates its state using values from previous states. Initially, the previous state (or in this case, the initial state) of the feature has to be set before tracking. The initial state of the feature can be set manually (using a known or predetermined state), or can be initialized interactively. Often the initial state of a feature is unknown at compile time, and can only be determined at run-time. For example, to track a face in a video sequence, the initial images can be run through a global search algorithm for facial features to find the most likely region of the image to be a face. Alternatively, features in XVision2 are designed to provide the mechanism of outlining (with the computer mouse) the face (or some other object) in the first few images of the sequence. Due to the design of the Console module (used for display of images) in XVision2 adding interactive initialization to a feature is straightforward. See the Appendix on XVision2 Consoles for further information.

- In computer vision, the proof that a tracking algorithm works is often gained visually, as much as numerically. This visual proof consists of outlining or drawing the tracked region on an image or console. XVision2 abstracts the drawing of a feature to the feature class itself. This design allows many features that simultaneously being tracked to be drawn simply by calling the show function of those features.

- For many tracking algorithms, warping (or de-warping) of the image before feature extraction is often necessary. The feature framework in XVision2 provides a warping function that performs any necessary warping of the image. The default behavior does nothing.

- Feature design includes maintaining and updating previous states. In order to compute future states of the tracked feature, a `feature` encapsulates previous states, allowing for easy access and maintenance.

- Computing the next state of the tracked feature, is the most important aspect of the feature framework. This is done through the step function, which provides a formalized method for computing the next state.

- The feature framework also provides performance mechanisms that determine the accuracy of the newly computed states. Each state has an associated error value, which provides a measure for the accuracy of that state.

3.3 Feature Implementation

The feature class `XVFeature<S, I>` is an abstract class templated on the state type (`S`) and the image type (`I`). This allows a feature to be easily implemented for all types of images, and all types of features. Often, implementing a feature only requires extending the `XVFeature<S, I>` class and overriding the `step` and `warp` functions. The `XVFeature<S, I>` class definition is shown in Figure 3.2.

3.3.1 Common Feature Types

There are a few feature types that are common enough to be implemented in the XVision2 library. These include generic blob features (such as color or motion), edge features, and features based on intensity differences (commonly known as sum-of-squared differences) [3].

Blob Feature The blob feature implements a simple segmentation based feature, such as color or motion segmentation. The `XVBlobFeature<I, O>` class extends from the `XVFeature<S, I>` class, and is templated on the input image type (before segmentation), and the output image type (after segmentation).

SSD Feature The SSD feature brings back a previous feature tracker used in XVision. The `XVSSDFeature<I, S>` in XVision2 is implemented with the standard inheritance relationship from `XVFeature<S, I>`, containing also an additional class: `XVSSDStepper<I, S>`. The "stepper" provides flexibility for the SSD feature, because it allows the SSD feature to have multiple types of warping and updating, all within the same class. The possible types of warping are shown in Figure 3.3, and include simple translation, SE2 (translation, rotation, and scale), and full affine warping.

Edge Feature provides simple edge detecting capabilities for arbitrary edges. The `XVEdgeFeature` class is derived from the `XVFeature` class and implements tracking techniques similar to those in [5].

The basic class diagram of the tracking framework is shown in Figure 3.4.

3.4 Complex Features

The structure of the feature framework using the inheritance properties of object oriented design allows us to build complex conglomerations of features more easily. Such a complex feature would need only to inherit the abstract `XVFeature<S, I>` class and maintain as parameters each of the more simple features it consists of.

Stepping through a frame and updating the state of the complex feature only requires stepping through each simple feature, and determining from the results the state of the complex feature. Examples of complex features and the simplicity of their implementation will be demonstrated in the following chapter.

3.5 Conclusion

While the tracking framework provides most of the necessary basic features required for simple tracking, a larger set of features including a generic corner feature consisting of multiple edges, and a curve feature using polynomial interpolation might prove useful tools for many vision applications.

```
template <class IMTYPE, class STATETYPE>
class XVFeature : public XVNode {

 protected:

  STATETYPE prevState;
  STATETYPE currentState;

 public:

  typedef STATETYPE STATE;
  typedef IMTYPE IMAGE;

  XVFeature() : warpUpdated(false) {}
  XVFeature(STATE init) :
    currentState(init), prevState(init), warpUpdated(false) {}

  virtual void initState(STATE init) {
    currentState = init; prevState = currentState;
  }

  virtual IMTYPE warp(const IMTYPE &) = 0;

  virtual const STATE & step(const IMTYPE &) = 0;

  virtual const STATE & step(const IMTYPE & im, const STATE & st){

    currentState = st;
    return this->step(im);
  };

  virtual const STATE & getCurrentState(){ return currentState; }
  virtual const STATE & getPrevState(){ return prevState; }
  virtual STATE diffState(){ return currentState - prevState; }

  virtual const STATE &
  interactiveInit(XVInteractive &, const IMTYPE &) = 0;

  virtual void show(XVDrawable &) = 0;
};
```

Figure 3.2: The `XVFeature<S, I>` class definition

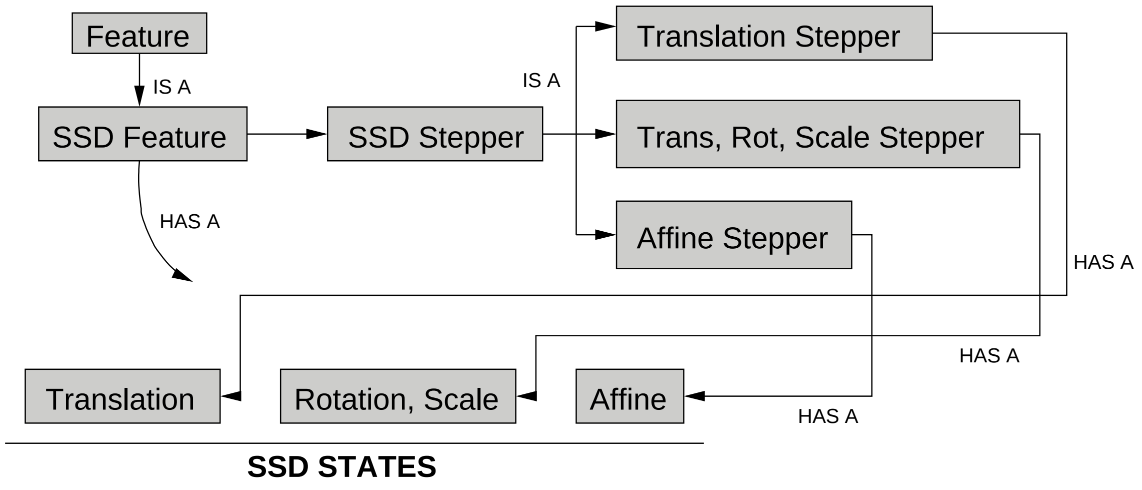

Figure 3.3: SSD Feature Class Diagram, demonstrating the role of the stepper in the framework, and the flexibility it provides

