A Framework for Real-time Window -Based Tracking using Off-The-Shelf-Hardware

Greg Hager
Sidd Puri
Kentaro Toyama

Research Report YALEU/DCS/RR-988
October 1993

A Framework for Real-time Window-Based Tracking using Off-The-Shelf Hardware

Gregory D. Hager Sidd Puri Kentaro Toyama

PRE-RELEASE DRAFT VERSION – October 12, 1993

1 Rationale

Historically, real-time vision has resorted to specialized, often custom-made, hardware in order to process the huge amount of information available in video images. This has tended to limit work on real-time vision and related applications such as visual servoing to a few research labs able to afford the cost of such hardware and to support the personnel required to maintain it. However, the speed of standard workstations continues to increase, prices continue to decrease, and multi-processor architectures are becoming more widely available and accessible. These advances anticipate the day when many vision applications that do not require massive data transfers (*e.g.* full frame image processing) can be run on standard workstations outfitted with a simple framegrabber. Applications that already use this type of approach include automatic driving [1] structure from motion using image streams [3], vision-based servoing [2].

We have constructed general-purpose software for experimenting with vision-based tracking applications. The emphasis has been on flexibility and efficiency on standard scientific workstations and PC's. The system is intended to be a portable, inexpensive tool for rapid prototyping and experimentation for teaching and research. The model of computation is one of independent tracking agents that communicate information to one another in a hierarchical network. Hence, once developed, porting to a prototype to a hard real-time architecture such as Transputers[1] should be relatively straightforward.

The remainder of this document is divided into three sections. The next section describes the general software architecture. Section 3 supplies details about the construction of individual tracking modules. Section 4 describes the construction of a constrained collection of tracked features. The final sections briefly describe some experimental results with the system. The appendix lists the individual class declarations.

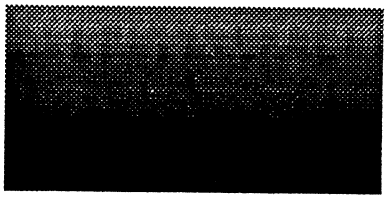
[1] TM

2 Software Design

The tracking system is based on two central ideas: the notion of window-based tracking, and the notion of state-based programming of networks of tracked objects (TO's).

Every tracked object occupies some window in a video image. A window is an image defined by its height, width, position, and orientation in the framebuffer coordinate system. All operations within the window are *relative to the local window coordinate system.* An example will help. Suppose that we have the following image with a window located as shown:

Then the image associated with that window (in window coordinates) looks like this:



Notice that regardless of orientation, the line appears roughly horizontal in window coordinates.

The beauty of this scheme is that most image processing operations such as line detection or feature matching can be implemented *independent of orientation or position.* As long as the detection or matching routines provide enough information to maintain the position and orientation of the window relative to the feature in the image, the image processing operations can presume a standard pose. This makes them simple (and fast) to implement. Acquiring windows at any position and orientation can also be implemented quickly by using ideas for fast drawing of lines and boxes borrowed from graphics.

A feature in a window can be thought of as a state-based system. Minimally, the state is the position and orientation of the feature (= that of the window) relative to the framebuffer coordinate system. The state of the system may be updated based on some combination of three sources of information: observation, prediction, and constraints from other tracked objects. For example, if one desires to track a corner consisting of two intersecting lines, it is possible to declare a "corner" object consisting of two contrast-based edge detectors. The state of an "edge object" is its location and orientation in the image. At each tracking cycle, the location of each edge is predicted, and this

estimation is refined by an edge detector running in a window at the predicted location. The state information of both edge objects is used to compute the location and orientation of the corner. The corner parameters can then be used as feedback to adjust the states of the individual edge trackers.

