

USC Robotics Laboratory
University of Southern California
Los Angeles, California, USA

Mezzanine User Manual

Version 0.00

Andrew Howard

This document may not contain the most current documentation on Mezzanine. For the latest documentation, consult the Player/Stage homepage:
<http://playerstage.sourceforge.net>

May 8, 2002

Contents

1	Introduction	1
1.1	Description	1
1.2	History	2
1.3	Licence	2
1.4	System Requirements	2
1.5	Getting Mezzanine	2
1.6	Building and Installing	3
1.7	Bugs	3
1.8	Acknowledgements	3
2	QuickStart Guide	4
3	Using mezzaine	5
3.1	Physical Set-up	5
3.2	Hardware	5
3.3	Running mezzanine	5
3.4	How it works; in brief	6
3.5	Configuration	7
3.5.1	fgrab : frame-grabber interface	8
3.5.2	classify : color classification	8
3.5.3	blobfind : color blob segmentation	8
3.5.4	dewarp : image dewarping; image-to-world transformation	9
3.5.5	ident : object identification	9
4	Using mezzcal	11
4.1	Running mezzcal	11
4.2	Calibration	11
4.2.1	Image mask	13
4.2.2	Color classes	13
4.2.3	Blob properties	14
4.2.4	Dewarping	15
5	Using libmezz	16
5.1	Compiling and linking	16
5.2	Functions	16
5.3	Structures	17

A	IPC Specification	18
A.1	Signal Handling	18
A.2	Memory-Mapped File Layout	18

Chapter 1

Introduction

Mezzanine is an overhead 2D visual tracking package intended primarily for use as a mobile robot metrology system. It uses a camera to track objects marked with color-coded fiducials, and infers the pose (position and orientation) of these objects in world coordinates. Mezzanine will work with most color cameras (even the really cheap ones), and can correct for the barrel distortion produced by wide-angle-lenses.

Mezzanine is language and architecture neutral: object poses are written into a shared memory-mapped file which other programs can read and make use of as they see fit. Mezzanine itself is written in pure C and was developed under Linux on x86 hardware.

1.1 Description

Mezzanine consists of three main components:

- `mezzanine` : the tracking program.
- `mezzcal` : a calibration program (for settings colors, etc).
- `libmezz` : an IPC library for communicating with `mezzanine`.

mezzanine : the tracking program

`mezzanine` captures images from a frame-grabber using Video4Linux, classifies and segments the images into colored blobs, groups the blobs into known types of objects, computes the world coordinates of these objects using a ground plane constraint (i.e., all objects are assumed to be either on the ground or at some constant height above the ground), and writes the location of objects into a shared memory-mapped file. Other processes can from read this file using `libmezz`, or through their own native memory-mapped file API.

On a 700Mhz PIII, `mezzanine` is able to process 30 frames/second (i.e. the full NTSC frame-rate) while utilising about 70% of the CPU.

mezzcal : the calibration program

`mezzcal` is a gui tool used calibrate the system. Users can control the color classes, blob properties, and image-to-world transformation through a simple drag-and-drop interface. `mezzcal` is witten using RTK2 (a GUI toolkit for robotics applications), which is in turn based on GTK+ (the GIMP Toolkit).

libmezz : the IPC library

The `libmezz` library provides a convenient C interface to `mezzanine`. It defines a small set of functions and structures through which client programs written by users can interact with `mezzanine`. Note that this library is provided mainly for convenience: users are free to write native interfaces to `mezzanine` in any language they see fit, so long as that language supports access to shared memory-mapped files.

See Appendix A for a complete IPC reference, and check out the `examples` directory for sample programs.

1.2 History

Mezzanine was developed at the USC Robotics Research Laboratory as part of the Player/Stage project <http://playerstage.sourceforge.net>, and is based partly on code developed for the University of Melbourne's RoboCup team <http://www.cs.mu.oz.au/robocup>.

1.3 Licence

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

1.4 System Requirements

Mezzanine is known to work on Linux/x86 using a 2.4 kernel. Video capture is done using Video4Linux, so it should work with any frame-grabber card supported by V4L. Mezzanine also requires the following third-party libraries:

- Video4Linux : Video support for Linux. See <http://www.video4linux.net/>.
- GTK+ 1.2 : The Gimp Toolkit is present on pretty much every Linux distro, and most other Unix'es. See <http://www.gtk.org>
- GSL 0.9 or above : The GNU Scientific Library is available as package in most distros, but often not installed by default. See <http://sources.redhat.com/gsl/>

1.5 Getting Mezzanine

The latest release of Mezzanine can be found at <http://playerstage.sourceforge.net>. From here you can also get the latest, bleeding edge version from the CVS repository.

1.6 Building and Installing

Mezzanine does not currently use autoconf, so you may have to tweak the make files a little to get it to compile on your system. On the other hand, if you already have the required third-party libraries (see above), and you're using a standard Linux distro, you should be able to just

```
make; make install
```

For more detailed instructions, open the Mezzanine tarball and read the README in the top-level directory.

1.7 Bugs

Mezzanine is research software, and is bound to contain some bugs. If you've broken something (or better yet, fixed something), please submit a bug report to <http://sourceforge.net/projects/playerstage>. Make sure you include the Mezzanine and OS version, together with a detailed description of the problem. While there is no warranty on this software, we'll try to fix things for you.

1.8 Acknowledgements

This work is supported by DARPA grant DABT63-99-1-0015 (MARS) and NSF grant ANI-9979457 (SCOWR), and possibly others.

Many thanks to Gavin Baker (gavinb@antonym.org) for providing the Video4Linux interface (`libfg`).

