Moritz Münch

Three-dimensional Geometric Models from Pictures

An extension for the program GeoMod

Master's thesis in Engineering and ICT
Supervisor: Sven Fjeldaas
April 2019

Master's thesis

NTNU
Norwegian University of Science and Technology

NTNU
Norwegian University of
Science and Technology

Moritz Münch

Three-dimensional Geometric Models from Pictures

An extension for the program GeoMod

Master's thesis in Engineering and ICT
Supervisor: Sven Fjeldaas
April 2019

Norwegian University of Science and Technology

NTNU
Norwegian University of
Science and Technology

For Tormod, who was a dear friend. You will always be remembered

NTNU

Faculty of Engineering
Department of Mechanical and Industrial Engineering

Date

MASTER'S THESIS AUTUMN 2018
FOR
STUD.TECHN. Moritz Münch

Three-dimensional Geometric Models from Pictures

Tre-dimensjonale geometriske modeller fra bilder

There is an ongoing project at NTNU aiming at generating three-dimensional geometric models from ordinary digital pictures automatically. Software is written in the C++ programming language. The candidate is expected to contribute to the development needed in this project.

It is reasonable to start in ordinary digital picture processing. The commonly available software library "Open CV" appears to be suitable. The task is to find operations that can identify and mark:

- Objects against their background.
- Planar and nearly planar parts of surfaces.

Initial experiments may well concentrate on pictures of planar polyhedra made of cardboard.

Objects and planar parts are marked as monochrome regions. Pictures containing selections of regions will be reduced to binary pictures.

The project already has satisfactory contour tracking algorithms for binary pictures. Contours of regions still need a simplification into chains of straight-line segments, but other candidates will handle the corresponding development.

The contours will appear as flat polygons in the YZ-plane of a three-dimensional geometric modeler. There are now interesting challenges in developing algorithms for:

- Joining a collection of polygons into flat networks having "nodes", "lines", and "regions".
    - Corners from different polygons, having nearly equal coordinates, are represented by a single "node".
    - Parallel, and nearly parallel, sides in different polygons are combined into "lines".
    - Areas of different polygons end up as regions.
- Setting the third vertex coordinates, here X, to values matching the scene photographed. Networks will then appear as open polyhedra, having "vertices", "edges", and "faces"

The third coordinate values are not directly available. In some cases shading in the original picture hold information on the orientation in space of more or less planar surface parts. In other cases it is known a priory that some surfaces are horizontal, others are vertical. In more general cases, angles between surfaces are known. Computing intersection between these planes can give sets of three-dimensional coordinates. Measurements of essential positions may occasionally be available.

Several parts of the concept for building three-dimensional geometric models from pictures will call for iterative approaches.

Source code with underlying mathematics and principles is made available to the candidate. This contains business secrets, and is to be handled accordingly. A further distribution is not permitted. The source code and the corresponding computer program shall be used to solve the given task only. The work of the candidate gives no basis for claims directed towards the owner of the code, in particular not claims on limitations, or claims for compensations.

This description of concepts, tasks, and limitations given above is to appear among the first pages of the work handed in by the candidate.

Contact:

At the department (supervisor, co-supervisor): Professor Sven Fjeldaas

From the industry:

Vedlegg: Kontraktsmal

Summary

The aim of this master thesis was to create a form detection in images. Several different filters were used to help identify and mark these forms against their background, based on colour and intensity. The filters enabled the program to identify more complex shapes with an increasing amount of points along their contour. The shape detection was implemented in a Python environment using the program OpenCV. The program OpenCV provides a lot of useful functions for detecting shapes, creating a hierarchy and drawing a shapes outline. In order to calculate a better approximation of a shape, a method that eliminates points that are in near proximity was implemented. This method would improve the result for further modification and preserve precious memory. Once a shape was detected, it was ported over to a program called GeoMod. The program GeoMod is developed by Professor Sven Fjeldaas at NTNU and has been the basis for several master theses. The program allows shapes to be linked in dynamically. A method to port shapes detected by the python part into the GeoMod program was implemented in the form of a simple read-write functionality.