There are five main classes that provide the substrate for implementing these functionalities:

1. The `Video` base class is the interface to the framebuffer. It provides basic image acquisition and display.
2. The `Image` base class provides a rudimentary level of image storage and manipulation.
3. The `Pattern` base class provides a generic interface for feature detection algorithms.
4. The `Thing` abstract base class is the substrate for a tracking agent. It provides an interface for managing abstract tracked objects. `Thing` is usually combined with another class derived from `Pattern` to form a complete tracking agent.
5. The `PolyThing` abstract derived class combines several `Thing` 's together, and can enforce constraints among them. It is derived from `Thing` itself, and hence provides all of the functionality of a `Thing` in addition to several features for manipulating multiple TOs. Since it is a `Thing` , `PolyThing` can be nested in a hierarchical structure.

3 Implementing Trackable Objects

This section describes the software structures used to define base level trackable objects.

3.1 Acquiring and Storing Images

The video class permits acquisition of images and display of images and graphics. For most applications, it will only be necessary to declare one or more video structures and pass them in certain function calls. The prototype for declaring a `Video` is:

```
Video(int channel = 0, int camera = 0)
```

For example, `Video v1,v2(1,1)` declares two video devices; one which runs on the default video device on the default channel, and one which uses channel one of video device one. The class listing in Appendix A describes other low-level functionalities such as drawing.

The `Image` class provides for manipulation of images. Each pixel occupies a (signed) integer. An image is created with a width and height which is retained throughout its storage life except in certain special cases such as assignment. Images are only used internally and are never seen explicitly from a user's point of view. Images are acquired or displayed by passing a video object to one of the acquisition or display functions (see Appendix B).

3.2 Finding Patterns

`Pattern` is a virtual base class the requires three functions: xpad, ypad, and offset. The former specify how much border region is needed outside the effective detector area, and the latter find the pattern in an image. `Pattern` is used by `Image` routine `find` (see Appendix B and Appendix B).

Since it is virtual, only derivatives of the pattern class are actually used in programming. The two derivatives currently available are:

```
  Edge(int mwidth = 8, alpha = 15)
Absmedge(int mwidth = 8, alpha = 15)
```

The first pattern detects a dark-light step transition of width 16 (2 times 8) pixels, and oriented within 15 degrees of an expected orientation. The second detects an arbitrary contrast difference across a line. The major difference between the two is that the first expects a solid line; the second is more tolerant of changes in contrast. The second is also significantly slower than the first.

3.3 Thing

`Thing` manages the behavior of a TO. Essentially, it maintains internal state information for the TO and provides an interface to the following capabilities:

- Initialization: The position, orientation, and size of the feature window, as well as other application specific parameters.
- Updating: Locating the tracked pattern and updating TO parameters.
- Display: Showing the current state of the tracked object.

`Thing` is a virtual object that supports building other TO's. The user can customize this object by supply functions for the above as well as additional functionality such as prediction.

Currently, the only `Thing` that is available is `Line` . There are two instantiations:

```
Line (Video *v,int length_in = line_length, int width_in =line_width);
Line (Pattern *pat_in, Video *v,int length_in = line_length, int
                                  width_in = line_width);
```

In the first case, the line is created with a default pattern of `Edge` attached to the video source pointed to by `v`. The second permits explicit specification of the type of edge used to define the line.

`Line` has four state variables accessed by the following functions:

`float &x(), float &y(), float &angle(), float &strength().`

In addition, a special function `void move(float dist)` is defined. It moves the line along the contour as it is tracked.

A simple, complete program for tracking lines is:

```
#include "Console.h"
#include "Video.h"
#include "Pattern.h"
#include "Line.h"

main() {
  Console c;
  Video v;
  Edge e;

  Line l1 (&e,&v);

  l1.init (c);

  l1.show ();
  while (1) {
    Line prev = l1;
    l1.update ();
    prev.clear ();
    l1.show ();
  }
}
```

The variable type `console` is, in this case, an X-Windows console that supports interactive initialization of the line trackers.