Chapter 2

QuickStart Guide

If your frame-grabber card is installed and working, and is available as `/dev/video0`, you should be able to start Mezzanine by typing:

```
$ mezzanine --fgrab.norm NTSC
```

or

```
$ mezzanine --fgrab.norm PAL
```

depending on whether you're using an NTSC or PAL camera. You should see some introductory information about versions, etc, then a list of numbers showing timing information. In another terminal, you can now start the calibration program:

```
$ mezzcal
```

This should pop up three windows: one with an image from the camera overlaid with all sorts of cryptic symbols, another showing a YUV color-space with some random colored pixels in it, and a third containing a table of blob and object properties. You are now ready to calibrate the system, for which you will need to read Chapter 4. Once the system is calibrated, you can start writing your own client programs; see the `examples` directory for some code samples.

If the above sequence gave you no joy, you might want to try some of the following:

- Make sure `mezzanine` and `mezzcal` are in your executable path.
- Make sure you have read-write permission on `/dev/video0`.
- Is your camera plugged in/turned on?
- Is your frame-grabber supported by Video4Linux? Some cards wont actually generate images, even though V4L seems to run ok. Try running `xawtv` to see if you can get a picture.

If it still doesnt work, you will probably need to read either the V4L documentation or the rest of this manual.

Chapter 3

Using mezzanine

3.1 Physical Set-up

Mezzanine is designed to track *fiducials*, i.e., color coded markers attached to the objects we wish to track. Each fiducial is composed of a pair of solid colored circles, as shown in Figure 3.1. Any pair of colors can be used, but best results are achieved when the colors are strongly saturated and are well separated in color space; day-glo orange and green are a good combination, for example. The size and separation of the fiducials will depend upon the quality of the camera and its distance from the area of interest.

The fiducials allow Mezzanine can determine the position and orientation of objects, but not their identity. We therefore rely on track-continuation to generate consistent object labels. As a consequent, labels may swap if two fiducials come into very close proximity.

Mezzanine is designed to work with an overhead camera, as shown in Figure 3.2(a). Best results are achieved when the camera is located centrally over the area of interest and is oriented to point directly down. Mezzanine can also handle oblique camera angles, but tracking accuracy will be decreased to some extent.

Figure 3.2(b) shows a typical image taken through an overhead camera. Note the barrel distortion produced by the wide-angle lens; Mezzanine uses a second-order polynomial to remove this distortion, then uses a projective transform to convert from image to world coordinates. See Chapter 4 for details on how to calibrate this transformation.

3.2 Hardware

Mezzanine uses Video4Linux to capture images from a frame-grabber. Video4Linux supports most of the popular chipsets, including the Conexant series (BT848, BT878 and so on). See the Video4Linux documentation for information on installing and using frame-grabbers under Linux (<http://www.video4linux.net/>).

Mezzanine will work with pretty much any color camera, but will obviously give better results when used with good hardware. RGB cameras are particularly nice (but very expensive); using S-Video rather than Composite-Video will also help.

3.3 Running mezzanine

Start `mezzanine` by typing:

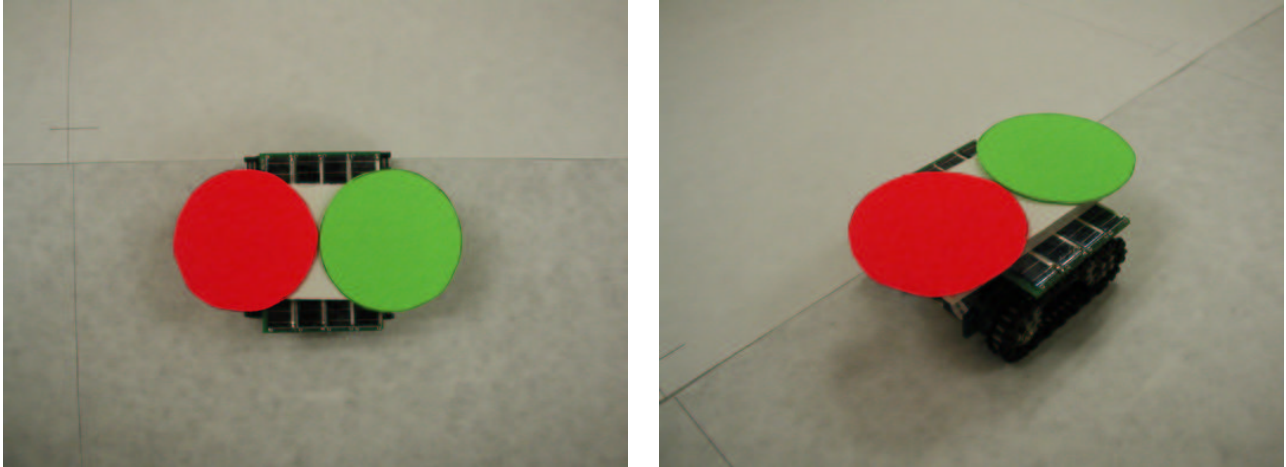


Figure 3.1: A fiducial marker composed of pink and green circles; each circle has a diameter of 50 mm. This fiducial has been mounted on top of a small tracked robot.

```
$ mezzanine
```

the program will read the default configuration file, then print out some version and timing information. The program can be stopped at any time using \hat{C} . To start `mezzanine` with a custom configuration file, use:

```
$ mezzanine myfile.opt
```

Options specified in the configuration file can also be over-ridden on the command line. For example:

```
$ mezzanine --fgrab.norm PAL myfile.opt
```

will command the frame grabber to acquire in PAL mode, irrespective of the settings in the configuration file.

Mezzanine's performance (measured in frames-per-second) will depend on the CPU and front-side bus speed. On a 700MHz Pentium III, `mezzanine` has been shown to run at the video frame rate (25Hz for PAL and 30Hz for NTSC) with about 80% CPU utilization. Note, however, that even on lightly loaded machines, `mezzanine` may still drop occasional frames; it does not run as a real-time process.

3.4 How it works; in brief

The algorithm for `mezzanine` is as follows.