Once a shape was read into the GeoMod program, it could be modified. The goal was to make the shapes have depth in the z-direction. Here, creating three-dimensional models of cubes were the main focus. General shapes such as rectangles and hexagons acquired their depth by using normal-vectors. This method could give even complex shapes a general, calculated depth and make them appear three-dimensional in the GeoMod View environment. Two methods were implemented to create a relatively accurate three-dimensional model of a cube, based on a two-dimensional image of a set view of a cube. One of the methods was dubbed “four-points” method, requiring only four points to create such a three-dimensional model. This method was less error-prone than the other standard method, but it could lead to inaccuracies for length between nodes in the model. The results of the other, standard method were more accurate, but edges between nodes could be drawn incorrectly for some models due to an error in the relation between nodes.

This thesis produces a way to identify shapes in a two-dimensional image, calculate a good approximation of its contour and port it to the GeoMod program. A three-dimensional model of that contour can then be calculated and shown in the camera-view of the GeoMod program.

Sammendrag

Denne masteroppgaven implementerer form deteksjon på bilder ved hjelp av programmet OpenCV og programmeringsspråket Python. OpenCV programmet har en god del hjelpefunksjoner, som gjør det enkelt gjenkjenne former på et bilde. Flere filter hjelper til med å separere former fra sin bakgrunn. I starten var det kun mulig å gjenkjenne veldig enkle omriss av former, som firkanter og femkanter. Per dags dato, ved hjelp av justering av de implementerte filtrene, er det mulig å gjenkjenne relativ komplekse former. Nivå av detalj kan selv velges av brukeren.

Dersom en form har blitt gjenkjent, kan den bli portet over til programmet GeoMod som lever i et C++-miljø. Programmet GeoMod er et program laget av Profesor Sven Fjeldaas. Dette programmet har eksistert i mange år, og mange masterstudenter har jobbet med det tidligere. Programmet støtter tegning av figurer i en kameravisning. Former som ble gjenkjent i Python-miljøet blir lest inn i GeoMod programmet ved hjelp av tekstfiler. Når formen er lest fra fil, kan den modifiseres. En sentral del av masteroppgaven var å implementere en støtte for å gjenkjenne kuber i bilder og å gjøre disse kubene tredimensjonale i GeoMod kameravisningen. Dette ble gjort ved at ulike beregninger ble utført på nodene som ble funnet av Python programmet. En relativt nøyaktig modell av en kube kunne nå bli tegnet opp i tre dimensjoner og bli påvirket i kameravisningen. To metoder for å finne denne tredimensjonale modellen ble implementert. Ved siden av en generell metode, ble det implementert en metode som ble kalt for ”Four-points” metode. Denne metoden tar kun utgangspunkt i fire noder som brukes til å beregne seg fram til en full kube. Fordelen med denne metoden var å minske sannsynligheten for feil med kanter, men en ulempe var at avstanden mellom enkelte noder kunne være litt lengre eller kortere enn på det originale bilde. En metode for å gi generelle former en dybde ble også implementert. Normalvektoren mellom tre punkter beregner retningen til dybden og gjør formen tredimensjonal.

Zusammenfassung

Diese Masterarbeit implementiert eine Formerkennung in Bildern. Das Programm OpenCV und die Programmierungssprache Python ermöglichen mittels eingebauter Funktionen die Detektion von Formen in Bilder. Anfangs wurden lediglich einfache Formen wie Vier- oder Fünfecke relativ problemlos gefunden. Mit Hilfe von Filtern ist es nun möglich, auch komplexere Formen zu entdecken, beispielsweise auch solche, die ihrem Hintergrund farblich ähneln. Diese Formen werden durch eine ”read & write” Methode an ein Programm namens GeoMod gesendet.

