A Control System for Autonomous Vehicles

Three-Dimensional Geometric Models from Pictures

Tony Gjendahl

Master of Science in Engineering and ICT
Submission date: June 2017
Supervisor: Sven Fjeldaas, MTP

Norwegian University of Science and Technology
Department of Mechanical and Industrial Engineering

Summary

This master's thesis describes an experimental process of recreating geometric models from images. This process is thoroughly described using functionality already present in the geometric modelling tool GeoMod and functionality added in this master's thesis. Some functionality is not yet implemented, but this is described with suggestions on implementation for further students. The framework described allows students to experiment with different algorithms that automatically recreates geometric models from images.

The implementation of dynamic linking of views and tools have been finished, and examples are shown. Libraries containing implementations of the BinPic-algorithm, OpenCV and OpenGL have been developed. The BinPic-algorithm creates binary geometric models of images that can be shown in the current view. OpenCV is an image-processing library that can be used to create better binary images than the ones created automatically using Qt-functions. OpenGL allows for the visualization of geometric models. This is similar to the camera-view already implemented in GeoMod, but it provides features like z-buffering, adding light sources, adding different colors inside and outside models and adding images as texture to geometric models. These libraries are used in the development of the framework for recreating models from images.

Sammendrag

Denne masteroppgaven beskriver en eksperimentell prosess om hvordan man kan gjenskape geometriske modeller fra bilder. Denne prosessen er beskrevet med funksjonalitet allerede eksisterende i den geometriske modellereren GeoMod og funksjonalitet utviklet i denne masteroppgaven. Det er fortsatt manglende funksjonalitet, men disse er beskrevet i detalj med forslag til implementering. Rammeverket lar senere studenter på samme prosjekt ekperimentere med forskjellige algoritmer for å gjenskape geometriske modeller fra bilder.

Jeg har implementert dynamisk linking av views og tools, og vist eksempler. Bibliotek med implementasjon av BinPic-algoritmen, OpenCV og OpenGL har blitt utviklet. BinPic-algoritmen gjenskaper binære geometriske modeller fra bilder, som kan bli visualisert i forskjellige kamera. OpenCV er et bilderedigerings-bibliotek som kan brukes til å lage bedre binære bilder enn de opprettet automatisk ved bruk av Qt-funksjoner. OpenGL visualiserer geometriske modeller. Dette er de samme modellene som er visualisert i det originale kameraet allerede implementert i GeoMod, men det gir oss mulighet til å innføre egenskaper som z-buffering, lyskilder, forskjellig farge på innsiden og utsiden av modeller og mulighet til å legge til bilder utenpå de geometriske modellene. Disse bibliotekene er brukt i prosessen med å gjenskape geometriske modeller fra bilder.

Preface

Autonomous vehicles have long been looked on with keen interest. In recent years several companies have begun development of their own vehicles for different purposes. Some of these companies use Trondheim as their test-area during this development. Examples of such planned vehicles are the passenger-ferry [1] developed by AMOS and the commercial ferry [2] being developed by Kongsberg. This shows that the GeoMod-project started over 10 years ago is highly relevant today.

In fall 2016 I received the current version of the GeoMod-program. It then showed a lot of signs of being transferred from a UNIX-platform to Windows, OS X and Linux. A lot of code was implemented, but not functional on the new platforms. Quite some time has been spent on rewriting old code, as well as developing new, to get the program fully operational. The program delivered at the end of this master's thesis has sorted out a lot of these problems. One example is the dynamic linking of geometric models, views and tools. Other practical tools like OpenCV and OpenGL have been included in separate libraries. These allows the program to utilize image processing functions, as well as a new, improved visualization on the new platforms. With an extra focus on the recreation of models from images, this gives later students a good framework for experimentation and further development. After compiling and running the code on both Windows, OS X and Linux I feel confident that the program now allows students to do further development without too much time spent on fixing old code.

I would like to thank my supervisor, Sven Fjeldaas, for guidance throughout the previous year. A lot of the bugs encountered, with fixes, and other results would not have been achieved without close cooperation and countless hours of discussions in his office.

Table of Contents

Summary **i**

Sammendrag **iii**

Preface **v**

Table of Contents **ix**

List of Figures **xii**

1 Introduction **1**

2 Program development-platform **3**
2.1 The programming language C++ 3
2.1.1 Pointers 3
2.1.2 Function overloading 4
2.1.3 Inheritance 4
2.1.4 Dynamic Linking 5
2.2 Qt Creator 5
2.3 Installation of Qt Creator 5

3 Dynamic linking of Views **9**
3.1 Dynamic Linking of Camera 9
3.1.1 Re-writing layout of the Views- and Tools-manager 10
3.1.2 Dynamically linking the camera 10
3.1.3 Remove statically linked Camera 12
3.1.4 Importing several instances of cameras 12
3.2 Structuring of Camera-views and Tools 12
3.2.1 Management of models in existing code 12
3.2.2 Management of views in existing code 13
3.3 Control-panel for the camera-view 14

3.3.1 Problem specification 14
3.3.2 Review of the current code 15
3.3.3 How the camera-view should work 16
3.3.4 Finding the models and drawing them 17
3.3.5 Linking control-panel as a separate tool 20
3.4 Linking in picture-view 21
3.4.1 Drawing models in the picture-view 23
3.4.2 Linking in letters in the picture-view 26

4 Create models from binary images 29
4.1 Dynamically linking tools 29
4.1.1 File-browser for finding images 30
4.1.2 Previewing .png-files 32
4.2 Adding the BinPic-algorithm to the library 34
4.2.1 Compiling the BinPic-algorithm 34
4.2.2 Troubleshooting BinPic 35
4.2.3 Small changes after discussions with my supervisor 38
4.2.4 Further development on the BinPic-algorithm 41
4.2.5 BinPic-algorithm with different extensions 42
4.2.6 Drawing the models in the current view 44
4.2.7 Solution to drawing the models in the current view 45

5 Image processing 47
5.1 Binary geometric models without image-processing 47
5.2 Image-processing 48
5.2.1 Library for image-processing 48
5.2.2 The HSV-format 52
5.2.3 OpenCV and the BinPic-algorithm 52

6 Experimental Process 55
6.1 Experimental process of creating models from images 55
6.1.1 Creating a physical model of the Triangle Prism 56
6.1.2 Creating binary images of the model 56
6.1.3 Identifying properties and recreating the model 58
6.2 Notes and possible problems during development 61
6.3 Summary 62

7 OpenGL 63
7.1 Re-implement OpenGL in GeoMod 63
7.1.1 Drawing the models in the system 65
7.1.2 Drawing the Robot-arm in the OpenGL-view 71
7.1.3 Drawing complex surfaces 74
7.1.4 Add colors to the models 76
7.1.5 Drawing edges in the system 78
7.2 OpenGL and further development 83
7.2.1 Drawing edges 83

7.2.2 Drawing points 84
7.2.3 Drawing surfaces with the same coordinates 84
7.2.4 Adding Texture to the Models 85
7.2.5 Unclosed Models 86
7.2.6 Adding Shadows 87
7.2.7 False Volumes 88
7.2.8 Using Code From the Old Drawing-algorithm 88
7.2.9 OpenGL and the old drawing algorithm 89

8 Conclusion **91**

Bibliography **91**

A Installing and testing OpenCV **95**
A.1 Installing and testing OpenCV on Ubuntu 95
A.2 Installing and testing OpenCV on Windows 97

B Short introduction to BMP and BMX **99**
B.1 The BMP-extension 99
B.2 The BMX extension 99

C General comments **101**
C.1 Dynamic libraries 101
C.2 Generally about the code 101

D Risk assessment **103**

List of Figures

2.1 Database-manager in GeoMod 6

3.1 Dynamically linked camera 11
3.2 Dynamically linked camera with models 15
3.3 Dynamically linked camera without models 16
3.4 Adding a new camera through the control panel 17
3.5 Widget for adding a new camera 17
3.6 Opening the newly added camera 18
3.7 Dynamically linked model not working 20
3.8 Dynamically linked picture-view not working 23
3.9 Statically linked picture-view 23
3.10 Letter A dynamically linked in picture-view 26
3.11 Dynamically linked letters in picture-view 27

4.1 Widget shown from dynamically liked test-tool 30
4.2 First widget that opens from the 10_BinpicTool-library 31
4.3 Filebrowser with .png-images 31
4.4 Window with png-images 32
4.5 Widget with a png-image of Super Mario [3] 33
4.6 Widget with kitten.jpg-image 34
4.7 Panel for choosing if the image is complete 35
4.8 Preview of the grey-scaled image A 37
4.9 The camera view when drawing the rhombus 38
4.10 Image of the statically linked view with rhombus 39
4.11 Updated previews 40
4.12 The BinPic-tool with new button and button-names 41
4.13 BinPic-model of A in the camera-view 42
4.14 Error-message when importing eps-files 43
4.15 Result of previewing A as a binary image 46

5.1 Preview and BinPic-model of the Linux-logo 48

5.2 Binary images created by filtering on the orange colors 50
5.3 Controlpanel for creating binary images 50
5.4 Graphical representation of the HSV-format 52
5.5 Widget for saving images 53
5.6 New binary image and model 53

6.1 Examples of different models 56
6.2 Images of the physical model of the prism 57
6.3 The red surface after using the OpenCV-library to create a binary image . 58
6.4 Binary prisms 58
6.5 Creating a model of the prism in GeoMod using BinPic 59
6.6 Prism model with edges and points identified 59
6.7 Example of recreated prism in GeoMod 60

7.1 Drawing a triangle with different colors using OpenGL 64
7.2 Figure of creating a cube in OpenGL 65
7.3 Other recreated models in OpenGL 71
7.4 Example of an unclosed triangle 71
7.5 Recreating the robot-arm using OpenGL 72
7.6 Robot-arm with offset 73
7.7 Robot-arm with rotation 74
7.8 Two different way of drawing polygons using triangles 75
7.9 Robot-arm with two different rotations 76
7.10 Models with colors 77
7.11 Robot-arm with shading 79
7.12 Models with edges 82
7.13 Several models with shade and z-buffering 83
7.14 Showing hidden edges in pyramid 84
7.15 Surfaces with the same coordinates 85
7.16 Drawing cubes with texture 86
7.17 The unclosed cube with four surfaces using OpenGL 87
7.18 The unclosed cube with four surfaces using the old drawing algorithm . . 87
7.19 Normal and false volume 88

A.1 Showing the Linux logo in a OpenCV-widget 97

Chapter 1

Introduction

GeoMod is a geometric modelling tool which allows the user to visualize and control geometric models. The finished version of the program is intended to control an autonomous underwater vehicle. GeoMod should navigate the vehicle without human interaction. For this to be possible it has to be able to perceive its surroundings, interpret it and find a safe navigation path.

The vehicle guided by the GeoMod-program is intended to perceive its surroundings through cameras and other sensors. This master's thesis will focus on feedback from cameras, namely images. The feedback should be used to build up an internal representation of the surroundings, that can be used to find a safe path for the vehicle. The problem description states that the final product of this master's thesis should be an outline of a process recreating 3D-models from images. Initial discussions with my supervisor revealed that this process should help build the internal representation of the surroundings. This process will be based on functionality already present in the system, and functionality to be implemented in this master's thesis. Some of the steps in this process are outside the scope of this master's thesis. These will be described, but left for further students to implement. The process should allow further students to see how the whole process works, and how their master's theses fit into it.

The GeoMod-program should allow vehicles to instantly respond to changes in the surroundings. The code needed to respond to these changes will be included into the main program when the changes occurs. Dynamic linking allows us to do exactly this. When a specific tool is needed, the program finds the code and adds it. The version of GeoMod received at the beginning of the master's project was designed in such a way that this could be achieved, but certain parts were yet to be implemented. At the end of my master's project the program allowed me to link in geometric models. The program is intended to link in geometric models through the Database-manager, different views for visualization through the Views-manager and different tools through the Tools-manager. For now, only the Database-manager works as intended, so this thesis will begin by implementing miss-

ing code in the Views- and Tools-manager.

The problem description states that simulated visualizations should be linked in dynamically, a program for following contours in a binary image re-implemented, and program for image-processing added to the system. This master's thesis will then continue by implementing separate libraries for a simulated camera-view and a picture-view. An algorithm called BinPic on the old UNIX-system will be re-implemented and tested with the current version of GeoMod. The resulting geometric models created by the BinPic-algorithm depends on the input-image. I will then find and incorporate an appropriate image-processing tool which allows the user to achieve better results. These will then be described as parts of the process of recreating models from images.

Chapter 2

Program development-platform

The code in this project is written using the programming language C++ and the IDE(Integrated Development Environment) Qt Creator. Both C++ and Qt Creator was used in my master's project. The code I will develop in my master's thesis is a further development of work done in my master's project, so everything is up and running at the beginning of this thesis. I will in this section briefly describe parts of C++ used here, and give an introduction to Qt Creator.

2.1 The programming language C++

C++ is an object-oriented programming language developed by Bjarne Stroustrup [4]. Shortly, object-oriented means that the programmer can instantiate classes as many times as needed, and use the functions defined in these classes through the instantiated object. I will in this master's thesis only comment on a tiny subset of C++ functionality, namely features essential to the GeoMod-project.

2.1.1 Pointers

Pointers allows the programmer to access addresses, and manipulate the content these addresses points to. This is very powerful when utilized correctly, and can increase both the efficiency and performance of a program. A drawback is that, if used incorrectly, it can cause inaccessible code and memory leaks [5]. It is possible to write C++-code without using pointers, but that is not desirable. Pointers separate themselves from ordinary variables in that they contain an address pointing to a location in the memory where the data is stored, and not a representation of the data itself. The location in memory the address points to is typically 8 bit, or one byte. To store an *int* in memory one needs 4 bytes. The address will then point to the location of the first byte, and internally remember the other three locations in memory where the rest of the *int* is stored. Creating a new pointer in

C++ is done in the following way:

```
int *p1;
```

p1 is here a pointer to an *int*. To create a variable that points to the value 2.0 we write the following:

```
double *p2 = 2.0;
```

p2 points to a location in memory containing the number 2.0. Pointers are not limited to numbers. They can point to strings, objects, functions and so forth. C++ also contains a type of pointer called an *opaque pointer*. Pointers of this type points to data-structures where the type hasn't been specified. This is especially handy when dealing with Dynamic Linking.

2.1.2 Function overloading

Function overloading allows the programmer to create several functions with the same name, differing in the number of input-parameters and implementation.

```
1 void print(int i) {
2     // Implementation 1 here
3 }
4
5 void print(double d, char c) {
6     // Implementation 2 here
7 }
```

The two functions above have the same name, but different implementation and input-variables. The function-call print(5) will call the first function, while print(2.0, "Hello World") will call the second. Function overloading allows us to have several functions with the same name, and the one used depends on the function-call.

2.1.3 Inheritance

Inheritance is central in object-oriented languages. It allows one class to inherit properties from another. The programmer can then use functions from the parent-class in the sub-class. New functions can also be defined in the sub-class. If one of the functions in the sub-class has the same name as one in the inherited class, the one in the sub-class will be used if this function is called. Inheritance of *abstract* classes is also possible. An abstract class is called an *interface* in C++. An interface contains functions that are defined, but not implemented. The class that inherits, or *implements*, the *interface* **has** to implement the functions defined. This ensures the programmer that all classes implementing the *interface* at least contains the defined functions. The programmer can also implement other

functions in the sub-class.

2.1.4 Dynamic Linking

C++ is an extension of the C programming-language which means that components from C can be used in C++. For us, it means that we can use functions for dynamic linking defined in C. C++ has no equivalent functions for dynamic linking. For the compiler to handle function overloading, C++ adds characters to the function-name describing the input-parameters. Standard C does not support function overloading, so the extra characters will not be added. C-code inside C++ then have to be declared with the word *extern*. I will not go into the specifics of this declaration, but the result is a pointer to the C-functions inside C++-code. Quite a few problems encountered in this master's thesis is concerned with these pointers. The dynamic linking allows us to add code when needed, which is very useful for autonomous vehicles.

2.2 Qt Creator

Qt Creator [6] is a platform-independent tool for program development using C++. In this master's thesis Qt Creator is used as an IDE, with functionality added through external libraries. The IDE allows me to write code that can be compiled directly in Qt Creator. I will not venture deep into the functionality added through Qt Creator, but I will highlight one, namely Qt Widgets [7]. Qt Widgets allows the user to easily create graphical elements and show them on the screen. An example of such an element is shown in figure 2.1. The figure shows the Database-manager in GeoMod. Here the user can link in a model through the buttons at the bottom of the widget. When a button is clicked the user expects something to happen. This brings me over to another feature used alongside Qt Widgets, namely Signals & Slots [8]. Signals & Slots allows the user to easily connect buttons to certain code that should be invoked when the button is clicked. This is one of the features that supports the choice of Qt Creator, because GeoMod consists of numerous buttons with different behaviour.

2.3 Installation of Qt Creator

One of the main focuses of my master's project was cross-platform development. There I created manuals for installing Qt Creator on Windows, OS X and Ubuntu. This thesis also has cross-platform development in mind, but it is not that evident since most of this work was completed in my project. The installation-manuals are accessible through my project report. I will go through the main points here, since some of these steps are important, especially on Windows.

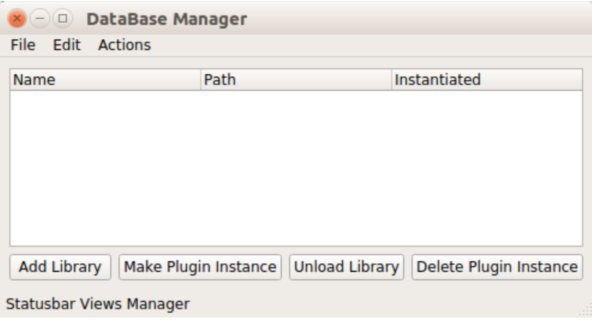


Figure 2.1: Database-manager in GeoMod

From version 5.4 of Qt Creator, the Qt-module QtWebEngine was discontinued and replaced with QtWebEngineWidgets. This replacement doesn't affect users on OS X and Ubuntu. The gcc-compiler that comes with these operating systems can handle this new module, but the MinGW-compiler on Windows cannot. To compile the program on Windows, the user has to install Microsoft Visual Studio and use the MSVC-compiler. Microsoft Visual Studio can be installed freely from [9]. The user then has to install a specific version of Qt Creator from [10] that sets up the computer to use the MSVC-compiler. Installing Qt Creator on Windows then consist of the two steps: 1) Install Microsoft Visual Studio. When doing this the user has to enter "advanced installation" and check of "Visual C++" under "Programming Languages". 2) Install Qt Creator. Here the correct version of Qt Creator has to be installed. When using the MSVC-compiler, the installation with "VS" in parenthesis has to be chosen.

The installation of Microsoft Visual Studios and Qt Creator has been shown to work on native Windows on my computer and on a virtual version of Windows on my Mac. It also worked on my supervisors old computer, and on the computer of another student on the same project. It did, however, not work on my supervisors new computer. All NTNU-computers are set up in a way such that the default location when storing files is on a cloud. This ensures that all files stored in a default location, for instance "Documents", can be recovered if the computer is lost or breaks down. This can be overridden by explicitly telling the computer to store each item on the "C:"-drive. This solution has been upgraded from the time my supervisor got his previous computer. This default behaviour causes some troubles when installing Qt Creator. The installation of Microsoft Visual Studio has been verified by creating a simple program printing "Hello World!" to the console. Installing and running a simple code in Qt Creator yields the following error: "'cl' is not recognized as an internal or external command". It means that Qt Creator cannot find the command-line tool installed with Microsoft Visual Studio. This is something that should have been set automatically by the installation, but the path seems to have been set wrong. This issue was fixed by adding the appropriate folder to the list of system variables. A description on how to add a path to system variables is shown in [11]. I added the path "C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin\amd64" and restarted the computer. This is the path to the command-line tool on my supervisors computer, a native 64-bit Windows.

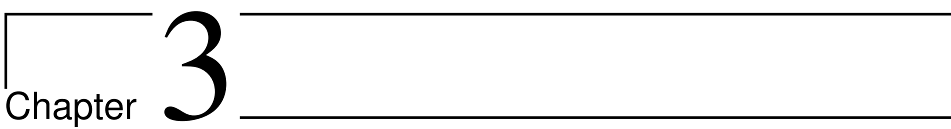
This solved the previous error, but running the code again yielded errors of the following kind: ”C1083: Cannot open include file: ’string’: No such file or directory”. This tells us that Qt Creator is unable find the file called ”string”. There are also similar errors with other files. These are C++-headerfiles that should have been found directly by Qt Creator. I located these files and added an INCLUDEPATH-statement in the .pro-file of the current project. This worked for the headerfiles, but similar errors regarding library-files appeared. I tried adding these with INCLUDEPATH as well, but this did not work.

After searching online and looking through error-messages, we found out that the root of the problem was the way the computer is set up. NTNU-computers do not allow the users, even local administrators, to edit the system-files. This in turn might explain why the installation works on other computers, and not on my supervisors.

My supervisor then contacted ”Orakeltjenesten”. They set up the computer and should therefore be able to grant the access-rights needed. It took numerous emails and visits before they finally provided the correct access-rights. It turned out that the root of the problem was that my supervisors user didn’t have the rights to read/write to the windows-registers. With the old access-rights the registers were unavailable, which could be seen by opening the windows-menu, typing ”regedit” and selecting ”regedit Run command”. This opens a new window where the registers should be available. This window only displayed a message saying that the registers were unavailable. Running the same command after ”Orakeltjenesten” had changed the access-rights shows the registers, and allows the user to edit them.

Compiling the GeoMod program-files in Qt Creator, after getting the correct access-rights, now worked without errors. The result that this worked right away was a bit surprising to both me and my supervisor. We thought that the programs only read/wrote to the registers once, namely during the installation, and that we had to re-install both Microsoft Visual Studio and Qt Creator. The fact that this isn’t necessary shows us that Qt Creator reads from the registers each time the program is started, and was able to find all libraries without re-installation.

It turned out, as seen in the text above, that the described installation worked on this computer as well. The problem here was due to the way the computer was set up from NTNU, and out of our hands. It did, however, take a lot of time and effort to try other installations and fixes before this was known. We tried installing several different versions of both Microsoft Visual Studios and Qt Creator, as well as the fixes outlined above.



Dynamic linking of Views

Dynamic linking is a central concept in this master's thesis. Most of the code written will be implemented in separate libraries that can be linked in dynamically. The main idea behind the dynamic linking is described in detail in my master's project, so only code relevant to the newly created libraries will be described here.

Dynamic linking allows the program to load code into the main program and execute it through an interface. This interface can be accessed through the Database-, Views- and Tools-manager. They link in models, views and tools, respectively. The models, views and tools are created in separate libraries implementing an interface, and contains the code for the models, views and tools themselves. When a library is added through a manager, the main program runs the code in the library through interface-functions. Being able to link in libraries dynamically allows us to keep a lot of the code separate from the main program. We can then have a smaller code-base with the necessary functionality, and link in other functionality when needed. It helps us keep a smaller code-base for the main program, which in turn means that we have less compiled code if it isn't needed. It also allows us to track errors more easily. If the program crashes when a library is linked in, the error is most likely contained within this particular library.

The dynamic linking worked on the UNIX operating system, so it is not a new feature on this project. The functionality did unfortunately not handle the transition to the new platforms. I have access to this implementation on the UNIX-platform. The camera-view is now linked in statically, so I will use this code combined with the implementation from UNIX when trying to link in views dynamically.

3.1 Dynamic Linking of Camera

The code for the camera-view is currently linked in statically. We want to move this functionality into a library, so that the camera-view can be linked in when needed. Before

I start on this task I have to rewrite the Views-manager to handle dynamic linking properly.

3.1.1 Re-writing layout of the Views- and Tools-manager

In my master's project I had to rewrite some code in the Database-manager for it to link in models correctly. This library works and looks as it should as I now am able to link in models through this widget. I will use this as a template for the Views-manager, so that it looks and behaves in the same manner. The same will also be done with the Tools-manager. The basic idea is that models should be linked in through Database-manager, camera and other views through Views-manager, and different tools through Tools-manager. I copied the code from the Database-manager to the Views- and Tools-manager. After changing names in the implementations, the program now has three widgets with the same appearance and code. This is a good starting point for the dynamic linking of views and tools.

3.1.2 Dynamically linking the camera

With the Views-manager looking and behaving as the Database-manager I started on the task of dynamically linking in the camera-view. I started by creating a new library that implements the functions in the interface. Then I copied the code for the statically linked camera into the library and called its constructor in the interface-function newInstance(). I will not show all of the code in the library here. It can be found in the folder "DynamicLinkingTests/07_Camera".

The code for the three managers are now equal. This means that models can be linked in using all of the three managers. To change this I added the following code in the function makePluginInstance() in "viewsman.cpp":

```
int index = factory.getPlugins().size() - 1;
CameraControlWdgt* cameraCtrlP = factory.getPlugins()[index].
                                    pluginP->getCamera();
cameraCtrlP->show();
```

Line 1 in the code above finds the index of the last plugin added to the system, now the camera-view. Then it uses the function getCamera() in the library. getCamera() is a new interface-function that returns a pointer to the camera-view created in the library. Calling the function show() on this camera opens the camera-view. This shows the camera, but none of the statically linked models appear. This comes from the fact that the two functions called showOnTop() and showCameraOnTop() are called in the library. The library doesn't know about the statically linked models the main program, and will not draw them. The same happens for dynamically linked models. These are added to the list of models in the main program, so they are invisible to the library as well. To solve this problem I added the two function-calls to the function makePluginInstance() in "viewsman.cpp":

```
int index = factory.getPlugins().size() - 1;
CameraControlWdgt* cameraCtrlP = factory.getPlugins()[index].
                                    pluginP->getCamera();
if(cameraCtrlP != nullptr) {
    cameraCtrlP->showOnTop();
    cameraCtrlP->showCameraOnTop();
}
else {
    std::cout << "The provided library is either buggy or not a camera- "
        "view. Have you checked that all interface functions are "
        "implemented?" << std::endl;
}
```

The code snippet above finds the camera in the list of linked plugins as earlier. It then gets the pointer to the camera through the interface-function getCamera(). The if-statement checks if this pointer actually points to a camera or if it is a nullpointer. The camera and its control panel will be shown if it is a pointer to a camera, otherwise a statement will be printed to the terminal/console telling the user that something went wrong. The function getCamera() has to be implemented in other libraries not containing a camera-view. Here this function should return a nullpointer which ensures that other libraries aren't linked in as camera-views.