1. `fgrab` : block until the next frame arrives.
2. `classify` : apply a mask to the image to decide which parts of the image should be processed and which should be ignored. The mask is a simple polygon: only those points inside the polygon will be processed.
3. `classify` : classify each pixel in the masked image as belonging to either one of the user-specified classes, or to none of these classes. Classification is done in YUV space.

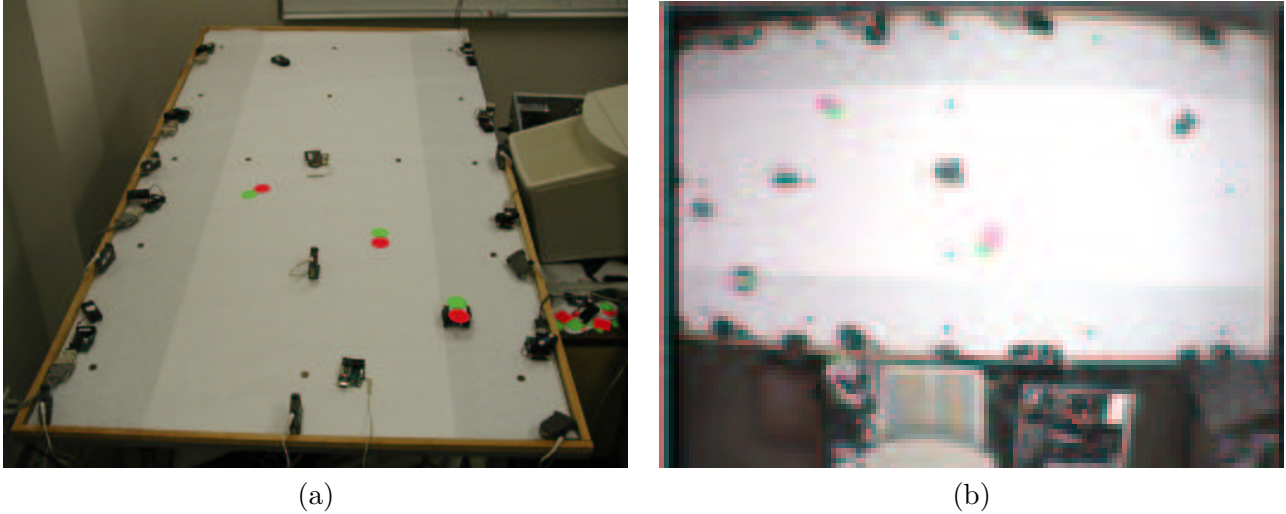


Figure 3.2: (a) Side view of a workspace; the camera is mounted in the ceiling directly above the center of the workspace. (b) The view from the overhead camera; note the distortion produced by the wide-angle lens.

4. `blobfind` : group classified pixels into blobs using a connected-component algorithm. Reject those blobs which are too big or too small.
5. `dewarp` : compute the pose of each blob in world coordinates.
6. `ident` : identify objects by searching for pairs of blobs with suitable geometry; use track continuation to label objects.
7. Raise a signal to notify waiting processes that new data is available.
8. Goto 1.

3.5 Configuration

Mezzanine requires quite a lot of configuration information (for describing color classes, the size of the fiducials, image-to-world coordinate transformations, and so on). Configuration information is stored in a file using a simple section-key-value syntax (and can be overridden on the command line as indicated previously). The frame-grabber section of the configuration file might look something like this:

```
# This is a comment
fgrab.norm = ntsc
fgrab.width = 640
fgrab.height = 480
```

A sample configuration file `mezzanine.opt` is supplied with the package.

Note that the calibration program `mezzcal` may alter some of the configuration settings. Such changes are written back into the configuration file when `mezzanine` terminates.

3.5.1 fgrab : frame-grabber interface

The `fgrab` section of the configuration file specifies the properties of the camera and frame-grabber.

`fgrab.norm`

Specifies the type of video signal. Valid values are NTSC or PAL.

`fgrab.width`

`fgrab.height`

Image width and height (typically 640x480 for NTSC and 768x576 for PAL). Make sure the values are valid for your frame-grabber, otherwise strange stuff will happen.

`fgrab.depth`

Color depth in bits. Valid values are 16 and 32, but only the former is recommended.

3.5.2 classify : color classification

The `classify` section of the configuration file defines the color classes that will be extracted from the image. Each color class is defined by a set of three polygons, one for each of the UV, YU and VY color spaces. Only those pixels that project into all three polygons are assigned to the corresponding color class. Most of the entries in this section are generated by the calibration program `mezzcal`; see Chapter 4 for a complete description of the recommended color calibration procedure.

`classify.mask.poly[n]`

Specifies the image mask: only those points inside the mask polygon will be processed. The polygon is specified as a series of points in image coordinates. Use `mezzcal` to edit the mask.

`classify.class[n].name`

`classify.class[n].color`

A descriptive name and color for the n'th color class (to be used in `mezzcal`); only two such classes are required.

`classify.class[n].vupoly[m]`

`classify.class[n].yupoly[m]`

`classify.class[n].vypoly[m]`

Class polygons in the VU, YU and VY projections of the YUV color space. Only those pixels that project into all three polygons will be assigned to this class. Use `mezzcal` to edit these values.

3.5.3 blobfind : color blob segmentation

The `blobfind` section controls the assignment of classified pixels into color blobs. Most of the entries in this section can be edited using the calibration program.

`blobfind.min_area`

`blobfind.max_area`

The minimum and maximum number of pixels in a blob; blobs which are either too small or too large will be discarded.

`blobfind.min_sizex`

`blobfind.max_sizex`

The minimum and maximum width of the blob (in pixels); blobs which are either too narrow or too wide will be discarded.

`blobfind.min_sizey`

`blobfind.max_sizey`

The minimum and maximum height of the blob (in pixels); blobs which are either too short or too tall will be discarded.