Das Programm GeoMod wurde von Professor Sven Fjeldaas entwickelt. Jahrelang stellte dieses Programm die Basis für mehrere Masterarbeiten dar. Nachdem die Formen von einer Textdatei gelesen wurden, werden sie in das GeoMod Programm geladen. Mit diesem Programm können die Knoten der Formen geändert werden. Im Rahmen dieser Arbeit ist es das Ziel, zweidimensionale Bilder von Würfeln in dreidimensionale Modelle zu verwandeln. Um dies zu ermöglichen wurden zwei Annäherungen implementiert. Zum Einen eine Methode, die die Tiefe des Modells durch Vektoren berechnen kann. Zum anderen eine Methode, die nur vier Punkte benötigt, um ein dreidimensionales Modell von einem Würfel zu erstellen. Zusätzlich ermöglicht eine weitere Methode, die Tiefe für generelle Formen zu berechnen. Diese Tiefe wird durch Normalenvektoren berechnet. Die Kameraansicht des GeoMod Programm ermöglicht es, die berechneten Figuren in einem dreidimensionalen Raum zu zeichnen.

Preface

A program called GeoMod has been in existence for quite a few years now. It is a geometric modelling program created by Professor Sven Fjeldaas. It has also been the basis for multiple master thesis. Previously, creating models for testing and use in the program's environment has been a tedious task. Hundreds of lines of code had to be duplicated and modified to create the basis for a single model, and the models complexity increased rapidly when creating it in three-dimensions.

A new solution was sought, something that would automate this task. This is where this thesis objective comes in. This thesis' code had the ambition to detect shapes in images, which can be loaded directly into the GeoMod environment. Creating these shapes to be three-dimensional, would be optimal for creating obstacles and a route for navigation. But creating a three-dimensional model from a two-dimensional image as source is easier said then done. How would one go about extracting the depth from the picture? What assumptions can be made that can help create something three-dimensional? This project aims to make the assumptions necessary to create a good estimation for the depth of various shapes.

I would like to say a big thank you to my supervisor Sven Fjeldaas, who has always provided superb guidance and help throughout the time I have known him; first through long discussions in his office and then later over e-mail. I have learned a lot from his insight, patience and I am very grateful for his support.

Table of Contents

List of Figures

Chapter 1

Introduction

Today in the age of technology, object recognition is getting more and more influential and essential. Trends to equip cell phones with multiple cameras in order to be able to recognize objects and depth have started to take over the mobile phone industry. Unlocking your phone with your face has become a reality and is used by people all over the world. AI cameras can recognize objects in the real world and help with useful tasks based upon what is recognized. Not even mentioning the role it plays within the surveillance sector. For most people, object recognition has become a part of everyday life without them even realising it.

This project implements the foundations of object recognition. Recognizing simple shapes and drawing them up in a separate view are the basis for most object recognition used today, and that is also how this project started. Once the groundwork has been laid, more complicated operations can be implemented, and it can be scaled until it could eventually be compared to something similar that is used by the big technology companies today.

Creating a three-dimensional model from a two-dimensional image is a challenging task. All approaches used today to create three-dimensional objects, use multiple images, sonar or images that contain a distance in each pixel. Even humans have two eyes to easier estimate depth. But even when using only eye, humans can still estimate depth to a certain extend. This is due to prior knowledge about an object or its surrounding. The shadow created by a light source, the objects shape, the objects size or its proximity to other objects of a known size can give an estimation of depth in a single image. This information is often called “a priori” information, knowledge which exists about an unknown object due to previous experiences. By using this “a priori” information, estimations of an objects shape, size and depth can be made, and three-dimensional model can be created. This approach is used in this thesis to create cubes from images.

In this thesis this approach is used to create cubes from images. The GeoMod program is used as a geometric modelling tool. This program can display a shape in a three-dimensional space, and it can modify and interact with the model. A potential calculated approximation for a three-dimensional model can be viewed and modified with the help of

the GeoMod program. The program already has a way to create and load two- and three-dimensional models. This method was quite inconvenient and resulted in lots of code being copied. Recognising the shape in an image and creating it directly in the GeoMod program could help automate this process.

The aim of this thesis therefore is to automate the creation of shapes and three-dimensional models in the environment of the GeoMod program through the detection of shapes in an image.

Chapter 2

Tools and program development-platforms

In this chapter, the tools to develop the code for this thesis are introduced. In this thesis two programming languages were used, Python and C++. A brief introduction of the two languages are given below, as well as some relevant functionality that was used.

