Pfinder.

Vision & Modeling Group.
MIT Media Laboratory; 20 Ames St., Cambridge, MA 02139

October 16, 1995

1 Introduction

Pfinder is a real-time system for tracking and interpretation of people. It runs on a standard SGI Indy computer, and has performed reliably on thousands of people in many different physical locations. The system uses a multi-class statistical model of color and shape to segment a person from a background scene, and then to find and track people's head and hands in a wide range of viewing conditions. Pfinder produces a real-time representation of a user useful for applications such as wireless interfaces, video databases, and low-bandwidth coding, without cumbersome wires or attached sensors.

2 Structure

The structure of the program is shown in figure 2.

The code is written in C and in C++ because of historical reasons. It is basically composed of two different types of code:

1. Backsub code: All the modules coming from this code start by **bs** and it's written in C. It performs tasks as segmentating the background from the foreground (bsClassify.c), finding the silhouette of the person (bsSegment.c) or determining the color classes of the foreground blobs (bsColor.c).

2. pfinder code: This is C++ code that extracts the contour of the person (pfContour.c++), the features (pfFeatures.c++), interprets the gestures (pfGestures.c++), applies connectivity constraints in the growing process (pfMorphology.c++).

The principal parts of it are:

2.1 Main program: pfinder-main.c++

The main program can be called with the following options:

```
 pfinder [-{nvX}] [-alive] [-expert] [-b $<$number $>$ ]  [-d $<$decimati$>$]
[-D $<$number$>$] [-T] [-t] [-R] [-single] [-seeds] [-S] [-E $<$erode$>$]
[-w $<$width$>$] [-h $<$height$>$] [-size $<$height$>$ $<$width$>$] [-z $<$zoom$>$]
[-f $<$max$>$] [-in $<$conn$>$] [-nb $<$number$>$] [-mn $<$stdev$>$] [-an $<$stdev$>$]
[-gs <x$>$ <y$>$] [-gl <x$>$ <y$>$] [-gp <sit$>$ <bent$>$] [-gm <move$>$] [-pg]
[-plrf $<$freq$>$] [-plrm] [-hand $<$min$>$ $<$max$>$] [-noslow] [-help]
[-dumpfile $<$file$>$] [-statfile $<$file$>$] [-contourfile $<$file$>$] [-savefile $<$file$>$] [-chroma $<$chroma$>$]

Required Arguments:
```