We can see in Figure 3.1 that the statically linked models now appear in the dynamically linked camera. This is because the two functions showOnTop() and showCameraOnTop() are called in an environment where the statically linked models are known. The dynamically linked camera now works as the statically linked camera. Dynamically linked models are also drawn when linked in.

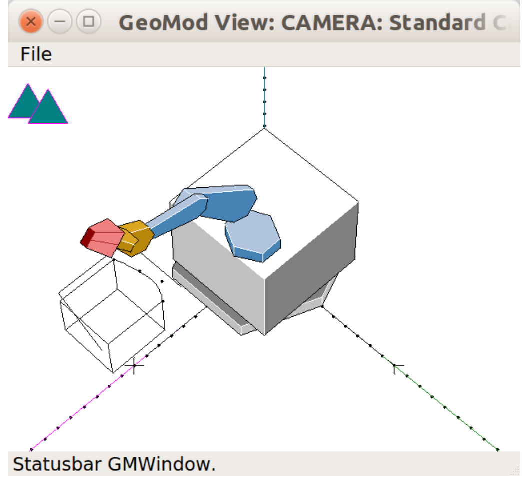


Figure 3.1: Dynamically linked camera

I added the same check as in the code above in the Database-manager to verify that only

models can be added using this manager. By trying to link in a camera or tool in Databasemanager an error message is displayed.

3.1.3 Remove statically linked Camera

At this point I tried removing the statically linked camera. When dynamically linking in the view now, nothing happens. When a new CameraControlWdgt is instantiated, a private vector called cameraManager is created. It contains information about the camera-view and its setup. This is now created in the library, and not in the main program. Calling the interface-function getCamera() returns a pointer to the camera-object and not the settings associated with the camera. When the function showOnTop() is called, it checks that the cameraManager-list is non-empty and tries to use this setup. In this case the main program thinks the cameraManager-list is empty because it does not know about the list created in the library. Which in turn means that the camera-view isn't shown. I fixed this by keeping the following line in "entrance.cpp":

```
camCtrlP = new CameraControlWdgt( nullptr );
```

This creates the cameraManager-list in the main program. The check in showOnTop() is now passed, and the camera-window is drawn on the screen. This is not an optimal solution, but it works. I will later look at how the cameras and other views are stored. I will then review and perhaps update this solution.

3.1.4 Importing several instances of cameras

On the UNIX-system several instances of the camera-view could be opened by the user. The models in the system could then be viewed from different angles at the same time. We want the same behaviour, but the current version only allows for one instance of the camera-view. Creating another instance currently prints an error-message to the terminal.

3.2 Structuring of Camera-views and Tools

I will in now shortly look at how the models are stored in our program and compare this to the way the views are stored. The models are now stored in such a way that several instances of the same model can be created. This is exactly the same behaviour as we want for the camera-views.

3.2.1 Management of models in existing code

The models in the main program are stored as a vector of pointers to several ModelData-instances. These instances are tied together by a doubly linked list which can be entered

through a pointer. Whenever a new model is created, the ModelData-pointer will be stored in the vector mdlDataPs. The doubly linked list, and entry point to it, is created when the constructor, Models(), in ”modelPP.cpp” is called.

3.2.2 Management of views in existing code

I will use this main idea for managing camera-views as well. They should now be stored in a vector containing several camera-views. A similar solution was implemented in the UNIX-system, but it was disabled after the migration to Windows. Getting the camera-view to work in the first place was prioritized, not the possibility of adding several instances.

The models are stored in a doubly linked list. Looking at the code for the models and views I found that storing the views as a doubly linked list isn't necessary. The desired functionality can be achieved in a simpler way by storing pointers to the camera-views in a vector. In ”camctrl.cpp” I changed the single pointer to the camera-view from:

```
CameraControlWdget *CameraControlWdgt::objectP = 0;
```

to a vector of pointers:

```
std::vector<CameraControlWdgt*> CameraControlWdgt::objectPs(0);
```

A few lines in the constructor also had to be changed to account for the possibility of adding more than one camera instance. The previous code set the CameraControlWdget-pointer, ”objectP”, equal to ”this”. If this pointer already had been set, an error was displayed, and the program quit. This means that adding more than one camera-view caused the program to close all windows. This was implemented as follows:

```
1 if(objectP) {
2     std::err << "For now, CameraControlWdgt only allows "
3              << "one instance of itself\n"
4              << "Change the implementation of "
5              << "CameraControlWdgt::updateAllLabels().\n";
6         exit(1);
7     } else objectP = this;
```

To account for several camera instances I replaced it with the following:

```
objectPs.push_back(this);
```

This line adds each new camera-pointer to a vector, which allows for more than one camera-view. When more than one camera-view is added we want to give them different camera-numbers. The code below sets the first camera-number to -1 and increments it directly. This means that the first camera-view will get the number 0. This number is incremented for each added camera-view, which means that every instance will get a unique

number. This allows us to access each individual camera-view.

```
if(objectPs.size() == 1) {
    cameraNumber = -1;
}
cameraNumber++;
```

The program now allows for more than one view, and each one can be accessed through an individual index. The code in the function makePluginInstance() in ”viewsman.cpp” that handles several views is shown below. Here only the code inside the if-statement is shown since the rest of the function is left unchanged.

```
cameraCtrlP->showOnTop();
camRecManP = cameraCtrlP->getCameraManagerP();
camRecManP->add("Camera " + std::to_string(index));
cameraCtrlP->showCameraOnTop();
RefreshAllViews::update();
```

The first line above shows the widget containing the control-panel. Line 2 gets the camera-manager and stores it in the variable camRecManP. The camera-manager is then used to add the current camera-view to the vector of camera-pointers in the main program. The view is here added to the same list as statically linked views. Each added camera-view is given a different name, ”Camera ” + the index starting at 0. The camera-view is shown in line 4, and updated in the last. This ensures that all models in the main program are drawn in the newly added views. Each new instance now gets a unique name, and the models are drawn correctly. The result can be seen in figure 3.2. I mentioned earlier that the following line in ”entrance.cpp” was added to get the dynamically linked camera to work:

```
camCtrlP = new CameraControlWdgt( nullptr );
```

This can now be removed since each camera-view is directly added to a vector in the main program.

3.3 Control-panel for the camera-view

All camera-views open a control-panel used to rotate and translate the views to see the models from different angles. These are currently not working. They are shown, but nothing happens when they are clicked. This section will look into this problem and propose possible solutions.

3.3.1 Problem specification

The control-panel opened alongside the camera-view only works when it's linked in statically. A similar control-panel is opened when a model is linked in. The same happens

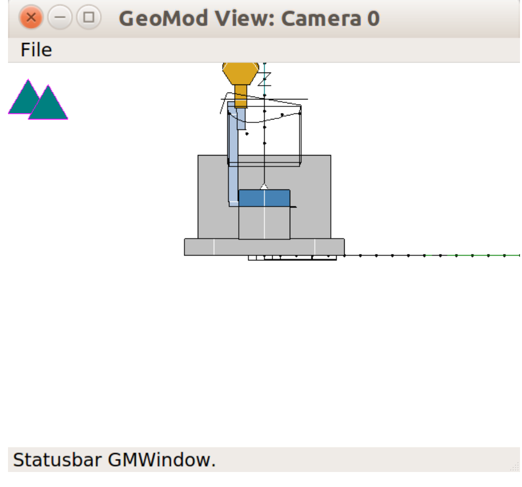


Figure 3.2: Dynamically linked camera with models

here, it only works for models linked in statically. A larger control-panel with more functionality is being developed by another student on this project. This should work on both models and views, but it's currently only implemented for models. This control-panel works on dynamically linked models, so I assume that it, when finished, will work on the views as well. This control-panel allows the user to perform the basic operations, as well as more complex operations. It is, however, desirable to get the smaller control-panels to work as well. This saves the user from opening a control-panel manually.

The control-panel is initialized alongside the camera-view in the library. The function showOnTop() in "viewsman.cpp" then shows this control-panel. Before this function is called, the pointer to the camera is found through the interface-function getCamera(). This pointer is added to the vector of cameras in the main program. The problem here is that the control-panel tries to do operations on the view created in the library, not the main program. The changes done on this view will not be shown in the main program. The view is already added to the vector in the main program, and it's no longer concerned about changes in the library. This means that the control-panel updates a views that isn't visible, and the one in the main program will not be changed.

3.3.2 Review of the current code

In the current version of the code, most of the functions related to showing the camera-view is called in "viewsman.cpp". This shows the models correctly inside the view, but the control-panel isn't working. At this point I tried moving the function-calls to the constructor in the library. The implementation in "viewsman.cpp" then becomes:

```
int index = factory.getPlugins().size() - 1;
CameraControlWdgt* cameraCtrlP = factory.getPlugins()[index].
                                    pluginP->getCamera();
if(cameraCtrlP != nullptr) {
        RefreshAllViews::update();
}
else {
    std::cout << "The provided library is either buggy or not a camera- "
        "view. Have you checked that all interface functions are "
        "implemented?" << std::endl;
}
```

We see that this code only calls the function RefresAllViews::update() when a camera-view is linked in. The other functions are moved to ”camera01_if.cpp” in the library. Linking in the library now gives us a working control-panel, but none of the models are shown in the resulting view. This can be seen in figure 3.3 where only the coordinate-system is visible. At this stage we have two different solutions. The first solution draws the models in the system, but the control-panel doesn’t work. The other solution doesn’t draw the models, but the control-panel works.

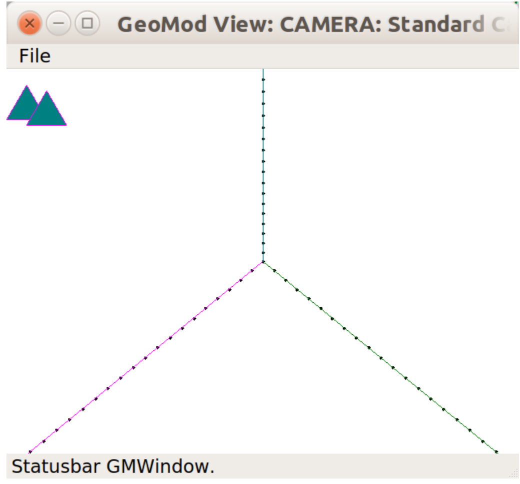


Figure 3.3: Dynamically linked camera without models

3.3.3 How the camera-view should work

Before continuing I discussed the problem with my supervisor. The dynamic linking of views worked on UNIX, so we looked through images showing the results on this platform. The dynamically linked camera-view showed the models, and a working control-panel could be opened. It then allowed the user to open new camera-views using this control-panel. A new widget is then opened where the user can choose the type of view and show

it with the main program. This functionality turned out to work in the second solution proposed above. The widget is opened as shown in figure 3.4. Pressing ”Add” opens the widget shown in figure 3.5. Choosing ”Standard Camera” in this dropdown-menu adds a new camera-view in the dropdown-menu at the bottom of the control-panel. Opening this view can be done by selecting it as shown in figure 3.6 and pressing ”Show Camera”. A new camera-view is then opened, and it can be controlled by the control-panel. This new camera-view looks the same as in figure 3.3. Now two camera-views and one control-panel is opened. The camera-view controlled is the one currently selected in the dropdown-menu at the bottom of the control-panel. The user can add as many cameras-views as needed, and control a given one by selecting it in the dropdown-menu and clicking ”Show Camera”. This behaviour is the same as on the old UNIX-system, except for the models not being drawn.

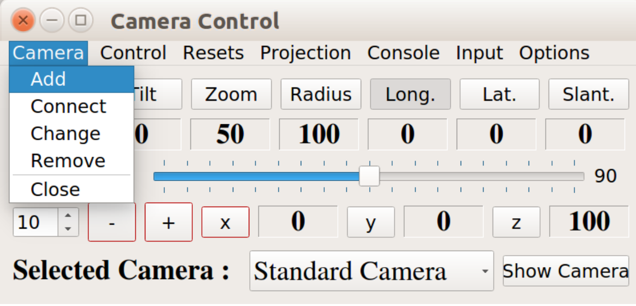


Figure 3.4: Adding a new camera through the control panel

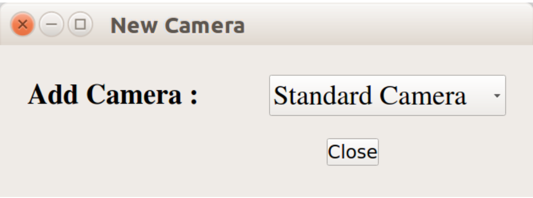


Figure 3.5: Widget for adding a new camera

3.3.4 Finding the models and drawing them

The next step is to draw the models in the dynamically linked camera-view. To do this I extended the interface in ”plugininterface.cpp” with a function called updateCamera(). This takes a vector of model-pointers as input. The implementation in ”camera01_if.cpp” in the library is shown below.

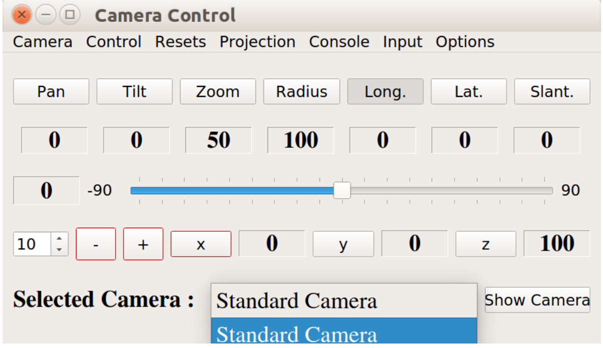


Figure 3.6: Opening the newly added camera

```
void Camera01_if::updateCamera(std::vector<Models::ModelData*> models) {
    for(unsigned int i = 0; i < models.size(); i++) {
        RefreshAllViews::update(models[i]);
    }
}
```

The function above loops through the vector of models and calls the function RefreshAllViews::update(model). It takes a single model as an argument, which means we have to loop through the vector. We now have a function that will draw all the models, but we still need to send the models from the main program to this function. Our models are stored in the vector mdlDataPs in ”modelPP.h”. This vector is private, so I implemented a get-function called getModels() that allows us to reach these pointers. The implementation in ”modelPP.h” is as follows:

```
std::vector<ModelData*>* getModels() {return &mdlDataPs;}
```

In the function makePluginInstance() in ”viewsman.cpp” I changed the code inside the if-statement to:

```
Models models;
factory.getPlugins()[index].pluginP->updateCamera(models.getModels());
```

”modelPP.h” was included at the beginning of the file to be able to use the newly created getModel(). Then I replaced the previous statement, RefreshAllViews::update(), with the code above. Line 1 creates a new models-object containing the models in the system. Line 2 calls the interface-function updateCamera() with the models as the only argument. This sends the models-pointer from the main program through the library.

The models are now drawn in the dynamically linked view, and the result is the same as in figure 3.2. The current solution seems to be working as on the UNIX-system. I can open several camera-views and control them with one control-panel. But, after some testing, bugs appeared. Now, the advanced control-panel described earlier doesn’t work on

the models shown in the camera-view. The reason is that the camera-view and the models now are contained within the library, and not the main program. The main program doesn't know that this camera-view with the models exists, and moving the models in the main program has no effect on the ones in the library. This was confirmed by the text printed to the terminal when a model is moved with the advanced control-panel. This shows that the control-panel is unable to find the camera-view in the library. The main program searches for a variable called recPV, which is a pointer to the camera. This pointer is now located in the library, and the main program is unable to find it. To solve this problem I added a new interface-function called getRecPV(). This should transfer this pointer from the library to the main program. The implementation of this function in the library is as follows:

```
std::vector<Camera*, std::allocator<Camera*>> Camera01_if::getRecPV() {
    return camRecManP->recPV;
}
```

This returns the recPV-pointer to the camera created in the library. In ”viewsman.cpp” I added the two following lines of code in the if-statement to set the recPV-pointer in the main program:

```
camRecManP = cameraCtrlP->getCameraManagerP();
camRecManP->recPV = factory.getPlugins()[index].pluginP->getRecPV();
```

The first line of code gets the camera-manager for the main program. Line 2 sets the recPV-variable in the main program equal to the one created in the library. The camera-view should now be visible to the main program. At this point the advanced control-panel works, but not dynamically linking in new models. Linking in a new model causes the previous models to disappear, and a few random lines are drawn. This can be seen in figure 3.7. Changing the camera-view using the control-panel draws the models correctly again, but the new model is not included. This new model does, however, appear in the advanced control-panel. This shows that the model is linked in to the main program, but the change is not reflected in the library. The current solution transfers the models to the library when it's linked in. This means that we only set the models in the library once, and it's not updated when a new model is added in the main program. This explains why the newly added model doesn't show up in the camera-view. A solution to this problem sounds easy enough: update the models in the library when a new one is linked in. Implementing it is not that simple. Models are linked in through the Database-manager, and the update then has to come from this manager. Finding and updating the correct views from this manager is difficult.

After some trial and error I found another solution. Instead of setting the recPV-pointer in the main program I chose to do it the other way around. I created a new interface-function, setRecPV(), that takes the pointer from the main program and sets it in the library. The function showCameraOnTop() creates the recPV-pointer, so this is now called in ”viewsman.cpp” instead of in the library. This solution shows all the models in the system and the control-panel works. Dynamically linked models also appear with the other models in the view. As a bonus, the previously mentioned control-panel for dynamically linked

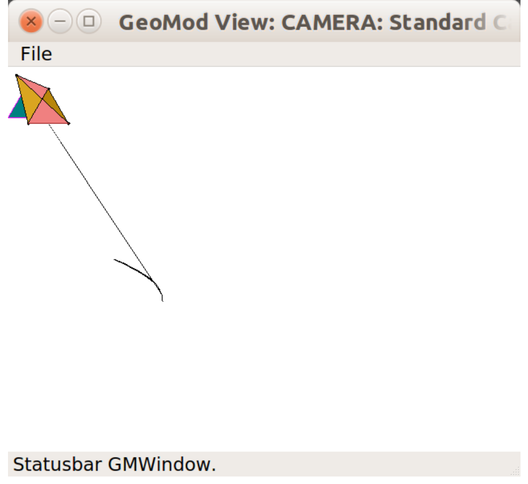


Figure 3.7: Dynamically linked model not working

models works as well. Creating a new camera-view is a bit different in this solution than the one described above. Opening a new view is now done by creating a new instance of the library instead of using the control-panel. This is the same process as creating several models. When discussing the different solutions with my supervisor we found this solution to be the best one so far. This has all the functionality present in the old UNIX-system. The only difference now is the creation of several camera-views. The final code inside the if-statement in the function makePluginInstance() in ”viewsman.cpp” is shown below.

```
camRecManP = cameraCtrlP->getCameraManagerP();
camRecManP->add("Camera " + std::to_string(index));
cameraCtrlP->showCameraOnTop();
factory.getPlugins()[index].pluginP->setCameraRecPV(camRecManP->recPV);
RefreshAllViews::update();
```

The first line gets the camera-manager. Line 2 uses this to add the new camera with an appropriate name. Line 3 calls showCameraOnTop(), which creates the recPV-pointer. This pointer is sent to the library and set through the interface-function setCameraRecPV(). The last line draws the models in the main program in the newly added camera-view.

3.3.5 Linking control-panel as a separate tool

One question that appeared when working on linking in the camera-view was the question of splitting up the view and control-panel. Namely, if the view itself should be linked in through the Views-manager and the control-panel through the Tools-manager. This might have been a good idea when writing the program from scratch. Then we could have made one class for the control-panel and one for the view, and put them in separate libraries. As the code is now, both the view and the control-panel can be created through the same class,

namely the CameraControlWdgt-class. So, for us to link these two parts through different managers we either have to use the same class twice or split it up. The first solution, to use the same class twice, does not make any sense. It could be done, but why split them up when both of them work in the same library? This only makes us do twice the work. The second solution, splitting the CameraControlWdgt-class, could be done. It does, however require us to rewrite both of the classes. For this to work we first have to separate the class, and then make the new ones work together. With this in mind we chose to keep the working implementation as it is.

3.4 Linking in picture-view

I will now try to link in another view called the picture-view. It allows the user to look directly at models in a 2D environment. This view is important in this master's thesis because it allows us to look directly at images. This can also be achieved by the camera-view, but here the view has to be rotated before we can look at the image directly. The picture-view does this automatically, so it will save the user the time of rotating the camera-view.

The picture-view worked on the old UNIX-system, and I got this implementation from my supervisor. This has never been compiled on the new platforms, so I rewrote it using the code for the camera-view as template. I began by comparing the old code for the camera-view with the picture-view. This allowed me to see the differences in the implementations. These were mainly in the way vectors were calculated. The picture-view is designed to show the result in 2D, not 3D, so the transformations are naturally different. Several functions were similar in the old code for camera-view and the picture-view. Here I used the new code for the camera-view to update these functions in the picture-view. The picture-view used a class called Switches. This is not used in the camera-view, and the files defining the class have been removed from the program. Functions using the Switches-class in the picture-view were replaced by similar functions in the camera-view, not using this class. Code without an apparent counterpart in either the old or the new code for the camera-view were kept for now. Which of these functions that have to be updated will be shown when the code is compiled. I added the two resulting files, "view_xy.h" and "view_xy.cpp", to "MaxLib/cmra" and "maxlib.pro". Compiling the code showed quite a few errors. I will not show them here, but most of them were due to the difference in Qt Creator versions. These were solved by changing the include-statements for replaced or moved Qt-modules. Other errors were related to files moved inside the GeoMod-program. These were solved by changing the path of the file to be included. Qt Creator has also changed the way widgets are created since the last time these files were compiled. This meant rewriting the functions that actually creates the widgets. I will not the show the code here, but it can be seen in the files mentioned above. I have commented out the old functions and replaced them by the updated ones. After these changes the MaxLib-library compiles without errors.

The next step is to compile the code for the control-panel. I used the same approach here as with the view. Compare the code for the control-panel with the code for the camera-view

and compile it. The code for the control-panel is in the files ”victrl_xy.h” and victrl_cpp”. They are also added to ”MaxLib/cmra” and to ”maxlib.pro”. Compiling the MaxLib-library showed a few errors similar to the ones above, which were solved by changing the corresponding include-statements. Other errors were caused by the creation of the widget for the control-panel. This used the now deprecated QPopupMenu. I ended up rewriting the control-panel to using QMenu instead of QPopupMenu. When using QMenu, the class has to inherit QMainWindow instead of QWidget. This in turn meant that I had to rewrite all components using code from QWidget. After some rewriting the MaxLib-library eventually compiled without errors.

Now I have to test if the code actually works. I created a library in ”DynamicLinkingTests/08_ViewXY”, which implements the interface-functions and calls the ViewXY-constructor. This should open the view itself and the control-panel. I created a new interface-function called getViewXY() which returns the pointer to the picture-view. This is used in the function makePluginInstance() in ”viewsman.cpp” to get the pointer from the library to the main program. This compiled, but crashed when I tried to link it in. An error-message told me that a function in ”victrl_xy.cpp” used a module called treeViewBasis, which caused the crash. I found the same function in the code for the camera-view. This was commented out with a text telling that it was no longer in use. I commented out the function in the picture-view as well. The error then disappeared, but the program still crashed. This time no error-message was displayed, so the reason was not as easily found. Eventually it turned out that the following line in the function showViewXYWdw() in ”victrl_xy.cpp” caused it:

```
r_MapManager[r_PctNumber].updateLayout();
```

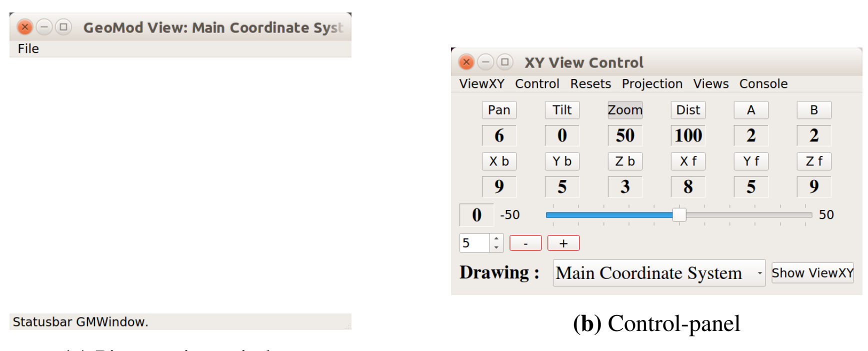
I inspected the function updateLayout() and found out that a function called update() caused it to crash. Here I found the root of the problem, namely the following line:

```
if( !window_Pm->isVisible() ) return;
```

To solve this bug I added the following line in the constructor in ”view_xy.cpp”:

```
window_Pm  = getWindowP();
```

Without this line the if-statement above causes the program to crash, since the variable ”window_Pm” isn’t defined. After adding this line both the picture-view and control-panel opens. It is worth noting that I have added a few lines of code in the Views-manager that allows me to open the view and control-panel. This code will be explained in the next subsection. The view is shown in figure 3.8a and the control-panel in figure 3.8b. The widgets are opened as wanted, but the models are not drawn. This is the same problem as we had for the camera-view, so I will try the same solution in the following subsection.