3.5.4 `dewarp` : image dewarping; image-to-world transformation

The `dewarp` section controls both the image dewarping (i.e., the removal of barrel distortion on wide angle lenses) and the transformation from image to world coordinates. For this, `mezzanine` needs a set of calibration points whose position in both image and world coordinates is known. Some care must be taken in the selection of calibration points: `mezzanine` uses a 3rd order polynomial which requires at least 7 non-colinear calibration points. The image coordinates of the calibration points can be specified using `mezzcal`; see Chapter 4 for details.

`dewarp.wpos[n]`

World coordinates of the calibration points.

`dewarp.ipos[n]`

Image coordinates of the calibration points. Use `mezzcal` to generate these values.

3.5.5 `ident` : object identification

The `ident` section controls the number of objects that will be tracked, and describes the properties of the fiducials.

`ident.object_count`

Number of objects to track.

`ident.class[0]`

`ident.class[1]`

Class index for each of the two colors making up each fiducial. These will generally be 0 and 1, respectively.

`ident.min_sep`

`ident.min_sep`

Minimum and maximum separation (in world coordinates) between the two colors making up each fiducial.

`ident.max_disp`

Maximum displacement of the fiducial between successive frames (in world coordinates).
Set this to a smaller value if labels are getting swapped when objects are in close proximity.
Set this to a larger value if objects are getting lost when moving at high speed.

`ident.max_missed`

Maximum number of frames that can be missed (i.e., in which the object is not seen) before trying to re-acquire the object elsewhere in the image.

Chapter 4

Using mezzcal

The `mezzcal` program allows users to calibrate `mezzanine` through a simple GUI interface. Note that `mezzanine` will run significantly slower when `mezzcal` is running; it is therefore recommended that users run `mezzcal` only during system calibration.

4.1 Running mezzcal

After starting `mezzanine` as per the instruction in Chapter 3, users can start `mezzcal` by typing:

```
$ mezzcal
```

Three windows should appear, as shown in Figure 4.1; these windows are called the *image*, *color* and *table* windows.

- The image window shows the raw camera image, overlaid with various calibration information. Use the **View** menu to turn these overlays on or off, and use the **File** menu to save calibration changes.
- The color window shows three projections of the YUV color space, with the color classes indicated. Use the **View** menu to turn the overlays for different classes on or off.
- The table window shows various editable properties, such as the minimum and maximum allowed blob sizes.

Note that these window update at a rate of only few hertz; for calibration, it is not necessary to update at the video frame rate.

4.2 Calibration

There are four steps in the calibration, which need only be done once for any given camera set-up.

1. Define the image mask.
2. Define the color classes.
3. Define the blob properties.
4. Calibrate the image-to-world transformation (dewarping).

These steps are detailed in the following sections.