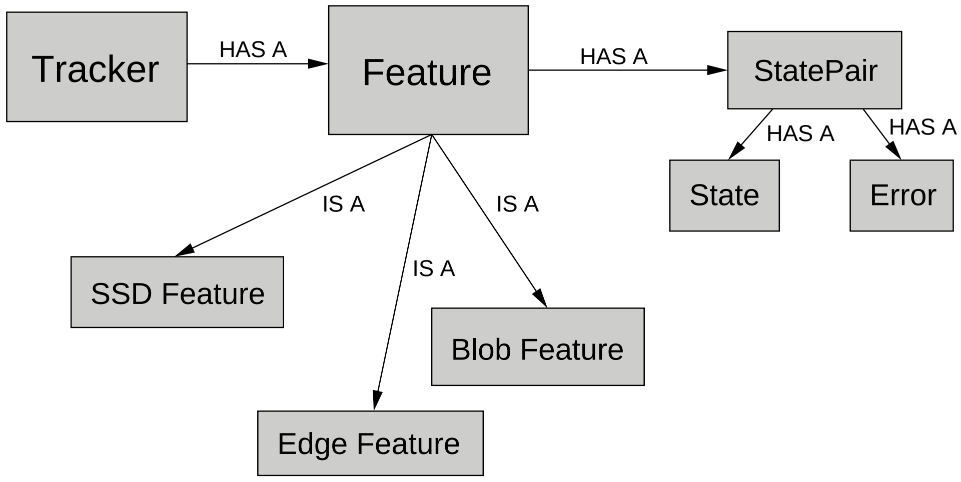

Figure 3.4: Simple Class Diagram of the Tracking Framework for XVision2

Chapter 4

Applications

4.1 Introduction

Recently, computer vision has been applied to many areas of the medical field, specifically in the area of surgery. We provide two examples of the uses of computer vision for two separate medical applications, both within the microsurgical domain. Not only do both examples use XVision2, but are good examples of the unique features and design of the library.

4.2 Example 1: Stapes Tracking

An example of using and extending the XVision2 library is in the tracking of a tiny stapes prosthesis while it is being crimped to the incus bone, a tiny bone in the inner ear. This procedure attempts to immobilize the stapes bone in the ear, which allows partial or full recovery from hearing loss. Due to the microscopic nature of the procedure, a robot is used to augment the actions and motions of the surgeon. To prove that using the robot improves accuracy and dexterity of the procedure, a set of microscopic cameras image the operating area. Vision software written with XVision2 monitors the position and movement of the prosthesis during a set of trials on an artificial ear [7]. Images from the tracked prosthesis are shown in Figure 4.1.

4.2.1 The Stapes Tracker

The features found to be most easy to track in the prosthesis were the left and right vertical edges, the top edge of the prosthesis, and the wire. This was not simply a matter of tracking edges, but of tracking a conglomeration of edges together. Due to the modularity and simplicity of the `XVFeature<S, I>` and `XVTracker<S, I>` classes in XVision2, we were able to extend these classes and add a few specifics to perform tracking of the prosthesis. A diagram of the different feature types used, and how they were put together is shown in Figure 4.2. First, a Stapes class was designed, consisting of the different components of the object to be tracked. The declaration for this class is shown in Figure 4.4. As shown in Figure 4.2, the Stapes class consists of a positions (x and y) and the orientation of the prosthesis. It also maintains the individual feature states: the left and right edges of the prosthesis, which are used for determining the horizontal position; the top edge of the prosthesis, which is used for determining the vertical position; and the wire orientation, keeps the orientation of the entire stapes feature aligned with the prosthesis itself. The Stapes class was then used as the template type to the `StapesFeature` class, shown in Figure 4.3. This is a relatively small and simple class to write, demonstrating the ease of writing even a complex tracking framework in the XVision2 environment. The vector of edge features is the list of basic features, consisting of the left, right and top edges, built as MaxEdges, which maintain perpendicular position only, no orientation. The wire edge or strip, is built as a Strip Edge, which determines both orientation and position. The `step()` function of the `StapesFeature` class consists of iterating through these simple edge features, and calling *their* `step()` functions. The results are then sent to the `Stapes` state class which computes its own state from the individual states. The step function is shown in Figure 4.5.

Building a tracker using this bottom-up approach based on combining the simplest possible features to form more complex features, provided us with the ability to perform tracking in a near real-time environment, whereas imposing geometric constraints on the problem in a top-down fashion often results in a highly robust but impossibly slow tracking application.

4.2.2 Results

The 2 sets of 2D positions of the wire-prosthesis intersection point were tracked and logged over the sequence of a trail. With this data, estimates of the normalized 3D positions computed using an Affine Camera model were found. The resulting plots of a few such trials are shown in Figures 4.6, 4.7, 4.8, 4.9. In the plots of Figures 4.6, 4.8, the tracked X and Y positions in the images are shown over a trial's sequence. The upper plot shows the tracked positions from the left hand view, and the lower plot shows those from the right. In both the plots of Figures 4.7 and 4.9, the normalized X, Y, Z positions over the sequence of a trial are shown from the top down (X at the top, Y in the middle, Z at the bottom). The plot of the 3D positions reveals a number of things. There are long intervals during which the position does not change. This is verified by watching the video sequences and by the 2D plots, where the prosthesis moves in spurts, and the position changes are dramatic rather than smooth. Also, by comparing the 2D plots with their 3D counterparts, correlations between the two can be seen. While the 3D positions are normalized and only rudely accurate, changes in the position of the prosthesis can be clearly seen, demonstrating the validity of the prosthesis tracking algorithm described here.

4.3 Example 2: Virtual Fixtures and Path Following

Just as in the previous example, this example uses a "steady hand" technique to improve performance at the microscale. Tests have suggested that augmentation of a human's performance at the microscale has improved the speed and efficiency of vitreoretinal microsurgery [8]. Further work with virtual fixtures on the Johns Hopkins University Steady Hand Robot (JHU-SHR) have shown that using different levels of compliance also improves performance of tasks such as path following at the microscale [2].