(a) Picture-view window

(b) Control-panel

Figure 3.8: Dynamically linked picture-view not working

3.4.1 Drawing models in the picture-view

To test that the problem isn't with the picture-view itself, I linked it in statically. The models are now drawn correctly, as can be seen in figure 3.9. The control-panel works as well, which tells me that the picture-view works as intended.

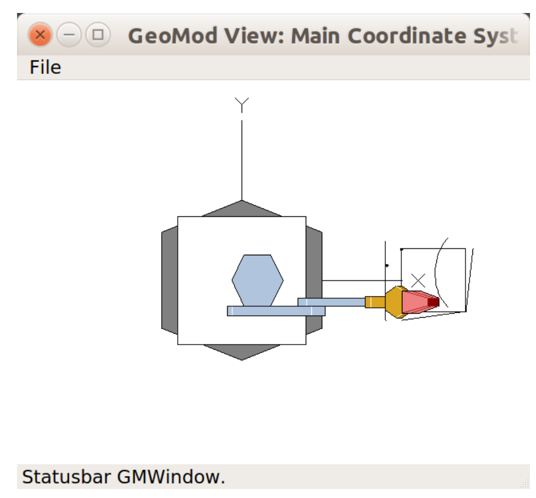


Figure 3.9: Statically linked picture-view

The next step is now to show the models in the dynamically linked picture-view. I began by extending the interface in "plugininterface.h" with a function called setViewXYRecPV(). This is similar to setCameraRecPV() for the camera-view. I cannot use the same function here, because this expects a pointer of type Camera, and we now have the type ViewXY. The implementation of setViewXYRecPV() in the library is as follows:

```
void ViewXY01_if::setViewXYRecPV(std::vector<ViewXY *,
                                  std::allocator<ViewXY *> > recPV) {
    viewxyRecManP->recPV = recPV;
}
```

The picture-view is opened through the ViewXY-manager, which is stored in the variable r_MapManager in "victrl_xy.cpp". r_MapManager is a private variable, so I implemented the following function returning a pointer to the manager:

```
ViewRecManager<ViewXY>* Xy_viControlWidget::getr_MapManagerP() {
  return &r_MapManager;
}
```

This function can be called from "viewsman.cpp" in the same manner as getCameraManager(). The camera-manager contains a function called showCameraOnTop() that opens the camera-view itself. The picture-view has an equivalent function called showViewXYWdw(), but this is private in "victrl_xy.cpp". I created a new, public function called showViewXYOnTop() that calls showViewXYWdw(). This new function is then called in makePluginInstance in "viewsman.cpp". The full code for dynamically adding both the camera- and picture-view is shown below. We see that the code for adding the picture-view is similar to adding the camera-view, but with functions called from different libraries.

```
int index = factory.getPlugins().size() - 1;
cameraCtrlP = factory.getPlugins()[index].pluginP->getCamera();
viewxyCtrlP = factory.getPlugins()[index].pluginP->getViewXY();
if(cameraCtrlP != nullptr) {
    camRecManP = cameraCtrlP->getCameraManagerP();
    camRecManP->add("Camera " + std::to_string(index));
    cameraCtrlP->showCameraOnTop();
    factory.getPlugins()[index].pluginP->
                            setCameraRecPV(camRecManP->recPV);
    RefreshAllViews::update();
}
else if(viewxyCtrlP != nullptr) {
    viewXYRecManP = viewxyCtrlP->getr_MapManagerP();
    viewXYRecManP->add("View" + std::to_string(index));
    viewxyCtrlP->showViewXYOnTop();
    factory.getPlugins()[index].pluginP->
                            setViewXYRecPV(viewXYRecManP->recPV);
    RefreshAllViews::update();
}
else {
    std::cout << "The provided library is either buggy or not a camera- "
                 "view. Have you checked that all interface functions are "
                 "implemented?" << std::endl;
}
```

Linking in the picture-view dynamically now works the same as when linked in statically. The models are drawn as in figure 3.9 and the control-panel works. The next step is then

to link in more than one picture-view. Here we have the same problem as with the camera-view described earlier. The code only allows for one instance of the picture-view. I will use the same approach here as well, so the code will not be shown. The idea is to store the view-pointers in a vector instead of in a single variable. Each new view is then added to this vector. Some of the code in "victrl_xy.cpp" was written to handle a single pointer. This had to be updated to handle a vector of pointers. One of these functions was updateAllLables(). Earlier this updated all labels in the single view. This now loops through the vector and updates all of the views. These changes now allow users to add as many picture-views as wanted.

Each time the control-panel used to move models is clicked, it calls the function RefreshAllViews::update() in "all_views.h". This function updates the camera-view to show the change done by the control-panel. The picture-view is currently not updated when a model is moved. RefreshAllViews::update() calls the function AllViews::update() in "allviews.h", which again calls the function update() in "viewrec.h". This function decides which views to update. Previously this function contained the one line:

```
ViewRecManager< Camera >::update();
```

This updates the currently opened camera-views. We now want to update the picture-view as well, so I added the following line:

```
ViewRecManager< ViewXY >::update();
```

After adding this line, the picture-view works in exactly the same manner as the camera-view. We are now able to add several instances, move the view-point, add models and move them using a different control-panel. I also tried opening both the camera-view and picture-view, and moved the models around. This showed the models moving in both views at the same time.

During the process I have added quite a few interface-functions. Some of these are no longer in use, for instance updateCamera() and getRecPV(), and have been commented out. I have chosen not to remove them completely in case they are needed at a later stage. The interface grows quickly, as can be seen in the fact that I have added almost half of the functions in my master's project and master's thesis. We want to keep the interface small so that it easily can be implemented. I have during this thesis regularly looked through all the libraries, and updated them according to the current interface. This is not a difficult task, but it takes some time for persons not familiar with it. It's natural for me to do this since I have worked a lot with the interface and developed my share of the libraries. One thing to note here are the functions getModel(), getCamera() and getViewXY(). Each library will only use one of these, since they contain one model or view. The other functions are not used, but the library still has to implement them since they implement the interface. These functions then return a nullpointer to meet this criteria. This also helps in controlling the managers. Trying to link in a view in the Database-manager will return a nullpointer, and an error-message will be displayed.

3.4.2 Linking in letters in the picture-view

Now I want to test if the transformations in the picture-view are correct by creating libraries containing 2D-models of the letters A and B. An old implementation of the letters, working on UNIX, was given to me by my supervisor. I began by creating a new library called ”11_AUpper” with code implementing the interface. I then added the old code for the letter A and called its constructor. Here some include-statements related to moved files and updated Qt-modules had to be changed.

The old code contained an implementation of a control-panel for the letter in the files ”move_A01.h” and ”move_A01.cpp”. This control-panel also uses the deprecated QPopupMenu, so it has to be rewritten. When discussing it with my supervisor, we concluded that spending time on rewriting an old control-panel that might not work isn't the best use of time. Instead we decided to use a control-panel from one of the newer libraries that we know works with the current version of Qt Creator. I then commented out all the code for the old control-panel and replaced it with the control-panel from a library containing a cube. This library was created in my master's project, and we know that it works. After renaming classes, functions and variables the library compiled without errors. Linking in this library and opening the picture-view is shown in figure 3.10. The control-panel shows up, and allow us to move the model.

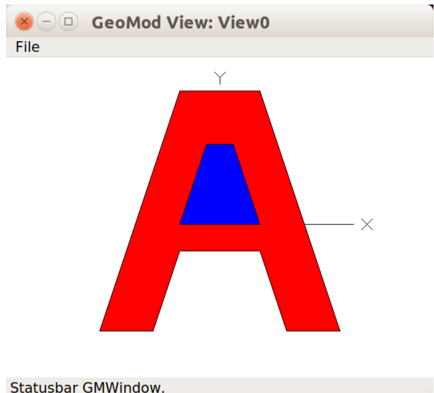
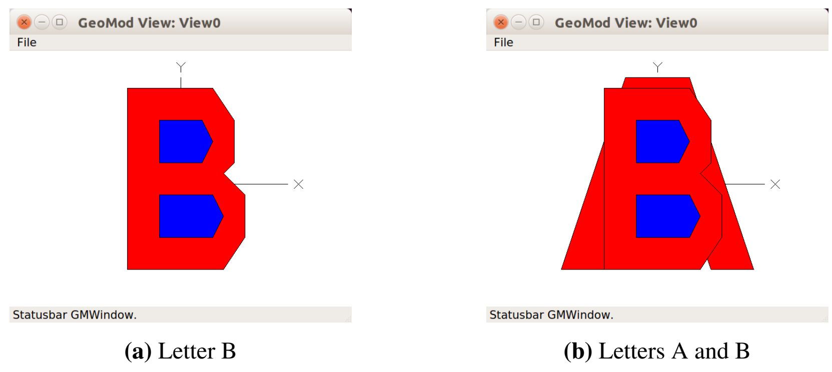


Figure 3.10: Letter A dynamically linked in picture-view

Later in my thesis I want to display images in the picture-view. It is then desirable that the images are displayed with the correct orientation. With orientation I mean that the top of the picture is on the top of the view and the right of the picture being on the right of the view. For this to happen the transformations have to be correct. We can see in figure 3.10 that the top of the model is depicted correctly. The letter A is symmetrical about the y-axis, which means that we cannot determine if the right of the letter actually is shown on the right in the view. The case might be that the letter is rotated 180 degrees about the y-axis, and that the back of the model is shown. To test this I will create a library containing a model of the letter B as well. This isn't symmetrical about the y-axis, so these two models



(a) Letter B

(b) Letters A and B

Figure 3.11: Dynamically linked letters in picture-view

combined will verify if the transformations are correct.

This library was created in the same manner as the one above. Create a new library which implements the interface, copy the old code for the letter and add code for a functional control-panel. This library can be found in ”DynamicLinkingTests/12_BUpper”. The result of compiling and linking it in through the Database-manager is shown in figure 3.11a. The letter is drawn with the correct orientation and a working control-panel. The result of linking in both the A and B can be seen in figure 3.11b. We see that both are drawn with the correct orientation, which means that the transformations are correct.

I have in this chapter dynamically linked in two views that work in the same manner as they did in the old UNIX-system. The old files contain other views that it is desirable to link in as well. This task is not the main focus of this master's thesis, so the implementation of the other views have been left for further students. The libraries created here shows how the old code can be rewritten and linked in. These can be used as templates when the need arises for more views to be linked in.

Create models from binary images

The previous chapter described the process of dynamically linking in the camera- and picture-view. This section will use these views to show the results of creating binary models from images. In an earlier version of the code on UNIX, an algorithm called BinPic was used to create binary geometric models from images. The algorithm creates wireframes filled with black areas where the input-image is darker than a given threshold. This model could then be shown in both the camera- and picture-view. I will in this chapter re-implement this algorithm in a library that can be linked in as a tool. The code has to be adjusted to work on the new platforms, as well as with the current version of Qt Creator. The resulting model should be added to the vector of models in the system and drawn in the visible views. I will then discuss the different image-formats the algorithm can handle.

Before starting the work with the algorithm, the Tools-manager has to be rewritten to allow for tools to be linked in. Here a test-library will be created to check that the dynamic linking works as intended.

4.1 Dynamically linking tools

The Tools-manager is now implemented exactly like the Database-manager, as explained at the beginning of the previous chapter. This means that it now can be used to link in models. We now want to link in tools, so the implementation in makePluginInstance() in ”toolsman.cpp” is changed to:

```
int index = factory.getPlugins().size() - 1;
QWidget* widget = factory.getPlugins()[index].pluginP->getTool();
if(widget != nullptr) {
    widget->show();
}
else {
    std::cout << "The provided library is either buggy or not a tool."
                 "Have you checked that all interface functions are "
                 "implemented?" << std::endl;
}
```

This code is similar to the Database- and Views-manager, but it allows us to import a tool instead of models or views. A new interface-function, getTool(), is used to get the pointer to the tool from the library to the main program. This function expects a pointer of type QWidget. The tool itself will be called through a button on this widget. This allows us to keep the code in the Tools-manager, and the interface, as simple as possible. Starting each tool directly would mean creating an interface-function for each tool, and determining the type in the Tools-manager. By wrapping it in a widget we can show this widget directly, and open the tool from this.

I have created a test-library in ”DynamicLinkingTests/09_TestTool” to see if the code above works as intended. This library contains an implementation of the interface-functions as well as code for creating a QWidget with a certain geometry and a button. It should only tell us if the Tools-manager works, so it doesn't contain any functionality. The resulting widget created when the library is linked in is shown in figure 4.1. The widget is shown as expected, which tells us that the dynamic linking works.

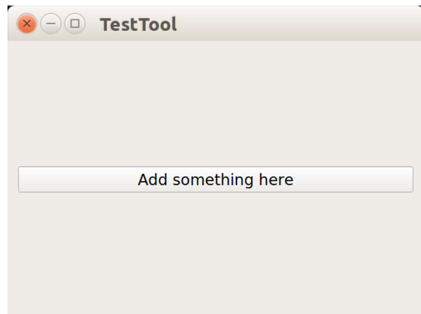


Figure 4.1: Widget shown from dynamically liked test-tool

4.1.1 File-browser for finding images

I will in this section create a library containing the BinPic-algorithm in ”DynamicLinkingTests/10_BinPicTool”. But before I add the algorithm itself I want to build up a widget that can utilize it. First, I want to add a button that opens a file-browser when clicked.

This browser should be used to locate the images we want to run the algorithm on. The BinPic-tool will now look like figure 4.2. Pressing the button ”Add Image” opens a file-browser similar to the ones in the Database-, Views- and Tools-manager. This file-browser can be seen in figure 4.3. It allows the user to search through local files on the computer and select png-images. In the code for the file-browser, a variable called fileFilters tells the code which files to show in the results-column on the right. For now this variable only includes png-images, so the browser will only show files with the png-extension. Other extensions will here be added when needed during the testing of the algorithm. In figure 4.3 we see that only the png-images in the current folder is shown. I will not show the code for the widget and file-browser here. It is similar to other code in the system, and can be found in ”binpictool.cpp” and ”binpicbrowser.cpp”.

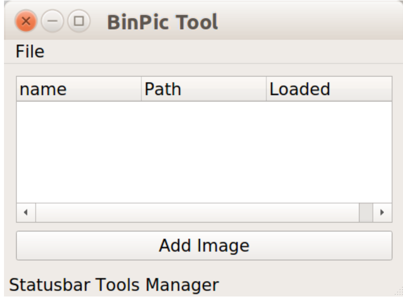


Figure 4.2: First widget that opens from the 10_BinpicTool-library

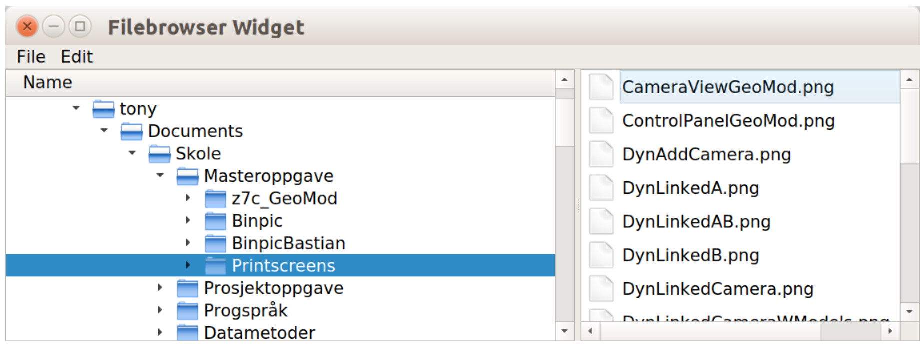


Figure 4.3: Filebrowser with .png-images

When an image is double-clicked in the file-browser it should be added to the BinpicTool-widget shown in figure 4.2. This functionality is implemented in the function addLibrary() in ”binpictool.cpp”:

```
void BinPicTool::addLibrary(QString pathToPlugin) {
    QStringList path = pathToPlugin.split("/");
    QString fileName = path.at(path.size()-1);
    QTreeWidgetItem *widgetItem = new QTreeWidgetItem();
    widgetItem->setText(0, fileName);
    widgetItem->setText(1, pathToPlugin);
    treeWidget->addTopLevelItem(widgetItem);
}
```

The first line in the function splits the input-string into a QStringList [12] and stores it in the variable path. A QStringList is a list of strings containing all parts of the path separated by ”/”. The file-name itself is found in line 3 by getting the last entry in the list of strings. Line 4 creates a new QTreeWidgetItem, and lines 5 and 6 adds the file-name to the first column and the full path to the second column in Figure 4.2. The third column is left empty for now, and should be set equal to ”yes” when a preview of the file has been opened. Line 7 adds the QTreeWidgetItem to the widget. The png-files are now added to the BinPicTool-widget. An example where three images have been added to the file-browser is shown in figure 4.4. I have also added a button called ”Show Preview” that will be used to show a preview of the selected image.

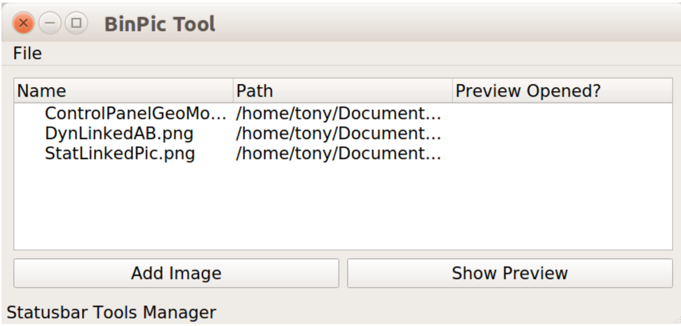


Figure 4.4: Window with png-images

4.1.2 Previewing .png-files

The next step is to connect the button ”Show Preview” to a function that opens a preview of the selected image. This allows the user to preview the images to verify that the correct image has been added. The function opening a preview, openShowPreview(), in ”binpic-tool.cpp” is implemented as follows:

```cpp
void BinPicTool::openShowPreview() {
    int columnCheck = treeWidget->currentColumn();

    if(columnCheck != -1) {
        QString pluginPath = treeWidget->currentItem()->text(1);
        QWidget *frame = new QWidget();
        frame->setGeometry(740, 450, 500, 320);
        frame->setWindowTitle("Image preview");
        frame->setStyleSheet("image: url(" + pluginPath + ")");
        frame->show();
        treeWidget->currentItem()->setText(2, "Yes");
    }
}
```

Line 2 in the function above returns 1 if a column is selected in the BinPicTool-widget. The if-statement in line 4 will be entered if an image has been selected. If no images are selected, line 2 returns -1, and nothing happens when the button is pressed. Inside the if-statement the path to the image is stored in the variable pluginPath in line 5. Line 6 creates a new QWidget which will contain a preview of the selected image. The geometry and title of this widget is set in lines 7 and 8. Line 9 adds the image with the function setStyleSheet(). This finds and sets the image using the path stored in pluginPath. The widget is shown on screen in line 10. The last line in the if-statement sets the column "Preview Opened" in figure 4.4 equal to "Yes", indicating that a preview has been opened. The result of adding a png-image of Super Mario [3] and pressing "Show Preview" is shown in figure 4.5.

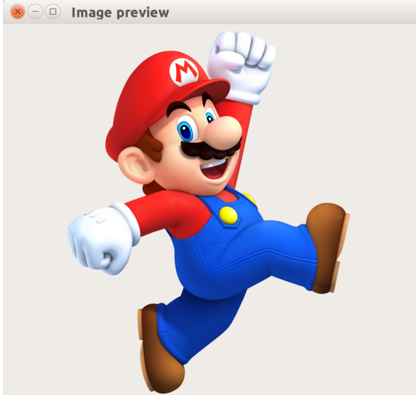


Figure 4.5: Widget with a png-image of Super Mario [3]

The code works as expected, and the selected image is shown in a new widget. Next I want to test if jpg-files can be previewed as well. Before they can be shown I have to make sure that the file-browser can find them. The fileFilters-variable described earlier contains all formats the browser can find. Adding the following line in "binpicbrowser.cpp" allows the browser to find jpg-files as well:

```
fileFilters << "*.jpg";
```

Both jpg- and png-files are now displayed in the file-browser. Figure 4.6 shows a preview of a jpg-image of a little kitten using the code above. This shows that it is able to preview jpg-images as well as png-images. I will go back and test other formats when I know which formats the BinPic-algorithm supports.

Figure 4.6: Widget with kitten.jpg-image

4.2 Adding the BinPic-algorithm to the library

I mentioned earlier that the code for the BinPic-algorithm worked on the UNIX-platform. I will now test to see if it still works after the migration to the new platforms. Before adding the code I talked with my supervisor. He told me that the user should be able to apply the algorithm on selected images. The library created in the previous section contains a files-browser, so I will continue with this library, and apply the algorithm on the selected image. I added the code for the algorithm to the library and included them in ”binpictool.pro”.

4.2.1 Compiling the BinPic-algorithm

Compiling the code resulted in a few errors regarding include-statements. These were mostly due to new Qt-modules and files being moved since the implementation on UNIX. After fixing these statements the code compiled without errors. The code is now compiled with the library, but never used. The next step is to actually use the code to create binary geometric models. I created a new button called ”Create Binary Picture” in the BinPicTool-widget shown in figure 4.4. When pressed, the function connected to this button calls the BinPic-constructor with the image selected. Compiling the code still worked, but running the algorithm caused the program to crash. It turned out that the algorithm

cannot be run with a color-image as input. This means that the input-image has to be converted before the algorithm is called. Looking through the old UNIX-code I found the function calling the BinPic-constructor in a file called ”picctrl.cpp”. This opens a widget which allows the user to choose if the input-image is part of a larger image, or if it's the whole image to be processed. Then it converts the image before calling the BinPic-constructor with it. I added this code to ”binpictool.cpp” before the call to the constructor. This code works on an outdated version of Qt Creator, so some changes had to be made for the code to compile. One of these changes were the conversion of the input-image. Earlier the code was as follows:

```
1 QImage image = QImage( (const char *)f ).convertDepth( 1 ).
2                                convertBitOrder( QImage::LittleEndian );
```

This was changed to:

```
1 QImage image = QImage( f.toStdString().c_str() ).
2                                convertToFormat( QImage::Format_RGBA8888 );
```

First, I had to change the conversion of the path to the input-image. The casting (const char*)f was allowed in an earlier version of C++, but it has now been replaced by function-calls. I first have to convert the std::string to a QString using the function toStdString(). Using the function c_str() on this result gives me the path on the same format as the casting above. Next, I had to change the code for the conversion of the input-image. In the old code the functions convertDepth() and convertBitOrder() converted the input-image to a format the BinPic-algorithm could handle. Both of these functions have been discontinued by Qt Creator and replaced by the function convertToFormat(). The function for the format, QImage::LittleEndian, has also been replaced by QImage::RBGA8888 [13]. After applying these changes the code compiled without errors. Linking in the library, selecting an input-image, and pressing the button ”Convert Binary Picture”, opens the widget shown in figure 4.7. Pressing ”Complete” makes this widget disappear, but nothing happens. At this point the newly created picture-model should have appeared in the camera-view.

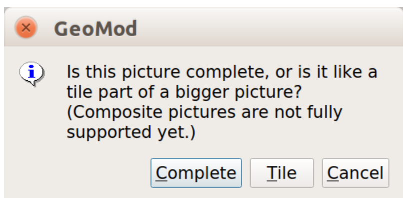


Figure 4.7: Panel for choosing if the image is complete

4.2.2 Troubleshooting BinPic

The code in the library now compiles and runs, but without the desired results. Without any error-messages, finding the reason might take some time. At this point I was lucky

and got old exercises and notes from my supervisor regarding the last implementation of the algorithm. The exercises and notes are from an old PhD-subject discontinued several years ago. They allowed me to get a better understanding of the algorithm, its implementation and the expected input and output. Comparing this with the current solution showed some differences regarding the input-image. In the old code the function convertDepth() converted the input to a binary image. convertDepth(1) sets the bit-order to 1, meaning the image is represented using two colors, namely black and white. In the current solution the bit order is never set explicitly, so I want to see if the input-image actually is binary. To test this, I added the following lines of code in ”binpictool.cpp” directly after the image has been converted.

```
QWidget *testImg = new QWidget();
testImg->setGeometry(740, 450, 500, 320);
QLabel *myLabel = new QLabel(testImg);
myLabel->setPixmap(QPixmap::fromImage(image));
img->show();
```

The first two lines creates a new widget and sets its geometry. Line 3 creates a label contained within the widget. This is done by adding the widget to the QLabel-constructor. The image is shown inside the label by adding it through the function setPixMap(). The last line shows the widget containing the image. The resulting widget opened when running the code shows me that the image still is stored in colors. This means that the function convertToFormat() described above only converts the image to a given format, and not into a binary one. Searching through the documentation in [13] I found that passing the argument ”FORMAT_Mono” converts the image to grey-scale. I found no functionality converting it directly to binary, so I will see if the BinPic-algorithm handles grey-scaled images. The BinPic-algorithm expects the input-image to be on the LittleEndian-format. Therefore, I ended up with the following code:

```
QImage image = QImage( f.toStdString().c_str() ).
        convertToFormat( QImage::Format_Mono ).
        convertToFormat( QImage::Format_RGBA8888 );
```

The function convertToFormat() is now called twice. Once to convert the image to grey-scale, and once to convert it to the correct format. The widget now contains a grey-scaled image. The result of using a screen-shot of the letter A as input can be seen in figure 4.8.