4 Using a PolyThing

The `PolyThing` class adds the functionality required to manipulate ensembles of tracked objects. An instance of an `PolyThing` can be given one or more `Thing` 's. An object of type `Thing` is added to a polything by using the operator `+=`. For example, if we declare `Line l1` and `Polything p`, then `p += l1` would "add" the line to `p`. By default, a raw `PolyThing` is just a "container" that manages a number of unrelated `Thing` 's. In this case, the state of the `PolyThing` is the concatenation of the state of the `Thing` 's. Initialization and display simply execute the initialization and display routines of the children TO's.

A PolyThing can be customized by overriding any or all of these functions. For example, the display function for a PolyThing may be different than the display function of its components. Additional state information can be added when a derivative PolyThing is defined. This extra state information can be managed by a *constraint function.* By default, after all of the children are updated, the PolyThing update function calls constraint(). This function can then manipulate the state array in whatever fashion it pleases. Any changes it makes to the state of children is propagated to the children before the next tracking cycle. The summary in Appendix C describes the functions in more detail, and the examples later in this section describe currently available PolyThing 's.

By using a PolyThing , the previous program can be extended to track multiple lines:

```
#include "Console.h"
#include "Video.h"
#include "Pattern.h"
#include "Line.h"
#include "Polything.h"

main() {
  Console c;
  Video v;
  Edge e;
  Line l1 (&e,&v), l2(&e,&v);
  Polything p;

  p += l1;
  p += l2;

  p.init (c);

  p.show ();
  while (1) {
    Polything prev = p;
    p.update ();
    prev.clear ();
    p.show ();
  }
}
```

Note the use of the operator += as a means of incorporating the individual lines into the PolyThing .

4.1 Implementing Intersecting Lines Using Constraints

General line intersection, the `GILine` class, is implemented as a derivative of the `PolyThing` class. This class constrains two lines to meet at a setpoint defined relative to the center of the line. This is done by "sliding" the lines along their respective contours until they meet at the appropriate points. The constraint function for doing this is is implemented as follows: let θ_1 and (x_1, y_1) be the orientation and position of line 1, and θ_2 and (x_2, y_2) be the orientation and position of line 2. Solving for λ_1 and λ_2 in the following equations determines the intersection points of the lines:

$$x_1 + \lambda_1 \cos(\theta_1) = x_2 + \lambda_2 \cos(\theta_2)$$
$$y_1 + \lambda_1 \sin(\theta_1) = y_2 + \lambda_2 \sin(\theta_2)$$

The solution is:

$$\lambda_1 = \frac{(x_1 - x_2)\sin(\theta_2) - (y_1 - y_2)\cos(\theta_2)}{\sin(\theta_1 - \theta_2)}$$
$$\lambda_2 = \frac{(x_1 - x_2)\sin(\theta_1) - (y_1 - y_2)\cos(\theta_1)}{\sin(\theta_1 - \theta_2)}$$

Given setpoints λ_1^* and λ_2^*, the line positions are updated by:

$$x_1 := x_1 + (\lambda_1 - \lambda_1^*)\cos(\theta_1) \qquad x_2 := x_2 + (\lambda_2 - \lambda_2^*)\cos(\theta_2)$$
$$y_1 := y_1 + (\lambda_1 - \lambda_1^*)\sin(\theta_1) \qquad y_2 := y_2 + (\lambda_2 - \lambda_2^*)\sin(\theta_2)$$

`GILine` is defined as a derivative of `PolyThing` that has an additional two state variables—the position of the corner. Two access functions are defined to access these state variables:

```
float &x() {return state[0];}
float &y() {return state[1];}
```

Note that the additional state variables appear at the beginning of the state array.