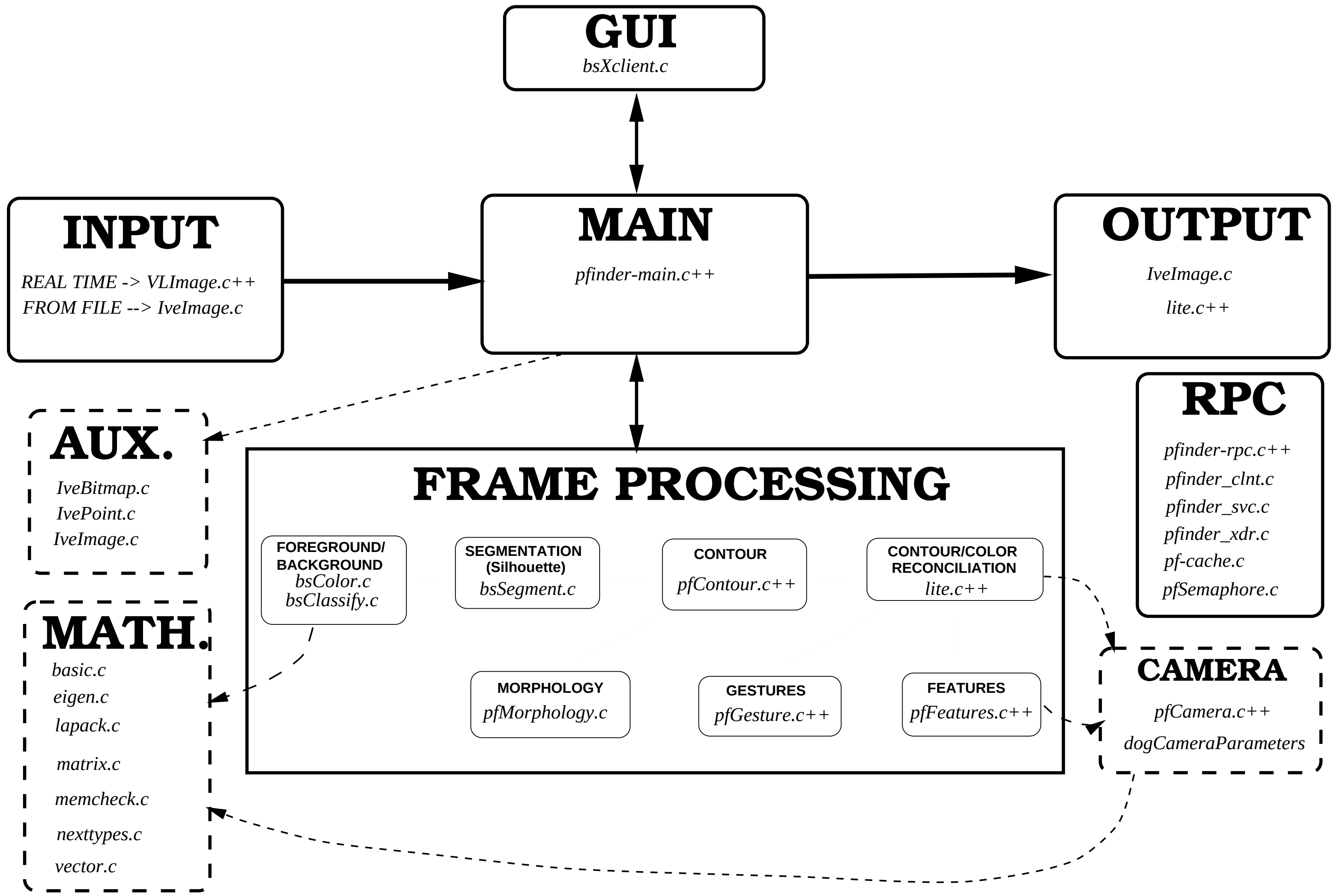

Figure 1: Pfinder block structure.

```
Optional Arguments:

        -n : Toggle whether to use ALIVE interface [1|0]

        -v : Main program verbose mode on [0|1]
                bsColor and VL modules verbose

        -X : Use X Interface [1|0]

        -alive : Alive Demo

        -expert : Expert Mode (anti-ALIVE)

        -b $<$number$>$ :
                $<$number$>$ = Number of BG Frames (integer) [40]

        -d $<$decimati$>$ :
                $<$decimati$>$ = Decimation factor (integer) [1]

        -D $<$number$>$ : Dump mode on, images from VL to cout
                $<$number$>$ = Number of frames to dump (integer) [0]

        -T : Thad mode: track green and red blobs

        -t : Stat mode: vlob centers to cout

        -R : Read mode on, images fm cin

        -single : loop on first captured image [0|1]

        -seeds : mark zoomgrow seeds on image [0|1]

        -S : Save mode on, bitmaps to cout

        -E $<$erode$>$ :
                $<$erode$>$ = erode multiple (integer) [10]

        -w $<$width$>$ :
                $<$width$>$ = Width of images (integer) [640]

        -h $<$height$>$ :
                $<$height$>$ = Height of images (integer) [480]

        -size $<$height$>$ $<$width$>$ :
                $<$height$>$ = Height of images (integer) [480]
                $<$width$>$ = Width of images (integer) [640]
```

```
-z $<$zoom$>$ :
        $<$zoom$>$ = Zoom factor (integer) [4]

-f $<$max$>$ :
        $<$max$>$ = Max number of foreground classes (integer) [8]


-in $<$conn$>$ :
        $<$conn$>$ =
                0: INDY_COMPOSITE
                1: INDY_SVIDEO
                0: GALILEO_SVIDEO_1
                1: GALILEO_SVIDEO_2
                2: GALILEO_SVIDEO_3
                3: GALILEO_COMPOSITE_1
                4: GALILEO_COMPOSITE_2
                5: GALILEO_COMPOSITE_3
                6: GALILEO_YRB_1
                7: GALILEO_YRB_2
                         (integer) [1]

-nb $<$number$>$ :
        $<$number$>$ = Number of VL buffers to use (integer) [3]

-mn $<$stdev$>$ :
        $<$stdev$>$ = KF: Std dev of mmt noise (float) [1]

-an $<$stdev$>$ :
        $<$stdev$>$ = KF: Std dev of accel noise (float) [1]

-gs $<$x$>$ $<$y$>$ :
        $<$x$>$ = x small (float) [0.3]
        $<$y$>$ = y small (float) [-0.35]

-gl $<$x$>$ $<$y$>$ :
        $<$x$>$ = x large (float) [0.38]
        $<$y$>$ = y large (float) [0]

-gp $<$sit$>$ $<$bent$>$ :
        $<$sit$>$ = sit (float) [0.8]
        $<$bent$>$ = bent (float) [0.9]

-gm $<$move$>$ :
        $<$move$>$ = move (float) [2]

-pg : print gestures [0|1]
```

```
-plrf $<$freq$>$ :
        $<$freq$>$ = freq of report (integer) [10000]

-plrm : report moves [0|1]

-hand $<$min$>$ $<$max$>$ :
        $<$min$>$ = min hand (integer) [40]
        $<$max$>$ = max hand (integer) [80]

-noslow : Turn off rate reporting [1|0]

-help : Prints usage

-dumpfile $<$file$>$ :
        $<$file$>$ = File to dump input to (string) []

-statfile $<$file$>$ :
        $<$file$>$ = File to save statistics to (string) []

-contourfile $<$file$>$ :
        $<$file$>$ = File ot save contours to (string) []
-savefile $<$file$>$ :
        $<$file$>$ = File to save segments to (string) []

-chroma $<$chroma$>$ : use chromakey instead of aquiring background
        $<$chroma$>$ = chroma to key on %2x%2x%2x (string) []
```

None of the previous options is necessary to run the program. They have different purposes, such as verbose mode, debugging information, specifying the dimensions of the input images or the output files where to dump some useful information.

