1. Introduction

1.1 Introduction to AGV’s

AGV stands for **Autonomously Guided Vehicles**. They represent a class of robots which, by and large, are capable of making decisions on their own with a reduced degree of human intervention. They differ from teleoperated robots, which are robots which are simply controlled remotely or onsite by a human operator. Autonomously Guided Vehicles are proving their utility in various areas which involve operations too dangerous for human beings to perform or too far off for even remote control signals to reach the robot.

AGV’s normally come equipped with an array of sensors. These sensors act as their eyes and ears and enable it to make intelligent decisions about its environment. There are several modes of sensing, some of which include sonar, infrared (IR), tactile sensors and vision.

1.2 IRIS: a machine vision system for AGV’s

The main challenge for AGV’s is to reach a degree of autonomy which makes it reliable enough to perform certain tasks without human intervention at all. However, this kind of operation requires sensory data far richer than either IR or sonar or even tactile sensors. The richest source of information available to us is vision and it is natural that we try to incorporate such a system in AGV’s too. Machine vision systems thus allow an AGV to ‘see’, interpret its surroundings and act in a manner appropriate for the completion of its assigned task. IRIS is one such vision system currently under active development.

IRIS presents the programmer/user with a full complement of 2D image processing algorithms and easy-to-use data structures which may be extended, modified and reused as seen fit. In addition, it has an underlying (basic) 3D graphics engine which has used to implement useful algorithms which work in three dimensions, like stereovsion and 3D space carving.

IRIS is designed to be used as a standalone application in its own right as well as a library where it is plugged in and used as a module. Being extremely generic, the IRIS core works with little or no modifications on Windows and Linux platforms and can be used for any project involving machine vision or image processing.

1.3 Introduction to the test platform: COMRADE

The test platform on which IRIS runs is called COMRADE. COMRADE stands for Cooperative Mobile Robots for Autonomous Decisive Exploration. It is a research project funded by Yahoo! Inc. to investigate the viability of constructing cost-effective yet reasonably complex robots with powerful inductive machine vision intelligence and cooperative capabilities to accomplish tasks collectively.

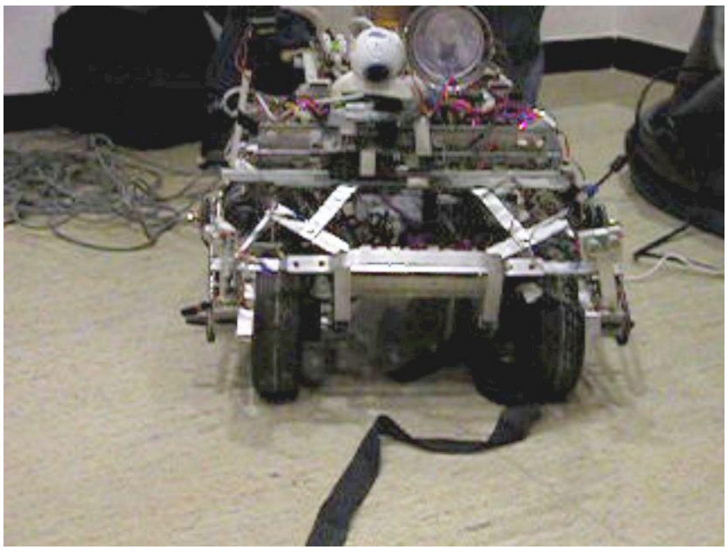


Fig. 1.1: ADAM, the first robot of the COMRADE project

2. Theory of a machine vision system

2.1. The vision data stream

The data stream in most cases consists of frames captured continuously by a sensor of some sort like web cams or digital cameras. The quality and the rate of capture depend largely upon the sensor, the speed of the ADC's and the data link. Since technology has become smaller and faster, very good frame rates is now the norm (~30 fps). Image quality has also improved substantially, so that most of the preprocessing can be performed within the sensor itself. In such cases, a substantial part of the work of the vision system is solved immediately. However for less powerful sensors, preprocessing must be performed using software.

The images received are usually interpreted as RGB streams, i.e., the color each pixel of the image is described by the relative proportions of the three primary colors (red, green, blue) used to create that color. In the case of bitmaps (.bmp files), each of the three channels can assume a value from 0 to 255. Thus 255-255-255 represents white, while 0-0-0 represents black.

However, the RGB color model is not amenable to most preprocessing operations. The HSV/HLS (Hue-Saturation-Value and Hue-Luminance-Saturation) color models are easier and more intuitive to work with. The diagram below shows the geometric relationships between hue, color and luminance:

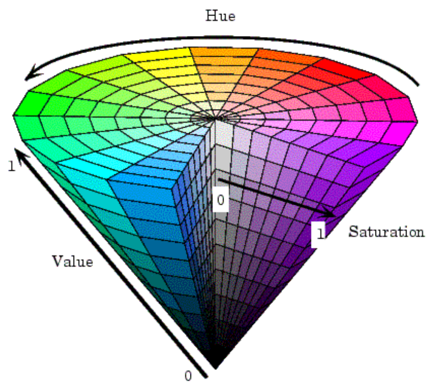


Fig. 2.1: HSV (Hue-Saturation-Value) color space

The sensor software writes each captured frame directly into primary memory. This is a very good method since file I/O is eliminated. However, this comes at the expense of requiring large amounts of runtime primary memory in case the preprocessor must store multiple frames. It may be possible to cache some important frames as files, but this should be done sparingly.

IRIS implementation: IRIS-XT currently receives its vision data from a digital camera. It is capable of accepting images as bitmap (.bmp) files as well as direct writing to specified RGB buffers. IRIS uses a templatised *Buffer* class such that we do not have to design a separate buffer class for the RGB format, one for the HSL format, and so on. This class allows basic copying to and from other buffers of the same type. IRIS currently does not support caching frames as files. Buffers may be resized.
Besides that, a *BufferManager* class is responsible for copying entire buffers or subsets of these buffers to and from bitmap files. Using separate classes decouples much of the algorithmic responsibility of the *Buffer* class, since it is used and referred to heavily by most IRIS functions.

IRIS has two fundamental data types, representing the RGB and the HSL color spaces. They are *RGBstruct* and *HSLstruct*. A *RGB* structure is used to represent non-normal RGB values (0-255). Buffers are described mostly using these two data types.

As noted previously, the vision data stream may have imbedded noise as well as artifacts. The preprocessing stage is responsible for preparing the image for further analysis. The next section discusses that topic.

2.2. Image preprocessing

If the sensor does not possess any image processing capabilities, or if the environment is noisy to the extent that the sensor's operations are not sufficient to attain the desired level of clarity, software processing of the image data must be performed. There are several algorithms that are available to preprocess the incoming data prior to analysis. Some of the techniques and their pertinent areas of application are described briefly below:

- Independent channel adjustment: RGB has three channels: red, green and blue. Likewise, HSL has three channels: hue, saturation and luminance. During occasions it is necessary to independently adjust the value of a particular channel. For example, if the color content of the image is not distinct enough, the hue may be increased to the desired level.
- Grayscale conversion: In some analyses, color content is not important. A grayscale intensity image is ideal for such cases. Grayscale conversion may be performed using more than one method, as outlined below:
    - Equate every pixel's red and green component to its green channel component. That is:

$$\boxed{\begin{array}{l} R(x, y)=B(x, y)=G(x, y) \\ \text{for all } (x, y)(limitx, limity) \end{array}}$$

    - For every pixel, equate its three channels to the average of their original intensity values. That is:

$$\boxed{\begin{array}{l} R(x, y)=B(x, y)=G(x, y) \\ = \dfrac{R(x, y)+G(x, y)+B(x, y)}{3} \\ \text{for all } (x, y)(limitx, limity) \end{array}}$$

- <u>Contrast stretching</u>: The human eye is most sensitive to the median range of electromagnetic frequencies. If the relative strength of this band of frequencies in the image is weakened, the image appears 'washed-out' due to low contrast. Contrast stretching corrects this problem by using a transfer function with slopes different from unity for various intensity levels to increase contrast. The transfer function might be as below:

I' (x, y)=. I (x, y)   if 0<I (x, y)<a

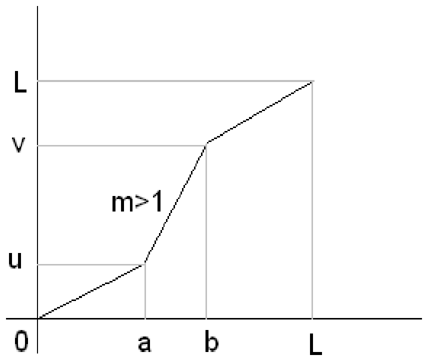
=. (I (x, y) – a) if a<I (x, y)<b

=. (I (x, y) – b) if b<I (x, y)<L

where a=L/3, b=2L/3

0<<1, >1, 0<<1

L is the maximum possible channel value



<u>Transfer function for contrast stretching</u>

Fig. 2.2

- <u>Histogram equalisation</u>: In many cases, due to shortcomings of the sensor or inadequate illumination, the nonzero values occupy a narrow band in the intensity spectrum. Analysing the histogram of a given image and spreading it out such that it covers the entire image spectrum, while maintaining its original shape, may correct this. This results in a more uniform distribution of intensity values. For low-quality sensors or those with a small dynamic response, histogram equalisation is a necessity.

- Two-dimensional convolution: Convolution is easily one of the most powerful techniques for manipulating a desired image. Typically, convolution involves multiplication of the image with a 2D mask that is slid across the image to calculate the convolved value for each pixel. The mask's dimensions are generally smaller than those of the image. The convolved intensity/ channel value of pixel depends upon its own as well as the surrounding pixel values. The operation is given as:

$I(x0, y0) = I(x, y) * M(r - x, c - y)$

for all $(x0, y0)(limitx, limity)$

x varies from $x0-c/2$ to $x0+c/2$

y varies from $y0-r/2$ to $y0+r/2$

r, c are the dimensions of the mask M

Most image processing techniques rely on convolution at some stage. For example, the Sobel, Prewitt and Canny edge detection techniques use convolution masks. Operations like blurring, smoothing and a number of generic channel transfer functions may be implemented quite easily using two-dimensional convolution.

It may be noted that convolution with a mask is mathematically equivalent to correlation with the same mask reflected along its antidiagonal.

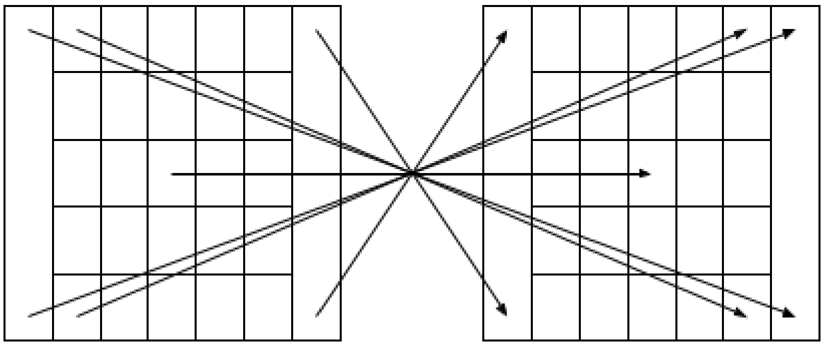


Fig. 2.3: Demonstrating 2D convolution

- Unsharp masking: If the image received is out of focus, the unsharp mask operation may be used to increase its sharpness. Sharpening of images is a necessary step prior to edge detection. However, a smoothing function must also be applied to the image to reduce the effects of accentuated noise due to the unsharp mask operation.

- Dilation and erosion: These two operations belong to the class of morphological operations. Fundamentally, neither these operations are asymmetric and are not inverses of each other, nor commutative, though the effects are somewhat the reverse of each other. Mathematically both are quite simple. Dilation involves assigning to a pixel the maximum value that is present in any of its eight immediate neighbors. Erosion involves assigning to a pixel the minimum value that is present in any of its eight immediate neighbors. Formally, we may write them as:

Dilation: I (x0, y0)=max (I (x, y))
for (x, y) varying from (x0-1, y0-1) to (x0+1,y0+1)
with (x, y)(x0, y0)

Erosion: : I (x0, y0)=min (I (x, y))
for (x, y) varying from (x0-1, y0-1) to (x0+1,y0+1)
with (x, y)(x0, y0)

- Edge detection: There are several methods for performing edge detection. Some of them are Robert's Cross Operator, Sobel, Prewitt, Laplacian and Canny. All of these edge detection schemes use convolution as one of the operations. Information on these schemes is available in literature.

It must be noted that a preprocessor like this can be extended to include any number of desired functions. It is also possible to use this system as a separate image processing API for noncritical applications. Parts of the system may be reused for other applications. In some cases, the preprocessor can also perform some rudimentary tasks of analysis, thus freeing up the following stages from some responsibilities.

IRIS implementation: The IRIS-XT image preprocessor has functions for all the techniques described above. Functions that do not require the use of convolution are available under an AlgorithmManager class. Convolution itself has been written to maximise performance while maintaining portability. The programmer may specify the mask size as well its elements. Since the mask coefficients cannot be multiplied with the invalid values encountered when boundary pixels are evaluated, these pixels are not evaluated at all, which is not a very serious shortcoming. The preprocessor also contains a preliminary multipass skeletonizer which can come in useful when analysing the skeletal organisation of a structure or object.