2.1 The programming language Python

Like C++, Python is a high-level programming language. It is relatively young compared to some other programming languages, as it was released in 1991. In a few words, Python is an object-oriented, functional, imperative and procedural language and it comes with a large library. The main philosophy behind its design is code readability, for which it has become famous for. Line indents and white-space make this language very easy to read and it is therefore often used as an entry language for beginners in programming. A few of its aphorisms are therefore:

"Beautiful is better than ugly"
"Simple is better than complex"
"Readability counts"
"Complex is better than complicated"

"..." [2]

2.2 The programming language C++

C++ is a much older programming language than Python. It was originally designed by Bjarne Stroustrup in 1984 and is also an object-oriented language. Since C++ builds upon the programming language C, it uses many components from C. When it was first released,

it introduced new features that differentiated it to its C language origin such as classes, member functions and much more.

Pointers are a central part of the C++ language. Pointers let you access addresses in memory and manipulate the content of these addresses. Using pointers effectively will result in powerful programs with high efficiency and effectiveness. Drawbacks of not using them correctly may be memory leaks or unattainable code. Pointer are used frequently in the GeoMod program.

2.2.1 Headers and sources

Information that is re-used multiple places in a code, and which has to be exactly the same for each use scenario is placed in the header file. This saves both time re-copying already implemented information and it eliminates the chance for errors during duplication.

The header file tells the compiler about the function names, return types and parameters. The C++ header file is usually included in the source file with an include statement at the top. For the compiler, the include statement is substituted by the content of the file that is included in the include statement.

The source file contains the code functionality of the program. Using header and source files gives a clearer and more easily readable code. Source files can include multiple header files and therefore extend its functionality. GeoMod showcases this method of implementation really well. This short code snippet shows how a file in the GeoMod usually imports necessary functions.

```
1 //  GeoMod headerfiles, other directories:
2 #include "../../../../MaxLib/math/vec.h"            // To have access to the
      subroutines set and get for IDMthVec gravityCenter.
3 #include "../../../../MaxLib/net_G/geomnode.h"      // Geometric nodes.
4 #include "../../../../MaxLib/math/extbas.h"         // Extended basis.
5 #include "../../../../MaxLib/net_T/tgroup.h"        // Transformation group.
6 #include "../../../../MaxLib/allviews/all_views.h"
7 #include "../../../../MaxLib/displays/palette.h"
8 #include "../../GeoMod/path_des/geom_path_des/path.h"
```

Listing 2.1: Header example

2.2.2 Pointers

Pointers are a central part of the GeoMod program and the programming language C++. Pointers allow the access of addresses and they can manipulate the content at these addresses directly. Pointers contain an address pointing to a location in the memory where the data is stored, they do not contain the representation of data themselves. To declare a new pointer, in C++ the symbol is used. This line creates a pointer to a *double* and a *char*.

```
1 double *pointer1;
2 char *pointer2;
```

Listing 2.2: Pointer example

To create a pointer which points to a value in the memory, it can be done like this.

```
1 double temp = 3.14; //variable declaration
2 double *pointer3; // create pointer
3
```

```
4 pointer3 = &temp; // stores the address of temp at the address of the pointer
```

Listing 2.3: Pointer example with value

This pointer now points to an address in the memory which contains the value *3.14*. This value can be accessed by using **pointer3*. If the value of the pointer would be changed such as here in listing 2.4

```
1 *pointer3 = 2;
```

Listing 2.4: Changing the value

the content of the address would be changed to *2*, and the variable *temp* would also now have the value *2*.