The input-image is grey-scaled and on the correct format, but still nothing happens when the BinPic-algorithm is run. The last re-implementation of the algorithm was in 2003. It has only been shown to work on UNIX, but no error-messages are shown, so the problem might lie in other parts of the code. To test if the problem is within the BinPic-algorithm or in other parts of the code, I manually created a new model inside the BinPic-library. I will not show the code for the model here, but it’s based on other models linked in through the Database-manager, so it should be drawn correctly. The constructor for this model will be called instead of the function buildModel() that creates the binary geometric model. If the model is drawn, we know that the problem is contained within the code for

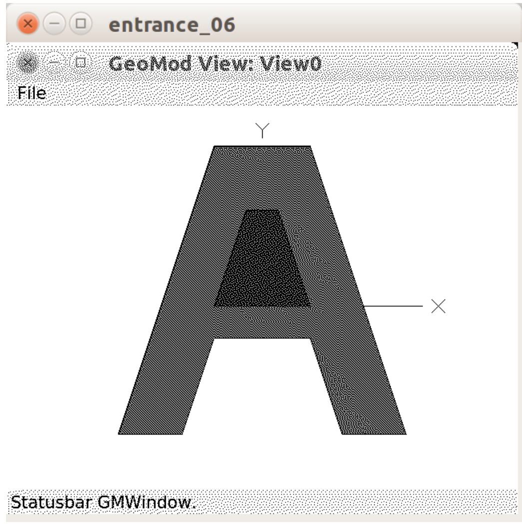


Figure 4.8: Preview of the grey-scaled image A

the BinPic-algorithm. I created a geometric model of a rhombus in the files ”testrhombus.h” and ”tesrhombus.cpp” to separate its code from the rest of the library. These were included in ”binpictool.pro” and compiled with the library. Compiling and running the code shows no errors, but the model is not drawn in the camera-view. I then tried calling the function RefreshAllViews::update(rhombus). This function should add the rhombus to the list of models and draw it with the others. Again nothing is drawn, but I finally got an error. This error was the result of the library not knowing about the views in the main program. Updating the views causes an error, because the library cannot find them. To the library no views are currently opened. I then tried telling the library about the view with the code shown below. This code is inside the if-statement in makePluginInstance() in ”toolsman.cpp”. The rest of the code is unchanged, so it will not be shown here.

```
1 ViewRecManager<Camera> viewrecP;
2 ViewRecManager<ViewXY> viewXYrecP;
3 factory.getPlugins()[index].pluginP->setCameraRecPV(viewrecP.recPV);
4 factory.getPlugins()[index].pluginP->setViewXYRecPV(viewXYrecP.recPV);
5 widget->show();
```

This code uses the same interface-functions, setCameraRecPV() and setViewXYRecPV(), described earlier when linking in the views. Lines 1 and 2 create variables for the current camera- and picture-view, and lines 3 and 4 set these pointers in the library through the interface-functions. The library should now know about the visible views and be able to update them. Running the code again yields the result in figure 4.9. We see that all the models disappear, and a small plane is drawn in top-left corner. Unfortunately, this is not the rhombus. The model of the rhombus is a lot bigger and positioned at the center of the view. It is reasonable that the other models disappear, since the library has no idea about

the models in the main program, but the rhombus should still be drawn correctly.

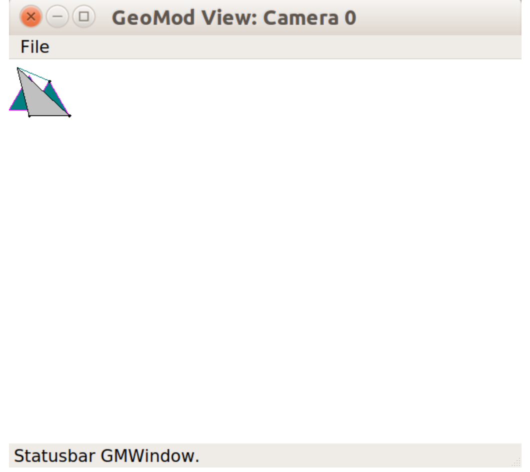


Figure 4.9: The camera view when drawing the rhombus

To make sure that the problem isn't with the model of the rhombus I instantiated a camera-view inside the BinPic-library. The code for adding the statically linked camera-view is shown below. It's added in the function contructBinaryPicture() in "binpictool.cpp". The code creates the camera-view, shows the view itself and a control-panel. I also added the function buildModel() to see if the binary model is created by the BinPic-algorithm. This result is seen in figure 4.10. We see that the rhombus is drawn correctly in the middle of the view. A black surface has also appeared in the xy-plane. This is the model created by the BinPic-algorithm. It's not correct, but it shows us that the algorithm creates a geometric model. The statically linked camera-view allows me to see the geometric model created by the BinPic-algorithm. I will therefore continue the work on the algorithm with this view, and come back to the problem of drawing the models in a dynamically linked view later.

```
1 CameraControlWdgt *cmra = new CameraControlWdgt(nullptr);
2 cmra->showCameraOnTop();
3 cmra->showOnTop();
```

4.2.3 Small changes after discussions with my supervisor

At this stage I discussed the code developed so far with my supervisor, and a few changes were made. First, the file-name and extension was added to the widgets opened when previewing images. Users might have several previews opened, and adding the file-name with extension in the title makes it easier to see which image it's created from. The function openShowPreview() is called when the button "Show Preview" is pressed. After adding

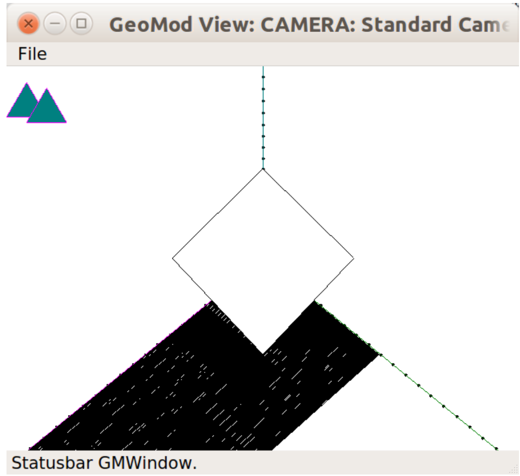


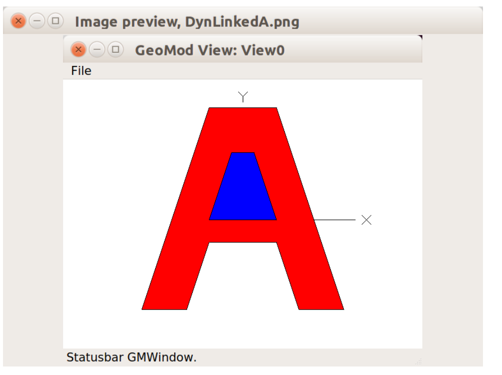
Figure 4.10: Image of the statically linked view with rhombus

file-name and extension, the implementation of this function is as follows:

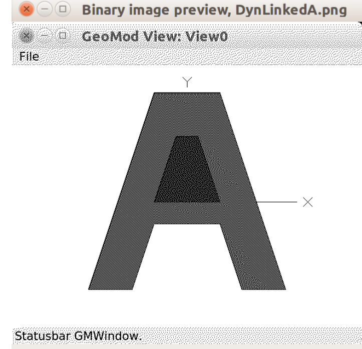
```
1  void BinPicTool::openShowPreview() {
2      int columnCheck = treeWidget->currentColumn();
3
4      if(columnCheck != -1) {
5          QString pluginPath = treeWidget->currentItem()->text(1);
6          QString fileName = treeWidget->currentItem()->text(0);
7          QWidget *frame = new QWidget();
8          frame->setGeometry(740, 450, 500, 320);
9          frame->setWindowTitle("Image Preview, " + fileName);
10         frame->setStyleSheet("image: url(" + pluginPath + ")");
11         frame->show();
12         treeWidget->currentItem()->setText(2, "Yes");
13     }
14 }
```

This code is similar to the one described earlier, except for the two lines 6 and 9. Line 6 get the name of the current image, and line 9 sets the title of the window to ”Image Preview, ” + file-name with extension. The name of the image is now displayed in the title, as can be seen in Figure 4.11a.

Next my supervisor wanted me to add a button on the BinPicTool-widget that lets the user preview grey-scaled images. I added a new button called ”Show Binary Picture”. When clicked, this calls the function openShowBinaryImage() with the implementation below.



(a) Result of previewing A after name is added



(b) Result of previewing A as a grey-scaled image

Figure 4.11: Updated previews

```
1  void BinPicTool::openShowBinaryImage() {
2      int columnCheck = treeWidget->currentColumn();
3
4      if(columnCheck != -1) {
5          QString pluginPath = treeWidget->currentItem()->text(1);
6          QString fileName = treeWidget->currentItem()->text(0);
7          QWidget *frame = new QWidget();
8          frame->setGeometry(740, 450, 500, 320);
9          frame->setWindowTitle("Binary image preview, " + fileName);
10         QImage image = QImage(pluginPath.toStdString().c_str()).
11                 convertToFormat(QImage::Format_Mono).
12                 convertToFormat(QImage::Format_RGBA8888);
13         QLabel *myLabel = new QLabel(frame);
14         myLabel->setPixmap(QPixmap::fromImage(image));
15         frame->show();
16     }
17 }
```

This code is similar to the implementation of openShowPreview() above. The main difference between the two are the lines 10 - 14. Lines 10 - 12 create a new image that is converted to grey-scale and the ”LittleEndian”-format. Line 13 creates a label within the widget to be shown, and line 14 adds the grey-scaled image to this label. Line 9 has been changed to tell the user that the widget previews a binary image. The title will now be displayed on the following form: ”Binary image preview, ” + file-name with extension. The resulting widget using the same image as in figure 4.8 is shown in figure 4.11b.

My supervisor also wanted me to change the name of two buttons. ”Create Binary Picture” now says ”Create Picture Model”, and ”Add Image” says ”Add Picture”. The BinPicTool-widget now looks like figure 4.12.

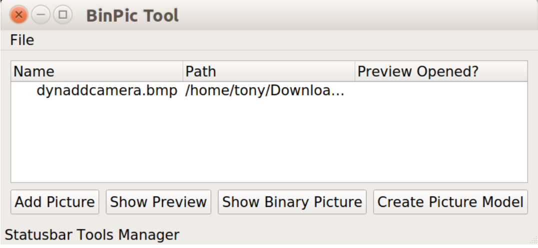


Figure 4.12: The BinPic-tool with new button and button-names

4.2.4 Further development on the BinPic-algorithm

After applying the changes above I turned my attention back to the BinPic-algorithm. The result in the statically linked camera-view so far is a black surface in the xy-plane as shown in figure 4.10. When discussing the current solution with my supervisor he told me that the input-image had to be binary, not grey-scaled. The functions convertDepth() and convertBitOrder() in the old code were not used to convert the image to binary. The image was then already on binary form, and the functions were used to convert the image to a format the algorithm could handle. In older versions of the code a third-party software was used to convert the images before they were sent to the algorithm. I want to let the user skip the step of converting the image manually. After searching online I found the class QBitmap [14]. This takes a colored image as input and returns a binary version of it. The QBitmap-constructor takes in a QString containing the path to the image as input. Calling the function toImage() on this result returns a QImage. The previous code for converting the image now becomes:

```
QBitmap bMap = QBitmap(f);
QImage image = bMap.toImage();
```

The result stored in the variable image is now a binary image. The image was earlier stored in the ”LittleEndian”-format, but this conversion has been removed. For the BinPic-algorithm to be called with this new image as input, the following if-statement had to be changed:

```
if( image.format() == QImage::Format_RGBA8888 ) {
    // Image processing code
}
```

This checks if the input-image is on the ”LittleEndian”-format. Since it no longer is, I changed the if-statement to:

```
if( image.depth() == 1 ) {
    // Image processing code
}
```

The new condition, image.depth() == 1, checks if the image is stored using two colors,

namely black and white. The image is now binary, so this condition will be true. Running the code on the image in figure 4.11a now gives us the result shown in figure 4.13. We see that the binary geometric model is created, and the letter A is shown in black. We also see two black stripes at the top and bottom of the model. These come from the geometric model being created from a screen-shot. The screen-shot contains darker areas at the top and bottom which is shown in black in the resulting model.

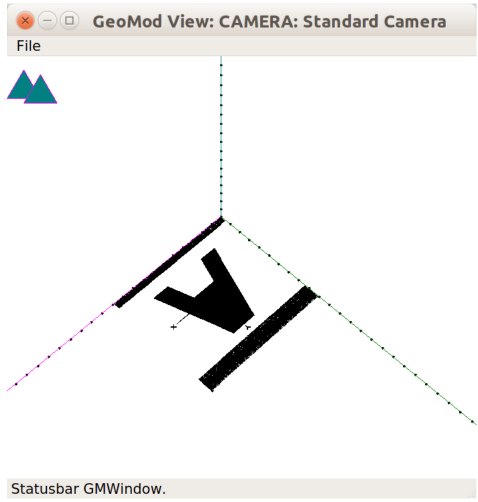


Figure 4.13: BinPic-model of A in the camera-view

The binary geometric model is now created correctly. This shows that the code works without the use of a third-party software for converting the input-images. This saves the user from downloading and using such a third-party software when a binary model is needed.

4.2.5 BinPic-algorithm with different extensions

Images can be stored in a variety of different formats, and I want to test the BinPic-algorithm on some of these. In the notes and exercises mentioned earlier regarding the implementation of the BinPic-algorithm, a section is written about the format of the input-image. It is here written that the old code only handles images stored on the BMP- and BMX-format. These formats are not widely used today, and the reader might not be familiar with them. I have therefore written a short explanation in Appendix B. The image used to obtain the result in figure 4.13 is stored as a png-image. This shows that the current code handles more than the two formats mentioned above. I have not changed the implementation of the BinPic-algorithm itself, but rather how the input-images are handled before it's called. The algorithm was developed to work on bitmap-images. BMP- and BMX-files are stored as bitmap-images, so the algorithm handles these directly. In the old code the images were stored directly in a QImage-variable and sent to the constructor. This means that all images had to be on the bitmap-format before being used as input. In the new

code the QBitmap-class stores the resulting image in the bitmap-format. It means that the input-image only has to be on a format this class can handle.

Our code should handle images already on the bitmap-format, and others that can be converted by the QBitmap-class. To test this I imported the same image as earlier, but in different formats. I began by converting it to BMP, BMX, gif and xpm. This conversion is straight forward on Ubuntu using the terminal. The line ”convert DynLinkedA.png DynLinkedA.extension” converts the image to the selected format by replacing ”extension” with the format wanted. This tool is built-in on both Linux and OS X. On Windows a tool called ImageMagic [15] is available. It can be downloaded from [16]. This allows for the same conversion on Windows with the following command-line: ”magic DynLinkedA.png DynLinkedA.extension”. I have also converted images using a free online-converter. All of the formats mentioned above, both when converted using the terminal and the online-converter, show the same result as in figure 4.13. The resulting geometric models are equal, so the format of the input-image doesn’t seem to affect the quality of the result.

Some old files in the project are stored on the eps-format, so I tried this as well. This format is not supported by the algorithm, and the widget in figure 4.14 is shown. eps is a vector-based format, which means that the QBitmap-class is unable to convert them into bitmap-images. The result will then be stored as a null-image, and the error-message displayed.

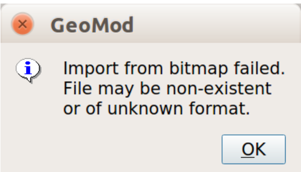


Figure 4.14: Error-message when importing eps-files

One of the most popular formats, jpg, was also tried. jpg-images are stored in a different manner than bitmap-images [17]. They are stored in a compressed, vector-based format. This allows for good quality when the image is compressed, but it means that the QBitmap-class cannot convert them. This causes the program to crash without any error-messages. To prevent the program from terminating I added the following lines of code, which opens an error-message when jpg-images are imported.

```
1 QString extension = treeWidget->currentItem()->text(0).split(".").at(1);
2 QImage image;
3 if( extension == "jpg" ) {
4     image;
5 } else {
6     QBitmap bMap = QBitmap(f);
7     image = bMap.toImage();
8 }
```

Line 1 gets the file-name of the current image, splits it on "." and stores the second item in the variable extension. Line 2 creates a variable called image, which by default is a null-image. The if-statement in line 3 checks if the extension of the current image is "jpg". If this is the case nothing happens, and the image will remain a null-image. This will in turn open the widget in figure 4.14. If the image is on another format, the else-statement will create the binary image as before.

I have verified that the code handles BMP- BMX-, png-, gif- and xmp-formats, while the eps- and jpg-formats displays an error-message. Should other formats be tested at a later stage, it can simply be done by adding the extension to the fileFilters-variable in the constructor in "binpicbrowser.cpp" and by running the code.

4.2.6 Drawing the models in the current view

The current code works in the statically linked view, but we want to create the binary geometric model in the dynamically linked view with the other models in the system. After debugging the program I have a good understanding of the problem. I will describe the problem itself in this subsection before proposing a solution in the next.

The first step is to understand how models linked in through the Database-manager are drawn in the current view, and then relate this to the problem at hand. The Database-manager imports models written in a pre-compiled library, and accesses them through the interface-function getModel(). The model-pointer is then passed to the function RefreshAllViews::update(model). This adds the pointer to the vector of models in the main program. The drawing-algorithm draws the models that are stored in this vector. In short, this means that the model is brought from the library and added with the other models in the program. With the model from the library stored in the same place as the statically linked models, all of them are drawn.

One could then think that this approach would work on this problem as well, but unfortunately it doesn't. With the same approach, the functions above would be called in the Tools-manager. This code is executed when the library is linked in, which is before the geometric model has been created. In the BinPicTool the model is created when the BinPic-algorithm is executed on an input-image, and not when the library is linked in. This means the approach above will try to add the model before it has been created, and the program crashes.

The reasonable step would then be to call the function RefreshAllViews::update(model) when we know that the model has been created. Implementing and running this gives us the same result as in figure 4.9. The library updates the correct view, but the result is not correct. The models in the main program disappear, and the added model is not drawn. The current code allows the library to update the correct view, but it doesn't know about the models already present in it. To let the library know about these models I called the previously defined interface-function setModels() in makePluginInstance() in "toolsman.cpp". This allows the library to see the models in the system, so they shouldn't disappear when

the model created by the library is added. This implementation draws our new model with the ones already in the system, but it still has bugs. Updating the view in the main program now causes the newly added model to disappear. This happens because the new model is added to the list of models in the library, but not in the main program. Updating the view in the main program only draws the models known here. The new model is not a part of these, so it is not drawn. This solution transfers the models from the main program to the library, but we want it to be the other way around by sending the newly created model to the main program.

4.2.7 Solution to drawing the models in the current view

At this point my supervisor told me that the other students on the same project had similar problems with their libraries. We then decided to work on a solution together rather than developing one each. We started by discussing the solutions we had tried and the reasons why they didn't work. After a few tries we came up with the following solution. We added a new button called ”Update Models” in the Tools-manager. This button should be pressed after the model has been created in the library. It then adds this model to the main program using the function RefreshAllViews::update(model). This function is now called in the main program after the model has been created, which solves the problems outlined in the previous chapter. The button is connected to the function updateModels() in ”toolsman.cpp” with the following implementation:

```
1  void ToolsManagerWidget::updateModels() {
2      QTreeWidgetItem *item = treeWidget->currentItem();
3
4      if(item) {
5          if(item->childCount() == 0) {
6              item = item->parent();
7          }
8          int index = treeWidget->indexOfTopLevelItem(item);
9          if(index != -1) {
10             Models::ModelData *model = factory.getPlugins()[index].
11                                        pluginP->getModel();
12
13             if(model != nullptr) {
14                 RefreshAllViews::update(model);
15             }
16         }
17     }
18 }
```

When called, this function gets the currently selected item and adds it to a variable called item. If no items are selected, nothing happens. Otherwise it checks if the item has a child-item. If this is the case, the variable item is updated to contain the child-element. Line 8 gets the index of the selected item in the plugins-vector. The function indexOfTopLevelItem(item) is used to obtain this index, and the result is stored in a variable called index. If this index is -1, meaning that the item isn't part of the plugins-vector, nothing

happens. Otherwise the interface-function getModel() is used to return a pointer to the model created in the library. If this function returns a nullpointer it means that the model hasn't been created yet, and nothing happens. If the model is created, the function RefreshAllViews::update(model) adds this model to the vector of models in the system and updates the current views.

In practice, the button with the implementation above does the following when pressed: If no library is selected in the Tools-manager, nothing happens. If a library is added, but the user hasn't made an instance of it, nothing happens. This means that a library has to be linked in and the button called "Make Plugin Instance" has to be pressed. Then the user has to select either the library itself or the plugin-instance and press the "Update Models"-button. When this button is pressed the program will try to get the pointer to the model in the library and add it to the current view. For the model to be added to the current view, the user has to add a new image to the BinPic-tool and press the "Create Binary Model"-button. The result of creating a binary model of the uppercase "A" using this approach can be seen in Figure 4.15. We see that the binary model of the screen-shot of the preview of the letter "A" is shown in the current camera-view with the other models in the system. This approach works on the picture-view as well. Now the model is added to the vector of model-pointers in the main program, and updating the view will not cause the binary model to disappear. I have translated and rotated the new model to show that the BinPic-algorithm actually has created a geometric model.

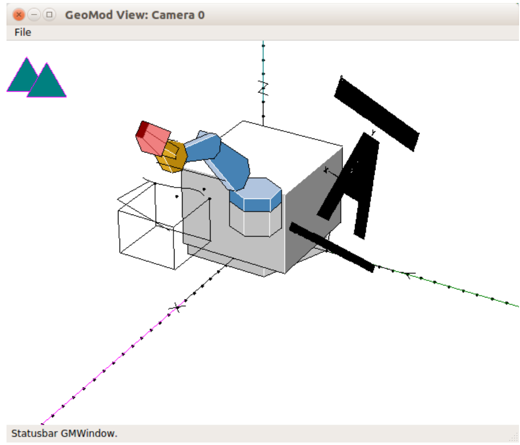


Figure 4.15: Result of previewing A as a binary image

The library now allows us to create binary geometric models from images with different extensions. It works similarly to the old UNIX-system, but it allows users to add colored images directly without conversion using third-party software.

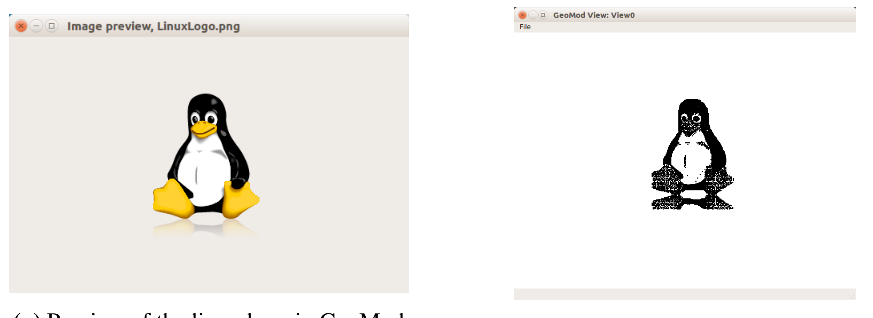
Chapter 5

Image processing

One of the main tasks in this master's thesis is to develop an environment where 3D-models can be recreated from images. This environment will be used by further students to develop a fully automated algorithm that takes images as input and recreate 3D-models from them. One part of this process will be the BinPic-algorithm described in the previous chapter. This recreates a given image in 2D, and should eventually be part of an implementation that recreates 3D-models. This section will describe another part of the process, namely a library that can be used to pre-process the images before they are recreated. The library will use an image processing tool called OpenCV. It should eventually be used to improve the quality of the recreated models and to detect points, edges and surfaces in the input-image. This chapter will only use 2D-images, while the next chapter will look at applications in 3D.

5.1 Binary geometric models without image-processing

We have so far used the BinPic-algorithm on images without pre-processing them. These results have been fairly good on the images tested so far. The colors on these images have been uniform, so the different regions are shown with the same value in the resulting binary geometric model. The image of the Linux logo in figure 5.1a has different shades of yellow and grey areas. The result of running the BinPic-algorithm on this image, and showing it in the picture-view, is shown in figure 5.1b. This shows black dots on the pixels inside the penguin and on its beak and feet. The density of the black dots represents the intensity of the color in the original image. This is correct, but we would rather have a resulting model where these areas are either all black or white. For us to control the resulting model we need to process the image before it's sent to the BinPic-algorithm.



(a) Preview of the linux logo in GeoMod

(b) Result of applying BinPic to the Linux logo

Figure 5.1: Preview and BinPic-model of the Linux-logo

5.2 Image-processing

On the UNIX-system, a third-party software called Gimp [18] was used to pre-process the images. Gimp is a free, cross-platform image editor available on Windows, OS X and Linux. The user had to open Gimp, edit the image and use the result as input to the BinPic-algorithm. This solution works, but I want the user to be able to do the image-processing directly in GeoMod without opening a separate program. I looked through the documentation for Gimp to see if it has an API allowing me to use the image-processing functions directly in C++. No such API exists, but I found out that running Gimp on a server achieves this [19]. For this to work, a running instance of Gimp has to be on a server with an API allowing us to utilize its functions.

After searching online I found several C++-libraries designed for image-processing. One of these is a library called OpenCV [20], which is well integrated with Qt Creator [21]. After looking through the documentation I found that OpenCV has all the image-processing functions needed in this project. Adding platform-specific code in the pro-file also allows us to utilize it on all our platforms. This library allows us to utilize all the image-processing functions we need, with code, and it doesn't have to be running on a server. OpenCV is a lot simpler than the solution described above, so I chose to use it in our program instead of Gimp. The installation and a simple test-code for OpenCV on Ubuntu and Windows is described Appendix A.