To demonstrate this, a micro-camera (or an endoscope) was attached to the JHU-SHR, which provided a video feedback to the current position and orientation of the JHU-SHR[1]. Figure 4.10 shows the setup. As a path following experiment, the project required a vision framework that would track lines and curves (such as blood vessels on the retina) and determine the orientation, position, and size of such features [10, 9, 2, 5]. The XVision2 library provided much of the implementation required for this project already, but due to microscale issues, the lines and curves to be tracked in the video images often varied in width and intensity. To have a robust tracker, two edges oriented in opposite directions were combined to have a strip tracker, which tracked the borders of a strip, line, or curve. The two edge features had to be coupled to keep them from drifting apart. Accuracy could be further improved by determining the best responses from the two edges and updating both accordingly. The two images in Figure 4.11 show the tracked curve. The image on the left is taken from an image sequence provided the endoscope camera. The image on the right outlines the tracked feature. The overlayed lines on the edges of the thicker line and the one in the middle show the positions of the two basic edge features. The cross in the center of the image refers to the position of the center of the image (assumed to be the center of the endoscope and hence of the robot). The short lines extending from the center of the cross are the error distance of the curve from the center, and the orientation of the curve.

Due to the requirements of the robotic controller, the performance of the tracker was increased to twice the standard rate (60 Hz). Due to the tracking framework designed, the tracker was able to maintain these performance requirements.

4.4 Conclusion

Here we show that for many vision applications, a tracking framework which can perform in a real-time environment is often necessary. The design of complex features built from the bottom-up with a set of basic features allows for real-time tracking with

[1]The setup is commonly known as eye-in-hand in robotics

robust results. For both of the examples demonstrated in this chapter, much of the work could have been extended. For the prosthesis tracking project, although tracking was demonstrated to be robust for most cases, accuracy may have been improved with more complex sets of features, such as tracking the corners of the prosthesis. Also, for this project, occlusion was not handled, which caused the tracked to get "lost" for some cases. The virtual fixturing tracking system was fairly robust, despite the blur in the images induced by the endoscope camera, and the frequent changes in lighting. Further performance may have been improved using curvature estimation to determine the precise position of the curve.

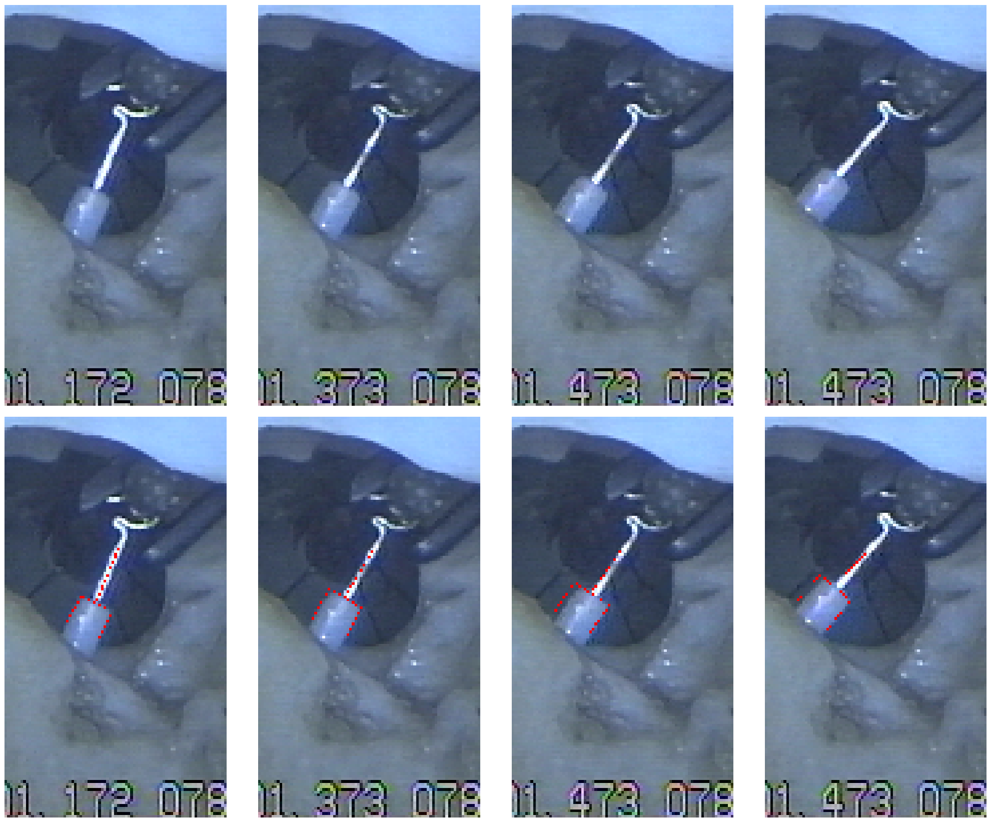

Figure 4.1: A sequence of images where the Stapes is being tracked. The upper row shows images directly from video. The rectangular object with the vertical wire is the prosthesis. The wire is crimped or tightened at the top during the procedure. The lower row shows images with the prosthesis being tracked