Members of the queue of tracked objects can be accessed using the pre-defined function `Polything *child(int n)`. Since we know that `GILine` will have two lines as "children," we may decide to keep two explicit pointers to the children, for example:

```
Line *l1 = (Line *)child(0);
Line *l2 = (Line *)child(1);
```

This piece of code appears at the end of every constructor for the class. With these declarations, the constraint function would be:

```
void GILine::constraints()
{
  double s1,s2,c1,c2,dx,dy,dt,lam1,lam2;
  char *c;

  sincos(l1->angle(),&s1,&c1);
  sincos(l2->angle(),&s2,&c2);

  dt = sin(l1->angle()-l2->angle());
  dx = l1->x() - l2->x();
  dy = l1->y() - l2->y();

  lam1 = (dx * s2 - dy * c2)/dt - setpoint[0];
  lam2 = (dx * s1 - dy * c1)/dt - setpoint[1];

  l1->x() += lam1 * c1;
  l1->y() += lam1 * s1;

  l2->x() += lam2 * c2;
  l2->y() += lam2 * s2;

  x() = l1->x() + cos(l1->angle())*setpoint[0];
  y() = l1->y() + sin(l1->angle())*setpoint[0];

}
```

The array `setpoint` holds the specified setpoints for the two lines.

There are several derivative classes of intersection based on different setting of λ_1^* and λ_2^*. They are:

`Cross` : $\lambda_1^* = \lambda_2^* = 0$.

`Tee`: $\lambda_1^* = l/2$ and $\lambda_2^* = 0$ where l is the length of the line.

`Corner`: $\lambda_1^* = \lambda_2^* = l/2$.

There are currently three methods of creating a `GILine` :

```
GILine(Video *v,float s1, float s2, int color_in = no_color);
GILine(Edge *e, Video *v, float s1, float s2,
       int size = line_length, int color_in = no_color);
GILine(Line l1, float s1, Line l2, float s2, int color_in = no_color);
```

The first is the most generic. Given a video device it internally creates two default lines. The second internally creates two lines using the specified pattern, and the last lets the user explicitly specify the lines. All of the derivative classes mirror these calls except that explicit values for `s1` and `s2` are provided based on the type of object.

Here is an example program showing how to track two crosses on two different cameras:

```
#include "Console.h"
#include "Video.h"
#include "Pattern.h"
#include "Line.h"
#include "Polything.h"

main() {
  Console c;
  Video v1(0,0),v2(0,1);
  Edge e;
  Line c1 (&e,&v1), c2(&e,&v2);
  Polything p;

  p += c1;
  p += c2;

  p.init (c);

  p.show ();
  while (1) {
    Polything prev = p;
    p.update ();
    prev.clear ();
    p.show ();
  }
}
```

5 Experiments

5.1 Tracking a Rotating Box

The following is an edited version of report submitted by Kentaro Toyama as part of our graduate vision course. He used an earlier version of this system to track rotating boxes. This material demonstrates how the system can be used to build tracking applications and also provides some experimental data on how the system performs.

Tracking a Face Four crosses can be tracked "simultaneously" by tracking one cross after the other. Given the coordinates of the crosses in "box coordinates" and/or the assumption that the crosses lie in a plane, scale and rotation information can be directly deduced from locations of the tracked crosses.

A straightforward but expensive method is to solve a system of linear equations to find the rotation matrix and the translation vector of the box. With four points and the planar constraint, the scale factor can be computed as well. This standard algorithm, however, fares badly when the tracked points are significantly off the mark (which happens as the tracked face turns more and more to the side). Instead, a computationally simpler approximation of the algorithm is used to determine the scale factor and rotation.

The difference in y-coordinates of two vertically aligned crosses is averaged with the difference in y-coordinate of the other two crosses. The average of these numbers is the approximate height of the rectangle with corners at the four crosses, and this result is used to compute the scale of the box in relation to the known model. Using this scale factor, the algorithm then uses an average of the differences in the x-coordinates of the horizontally aligned pairs of crosses to determine the rotation: first measuring the foreshortening effects of the face, and then using trivial trigonometric identities.

Interleaved with this computation, which relies on reliable tracking of the four crosses, there are constraints imposed upon the (perceived) location of the crosses. Using the most recent information about the rotation of the box and the location of the crosses from the model, each cross is constrained to lie within a certain threshold allowance of its expected location, with respect to the other crosses. At least three of the crosses must by tracked accurately in order for this constraint to work well, but with constraint reinforcement occurring every track cycle (where each cross is tracked once for its new location), mistracking should be rare once the algorithm locks properly onto the four crosses.