All of these operations are implemented independently such that the programmer using the IRIS API can configure the execution sequence of some (or all) of these functions in a way most suited to the application. Moreover, since the base class hierarchy exists within a separate namespace, the IRIS Foundation. It is optimally flat, so changes and additions can be easily incorporated.

The majority of composite data types defined by IRIS Foundation serve as the base data types for both IRIS-XT and IRIS-3D. Thus, complete decoupling of features between the two segments is achieved.

In addition to the above, IRIS-XT has a host of other support functions like histogram calculation functions, the generalised Hough Transform engine, fast shape recognition routines and statistical regression.

2.3. Typical objectives of IRIS

This is a possible list of the type of functions a system might be expected to perform. However, this is biased heavily towards our current work on IRIS. Other, equally valid, objectives exist and are attainable by extending the current architecture.

- Given a line painted on the floor in a particular color, the vision system should be able to continuously track it. Specifically, it should be able to return the angle the line makes with some constant reference.
- Given an image, it should be able to decompose it into blocks of homogeneous color/intensity for a color distribution analysis.
- Given a shape, it should be able to recognise other instances of it even when they are rotated, scaled or translated versions of the original one.

- Given a colored scene, it should be able to perform fast color filtering, recognition and be able to lock onto an object.

Most of these objectives would be unattainable in a rigid, crisp-logic frame of computation. Thus, imprecision must be built into the recognition engine. There are several candidates that qualify for such a development. Below we describe some of the different approaches and techniques that may be realistically incorporated in a vision system.

2.4. Line following and the Hough Transform

Most industrial robots implement some form line following. The sensors involved may be magnetic or visual in nature. However, the forms of processing and triangulation involved are primitive in that they are not at all suited for generic applications that require some autonomy or in noisy environments, like similarly painted lines or stray magnetic fields from random sources.

Thus, line following as a feature is best incorporated into the same environment where imprecision will be built in.

The most direct method of determining information about the orientation of a set of given points is statistical regression. However, there are some problems. They are discussed below:

- The presence of noise is an important factor. If the line is thick and well painted, then the regression routine will be insensitive to a significant amount of noise. However, in the presence of similarly colored objects or lines, the regression routine will give plainly wrong results.
- Too many variations in the luminance of the line (due to point illumination sources) may cause errors in the regression routine. Increasing some tolerance limits too high may also cause

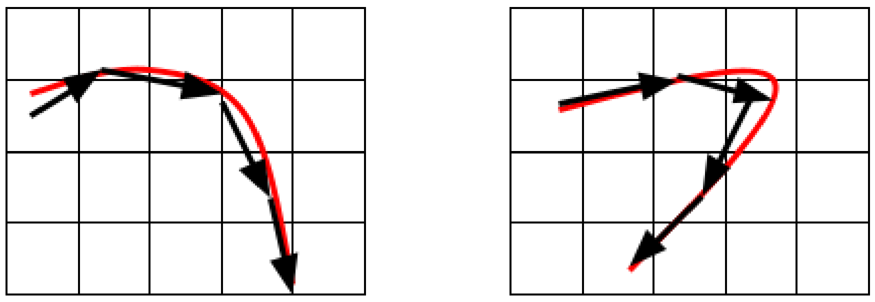
inclusion of noise points, which is again detrimental to the regression routine's performance.

The first problem cannot be overcome by any easy means. Skeletonisation, however, offers a partial solution to the problem. Erosion is also a viable, though more expensive, alternative; though determining if there are multiple lines/entities requires use of more complicated algorithms, like a variation of quadtree decomposition.

The second problem may be reduced externally by providing uniform illumination or through software compensation like histogram modification or luminance equalisation.

Another issue is the number of points to consider for regression analysis. Logic dictates that regression be carried out on a scaled down version of the line, since the line will have thousands of points to analyse. However there is a hidden performance issue. Resizing buffers takes up time. This has been observed during testing of pre-alpha versions of IRIS-XT. Using all the points ironically takes less time.

There is a further issue of curved lines. Currently, linear regression in the first power of x performs satisfactorily. If curve fitting to a good approximation is required, a higher order regression routine may be written. However, in most cases this is not necessary. This is possible if only a small segment of the line is seen at a time. This is equivalent to observing small straight segments approximating a curve. This approach works well for gentle curves. For radical turns, the result may be an angle that leads off the line, avoiding the sharp angles and leading to the next, straighter, continuing segment.



Regression performance for gradual curve

Regression performance for sharp curve

Fig. 2.4 : Regression performances

2.4.1. The basic Hough Transform and its generalisation

The basic Hough Transform is a very robust method of detecting 2D shapes which can be represented using algebraic equations. The most basic form of this transform is also one of the most useful: specifically in the area of line following.

Consider a single isolated edge point (*x, y*)—there could be an infinite number of lines that could pass through this point. Each of these lines can be characterized as the solution to some particular equation. The simplest form in which to express a line is the *slope-intercept* form:

$y = mx + b$

where m is the slope of the line and b is the y-intercept (the y value of the line when it crosses the y axis). Any line can be characterized by these two parameters m and b.

We can characterize each of the possible lines that pass through point (x, y) as having coordinates (m, b) in some slope-intercept space. In fact, for all the lines that pass through a given point, there is a unique value of b for m:

$b = y - mx$

The set of (*m, b*) values corresponding to the lines passing through point (*x, y*) form a line in (*m, b*) space. Every point in image space (*x, y*) corresponds to a line in parameter space (*m, b*) and each point in (*m, b*) space corresponds to a line in image space (*x, y*).

2.4.2 Accumulators

The Hough transform works by letting each feature point (*x, y*) vote in (*m, b*) space for each possible line passing through it. These votes are totaled in an *accumulator*.

Suppose that a particular (*m, b*) has one vote—this means that there is a feature point through which this line passes. What if it has two votes? That would mean that two feature points lie on that line. If a position (*m, b*) in the accumulator has *n* votes, this means that *n* feature points lie on that line.

2.4.3 The Hough Transform Algorithm

The algorithm for the Hough transform can be expressed as follows:

1. Find all of the desired feature points in the image.

2. For each feature point

3. For each possibility *i* in the accumulator that passes

through the feature point

4. Increment that position in the accumulator

5. Find local maxima in the accumulator.

6. If desired, map each maximum in the accumulator

back to image space. For finding lines, each feature point casts a line of votes in the accumulator.

2.4.4. A Better Way of Expressing Lines

The slope-intercept form of a line has a problem with vertical lines: both *m* and *b* are infinite.

Another way of expressing a line is in (ρ, θ) form:

$x \cos \theta + y \sin \theta = \rho$

One way of interpreting this is to drop a perpendicular from the origin to the line. θ is the angle that the perpendicular makes with the x-axis and ρ is the length of the perpendicular. θ is bounded by $[0, 2\pi]$ and ρ is bounded by the diagonal of the image.

Instead of making lines in the accumulator, each feature point votes for a sinusoid of points in the accumulator.

Where these sinusoids cross, there are higher accumulator values. Finding maxima in the accumulator still equates to finding the lines.

2.4.5 Circles

We can extend the Hough transform to other shapes that can be expressed parametrically. For example, a circle of fixed radius can be described fully by the location of its center (x, y).

Think of each feature (edge) point on the circle as saying, ”if I'm on the circle, the center must be in one of these places”. It turns out that the locus of these votes is itself a circle.

But what about circles of unknown size? In this case, we need a third parameter: the radius of the circle. So, we can parameterize circles of arbitrary size by (x, y, r). Instead of casting votes in a circular pattern into a two-dimensional accumulator, we cast votes in circles of successively larger size in a three-dimensional accumulator.

2.5. Complex shape detection using the GHT

Shape recognition may be performed after edge detection and noise removal in an image. The fundamental requirements for effective performance in this area (assuming no noise) are as follows:

- Rotational invariance

- Translational invariance
- Scaling invariance
- Distortion invariance (optional)

These features may be incorporated into a vision system by a technique we call perimeter sampling.

2.5.1 Perimeter sampling

Consider an arbitrary shape as below.

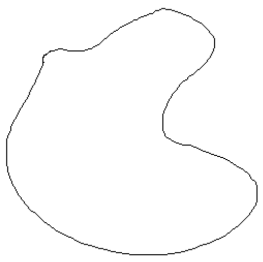


Fig. 2.5: Initial shape

Let us choose an arbitrary point within the shape somewhere near the center. The selection of the point is not important except under extreme circumstances. Now, let us calculate the distances from this point to the perimeter at increments of angle x, like so:

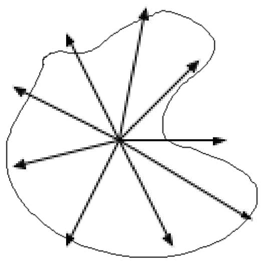


Fig. 2.6: Shape sampling

Let these distances be stored as x1, x2, x3, x4, etc. Now, assume that these values are displayed on a graph with angle as the x-axis.

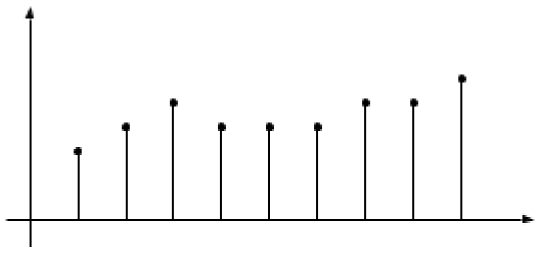


Fig. 2.7: Sampled signal

Note that the last first value repeats immediately after the last value, since we now have an angle equal to nx+360 degrees (n=sample number, x=angle increment). Thus, in the theta-domain (angle is the x-axis), we have a discrete periodic signal. This is the R-table, and we'll discuss its importance in the next section.

Now, we may normalise this set of values so that the maximum value is 1. The set of values so obtained may be termed as the shape signature.

2.5.2 More Complicated Shapes: The Generalized Hough Transform

Some shapes may not be easily expressed using a small set of parameters. In this case, we must explicitly list the points on the shape. Suppose that we make a table that contains all of the edge pixels for our target shape. We can store for each of the pixels its position relative to some reference point for the shape. We can then feature point "think" as follows: "if I'm pixel *i* on the boundary, the reference point must be at ref[*i*]."

This is called the *Generalized Hough Transform* and can be expressed as follows:

1. Find all of the desired feature points in the image.

2. For each feature point

3. For each pixel *i* on the target's boundary

4. Get the relative position of the reference point from *i*

5. Add this offset to the position of *i*

6. Increment that position in the accumulator

7. Find local maxima in the accumulator.

8. If desired, map each maxima in the accumulator back to image space using the target boundary table

We can build a table that records for each point with edge tangent orientation θ the direction a and distance r to the reference point. Thus, when we find a point with edge tangent orientation θ, we have to vote only in the direction of r, a. Of course, depending on the complexity of the shape, there may be multiple such points with tangent orientation θ. We thus build our table to store (and vote for!) all such points. This is called an *R-table*.

2.6. Color segmentation

Color segmentation is an important component in a vision system. Color differentiates; color distinguishes. Many target detection tasks can be solved very quickly simply by using color filtering and some careful error checking.

However, at times, it is necessary to analyse the color distribution of an image more carefully. In such a case, color segmentation is invaluable. This segmentation may be performed at various levels, depending upon the detail required. Here we describe a possible use of the quadtree decomposition for color segmentation.

2.6.1. Quadtree decomposition

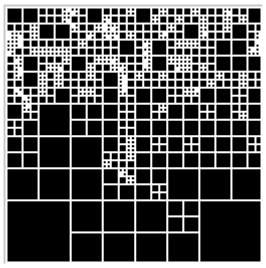
Quadtree decomposition is an analysis technique that involves subdividing an image into blocks that are more homogeneous than the image itself. This technique reveals information about the structure of the image. It is also useful as the first step in adaptive compression algorithms.

A typical quadtree decomposition function works by dividing a square image into four equal-sized square blocks, and then testing each block to see if it meets some criterion of homogeneity (e.g., if all of the pixels in the block are within a specific dynamic range). If a block meets the criterion, it is not divided any further. If it does not meet the criterion, it is subdivided again into four blocks, and the test criterion is applied to those blocks. This process is repeated iteratively until each block meets the criterion. The result may have blocks of several different sizes. For example, suppose we want to perform quadtree decomposition on a 128-by-128 intensity image. The first step is to divide the image into four 64-by-64 blocks. We then apply the test criterion to each block; for example, the criterion might be

Max (block_intensity) – min (block_intensity) <= 0.2

If one of the blocks meets this criterion, it is not divided any further; it is 64-by-64 in the final decomposition. If a block does not meet the criterion, it is then divided into four 32-by-32 blocks, and the test is then applied to each of these blocks. The blocks that fail to meet the criterion are then divided into four 16-by-16 blocks, and so on, until all blocks 'pass.' Some of the blocks may be as small as 1-by-1, unless we specify otherwise.

Original image


Quadtree organisation

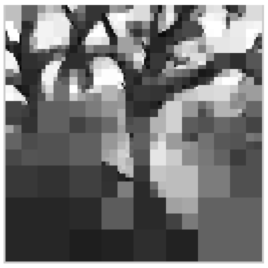

Block mean image

Fig. 2.8: Quadtree segmentation

The advantage of this technique is that it allows partition of the image at various resolutions. Increasing tolerance limits results in a blocky representation, which may be the maximum detail we may need most of the time. Only occasionally will we need a higher resolution analysis (but not too high, since that results in the original image itself). Then it is a simple task of tightening the appropriate tolerance factors.

