

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

# Mezzanine User Manual

Version 0.00

Andrew Howard

**DRAFT ONLY! THIS DOCUMENT IS INCOMPLETE.**

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 5, 2002

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Description . . . . .	1
1.2	History and Legal Stuff . . . . .	2
1.3	System Requirements . . . . .	2
1.4	Getting Mezzanine . . . . .	2
1.5	Building and Installing . . . . .	3
1.6	Bugs . . . . .	3
1.7	Acknowledgements . . . . .	3
<b>2</b>	<b>QuickStart Guide</b>	<b>4</b>
<b>3</b>	<b>Using mezzaine</b>	<b>5</b>
3.1	<code>fgrab</code> : frame-grabber interface . . . . .	5
3.2	<code>classify</code> : color classification . . . . .	6
3.3	<code>blobfind</code> : color blob segmentation . . . . .	6
3.4	<code>dewarp</code> : image dewarping; image-to-world transformation . . . . .	7
3.5	<code>ident</code> : object identification . . . . .	7
<b>4</b>	<b>Using mezzcal</b>	<b>9</b>
<b>A</b>	<b>IPC Specification</b>	<b>10</b>
A.1	Signal Handling . . . . .	10
A.2	Memory-Mapped File Layout . . . . .	10

# 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 almost any color camera (even the really cheap ones), and can correct for the barrel distortion produced by wide-angle-lenses.

Mezzanine is also 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 the tracking program.

#### **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 simple C interface for reading data from and writing commands to *mezzanine*. This library is provided partly as a reference and partly as a convenience. Users can write native interfaces to *mezzanine* in any language that supports access to shared memory-mapped files. See Appendix A for a complete reference, and check out the **examples** directory for sample programs.

## **1.2 History and Legal Stuff**

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>. Mezzanine is free software, released under the GNU Public License. There is absolutely NO WARRANTY.

## **1.3 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:

- 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/>

## **1.4 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.

## 1.5 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.

## 1.6 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/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.

## 1.7 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](mailto: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:

```
[someuser]$ mezzanine -fgrab.norm NTSC
```

or

```
[someuser]$ 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:

```
[someuser]$ 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.

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 mezzaine

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 ???. Any pair of colors can be used for the fiducials, 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. With these fiducials, Mezzanine can determine the position and orientation of objects, but not their identity. We therefore rely on track-continuation to generate consistent object labels. Beware that labels may swap if two fiducials come into very close proximity.

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. Thus an entry in the configuration file that specifies NTSC format for image capture:

```
fgrab.norm = NTSC
```

can be overridden on the command line to specify PAL format:

```
[someuser]$ mezzanine -fgrab.norm PAL
```

In this example, `mezzanine` will use the default configuration file; users may also specify their own configuration file on the command line:

```
[someuser]$ mezzanine -fgrab.norm PAL myfile.opt
```

which will load the file `myfile.opt`, then override the capture format setting. Note that the configuration program (`mezzcal`) may write changes into this configuration file.

### 3.1 fgrab : frame-grabber interface

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

<code>fgrab.norm</code>	Specifies the type of video signal. Valid values are NTSC or PAL
<code>fgrab.width,</code> <code>fgrab.height</code>	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.
<code>fgrab.depth</code>	Color depth in bits. Valid values are 16 and 32, but only the former is recommended.

### 3.2 `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.

<code>classify.mask.poly[n]</code>	Specifies the image “mask” polygon: only those points inside the polygon will be processed. The polygon is specified as a series of points in image coordinates; use <code>mezzcal</code> to edit the mask.
<code>classify.class[n].name</code> <code>classify.class[n].color</code>	A descriptive name and color for the n'th color class (to be used in <code>mezzcal</code> ); only two such classes are required.
<code>classify.class[n].vupoly[m]</code> <code>classify.class[n].yupoly[m]</code> <code>classify.class[n].vypoly[m]</code>	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 <code>mezzcal</code> to edit these values.

### 3.3 `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.

<code>blobfind.min_area</code>	The minimum and maximum number of pixels in a blob; blobs which are either too small or too large will be discarded.
<code>blobfind.max_area</code>	
<code>blobfind.min_sizex</code>	The minimum and maximum width of the blob (in pixels); blobs which are either too narrow or too wide will be discarded.
<code>blobfind.max_sizex</code>	
<code>blobfind.min_sizey</code>	The minimum and maximum height of the blob (in pixels); blobs which are either too short or too tall will be discarded.
<code>blobfind.max_sizey</code>	

### 3.4 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 recommended procedure for the dewarp calibration is described in Chapter 4.

<code>dewarp.wpos[n]</code>	World coordinates of the calibration points.
<code>dewarp.ipos[n]</code>	Image coordinates of the calibration points. Use <code>mezzcal</code> to generate these values.

### 3.5 ident : object identification

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

<code>ident.object_count</code>	Number of objects to track.
<code>ident.class[0]</code>	Class index for each of the two colors making up each fiducial. Will generally be 0 and 1, respectively.
<code>ident.class[1]</code>	
<code>ident.min_sep</code>	Minimum and maximum separation (in world coordinates) between the two colors making up each fiducial.
<code>ident.min_sep</code>	
<code>ident.max_disp</code>	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.
<code>ident.max_missed</code>	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

[to do]

# 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

[autogenerate here]