Tracking Two Faces When one face is tracked successfully and the angle between the camera's z-axis and the surface normal exceeds a certain threshold (set at 45 degrees based on empirical observation), the program attempts to track a second face, tracking the four crosses that are newly visible to the camera.

Using the scale factor and rotation angle determined from tracking a single face, the approximate locations of the new crosses can be determined by manipulating the box model with a little trigonometry.

Originally, cross objects were placed at the expected positions and slightly shifted in various directions to determine the best location to begin tracking. However, this proved costly as new crosses were created for each shift, requiring more data transfers from the frame buffer. In the current implementation, two separate horizontal and vertical lines are tracked at the expected locations to determine where the crosses should be positioned. The lines that track the initial positions use windows which are wider than normal, to allow for a significant error in the original approximation. This also eliminates the necessity of explicit shifting to find the optimal location.

The speed improvement of the second method over the first was such that whereas originally, the

four crosses from the original face needed to be tracked as the search proceeded from the location of the new crosses, the new cross positions are now found quickly enough that tracking of the first set of crosses can resume smoothly even after the search.

When tracking two faces, additional information is gained to strengthen the constraints on the positions of the tracked objects. Again, the x and y-coordinates of each cross are watched carefully, with respect to all other crosses tracked. With more information, the algorithm makes more confident decisions about when a track has gone astray. Extra weight (double) is given to the crosses of the first face tracked, since it is assumed that those crosses were successfully tracked. The new crosses have a slight change of mis-initialization which is promptly corrected by the constraints.

Again, a relatively straightforward method to impose these constraints would be to use a least-squares method that tries to maximize the weighted sum of tracked positions and model-consistent positions, but this, too, was given up in favor of speed. Instead, y-coordinates of the positions are constrained by the average of four roughly collinear crosses ("roughly" because of foreshortening effects, and because the crosses do not line up from one face to another – appropriate corrections are made using the model), and x-coordinates are matched using a combination of averages and expected rotation angle.

Losing Track of a Face If, while two faces are being tracked, one of the surface normals forms an angle with the optical axis greater than a particular threshold, that face is relinquished by the algorithm. The threshold was set at about 60 degrees, since empirically it was observed that crosses become far more difficult to track at that angle.

Another method to determine when a face should cease to be tracked is to monitor how well the crosses are tracked. But no consistent measure of a good track was discovered, and because any given tracked cross might exhibit migrant behavior from time to time, this algorithm often gave up tracking a face too easily.

Experiment Equipment The camera was a Cohu 8mm black and white TV lens connected via a frame buffer to a Sun Sparcstation 2. Images were displayed on a color terminal, with tracked lines overlaid in color on the black and white camera image. All computation was done on the Sun workstation, using the tracking software developed by Greg Hager, along with support code for camera I/O. The main program, written in C++, calls routines in the tracking software for tracking edges, crosses (as intersecting lines), and MTs. The pan-tilt head (of which only the panning capabilities were used) was also controlled by the workstation, using a C++ program written by Sean Engelson. The pan-tilt program allowed fairly accurate control of both angular velocity and rotational position.

The Boxes Two different types of boxes were used for the experiment. The relevant dimensions of Box A were approximately 10cm x 7cm on two faces, and 13cm x 7cm on the complementary faces. Crosses were drawn about 2cm in from each corner of each face, with some variation. Each cross was an arrangement of four square-tile patches, with two black and two white patches each. Patches of the same color were positioned diagonally opposite each other. The crosses were oriented to line

up with the edges of the box, so that they had only horizontal and vertical edges. The model contained information about the relative distance between the crosses and between crosses and vertical edges of the face. The topological relationship of the faces was also known implicitly by the algorithm, e.g, it expects to see Face 2 always at the right of Face 3 and so forth.

