

LTU, EISLAB

Pick and Place

D7004EProject course 2008

Fredrik Kers, Anders Lindmark, Henrik Mäkitaavola, Joakim Söderberg
5/23/2008

Introduction

This project report is about the Pick and Place software that was developed in the D7004E project course 2008. The report is separated into three parts, one discussing the Machine API which is used for controlling the machine, one deals with the Camera API that handles camera control and image manipulation and the last part is about the GUI.

Contents

Machine API	1
Introduction	1
Result	1
Design and architecture	1
Usage manual.....	4
Commands	4
Adding a command	5
Events.....	6
Evaluation	7
Building the API.....	7
Known issues.....	7
Discussion.....	7
Future work.....	7
Camera API	8
Introduction	8
Design.....	9
Camera manager.....	9
Driver.....	9
Camera	10
Filter	10
Image	10
Barrel distortion	10
Code example	12
Graphical User Interface (GUI)	13
GUI Framework	13
Qt	13
Slots and signals	13
GUI Code Design	13
GuiCommands.....	13
CameraWidget	13

PicknPlaceGui.....	15
Slots.....	15
Running commands	15
Resources	16
GUI Layout.....	17
Overview	17
1 – The toolbar	18
2 – Brightness slider	18
3 – Tool specific controls.....	18
4 – The camera widget.....	18
5 – The enqueue button	18
6 – The movements controls	19
7 – The list of commands	19
Future work.....	19
Easy	19
Medium.....	20
Hard.....	20
Conclusion	21
Results.....	21

Machine API

Introduction

The Machine API is used to communicate with the machine via a serial cable.

The goal with the API is to provide a simple way for program developers to interact with the Pick and Place machine.

Not all functions that are available in the machine have been implemented in the API, only the functions that were requested by the customer.

The Machine API is designed to be used both by the GUI team but also to enable other people to build programs that use the machine.

Result

The Machine API can do all of the things that is required of it. All goals were met. The outcome of the design and implementation is as desired, except for maybe a minor hassle when adding a command to the API since this requires changes in multiple files.

Because of the way it is implemented the API only works in a Microsoft Windows environment. The reason for this is the use of windows threads and serial port libraries. This should be fairly easy to port to other platforms.

Design and architecture

When designing the API there were a few requirements to satisfy

- should not block when executing a command
- easy to extend and add new commands
- able to handle communication failures and unexpected errors
- fast and memory efficient
- keep as simple as possible, avoiding complex solutions

From these requirements a UML diagram was crafted.