2.7 Camera models and projective geometry

Perspective projection (central projection) describes image formation by a pinhole camera or a thin lens.

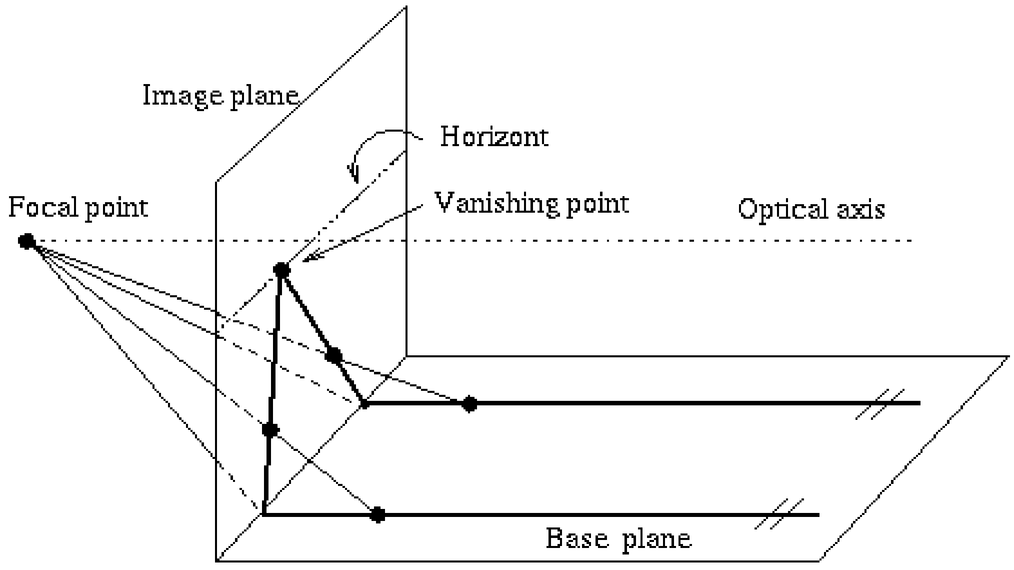


Figure 9.2 *Perspective projection of parallel lines.*

Fig. 2.9: Pinhole camera

The single perspective camera

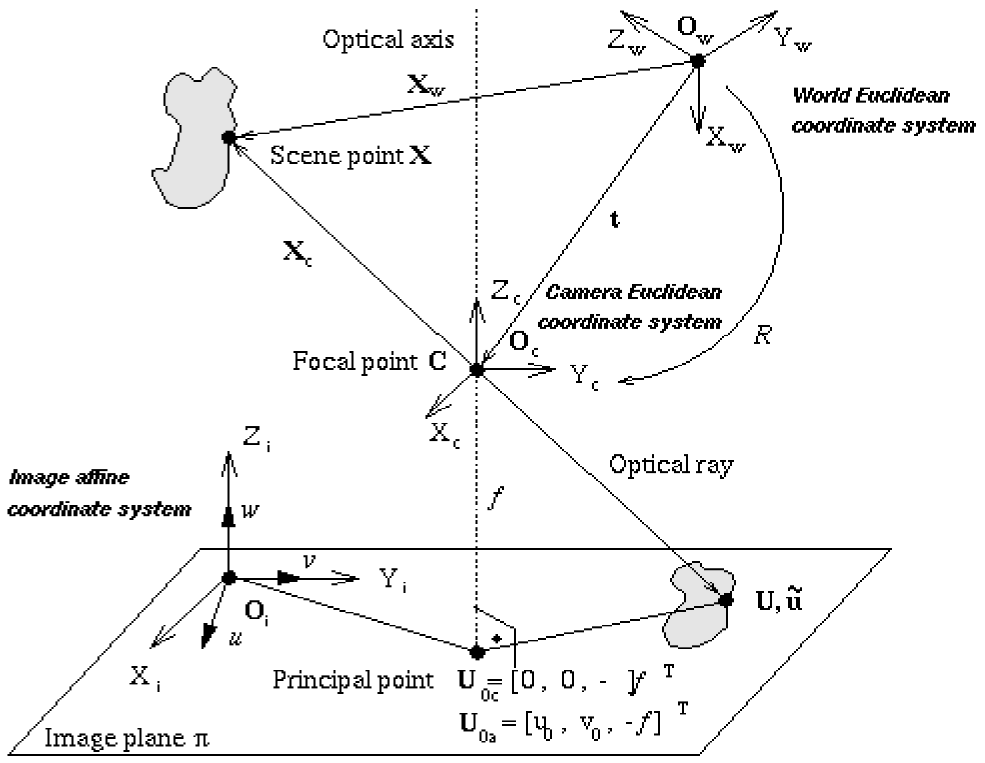

Optical axis
Z_w
O_w
Y_w
X_w
World Euclidean coordinate system
Scene point X
X_w
t
X_c
Z_c
Camera Euclidean coordinate system
O_c
R
Focal point C
Y_c
X_c
Z_i
Image affine coordinate system
w
f
Optical ray
v
Y_i
O_i
U, ũ
u
X_i
Principal point $U_{0c} = [0, 0, -f]^T$
$U_{0a} = [u_0, v_0, -f]^T$
Image plane π

Figure 9.3 *The geometry of a linear perspective camera.*

Fig. 2.10: Different coordinate systems in projective geometry

- Consider the case of one camera with a thin lens (simplest approximation). The pinhole camera performs perspective projection.
- The geometry of the device is depicted in Figure above; the plane on the bottom is an **image plane pi** to which the real world projects, and the vertical dotted line is the **optical axis**.
- The lens is positioned perpendicularly to the optical axis at the **focal point C** (also called the **optical center**). The **focal length f** (sometimes called the principal axis distance) is a parameter of the lens.

- The projection is performed by an optical ray (also a light beam) reflected from a scene point X. The optical ray passes through the optical center C and hits the image plane at the point U.
- Let's define four co-ordinate systems:
    - **The world Euclidean co-ordinate system** (subscript _w) has origin at the point O_w.
    - Points X, U are expressed in the world co-ordinate system.
- **The camera Euclidean co-ordinate system** (subscript _c) has the focal point C = O_c as its origin.
- The co-ordinate axis Z_c is aligned with the optical axis and points away from the image plane.
- We can align the world to camera co-ordinates by performing an Euclidean transformation consisting of a translation t and a rotation R.
- **The image Euclidean co-ordinate system** (subscript _i) has axes aligned with the camera co-ordinate system, with X_i, Y_i lying in the image plane.
- **The image affine co-ordinate system** (subscript _a) has co-ordinate axes u, v, w, and origin O_i coincident with the origin of the image Euclidean co-ordinate system.
- The axes w, v are aligned with the axes Z_i, X_i, but the axis u may have a different orientation to the axis Y_i.
- The reason for introducing the camera affine co-ordinates is the fact that in general, pixels need not be perpendicular and axes can be scaled differently. A camera performs a linear transformation from the 3D projective space P^3 to the 2D projective space P^2.
- A scene point X is expressed in the world Euclidean co-ordinate system as a 3x1 vector. To express the same point in the camera

Euclidean co-ordinate system, i.e. X_c, we have to rotate it as specified by the matrix R and translate it by subtracting vector t.

$$\mathbf{X}_c = \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = R\,(\mathbf{X}_w - \mathbf{t}) \qquad (9.3)$$

(Eqn 2.1)

- The point X_c is projected to the image plane pi as point U_c.
    - The x and y co-ordinates of the projected point can be derived from the similar triangles illustrated in Figure 9.4.

$$\mathbf{U}_c = \left[\, \frac{-fx_c}{z_c},\ \frac{-fy_c}{z_c},\ -f \,\right]^T \qquad (9.4)$$

(Eqn 2.2)

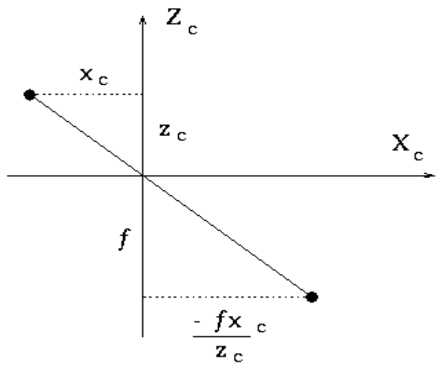


Fig. 2.11: Similar triangle for determining projections

Figure 9.4 *Calculation of the co-ordinates of the projected point.*

- *It remains to derive where the projected point U_c is positioned in the image affine co-ordinate system, i.e. to determine the co-ordinates which the real camera actually delivers.*
- The image affine co-ordinate system, with origin at the top left corner of the image, represents a shear and rescaling (often called the **aspect ratio**) of the image Euclidean co-ordinate system.
- The principal point U_0 - sometimes called the center of the image in camera calibration procedures is the intersection of the optical axis with the image plane pi. It is expressed in the image affine co-ordinate system as U_0a=[u_0,v_0,0]^T.

- The projected point can be represented in the 2D image plane pi in homogeneous co-ordinates as ~u = [U,V,W]^T, and its 2D Euclidean counterpart is u = [u,v]^T = [U/W,V/W]^T.
  - Homogeneous co-ordinates allow us to express the affine transformation as a multiplication by a single 3x3 matrix where unknowns a, b, c describe the shear together with scaling along co-ordinate axes, and u_0 and v_0 give the affine co-ordinates of the principal point in the image.

$$\tilde{\mathbf{u}} = \begin{bmatrix} U \\ V \\ W \end{bmatrix} = \begin{bmatrix} a & b & -u_0 \\ 0 & c & -v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{-fx_c}{z_c} \\ \frac{-fy_c}{z_c} \\ 1 \end{bmatrix} \qquad (9.5)$$

(Eqn 2.3)

- We aim to collect all constants in this matrix, sometimes called the **camera calibration matrix K**.
  - Since homogeneous co-ordinates are in use, the equation can be multiplied by any nonzero constant; thus we multiply by z_c to remove this parameter.

$$z_c\,\tilde{\mathbf{u}} = z_c \begin{bmatrix} -fa & -fb & -u_0 \\ 0 & -fc & -v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{x_c}{z_c} \\ \frac{y_c}{z_c} \\ 1 \end{bmatrix} = \begin{bmatrix} -fa & -fb & -u_0 \\ 0 & -fc & -v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} =$$

$$= \begin{bmatrix} -fa & -fb & -u_0 \\ 0 & -fc & -v_0 \\ 0 & 0 & 1 \end{bmatrix} R\,(\mathbf{X}_w - \mathbf{t}) = K \;\; R\,(\mathbf{X}_w - \mathbf{t}) \qquad (9.6)$$

(Eqn 2.4)

- The **extrinsic parameters** of the camera depend on the orientation of the camera Euclidean co-ordinates with respect to the world Euclidean co-ordinate system.
- The rotation matrix R expresses three elementary rotations of the co-ordinate axes -- rotations along the axes x, y, and z are termed **pan**, **tilt**, and **roll**, respectively.
- The translation vector t gives three elements of the translation of the origin of the world co-ordinate system with respect to the

camera co-ordinate system. **Thus there are six extrinsic camera parameters; three rotations and three translations.**

- The camera calibration matrix K is upper triangular as can be seen from the equation. The coefficients of this matrix are called **intrinsic parameters of the camera**, and describe the specific camera independent on its position and orientation in space.
- *If the intrinsic parameters are known, a metric measurement can be performed from images.*
- Assume momentarily the simple case in which the world co-ordinates coincide with the camera co-ordinates, meaning that X_w = X_c.
    - o Then equation (9.6) simplifies to

$$z_c\,\tilde{\mathbf{u}} = z_c \begin{bmatrix} U \\ V \\ W \end{bmatrix} = \begin{bmatrix} -fa & -fb & -u_0 \\ 0 & -fc & -v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} \qquad (9.7)$$

(Eqn 2.5)

- Two separate equations for u and v

$$\begin{array}{lllllllll} u & = \frac{U}{W} = & -fa\frac{x_c}{z_c} & - & fb\frac{y_c}{z_c} - u_0 & = & \alpha_u\frac{x_c}{z_c} & + & \alpha_{shear}\frac{y_c}{z_c} - u_0 \\ v & = \frac{U}{W} = & & - & fc\frac{y_c}{z_c} - v_0 & = & & & \alpha_v\frac{y_c}{z_c} - v_0 \,. \end{array} \qquad (9.8)$$

(Eqn 2.6)

where we make the substitutions alpha_u = -fa, alpha_shear = -fb, and alpha_v = -fc.

- Thus we have five intrinsic parameters, all given in pixels.
- The formulae also give the interpretation of the intrinsic parameters:
    - o alpha_u represents scaling in the u axis, measuring F in pixels along the u axis,
    - o alpha_v similarly specifies f in pixels along the v-axis.

   - alpha_shear measures in pixels in the v-axis direction how much is the focal length f coincident with u-axis slanted from the Y_i-axis.