Box B did not have any crosses drawn on the faces. Instead, alternating faces were solidly colored either black or white. Against a gray background, the corners were expected to stand out as cross-like figures. The dimensions of the black faces were approximately 5cm x 8cm, and the white faces were 6cm x 8cm. The model contained the same information as for Box A, but since the "crosses" and the edges coincided, the distances between crosses and vertical edges was zero throughout.

Experimental Results Both boxes were tracked at three different distances: at 100cm, 80cm, and 60cm from the camera. Panning speeds were started at 1 degree per second and were raised gradually until tracking ability deteriorated. Tracking was considered "successful" if the program was able to follow 360 degree box rotation without error for more than two-thirds of the trials.

Experiments on Box A were performed with both a slow algorithm and a fast algorithm. The slow algorithm used least-square constraints and cross-shifting to find the positions of new crosses (see Section 5.1). The fast algorithm relied on approximate methods for the constraints (see Sections 5.1 and 5.1) and found new crosses by using single track attempts with lines. Experiments on Box B were performed using only the fast algorithm.

The slow algorithm was able to track single faces fairly well, but was unable to make smooth transitions to track new faces. At best, it could track Box A at speeds of 4 or 5 degree per second, at each distance.

The fast algorithm performed excellently on Box A, comfortably tracking faces at any distance with angular velocity up to 6 degrees per second. The algorithm fared better at the far distances, tracking up to 9 degrees per second at a distance of 80cm, and even up to 12 degrees per second at 1 meter. Tracking at speeds greater than 10 degrees per second, however, was unreliable, and suspect to slight variations in the tilt of the box or the camera.

Informal translation and rotation experiments were performed with Box A using the fast algorithm, by moving the pan-tilt head by hand. While no accurate measures of translation speed were made, the algorithm was able to track slow translations (approx. 2cm/sec) at a distance of about 1 meter, provided that panning speed was not raised above 6 degrees per second. In particular, tracking ability deteriorated when the direction of translation coincided with the optical movement of the crosses.

Despite the noticeable instability of the crosses overlaid on the image, Box B was almost equally successfully tracked as Box A with the fast algorithm. At distances of 100cm and 80cm, Box B could be tracked with angular speeds of up to 9 degrees per second, and at 60cm, it was tracked up to 6 degrees per second, but it failed almost a third of the trials at the latter distance.

Interpretation of Results Most tracking failures were one of two kinds. In the first type of failure, the difference in y-coordinates of vertically aligned crosses was mis-tracked, resulting either in the mistake of believing a face to be facing the wrong direction with respect to the optical axis, or in a misjudgment of scale. The second type of failure arose when the boundaries of a cross object were tracked instead of the actual cross itself. Both resulted in a miscalculation of the rotation angle, which led to poor expected positions for any new crosses. Performance degradation, therefore, was usually very poor – experimental setups with successful tracks at 10 degrees per second sometimes failed completely at 10.5 degrees per second.

For all distances, Box A usually failed to track when the box, and consequently the image of the crosses, moved too quickly. The tracker was unable to find the position of the new cross at each moment, and often locked onto an object boundary, instead of at the center. At near distances, higher rotational velocity corresponds to greater absolute velocity of the objects in the image, so tracking was less successful, despite the fact that the tracked objects appeared larger. Greater foreshortening effects also contributed to the poorer performance of the tracker at close range. Translational movement in one direction (if it coincided with the apparent motion of the crosses) exacerbated this problem but eased it in the opposite direction.

The results for Box B reflect the same problems experienced by tracking Box A, but slightly more emphasized. Since the corners tracked were not actual crosses and because they occurred at the boundary of the box, the program tended to lock on to background edges and lines. Nevertheless, tracking remained surprisingly successful, due to the flexible implicit definition of a cross as consisting of two approximately perpendicular lines.