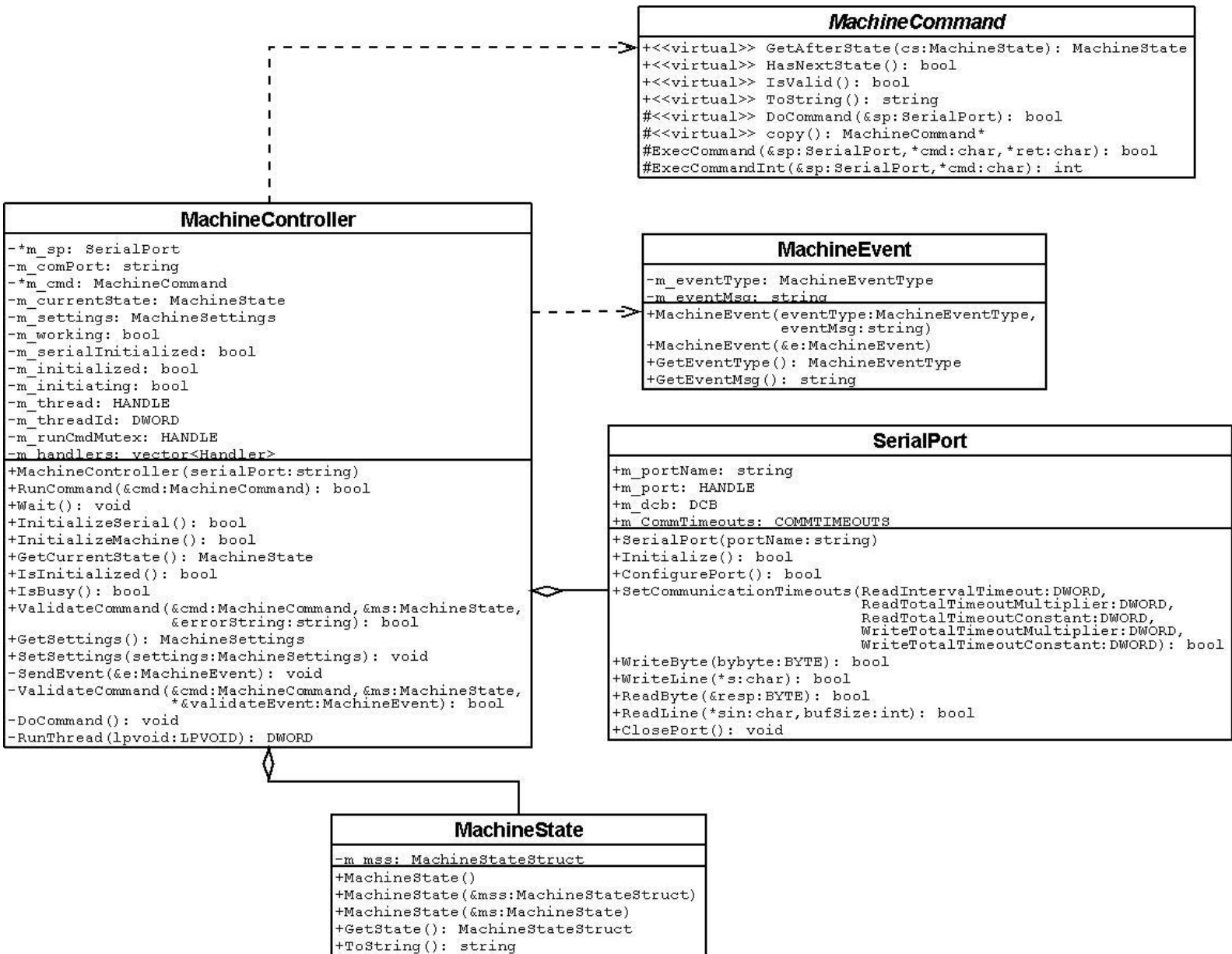


Figure 1 UML diagram.

A main class, *MachineController* handles communication with the machine and is also the interface that applications use.

The *MachineCommand* interface was created to make creating new commands easy. This also has the benefit of making the design structured; with every new command being in its own class, implementing the *MachineCommand* interface.

To simplify the design, the API only handles one command at a time, although commands can be bundled with a *MachineWrapperCommand* object. Any command queuing is up to the caller to implement.

Execution of a command is done in a separate thread. Communication with the caller is done through an event system. An event is sent using a *MachineEvent* object, which contains the event type and an event message. These events are sent whenever a command has finished executing or an internal error has occurred. The caller registers a callback function in the *MachineController* that handles these events. Multiple callback functions can be used.

The *MachineController* contains the current state of the machine. This information is saved in the *MachineState* class. Information stored in this class is used to validate commands, store offsets for picking and dispensing, etc.

For more information about the commands and classes, see the doxygen documentation provided with the source code.

Usage manual

Commands

Creating a *MachineController* instance and initializing it:

```
#include "MachineController.h"

string comPort = "com1"; // Which com-port the machine is connected to
MachineController mc(comPort); // Create a new machine controller
instance

/*
 * Initialize the machine. This initializes the serial port and
 * issues a MachineInitCommand.
 * When this is done the machine is ready to accept commands.
 * This function returns true when the serial port is initialized,
 * but to know when the machine is ready a check for a MachineEvent
 * of type EVENT_MACHINE_INITIALIZED needs to be done.
 */
if ( !mc.InitializeMachine() )
{
    // Machine failed to initialize
    return;
}
// Wait for the thread doing the initialization to complete.
// (This is never required, but no command can be issued until
// the current command is done)
mc.Wait();
```

Creating and running a command. Running any type of command is the same procedure.

```
// Move to x,y,z = ( 10000, 20000, 1000)  $\mu$ m from origo
MachineMoveAllCommand cmd(10000, 20000, 1000);

mc.RunCommand(cmd); // Validate and run the command.
mc.Wait(); // Wait for the machine to finish. (Optional)
```


Creating a polygon dispense command.

```
MachinePolygon mp;          // Create a polygon to define the path
// Add points to the polygon:
mp.AddPoint(MachinePolygonPoint(10000, 10000)); // Index 0
mp.AddPoint(MachinePolygonPoint(10000, 50000));
mp.AddPoint(MachinePolygonPoint(20000, 50000));
mp.AddPoint(MachinePolygonPoint(20000, 10000));
mp.AddPoint(MachinePolygonPoint(30000, 10000));
mp.AddPoint(MachinePolygonPoint(30000, 50000));
mp.AddPoint(MachinePolygonPoint(40000, 50000));
mp.AddPoint(MachinePolygonPoint(40000, 10000)); // Index 7

mp.DelPoint(0); // Delete the point on the polygon with index 0
// (The point that used to be index 1 has now been moved to index 0)

mp.DelPoint(1); // Delete the point on the polygon with index 1

// Create a polygon dispense command from the polygon
MachinePolygonDispenseCommand cmd(mp);

mc.RunCommand(cmd);
mc.Wait();
```

Wrapping several commands into one

```
// Create a few commands
MachineMoveAbsoluteCommand moveX(AXIS_X, 10000);
MachineMoveAbsoluteCommand moveY(AXIS_Y, 20000);
MachineMoveAbsoluteCommand moveZ(AXIS_Z, 1000);

// Create the wrapper and add the commands to it
// A wrapper takes several commands and bundles them into a command
// that executes them sequentially (and then optionally returns the
// machine to the coordinates
// it was at before the wrapper started.)
MachineWrapperCommand wrapper;
wrapper.Add(moveX);
wrapper.Add(moveY);
wrapper.Add(moveZ);

mc.RunCommand(wrapper);
mc.Wait();
```

For more information about each command, see the doxygen documentation.

Adding a command

Create a new class that extends the MachineCommand class.