- Returning to the general case given by the equation (9.6) ...if we express the scene point in homogeneous co-ordinates ~X_w = [X_w,1]^T, we can write the perspective projection using a single 3x4 matrix. The leftmost 3x3 submatrix describes a rotation and the rightmost column a translation
   - The delimiter | denotes that the matrix is composed of two submatrices.

$$\tilde{\mathbf{u}} = \begin{bmatrix} U \\ V \\ W \end{bmatrix} = [KR \,|\, -K\ R\mathbf{t}] \begin{bmatrix} \mathbf{X}_w \\ 1 \end{bmatrix} = M \begin{bmatrix} \mathbf{X}_w \\ 1 \end{bmatrix} = M\tilde{\mathbf{X}}_w \qquad (9.9)$$

(Eqn 2.7)

where ~X is the 3D scene point in homogeneous co-ordinates.

- The matrix M is called the **projective matrix** (also **camera matrix**).
- It can be seen that the camera performs a linear projective transformation from the 3D projective space P^3 to the 2D projective plane P^2.
- Introduction of projective space and homogeneous co-ordinates made the expressions simpler. Instead of the nonlinear equation, we obtained the linear equation.
- The 3x3 submatrix of the projective matrix M consisting of three leftmost columns is regular, i.e. its determinant is non-zero.
- The scene point ~X_w is expressed up to scale in homogeneous co-ordinates (recall that projection is expressed in the projection space) and thus all alpha, M are equivalent for alpha not equal to 0.

- Sometimes the simplest form of the projection matrix M is used.

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad (9.10)$$

(Eqn 2.8)

- This special matrix corresponds to the **normalized camera co-ordinate system**, in which the specific parameters of the camera can be ignored.
- This is useful when the properties of stereo and motion are to be explained in a simple way and independently of the specific camera.

Overview of single camera calibration

- The calibration of one camera is a procedure that allows us to set numeric values in the camera calibration matrix K or the projective matrix M.
- I. Intrinsic camera parameters only
  - If the camera is calibrated, and a point in the image is known, the corresponding line (ray) in camera-centered space is uniquely determined.
- II. Intrinsic and extrinsic parameters.
- Basic approaches to the calibration of a single camera.
- I. Known scene

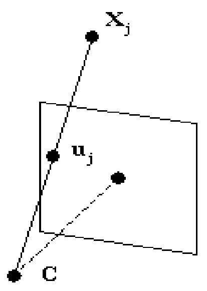


Figure 9.5 *Camera calibration from a known sce sponding pairs of scene points X_j and image points camera.*

Fig. 2.12: Calibration from a known scene

- A set of n non-degenerate (not co-planar) points lies in the 3D world, and the corresponding 2D image points are known.
    - Each correspondence between a 3D scene and 2D image point provides one equation

$$\alpha_j \tilde{\mathbf{u}}_j = M \begin{bmatrix} \mathbf{X}_j \\ 1 \end{bmatrix} \qquad (9.11)$$

(Eqn 2.9)

- The solution solves an over-determined system of linear equations.
- The main disadvantage is that the scene must be known, for which special calibration objects are often used.
- II. Unknown scene:

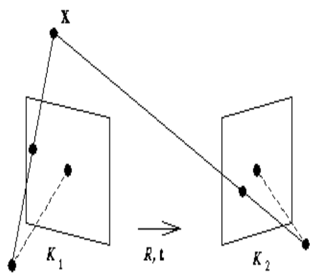


Figure 9.6 *Camera calibration from an unknown scene. At least two views are needed. It is assumed that the intrinsic parameters of the camera do not change, so $K_1 = K_2$.*

Fig. 2.13: Calibration from a unknown scene

- More views of the scene are needed to calibrate the camera.
- The intrinsic camera parameters will not change for different views, and the correspondence between image points in different views must be established.
- A. Known camera motion:
    - Both rotation and translation known:
        - This general case of arbitrary known motion from one view to another has been solved.
    - Pure rotation:

        - If camera motion is restricted to pure rotation, the solution can be found.
    - Pure translation:
        - The linear solution (pure translation) can be found.
- B. Unknown camera motion:
    - No a priori knowledge about motion, sometimes called **camera self calibration**.
    - At least three views are needed and the solution is nonlinear.
    - Calibration from an unknown scene is still considered numerically hard, and will not be considered here.

Calibration of one camera from the known scene

- Typically a two stage process.
- 1. The projection matrix M is estimated from the co-ordinates of points with known scene positions.
- 2. The extrinsic and intrinsic parameters are estimated from M.
    - (The second step is not always needed -- the case of stereo vision is an example.)
- To obtain M, observe that each known scene point X=[x,y,z]^T and its corresponding 2D image point [u,v]^T give one equation (9.11) - we seek the numerical values m_ij in the 3x4 projection matrix M.
- Expanding from Equation (9.11)

$$\begin{bmatrix} \alpha u \\ \alpha v \\ \alpha \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad (9.12)$$

$$\begin{bmatrix} \alpha u \\ \alpha v \\ \alpha \end{bmatrix} = \begin{bmatrix} m_{11}x + m_{12}y + m_{13}z + m_{14} \\ m_{21}x + m_{22}y + m_{23}z + m_{24} \\ m_{31}x + m_{32}y + m_{33}z + m_{34} \end{bmatrix} \qquad (9.13)$$

$$\begin{aligned} u(m_{31}x + m_{32}y + m_{33}z + m_{34}) &= m_{11}x + m_{12}y + m_{13}z + m_{14} \\ v(m_{31}x + m_{32}y + m_{33}z + m_{34}) &= m_{21}x + m_{22}y + m_{23}z + m_{24} \end{aligned} \qquad (9.14)$$

(Eqn 2.10)

- Thus we obtain two linear equations, each in 12 unknowns m_11, ... , m_34, for each known corresponding scene and image point.
    - If n such points are available, we can write the equations 9.14 as a 2n x 12 matrix

$$\begin{bmatrix} x & y & z & 1 & 0 & 0 & 0 & 0 & -ux & -uy & -uz & -u \\ 0 & 0 & 0 & 0 & x & y & z & 1 & -vx & -vy & -vz & -v \\ & & & & & & \vdots & & & & & \end{bmatrix} \begin{bmatrix} m_{11} \\ m_{12} \\ \vdots \\ m_{34} \end{bmatrix} = 0 \qquad (9.15)$$

(Eqn 2.11)

- The matrix M actually has only 11 unknown parameters due to the unknown scaling factor, since homogeneous co-ordinates were used.
- To generate a solution, at least six known corresponding scene and image points are required.
- Typically, more points are used and the over-determined equation (9.15) is solved using a robust least squares method to correct for noise in measurements.
- The result of the calculation is the projective matrix M.

- To separate the extrinsic parameters (the rotation R and translation t) from the estimated projection matrix M, recall that the projection matrix can be written as

$$M = [KR \,|\, -K\,R\mathbf{t}] = [A \,|\, \mathbf{b}] \qquad (9.16)$$ (Eqn 2.12)

- Determining the translation vector is easy; we substituted A = K R in equation (9.16), and so can write t = -A^-1 b.
- To determine R, note that the calibration matrix is upper triangular and the rotation matrix is orthogonal.
- The matrix factorization method called QR decomposition will decompose A into a product of two such matrices, and hence recover K and R.
- So far, we have assumed that the lens performs ideal central projection as the pinhole camera does.
- This is not the case with the real lenses.
- Such a typical lens performs distortion of several pixels.
- A human observer does not notice it if he looks at general scene.
- In the case an image is used for measurements, the distortion from the idealized pinhole model should be compensated.
- When **calibrating** a real camera, the more realistic model of the lens includes two distortion components.
- First, the **radial distortion** bends the ray more or less than in the ideal case.
- Second, the **decentering** displaces the principal point from the optical axis. Recall that the five intrinsic camera parameters were introduced in equation.
- Here, the focal length f of the lens is replaced by a parameter called the **camera constant**.

- Ideally, the focal length and the camera constant should be the same.
- In reality, this is true when the lens is focused at infinity. Otherwise, the camera constant is slightly less than the focal length. Similarly, the coordinates of the principal point can slightly change from the ideal intersection of the optical axis with the image plane.
- The main trick of the intrinsic parameters calibration is to observe a known calibration image with some regular pattern, e.g. blobs or lines covering the whole image. The observed distortions of the pattern allow estimating the intrinsic camera parameters.
- Both the radial distortion and the decentering can be treated in most cases as rotationally symmetric, often modeled as polynomials.
- Let u, v denote the correct image coordinates; ~u, ~v denote the measured uncorrected image coordinates that come from the actual pixel coordinates x, y and the estimate of the position of the principal point ^u_0, ^v_0.

$$\begin{aligned}\tilde{u} &= x - \hat{u}_0\\ \tilde{v} &= y - \hat{v}_0 \qquad (9.18)\end{aligned}$$

(Eqn 2.13)

- The correct image coordinates u, v are obtained if the compensations for errors delta u, delta v are added to the measured uncorrected image coordinates ~u, ~v.

$$\begin{aligned}u &= \tilde{u} + \delta u\\ v &= \tilde{v} + \delta v \qquad (9.19)\end{aligned}$$

(Eqn 2.14)

- The compensations for errors are often modeled as polynomials in even powers to secure the rotational symmetry property.

- Typically elements up to maximally degree six are considered.

$$\begin{aligned} \delta u &= (\tilde{u} - u_p)(\kappa_1 r^2 + \kappa_2 r^4 + \kappa_3 r^6) \\ \delta v &= (\tilde{v} - v_p)(\kappa_1 r^2 + \kappa_2 r^4 + \kappa_3 r^6) \end{aligned} \qquad (9.20)$$

(Eqn 2.15)

where u_p, v_p is the correction to the position of the principal point.

- The r^2 is the square of the radial distance from the centre of the image.

$$r^2 = (\tilde{u} - u_p)^2 + (\tilde{u} - u_p)^2 \qquad (9.21)$$

(Eqn 2.16)

- Recall that ^u_0, ^v_0 were used in equation (9.18).
  - The u_p, v_p are corrections to ^u_0, ^v_0 that can be applied after calibration to get the proper position of the principal point.

$$\begin{aligned} u_0 &= \hat{u}_0 + u_p \\ v_0 &= \hat{v}_0 + v_p \end{aligned} \qquad (9.22)$$

(Eqn 2.17)

- The typical radial distortion of the lens for the simple second order model is a special case of the equation, i.e. no decentering is assumed and second order polynomial approximation is considered

$$\begin{aligned} u &= \tilde{u}(1 \pm \kappa_1(\tilde{u}^2 + \tilde{v}^2)) \\ v &= \tilde{v}(1 \pm \kappa_1(\tilde{u}^2 + \tilde{v}^2)) \end{aligned} \qquad (9.23)$$

(Eqn 2.18)

- The original image was a square pattern. The distorted images are shown

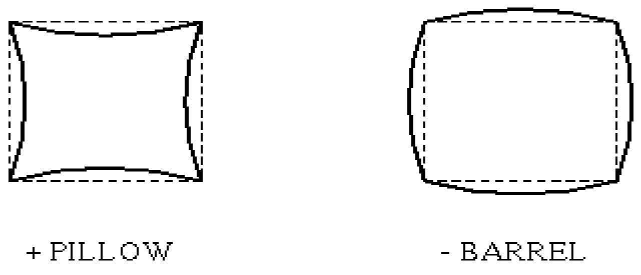


Fig. 2.14: Lens distortions

Figure 9.7 *Radial distortion of a off shelf lens.*

- The left part of the figure shows the pillow like distortion (minus sign in the equation, whereas the right part depicts the barrel like distortion corresponding to the plus sign.
    - There are more complicated lens models that cover tangential distortions that model such effects as lens decentering.

2.8 Stereopsis (two cameras)

- Calibration of one camera and knowledge of the co-ordinates of one image point allows us to determine a ray in space uniquely.
- If two calibrated cameras observe the same scene point X, its 3D co-ordinates can be computed as the intersection of two such rays.
- This is the basic principle of **stereo vision** that typically consists of three steps:
    - Camera calibration;
    - Establishing point correspondences between pairs of points from the left and the right images

- Reconstruction of 3D co-ordinates of the points in the scene.

Fig. 2.15: Stereoscopic vision

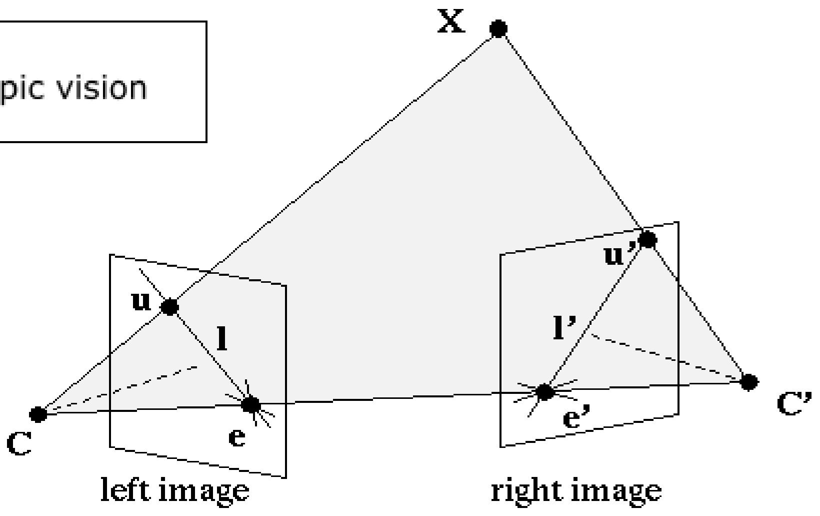


Figure 9.8 *Epipolar geometry in stereopsis.*

- The line connecting optical centers C and C' is called the **baseline**. Any scene point X observed by the two cameras and the two corresponding rays from optical centers C, C' define an **epipolar plane**. This plane intersects the image planes in the **epipolar lines** l, l'. When the scene point X moves in space, all epipolar lines pass through **epipoles** e, e' - the epipoles are the intersections of the baseline with the respective image planes.
- Let u, u' be projections of the scene point X in the left and right images respectively. The ray CX represents all possible projections of the point X to the left image, and is also projected into the epipolar line l' in the right image.
- The point u' in the right image that corresponds to the projected point u in the left image must thus lie on the epipolar line l' in the right image.
  - This geometry provides a strong **epipolar constraint** that reduces the dimensionality of the search space for a correspondence between u and u' in the right image from 2D to 1D.