The first taks to do is the initialization of all the structures handled by the program. Depending on the options typed by the user some special initializations must be done. Among the initializations that are performed, we have: `lite::initialize()` for initialising `lite.c++`, `bsClassifyInitialize` for the background classes, `bsXInitialize` for the X interface as well as the procedures for creating the user's display.

The program can read the input images from a file or from the camera. In both cases, it starts an infinite loop and process each frame by calling the internal function `processFrame`. If there is an X user interface, then it also handles all the events coming from the client. This function performs all the processing to the frames. All the functionality of pfinder is done within it. The flow of the program is as follows:

2.1.1 Background extraction.

Before starting the processing of the input images, pfinder needs to characterize the background. In order to do that, it reads the first `numberOfBGFrames` of the sequence. These frames cannot have any moving object in them because the program assumes that they are just background and extracts all the background information. Once it has read the frame number `numberOfBGFrames`, it calls the `bsClassifyMakeBackgroundClasses` to make the background classes. This fuction just calls another function:`bsColorComputeBackgroundClassesBatch`, which computes the initial background classes statistics (mean and covariance). The initial values are **heuristics**.

2.1.2 Finding the silhouette.

The following step is to find the silhouette of the person. This task is carried out by the function `bsSegmentFindSilhouette` which calls the function `Grow` for growing the silhouette from a seed point:

- If the centroid (seed) is the initial one (0,0), then it calls `PixelClass` which assigns to this centroid a foreground class value.
- If the centroid is different from (0,0) then it calls `GetEdgePoint` that looks for an edge point applying some heuristics thresholds.

Then it gets the number of foreground classes (`bsClassifyGetNumClasses`). For each class, it gets the mean of the class (`bsClassifyGetClassPosition`) and a confidence heuristics measure about whether a pixel belongs to the class or not.

Next it looks for the feet: it defines a bounding box (`pfMainGetBoundingBox`) from the centroid point and looks for the right and left at the bottom part of this box.

All the pixels that verified any of the previous conditions are marked as 'burning points'. Once it has finished this task, it looks at a neighborhood of all the burning points using a subsample factor and marks the pixels as 'visited'. Besides if the pixel belongs to the foreground, it is added to the 'catching' array. Both the 'burning' and 'catching' arrays are used for iteratively refining the definition of the various seed blobs. Finally it returns the centroid of the largest of the various seed blobs. During all the process the 'burning-catching' points are also loaded in the `seed_points_for_morphology` array, for the posterior Morphology computation.

The result is the `centroid` -of type `Point`- of the seed blob of the silhouette.

2.1.3 Setting the Foreground Statistics (`bsClassifySetForegroundStats`):

This function calls two additional functions:

1. `bsColorComputeForegroundClasses` that updates the foreground classes statistics;
2. and `bsColorInitClasses` that sets to 0 the accumulated mean and covariance values, so that in the next update of the statistics the accumulated values start from 0.

2.1.4 Updating the Background (`bsClassifyUpdateBG`):

The background pixels are also updated by calling the function `bsColorUpdateBackgroundPixel`. This latter function computes the next state of the visible background pixels by applying a simple adaptive filter with update rate (`BGUPDATERATE`). This approach allows to compensate for changes in lighting and even for object movement.

2.1.5 Contour Generation (`contour.generate`):

It looks first for the bottom limit of the contour. Then it looks for a good side start point and, once it has found such a point, it starts following clockwise the contour from this initial point.

2.1.6 Contour Interpretation (`lite::interpretPfContour`):

Firstly it checks whether there is a person or not in the image by calling the `features.ProjectPfContourToUserPlane` function, that projects to the user plane the contour information given the camera parameters and determines if the contour is to be considered of a person or not. If there is a person, then finds his/her features. The function `PFind` does this task: it validates first the feet, then the head and the hands. At this moment there are two different representations: the color and the contour representation that must be reconciled. The function `_reconcileWithColorClasses` accomplishes this task: it reconciles each of the features by calling `_reconcileColorClass` with the kind of feature as parameter. This latter function sets confidence values for color and contour, computes scores for contour and color, the euclidean distance between both representations and chooses to trust either the contour feature or the color feature.

The last step is the update of the **Kalman Filter** that predicts each feature in the following image using the current state model. One part of the update is the interpretation of the body, posture and hands actions. The other part is the estimation of the following state.

Once it has updated the Kalman Filter it reads the following frame and repeats all the processing explained above.