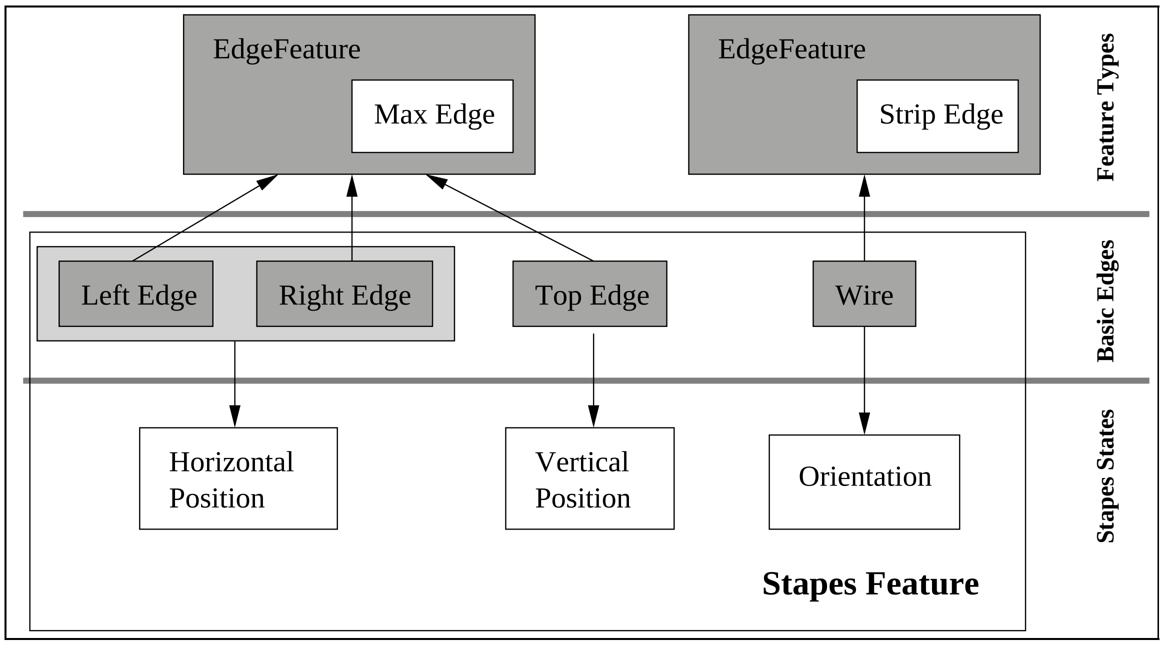

Figure 4.2: A class diagram of the feature types used for tracking the prosthesis.

```
class Stapes {

 protected:

  XVPosition center;
  double     angle;
  XVSize     regionSize;

  XVLineState * lines;

  XV2Vec<double> trackPoint;

 public:

  Stapes(XVPosition c, XVSize s, double a,
 int nS, int nPE, int nRE)
    : numStrips(nS), numPosEdges(nPE), numResponseEdges(nRE);
  ~Stapes(){ for(int i = 0; i < totalEdges; ++i);
  XV2Vec<double> findCenter();
  void updateState(vector<FEATURE *> &);
  void setEdgeFeatures(vector<FEATURE *> &);

  Stapes operator += (Stapes & s);
  Stapes operator -= (Stapes & s);
};
```

Figure 4.3: The Stapes Class definition

```
class StapesFeature : public XVFeature<XVImageScalar<int>,
       StapesState> {

  vector<XVEdgeFeature<IM_TYPE, XVEdge> > edges;
  XVSampleIterator<double> iter;

 public:

  StapesFeature(XVSampleIterator<double> i,
                vector<XVEdgeFeature<IM_TYPE, XVEdge > > e)
: iter(i), edges(e) {}

  virtual XVImageScalar<int> warp(const XVImageScalar<int> &);
  virtual StapesState step(const XVImageScalar<int> &);
};
```

Figure 4.4: The Stapes Feature Class definition

```
const StapesState &
StapesFeature::step(const XVImageScalar<int> & im) {

  if(!warpUpdated) this->warp(im);
  warpUpdated = false;

  vector<double> responses;
  XVLineState newWireState = edges[0]->step(warped);
  responses.push_back(newWireState.error);

  double tmpResponse = edges[1]->step(warped).error;
  responses.push_back(tmpResponse);

  XVLineState newTopState = edges[2]->step(warped);
  responses.push_back(newTopState.error);

  tmpResponse = newWireState.error;
  responses.push_back(tmpResponse);
  for(int i = 1; i < NUM_EDGES; ++i) {
    tmpResponse = edges[i]->step(warped).error;
    responses.push_back(tmpResponse);
  }

  currentState.updateState(edges);
  currentState.updateError(responses);
  return currentState;
};
```

Figure 4.5: The step function in the StapesFeature class

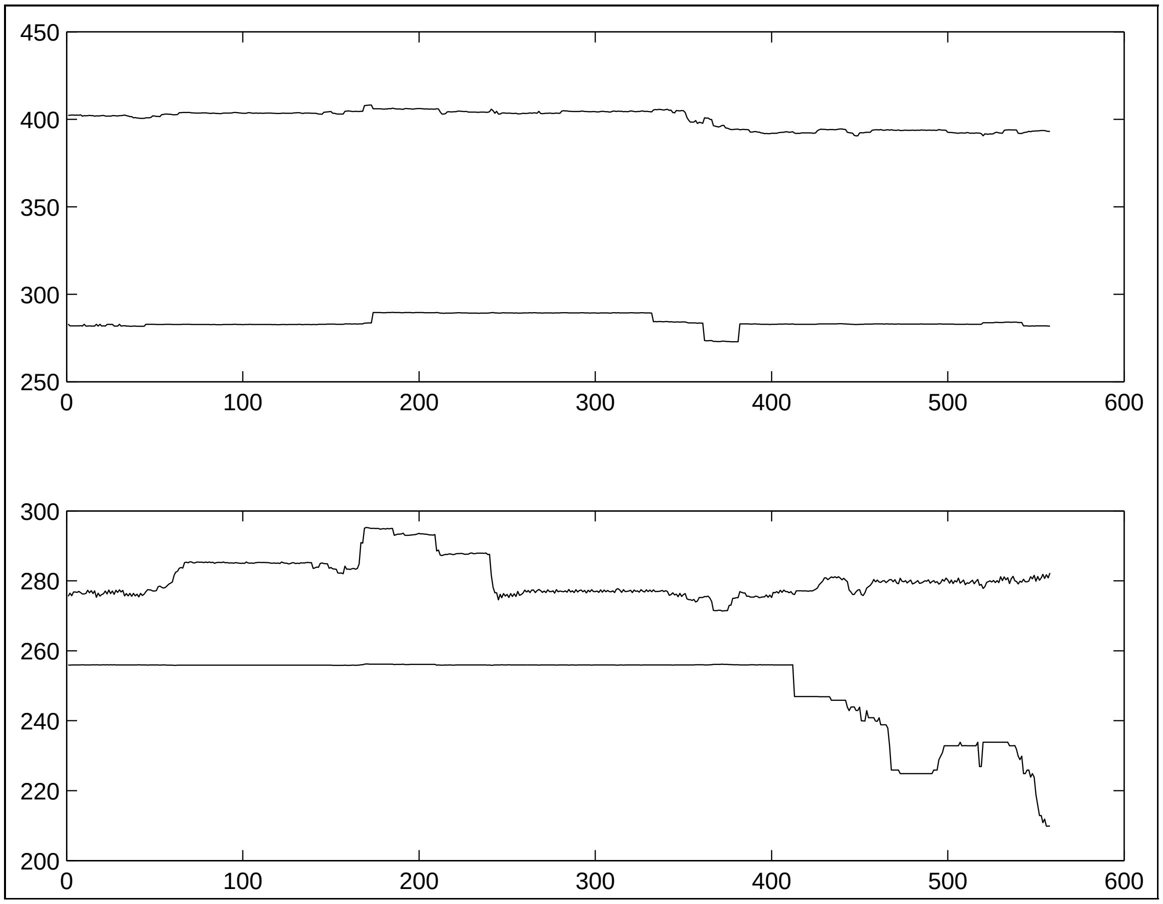

Figure 4.6: Plot of the tracked X and Y positions in the image. The sequence numbers extend along the x-axis, while the normalized positions of X and Y are shown alon the y-axis (X is above Y). The upper plot shows data from the left hand view, while the lower plot shows data from the right hand view. Data from Experiment LUSTIG-1H with outliers removed.