- A special arrangement of the stereo camera, called the canonical configuration is often used.
    - The baseline is aligned to the horizontal co-ordinate axis, the optical axes of the cameras are parallel, the epipoles move to infinity, and the epipolar lines in the image planes are parallel.

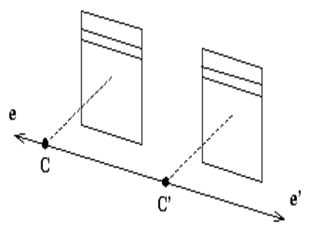


Figure 9.9 *The canonical stereo configuration where the epipolar lines are parallel in the image, and epipoles move to infinity.*

Fig. 2.16: Canonical stereo configuration

- For this configuration, the computation is slightly simpler.
- It is easier to move along horizontal lines than along general lines. The geometric transformation that changes a general camera configuration with nonparallel epipolar lines to the canonical one is called image rectification.
- There are practical problems with the canonical stereo configuration, which adds unnecessary technical constraints to the vision hardware.
    - If high precision of reconstruction is an issue, it is better to use general stereo geometry since rectification induces resampling that causes loss of resolution.
- Let's consider an easy canonical configuration and recover depth.
- The optical axes are parallel, which leads to the notion of disparity that is often used in stereo literature.
    - In Figure, we have a bird's eye view of two cameras with parallel optical axes separated by a distance 2 h.

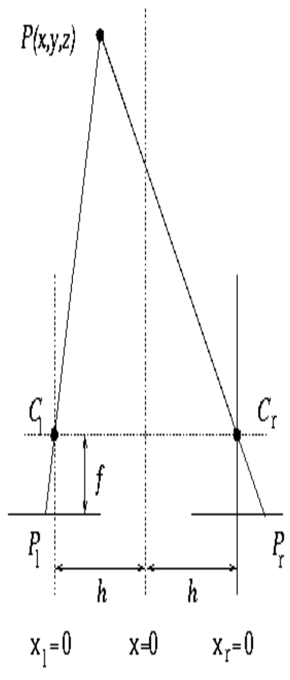


Fig. 2.17: Depth recovery from the canonical stereo configuration

Figure 9.10 *Elementary stereo geometry in canonical configuration.*

- The images they provide, together with one point P with co-ordinates (x,y,z) in the scene, showing this point's projection onto left (P_l) and right (P_r) images.
- The co-ordinates have the z axis representing distance from the cameras (at which z=0) and the x axis representing horizontal distance (the y co-ordinate, into the page, does not therefore appear).
- x=0 will be the position midway between the cameras; each image will have a local co-ordinate system (x_l on the left, x_r on the right) which for the sake of convenience we measure from the center of the respective images; that is, a simple translation from the global x co-ordinate.
- P_l will be used simultaneously to represent the position of the projection of P onto the left image, and its x_l co-ordinate - its distance from the center of the left image (and similarly for P_r).
- It is clear that there is a disparity between x_l and x_r as a result of the different camera positions (that is, | P_l - P_r | >

0); we can use elementary geometry to deduce the z co-ordinate of P.

- P_l, C_l and C_l, P are the hypotenuses of similar right-angled triangles.
    - h and f are (positive) numbers, z is a positive co-ordinate and x, P_l, P_r are co-ordinates that may be positive or negative, we can then write:

$$\frac{P_l}{f} = -\frac{h+x}{z} \qquad (9.24)$$

(Eqn 2.19)

- and similarly from the right hand side of Figure 9.10

$$\frac{P_r}{f} = \frac{h-x}{z} \qquad (9.25)$$

(Eqn 2.20)

- Eliminating x from these equations gives

$$z\,(P_r - P_l) = 2hf \qquad (9.26)$$

(Eqn 2.21)

- and

$$z = \frac{2hf}{P_r - P_l} \qquad (9.27)$$

(Eqn 2.22)

- Notice in this equation that P_r - P_l is the detected disparity in the observations of P.
- If P_r-P_l = 0 then z = \infinity.
- Zero disparity indicates the point is (effectively) at an infinite distance from the viewer.

3. Description of the IRIS engine

3.1 Components of IRIS

Structural descriptions of various vision systems may vary with possibly minor localised differences in most cases. The structure presented here is motivated by our ongoing work on a vision system for 'intelligent' machines (mobile or otherwise) called IRIS.

For ease of design, a system like IRIS is composed of two related, interdependent segments. As of now, they are:

- IRIS-XT: This is the segment primarily responsible for preprocessing incoming images to compensate for inadequate quality, luminance, sharpness, contrast, color distribution and a host of other attributes. It is also responsible for performing 2D machine vision tasks like line-following and shape recognition. IRIS-XT works primarily using two color spaces, RGB and HSL. Some of the tasks are mathematical regression (line-following), quadtree decomposition (color recognition) and perimeter sampling (shape recognition). More detailed descriptions of a generic IRIS-XT-like image processing system are given later.
- IRIS-3D: This is the segment primarily responsible for analysis, reconstruction of 3D models and stereovision. Some operations that may be performed upon the image in this stage are space carving, stereoscopic vision and 3D map building combined with localisation. Success in the set task is achievable by using one or a combination of these and other techniques. A basic set of techniques required for a IRIS-3D-like system is discussed later.

In addition, several extensions to IRIS also exist which further assist in abstracting the hardware details of the robot from the algorithm programmer. These are discussed later under IRIS Extensions.

3.2 The data structures: IRIS Foundation

3.2.1 What is IRIS Foundation ?

IRIS Foundation contains all the data structures required by the other, higher-level modules. It also contains many algorithms (trivial or otherwise) which are frequently used by the vision modules. Many of its structures are templatised, so that they can be used in a variety of situations.

3.2.2 A rough structural description

IRIS Foundation resides in the namespace Comrade::IrisFoundation, so to use it, you'll either have to refer to its data types and functions in a fully qualified fashion, or bring in the whole thing with the using directive.

The classes currently implemented in the IRIS Foundation are as follows:

- **BitmapStreamer**: This class is a filter for reading and writing to 24-bit bitmap files. Filters for reading other formats were not designed intentionally because in the final version, IRIS-XT and IRIS-3D will read in the images from the memory itself (placed there by the framegrabber driver **libfg** for Linux).
- **RGB** *and* **RGBstruct**: These two data structures are for storing RGB values for use by other modules. RGBstruct is the normalised version while the values of components in an RGB structure can be between 0 and 255. The RGB model is one of several color spaces.
- **HSLstruct**: This data structure is another way of representing color, and is used by the image processing algorithms more often than the RGB and RGBstruct types. HSLstruct represents the HSL color space.

- **ColorSpaceConvertor**: This class encapsulates the conversion functions between the RGB and HSL color spaces. Note that this operates only on individual pixels.
- **Buffer**: This is the single most important class used by the higher modules. It is mostly used to store images in a way similar to bitmaps. Being templatised, it can represent RGB-style or HSL-style images (or any other image format that may be used). It is also used for other purposes like camera flag arrays in 3D reconstruction. It is resizeable and supports deep copy semantics.
- **RGB_BufferManager**: This class operates exclusively on RGB Buffer objects and decouples the Buffer class from the actual source of a RGB image. In addition, it supports copying and pasting of rectangular blocks between RGB blocks.
- **BufferConvertor**: This class has a number of static functions which are used for converting between Buffer structures of different color spaces. Most image processing routines using the HSL color space for operation make use of the functions in this class.
- **Tree**: This class is used for pyramidal image representations. Pyramidal representations allow representation of scenes at different scales, thus permitting variable scale-space analysis. Currently, only the quadtree segmentation code uses this class, but more applications of this class are expected soon.
- **QuadtreeSegmenter**: This class performs quadtree segmentation upon images using the Tree structure. Parameters can be varied to give segmentation at different scales.

In addition, there are other classes like ByteStreamer and functions like min() and max(), but these are not meant to be used directly by the application programmer.

3.3 The image preprocessor: IRIS-XT

3.3.1 What is IRIS-XT?

IRIS-XT is the component of the IRIS vision system which is responsible for basic image processing tasks as well as 2D machine vision functions like line following and object detection, IRIS-XT uses the classes defined by IRIS Foundation and also defines some data structures of its own. IRIS-XT is the second oldest component of the entire system and thus is reliable and easy to use.

Detailed documentation is not the aim of this section, which provides a general overview of IRIS-XT's structure, so as to give the potential reuser a better inital understanding of the system architecture.

3.3.2 Architecture of the IRIS-XT subsystem

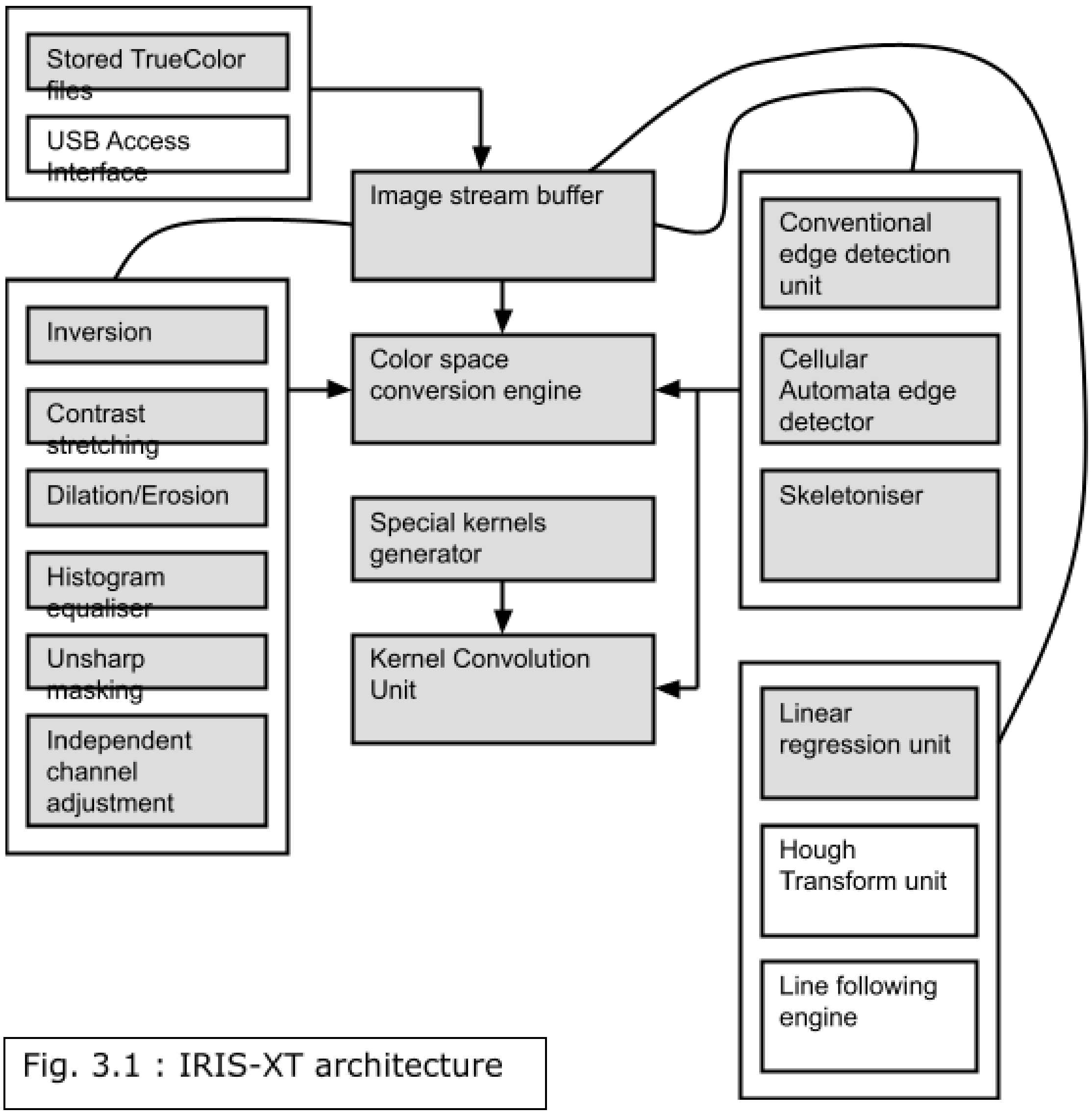


Fig. 3.1 : IRIS-XT architecture

3.3.3 A rough structural description

IRIS-XT resides in the namespace Comrade::IrisXT, so to use it, you'll either have to refer to its data types and functions in a fully qualified fashion, or bring in the whole thing with the using directive.