Possible Improvements By observing the overlaid crosses on the monitor, it was clear that at higher rotational speeds, the track positions of the crosses always lagged behind the actual position of the crosses. This might be remedied by including some simple form of prediction in the tracking of a single cross. Just by maintaining the positions of the two most recent track positions, a cross could linearly extrapolate the new position of a cross. Similarly, the expected positions of new crosses could be extrapolated by maintaining a finite record of past rotational angles. Since movement tends to be smooth and fairly straight for the experiments performed, this sort of simple prediction may greatly enhance trackability.

For an object like Box B, tracking crosses as such is wasteful and ineffective. An oriented corner may be better suited for the job. The routine for "intersecting lines" in the most recent version of the tracking software is likely to remedy this problem. In addition, because the tracked objects occur at the boundaries of the faces, only six objects need to be tracked when tracking two faces, instead of eight.

5.2 Experiment 2: Visual Tracking for Robot Control

In this experiment, a robot is aligning a floppy disks that it is holding with a target disk based on visual feedback. In order to do this, it tracks two corners of each disk in two cameras. The problem for doing this is essentially an extension of the one defined above for tracking crosses in two cameras. In end-effect, this calls for tracking 16 line segments and performing some additional

constraint computations. For Line segments of length 16 pixels and a capture width of ± 10 pixels, we were able to track consistently at a rate of 25 Hz.

6 Continuing Development

This software is still changing rapidly. Any applications based on it should expect things to change under their feet. We are also continuing to add functionality and perform other experiments. A wide-open issue is how to automatically initialize trackers.

References

[1] E. D. Dickmanns and V. Graefe. Dynamic monocular machine vision. *Machine Vision and Applications*, 1:223–240, 1988.

[2] G. D. Hager, W.-C. Chang, and S. Morse. Robot feedback control based on stereo vision: Towards calibration-free hand-eye coordination. Submitted to the 1994 International Conference on Robotics and Automation., 1993.

[3] C. Tomasi and T. Kanade. Shape and motion from image streams: a factorization method, full report on the orthographic case. CMU-CS 92-104, CMU, 1992.

A Console and Video Classes

```
class Console {

public:
  Console();
  ~Console() { cleanup(); }
  void cleanup();

  cwindow open (int width, int height, char *name = "Tracker");
  void close (cwindow window);
  void show (cwindow window, int *image);
  void line (cwindow window, int x, int y, int length, float angle,
             int pixel = -1);
  void clearline (cwindow window, int x, int y, int length, float angle)
    { line (window, x, y, length, angle, pixels[128]); }

  position getpos (Video &v, char *name = "Get position",
                   int length = default_length);
};



class Video {
  cwindow window;

public:
  Video (int channel = 0);                    // initialize hardware
  ~Video() { cleanup(); }
  void cleanup();

  cwindow getwindow() { return window; }

  void grab (float x, float y,                // cut out rectangle
             int width, int height,
             int *image);
  void grab (float x, float y,                // cut out tilted rectangle
             int width, int height,
             int *image, float angle);
  void show (int width, int height,           // show image in top-left corner
             int *image);
  void open (Console &c, char *name = "Video")
  void close (Console &c)
  void line (float x, float y, int length,    // draw an overlayed line
             float angle,
             int color = default_color);
  void clearline (float x, float y,           // clear overlayed line
                  int length, float angle)
};
```

B Image and Pattern Class

```
class Image {
public:
  Image (int width_in, int height_in);
  Image (Image &im);
  ~Image();

  Image &operator= (Image &im);

  int *data();

  void grab (Video &v, float x, float y);      // refresh image
  void grab (Video &v, float x, float y, float angle); // refresh tilted image
  void show (Video &v);                        // display image contents

  void open (Console &c, char *name = "Image");
  void show (Console &c);
  void close (Console &c);
  offset find (Pattern *p);                    // locate a pattern

};

class Pattern {
public:
  virtual ~Pattern() {};
                                               // amount of padding needed to
                                               // detect the pattern on the edges
                                               // of an image
  virtual int xpad (int width, int height) = 0;
  virtual int ypad (int width, int height) = 0;

  virtual offset find (int width, int height, int *image) = 0;
};
```