2.2.3 Inheritance

Like other programming languages inheritance plays a big part in C++. Inheritance removes the need to duplicate code and enables the ability to reuse functions shared by different objects and classes. In programming the class that inherits lines of code is called the child class or subclass. The source class which provides the code that is inherited, is called the parent class or superclass. After inheriting, the child class has the same functionality as its parent class, and it can extend its functionality even further by defining its own functions. C++ allows inheritance from abstract classes or interface classes. Each child class that inherits from an abstract parent class has to contain the function of the abstract class and implement their functionality as well. An example of that is this code snippet taken from the GeoMod program:

```
1  public:
2      virtual PluginInterface* newInstance() = 0; // Implemented in 'cube1.h/.cpp' etc.
3      // Called from 'pluginfactory.h - instantiatePlugin_(..)
4      virtual void deleteInstance() = 0;          // Not tested yet.
5      // Called from 'pluginfactory.h - deletePluginStruct_(..)
6      // virtual QString Name() = 0;
7      virtual QString getName() = 0;      // Implemented in 'cube01_if.h/.cpp' etc.
8      virtual void forwardCentralP(Central *centP) = 0;     // Implemented in 'cube01_if.
       h/.cpp' etc.
9      virtual void forwardGlobalPointer(void (*functionPtr)(void)) = 0;     //
       Implemented in 'cube01_if.h/.cpp' etc.
10     virtual void forwardDatBsManP(DataBsManagerWidget *dbsP) = 0; // Implemented in '
       cube01_if.h/.cpp' etc.
```

Listing 2.5: Virtual functions from the pluginInterface class

2.2.4 Function overloading

Function overloading is another feature that C++ supports. Function overloading means that multiple functions in a class can have the same name. They are uniquely identified by what parameters they take as an input. The input decides which function implementation is used if the function has multiple implementations. The example below shows two functions with the same name. The first takes an int as input, the other takes a double as input. Calling the print function with a double as an argument, would invoke the function at line 6.

```
class printData {
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }
      void print(double  f) {
        cout << "Printing double: " << f << endl;
      }
};
```

Listing 2.6: Function overloading example

2.2.5 Recursive function

A recursive function is a function which calls upon itself during its execution [4]. These functions are common in Computer Science because it allows for very efficient programming with a minimum amount of code. Programming the Fibonacci sequence [5] with recursion is a very good example.

```
1 int fib(int n) {
2     if (n <= 2) return 1
3     else return fib(n-1) + fib(n-2)
4     }
```

Listing 2.7: Fibonacci sequence with recursion

Here the important line is line number 4, where the function calls itself again twice, to calculate the next Fibonacci number.

2.3 Development Tools

2.3.1 Qt

In order to compile, edit and write code for the GeoMod program, the Integrated Development Environment (IDE) Qt Creator [27] was used in combination with the programming language C++. The main reason for using the IDE Qt, is that it offers cross-platform development. This means that the program should be able to compile on macOS, Linux and Windows if the application Qt Creator is installed on those operating systems. In addition to cross-platform development, Qt offers a big library with useful tools for the developer. Its UI elements are today widely used in application for almost every modern OS imaginable.

To get Qt to work with a Windows PC on a free license was not as straight forward as one might think, so in the appendix an installation guide is included to help future readers. The creators of Qt are releasing continuously updates for its software, which is not always helpful as it has given a break in compatibility with the GeoMod program.

2.3.2 Jupyter

To compile and edit the Python part of the code, a program called Notebook was used. Notebook is part of an open source project called Project Jupyter [28]. It is free to use with

an easy layout and good support for many programming languages, especially Python. Notebook is a web-based interactive computational environment which uses the JSON-format to store its code. This also makes it quite easy to convert code into other formats. Since it is web-based it can be run in your preferred web-browser (Chrome in this thesis' case).

2.3.3 OpenCV

OpenCV (Open Source Computer Vision Library) [29] is an open source computer vision and machine learning software library. Originally developed by Intel, it is cross-platform and free to use. As with Qt it was important that the computer vision program used for this thesis is free to use and cross-platform compatible. Since OpenCV has a lot of algorithms for picture manipulation, in fact more than 2500, it is an effective tool for manipulating and analysing images. Especially the contour algorithms, which OpenCV implements have been instrumental in this thesis. OpenCV is being used by well-established companies such as IBM, Toyota and Google, which attests to the high quality of the program. This is very assuring that everything OpenCV provides is of the highest quality. The OpenCV community is very large and answers to possible questions can be found on various support websites. Since OpenCV is based on C and C++ it is very well optimised, which gives acceptable run-times even with large data on an average laptop.