The classes currently implemented in the IRIS-XT are as follows:

- **KernelMatrix**: This structure holds the values of the kernel during the convolution of an image with a mask. It can be of any size, and can be generated automatically in some situations like edge detection and Gaussian masking by setting appropriate parameters to desired values.
- **KernelOperator**: This class is responsible for the actual convolution process between a HSL image and a KernelMatrix mask. It also perform greyscale convolution of a RGB image.
- **KernelGenerator**: This class can generate special standard masks which are used frequently enough to warrant automated generation. Most of these masks are used different edge detectors. They are:
    1. Roberts' cross operator
    2. Sobel's operator
    3. Prewitt's operator
    4. Isotropic operator
    5. Laplacian operator (zero crossing operator)

Other than that, there is the Gaussian kernel which is used for the Canny edge detector.

- **EdgeDetectorManager**: So as to ease programming, this class shields the programmer from explicitly defining kernels for edge detection. The kernels are generated internally by calling the appropriate function and passing the specified arguments.

- **CA_EdgeDetector**: This class implements a very simple, but useful edge detector of my own design. It is based on the principles of Cellular Automata, and it is certainly faster than convolution, though the results show some noise. But it may be adequate for most purposes.

- **AlgorithmManager**: This class encapsulates all functions which cannot be achieved using kernel convolution. There are several functions defined here, and all are useful in their own right. They are:

    1. Dilation
    2. Erosion
    3. Conversion to greyscale image
    4. Conversion to negative image
    5. Independent RGB channel adjustment
    6. Unsharp masking
    7. Range compression
    8. Contrast stretching
    9. Histogram equalisation

In addition, there are some internal functions which perform calculations on images, but these are not meant to be accessed by the programmer.

- **StraightLineDetector**: This class performs the Hough Transform on an image already processed by an edge detector and determines possible straight lines. The result is an angle with respect to some reference axis. This is currently under development, i.e., not tested thoroughly as yet. Actually, another function regression() is also capable of finding straight lines but is more susceptible to noise, though faster.

- **BasicCircleDetector**: This class performs the Hough Transform on an image already processed by an edge detector and determines possible circles. The result is a coordinate pair with respect to some reference axis, and the radius of the circle detected. This is currently under development, i.e., not tested thoroughly as yet.
- **BasicShapeDetector**: This class, together with the StraightLineDetector and CircleDetector class, forms the Hough Transform Engine. This class performs the Generalised Hough Transform on an image, and will detect arbitrary shapes even in the presence of noise and/or occlusions.
- **ShapeSampler**: This class is used to sample edge-detected shapes, i.e., build a table of distances between some point and the (potential) perimeter of a shape in increments specified by the programmer. This gives a shape table which may be correlated with some stored shape table or may be used as a prototype shape table for future matching by direct comparison. This class is also used by the Hough Transform Engine.
- **SequenceAnalyser**: This class is used to correlate shape tables to provide a measure of the degree of correlation between them. The presence of noise and false terminations is accomodated for by providing a cumulative error tolerance model. This provides an alternative to the Hough Transform Engine, but more work is required to increase the robustness of the method used.

Besides these, there is an experimental skeletoniser algorithm which awaits further development. Also note that most of the functions implemented in IRIS-XT can be applied to any rectangular region within an image, instead of the entire image. This provides opportunities for optimisation.

3.4 The 3D reconstruction system: IRIS-3D

3.4.1 What is IRIS-3D?

IRIS-3D is the component of the IRIS vision system responsible for endowing COMRADE with a sense of the third dimension - depth, distance, whatever you call it - thus enabling us to program robots to perform even more complex tasks.

IRIS-3D is a newer component of the IRIS system and thus is extremely prone to changes (almost daily in an average). But most of the architecture is in place; it remains to tune the algorithms or replace them using better/faster ones. IRIS-3D already shows promising results, and will only get better with time.

3.4.2. Architecture of the IRIS-3D subsystem

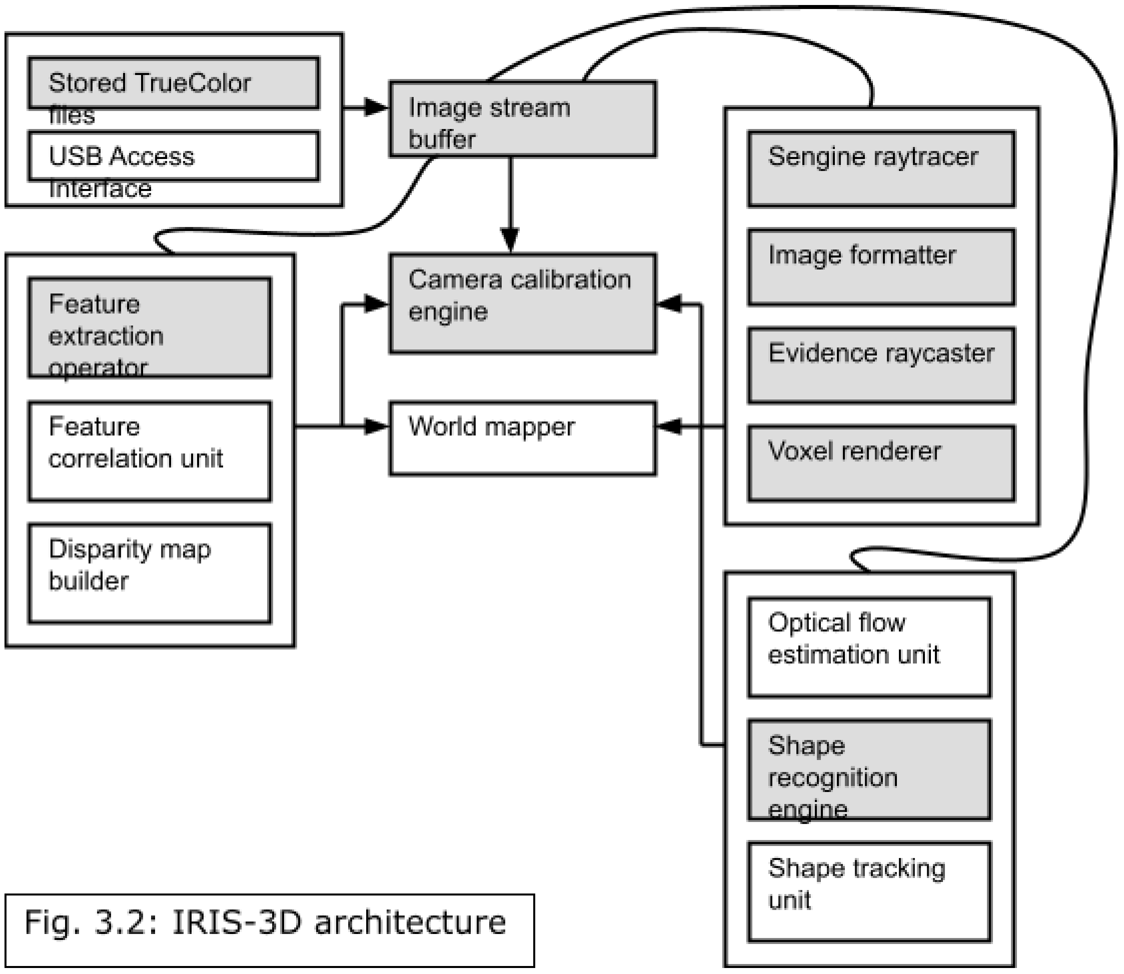


Fig. 3.2: IRIS-3D architecture

3.4.3 A rough structural description

IRIS-3D resides in the namespace Comrade::Iris3D, so to use it, you'll either have to refer to its data types and functions in a fully qualified fashion, or bring in the whole thing with the using directive.

The classes currently implemented in the IRIS-3D are as follows:

- **Matrix4x4**: This structure is almost exclusively used for homogeneous 3D operations on a point. For efficiency purposes, therefore, its dimensions are fixed to 4-by-4.
- **Coordinate**: This is IRIS' basic 3D point structure. It can be translated, rotated or multiplied by any arbitrary Matrix4x4 object. Note that the axis of rotation can also be arbitrary, thus making this a very flexible structure at the lowest hierarchical level.
- **Voxel**: This represents the basic volume element in 3D space and is used in the 3D space carver engine. It is important to note that since the voxel is a finite cube, its footprint will be more than one pixel if it is 'projected' onto a screen during a projective transformation.
- **Sensor**: This class represents the mathematical model of the imaging sensor, i.e., the camera. Strictly speaking, there should be other functions in addtion to the ones present to optimise the calibration matrix of the camera in a least-mean-squares sense. Also, the camera model should also model (at a minimum) a first-order projective distortion. All of this is currently being worked upon, though the model as it is adequate for approximate results.
- **VoxelWorker**: This class is responsible for calculating various mathematical interactions of a Voxel object with its environment; for example, determining the footprint of a voxel on an arbitrary plane.

- **WorldSpace**: This represents a cuboidal arrangement of voxels (a 'box') and is used by the space carver engine to reconstruct the model from its N projective views within this space.

Besides the above, there are a few other data structures like Point and Parametric, but they are not for direct use by the programmer.

3.4.4 Stereovision and space carving

The two most important algorithms in IRIS-3D are currently not encapsulated inside classes because they are not the final versions. Nevertheless, they still provide useful results. They are as follows:

1. **Stereovision algorithm**: It enables binocular stereoscopic vision, by analysing pairs of images. Currently, the method implemented is a fixed-window correlation method, which gives useful results already. However, this causes the so-called *corona* effect at discontinuities, in addition to being slow. For this reason, a new fast multiresolution, variable window, stereovision algorithm has been designed. Initial results are already interesting.

2. **Space carving engine**: This allows the robot to reconstruct the 3D model of an object (upto an approximation) from N calibrated views of the same. This uses the basic space carving algorithm given by Kyros Kutulakos. Currently, it performs reconstruction with a very good degree of accuracy. Photorealism will involve mapping the image texture onto the model, and is nice to look at, but not very useful right now. Also, until semiautomatic camera calibration is brought online, this will not be of much use because it requires the camera positions to perform its calculations.

3.5 Brief description of a new stereovision algorithm

The spirit of the algorithm is based on the premise that for most purposes, depth maps of subpixel precision are not necessary. However, there will arise situations where calculations at high resolutions is required. This algorithm addresses this need effectively. The basic stereo algorithm based on area matching uses windows of fixed size which cause the so-called corona effect at discontinuities. This algorithm shows how to avoid that effect and simultaneously achieve multiresolution stereo vision in realtime.

Introduction

The problem of stereo vision is a much-studied one; several solutions have been proposed. Most of these solutions are concerned with the recovery of accurate depth maps with minimum number of mismatches. These algorithms are ranked with reference to some ground truth. However, they are not suited for realtime implementation, not at least without the use of specialised hardware. Moreover, it is to be noted that humans do not need an especially accurate depth map for reasonable stereo vision. That is, if the errors in recovering the depth map are kept below some maximum level, it is to be expected that most of the relevant informmation can still be recovered without significantly affecting plant performance.

This is the basic compromise that our algorithm makes. It forgoes complicated analyses which might determine and compensate for the effects of occlusion and other effects. However, the possible resulting loss in performance is made up to a very large extent by using the variable window approach. The resulting performance in terms of speed is very good, with errors kept within acceptable limits. Indeed, we intend to use this particular approach to build 3D models of the environment in realtime, which involves the fusion of multiple depth maps. Most errors in the individual depth maps are averaged out in the final 3D map.

These are the following steps of the algorithm:

- Segment the left (or right) image using quadtree segmentation with some set parameters.

- Correlate the blocks of the left (or right) image with its exact counterparts in the right (or left) image. Assign uniform depth to each block separately.

4. Platform issues

4.1 Intoduction

A discussion on programming platform and optimisation issues is essential when considering a (largely) platform independent design. At this point, it is impossible to be general. Thus, specific references to operating systems and processors will be made throughout this discussion.

As emphasised throughout, the faster the available processor, the better. However, support, maintenance and readily available information upon a chosen processor are also important criteria for processor selection. Currently, this implies working largely on the Intel x86 platform. IRIS is projected to run (very slowly) on a baseline configuration speed of 200 MHz. Higher speeds will improve its performance.Information on the Intel x86 platform is easily available. We demonstrate the IRIS vision system running on a Pentium IV 2.4 GHz uniprocessor mounted on the robot itself.

However, alternatives are not impossible. We can easily envisage running IRIS on the current generation of ARM processors. The Intel StrongARM SA-1110 and its variations are rated well for their low power consumption and high speeds. These characteristics make them suitable for functioning in embedded microprocessor environments needing machine vision. However, the peak performance of the SA-1110 is equivalent to a 500 MHz Celeron processor; thus IRIS in its complete form will not run fast enough. Instead, a stripped version of IRIS, named IRIS-Lite is being planned to fit into a smaller memory and processing footprint.

IRIS has been developed using Standard C++. This (and C) is the language of choice for any programmer desiring speed and economy of expression. Standard C++ mandated by the ANSI/ISO X3J16 Committee is being used. No nonstandard features or libraries are being made use of. Thus, the basic IRIS system is capable of being compiled under Windows or Linux environments without a single glitch.

However, a study of the assembly language generated by current C++ compilers reveals that the quality of the code leaves much to be desired. Ideally, we should waste no more cycles than necessary. But that means writing IRIS entirely in x86 assembly language, rendering it nonportable. Thus, the best approach is to write the majority of IRIS in C++, substituting assembly language in time-critical sections of the code. A generic C++ version of IRIS will also be available for processor platforms not supported by IRIS at the machine code level. The assembly part of the code will have to be rewritten for the particular processor used, resulting in several optimised ports of IRIS, and one Standard C++ port.