5.2.1 Library for image-processing

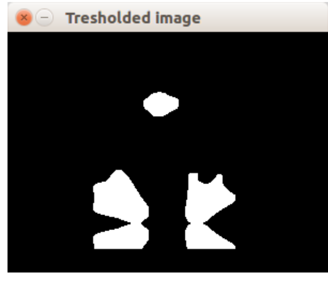
The test-code in Appendix A creates a library that opens a widget with two buttons. The first button, "Add Image", opens a file-browser that can be used to find an image on the computer and add it to the widget. The second button, "Test OpenCV", opens an OpenCV-widget previewing the selected image. I will now add a new button, "Detect Orange Areas", that detects orange areas in the input-image. These areas will be shown as white in a new binary image while the rest of the image is black. When using the image of the Linux logo, the beak and feet of the penguin should be shown in white. The function connected

to this button is implemented as follows:

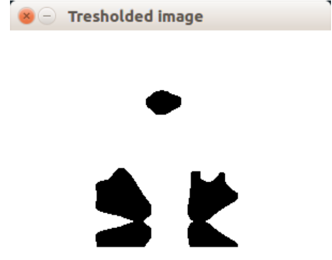
```
void OpenCVTool::openDetectOrangeAreas() {
    int columnCheck = treeWidget->currentColumn();

    if(columnCheck != -1) {
        std::string pluginPath = treeWidget->currentItem()->
                                 text(1).toStdString();
        Mat img = imread(pluginPath);
        Mat imgHSV;

        cvtColor(img, imgHSV, COLOR_BGR2HSV);
        Mat imgThresholded;
        inRange(imgHSV, Scalar(15, 0, 0), Scalar(38, 255, 255),
                imgTresholded);

        imshow("Thresholded image", imgTresholded);
        imshow("Original image", img);
    }
}
```

The if-statement in line 4 ensures that nothing happens as long as an image isn't selected. Lines 5 - 6 inside the if-statement store the path of the image in the variable pluginPath. Line 7 uses this path to store the original image in the variable img. Line 10 uses the function cvtColor() to store the image in the variable imgHSV. This image is now stored as a grey-scaled image in the HSV-format. This format will be described later in this section. Line 12 returns the parts of the image in the hue color-space from 15 - 38 as white, and the rest of the image as black. This result is stored in the variable imgThresholded. The range 15-38 in the hue colors-space represents the different shades of orange. Lines 15 and 16 show the newly created image and the original image in two separate widgets. The newly created binary image is shown in figure 5.2a. We see that the beak, the feet and its shadow are shown in white on a black background. These are all the orange areas in the original image, so the code works as intended. The areas we are interested in are now shown in white. In this example it might be natural to show these in black on a white background. An example of this is shown in figure 5.2b. This figure is generated using the same code, but with different values for the hue. I will later in this master's thesis work with the binary geometric models in 3D. It is then easier to determine the plane of the selected model when the background is black. These figures show us that the desired result can be obtained by changing the values used when the images are generated.

The values for detecting different areas in the input-image are now hard-coded to orange. The only way to obtain another result is by changing the source-code. This is not practical, so I want to add a control-panel that allows the user to adjust the color and two other properties. This control-panel can be seen in figure 5.3. We see that it consists of six sliders. The first two allow the user to set the interval for the property hue. The next two sliders allow the user to set interval for saturation, while the last two set the interval for value. Hue represents the colors to be filtered out, saturation the amount of white this color is mixed with, and value the amount of black it is mixed with. In practice, the process of creating



(a) Showing the orange areas as white



(b) Showing the orange areas as black

Figure 5.2: Binary images created by filtering on the orange colors

the binary image will be: filter out the colors using the hue-sliders and improve the result using the other four. Hue is here represented as values from 0 - 179. The corresponding values for the different colors are as follows:

Orange: 0-22
Yellow: 22-38
Green: 38-75
Blue: 75-130
Violet: 130-160
Red: 160-179

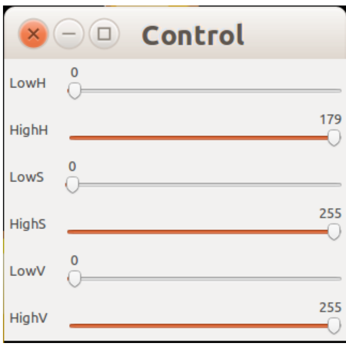


Figure 5.3: Controlpanel for creating binary images

The button called "Detect Orange Areas" has been renamed to "Open Processing" and connected to the function openOpenProcessing() in "opencvtool.cpp". The code inside this function is as follows:

```
cvNamedWindow("Control", CV_WINDOW_AUTOSIZE);

int iLowH = 0;
int iHighH = 179;

int iLowS = 0;
int iHighS = 255;

int iLowV = 0;
int iHighV = 255;

cvCreateTrackbar("LowH", "Control", &iLowH, 179); // Hue (0-179)
cvCreateTrackbar("HighH", "Control", &iHighH, 179);

cvCreateTrackbar("LowS", "Control", &iLowS, 255); // Saturation (0-255)
cvCreateTrackbar("HighS", "Control", &iHighS, 255);

cvCreateTrackbar("LowV", "Control", &iLowV, 255); // Value (0-255)
cvCreateTrackbar("HighV", "Control", &iHighV, 255);

Mat img = imread(pluginPath);
Mat imgHSV;
Mat imgThresholded;

imshow("Original image", img);
cvtColor(img, imgHSV, COLOR_BGR2HSV);

while(waitKey(30) != 27) {
    inRange(imgHSV, Scalar(iLowH,iLowS, iLowV),
            Scalar(iHighH, iHighS, iHighV), imgThresholded);
    imshow("Tresholded image", imgTresholded);
}
```

Line 1 creates a new widget called ”Control” which contains the sliders. Lines 3-10 store the initial values for these sliders in appropriate variables. Lines 12-19 create the sliders with correct initial and max values inside the widget. The input-image is stored in the variable img in line 21, and a new widget containing this image is shown on screen in line 25. The HSV-version of this image is created and stored in the variable imgHSV in line 26. The while-statement in line 28 applies the hue, saturation and value, defined by the sliders, to the HVS-image, and stores the result in imgThresholded. This image is shown in a new widget in line 31. The while-statement updates the output-image until the escape-key is pressed. This means that the output-image is updated when the sliders are moved.

The result in figure 5.2a can be obtained by setting the sliders for the hue to the previously hard-coded values, namely 15 and 38.

The goal is to have a fully autonomous system. In this library, that can be achieved by looping through the values in the sliders until a satisfactory binary image has been created. The image should then be sent to the BinPic-algorithm. An autonomous system will not be achieved in some time, so the control-panel currently allows for user-testing.

5.2.2 The HSV-format

The HSV-format is used instead of RGB because it allows us to adjust the color-intensities as well as the color itself. This is important for two reasons: 1) It allows us to filter out both shadows and light when creating a binary image. A widely used example for this is an image of a stop-light. A person can clearly see when a stop-light is red, but for a computer-program this is a lot harder to determine. Surprisingly, red is often not the most dominant RGB-component in such an image. Shadows and light makes it hard for the computer-program to process the image correctly. The HSV-format allows us to filter out the shadows and light, and the shades of red in the resulting image can easily be identified. 2) Shadows can be used to determine 3D-coordinates. The reflection on a surface can tell us about the geometric properties of the model depicted when we know the location of the light source.

A color in the HSV-format is represented as a cylinder of hue, saturation and value. These are the same properties we are adjusting with the sliders. A graphical representation of this cylinder can be seen in figure 5.4. The color itself changes with the value of hue. This gives us the ability to choose a specific color or a selection of several colors. We see from the figure that the color turns white when the radius of the cylinder approaches zero, regardless of the hue. This property is called saturation. As the saturation increases the color gets darker. The third property, value, works in a similar manner from black at the base of the cylinder to lighter colors at the top. If we choose all sliders to go from zero to the maximum value in our library, all possible colors will be selected. No colors will then be filtered out, and the resulting binary image will be all white.

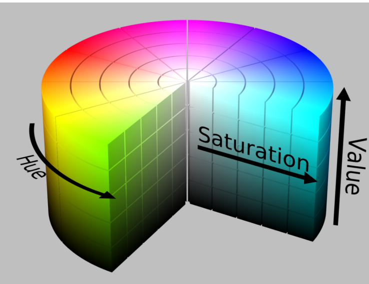


Figure 5.4: Graphical representation of the HSV-format

5.2.3 OpenCV and the BinPic-algorithm

The resulting binary image created using the OpenCV-library should be used as input to the BinPic-algorithm. This image is currently stored locally and shown in a widget. We

want to store this image so the BinPicTool-library can find and use it. I created a new button called ”Save Image” connected to the function openSave() in ”opencvtool.cpp”. When pressed, this opens the widget shown in figure 5.5. The text-box contains the file-name the binary image should be stored with. It is initialized with the file-name of the input-image, and it can be edited by the user to store it with a different name. Pressing the button ”Save” will store the new binary image in the folder ”BinaryImage” in the home-directory of the project. I have also added a check that only allows the user to store the image after the binary image has been created.

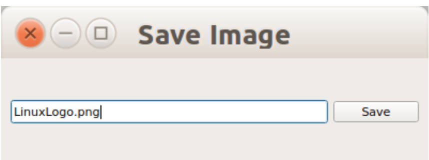
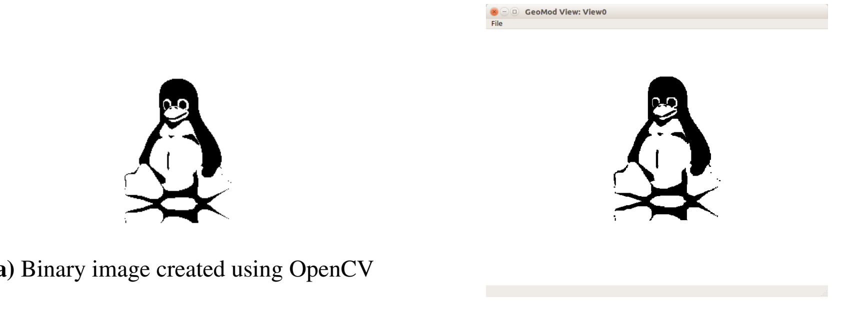


Figure 5.5: Widget for saving images

It is now time to test if using the OpenCV-library improves the resulting binary geometric model. I created a new binary image using the OpenCV-library. The same image as before is used, namely the Linux logo shown in figure 5.1a. I adjusted the sliders to create the binary image in figure 5.6a. This was saved and used as input to the BinPic-algorithm. The resulting binary geometric model shown in the picture-view can be seen in figure 5.6b. When using a color-image as input to the BinPic-algorithm, we got black dots inside the resulting model. By comparing this new result with the one in figure 5.1b we see that these are almost gone when pre-processing the image, so using the OpenCV-library library produces the desired results.



(a) Binary image created using OpenCV

(b) Binary model created from new binary image

Figure 5.6: New binary image and model

One thing to note is that the QBitmap-class doesn’t do anything when the input-image already is on binary form. This means that the user can import both color-images directly or create binary ones using the OpenCV-library.

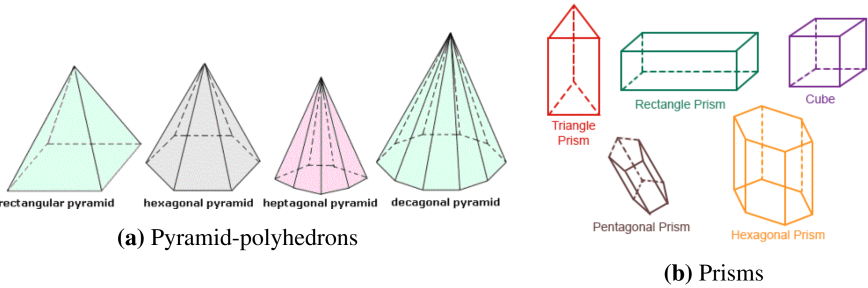
Experimental Process

The main focus of this master's thesis is developing a framework that allows for experimentation with different algorithms building 3D geometrical models automatically from digital images. The dynamic linking of views, BinPic-algorithm and pre-processing using OpenCV are all parts of this process. These tools can be used to visualize, build and improve two-dimensional geometric models manually. This process has to be adapted to 3D-models as well as being automated. Implementing the whole process of creating 3D-models autonomously is to big for this master's thesis, but this section will outline the process. I will describe the parts that aren't implemented, and how the tools already developed fit into the process. This is a good starting point for later students working on this project. It allows them to see how their master's project and master's thesis fits into the overall process.

6.1 Experimental process of creating models from images

This section will describe the process from a physical model to a geometric model. I will describe the shortcuts I have taken where further development is needed. One model will be used throughout this section to show the steps this goes through. It is here natural to choose one of two different types of models, either a pyramid-polyhedron or a prism. Examples of these types can be seen in figures 6.1a and 6.1b.

I want to show the process on a simple model. It doesn't matter which one is chosen, so the model similar to the ”Triangle Prism” in figure 6.1b will be used. This model is simple and different from the other models currently implemented in the program.



(a) Pyramid-polyhedrons

(b) Prisms

Figure 6.1: Examples of different models

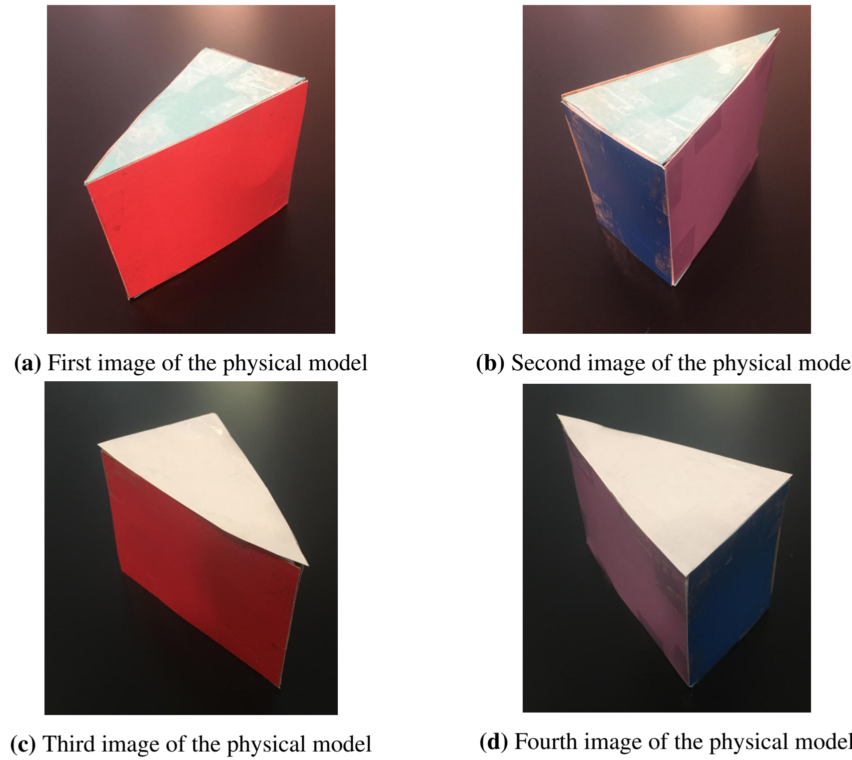
6.1.1 Creating a physical model of the Triangle Prism

I created a physical model of the triangle prism using cardboard. Each of the five surfaces in the model are created with different colors. It is then easier for both the human readers and the OpenCV-library to distinguish them from each other. Then I photographed the model in an environment with a neutral background. These images are shown in figures 6.2a, 6.2b, 6.2c and 6.2d.

6.1.2 Creating binary images of the model

Next I used the OpenCV-library created in the previous section to create a binary image of each surface. Creating one binary image for each surface is relatively easy since each surface has a different color. It might not be that easy for other images where a model has the same color on each surface. Therefore, an algorithm should be developed that detects the reflection of each surface. This reflection can then be used to find intersections between surfaces to tell them apart. The binary image created of the red surface in figure 6.2c is shown in figure 6.3. Binary images of the other surfaces were created as well, but I will not show all of them here. Several of the surfaces appear in more than one input-image. I have here chosen the input-image yielding the best binary image when recreating the model. The binary images are created with the surface shown in white on a black background. This makes it easier to determine the different planes when the geometric models created from these images are shown in the camera-view.

The binary images created by the OpenCV-library were positioned relative to each other in an online tool. These allow us to see the binary outline of the prism in figures 6.4a and 6.4b. These aren't perfect, but they clearly show the outline of the model represented by the individual binary surfaces. Before we move on I want to discuss the way these images are created. I used an online tool to create these models by hand. Our program cannot use an online tool, so it has to build this representation itself. I tried doing this by creating binary geometric models of the images using the BinPic-algorithm, and positioning these in the camera-view. This result is harder to interpret, as can be seen in figure 6.5. The black background hides parts of the models which makes it harder to see its outline.



(a) First image of the physical model
(b) Second image of the physical model
(c) Third image of the physical model
(d) Fourth image of the physical model

Figure 6.2: Images of the physical model of the prism

To get a better representation of the binary models in the camera-view we have two possible solutions. 1) Show the surfaces in black, on a white background, when creating the binary images. The result shown in the camera-view will then allow us to see the outline of the original model better. However, with this solution it's harder to determine the planes of the surfaces, which makes it harder to position the surfaces correctly. 2) Create one binary image from the input-image with all the surfaces. In this resulting image the surfaces are already positioned correctly. Creating this image will most likely be harder than in the first case. If the surfaces in the input-image have different colors, one of them might be filtered out when removing colors in the background. When surfaces in the input-image have the same color they might be shown as one surface in the resulting binary image. Such surfaces might be separated by adjusting the reflection from the light-source, but this could be difficult if they have the same orientation.

Both of these solutions have pros and cons, and the final implementation has to work autonomously. In the first solution this means an implementation positioning the surfaces automatically. If the second solution is chosen, the implementation has to distinguish the surfaces based on both color and reflection. The binary images shown in figures 6.4a

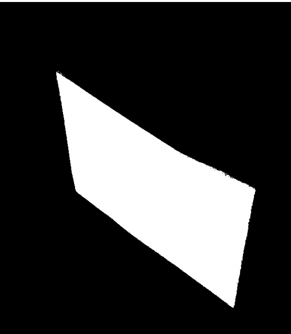
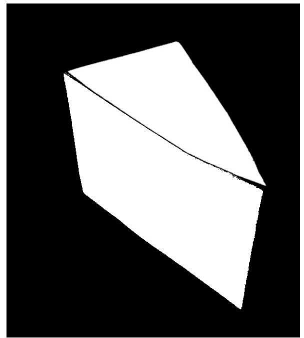
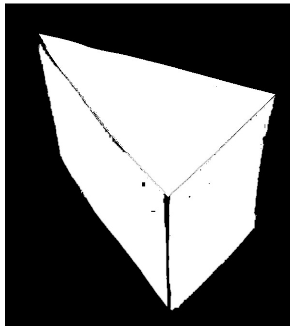


Figure 6.3: The red surface after using the OpenCV-library to create a binary image



(a) Binary prism shown from one side



(b) Binary prism shown from the other side

Figure 6.4: Binary prisms

and 6.4b are only used as an internal representation to help the program find the surfaces, edges, and points. The result in the camera-view in figure 6.5 is only for a human user to visualize the problem, and isn't necessary to solve it. I have here taken a shortcut and created the images in figures 6.4a and 6.4b in an online tool. Which solution to choose, and the implementation itself, is left for further students.

6.1.3 Identifying properties and recreating the model

I continue based on the assumption that models similar to figures 6.4a and 6.4b have been created in the GeoMod program. The next step is to identify surfaces, edges, and points from these models. This is not implemented, so I will explain the process. One model is enough to explain the concepts, so I will only use the model shown in figure 6.4a. The system should autonomously identify the points and edges in this model. A manually created example of this is shown in figure 6.6. I have added colored points and lines on top of the visible corner-points and edges. Showing this result in GeoMod is not necessary to find these properties, but a similar visualization should be created to allow the programmer to

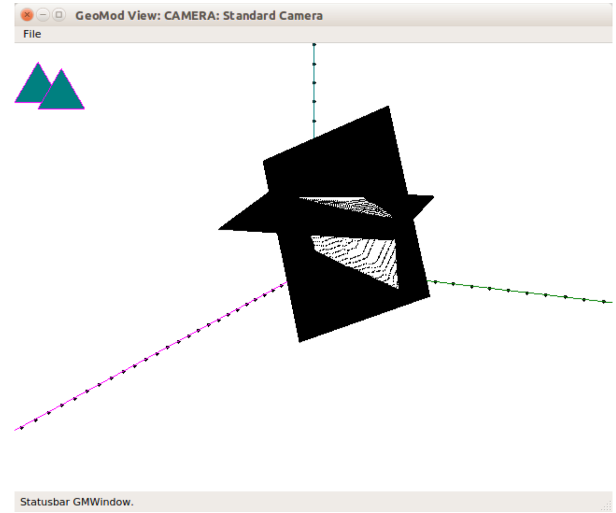


Figure 6.5: Creating a model of the prism in GeoMod using BinPic

verify the results. When the system is fully autonomous these properties should be stored and used in the recreation of the model.

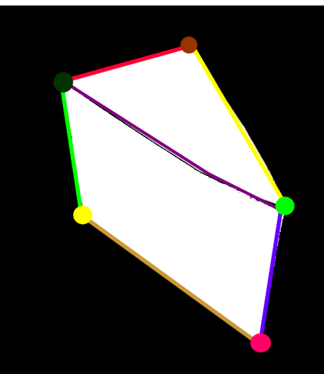


Figure 6.6: Prism model with edges and points identified

The last step is to develop an algorithm recreating the model based on the properties stored in the processes above. A good starting-point is to draw the points and edges in the camera-view and see if this outline resembles the input-images. This is not necessary for the autonomous system, but it shows the developer if the steps above are implemented correctly. At this stage the points and edges should define each surface in the input-images. The algorithm should use this information to create the surfaces as geometric models in GeoMod. Some of the input-images might show the same surface. This means that the program has to keep an internal representation of the whole model and recognize if a sur-

face has been drawn earlier. If not, the new surface is added to the model. Otherwise, this new representation can be used to improve the surface in the model. The input-images might only describe parts of the model. This means that we will not be able to correctly recreate the surfaces we cannot see. We here have a few possible solutions. 1) Only draw the surfaces visible in the input-images. In this case, we end up with an unclosed model until we get images describing all the surfaces. In the current camera-view only the front of the surfaces are drawn. Looking at the model from the back will then make the model disappear because the back of the surfaces are invisible. 2) Enclose the back of the model with a simple plane. This plane will be visible in the camera-view, which tells us that there is a model there. This surface has to be removed and recalculated when images describing new surfaces appear. 3) Analyze the surfaces in the current model and guess the shape of the rest of the model. In figure 6.4a it's quite easy to see what the resulting model should look like. An algorithm could be developed, which approximates the model based on the surfaces defined so far. This means that several reference-models have to be stored and used by the algorithm. The approximated model has to be updated each time images of new surfaces arrive. This solution is the most complex, and might give unexpected results. I would then suggest closing the model with a simple plane or modifying the camera-view to draw both the front and back of the surfaces.

The recreation of the models described above is not implemented, so I will recreate the model manually in a separate library. This model is used to show an approximation of what the resulting model should look like. The dimensions of the model are based on approximations from the physical model and the images. The exact geometry of the recreated model will not be correct, but similar to the physical model. The model is created in the library "DynamicLinkingTests/16_RecreatedPrism". It is created in the same manner as other models in the system, so I will not show this code here. The model is shown in the camera-view in figure 6.7.

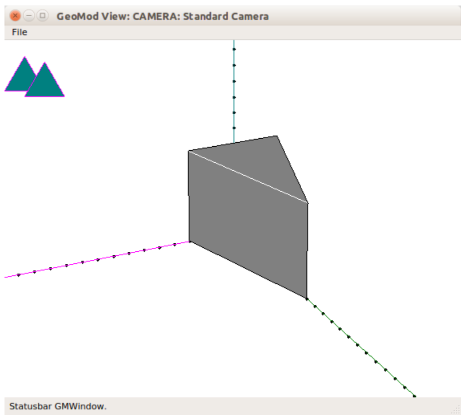


Figure 6.7: Example of recreated prism in GeoMod

6.2 Notes and possible problems during development

The model in figure 6.7 only has grey surfaces. The color of each surface is not important for the autonomous vehicle, but it might help the developer compare the recreated model to the original one. The color of each surface can be stored when the binary images are created by the OpenCV-library. It can then be added when the surfaces are created.

The x- and y-coordinates of the surfaces, edges, and points can fairly easy be found from the binary images. The z-coordinates can also be approximated, but these are harder to determine. I will here mention a few ways this can be done. 1) Use the reflection of the light on each surface in the original image. This assumes that the program knows the position of the light-source. It can relatively easy be determined in a controlled environment, but it might be harder to determine in real-world images with several light-sources. 2) Combine images from several angles and calculate the coordinates. This assumes several images of the same surface from different angles. 3) Combine input-images with other types of inputs, for instance sonar. This assumes the availability of other types of inputs. 4) Compare the input-images with a selection of reference-models, and assume that the model to be recreated has the same geometric proportions. For this to work, a reference-model of the model to be recreated has to exist.

None of these solutions works for all scenarios. Problems arise if the position of the light-source cannot be determined, only one input-image is available, information from other sensors are unavailable or if the model is different from the reference-models. With this in mind, at least two of the solutions should be implemented. One can then use one solution as default, with the rest as backup if this one fails. The implemented solutions can also be combined to improve the accuracy of the coordinates calculated.

When the model has been recreated, an iterative process should be used to improve the resulting model. The system should take images of the recreated model and compare this to the input-images. The difference between these images should then be used to improve the resulting model. This process is repeated until the difference between them are below a given threshold.