2.3.4 Numpy

Numpy is a package extension for Python. It is used in this project for the shape detection part using built in functions from the OpenCV environment. Numpy arrays support n-dimensional array objects, linear algebra, Fourier transformations and much more [12].

2.4 Dynamic Linking

The programming language C++ allows a program to extend itself during run-time by using dynamic linking. Dynamic linking includes the name of the external libraries, but they are first linked into the executable during run-time when both the libraries and the executable file are placed into memory. Only a single copy of the shared library is kept in memory, and that reduces the size of the executable file significantly.

The biggest difference between static and dynamic linking is that if something changes in the code of the static linked library, the executable file has to be re-linked and recompiled all over again, otherwise the changes wont be applied. This is not necessary in dynamic linking. Individual external modules can be changed, recompiled and added during run-time of the executable file.

Dynamic linking is mentioned here because it plays a big role in the GeoMod program. Both the camera view and the generated model frequently mentioned in this thesis are linked in dynamically. This gives the generated model class a huge advantage of being able to change while the program is running.

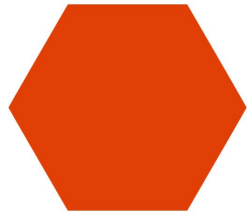
Chapter 3

Image Manipulation

3.1 Source Images used for testing

In this section the test images that are used to build this program are introduced. The goal from the beginning was to transform shapes from images and plot them inside the camera view of the GeoMod program. Simple shapes such as rectangles, hexagons and triangles were used to lay a groundwork (3.1).

(a) Simple rectangle used as test images



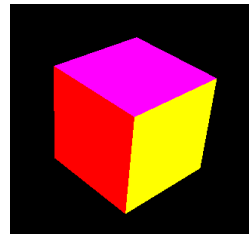
(b) Simple hexagon shape

Figure 3.1: Simple shapes that were used to lay some groundwork

After the groundwork was laid, shapes in three dimensions were built upon that foundation. Cubes were the central focus of this thesis. Images of cubes with a special layout were chosen such that three sides of the cube could be identified on the image. Figure 3.2 demonstrates two typical sample images that were used in the early stages of the thesis. As seen on those images, each side has a different colour, making it easier to identify the border between each rectangle plate.

Next cubes with one uniform colour, the sides differing only due to the effect of shadows, are introduced. The figures 3.3 show these more challenging images. These images were a lot harder to analyse and work with, and here the importance of colour manipulation was central.

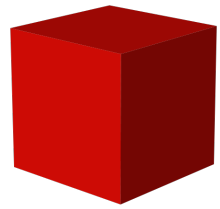
(a) Simple picture of a cube



(b) Simple picture of a cube with black background

Figure 3.2: These images were the basis for the construction of a 3D model

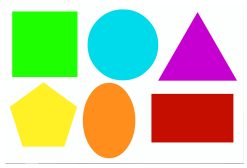
(a) Cube with black sides



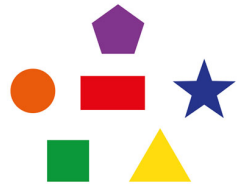
(b) Cube with red sides

Figure 3.3: More advanced images in need of colour manipulation

Supporting the recognition of multiple shapes in one single image has always been very central throughout this project. This made the work a lot more complicated. Here are two images used for testing multiple shapes in one image, *figure 3.4*.



(a) Simple shapes in one image



(b) More advanced shapes in one image

Figure 3.4: Images with multiple shapes

At the end more irregular shapes were tried to be modelled in the GeoMod environment such as a fish (*fig 3.5a*), and a heart (*fig 3.5b*). The results will be shown later on (*chapter* 7).

(a) Picture of a fish

(b) Picture of a heart

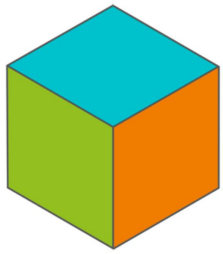
Figure 3.5: Images with multiple shapes

3.2 Image Manipulation

Before shapes can be detected on an image, it is advised to perform various manipulations on it first. This will make it a lot easier to detect shapes and objects. It will also save a lot of memory space since the number of possible pixels for calculations is greatly reduced after applying various filters.