Example, adding “TestCommand”:

TestCommand.h

```
#ifndef __TESTCOMMAND_H__
#define __TESTCOMMAND_H__
#include "machinecommand.h"

class TestCommand : public MachineCommand
{
    // Make the private methods of other commands
    // available to this command.
    MACHINE_COMMAND_FRIENDS;
public:
    TestCommand(Data inData);
    ~TestCommand(void);

    string ToString();
    MachineState GetAfterState(MachineState &oldms);

private:
    TestCommand *Copy();
    bool DoCommand(SerialPort &sp);

    Data m_inData;
};

#endif // __TESTCOMMAND_H__
```

Then add the command to the list in *MachineCommands.h* and to MACHINE_COMMAND_FRIENDS in *MachineCommand.h*.

If the command has several states, the *HasNextState()* methods needs to be overridden.

For more information, see the implementation of the commands.

Events

Events are passed by calling a callback function. A callback functions should look something like this:

```
void handleEvent(MachineEvent *e)
{
    // Code for handling the event

    delete e;    // The event must be deleted when done
}
```

The callback function should then be registered to the API, like so:

```
mc.AddEventHandler(&handleEvent);
```

Now an event will be passed to the event handler when a command is finished, or has failed.

Evaluation

We found a few memory leaks and situations where buffers could overflow. These were fixed.

Building the API

The API has been successfully built using Microsoft Visual Studio 2008 and no other build systems have been tested.

Using Visual Studio 2008, open the solution file (MachineAPI.sln) located in the MachineAPI folder in the source code.

When compiling the entire project, this solution file should not be used. See the section for building the project instead.

When building, a library file (MachineAPI.lib) is produced, which can be linked from other projects.

Known issues

The API is not in a specific namespace. Problems can occur with multiple classes of the same name.

Discussion

The API works as desired.

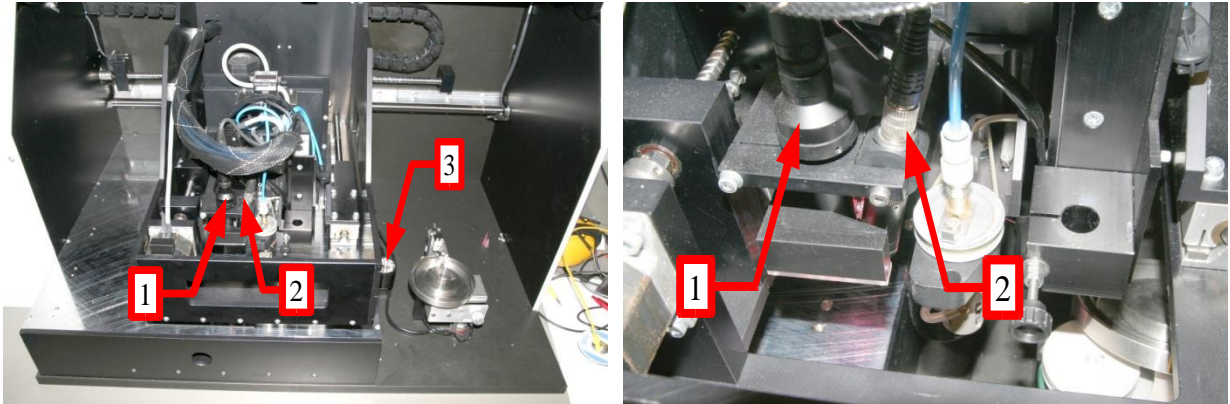
Future work

- Only use one thread for the commands, instead of spawning and new one every time and tearing it down after execution is completed.
- Implementing new commands if needed.
- Event on progress of wrapper command.

Camera API

Introduction

The pick and place machine has three cameras. Two of them are placed on the moving head of the machine facing downwards. The “detail camera” (1) and the “overview camera” (2). The “lookup camera” (3) is placed beside the working area, facing upwards.