The developer might also take a look at the geometric models created by the BinPic-algorithm. All edges in these models, both around planes and edges themselves, are composed of several short vectors. The BinPic-algorithm uses scan-lines to detect extremal-points. An extremal-point is found when a bit in the binary image changes from white to black, or black to white. When two extremal-points are found in consecutive scan-lines the algorithm creates a vector between them. This means that the algorithm creates a vector between an extremal-point in one scan-line and an extremal-point in the next. Two scan-lines are one pixel apart, so we end up with one vector per scan-line. Often extremal-points span several scan-lines to form a line. Rather than creating this line by several short vectors, we want it to be composed of one, longer vector. The program should here do a regression analysis to compose the smaller vectors into longer ones. For models composed of straight lines, like simple polyhedron and prisms, a linear regression analysis would be enough. For models containing curves, a more complex regression anal-

ysis is required. A linear regression analysis is a good place to start when developing this comparison-algorithm. Implementing such an algorithm would allow the program to draw and compare the models more effectively.

6.3 Summary

The system should begin by creating binary images of surfaces in the input-images, using the OpenCV-library. This functionality is implemented, but done manually. These binary images have to be positioned relative to one another to obtain the outline of the original model. At this point the program has a shell of the original model. This can then be used to determine the surfaces, edges, and corner points. These properties, combined with an algorithm for determining the z-coordinates, should be used to recreate the model. The system should perform an iterative process where this model is gradually improved until it is considered close enough to the original model depicted in the input-image.

This chapter explains the whole process from input-images to a 3D-model shown in the camera-view. I have taken several shortcuts to show the whole process, since parts of the process isn't implemented. Further development should start by implementing the missing algorithms, and running the process with human interaction. After this, the attention can be switched over to running it autonomously.

My assignment states that I should develop a platform which allows for experimentation with different algorithms for building 3D-models from images. It also states that image-processing should be emphasized. I have installed OpenCV and created a library that allows the user to create binary images with a control-panel. This image can then be used by the BinPic-algorithm where a binary geometric model is created. At this point conversations with my supervisor revealed that I am at the point I wanted to be at the end of my thesis. There are still quite a few parts to be implemented in this process, but these will be left for further students on this project. I will instead move the focus over to the visualization of the models in the program.

Chapter 7

OpenGL

The GeoMod-program uses a class called GMWindow to visualize the geometric models. This class is developed by my supervisor, and works as intended, but it lacks some desirable functionality. Two additional properties we want are z-buffering and adding a light-source. Z-buffering determines the parts of the models that are closest to the camera, draws these and hides the parts hidden by other surfaces. The GMWindow-class draws the surfaces a bit at random. The back of each model is correctly hidden, but which parts of the front-surfaces to draw is not implemented. Showing parts of models that should have been hidden might confuse the program and give unwanted results. In a fully autonomous system this might be the difference between navigating safely around an object and crashing into it. Adding a light-source helps us recreate models from images correctly. A surface has a different reflection depending on the location of the light-source. Adding a light-source in the view gives us a reflection on the surfaces in the recreated models. These reflections can then be compared with the reflections on the same surfaces in the input-images. This can help us to determine if a model has been recreated correctly.

In my master's project I looked into the possibility of adding these properties to the GMWindow-class. I found that these had to be implemented from scratch, which is a big task. After a bit of searching online, I found a library called OpenGL [22]. OpenGL allows the programmer to render 2D and 3D graphics with Qt Creator. It also has the possibility of adding both z-buffering and light-sources. When discussing it with my supervisor, we figured that OpenGL was worth a try. This chapter describes the implementation of OpenGL and the challenges encountered in this process.

7.1 Re-implement OpenGL in GeoMod

OpenGL has been tried implemented in the program in the past. The code contains implementations of old OpenGL-functions, commented out. My supervisor told me that this implementation didn't work. He wasn't sure why, just that the models weren't drawn cor-

rectly. This old code is written in the same files as the GMWindow-class. The thought was that the programmer could choose to use either the camera-view or the OpenGL-view based on certain hard-coded variables. The GMWindow-class has been updated quite a lot in recent years, but the OpenGL-implementation is still the same. This means that the code for OpenGL is based on a prevous version of both Qt Creator and GeoMod. The OpenGL-functions have also been updated, as can be seen in the section "Relation to QGLWidget" in [23]. Because of this, compiling the old OpenGL-code causes a lot of errors. The classes using the deprecated QGLWidget have to be re-implemented to use the new class QOpenGLWidget, and the code has to be rewritten to work with the newer version of Qt Creator. Spending time to fix this on an implementation that originally didn't work is probably a waste of time, so I decided to start from scratch.

I created a new library in "DynamicLinkingTests/14_TestOpenGL" that can be linked in through the Tools-manager. It contains an implementation of the interface-functions and returns a widget with a single button. This button opens a new OpenGL-window containing a triangle drawn with different colors. I will not show the implementation of the triangle here, since the code is the same as the example in [24]. It is only created to test that OpenGL can be used in a library with the current version of Qt Creator. The resulting OpenGL-window can be seen in figure 7.1. We see that the triangle is created and shown in a new widget.

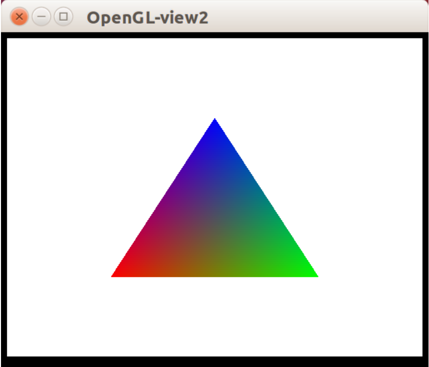


Figure 7.1: Drawing a triangle with different colors using OpenGL

The triangle above is a 2D-model, and the next step is to test if we can create 3D-models. I followed the tutorial in [25], which contains a 3D-model of the Qt-logo that can be rotated using sliders or the mouse. I replaced the logo with an implementation of a cube. The code for this is quite long and almost identical to the one shown in [25], so it will not be shown here. One thing to note is that I replaced the line

```
Logo m_logo;
```

with

```
TestCube m_model
```

in ”glwidget.h”. This tells the code to draw the cube instead of the logo. For this to work I had to replace all occurrences of m_logo with m_model in ”glwidget.cpp”. The result can be seen in figure 7.2. The cube is shown in a OpenGL-window and can be rotated using the mouse or the sliders on the right. This OpenGL-window allows me to draw 3D-models and rotate them in a simple manner, so I will use this as a platform for further development of OpenGL.

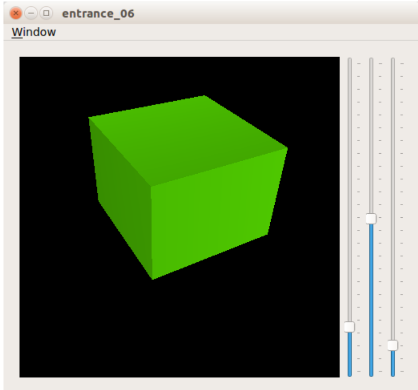


Figure 7.2: Figure of creating a cube in OpenGL

7.1.1 Drawing the models in the system

We now want to be able to draw the models currently in the program in this OpenGL-view. This code will be stored in the files ”geomodtest.h” and ”geomodtest.cpp” in the TestOpenGL-library. To show the models created in these files I changed the line

```
TestCube m_model
```

to

```
GeoModTest m_model;
```

in ”glwidget.h”. This creates a new instance of the GeoModTest-class, and draws the models created here instead of the cube. I also included ”geomodtest.h” at the beginning of the file. Now we need to find, convert, and draw the models from the main program in the OpenGL-view. Before beginning on this task I want to explain how models are drawn in OpenGL. OpenGL has three drawing-options, namely drawing points, lines and triangles. We want to draw 3D-models, so we have to use the last option, triangles. Drawing points and edges can be used to show the outline of the models, but they cannot be used to draw

the surfaces. In GeoMod, models are created by adding corner points. Edges are then created from these points, and used to create the surfaces. Any number of points can be used to create a surface. OpenGL only allows us to create triangles, so we have to convert the surfaces in GeoMod to triangles before drawing them. My supervisor and I discussed the possibility of storing the models in the system as triangles, instead of the way they are stored now, but we chose not to. Changing the way the models are stored would mean rewriting a lot of existing code. We instead chose the solution where we convert the existing models to triangles in the OpenGL-library.

I will begin by drawing the recreated prisms shown in figure 6.7 in the OpenGL-view. Selecting a single model from the main program easily allows me to see if it is drawn correctly. Before an algorithm that draws the models in the OpenGL-view can be implemented, they have to be sent to the library. I re-implemented the interface-function setModels() in ”plugininterface.h” which allows me to do this. The function is called in makePluginInstance() in ”toolsman.cpp”. This code is executed when the library is instantiated, and the models should be sent to the library. The added code is shown below. The first line creates a variable, models, which contains a pointer to the models in the system. This variable is then used as input to setModels().

```
1  Models* models;
2  factory.getPlugins()[index].pluginP->setModels(models->getModels());
```

setModels() is an interface-function, so it’s implemented in the library as well. The implementation is found in ”testopengl_if.cpp” and is as follows:

```
1  void TestOpenGL_if::setModels(std::vector<Models::ModelData*> modls) {
2      models->setModelsP(modls);
3  }
```

The input to this function is the pointer to the models in the main program. Line 2 sets a local models-vector, models, equal to the input-variable. This ensures that the library contains a pointer to all the models in the main program. The function is called once, so the library only contains the models present when the OpenGL-library is instantiated. When linking in a new model in the main program, we have to create a new instance of the library to add this to the OpenGL-view. This is not optimal, but it allows us to test our code.

With a copy of the models in the library, the work of drawing them in the OpenGL-view can begin. The models will be converted in the file ”geomodtest.cpp”, so we have to transfer them here. To get the recreated prism, we add the following code in the constructor:

```
1  TestOpenGL_if* top;
2  std::vector<Models::ModelData*> mdls = top->models->getModels();
3  Models::ModelData* mdl = mdls.at(6);
```

Line 1 creates an instance of the TestOpenGL_if-class and stores it in the variable top. Line 2 uses the function getModels() on the vector of models in top, and stores them in the variable mdls. Model number 6, the recreated prism, in mdls is stored in the variable mdl

in line 3. The recreated prism is not statically linked in, so it has to be linked in dynamically before the OpenGL-library is instantiated.

At this point I created a function called drawTriangle() in ”geomodtest.cpp”. With the following implementation this function allows me to draw a triangle in the OpenGL-view from three points.

```
void GeoModTest::drawTriangle(GLfloat x1, GLfloat y1, GLfloat z1,
                              GLfloat x2, GLfloat y2, GLfloat z2,
                              GLfloat x3, GLfloat y3, GLfloat z3) {

    QVector3D n = QVector3D::normal(QVector3D(x3 - x1, y3 - y1, z3 - z1),
                                    QVector3D(x2 - x1, y2 - y1, z2 - z1));

    add(QVector3D(x1, y1, z1), n);
    add(QVector3D(x3, y3, z3), n);
    add(QVector3D(x2, y2, z2), n);

    add(QVector3D(x3, y3, z3), n);
    add(QVector3D(x1, y1, z1), n);
    add(QVector3D(x2, y2, z2), n);
}
```

The inputs to this function are 3 points defined by x-, y- and z-values, which in total means 9 numbers. These define the corner-points of the triangle to be drawn. Lines 5 - 6 creates a normal-vector used to determine the inside and outside of the triangle. The 6 calls to the function add() in lines 8 - 14 adds the points to the OpenGL-buffer twice. Once for the inside and once for the outside, with the same normal-vector. The points are added in a different order to ensure that the normal-vector is positive for the outside and negative for the inside. The inside will then be drawn in a darker color than the outside, to easily tell them apart. The same surface has to be added twice for OpenGL to draw both the inside and the outside. This is because one side, by default, is invisible when a surface is added. Adding the surface again, in a different order, will draw both sides. The function drawTriangle() is similar to triangle() in [25]. drawTriangle() creates a single surface that can be seen from both sides, while triangle() creates two surfaces z apart, that only are visible from one side. The function add() is the same as in [25], so I will not show the implementation here. It adds the vector defining the three points and the normal-vector to the OpenGL-buffer.

The next step is to implement an algorithm that gets 3 and 3 points from the prism-model, and draws them using the function above. Note here that only triangles part of a surface in the original model should be drawn. The implementation is quite long and complicated due to the way the models are stored in the main program. It is implemented in the constructor in ”geomodtest.cpp” after the prism has been stored in the variable mld, described earlier. The complete code is as follows:

```
1  for(int j = 0; j < mdl->getnumber_of_Tgroups(); j++) {
2      Tgroup* modelTgroup = mdl->getTgroupPModifiable(j);
3      ExtBasis* basisP = modelTgroup->getBasisPModifiable();
4      GeomNet* networkP = modelTgroup->getNetwork();
5
6      for(std::list<GeomRegion*>::iterator region=networkP->
7          getRegionsBegin(); region != networkP->getRegionsEnd(); region++){
8
9          currentRegion = (*region);
10
11         std::vector<GeomNode*> nodesToDraw;
12         bool completed = false;
13         bool currentEdgeAlong = currentRegion->entryAlong();
14         GeomEdge* currentEdge = currentRegion->entryEdge();
15         GeomNode* currentHeadNode;
16         GeomNode* currentTailNode;
17
18         bool negEdge;
19         while(!completed) {
20             currentHeadNode = currentEdge->headNode();
21             currentTailNode = currentEdge->tailNode();
22
23             if(currentEdgeAlong) {
24                 currentEdgeAlong = currentEdge->posRegionNextEdgeAlong();
25                 currentEdge = currentEdge->posRegionNextEdge();
26                 negEdge = false;
27             } else {
28                 currentEdgeAlong = currentEdge->negRegionNextEdgeAlong();
29                 currentEdge = currentEdge->negRegionNextEdge();
30                 negEdge = true;
31             }
32
33             if(negEdge) {
34                 nodesToDraw.push_back(currentTailNode);
35             } else {
36                 nodesToDraw.push_back(currentHeadNode);
37             }
38
39             if(currentEdge == currentRegion->entryEdge()){
40                 completed = true;
41             }
42         }
43         drawModel(nodesToDraw);
44     }
45 }
```

The code above loops through all Tgroups in the current model. The Tgroups contain all necessary information about the points, edges and surfaces. Line 2 stores the current Tgroup in the variable modelTgroup. In line 3, modelTgroup is used to get the current basis and store it in the variable basisP. Line 4 gets the network of the current Tgroup and stores it in networkP. This contains information about the regions in model. The for-loop

starting in line 6 loops through each region using an iterator. The current region in the iteration is stored in the variable currentRegion in line 9. Line 11 creates a vector of GeomNodes called notesToDraw. This will eventually contain the points to be drawn in the OpenGL-view. Lines 12-18 define variables used in the while-loop starting in line 19.

The while-loop runs as long as the variable completed is false. This variable is initialized to false, so the loop will run at least once. Lines 20 and 21 set the variables currentHeadNode and currentTailNode equal to the head- and tail-node of the current edge. The current edges is initialized to the entry-edge in line 14. The if-statement in line 23 checks if the variable currentEdgeAlong is true. currentEdgeAlong is true if the next edge is a forward edge, and false if it's a backwards edge. An edge can be defined both ways, hence the two different names. For instance, the edge "BC" is defined by the same points as "CB", but they are opposite one another. The point "B" is the head of the edge "BC", and "C" the tail. In "CB", "C" is the head and "B" the tail. Let us now assume that we have the surface "ABCA", and that "AB" is the entry-edge. currentEdgeAlong will then be true if the next edge is defined as "BC", and false otherwise. If currentEdgeAlong is true, the if-statement in line 23 is entered. If it is false, the else-statement in line 27 is entered. Inside the if/else-statement currentEdgeAlong is updated based on the direction of the next edge in the surface. Then the variable currentEdge is set to the next edge. This will be set to the positive edge in the if-statement, and the negative edge in the else-statement. The boolean variable negEdge is set to false when the if-statement is entered, and true when the else-statement is entered. This variable is used in the if-statement in line 33. The node stored in currentTailNode is added to the vector nodesToDraw if negEdge is true. If negEdge is false, the node stored in currentHeadNode is added.

The if-statement in line 39 is entered if the current edge is equal to the entry edge. The variable completed is then set to false, which stops the while-loop. This means that all edges in the region have been visited, and all points to be drawn have been added to the vector nodesToDraw.

The process above gets the points defining each surface, and adds them to nodesToDraw. nodesToDraw is now a vector of n points, where n is the number of points defining the current surface. The prism described earlier consists of 2 triangles and 3 rectangles. n will then be 3 when a triangle is added, and 4 when a rectangle is added. nodesToDraw is reset for each iteration.

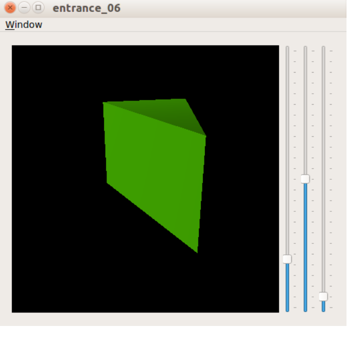
The last line in the code calls the function drawModel(), with nodesToDraw as input. The implementation of this function in "geomodtest.cpp" is as follows:

```
void GeoModTest::drawModel(std::vector<GeomNode *> model) {
    std::vector<GLfloat> points;
    for(int i = 0; i < model.size(); i++) {
        points.push_back(model[i]->getModablXyzP()->get_x());
        points.push_back(model[i]->getModablXyzP()->get_y());
        points.push_back(model[i]->getModablXyzP()->get_z());
    }

    switch(model.size()) {
    case 3:
        drawTriangle(points[0], points[1], points[2],
                     points[3], points[4], points[5],
                     points[6], points[7], points[8]);
        break;
    case 4:
        drawTriangle(points[0], points[1], points[2],
                     points[6], points[7], points[8],
                     points[3], points[4], points[5]);
        drawTriangle(points[0], points[1], points[2],
                     points[9], points[10], points[11],
                     points[6], points[7], points[8]);
        break;
    default:
        std::cout << "The current surface is not valid" << std::endl;
        break;
    }
}
```

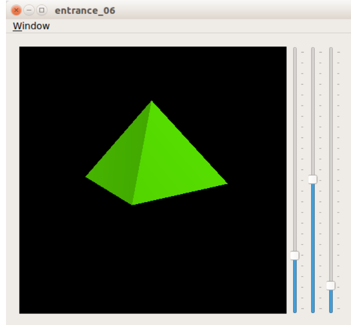
The points to be drawn are now stored in the input-variable model. Line 2 defines a new vector called points, which is populated in the for-loop starting in line 3. This loops through all points in model, and adds the x-, y- and z-coordinates as GLFloats. This is the data-type the previously defined function drawTriangle() expects as input. The switch-statement in line 9 uses model.size() as the argument. This means that the number of points in the current surface determines which code to execute. The code in "case 3" will be executed if the current surface is created from 3 points, which means that it's a triangle. The function drawTriangle() is then called with the 3 points defining the surface. In the vector points this means 3 points with x-, y- and z-coordinates, in total 9 points. This can be seen in lines 11 - 13. The code in "case 4" will be executed if the current surface is a rectangle. A rectangle can be created from two triangles, so the function drawTriangle() is here called twice.

The code above only draws triangular and rectangular surfaces. If other surfaces are encountered, line 24 will print an error-message to the terminal. The prism is defined only by triangles and rectangles, so the code should be able to draw it. I will look at the drawing of more complex surfaces later in this chapter. Running the code on prism after linking it in dynamically gives me the OpenGL-view shown in figure 7.3a. We see that the model is similar to the one in figure 6.7, so the code works as expected.

Drawing the pyramid from the main program in the OpenGL-view is shown in figure 7.3b.



(a) Recreated prism in OpenGL



(b) Recreated pyramid in OpenGL

Figure 7.3: Other recreated models in OpenGL

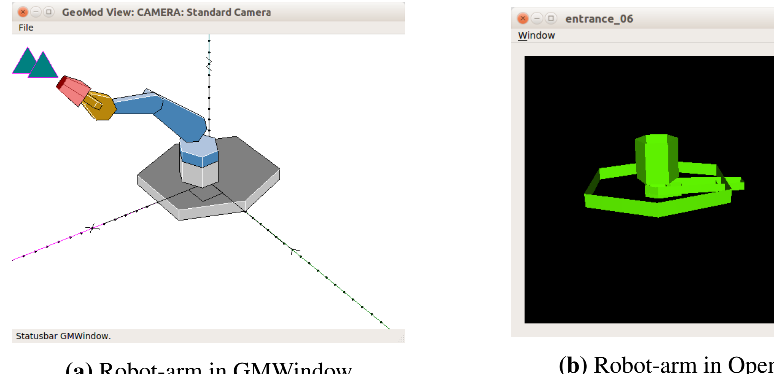
We see that this model is created correctly as well. The pyramid is also defined by triangles and rectangles, so the OpenGL-view is able to draw it.

Before moving on I want to look at a potential side-effect of drawing models using triangles. It was pointed out to me by my supervisor that drawing triangles from edges might cause errors. Figure 7.4 shows an example of an unclosed triangle. We see that one of the edges defining the triangle is too short. Trying to draw such a model on a computer might give unwanted results. The computer cannot determine if the points at the top are inside our outside of the triangle. Drawing this might result in the bottom part, up til the top of the right edge, having one color. The part above the right edge, and all the way to the right of the screen, might get another. These problems appear when triangles are defined by edges. Such problems are fortunately not a problem in our code. OpenGL draws triangles from the three corner-points, which means that the triangle will never be unclosed. We might get an unexpected triangle, but it will still be draw as one.

Figure 7.4: Example of an unclosed triangle

7.1.2 Drawing the Robot-arm in the OpenGL-view

The current implementation of the OpenGL-view allows us to draw simple models. An example of drawing a more complex model with this implementation is seen in figure 7.5b.



(a) Robot-arm in GMWindow

(b) Robot-arm in OpenGL

Figure 7.5: Recreating the robot-arm using OpenGL

This is the result of drawing the robot-arm shown in figure 7.5a. We see that the surfaces consisting of triangles and rectangles are drawn, but they are all drawn on top of each other. This comes from the fact that the models created in the OpenGL-view are drawn relative to the origin. In figure 7.5a each individual part is moved to the correct position. This is currently not implemented in the OpenGL-view. This position, or offset, has to be added to each individual point. For this to be done, we have to get the offset for each part. A function called O() allows us to get the offset in the following way:

```
MathVec offsetP = modelTgroup->getBasisPModifiable()->O();
```

This line is added to the constructor in ”geomodtest.cpp” before we loop through each surface and store the points to be drawn. O() is implemented in ”MaxLib/math/extbas.h” and returns the offset in x-, y- and z-direction. The offset is stored in the variable offsetP, and passed as input to the function drawModel(). The offset is added to each individual point to be drawn. This is done with the code below in drawModel(). The rest of the function is left unchanged.

```
1  std::vector<GLfloat> points;
2  for(int i = 0; i < model.size(); i++) {
3      points.push_back(model[i]->getModablXyzP()->get_x() + offset.get_x());
4      points.push_back(model[i]->getModablXyzP()->get_y() + offset.get_y());
5      points.push_back(model[i]->getModablXyzP()->get_z() + offset.get_z());
6  }
```

Running the code with the added offset yields the result shown in figure 7.6. We see that the offsets are added correctly, but the parts are not correctly oriented. The rotation of each part is added and stored when the robot-arm is created. To account for this rotation in the OpenGL-view I created a new function called rotateNode(). It uses the basis stored in basisP to calculate the rotation of each individual point. The implementation of this function can be seen below.

Figure 7.6: Robot-arm with offset

```
GeomNode* GeoModTest::rotateNode(ExtBasis *basisP, GeomNode *prevPoint) {
    GeomNode* newPoint = new GeomNode;
    IDMthVec* previousPointVec = prevPoint->getModablXyzP();

    newPoint->getModablXyzP()->set_x(
                previousPointVec->get_x()*basisP->X().get_x() +
                previousPointVec->get_y()*basisP->Y().get_x() +
                previousPointVec->get_z()*basisP->Z().get_x());
    newPoint->getModablXyzP()->set_y(
                previousPointVec->get_x()*basisP->X().get_y() +
                previousPointVec->get_y()*basisP->Y().get_y() +
                previousPointVec->get_z()*basisP->Z().get_y());
    newPoint->getModablXyzP()->set_z(
                previousPointVec->get_x()*basisP->X().get_z() +
                previousPointVec->get_y()*basisP->Y().get_z() +
                previousPointVec->get_z()*basisP->Z().get_z());

    return newPoint;
}
```

The input to this function is the basis for the current surface and the points to be rotated. Line 2 creates a new GeomNode in the variable newPoint. This will contain the rotated point and is returned in line 18. Line 3 stores the vector defining the current point in the variable previousPointVec. Lines 5 - 16 calculate the x-, y- and z-rotation, and stores the result in newPoint. The rotation of each point is calculated as follows:

$$\begin{bmatrix} newX \\ newY \\ newZ \end{bmatrix} = oldX * \begin{bmatrix} basisX1 \\ basisX2 \\ basisX3 \end{bmatrix} + oldY * \begin{bmatrix} basisY1 \\ basisY2 \\ basisY3 \end{bmatrix} + oldZ * \begin{bmatrix} basisZ1 \\ basisZ2 \\ basisZ3 \end{bmatrix} \tag{7.1}$$

The old x-, y- and z-coordinates are multiplied with the basis to give us the new, rotated

coordinates. This function is called before the vector nodesToDraw is populated. Earlier we had the two following lines in ”geomodtest.cpp”:

```
nodesToDraw.push_back(currentHeadNode);
nodesToDraw.push_back(currentTailNode);
```

These are now replaced with:

```
nodesToDraw.push_back(rotateNode(basisP, currentHeadNode));
nodesToDraw.push_back(rotateNode(basisP, currentTailNode));
```

The implementation now includes both the offset and the rotation, as can be seen in figure 7.7.