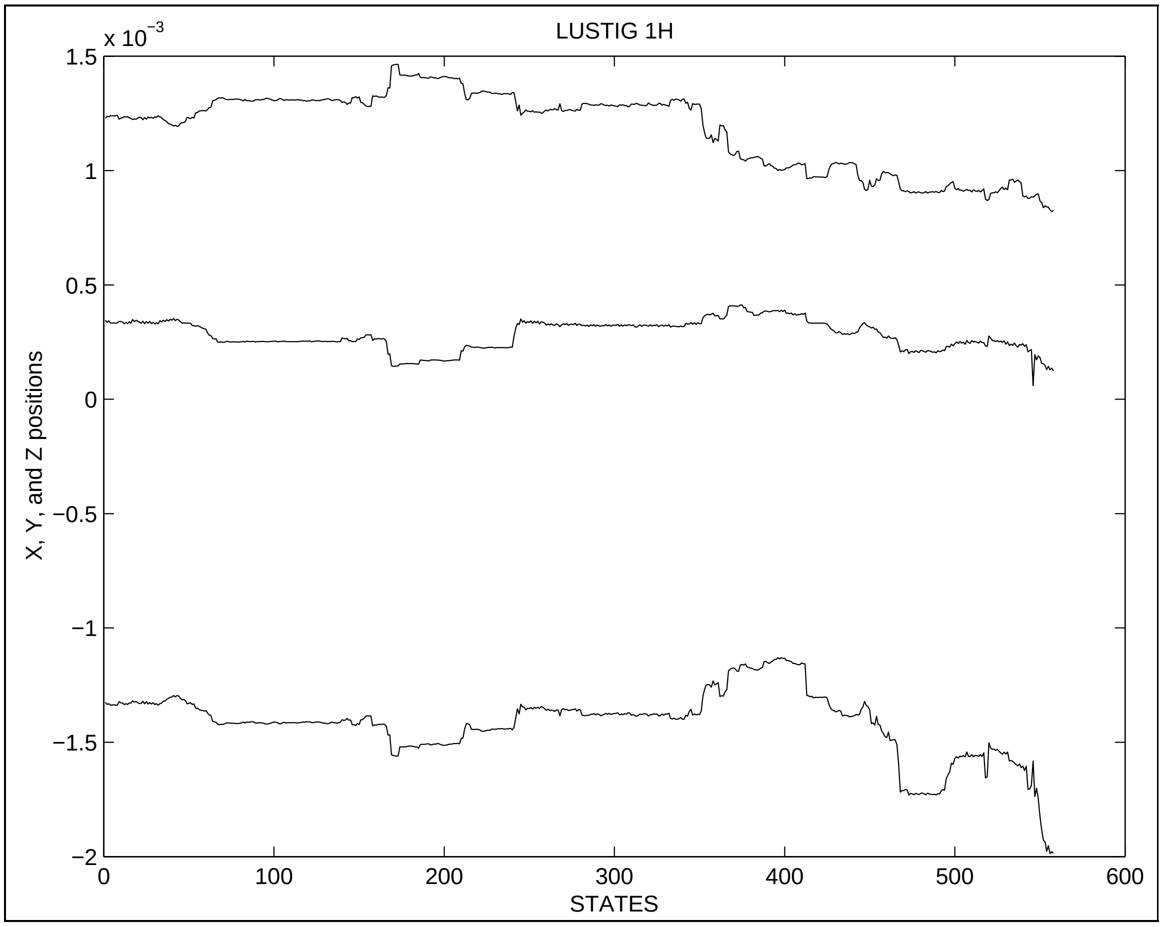

Figure 4.7: Plot of the estimated X, Y, and Z positions of the prosthesis intersection point. The sequence numbers (time values) extend along the x-axis, while the normalized positions of X, Y, and Z are shown from the top of the graph downward. Data from Experiment LUSTIG-1H with outliers removed.

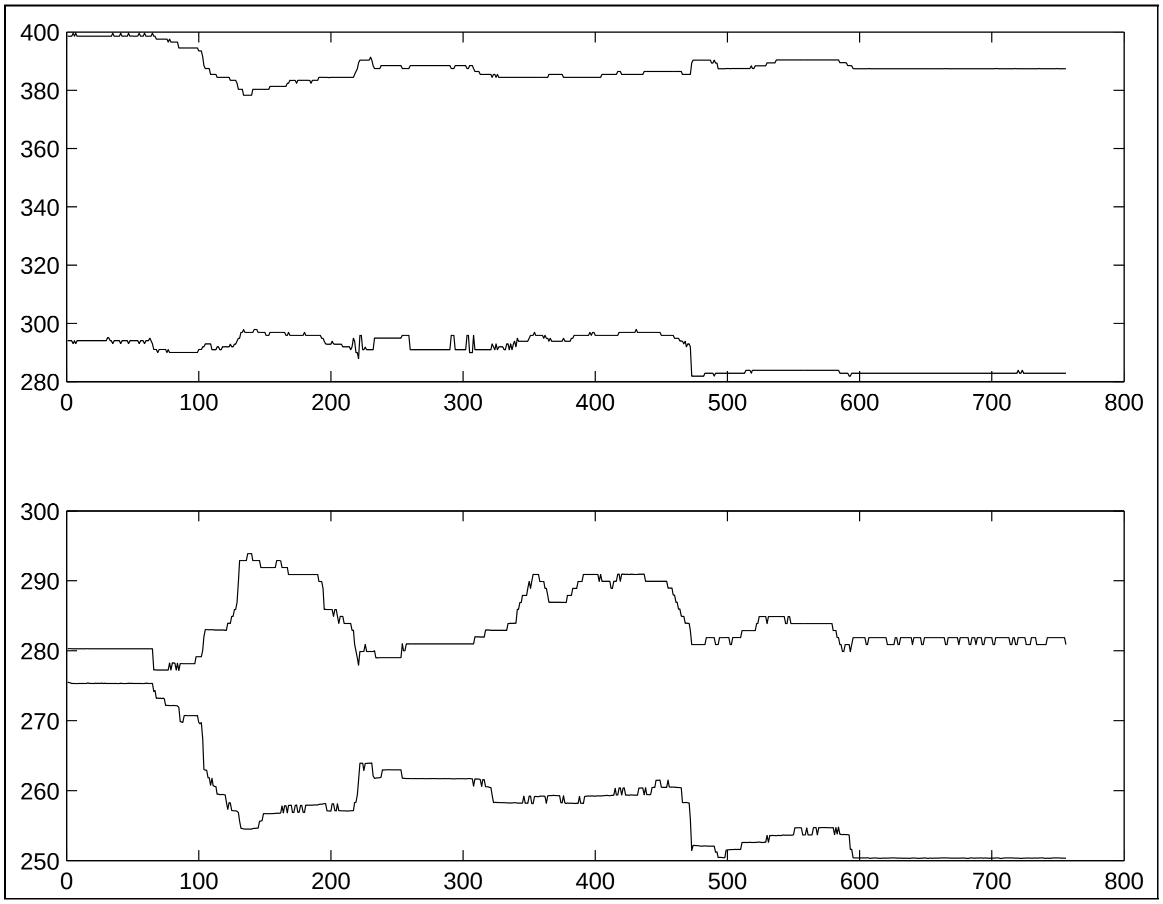

Figure 4.8: Plot of the tracked X and Y positions in the image. The sequence numbers extend along the x-axis, while the normalized positions of X and Y are shown alon the y-axis (X is above Y). The upper plot shows data from the left hand view, while the lower plot shows data from the right hand view. Data from Experiment LUSTIG-6R with outliers removed.