3.2.1 Gaussian Blur

One of the filters applied to the image is called Gaussian Blur which is a very common and effective filter method. Gaussian Blur is often applied in professional photo editing software such as Photoshop. First a Gaussian distribution is created and the distribution is then used to create a convolution Matrix [50]. Here, the distance between each point will have an effect on the weight of the pixel. The further the distance, the less it weighs upon its neighbours. Later on in this paper, the Canny Edge Detection algorithm is applied to images, which also includes a form of Gaussian Blur. Here is a picture showcasing the Gaussian Blur effect on one of the test picture.



(a) Without Gaussian Blur

(b) With Gaussian Blur

Figure 3.6: Showing the effect of the Gaussian Blur

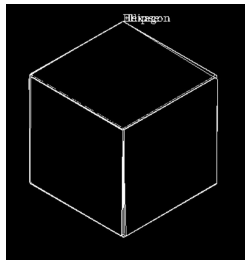
3.2.2 Bilateral Filter

Another filter that was applied was the Bilateral Filter, which reduces unwanted noise while keeping edges sharp. Like Gaussian Blur this filter is quite slow, but since run-time

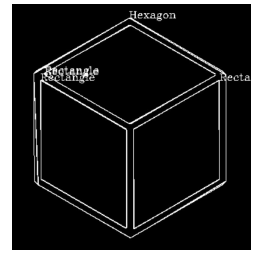
was not a really relevant factor in this thesis, both filters are applied to the image. The user can adjust the value of the sigma variable used in this function. The sigma variable determines how pixel influence each other over distance. The increase in the sliders value is proportional to the distance of pixel, a larger value means pixel over a greater distance influence each other.

3.2.3 Eroding and dilating

Eroding and dilating the image turned out to have a great effect on the image. By eroding the image, the borders of the shapes increase in size. This is due to the picture being in binary format and the brighter areas of the image getting thinner while border of the image is being pushed outward. Dilating does the opposite; it decreases the border size of the shapes in a picture. Here the user also can choose the iteration count of dilation and erosion. As default the image is eroded and dilated two times. This has proven to give a good initial result on all the test images. Shapes with many points benefit greatly from erosion and dilation to simplify their shape.



(a) Binary image with dilation of 2



(b) Binary image with erosion of 4

Figure 3.7: Effect of dilation and erosion

3.2.4 The Canny Edge Detection algorithm

The Canny Edge Detection algorithm is important for detecting shapes and their outline. This algorithm has multiple stages, the first being noise reduction in form of Gaussian Blur, which has been mentioned earlier.

After removing the noise, the edge gradient and direction for each pixel is found. This is done by filtering the image with a Sobel kernel, to get the derivative in the horizontal and the vertical direction. A gradient direction has to be perpendicular to the edges. A full scan of the image will then reveal all pixels that are not considered to be part of an edge. These pixels are then suppressed by setting the pixel value to zero.

For the last stage, the pictures threshold is determined for what values identify a pixel as an edge. A maximum and minimum value will determine whether a pixel is to be considered an edge or not. If it lies outside the threshold, but it is connected to a point that is known to be part of an edge, it will be considered to be part of an edge. Otherwise its value will be zero. Both the minimum and maximum of the threshold can be changed by the user. This is done by using the sliders created by this thesis' python code. Changing

the values of the maximum and minimum slider will have a drastic effect on the output of the image. By default, the minimum limit is set to 50 and the maximum to 100. This has proven to be a good scale for most images. The effect of the Canny Edge Algorithms is shown in the image below (Figure 3.8).