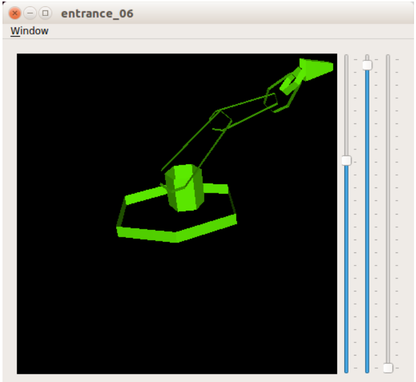
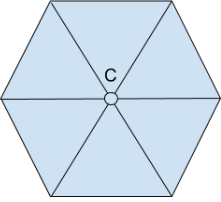


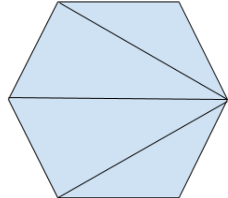
Figure 7.7: Robot-arm with rotation

7.1.3 Drawing complex surfaces

We see in figure 7.7 that only triangular and rectangular surfaces are drawn. We now want to draw the rest of the surfaces as well. There are a lot of traditions for splitting complex surfaces in for instance tessellation, photo-realistic images, FEM(Finite Element Method) and STL(3D printing). I will here keep it simple and propose two solutions that will solve our problem. 1) Create a new point located at the center of gravity of the surface. This point, and two others defining the surface, can then be used to create triangles. Continue this process with two and two points until the whole surface has been created. Using this approach on the surface at the base of the robot-arm will split the surface as in figure 7.8a. Point ”C” defines the center of gravity. Here, the surface is created using 6 triangles. This solution requires us to add and store the extra point ”C”. 2) Use one point as a reference-point and create triangles using this and two other points along the edge. This surface will then be split into the triangles shown in figure 7.8b. The surface is here created from four triangles without finding and storing the extra reference-point. This makes the implementation easier, so I chose to implement this solution in the function drawPolygon() in



(a) Create polygon from center of gravity



(b) Create polygon from existing points

Figure 7.8: Two different way of drawing polygons using triangles

"geomodtest.cpp":

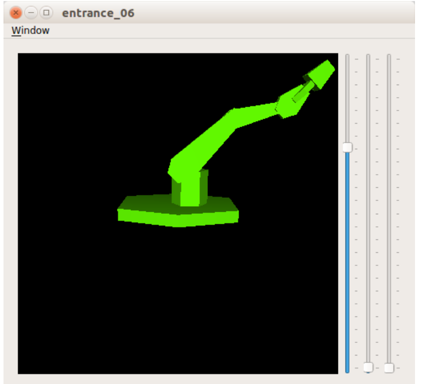
```
1  void GeoModTest::drawPolygon(std::vector<GeomNode *> model,
2                               std::vector<GLfloat> points) {
3      for(int i = 0; i < model.size()-2; i++) {
4          drawTriangle(points[0], points[1], points[2],
5                 points[3*(i+2)], points[3*(i+2) + 1], points[3*(i+2) + 2],
6                 points[3*(i+1)], points[3*(i+1) + 1], points[3*(i+1) + 2]);
7      }
8  }
```

The for-loop goes from 0 to the number of points in the current surface minus 2. Lines 4 - 6 calls the function drawTriangle() for each iteration. For the surface in figure 7.8b with 6 points, this means calling drawTriangle() 4 times. The first argument to drawTriangle() is the same for each iteration, namely the first point defined in x-, y- and z-direction. The loop uses this point and two other points for each iteration to draw the triangles. The first triangle is created by points 1, 3 and 2. The second using points 1, 4 and 3 and so forth, until all the triangles have been created. Here I add the point i+2 before i+1. This is done to ensure that each triangle has the correct normal-vector. The inside- and outside-surfaces will then be drawn correctly. The function also draws triangles and rectangles correctly. This means that we can remove the switch-statement in "geomodtest.cpp" and replace it with the single line:

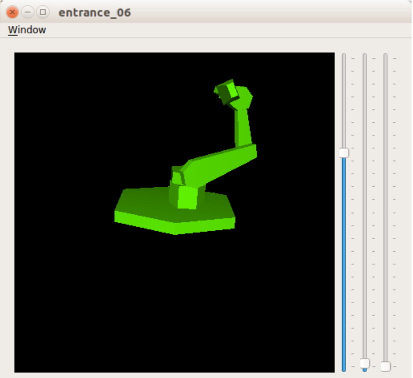
```
drawPolygon(model, points);
```

Triangles will then be drawn directly, and other surfaces will be split into several triangles. The result of drawing the robot-arm can now be seen in figure 7.9a. We see that all surfaces are drawn with the correct offset and orientation.

I explained earlier that the basis is used to obtain the correct orientation for each part. This basis changes each time a part or model is rotated in the main program using the control-panels. To test that it's updated correctly, I rotated each of the six joints in the robot-arm. The result in the OpenGL-view can be seen in figure 7.9b. We see that all joints have been



(a) Robot-arm with rotation and complex surfaces



(b) Robot-arm in a new position

Figure 7.9: Robot-arm with two different rotations

rotated. This is correct compared to the same robot-arm in the camera-view.

The main goal for this section was to add the properties z-buffering and a light-source. Both of these properties work in the models recreated. Figure 7.9a only show the parts of the surfaces that are closest to the camera. Which parts of the surfaces that are drawn changes when the model is rotated, as can be seen in figure 7.9b. We also see that the surfaces have different shades of green depending on their rotation. The surfaces shown directly in front of the camera have a brighter color than the tilted ones. This shows that the light-source is added correctly and affects the resulting color of the surfaces.

7.1.4 Add colors to the models

Figure 7.5a show that the parts in the robot-arm have different colors in the original camera-view. This makes it easier for the user to separate the parts in the model. I will now add this property to the OpenGL-view as well. The current code draws all the models in green, so we need to change this implementation. In the code for the camera-view, the RGB-components for the color are added to the surface when it's drawn. I will use the same approach here. Each time a surface is added through the function drawTriangle(), I add the RGB-components of the color to a vector called colors. The color of each surface can be accessed through the variable currentRegion in ”geomodtest.cpp”. The three lines below use this variable and adds the R-, G- and B-component to the colors-vector. These lines are added at the end of the function add() in ”geomodtest.cpp”. The rest of the function is the same as in [25], so I will not show it here.

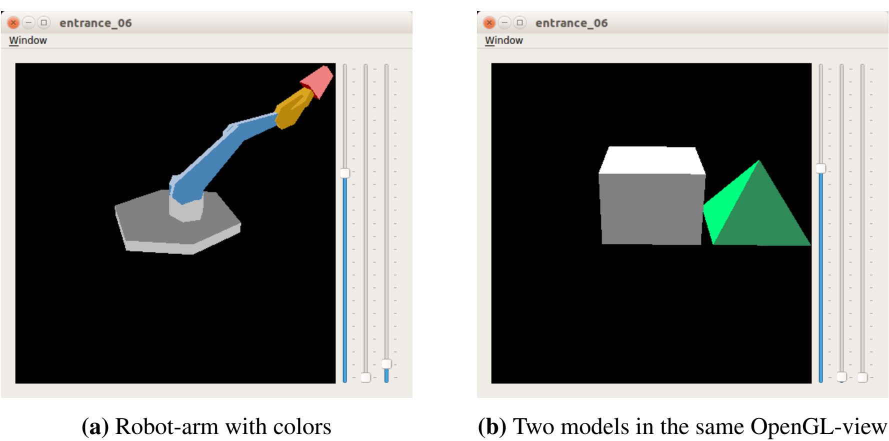
```
1 colors.push_back(currentRegion->getColorR());
2 colors.push_back(currentRegion->getColorG());
3 colors.push_back(currentRegion->getColorB());
```

The colors-vector is then used when the models are drawn in the function paintGL() in ”glwidget.cpp”. This function is defined in [25]. Here I commented out the line that draws the models, and replaced it with the following:

```
for(int i = 0; i < m_model.vertexCount(); i = i + 3) {
    m_program->setUniformValue(m_colorPosLoc,
                               QColor(m_model.colors.at(3*i),
                                      m_model.colors.at(3*i+1),
                                      m_model.colors.at(3*i+2)));
    glDrawArrays(GL_TRIANGLES, i, 3);
}
```

The for-loop loops through the buffer with a step-size of 3. This buffer is the one we filled with points in the constructor in ”geomodtest.cpp”. Lines 2 - 5 get the RGB-component for the current surface stored in the colors-vector, and tells OpenGL that it should be used when drawing the next triangle. For each iteration, glDrawArrays() in line 6 gets the next three points in the buffer and draws the triangle. The resulting OpenGL-view can be seen in figure 7.10a. Comparing this with the camera-view in figure 7.5a we see that the parts are drawn with the correct colors.

Figure 7.10b show the result of drawing two models with different colors. Both of them are drawn correctly, with the right color. The pyramid is moved in the main program to prevent it from being enclosed by the cube, and not drawn.



(a) Robot-arm with colors **(b)** Two models in the same OpenGL-view

Figure 7.10: Models with colors

The models are now drawn with colors, but the shading has disappeared. This might not be easy to see in the still image, but it’s evident when the view is rotated. This comes from the fact that the colors are now set uniformly across the whole surface. Earlier the green color was set through a shader-program, but this is not used in the solution above. OpenGL calculates the shade of a given surface using a fragment-shader. This is a small

program resting on the GPU, and it's called each time a surface is drawn. In our code, this program is stored in a variable called ”fragmentShaderSource”. It's loaded into the GPU at the beginning of the function intializeGL() in ”glwidget.cpp”. The original code for the shader can be found in [25]. This code is used as a template for the final version of the shader-program shown below.

```
1  static const char *fragmentShaderSource =
2      "uniform highp vec3 colorIn;\n"
3      "varying highp vec3 vert;\n"
4      "varying highp vec3 vertNormal;\n"
5      "uniform highp vec3 lightPos;\n"
6      "uniform highp vec4 fragColor;\n"
7      "void main() {\n"
8      "   highp vec3 L = normalize(lightPos - vert);\n"
9      "   highp float NL = max(dot(normalize(vertNormal), L), 0.0);\n"
10     "   highp vec3 color = colorIn;\n"
11     "   highp vec3 col = clamp(color * 0.2 + color * 0.8 * NL, 0.0, 1.0);\n"
12     "   gl_FragColor = vec4(col, 1.0);\n"
13     "}\n";
```

Lines 2 - 6 define variables used in the main()-function starting in line 7. The calculations inside the main()-function are the same as in [25], but I have here added a variable called colorIn. This allows us to set the input-color, instead of the previously hard-coded green. colorIn is a 3D-vector containing the RGB-components of the surface to be drawn. The shader-program will then calculate the color-gradient of this color and apply it to the current surface. The input-arguments to the function setUniformValue() had to be changed for this to work. This is the same function as the one we used to set the color above, and it's located inside the for-loop in the function paintGL() in ”glwidget.cpp”. The new code is as follows:

```
1  m_program->setUniformValue(m_colorPosLoc,
2                             QVector3D(m_model.colors.at(3*i)/255.0,
3                                       m_model.colors.at(3*i+1)/255.0,
4                                       m_model.colors.at(3*i+2)/255.0));
```

The second input-argument beginning in line 2 is now changed from a QColor to a QVector3D. QColor returns the normalized RGB-components, while QVector3D returns the components directly. This means that they have to be normalized by dividing each component by 255. This new code yields the result shown in figure 7.11. The difference between this figure and the one in 7.10a is not obvious when looking at the still image. It's easier to see when the view is being rotated, but looking closely at the bottom plate it's possible to see a difference.

7.1.5 Drawing edges in the system

Some models in the main program are created using edges only. The current implementation only draws surfaces, so these models will not be drawn at all. We now want to extend

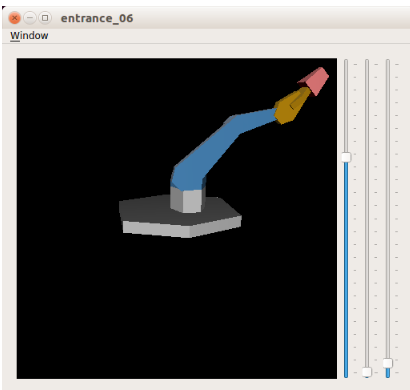


Figure 7.11: Robot-arm with shading

the implementation to draw edges as well as the surfaces. I mentioned earlier that the points used to create the triangular surfaces in OpenGL are stored in a buffer. This buffer is populated through the function add(), which is called by drawTriangle(). The code that actually draws the models specified in the buffer is located in paintGL() in ”glwidget.cpp”. This code was shown in the previous chapter, so I will not show it again here. The main idea is that all points in the buffer are looped through with a step size of three, the correct color is set and the triangles drawn. The flag ”GL_TRIANGLES”, is used to tell OpenGL that triangles should be drawn. The drawing-function, glDrawArray(), then gets three and three points from the buffer and draws the triangles specified by these points.

An edge is defined by two points. Let us assume that we add an edge to the buffer in the current implementation. When drawing, this will cause OpenGL to get the two points defining the edge, plus a third point from the next edge or surface, and draw these as a triangle. The buffer only consists of points, and it has no way of determining whether the next points define an edge or a triangle. This means that we cannot add the edges directly to the buffer when we find them. I will now try a solution where the buffer is split at an index. The first part of the buffer will contain points defining triangles, while the rest defines edges. This allows me to start drawing edges instead of triangles when this index is reached. The index will be stored in the variable buffersize. The following line sets this equal to the length of the buffer when all surfaces have been added:

```
buffersize = vertexCount();
```

The line above is added to the constructor in ”geomodtest.cpp”, after the outermost for-loop. At this point all surfaces have been added to the buffer, and we can start adding edges. The edges are added with the code below. It loops through all the models in the main program and then their Tgroups. These are the same for-loops as when adding surfaces. Lines 4 - 7 define the Tgroup, basis, offset and network for the current model. These variables are then passed to a newly created function called findVisibleEdges().

```
for(int i = 0; i < mdls->size(); i++) {
    Models::ModelData* mdl = mdls->at(i);
    for(int j = 0; j < mdl->getnumber_of_Tgroups(); j++) {
        Tgroup* modelTgroup = mdl->getTgroupPModifiable(j);
        ExtBasis* basisP = modelTgroup->getBasisPModifiable();
        MathVec offsetP = basisP->O();
        GeomNet* networkP = modelTgroup->getNetwork();

        findVisibleEdges(networkP, basisP, offsetP);
    }
}
```

The function findVisibleEdges() is defined below. It begins by creating a vector of GeomNodes called edgesToDraw. This vector will contain the points defining the edges to be drawn. The for-loop beginning at line 4 loops through all the visible edges in the system using an iterator. Line 7 stores the current edge in the variable edge. Lines 9 and 10 rotate these points with the previously defined function rotateNode(), and adds the result to edgesToDraw. edgesToDraw and offset is then used as arguments to the function drawEdge().

```
void GeoModTest::findVisibleEdges(GeomNet *network, ExtBasis *basis,
                                  MathVec offset) {
    std::vector<GeomNode*> edgesToDraw;
    for(std::list<GeomEdge*>::iterator e = network->getEdgesBegin();
        e != network->getEdgesEnd(); e++) {

        GeomEdge* edge = (*e);

        edgesToDraw.push_back(rotateNode(basis, edge->headNode()));
        edgesToDraw.push_back(rotateNode(basis, edge->tailNode()));
    }
    drawEdge(edgesToDraw, offset);
}
```

The implementation of drawEdge() is shown below. Line 2 defines a ”dummy” normal-vector. This has earlier been used to describe the orientation of the given surface. We are now drawing edges, so this vector doesn’t affect the result. It has to be defined here because the function add() expects a normal-vector as one of its input-variables. The for-loop starting at line 3 loops through the points defining the edges with a step-size of two. The head- and tail-nodes of the current edge is stored in the variables pointVecHead and pointVecTail in lines 4 and 5. Lines 6 - 13 add the x-, y- and z-coordinates, with the correct offsets, to the buffer using the add()-function.

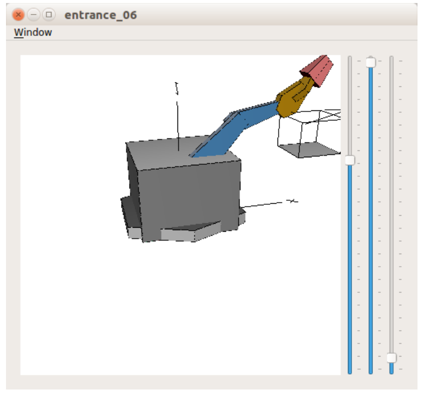
```
void GeoModTest::drawEdge(std::vector<GeomNode*> edges, MathVec offset) {
    QVector3D n = QVector3D(1, 1, 1);
    for(int i = 0; i < edges.size(); i = i + 2) {
        IDMthVec* pointVecHead = edges[i]->getModablXyzP();
        IDMthVec* pointVecTail = edges[i+1]->getModablXyzP();
        add(QVector3D(pointVecHead->get_x() + offset.get_x(),
                      pointVecHead->get_y() + offset.get_y(),
                      pointVecHead->get_z() + offset.get_z()),
                      n);
        add(QVector3D(pointVecTail->get_x() + offset.get_x(),
                      pointVecTail->get_y() + offset.get_y(),
                      pointVecTail->get_z() + offset.get_z()),
                      n);
    }
}
```

The buffer is now populated as wanted. Points defining the surfaces are stored up until the index buffersize, and points defining the edges are stored in the rest of the buffer. The next step is to start drawing edges at the correct time, which is achieved with the following code:

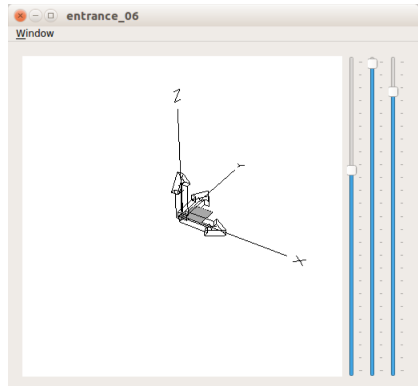
```
for(int i = 0; i < m_model.buffersize; i = i + 3) {
    m_program->setUniformValue(m_colorPosLoc,
                               QVector3D(m_model.colors.at(3*i)/255.0,
                               m_model.colors.at(3*i+1)/255.0,
                               m_model.colors.at(3*i+2)/255.0));
    glDrawArrays(GL_TRIANGLES, i, 3);
}

m_program->setUniformValue(m_colorPosLoc, QVector3D(0, 0, 0));
for(int i = m_model.buffersize; i < m_model.vertexCount(); i = i + 2) {
    glDrawArrays(GL_LINES, i, 2);
}
```

The first for-loop drawing the triangles are almost identical to the one shown earlier. The difference is that it now stops at the variable buffersize. The code inside this for-loop is the same as before. Line 9 sets the color of the edges to be drawn to black. The second for-loop starting in line 10 loops through the rest of the buffer, from buffersize, with a step-size of two. In line 11, the flag ”GL_LINES” is used in the function glDrawArrays(). This tells OpenGL that we now want to get two and two points from the buffer and draw the result as an edge. Figure 7.12a shows the OpenGL-view with the new implementation. Here, all models have edges around them, as well as a coordinate system and a cube to the right. These are the same lines as in the camera-view, so the edges are drawn correctly. Figure 7.12b shows the OpenGL-view with only the coordinate system. This shows the edges more clearly. I have here changed the background to white to make the edges visible.

In figures 7.12b and 7.12a we see a grey surface at the bottom of the cube defining paths and in the coordinate system. These are not visible in the camera-view, but they are present there as well. They are created as white surfaces where the user can create new paths for the robot-arm. The shading added earlier makes them appear grey when they are viewed



(a) All models in the system with edges



(b) Coordinate system

Figure 7.12: Models with edges

from an angle. Rotating the view, and seeing them from directly above makes them turn white and disappear.

The library created in this section draws the models in the main program using OpenGL. The models are transferred from the main program, and drawn with the correctly shaded color, rotation, and offset. Figure 7.13 shows all the models currently in the system in one OpenGL-view. The cube, pyramid, and coordinate system have been moved, and the robot-arm rotated, in the main program. These actions are correctly depicted in the OpenGL-view. We also see that the z-buffering works as intended. Parts of models are hidden by others, while the surfaces closest to the camera are shown completely. This can easily be seen in the coordinate system and at the top of the pyramid. The coordinate system is moved in front of the other models and drawn completely. The top of the pyramid is drawn, while some of the sides are hidden by the surface intersecting it. We also see that all the edges are drawn correctly. The black edges easily mark the intersection between two neighbouring surfaces, as can be seen in the grey cube. The edges have the same coordinates as the end of the surfaces. I would have anticipated that the z-buffering would cause the edge and surface to fight for visibility, but it seems that OpenGL automatically draws the lines on top. This is a good feature for us, because it's exactly what we want. When discussing this with my supervisor, we wondered if the edges are drawn on top because they are drawn last. I tested this by reversing the order of the drawing, but the result was still the same. This was tested so that later students are aware of this standard in OpenGL.

The edges easily let the user distinguish intersecting surfaces with the same color. But this property is more important when recreating model from images described in the previous chapter. Two neighbouring surfaces might have the same color after being recreated. If these have the same orientation it might be difficult for the program to tell them apart. The edges easily allow us to determine that there are two surfaces, and if this corresponds to

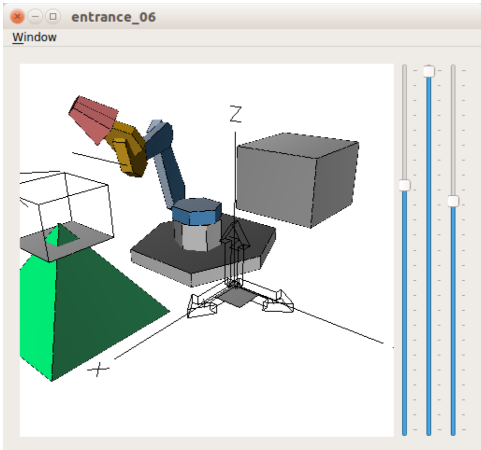


Figure 7.13: Several models with shade and z-buffering

the surfaces seen in the input-images.

7.2 OpenGL and further development

The OpenGL-library draws the models in the system correctly, but there is still work to be done. This section will discuss improvements and further development regarding OpenGL.

7.2.1 Drawing edges

The original camera-view doesn't have z-buffering, so the implementation in the drawing-algorithm determines which edges to draw. The programmer here has an option of drawing a dotted line if an edge is hidden behind a surface. This allows the camera-view to show the full geometry of the model from a single view-point. An example of this is seen in figure 7.14. We clearly see the geometry of the model without rotating it. This functionality is currently not available in the OpenGL-view. Adding the hidden edges, dotted, will cause the z-buffering not to draw them. One possible solution is to give the dotted edges coordinates closer to the camera. The edges will then have to be scaled down to look like they are inside the model. Only changing the coordinates would make the lines appear larger than desirable due to the projection of the camera. Such an implementation will require time. The hidden edges have to be found and scaled down with the correct amount. The dotted edges also have to be updated each time the view is rotated. Another possible solution is to adjust the alpha-value assigned to the visible surfaces. Making the surfaces partially transparent might show the edges hidden behind. The question here is whether the z-buffering draws these edges or not.

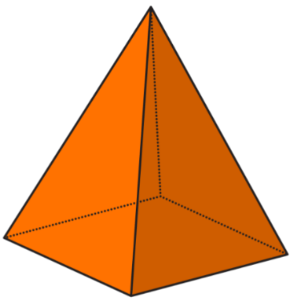


Figure 7.14: Showing hidden edges in pyramid

7.2.2 Drawing points

The user might be interested in drawing single points. Points can be drawn using the same approach as for edges described above. First, add the points to be drawn at the end of the buffer, store the index of the first point to be drawn, and then tell OpenGL where to start drawing points. The flag ”GL_POINTS” can then be used in the function glDrawArrays() to get a single point at a time and draw it. This process is similar to the one described for edges, so I will not go into more details here.

When the points are drawn, the user might want to name them using letters. Drawing letters in OpenGL seems to be harder than anticipated. There are no OpenGL-functions available for drawing text directly. This means that the letters have to be drawn manually. An example of drawing text manually using lines can be seen in the naming of the coordinate-axes in figure 7.13. Here we see the letters X and Z. These are created using edges in the library for the coordinate system. Coding letters manually takes time, but it's currently the only way to do it in OpenGL.

7.2.3 Drawing surfaces with the same coordinates

The next property I want to look at is drawing surfaces with the same coordinates. An example of this is shown in figure 7.15 where the models are shown from below. Looking closely, we see that the bottom of the triangle(red), cube(grey) and drawing-surface of the coordinate system(white) are mixed together. This is because OpenGL doesn't know which surface to draw when they all have the same coordinates. This problem doesn't affect the behaviour of the models, but the user should be aware that this might happen. Moving the surfaces so they don't overlap solves this small problem.