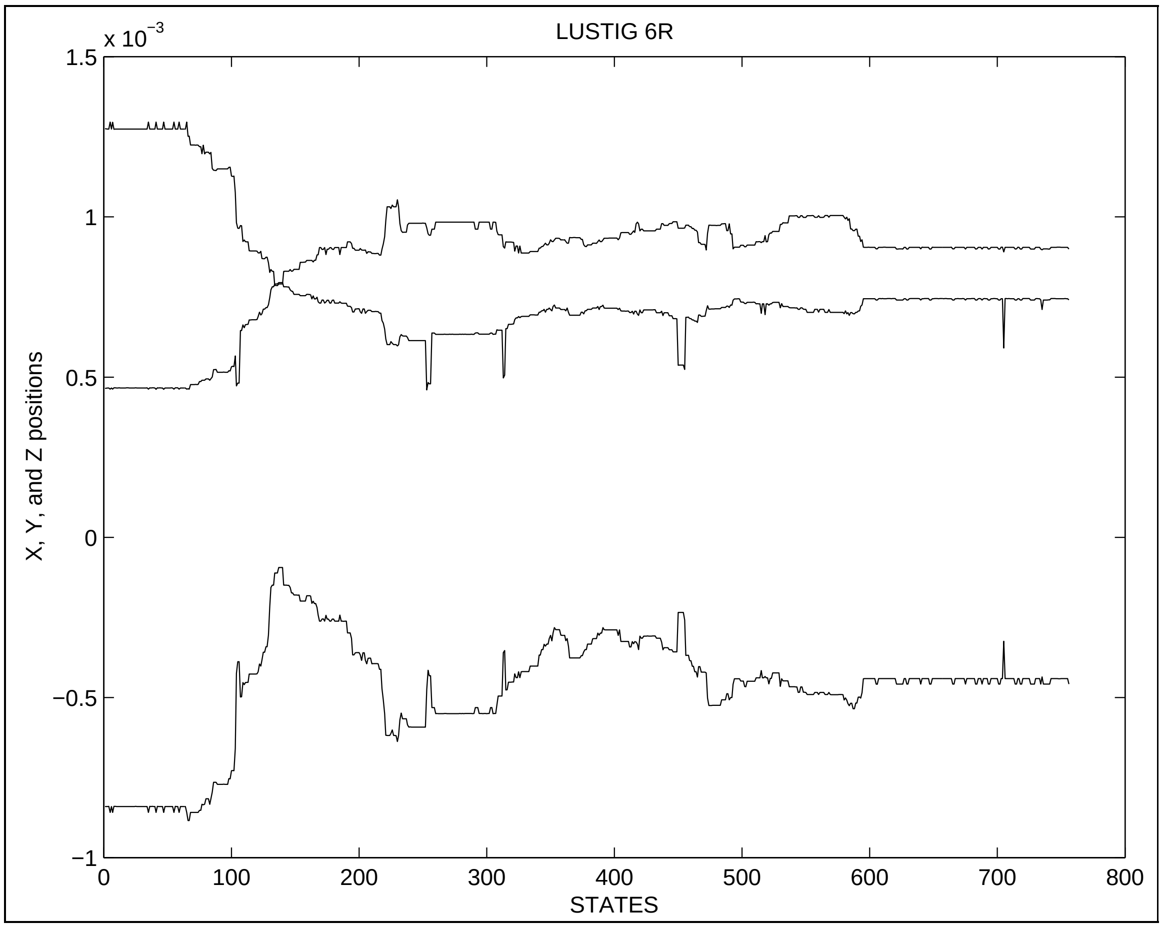

Figure 4.9: Plot of the estimated X, Y, and Z positions of the prosthesis intersection point. The sequence numbers (time values) extend along the x-axis, while the normalized positions of X, Y, and Z are shown from the top of the graph downward. Data from Experiment LUSTIG-6R with outliers removed

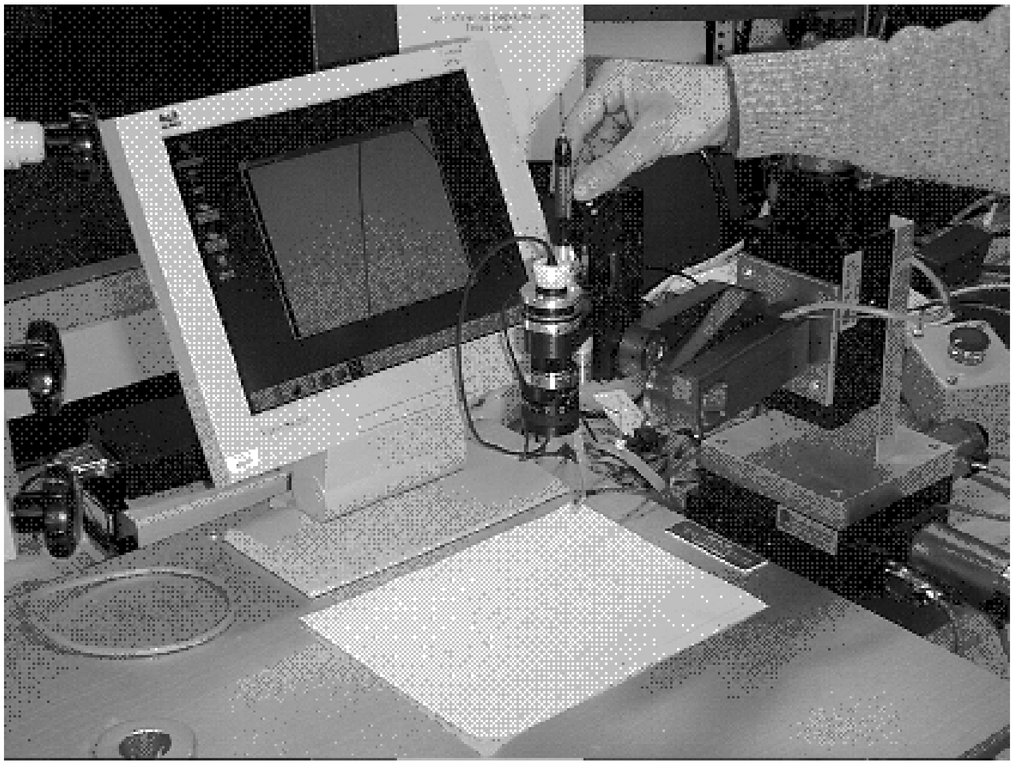

Figure 4.10: Shows the setup for the Virtual Fixturing experiments. The user holds the force sensor which controls the robot. Images from the camera attached to the force sensor are fed to the display