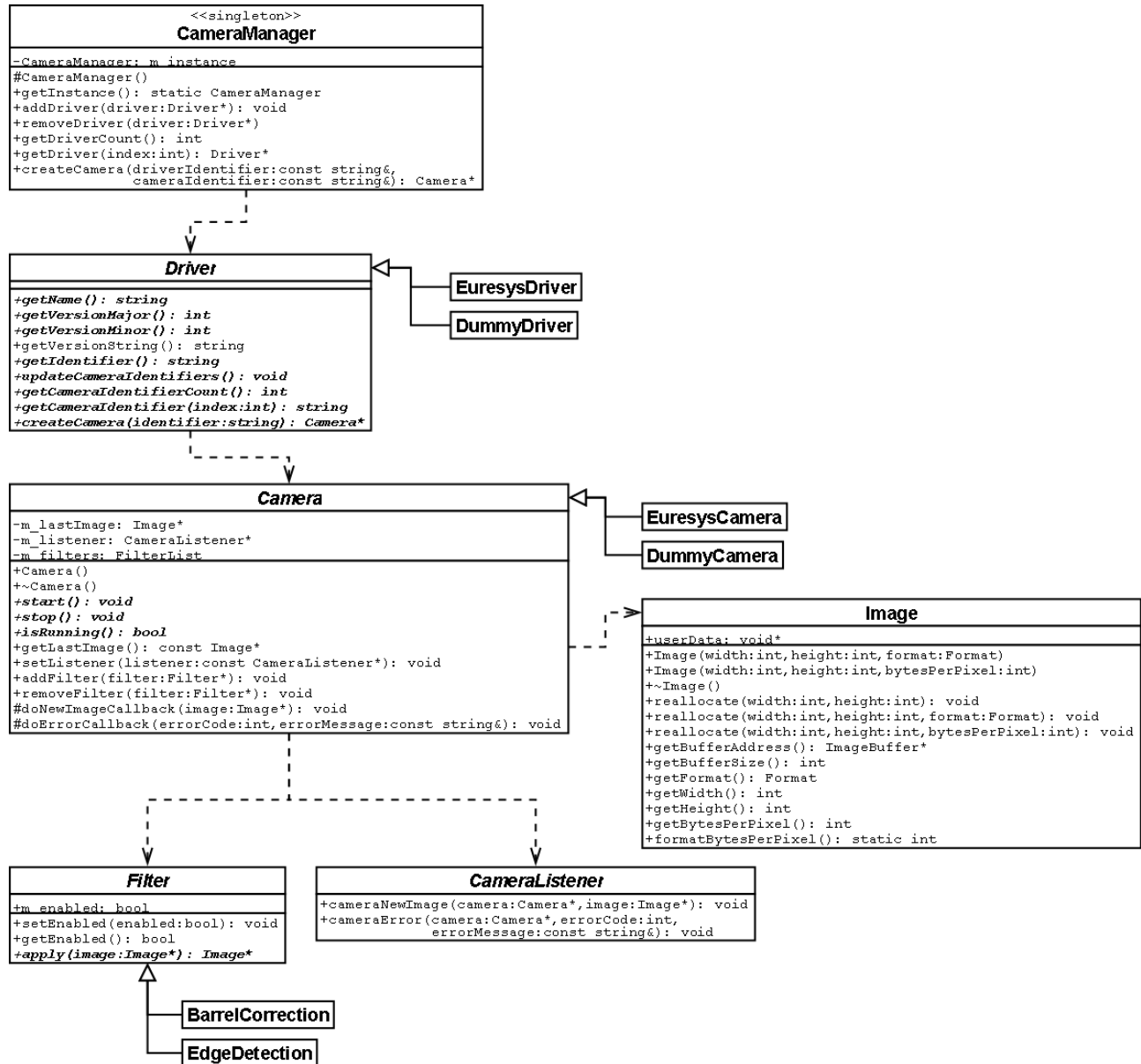


These cameras are connected to a frame grabber card (Euresys Picolo Pro 2) in the computer. To be able to get the image from the cameras into the program without having to write driver specific low level code, we developed the camera API.

The goals when designing the camera API was:

- It should be easy to add support for new frame grabber cards.
- It should be possible to apply filters for doing image enhancement and modification.
- The allocation and deallocation of image buffers should be managed by the API.
- The API should be easy to understand and use.
- All classes in the API should be well documented using doxygen.

Design



Camera manager

The **CameraManager** class is used to gather all drivers in one place and simplifies the creation of Camera instances. The **CameraManager** implements the singleton design pattern to make all drivers available anywhere in the application.

Driver

A driver is used to list and create cameras for a specific frame grabber card. At the moment there are two implementations of the Driver class. A dummy driver that can be used for testing on a computer without any frame grabber card, and a Euresys driver that adds support for the Euresys Pico Pro 2

(with driver version 3.x). To add support for another frame grabber card a version of the driver class for that card can be implemented.

Camera

The camera, like the driver, only supports a specific frame grabber card. The camera is used to start and stop the stream of images.

Filter

Filters are used to modify images coming from the camera before they are passed on to the camera listener. Many image filters (like brightness, saturation, inversion, etc.) can be done directly on the incoming image (in-place). The apply method then returns a pointer to the input image. Other filters (like edge detection, blur, resizing, etc.) must have a separate result image that is returned.

Image

The image class is used to simplify the memory allocation and handling. It also holds some properties for the allocated image buffer like width, height and pixel format. The pixel format describes how the information for each pixel is encoded.

Barrel distortion

Because the overview camera has a wide angle lense and is tilted to capture the same area as the detailed camera, the image from it is distorted. The wide angle lense will produce a barrel distortion while the tilting will give a perspective distortion. To be able to pick coordinates from the image and draw graphical feedback of the commands the machine should execute, the image needed to be corrected.

A barrel correction filter was developed that corrected both the barrel distortion and the perspective distortion. The filter calculates the mapping from the correct coordinates (x, y) to the distorted coordinates (\hat{x}, \hat{y}) and then copies the pixel color from that pixel to the corrected image.

$$CorrectedImage(x, y) = DistortedImage(\hat{x}, \hat{y})$$

The distorted coordinates is calculated using the following equations:

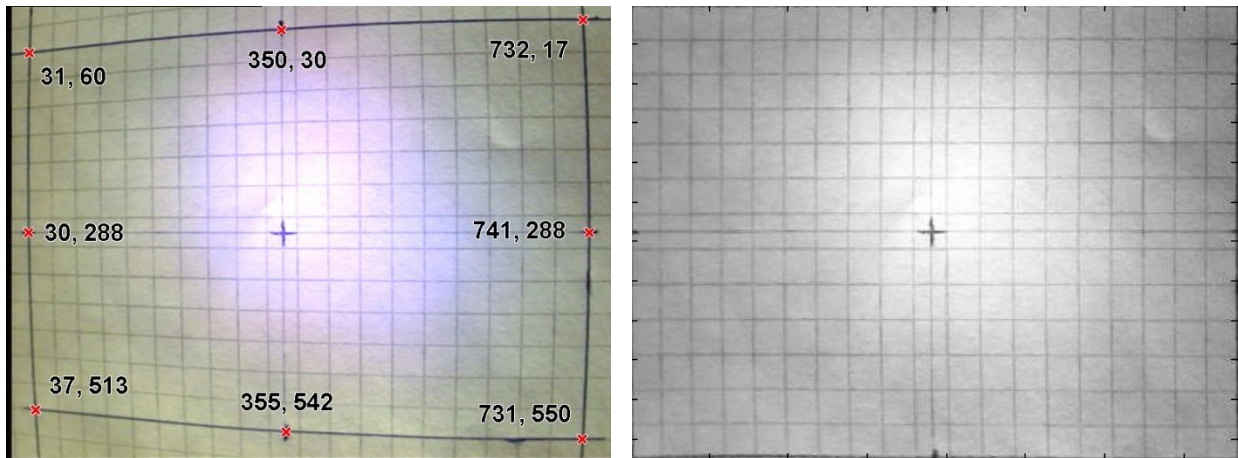
$$\begin{aligned}\hat{x} &= a_1x^2y + a_2xy^2 + a_3x^2 + a_4y^2 + a_5xy + a_6x + a_7y + a_8 \\ \hat{y} &= b_1x^2y + b_2xy^2 + b_3x^2 + b_4y^2 + b_5xy + b_6x + b_7y + b_8\end{aligned}$$

The constants $a_{1..8}$ and $b_{1..8}$ are calculated by solving the linear equations $\hat{x} = M * a$, $\hat{y} = M * b$ where

$$M = \begin{bmatrix} (x_1^2y_1) & (x_1y_1^2) & (x_1^2) & (y_1^2) & (x_1y_1) & (x_1) & (y_1) & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ (x_8^2y_8) & (x_8y_8^2) & (x_8^2) & (y_8^2) & (x_8y_8) & (x_8) & (y_8) & 1 \end{bmatrix}$$

By picking coordinates on the distorted image that should represent a proper rectangle in an undistorted image, one for each corner, and the midpoints between them, when calculating the mapping, the distorted rectangle will correspond to the border pixels of the corrected image.

Here is an example of an image from the overview camera that has been corrected using the barrel correction filter.



Code example

The following example shows how to create a camera and start the acquiring.

```
using namespace camera;

// Get the camera manager instance.
CameraManager *cameraManager = CameraManager::getInstance();

// Create a dummy driver and add it to the camera manager.
DummyDriver *dummyDriver = new DummyDriver();
dummyDriver->setImageSize(768, 576);
cameraManager->addDriver(dummyDriver);

// Print all drivers and the available cameras for each driver.
for(int i = 0; i < cameraManager->getDriverCount(); i++)
{
    printf("Driver #%d: %s", i, cameraManager->getDriver(i)->getVersionString());
    for(int j = 0; j < cameraManager->getDriver(i)->getCameraIdentifierCount(); j++)
    {
        printf("Camera identifier #%d: %s",
            j, cameraManager->getDriver(i)->getCameraIdentifier(j));
    }
}

// Create the first camera of the first driver.
std::string driverIdentifier = cameraManager->getDriver(0)->getIdentifier();
std::string cameraIdentifier = cameraManager->getDriver(0)->getCameraIdentifier(0);
Camera *camera = cameraManager->createCamera(driverIdentifier, cameraIdentifier);

// Add the barrel correction filter to the camera.
int distortedRectangle[8][2] = {{31, 60}, {350, 30}, {732, 17}, {30, 288}, {741, 288},
{37, 513}, {355, 542}, {731, 550}};
BarrelCorrection *barrelCorrection = new BarrelCorrection(distortedRectangle);
camera->AddFilter(barrelCorrection);

// Set the listener and start the camera.
camera->setListener(myCameraListener);
camera->start();

// Use the camera images sent by the camera...

// Make sure to delete the camera and the filter when done.
delete camera;
delete barrelCorrection;
```


Graphical User Interface (GUI)

GUI Framework

Our initial idea was to use C# and the Windows Presentation Foundation for the GUI, but when it turned out the Capture card API for .NET didn't support the capture card used in the computer connected to the Pick and Place machine. We then decided to use Qt instead. Most likely using WPF would've sped up the GUI development considerably. Another scenario would've been to write our own wrapper for the Euros camera api, but we decided for using Qt/C++ in the end.

Qt

Qt is a multi-platform application framework, including classes for threading, string handling, database access, XML and of course User Interface building. The framework is object oriented and written in C++, and used by big projects such as KDE, and recently Nokia bought the company that develops it. The commercial version also includes Visual Studio integration for the UI editor, but we had to settle with the standalone GUI Designer.

Slots and signals

Qt uses something called signals and slots for communicating between objects as an alternative to using callback functions. Whenever an event occurs a signal is emitted. A slot is a function that is called in response to such a signal. A signal can be connected to several slots. To read more about how this works (which is essential to using Qt) check this url:

<http://doc.trolltech.com/signalsandslots.html>

GUI Code Design

The GUI consists of three projects, *PicknPlaceGui*, *GuiCommands* and *CameraWidget* (with the *Settings* project intended to be used with it also) The main GUI is located in the *PicknPlaceGui* project, which uses both *GuiCommands* and *CameraWidget*.