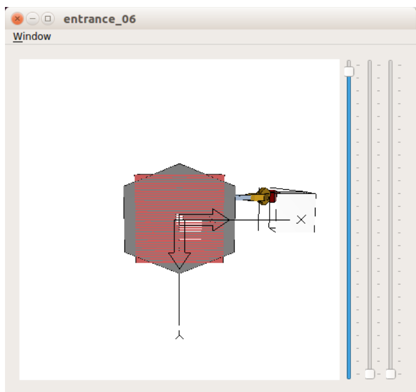
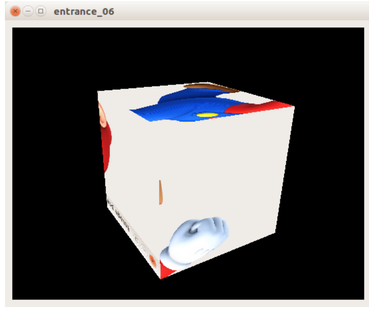


Figure 7.15: Surfaces with the same coordinates

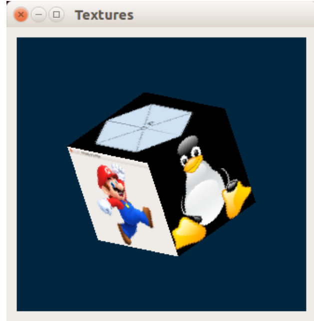
7.2.4 Adding Texture to the Models

We want to be able to recreate the surroundings as accurate as possible. One part of this is to add a representation of the background in the OpenGL-view. This is currently set to white, but we want to represent the actual background using images. My supervisor outlined a solution to this problem as follows: Create a sphere with a large radius, and add images of the background as textures inside this sphere. This would allow us to see the background, even when the view is rotated. The images depicted on this sphere should be updated when the vehicle have moved a given distance towards it. Updating the background will then show models that previously were hidden behind the sphere. This saves the effort of updating the background continuously. I will not implement the full solution here, but rather create a library showing that images can be added as texture to models. I will in this library add images on top of a cube. The library is located in ”DynamicLinkingTests/19_CubeWithTexture”, and based on the example in [26]. This shows a cube with texture making it resemble a dice. I have implemented the interface-functions and modified the example so that it can be linked in through the Tools-manager. The input-image has also been changed to an image of Super Mario [3]. The result can be seen in figure 7.16a. We see that the input-image is split up and shown on the different surfaces in the cube. One thing to note is that the OpenGL-view is different from the one shown earlier. This is the result of using the view already implemented in [26]. This section will only show that adding images as texture is possible, so I will not use time on recreating the same layout here.

In figure 7.16a the image is split up, showing only parts of it on each surface. The reason is that the example in [26] takes in one image with all the textures. This is practical when adding the same texture on the whole model, for instance when we want it to represent a certain material. Other times, this is not the case, so I will create another library that allows us to add one image for each surface. This library is based on the example in [27] and can be found in ”DynamicLinkingTests/20_CubeWithTexture2”. I implemented the interface-functions and added a few lines of code for the library to be linked in dynami-



(a) Textured cube based on the example in [26]



(b) Textured cube based on the example in [27]

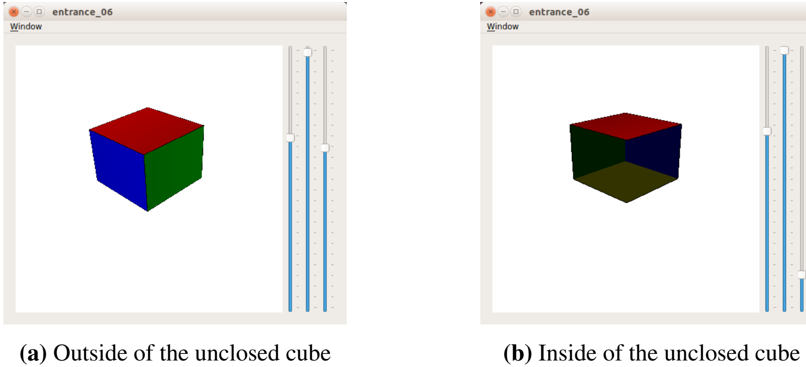
Figure 7.16: Drawing cubes with texture

cally. Running the code in the example directly opens a widget with six frames showing the same model. I changed this to only show one, and changed the input-images. The result can be seen in figure 7.16b. We see that each image is displayed on one surface in the cube.

This subsection shows that adding textures to models is possible in OpenGL. The main application in this project is to add a representation of the background. But it can also be used to create models with certain features. For instance adding windows and doors to a model of a house or windows, doors and wheels to a model of a car. These features can then be added as textures instead of modelling them, which is quicker.

7.2.5 Unclosed Models

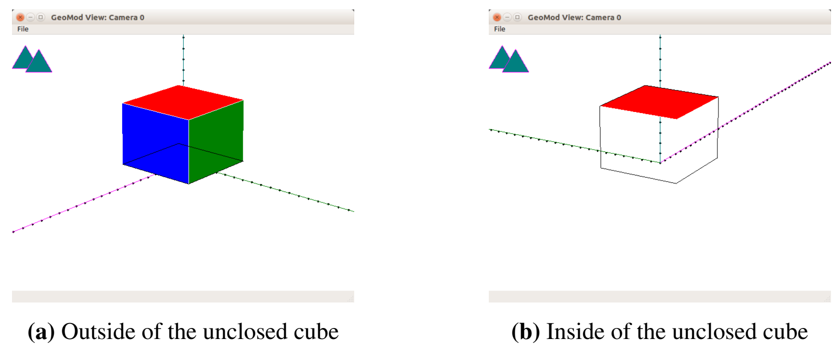
Drawing unclosed models is another property supported by OpenGL. This helps us when recreating models from images described in the previous chapter. I mentioned there that only parts of the model might be recreated from the input-images. OpenGL allows us to draw these partially recreated models directly. These models will then be drawn with different colors on the inside and outside. This can be seen in figures 7.17a and 7.17b where I have created a library with an unclosed model. The same unclosed cube can be seen in the camera-view in figures 7.18a and 7.18b. We see that the outside is similar to the OpenGL-view, but the inside is white. This makes it harder to determine whether the surfaces are there or not. The edges give us an idea, but it's harder than looking at the OpenGL-view. Figure 7.17b lets us see the defined surfaces right away. The darker colors also indicate that we are looking at the back-surfaces. This means that the old drawing-algorithm can handle unclosed models, but the visualization could be improved. This should be fairly easy to implement, but might not be necessary now that we have the functionality in the OpenGL-view. The model shown in these figures are implemented in the library ”DynamicLinkingTest/17_UnclosedCube”.



(a) Outside of the unclosed cube

(b) Inside of the unclosed cube

Figure 7.17: The unclosed cube with four surfaces using OpenGL



(a) Outside of the unclosed cube

(b) Inside of the unclosed cube

Figure 7.18: The unclosed cube with four surfaces using the old drawing algorithm

7.2.6 Adding Shadows

Another feature discussed with my supervisor is the possibility of adding shadows to the models. Initially I though OpenGL might have developed functions for this in the same manner as z-buffering. It turned out not to be the case. I found a few tutorials on how this could be achieved, but these require the programmer to calculate the shadows manually based on the light-position. 2D-models then have to be created and drawn. This task is quite big, so I will not try to implement it here. It should, however, be looked into at a later stage. Both my supervisor and I was a bit surprised that shadow-rendering isn't supported directly in OpenGL. After searching online I think it might come from the fact that there are a lot of different options in number of light-source and camera-projections available. Implementing a general shadow-mapping seems a lot harder than the user creating the custom shadow-mapping needed.

7.2.7 False Volumes

Drawing false volumes is a problem in both the original camera-view and the OpenGL-view. A false volume is a model where a point has been moved in such a way that the surfaces intersect one another. Figure 7.19a shows a cube with 8 corner-points. The same points are shown in figure 7.19b, but here the point A has been moved below the surface EFGH. In this model we have intersecting surfaces, which means that this can be seen as two volumes put together. The drawing-algorithms doesn't know this, so the drawing will be incorrect. This is not a common problem, but it should be looked into before the drawing algorithms are considered finished.

(a) Normal Cube

(b) False volume

Figure 7.19: Normal and false volume

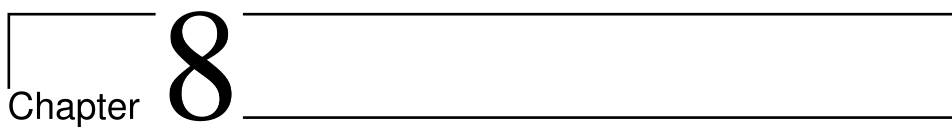
7.2.8 Using Code From the Old Drawing-algorithm

The OpenGL-view improves the visualization on several aspects, but there are still a few things to consider. A lot of functionality works with the camera-view, and not on the OpenGL-view. One example is changing the position of the camera. This is now hard-coded in a fixed position in the OpenGL-view. We can rotate the models in the OpenGL-view, but these changes are not reflected in the main program. A consequence of rotating the models instead of the view is that a complex scene might be rendered differently. It would then be beneficial to be able to rotate the position of the view itself. The main program has a control-panel that can change the position of the camera-view, but this will not work directly with OpenGL. It's possible to create a similar control-panel in the OpenGL-library, but it would be beneficial if the one in the main program could be used on both views. This would take some time, but using the same control-panel for both views is desirable. It would mean that the same code could be used for both views, which keeps the total code-base smaller. It would also free up space on the screen. If the user has four camera- and OpenGL-views opened this would mean four control-panels instead of eight. Other functionality working with the camera-view should also be adapted to work with the OpenGL-view.

7.2.9 OpenGL and the old drawing algorithm

This chapter has focused on developing the OpenGL-library, but my supervisor wanted me to shortly describe the drawing-process for the camera-view. This process is complex, and there are a lot of steps before anything actually is drawn in the view. RefreshAllViews::update() is called when the programmer wants to draw something. Here, there are a series of function-calls before the actual drawing is done: Determining the number of visible views, the position of the camera, drawing the triangle-logo and coordinate system and determining which surfaces to draw. After RefreshAllViews::update() is called, the following functions performs these actions: AllViews::update(), ViewRecManager::update(), Camera::update(), Camera::draw() and Tgroup::draw(). The last function, Tgroups::draw(), draws the models currently in the system. This makes sense since the transformation groups contain all the information needed to draw the models. All of these functions perform actions required for the drawing to work, but it might be difficult for the programmer to fully understand the process. It might be a good idea to test all the code, and comment out parts no longer needed to keep the total code-base smaller.

I have earlier mentioned that OpenGL has been tried implemented in the past. This code doesn't work as intended, and it's written using older versions of both OpenGL and Qt Creator. It has been left there for other students to look at if another attempt at OpenGL was implemented. I now have a library with OpenGL that works better than the old code, so it's no longer needed. I have therefore looked through the code and removed all parts of the old implementation.



Conclusion

My assignment is split into two parts. The first part focuses on implementations needed in the experimental process described in chapter 6. Here, libraries for the camera- and picture-view have been created. These allow the program to dynamically link in views when the need arises. A re-implementation of the BinPic-algorithm can be used to create binary geometric models from both colored and binary images. This can also be linked in dynamically when such a model is needed. The image-processing tool OpenCV has been installed and tested. It allows the program to create different binary images from the same input-image.

The second part of the assignment is developing a platform for recreating models. This allows further students to experiment with algorithms for building 3D-models from images. The whole process is described using the tools implemented in the first part of the assignment. The finished result should be a 3D-model shown inside the current view. This will then be compared to the model depicted in the input-images. I have here implemented OpenGL which allows us to check this more easily. OpenGL allows us to add light-sources which means that the resulting geometric models will contain reflections and shadows. The properties in the OpenGL-view can then be compared with the input-images to determine if the model has been recreated correctly.

The eventual goal is a fully autonomous system. Manual testing is a big part of developing such a system. Components have to be tested both separately and together to ensure that the code behaves as expected. Developing an easy-to-use user interface is emphasized in this master's thesis for easier testing and development. The set of subroutines behind the user interface will be the same eventually used by the autonomous system.

Bibliography

[1] Teknisk Ukeblad. Verdens første førerløse passasjerferge kan gå over en kanal i Trondheim. https://www.tu.no/artikler/verdens-forste-forerlose-passasjerferge-kan-ga-over-en-kanal-i-trondheim/363790.

[2] Teknisk Ukeblad. Norsk selskap bak verdens første autonome skip til kommersiell drift. https://www.tu.no/artikler/norsk-selskap-bak-verdens-forste-autonome-skip-til-kommersiell-drift/363811.

[3] Shigeru Miyamoto. Super Mario is registered as a trademark by Nintendo.

[4] Techopedia. C++ Programming Language. https://www.techopedia.com/definition/26184/c-programming-language.

[5] Chua Hock-Chuan. C++ Programming Language - Pointers, Reference and Dynamic Memory Allocation. https://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp4_PointerReference.html.

[6] Qt Creator. The IDE - Qt Creator. https://www.qt.io/ide/.

[7] Qt Creator. Qt Documentation - Qt Widgets. http://doc.qt.io/qt-5/qtwidgets-index.html.

[8] Qt Creator. Qt Documentation - Signals & Slots. http://doc.qt.io/qt-5/signalsandslots.html.

[9] Microsoft Visual Studio. Visual Studio Downloads. https://www.visualstudio.com/downloads/.

[10] Qt Creator. Qt - Download Open Source. https://www.qt.io/download-open-source/.

[11] Computer Hope. How to set path and environment variables in Windows. http://www.computerhope.com/issues/ch000549.htm.

[12] Qt Creator. Qt Documentation - QStringList Class. http://doc.qt.io/qt-5/qstring.html.

[13] Qt Creator. Qt Documentation - QImage Class. http://doc.qt.io/qt-5/qimage.html.

[14] Qt Creator. Qt Documentation - QBitmap Class. http://doc.qt.io/qt-5/qbitmap.html.

[15] ImageMagic Studio LLC. Command-line Processing @ ImageMagic. http://www.imagemagick.org/script/command-line-processing.php.

[16] ImageMagic Studio LLC. Download @ ImageMagic. http://www.imagemagick.org/script/download.php.

[17] How-To-Geek. What's the Difference Between JPG, PNG and GIF. https://www.howtogeek.com/howto/30941/whats-the-difference-between-jpg-png-and-gif/.

[18] GIMP. GNU Image Manipulation Program. https://www.gimp.org.

[19] StackOverFlow. Using GIMP as server on Windows. http://stackoverflow.com/questions/9001783/using-gimp-as-server-on-windows.

[20] StackOverFlow. Qt and image processing. http://stackoverflow.com/questions/5121913/qt-and-image-processing.

[21] Qt Creator. OpenCV with Qt. https://wiki.qt.io/OpenCV_with_Qt.

[22] Wikipedia. OpenGL. https://en.wikipedia.org/wiki/OpenGL.

[23] Qt Creator. Qt Documentation - QOpenGLWidget Class. http://doc.qt.io/qt-5/qopenglwidget.html.

[24] StackOverFlow. How do I render a triangle in QOpenGLWidget? http://stackoverflow.com/questions/31522637/how-do-i-render-a-triangle-in-qopenglwidget.

[25] Qt Creator. Qt Documentation - Hello GL2 Example. http://doc.qt.io/qt-5/qtopengl-hellogl2-example.html.

[26] Qt Creator. Qt Documentation - Cube OpenGL Es 2.0 example. http://doc.qt.io/qt-5/qtopengl-cube-example.html.

[27] Qt Creator. Qt Documentation - Textures Example. http://doc.qt.io/qt-5/qtopengl-textures-example.html.

[28] OpenCV. Releases. http://opencv.org/releases.html.

[29] Youtube. Setting up OpenCV in Visual Studios. https://www.youtube.com/watch?v=l4372qtZ4dc.

[30] Wikipedia. BMP file format. https://en.wikipedia.org/wiki/BMP_file_format.

Installing and testing OpenCV

OpenCV is an image-processing library that can be used directly through C++-functions. The library is not a part of the Qt Creator standard libraries, so it has to be downloaded and installed. This appendix goes through the installation on Ubuntu and Windows. A test-library will also be created to verify the installation.

A.1 Installing and testing OpenCV on Ubuntu

The OpenCV-library was downloaded from [28], and unzipped in the terminal using the following line:

```
unzip Downloads/opencv-3.2.0.zip
```

This unzips the files in a folder called ”opencv-3.2.0” in my home-directory. The following lines navigates into the folder ”opencv-3.2.0”, creates a new folder called ”release” and navigates into this.

```
1 cd opencv-3.2.0
2 mkdir release
3 cd release
```

The next step is generating a make-file. This is done in the terminal using ”cmake”:

```
cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local ..
```

The last step is to compile and install the library:

```
1 make
2 sudo make install
3 sudo ldconfig
```

To verify that the OpenCV-library is correctly installed I created a new library in ”DynamicLinkingTests/13_OpenCVTests”. This should use the OpenCV-functions imread() and imshow() to preview an image. I will not show the code for the whole library, but rather the implementations related to OpenCV. For the library to be linked in as a tool I implemented the interface-functions and created a new widget. I added a button called ”Add Image”. When pressed, this opens a file-browser similar to the one in the BinPic-library. It allows the user to browse through the folders on the computer and add images to the widget. A button called ”Test OpenCV” was also added. This calls a function called openOpenCV() when clicked, and has the following implementation:

```
void OpenCVTool::openOpenCV() {
    int columnCheck = treeWidget->currentColumn();

    if(columnCheck != -1) {
        std::string pluginPath = treeWidget->currentItem()->
                                    text(1).toStdString();
        Mat img = imread(pluginPath);
        imshow("Image", img);
    }
}
```

Line 2 gets the index of the currently selected image and stores it in the variable columnCheck. The code inside the if-statement in line 4 is executed if an image in the OpenCV-widget has been selected. Lines 5 - 6 gets the path to the current image as a std::string, and stores it in the variable pluginPath. This path is then used as input to the OpenCV-function imread() in line 7. The resulting image is then stored as the type Mat in the variable img. Mat is a n-dimensional array used by OpenCV to store images. Line 8 uses the OpenCV-function imshow() to show the image in a new widget. The implementation for showing the image using OpenCV-functions is now finished, but we have to tell the library where OpenCV can be found before we compile it. This is done by adding the following to the pro-file:

```
unix:!macx {
    LIBS += "../../../GeoMod/libMaxLib.a"
    INCLUDEPATH += /usr/local/include/opencv
    LIBS += -L/usr/local/lib \
        -lopencv_core \
        -lopencv_imgproc \
        -lopencv_highgui \
        -lopencv_ml \
        -lopencv_video \
        -lopencv_features2d \
        -lopencv_calib3d \
        -lopencv_objdetect \
        -lopencv_flann
}
```

Line 3 includes the path to the OpenCV-files, which tells the library where these files are. Lines 4 - 13 includes the files needed for OpenCV to work. The code above is specific for

Linux, so they will be different on Windows and OS X. Line 1 in the code above specifies that these files only should be included when the computer uses Linux.

Next, we have to include the OpenCV-files needed in our library. This is done in the file ”opencvtool.cpp” with the following two include-statements:

```
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/core/core.hpp>
```

The result of running the code and selecting the image of the Linux logo is shown in figure A.1. We see that the image is shown correctly inside a new OpenCV-widget. This means that the OpenCV-functions imread() and imshow() works, which tells us that OpenCV was installed and included correctly.

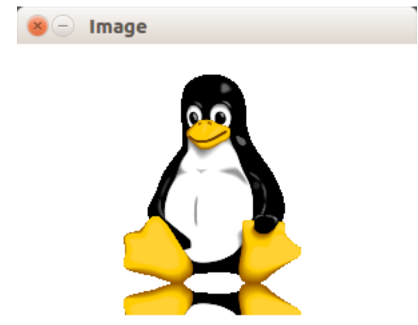


Figure A.1: Showing the Linux logo in a OpenCV-widget

A.2 Installing and testing OpenCV on Windows

The installation has been shown to work on Linux, so the next step is to install it on Windows. Windows in the main operating system for the development of the GeoMod-program, so it’s important that OpenCV works on this platform as well. I have Windows 10 on my computer, so I followed the tutorial in [29]. This explains the process of installing and running OpenCV alongside Microsoft Visual Studio, but it also works for Qt Creator. There are two ways to install OpenCV on Windows. The first is to use the library downloaded directly without compiling it. The second option is to configure the library in a similar manner as on Linux, using ”cmake”. We have no need to configure the library in this project, so we will use the library downloaded directly. This is an easier solution, which is good because the software has to be installed on several computers.

The OpenCV-files are downloaded from [28]. On Windows we can unzip these directly by double-clicking the downloaded file. A window asking for a target directory will then be opened. I chose to unzip the files directly on the C-drive. This creates a folder called

”opencv” in ”C:\” with the library-files. For Qt Creator to be able to find the OpenCV-library, I added them to the library’s pro-file. This code looks for the OpenCV-files on the C-drive, so it’s important that the files are unzipped there. The code is as follows:

```
win64 {
    LIBS += "../../../GeoMod/MaxLib.lib"
    INCLUDEPATH += C:/opencv/build/include
    LIBS += -LC:/opencv/build/x64/vc14/lib/ \
        -lopencv_world320
}
```

Line 3 adds the path to the library using INCLUDEPATH. Lines 4 - 5 then adds the library-files needed on Windows. Line 1 specifies that this file only should be included when the user has a 64-bit Windows-version. The same library should be included on a 32-bit Windows, so I have added similar code for this case.

For the library to compile without errors we have to add the OpenCV-folder to the list of system variables. The process of adding system variables on different version of Windows is shown in [11], so I will not go through it here. This tutorial is for an English version of Windows, but the steps are the same for a Norwegian version. The folder added here should be on the following form: ”C:\opencv \build**OS** **MSVC** \bin”. **OS** should be replaced by the Windows-version the user is running. I have a 64-bit Windows, so for me **OS** is replaced by ”x64”. **MSVC** should be replaced by the Microsoft Visual Studio-version that’s installed. I have Microsoft Visual Studio 2015, so ”vc14” is added here. The system variable added my computer is then ”C:\opencv\build\x64\vc14\bin”. The computer has to restarted for the changes to take effect.

The library can now be compiled without error. Selecting the Linux logo and opening the preview yields the same result as on Linux, the widget shown in Figure A.1. This shows that the installation of OpenCV was successful on Windows as well.

Short introduction to BMP and BMX

The original BinPic-algorithm only handled the two formats BMP and BMX. The algorithm now handles several other formats, but I want to give a short introduction to the two. Most of the other formats described in this master's thesis are widely used today, and the user probably knows a bit about them. The BMP- and BMX-formats are not that common today, so a short introduction might be desirable.

B.1 The BMP-extension

An image is stored in the BMP-format is an uncompressed raster image [30]. BMP-images are also known as bitmap image files, which is the expected input to the BinPic-algorithm. The format is used to store digital bitmap-images that can be displayed independently of the device. The pixels in a BMP-image is stored in a rectangular grid. In a binary image this grid would consist of zeros and ones. This makes the BMP-extension easy to work with when converting it to a model. I suspect this is the main reason why this format was chosen when the BinPic-algorithm was developed.

B.2 The BMX extension

Searching for the BMX-extension online yields few results. Mostly the resulting websites describes how to open the format and not about the format itself. This indicates that the format isn't used that much anymore. The sources I found describes BMX-files as raster image files which is consistent with the BMP-format. BMX-images are also referred to as Image Library Files, or Alpha Five Image Library Files.

The BinPic-algorithm handles BMX-images, so I assume that the BMX-images are uncompressed as well.

General comments

This appendix describes some general comments about the libraries and code developed during this master's thesis.

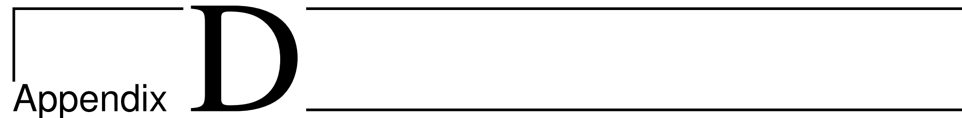
C.1 Dynamic libraries

Quite a lot of the dynamic libraries have been developed during my master's project and master's thesis. These contain different models like cubes and pyramids, views like the camera- and picture-view and tools like OpenCV and the BinPic-algorithm. Some of these libraries contains old code rewritten to work in the current program. Others are developed from scratch. It might not always be self-explanatory what the libraries do or which profile to include. I have therefore written Readme-files for all the libraries explaining about the libraries themselves and how to use them.

C.2 Generally about the code

A lot of code has been implemented during this master's thesis. Not all of this is shown, but I have tried to include code-snippets relevant to the current chapters. I have shown the code giving errors and the implementation to solve this. I have also tried to re-use as much code as possible, but this is sometimes difficult in such large projects. Some developed functionality might exist in other part of the program, but spending time looking for functions that might not exist takes time. I have therefore discussed different solutions with my supervisor before implementation. He has been a part of the development since the beginning, and has a better understanding of functionality that might already exist. I have more than once implemented functionality later found in other classes. To keep duplicate functionality at a minimum, I have deleted my implementation and used the old one.

The code developed in my master's project and master's thesis is tested on native Windows 10, OS X Sierra and Ubuntu 16.04. The code without OpenCV is also tested on a virtual version of Windows 10 and Ubuntu 16.04, so the code should work on most computers.



Risk assessment

NTNU	Hazardous activity identification process	Prepared by	Number	Date
HSE		HSE section	HMSRV2601E	09.01.2013
		Approved by		Replaces
		The Rector		01.12.2006

Unit: *(Department)* **Date:** 18.1.2017

Line manager:

Participants in the identification process (including their function)**:**

Short description of the main activity/main process: Master project for student Tony Gjendahl. A Control System for Autonomous Vehicles.

Is the project work purely theoretical? (YES/NO): Yes *Answer "YES" implies that supervisor is assured that no activities requiring risk assessment are involved in the work. If YES, briefly describe the activities below. The risk assessment form need not be filled out.*

Signatures: *Responsible supervisor:* Tom Fjeldaas *Student:* Tony Gjendahl

ID nr.	Activity/process	Responsible person	Existing documentation	Existing safety measures	Laws, regulations etc.	Comment
1	Software development and report					