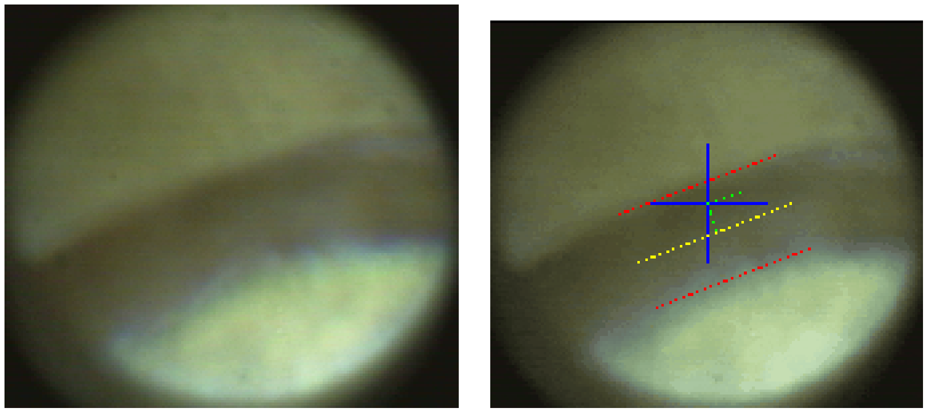

Figure 4.11: The tracked images of a curve.

Chapter 5

Conclusions

This document has described the design of the XVision2 library. Both the design and much of the implementation were completed in May 2001. A website where the library is maintained and further information is available, can be found at: `http://www.cs.jhu.edu/CIRL/XVision2/`.

XVision2 fulfills many of the design goals imposed upon it. Using features in the C++ language, such as parameteric polymorphism and virtual inheritance, the library has maintained modularity over its many different components. It provides support for many new video devices, and has a good mechanism for interaction with window displays. It also includes many of the features in the old XVision, especially its modular approach to feature tracking. One example of the library's robustness, is the use of the library for the R.A.P.I.D project [11], a project for implementing a functional programming framework for robotics based applications. It has also been used by the Johns Hopkins University's Robocup team, a soccer competition for mobile robots. The team used the library primarily for tracking using segmentation and color.

Drawbacks and Limitations

Although fairly robust, there are some drawbacks to the design of the library. As a real-time tracking library, speed is essential when thinking of design and implementation. Although the algorithms in the library are designed to be as efficient as possible, they do not take full advantage of the capabilities of current hardware, such as using MMX technology on Intel based processors for many of the image processing tasks.

Another drawback of the library is the use of templates. While this is one of the libraries strongest features, providing modularity and flexibility that would otherwise not exist, templates are often received with a negative response in the development community. Unfortunately, templates have been implemented differently by C++ compilers, despite their being included in the ANSI C++ standard. This is primarily why "Don't use Templates" is frequently found in many C++ style rules documents [14]. Thus, using templates reduces the portability of the XVision2 library, requiring compilicated pre-processing directives to conform to the many different C++ compilers. Further, C++ templates have a steeper learning curve compared to many of the other features of the language, and thus may reduce the ease of use of the library.

As previously mentioned, the notions of *features* and *states* developed into the design of the tracking framework, provide the ability to build complex features based on a set of primitives. The *state* of a *feature* provides a measure of its performance. Ideally, with such a rich framework for feature performance, the best configuration of feature combinations could be realized using a systematic approach. Unfortunately, feature performance is limited to the temporal domain, a comparison of a feature's current and past performance. Standard performance of a specific feature (such as an edge feature), which would allowing comparison of features of the same type or even different types, is not built into the tracking framework. The result is that most complex feature tracking development is performed in a somewhat ad-hoc manner, where constraints put on features are of a geometric form (what looks to be correct), instead of a reduntant set of features primitives where constraints are built not programmatically, but dynamically based on the performances of those constraints. Such a design would make life easy for the designer of a visual tracking system: instead of having to guess the best features to use and the constraints to put on them, he would only be required to localize the structure of the object, and let the tracking framework do the rest. While this is considered current research, building such a dynamic tracking system into XVision2 would provide the library with a greater robustness and ease of use.

Future Work

Future work for the library would include adding a platform for image processing using the previously mentioned MMX technology. Also, wrappers for other programming languages (such as the wrapper to Haskell, which is currently being developed) would allow the library to reach a broader group of software developers. An interface to the OpenGL library would provide fast hardware support for 3D rendering as well as image display.

Most importantly, a wider set of feature tracking primitives (such as a corner feature) would further the robustness of the library and provide a wider set of tools to build on for development of dynamic vision applications. The afore mentioned dynamic tracking framework will become essential to a library of this type, and its inclusion would help to advance the library toward its goals of short start-up times and ease of use.

Bibliography

[1] Documentation for XFree86. http://www.xfree86.org, 2001.

[2] A. Okamura A. Bettini, S. Lang and G. Hager. Vision assisted control for manipulation using virtual fixtures. Maui, Hawaii, October 2001. IROS.

[3] G. Hager and P. Belhumeur. Efficient region tracking with parametric models of geometry and illumination. *IEEE PAMI*, 20(10), 1998.

[4] Gregory D. Hager and Kentaro Toyama. *A Framework for Real-time Window-Based Tracking Using Off-The-Shelf Hardware*, August 1994. DRAFT Documentation Version 0.95 alpha.

[5] Gregory D. Hager and Kentaro Toyama. The xvision system: A general-purpose substrate for portable real-time vision applications. *Computer Vision and Image Understanding*, 69(1):23–37, 1998.

[6] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes. *Computer Graphics.* Addison Wesley, 1993.

[7] P.J. Berkelman, et al. Performance evaluation of a cooperative manipulation microsurgical assistant robot applied to stapedotomy. Utrecht, The Netherlands, October 2001. MICCAI.

[8] R. Kumar, G. Hager, P. Jensen, and R.H. Taylor. An augmentation system for fine manipulation. Springer-Verlag, 2000. Medical Image Computing and Computer Assisted Augmentation.

[9] Rajesh Kumar, et al. Experiments with a steady hand robot in constrained compliant motion path following. volume RO-MAN99, pages 92–97, Pisa, Italy, September 1999. IEEE.

[10] Russell Taylor, et al. A steady-hand robotic system for microsurgical augmentation. *International Journal of Robotics Research*, 18(12), December 1999.

[11] Shiran Pasternak. R.A.P.I.D. - robust approach to programming interaction dynamics. Master's thesis, Johns Hopkins University, 2001.

[12] C. Strachey. Fundamental concepts in programming languages. Copenhagen, August 1967. Lecture Notes for the International Summer School in Computer Programming.