4.2 The Win32 platform

There are, of course, differences between the two versions. IRIS was designed with the Linux platform in mind. However, platform independence was achieved in most areas because the bulk of IRIS algorithms were written under the Win32 platform. Here we describe the Win32 platform development system.

Operating system	Windows 95/98/Me/NT/200/XP
Compiler system	MinGW system (gcc 3.3.1)
Development environment	Dev-C++ 4.9.8/ MinGW Developer Studio 2.05
Minimum processor make, speed and RAM	Pentium MMX (200 MHz) with 32 MB RAM
Recommended processor make, speed and RAM	Athlon XP (2.4 GHz) with 192 MB RAM

4.3 The Linux platform

Linux was the real target operating system IRIS was designed for. The full functionality of IRIS is apparent when run on Linux, since many system-dependent features (like accessing the USB webcam and proper CPC functionality) are coded specifically for Linux.

Operating system (Linux build)	Debian (unstable Fedora core)
Compiler system	GCC system (gcc 3.3.3)
Development environment	Anjuta
Minimum processor make, speed and RAM	Pentium MMX (200 MHz) with 32 MB RAM
Recommended processor make, speed and RAM	Athlon XP (2.4 GHz) with 192 MB RAM

4.4 Fast interface design using FLTK

Interface is a major design component in any software which aims to interact with the user. IRIS is intended to run as a library under the COMRADE Preproduction Core, thus does not require a front end. However, the offline demo of IRIS requires the user to interact with it as an application, not as a library. Thus, fast interface design is a necessity. To this end, we have used the FLTK (pronounced "fulltick") library for GUI development. FLTK stands for **Fast Light Tool Kit**. It excels at rapid deployment of GUI's in a consistent fashion. In fact, the GUI development for IRIS was completed in FLTK in less than eight hours. The version of FLTK used is FLTK v2.00 (unstable version).

5. Results from the IRIS engine

Below we present some results after processing using the IRIS engine. They are presented in no particular order, and represent only a fraction of the potential of IRIS.

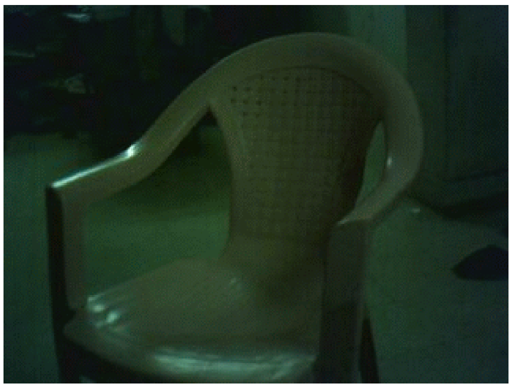


Fig. 5.1a: Before histogram equalisation

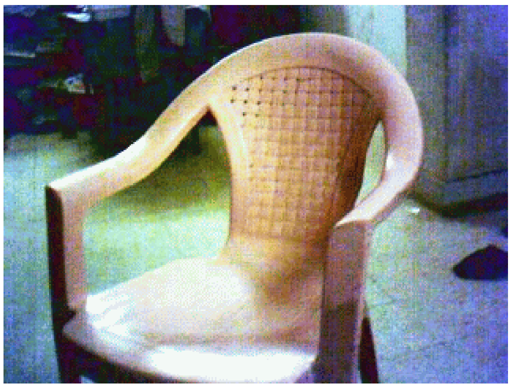


Fig. 5.1b: After histogram equalisation

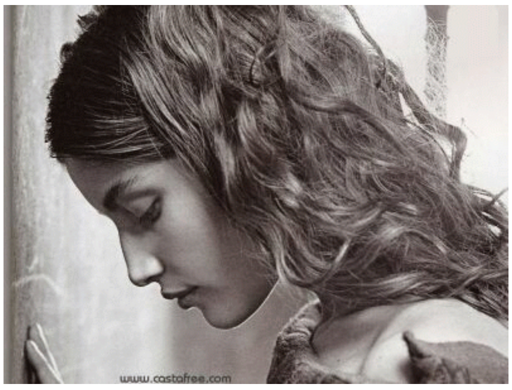


Fig. 5.2a: Before edge detection

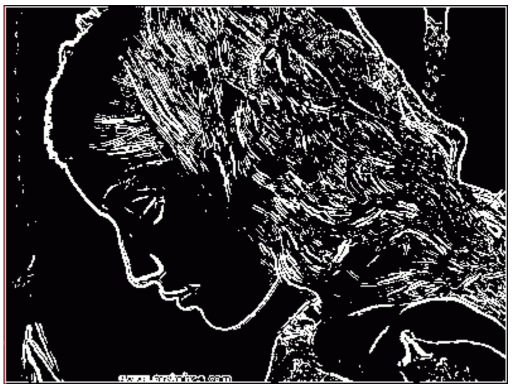


Fig. 5.2b: After edge detection

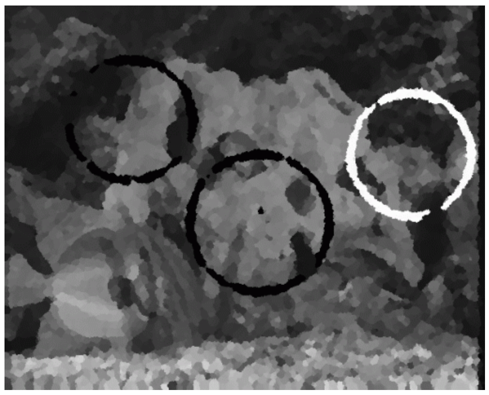


Fig. 5.3a: Noisy image with circles

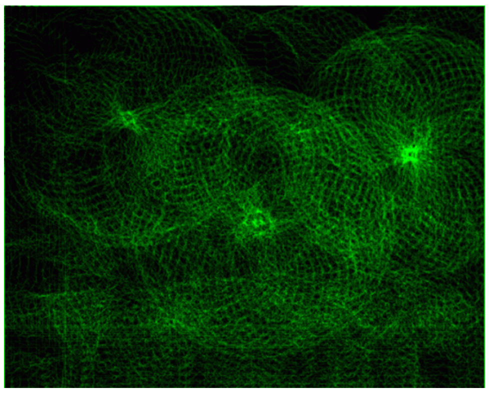


Fig. 5.3b: Hough Transform with detected centers

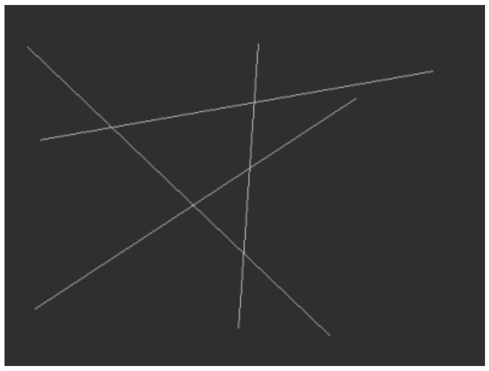


Fig. 5.4a: Simple line detection test

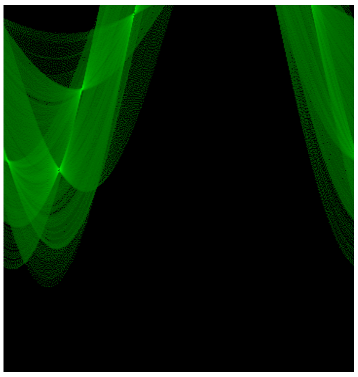


Fig. 5.4b: Hough Transform with detected lines (bright spots)