GuiCommands

This project is compiled as a static library and contains wrappers for the three main commands that the GUI allows you to create and run, *DispencePolygonCommand*, *DispenceDotCommand* and *PickAndPlaceCommand* who all inherits *GuiMachineCommand*. The commands are used by both the *CameraWidget* and *PicknPlaceGui*.

CameraWidget

The camera widget is the focal point of the GUI, where the user does the most interaction with the machine. This project is dependent on the *CameraAPI* for obvious reasons, and also uses the *GuiCommands* project to draw the list of current commands created by the user.

A pointer to the list of commands to draw is passed to the **CameraWidget** by the caller. The caller is also expected to pass on any changes of the machine's current position, since this is used to calculate the machine coordinate of points on the widget that were clicked by the user. To do this there's a slot named `setMachineCoordinates` that takes the x/y/z machine coordinates.

The **CameraWidget** can be in different states depending on what action the user wants to perform. The different states are:

- **Move** – If the user clicks the **CameraWidget** it will calculate the machine coordinate for the click and then emit a `newMachineCoordinates(int machine, intMachineY)` signal that contains the new x/y machine coordinates it wants to move to. It is then up to the caller to realize this.
- **Pick/Place** – The user sets three points on the chip that should be picked up, so that it outlines a corner, from this the center point and angle of the chip will be calculated. When both the three pick and place points have been positioned, a `commandReady(InteractionMode mode, QPoint *pickPoints, QPoint *placePoints)` signal will be emitted which the caller can act upon. These are pointers to two 3-element arrays. (This should probably be changed to two QLists instead).
- **DispenseDot** – The user clicks on the widget to chose where to dispense the dot and a `commandReady(CameraWidget::InteractionMode mode, QPoint dot)` signal is emitted.
- **DispensePolygon** – The user creates a polygon on the **CameraWidget** by clicking on it to add new points. When at least one point exists in the polygon a `commandReady(InteractionMode mode, DispencePolygonCommand *polygon)` signal is emitted.
- **Calibration** – Currently does nothing, but is meant to be used for calibrating the camera to fix the barrel distortion that the overview camera has. These values are hard coded at the moment. This is very simple to add, juts let the user click 8 points on the distorted image that should correspond to a rectangle on the fixed image (see the CameraAPI documentation for an example on how this is done).

Whenever the user does something to invalidate the current command being created a `commandInvalid()` signal will be raised.

The **CameraWidget** also contains methods for retrieving the current commands being created by the user:

```
getDispensePolygon()  
getDotDispensePoint()  
getPickPoints()  
getPlacePoints()
```

Since the *CameraWidget* is to be used in the GUI we need to compile it as a plugin for the Qt Designer, see more about this in the compiling section.

PicknPlaceGui

This is the main GUI project that uses all the libraries in the solution (except *Settings* currently). The GUI in this project was created using the Qt Designer application which then outputs a .ui-file in XML. This file is then compiled with the Qt-tool “uic” which outputs *ui_MainWindow.h* that contains a *Ui_MainWindow* class. The *MainWindow* class then includes an instance of this class as a member and creates/initializes the UI controls in its constructor by calling `this->m_ui.setupUi(this)`. Any access to a control object in the UI created in the designer is then done via this object.

When building the project the *MainWindow.h*-header file is first preprocessed using the Qt-tool “moc”, which will generate the file `<build type>\moc_MainWindow.cpp` (Where `<build type>` is either debug or release, depending on what you’re building), this file contains information that Qt uses for signals and slots to work.

Slots

The *MainWindow* class contains a number of slots that are connected to the different controls in the window, such as buttons, but most importantly the *CameraWidget* and *MachineAPI*.

The *CameraWidget*-signals it listens to are, `CameraWidgetCommandReady(...)`, `CameraWidgetCommandInvalid()` and `CameraWidgetNewMachineCoordinates(...)`. When the *CameraWidget* signals a command is ready/invalid the Enqueue button will be enabled/disabled accordingly.

The *MachineAPI* was written to be non-blocking, meaning it uses threads to communicate events to the UI. This creates a problem since only the GUI thread is allowed to change anything pertaining to the GUI. Also the threads used by the *MachineAPI* are not QThreads, but native Win32 threads, so signaling from it normally using QT is not possible. To add to this, the MachineAPI notifies the GUI via a C-style callback function (not object oriented).

To overcome all of this, we first of all have a global variable “mainwindow” that the *MainWindow* class sets to its this-pointer in its constructor. The *MachineAPI* is then passed a function pointer to the C function `void on_machine_event(MachineEvent *e)` that in turn passes on the message by calling the *MainWindow* slot `MachineEventOccured(...)` with `QMetaObject::invokeMethod(...)`. This command hands over the call to the GUI thread properly.

Running commands

When the user has created a list of commands and runs it, the first command will be sent to the machine, and all the controls in the GUI will be disabled. When the *MachineAPI* signals that the command is done, the next command will be sent to the machine, and so on, until the list of commands has been run. This is separate from when the user clicks the camera widget to initiate a move. Currently there is nothing that stops you from sending several of these commands in a row, but any command sent while another one is running will simply be ignored. This can cause a bit of inconsistency in the GUI

in some cases, such as with the brightness level slider, where you can end up with a brightness level that doesn't correspond to where the slider is because some commands are skipped.