[13] Emanuele Trucco and Alessandro Verri. *Introductory Techniques for 3D Computer Vision*. Prentice Hall, Inc., 1998.

[14] David Williams. *The Mozilla Project's C++ Portability Guide.* http://www.mozilla.org/hacking/portable-cpp.html, 1998.

Appendix A

Consoles and Devices

A.1 Consoles

A.1.1 Design and Features

For a vision library with video display support, since most of the processing power needs to be spent on the vision and tracking algorithms, the *Console* implementation needs to be fast, simple, and efficient at displaying video on the screen. It needs to display images from a video sequence without taking a significant amount of processing time, at a rate fast enough for standard video (30 Hz for NTSC). The current implementation of the *Console* class `XVWindowX<T>` fulfills these requirements using the X-windows system library (known as the X-server). Since X-windows runs on many different unix platforms, using the library provides portability from one unix system to the next [1]. It also takes full advantage of MIT's shared memory extension to the X-windows library.

Computer displays are available today that have millions of colors (24 bit has over 16 million). Operating systems also provide functionality for switching between different screen depths (from 8 bit to 32 bit). As a tracking library with color support (see the Images chapter for further information on color support), `Consoles` in XVision2 provide functionality for all possible screen depths. This allows XVision2 to be run on any type of color display available, set to any screen depth by the OS.

Consoles also provide functionality for drawing over images with simple shapes (such as rectangles, ovals, and lines), providing a simple mechanism for outlining tracked features and objects. Drawing, as well as selecting, are two of the important features of the *Consoles* framework. As mentioned in the Tracking chapter, one of the methods for initializing a tracker is interactively with the mouse. *Consoles* in XVision2 provide an interface for "selecting" regions on a window. For example, to select a rectangle, the `XVWindow<T>` abstract interface provides functionality for clicking the mouse at the upper left corner of a rectangular region in the window, dragging it to the lower right corner, and releasing. Many other region types (such as ellipses, lines, or just points) can be selected through the interface.

A.1.2 Implementation

The basic class of the *Consoles* framework is the `XVWindow<T>` class. This is an abstract class providing simple interfaces for creating a window on the screen, and pushing images to that window. The class is templated on the pixel type of the image to be displayed by the window (the screen depth). Classes which implement this functionality, extend the `XVWindow<T>` class. `XVWindowX<T>` is an example of extending the `XVWindow<T>` class, on the Unix/Linux environment with X-server support. Having the abstract class `XVWindow<T>` provides the user of XVision2 with the ability to switch seemlessly between one type of console to another, using the properties of polymorphism and abstraction.

Consoles are implemented with a standardized set of interfaces.

- The `XVDrawable` interface provides a set of function prototypes used for drawing shapes on the display. Each console class that supports drawing capabilities, extends `XVDrawable` and implements its function prototypes. One of the advantages of this design, allows functions to take `XVDrawable` types as parameters, without being templated on the image type of the actual console class. A perfect example of this feature in use is with the `XVFeature::show()` function, which outlines the tracked feature on the console. The `show()` function's prototype looks like this:

```
template <class S, class I>
void XVFeature<S, I>::show(XVDrawable &);
```

Because `XVDrawable` isn't templated on the image type of the console, the `show` function doesn't have to be templated on the image type. Due to the polymorphic properties of C++, a reference to any console type implementing the `XVDrawable` interface can be passed to the `show()` function.

- the `XVInteractive` interface is similar to the `XVDrawable` interface. It provides function prototypes for selecting regions (such as lines, rectangles, or ovals) on the console with the mouse. As in the case of `XVDrawable`, `XVInteractive` isn't templated on the image type, allowing it to be used as a parameter to functions, without requiring the templating of those functions on image types. To demonstrate, the function prototype for `XVFigure<S, I>::interactiveInit()` is shown here:

```
template <class S, class I>
S XVFigure<S, I>::interactiveInit(XVInteractive &, const I &);
```

These interfaces provide a simple mechanism for drawing and selecting on a console, without causing unnecessary clutter.

A.2 Devices

As video and camera hardware decreased in price, many different standards and formats for supporting video cards became available. Both the added complexity of color frame grabbers, and the new popularity of digital over analog standards (such as firewire) caused increase in the possible types of framegrabbers available. The *Devices* framework in XVision2 provides support for the most common types of frame grabbers and image sequence generators (an example would be an mpeg library), including both digital (firewire), analog, and mpeg.

A.2.1 Design

The design of the *Devices* framework is similar to the other components of XVision2 . Recognizing the similar structure of all image sequence generators, The abstract class `XVVideo<T>` provides functionality for “grabbing” a single frame (an image in the sequence) from a video device. The class is templated on the **image** type (not the pixel type), which allows support for frame grabbers that return images in a wide variety of color formats, such as YUV, HSV, or RGB. As with the *Consoles* framework, having an abstract class provides abstraction to the *Devices* framework, so that switching between two different devices is relatively simple.

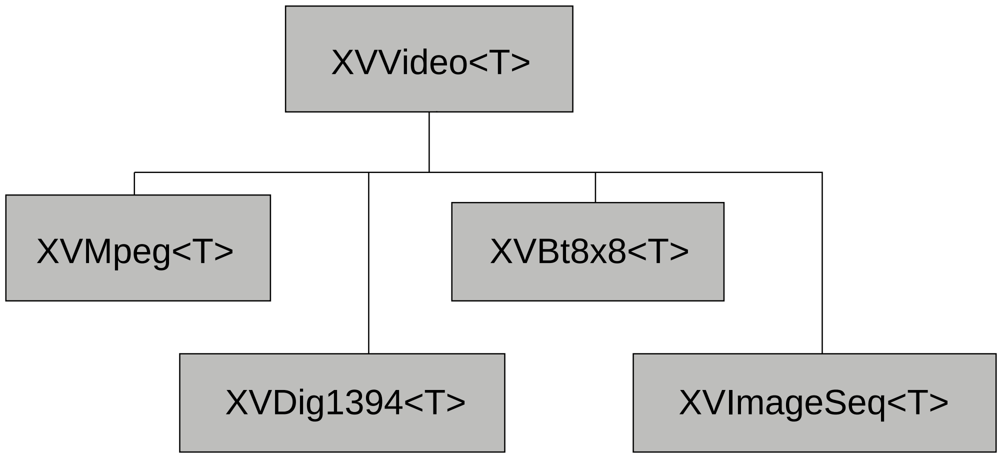

Figure A.1: The Class Diagram for Video Devices. This shows some of the different classes that extend from the XVVideo class

Vita

Samuel Lang received the Bachelor of Arts in Computer Science with a double degree in Mathematical Sciences from the Johns Hopkins University. In 1999, he began graduate studies and research in Computer Science at Johns Hopkins University under the guidance of Professor Greg Hager.