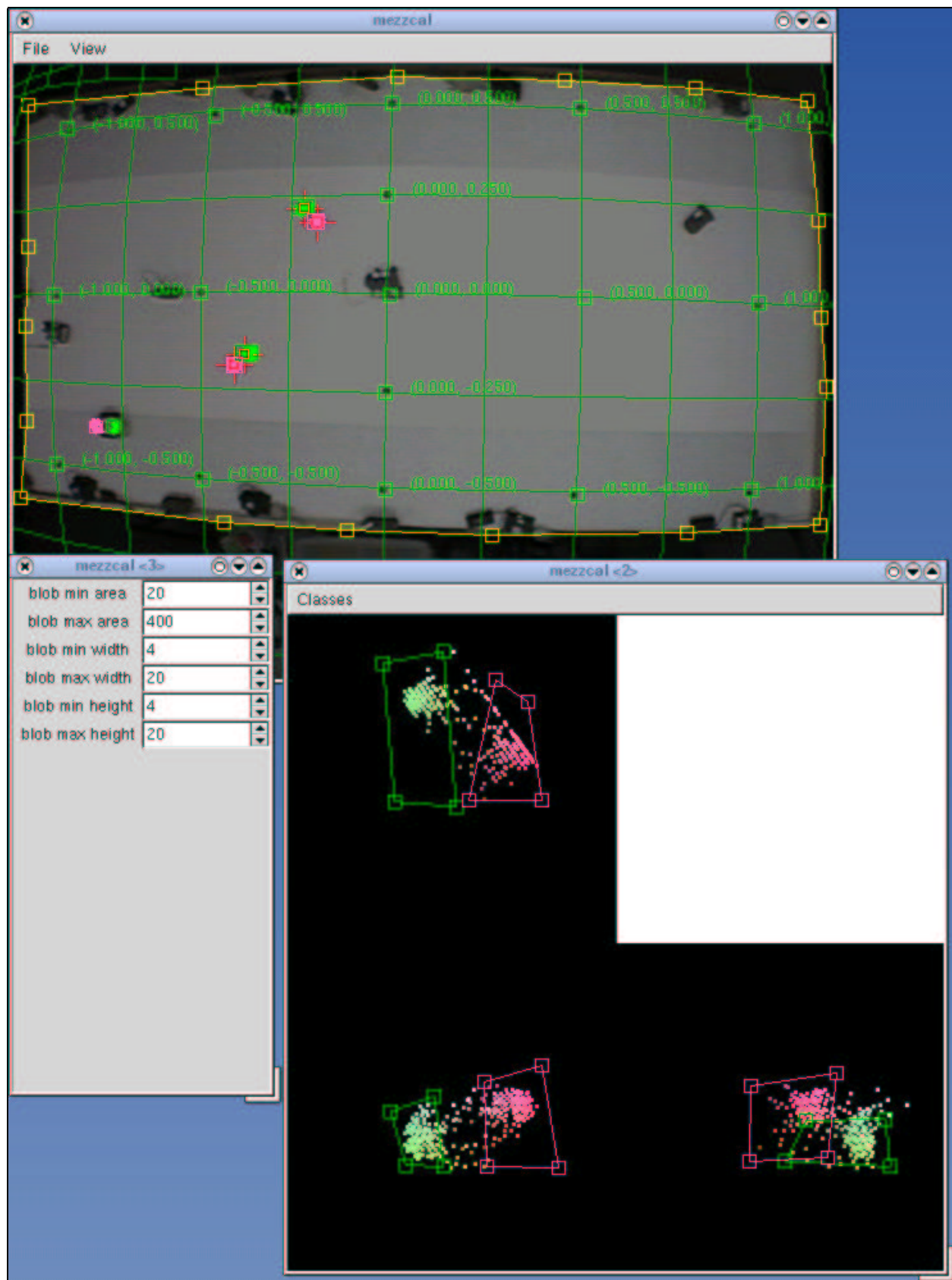


Figure 4.1: The mezzcal program. Top: the image window, showing the raw camera output. Bottom left: the table window. Bottom right: the color window, containing three projections of the YUV color space.

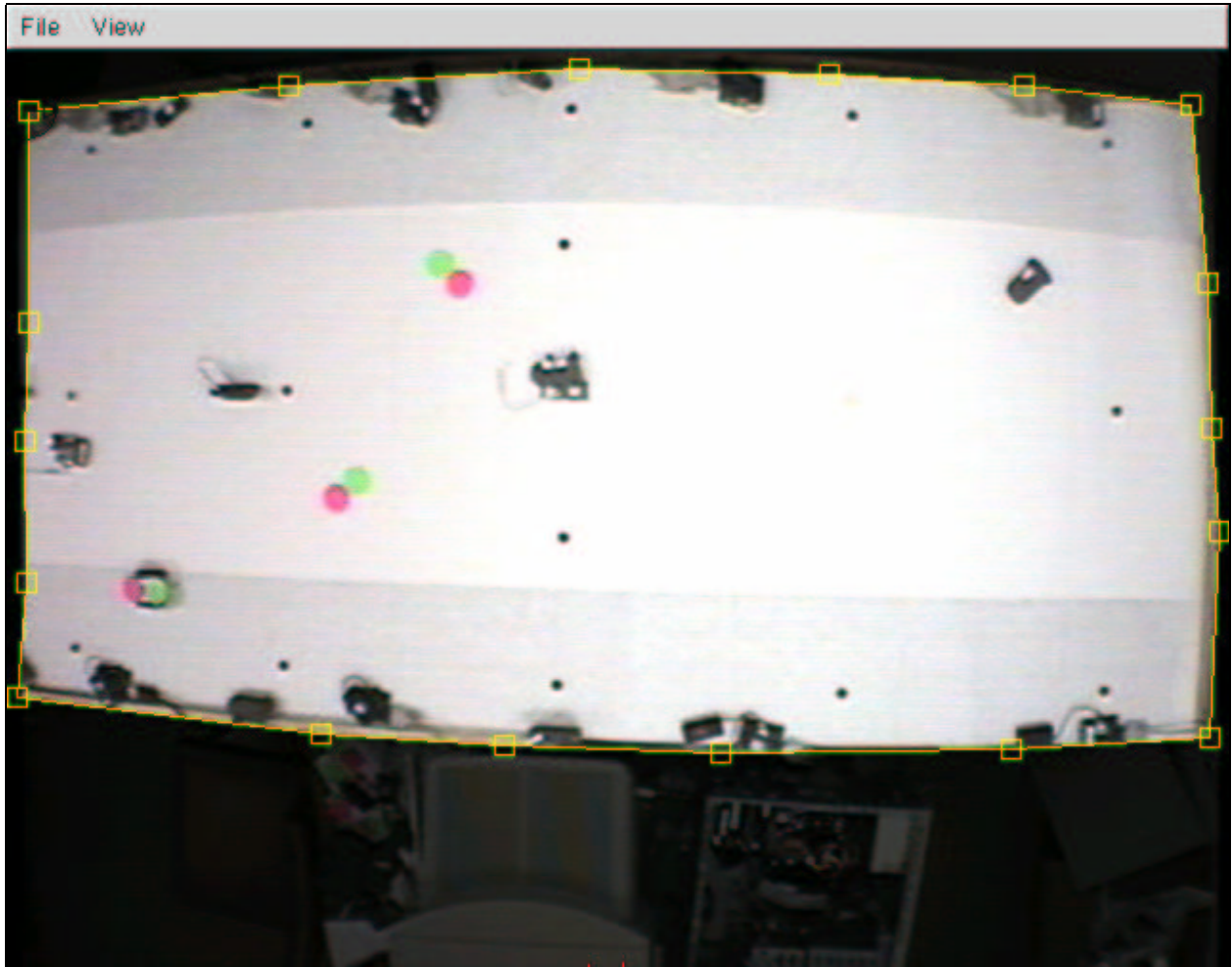


Figure 4.2: The image mask. Points outside the mask are dimmed.

4.2.1 Image mask

The image mask describes the portion of the image that should be processed; i.e., it allows us to ‘mask out’ parts of this image that do not interest us. The image mask is defined by a simple polygon: points inside the polygon are processed, points outside are ignored. Figure 4.2 shows the mask in the image window; the mask can be modified by dragging the control points to new locations.

4.2.2 Color classes

`mezzanine` classifies each pixel in the masked image based upon its color. Pixels can either belong to one of the user specified color classes, or they may belong to none of these classes. Pixels that fall into one of the user defined classes are shown in bright colors in the image window; un-classified pixels are shown dimmed.