Resources

To embed resources in the application such as icons, the file XML *pnp.qrc* is used. This file can also be edited/used from within the Qt Designer. The file is compiled using the “rcc” tool, which outputs *qrc_pnp.cpp* containing byte representations in code of the resources which are compiled with the application.

GUI Layout

The main goals of the GUI were to make it:

- Easy to use
- Intuitive
- Have minimal clutter
- Only show controls when they are useable, not to confuse the user.

Overview

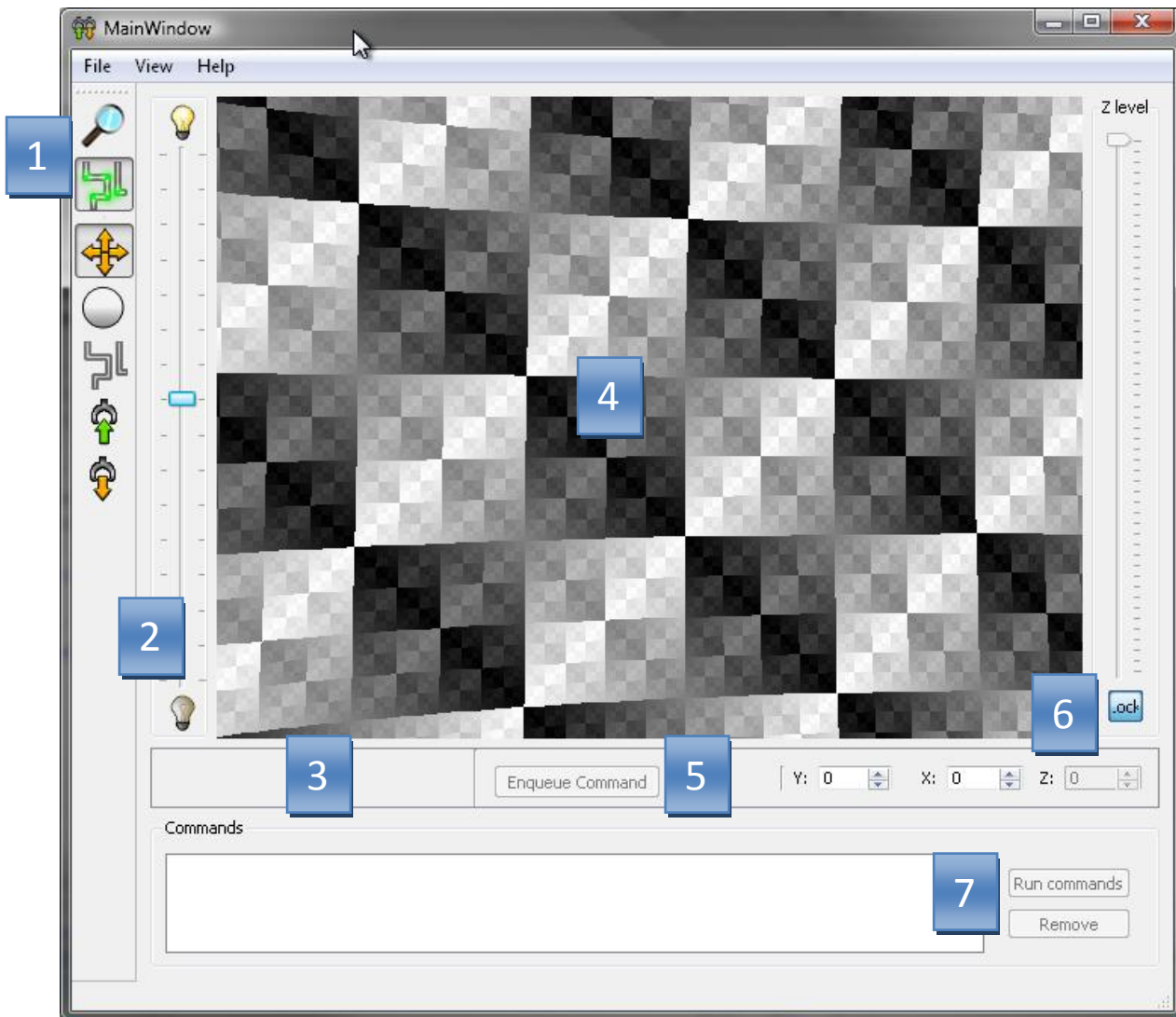









Figure 2 - GUI overview

1 – The toolbar

The toolbar consists of two parts. The “state”-controls:

-  The Zoom tool toggles the detailed camera view (not yet implemented).
-  The draw commands tool toggles drawing of the command list on the camera view.

The remaining tools changes the state of the machine, these are the “interaction”-tools:

-  The move tool changes the camera widget interaction mode to “move” which means that the machine head will move to any point that the user clicks on.
-  The dot dispense tool lets a user click where on the camera widget it wants a dot dispensed.
-  The polygon dispense tool enables the user to create a polygon to dispense by clicking on the camera widget.
-  The pick tool lets the user add three points of a component on the camera widget, outlining the corner of a component, which then will be used to calculate the midpoint and angle of the component.
-  Same as the pick tool, but for the place point.