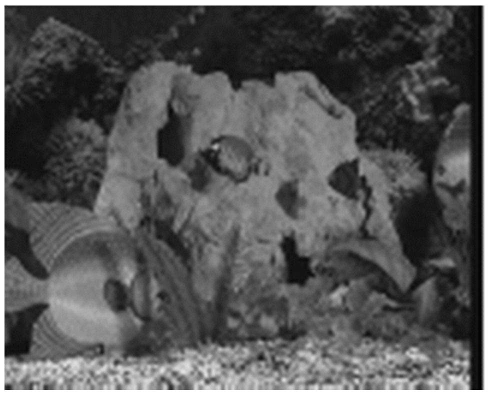


Fig. 5.5a: Left image of a stereo pair

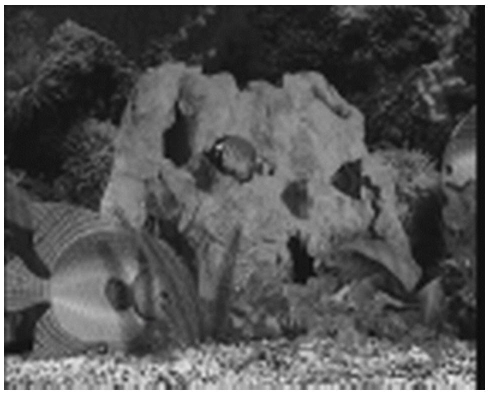


Fig. 5.5b: Right image of a stereo pair

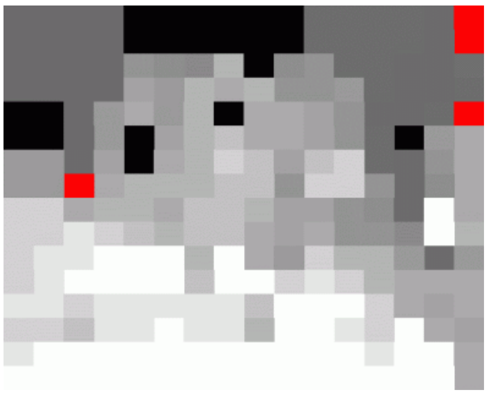


Fig. 5.6a: Depth map at low resolution

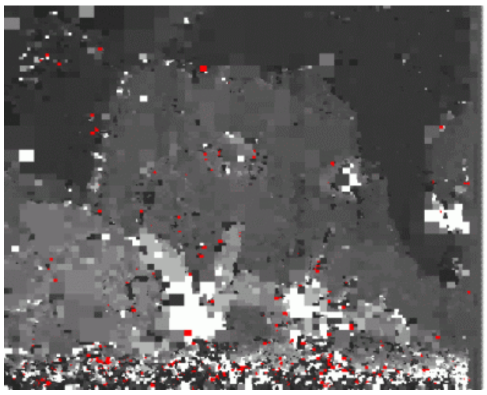


Fig. 5.6b: Depth map at high resolution

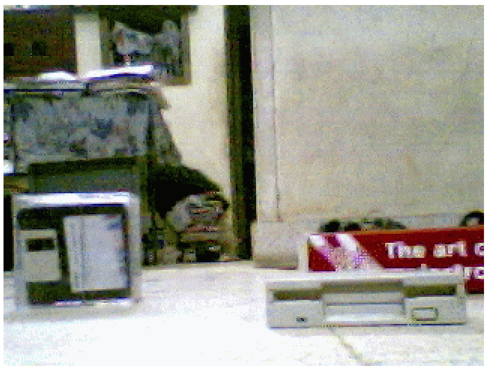


Fig. 5.7: Before quadtree segmentation

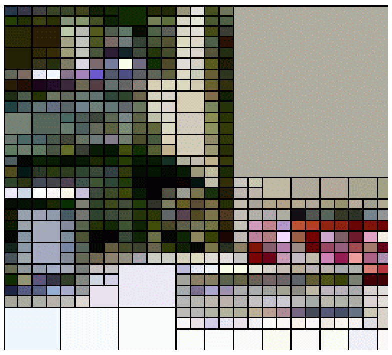


Fig. 5.8a: Medium block size

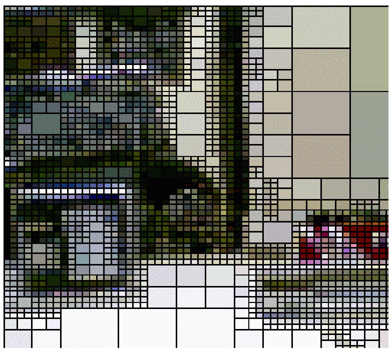
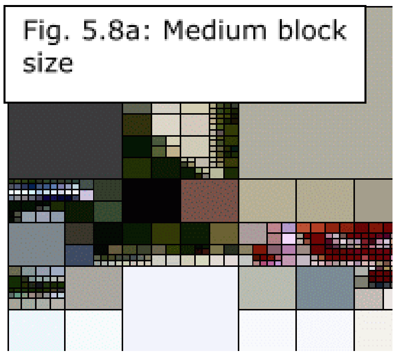
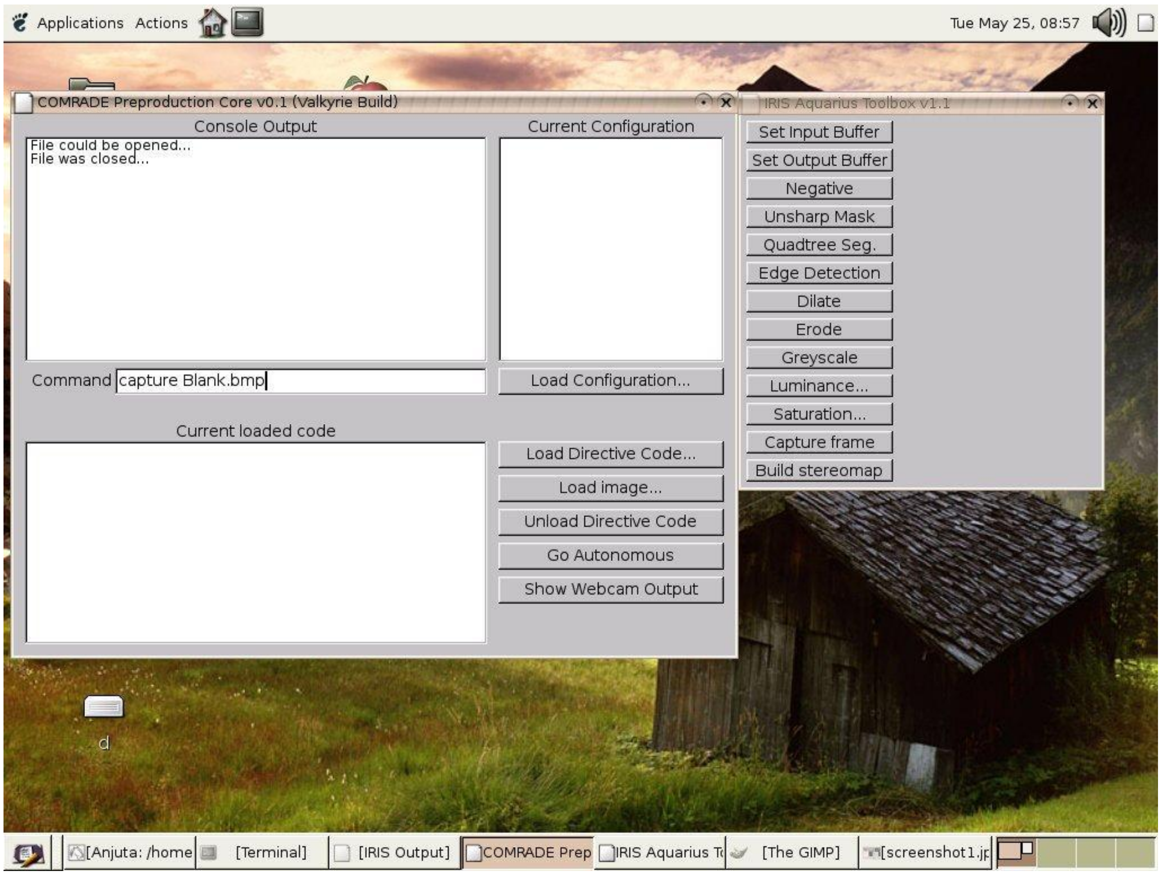


Fig. 5.8a: Medium block size

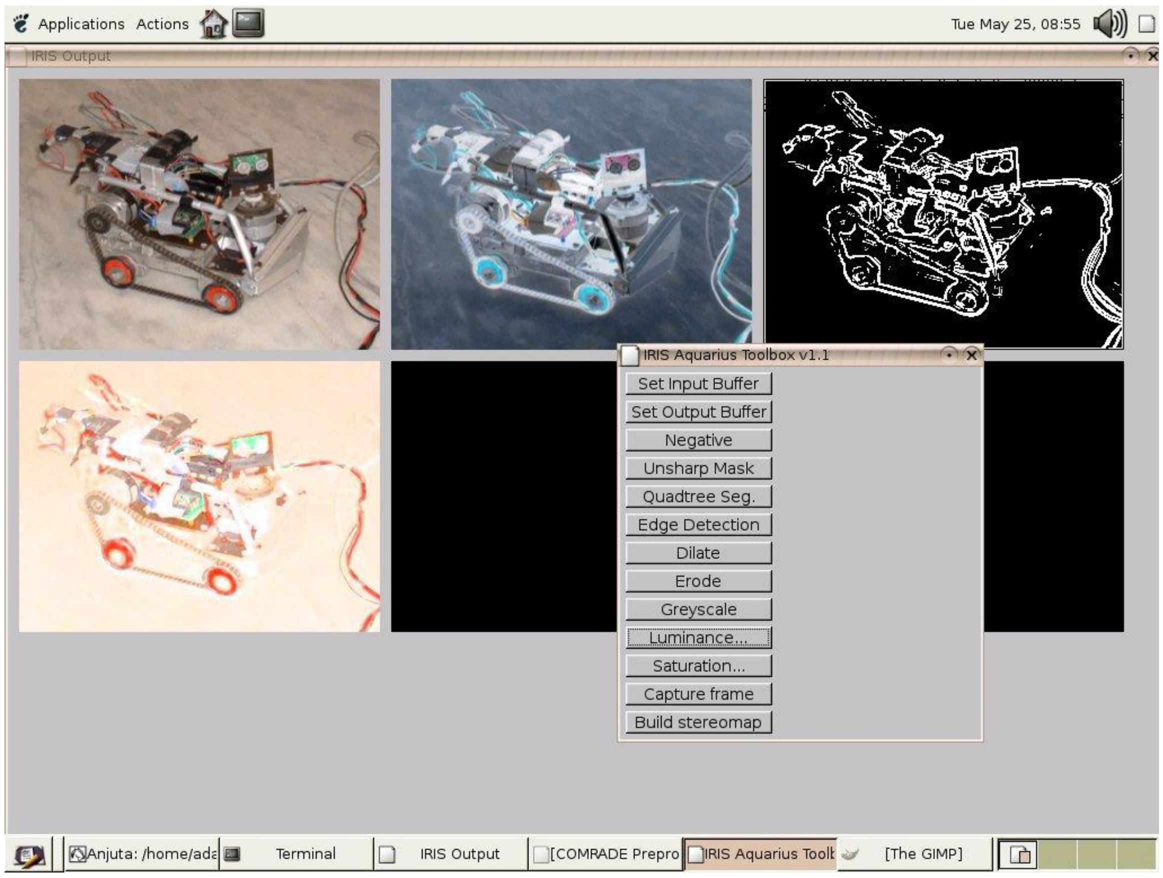


Screenshots of the IRIS Offline Preproduction Core



```
File could be opened...
File was closed...
```

Command: capture Blank.bmp


Applications
Actions
Tue May 25, 08:55
IRIS Output
IRIS Aquarius Toolbox v1.1
Set Input Buffer
Set Output Buffer
Negative
Unsharp Mask
Quadtree Seg.
Edge Detection
Dilate
Erode
Greyscale
Luminance...
Saturation...
Capture frame
Build stereomap
Anjuta: /home/ada
Terminal
IRIS Output
[COMRADE Prepro
IRIS Aquarius Toolb
[The GIMP]

6. Extensions and improvements

6.1 Introduction

These are extensions and improvements currently being planned for IRIS. Some are already in the development stage (though untested) and the others are still in the design phase.

6.2 IRIS Extensions

6.2.1. The Osiris abstraction system

Until the day, robot production is as standardised as that of computers or cars, they will have very different hardware implementations. The entire architecture may vary radically from robot to robot depending upon the intended application. This means that the software has to be written specifically for each robot. This causes waste of coding man-hours. To this end, the Osiris abstraction system is being developed as an adjunct to the IRIS system. Osiris is not related to IRIS as such, but grew out of our need to make IRIS fit into a host of (possibly) different software environments.

When finished, Osiris will provide a unified software model for the algorithm developer who will not have to worry about the underlying specifics and intricacies of the robot hardware involved. It will abstract away the sensor models as well as the control signals needed to be sent to the robot. This will be in the form of a layered query-based system which will be maximally modular and reuseable.

6.2.2. COMRADE Runtime

COMRADE will have different levels of control. It may assume autonomous control, or it may be issued commands by the user, or it may be used in its offline state. All these different methods of control ultimately pass the same basic commands to the robot proper; only the method of issuing them differs depending upon the interface. The COMRADE Runtime system is what allows multiple control interfaces to be hooked up to the COMRADE interpreter system, so that any desired form of control may be implemented with minimal reorganisation of the code.

6.2.3. The BorgCode decompositor

One of the most important (but late) stages of Project COMRADE is projected to be the development of an easy and extensible cooperative task-solving platform. There are several levels of complexity that we can build into the system, but whatever the solution adopted, it must be integrated seamlessly into the current architecture. Here, we present a (very) brief proposal for a high-level hybrid robot communications scripting language, called BorgCode. Note that this is very preproduction and has not been implemented yet; options are still open.

BorgCode allows the end-user to specify high-level directives in a sequential fashion which are then decomposed by the BorgCode Decompositor (BCD) into separate parallelly executable chunks of code, after which the rest of the cooperative environment operation follows. BorgCode is simple, because inspite of any complicated-looking code, internally it is interpreted as a state machine with a (hopefully) simple implementation.

BorgCode does not solve everything; further contention resolution is required among the machines. I'll develop a possible structure for a signal buffer and a scheme of contention resolution later.

A First Example

An example directive:

```
000 START
010 UNITS=1: LineFollow,ObjDetect;
020 UNITS=4: AutoExplore, MapBuild;
030 IF (REF(10):UNITS(1).ObjDetect==TRUE)
```

```
040 [
050     REF(20):UNITS(2 OF 4).Go(REF(10):UNITS(1).Target);
060 ]
070 IF (REF(10):UNITS(1).TimeOut==TRUE)
080 [
090     REF(10):UNITS(1).Stop
100     REF(20):UNITS(4 OF 4).Stop
110 ]
120 END
```

The BorgCode script is loaded into the Main Node. The script is initially completely sequential; it is the job of the BorgCode Decompositor to break down the code into parallelly executable chunks of code. Along the way, signals to respond to as well as contention resolution must be performed. However, these functions are not performed by the decompositor; the field machines do this themselves. Thus, the above script may be decomposed as below:

Chunk 1

```
010 UNITS=1
Signal(START): ResetTime,LineFollow,ObjDetect
Signal(OBJ_DET_TRUE,010): ResetTime,Multicast(OBJ_DETECT_TRUE,010)
// Note that this code is not wrong, the signal buffer can be filled by external multicast data as well as those from the machine's own program itself. The program generates this signal in the machine's buffer, which the machine then multicasts to the others. The signal also contains target information.
Signal(TIME_OUT_TRUE): Stop
ENDS 010
```

Chunk 2

```
020 UNITS=4
Signal(START): ResetTime,AutoExplore,MapBuild
Signal(OBJ_DETECT_TRUE,010): ResetTime,Go(OBJ_DETECT_TRUE.Target,010(1))
// The (1) is there if more than one machine were assigned to detection and detected positive.
Signal(TIME_OUT): Stop
ENDS 020
```

The above solution involving chunks has a problem. Note the line from the first program:

```
050     REF(20):UNITS(2 OF 4).Go(REF(10):UNITS(1).Target);
```

Here, 2 of the 4 units are supposed to go to that target. However, nowhere is this mentioned in the second chunk. The reason that we can separate this part out of the program easily from the program is because we can also scan for the UNITS(x OF Y) statements for this. But such a situation may easily occur many times. We can do this by introducing a general signal-based solution into BorgCode; by introducing the VOTE_WIN signal. The new chunks would thus be:

Chunk 1

```
010 UNITS=1
[
        Signal(START): ResetTime,LineFollow,ObjDetect;
        Signal(OBJ_DET_TRUE,010): ResetTime,Multicast(OBJ_DETECT_TRUE,010);
// Note that this code is not wrong, the signal buffer can be filled by external multicast data as well as those from the machine's own program itself. The program generates this signal in the machine's buffer, which the machine then multicasts to the others. The signal also contains target information.
        Signal(TIME_OUT_TRUE): Stop;
];
```

Chunk 2

```
020 UNITS=4
[
        Signal(START): ResetTime,AutoExplore,MapBuild;
        Signal(OBJ_DETECT_TRUE,010): UNITS=2
        [
                Signal(VOTE_WIN): ResetTime,Go(OBJ_DETECT_TRUE.Target,010(1))
                Signal(REACHED): Stop;
        ];

// The (1) is there if more than one machine were assigned to detection and detected positive.
Signal(TIME_OUT): Stop;
];
```

The extra syntax is there for ease of parsing. More information regarding BorgCode syntax will be released progressively with further refinements.

7. The future

7.1. Summary

This report described the current state of the IRIS vision engine. Though it is yet to support very advanced routines like automatic camera calibration (which involve complex algorithms like Levenberg-Marquardt minimisation), these will soon be incorporated over time into the easily extensible framework. Development on IRIS continues even as this written, and it is foreseeable that IRIS will be a contribution, even if a minor one, to the machine vision community.

7.2. Applications

- Real time industrial applications
- Remote surveillance machines like planetary rovers
- Offline/online satellite imagery analysis and reconstruction
- Commercial image processing applications
- As a testbed for more advanced machine vision algorithms

7.3 Conclusion

Machine vision is a massive field. It is plain impossible to attempt to summarise all current developments. Emergence of new technologies has resulted in cheaper sensors and MEMS (micro-electro-mechanical system) arrays. It is clear that software cannot fill all the needs of machine vision. Analog solutions are needed. Nevertheless, software vision analysis systems will continue to thrive in the near future. IRIS is a simple, yet effective solution to this end. Theoretical advances like cellular neural networks and pulse coupled neural networks are also paving the way for an unstoppable synergy between hardware and software. As the pace of these developments accelerates, only the future can tell what lies in store for machine vision.

THE END