Color classes can be conveniently defined using `mezzcal`. First, sample pixels must be identified; this is done by dragging the red sampling cross-hairs in the image window (there are four such cross-hairs) to the desired location in the image. Pixels in the vicinity of these cross-hairs are then

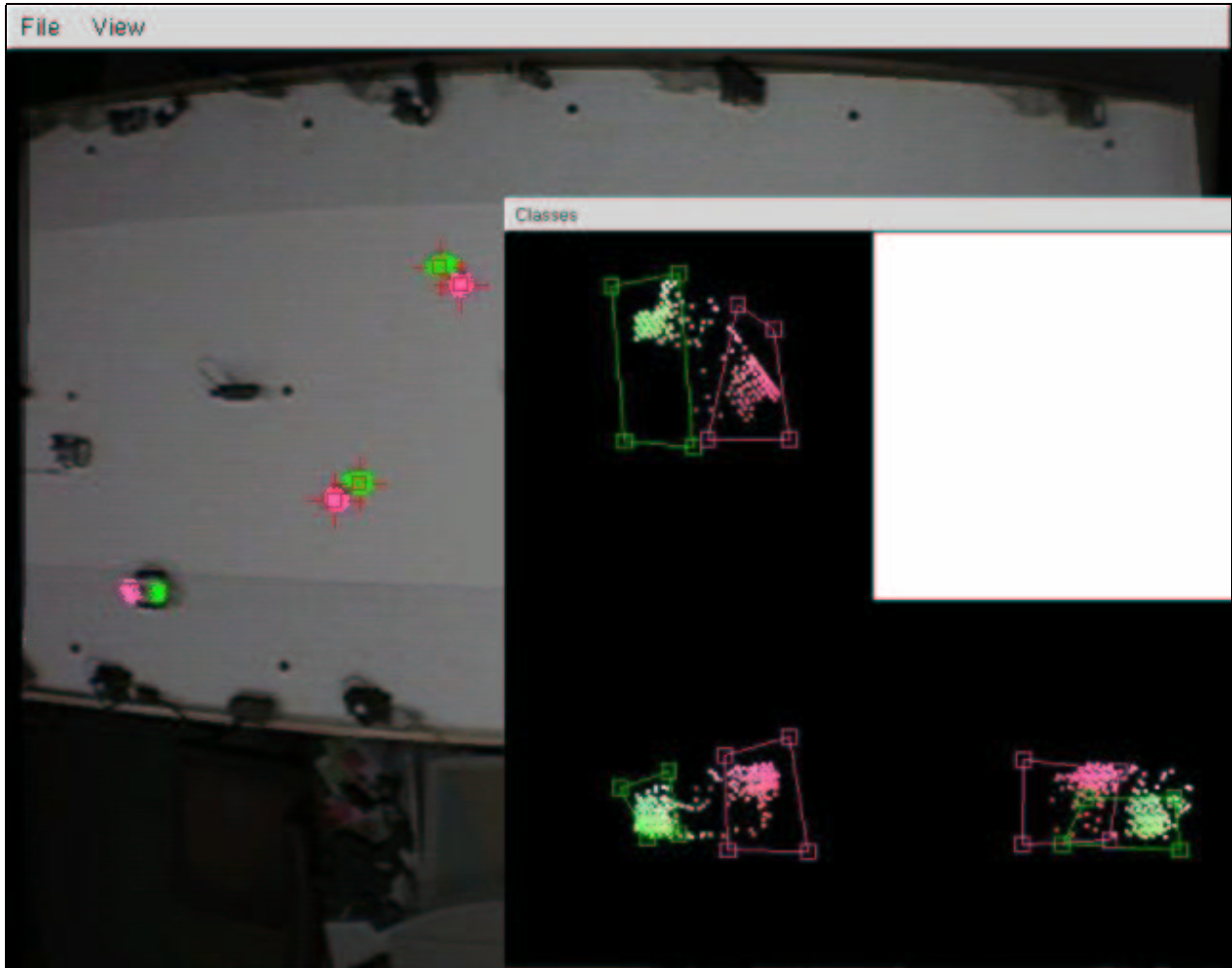


Figure 4.3: Color classification. The cross-hairs in the image window indicate the sample points; these sampled points are projected into the UV, UY and YV color spaces. The polygons define the class boundaries.

automatically projected into the YUV color space shown in the color window. In the color window, each class is defined by three polygons: one for each of the UV, UY and YV projections of the color space; only those points that lie inside all three polygons are assigned to the corresponding class. The class polygons can be modified by dragging their vertices. This process is illustrated in Figure 4.3.

4.2.3 Blob properties

After pixels have been classified, *mezzanine* uses a connected-component analysis to group pixels into blobs; these blobs are then analyzed to determine which ones correspond to fiducials. Thus, for example, blobs which are either too large or too small are rejected. The remaining valid blobs are shown in the image window, denoted by a rectangle drawn in the appropriate class color. The blob definition can be modified by adjusting the properties displayed in the table window.