2 – Brightness slider



The brightness slider allows the user to set the brightness of the camera light.

3 – Tool specific controls

This area will change depending on what tool is used. For instance, when creating a polygon, a “clear” and “remove last point” button will appear here.

4 – The camera widget

The main control, used to perform the actual interaction with the machine. The state of this changes based on what tool is selected in the toolbar.

5 – The enqueue button

This button will be activated when the user has created enough information on the camera widget so that a command can be created. When this button is clicked the command will then be added to the list of commands.

6 – The movements controls

If the user wants to move to a specific coordinate on the work surface, he can enter it here. To keep the user from accidentally lowering the machine head to an unsafe position, the Z movement control is by default locked, and requires the user to unlock it before changing its value.

7 – The list of commands

The list of commands shows the commands created by the user so far. By clicking the “Run commands” button, these commands will then be performed in the order they were created in (there is currently no way of reordering them). The user can also remove a command in the list using the “Remove button”. When the user selects a command in this list, and the “Draw commands” mode is set on the camera widget, it will be highlighted on the camera widget.

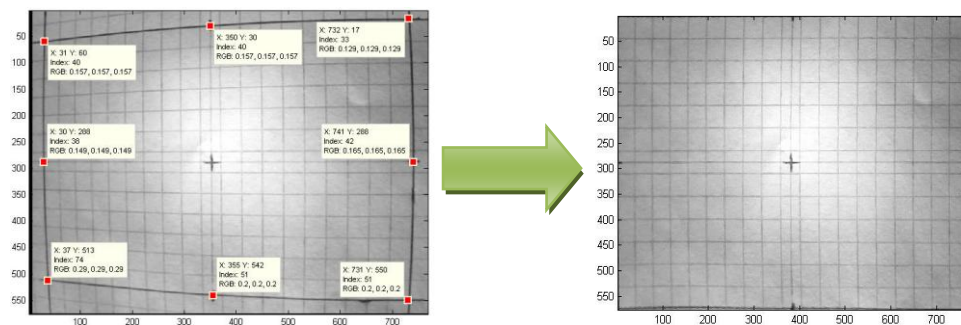
Future work

There are a lot left to be done with the GUI, here follows a list of all the things that come to mind:

Easy

- Create additional controls that let you set all the different parameters related to a command. For instance, when dispensing you'll need to set the speed, before time, after time and suck back time.
- Add settings for which camera channel to use (the eurosys card has 4 inputs, so you'd have to choose on which input what camera is located).
- Add a calibration mode for calibrating the barrel distortion correction filter using the camera widget. The user needs to be able to click on the camera widget to place 8 points on the distorted camera image, which should correspond to a rectangle on a non-distorted image.

Example:



- Add a calibration mode for setting the dispense offsets. For instance, letting the user draw a line that should be dispensed, and then after it has been dispensed, letting the user drawn the same line but in the actual position it was drawn and then calculating the offsets from this. This should also allow different calibration for different dispensing syringe, and then letting the user chose which one is used during dispensing.
- Add a calibration mode for the coordinate conversion between widget and machine coordinates.
- Calibration mode for setting the offsets for picking and placing.

- Allow the user to rearrange the commands in the command list.
- Before enqueueing a command, make sure it's valid by validating it with the MachineAPI
- Fix the progress dialog when running commands in the commands list. Currently, running a second batch of commands will hang the application, and the progress bar is not updated properly. Also if the user presses "Stop" the user will be dropped out into the GUI and all controls are enabled even though they shouldn't be until the current command has finished running (you can't interrupt a running command unless you turn the machine off).
- Load/Save all settings using the Settings library.
- Allow the user to save/load a list of commands.
- Allow the user to associate comments with commands, such as adding the name of a component that is to be picked up.
- When picking and placing, the height of the component needs to be taken into account.
- Allow editing commands that are already in the command list.

Medium

- Add an overview control (or include it as a mode in the camera widget), that gives you an overview of the work table. One idea is to save an image of the current position each time the machine head stops, and add that to the overview picture. In this way you can have a somewhat up to date picture where things on the work table are located. This should probably use some kind of compression for the stored image of the work table, since it will have a pretty high resolution.
- To allow massproduction of cards:
Allow the user to associate a command queue with a reference point, such as having a point on the circuit board that the user clicks before starting to create the command list. This reference point is then saved with the list of commands, and when the user reloads it and puts another circuit board onto the work board, the point is just clicked again, and all coordinates in the command list are adjusted to the new position.
- Make the machine useable from matlab (?). This was one of the initial wishes from the customer.

Hard

- Add auto-focus support. This is hard because there is no way of zooming the camera in any other way than moving the machine head in the Z-axis. Since you'd like to do a lot of small fine adjustment to find the proper focus, this will take a lot of time, since the machine is pretty slow to respond to commands.
- Automatically find components using image processing.

Conclusion

Results

At the beginning of the project a decision was taken that focus should be placed mostly on the API's, so that a GUI could easily be created with most of the hardware stuff out of the way.

As it turned out, the API's are fully functional and the GUI is mostly functional with a few bugs that need to be worked out and some features missing, for example offset-calibration and a "live" view of the entire workplace.

These shortcomings are explained in more details in the separate result parts for each module.

Overall the project turned out quite nicely.