C Thing and Polything Classes

```
class Thing {

public:

  virtual void init() = 0;
  virtual void init (Console &c, Video &v) { init(); }
  virtual Thing *dup() = 0;
  virtual void show (Video &v, int color_in = no_color) = 0;
```

```
  virtual void clear (Video &v) = 0;
  virtual void update (Video &v) = 0;


  Thing (int ncoords_in, int color_in = default_color);

  Thing (Thing &t);
  virtual ~Thing();

  virtual float operator[] (int n);          // for debugging only

  Thing &operator= (Thing &t);
};


class Polything: public Thing {

public:
  Polything (int pncoords_in = 0, int color_in = no_color);
  Polything (Polything &p);
  virtual ~Polything();
  Polything &operator= (Polything &p);

  virtual void init();
  virtual void init (Console &c, Video &v);
  virtual Thing *dup() {return new Polything (*this); };
  virtual void show (Video &v, int color_in = no_color);
  virtual void clear (Video &v);
  virtual void update (Video &v);
  virtual void constraints();

  Polything &operator+= (Thing &newthing);
};
```

D Edge and Line

```
class Edge: public Pattern {

public:
  Edge (int mwidth_in = default_mwidth, float alpha_in = default_alpha);

  int xpad (int width, int height);
  int ypad (int width, int height);
};

class Absmedge: public Pattern {
```

```
public:
  Absmedge (int mwidth_in = default_mwidth, float alpha_in = default_alpha);
  ~Absmedge() { delete mask; }

  int xpad (int width, int height);
  int ypad (int width, int height);
  offset find (int width, int height, int *image);
};



class Line: public Thing {

public:
  Line (int length_in = line_length, int width_in = line_width): Thing (3);

  Line (Pattern *pat_in,int length_in = line_length, int width_in = line_width): Thing (3);
  }
  Line (Line &l): Thing (3);

  void init() { init (half (x_max), half (y_max)); }
  void init (float x_in, float y_in, float angle_in = 0);
  void init (Console &c, Video &v);
  Thing *dup() { return new Line (*this); }
  void show (Video &v, int color_in);
  void clear (Video &v);
  void update (Video &v);

  float &x()     { return state[0]; }
  float &y()     { return state[1]; }
  float angle() { return internal_angle()+M_PI_2;};

  void open (Console &c, char *name = "Line")
  void close (Console &c)
  void show (Console &c)

};
```

E GILine and Related Classes

```
class GILine : public Polything
{

protected:

  Line *child(int n) {return (Line *)Polything::child(n);};
```

```
public:

  GILine(Line l1, float s1, Line l2, float s2, int color_in = no_color);
  GILine(Video *v,float s1, float s2, int color_in = no_color);
  GILine(Edge *e, Video *v, float s1, float s2,
         int size = line_length, int color_in = no_color);

  GILine(GILine &g) : Polything(g) {

  float &x();
  float &y();

  void constraints();

  void init (Console &c, Video &v);

};


class Cross : public GILine
{

public:

  Cross(Line l1, Line l2, int color_in = no_color) :
    GILine(l1,0,l2,0,color_in) {};

  Cross(Video *v, int color_in = no_color) :
    GILine(v,0,0,color_in) {};

  Cross(Edge *e, Video *v, int size = line_length, int color_in = no_color) :
            GILine(e, v, 0, 0, size, color_in) {};


};


class Corner : public GILine
{

public:

   Corner(Line l1, Line l2,  int color_in = no_color) :
            GILine(l1, line_length/2, l2, line_length/2,color_in) {};

   Corner(Video *v, int color_in = no_color) :
            GILine(v, line_length/2, line_length/2,color_in) {};

   Corner(Edge *e, Video *v, int size = line_length, int color_in = no_color) :
            GILine(e, v, (float)size/2.0, (float)size/2.0,
```

```
                    size, color_in) {};

};
```