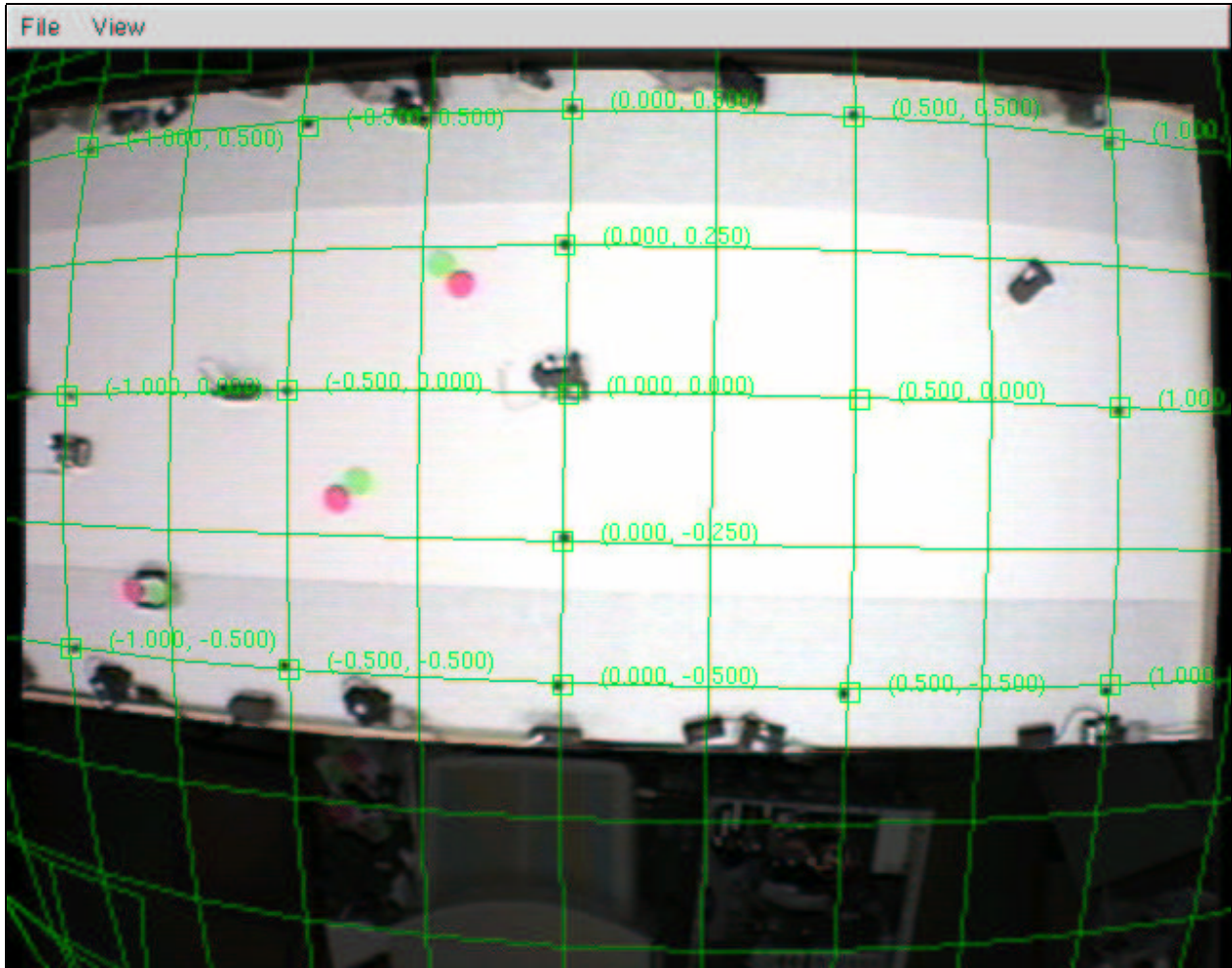


Figure 4.4: The dewarping grid. Each rectangle denotes a calibration point; the accompanying labels show the world coordinates.

4.2.4 Dewarping

The world coordinates of each blob is computed by transforming the blob's image coordinates through a 3rd order polynomial. The coefficients in this polynomial are calculated automatically by `mezzanine`, based on a set of calibration points (i.e. points whose position in both world and image coordinates is known). At least seven such points are required.

The world coordinates of the calibration points must be specified in the configuration file (see Section 3.5.4); the image coordinates can be specified using `mezzcal`. Figure 4.4 shows the dewarping grid: note the calibration points (labelled by their world coordinates). The image coordinates each point can be specified by dragging the point to the appropriate location in the image.

Chapter 5

Using libmezz

The `libmezz` library provides a convenient C interface to `mezzanine`. It defines a small set of functions and structures through which client programs can interact with `mezzanine`.

5.1 Compiling and linking

Client programs should include the header `mezz.h` and link to `libmezz.a`.

5.2 Functions

`libmezz` defines five functions:

`mezz_init()`

Initialise the IPC interface.

`mezz_term()`

Terminate the IPC interface.

`mezz_mmap()`

Return a pointer to the shared memory.

`mezz_wait_event()`

Block until new data is available.

`mezz_raise_event()`

Raises an event to indicate that new data is available. This function is used by `mezzanine` and should not be used in client program.

The Mezzanine IPC library uses Unix signals to notify processes that new data is available. After each new frame is acquired and processed, `mezzanine` sends a `SIGUSER1` to each registered process. These processes are expected to catch this signal and respond by reading the new data from the memory-map. Processes using `libmezz` are automatically registered when the library is initialised, and can use the `mezz_wait_event()` function to wait for the receipt of new data. See `examples/simple` for an example. Processes that do not wish to receive signals can poll the

`count` field in the memory-map, which is incremented after each new frame has been acquired and processed.

See `mezz.h` for up-to-date documentation on each of these functions, and see the `examples` directory for sample programs.

5.3 Structures

The `mezz_mmap()` function returns a pointer to a structure of type `mezz_mmap_t`; this structure contains all of the exported data from `mezzanine`, including the raw and classified images, extracted blobs, and objects. Some configuration information is also stored in this structure: `mezzcal` uses these fields to configure `mezzanine`.

See `mezz.h` or Appendix A for up-to-date documentation on the `mezz_mmap_t` structure.

Appendix A

IPC Specification

The primary specification for the IPC interface is the header file `mezz.h`; this appendix is provided mostly for convenience and is largely auto-generated.

A.1 Signal Handling

The Mezzanine IPC library uses Unix signals to notify processes that new data is available. After each new frame is acquired and processed, `mezzanine` sends a `SIGUSER1` to each registered process. These processes are expected to catch this signal and respond by reading the new data from the memory-map. Processes using `libmezz` are automatically registered when the library is initialised, and can use the `mezz_wait_event()` function to wait for the receipt of new data. See `examples/simple` for an example. Processes that are not using `libmezz` can register themselves by writing into the `pids[..]` array in the memory-map. Insert the process pid into the first null element in this array.

Processes that do not wish to receive signals can monitor the `count` field in the memory-map, which is incremented after each new frame has been acquired and processed.

A.2 Memory-Mapped File Layout

This is just a dump of `mezz.h`; I might do something more intelligent here at some point.

```
/*
 * Mezzanine - an overhead visual object tracker.
 * Copyright (C) Andrew Howard 2002
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
```

```

* along with this program; if not, write to the Free Software
* Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
*
*/
/*****
* Desc: Mezzanine IPC interface
* Author: Andrew Howard
* Date: 28 Mar 2002
* CVS: $Id: mezz.h,v 1.2 2002/05/03 04:55:52 inspectorg Exp $
* Notes:
*
* This library sets up a shared, memory-mapped object for exchanging
* data, and a signal for notifying other processes of updates to
* that object. The mezzanine program will create the ipc interface;
* other processes will connect to an existing interface.
*
* All units are SI unless otherwise noted; i.e. meters, radians,
* seconds. Standard coordinate systems are used: positive x is to
* the right and positive y is upwards.
*
*****/

#ifdef MEZZ_H
#define MEZZ_H

#include <sys/types.h> // For pid_t
#include <stdint.h> // For uint8_t, uint32_t, etc

// Some constants
#define MEZZ_MAX_AREA 768*576 // Maximum number of pixels in the image.
#define MEZZ_MAX_MASK 18 // Maximum number of points in the mask.
#define MEZZ_MAX_DEWARP 32 // Maximum number of dewarping calibration points.
#define MEZZ_MAX_CLASSES 4 // Maximum number of color classes we recognize.
#define MEZZ_MAX_BLOBS 100 // Maximum number of blobs we will extract.
#define MEZZ_MAX_OBJECTS 100 // Maximum number of extracted objects.

// Image mask info; used to identify the part of the image
// that corresponds to the area of interest.
typedef struct
{
    int poly[MEZZ_MAX_MASK][2]; // List of points defining a polygon.
} mezz_maskdef_t;

// Calibration info for dewarping
typedef struct
{
    int points; // Number of dewarp points
    int ipos[MEZZ_MAX_DEWARP][2]; // The image coords of the calibration points.
    double wpos[MEZZ_MAX_DEWARP][2]; // The world coords of the calibration points.
    double iwtrans[2][8]; // Parameters for the image-to-world transform.
    double witrans[2][8]; // Parameters for the world-to-image transform.
}

```

```

} mezz_dewarpdef_t;

// Structure describing a color class.
// Used by the vision system for color classification.
typedef struct
{
    int color;          // Descriptive color (RGB32); used by gui.
    char name[64];     // Descriptive name for this color; used by gui.
    int vupoly[4][2];  // Polygon in VU components
    int yupoly[4][2];  // Polygon in YU components
    int vypoly[4][2];  // Polygon in VY components
} mezz_classdef_t;

// List of color class definitions
typedef struct
{
    int count;
    mezz_classdef_t classdefs[MEZZ_MAX_CLASSES];
} mezz_classdeflist_t;

// Options for the color blob extraction.
typedef struct
{
    int min_area, max_area; // Allowed area (number of classified pixels).
    int min_sizex, max_sizex; // Allowed width.
    int min_sizey, max_sizey; // Allowed height
} mezz_blobdef_t;

// Information describing an extracted color blob.
typedef struct
{
    int class;          // What color class the blob belongs to.
    int min_x, max_x;  // Bounding box for blob (image coords)
    int min_y, max_y;  // Bounding box for blob (image coords)
    int area;          // Area of the blob (pixels).
    double ox, oy;     // Blob centroid (image coords).
    double wox, woy;   // Blob centroid (world coords).
    int object;        // Index of the object this blob is part of.
} mezz_blob_t;

// A list of blobs
typedef struct
{
    int count;
    mezz_blob_t blobs[MEZZ_MAX_BLOBS];
} mezz_bloblist_t;

```

```

// Structure describing an object in the image.
// We assume all objects are made up of two blobs.
typedef struct
{
    int class[2];          // Color class for the two blobs
    double max_disp;      // Maximum inter-frame displacement
    double max_sep;       // Maximum blob separation
    int max_missed;       // Max frames we can miss before we look for a new match.
    int missed;           // Number of missed frames (object not seen)
    double px, py, pa;    // Object pose (world cs).
} mezz_object_t;

// Structure describing a list of objects.
typedef struct
{
    int count;
    mezz_object_t objects[MEZZ_MAX_OBJECTS];
} mezz_objectlist_t;

// Combined memory map data
typedef struct
{
    pid_t pids[32];       // List of pid's that should get events
    double time;          // The system time
    int count;            // Frame counter
    int calibrate;        // Non-zero if a calibration program is running
    int save;             // Non-zero if changes should be saved
    int width, height;    // Image dimensions (pixels)
    int area;             // Number of pixels in the image.
    int depth;           // Image depth (bits): 16, 24 or 32.
    mezz_maskdef_t maskdef; // The image mask (identifies the field in the image).
    mezz_dewarpdef_t dewarpdef; // Dewarping calibration info.
    mezz_classdeflist_t classdeflist; // Color class definitions.
    mezz_blobdef_t blobdef; // Blob definitions.
    uint32_t image[MEZZ_MAX_AREA]; // The raw frame data.
    uint8_t mask[MEZZ_MAX_AREA]; // The image mask.
    uint8_t class[MEZZ_MAX_AREA]; // The classified frame data.
    mezz_bloblist_t bloblist; // The list of extracted blobs.
    mezz_objectlist_t objectlist; // The list of identified objects.
} mezz_mmap_t;

// Initialise the interprocess comms.
// The server sets create to 1 to create the shared resources.
int mezz_init(int create);

// Finalize the ipc
// The server sets destroy to 1 to destroy the shared resources.
void mezz_term(int destroy);

// Get a pointer to the memory map

```



```
mezz_mmap_t *mezz_mmap();

// Raise an event (to be recieved by other processes)
void mezz_raise_event();

// Wait for an event (that was sent by another process)
int mezz_wait_event();

#endif
```