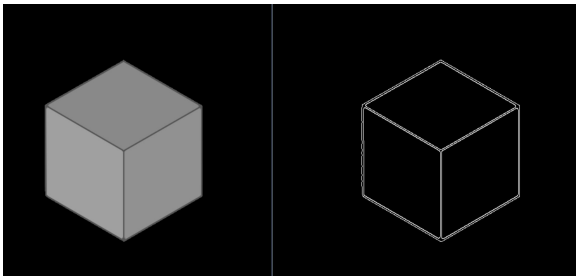


Figure 3.8: Picture showcasing the drastic effect the Canny Edge Detection Algorithm has on a picture. Left without-, right with Canny Edge Detection

3.3 Colour correction

Colours have a huge impact on how we humans can recognise and see objects and shapes in nature. Something with a bright colour will instantly stick out of its environment. That effect is widely seen in nature and applied by plants and animals alike. Humans have throughout history also learned how to use colours to manipulate their surroundings. But what meaning do colours have for a machine? Can it differentiate between a black and white image and one with colours? And will colours have an impact on how a machine can recognise objects and shapes in an image? This questions will be explored throughout this thesis.

Colours in an image are most commonly represented by the RGB colour space. The RGB colour space consists of all the colours which the triangle of the red, green and blue colours can produce. These colours are the primary colours of light. Mixing more colours together in the image will give a lighter image and in the end a white image.

Other format for representing colours in images are the HSL and HSV colour spaces. The HSL stands for Hue, Saturation and Lightness, while HSV stands for Hue, Saturation and Value. These colour spaces are just transformation of the RGB values, using simple equations to convert them into the colour spaces. So why do these colours spaces exist and why was the HSV colour space chosen for this project rather than the RGB colour space?

The answer is simple; the HSV and HSL colour spaces are much more user friendly than the RGB spectre. It makes a lot more sense for us humans to change a value in the saturation range and see the intensity of the colour change, instead of trying to figure out a lighter shade of orange by adding and subtracting values for red, green and blue.

By using the python program, the user can easily change all values the HSV colour space includes. Six sliders represent the minimum and maximum value of hue, saturation and value. By adjusting these sliders, the OpenCV shape detection implementation can

detect edges and shapes previously not possible. This is true especially for dark images with shadows, or colours that are very similar.

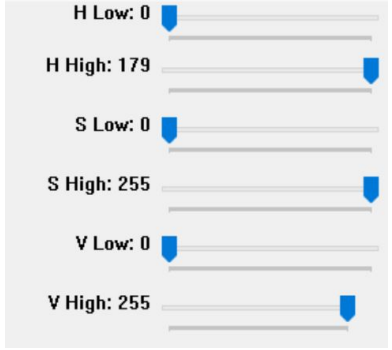
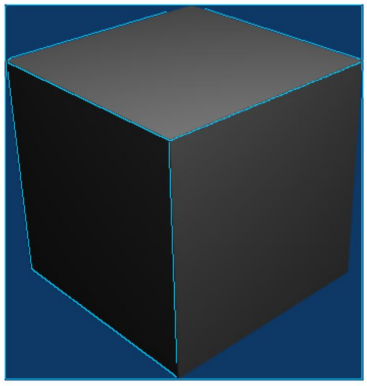


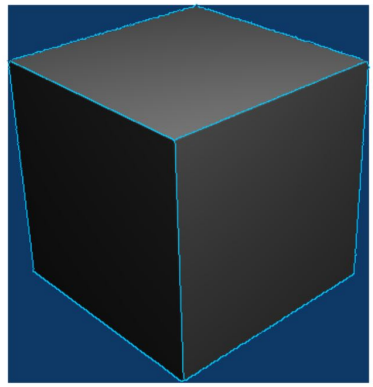
Figure 3.9: The control panel for the sliders for colour correction

The Graphical User Interface of OpenCV does not allow for a lot of customisation, so this panel of sliders (*Figure* 3.9) is the simplest version available.

For testing, an image of a dark cube was used. The sides of the cube are all variations of black and grey. When not using the colour correction option created for this thesis, only the hexagon outline of the cube can somewhat be determined. The background of the cube is recognised to be part of the cube shape as well. When using the sliders and adjusting the intensity of colours and their lightness, the program can find lines connected the sides of the cube and in the end a good approximation of a cube can be created.



(a) Standard shape detection without colour correction



(b) Shape detection with colour correction

Figure 3.10: With and without colour correction

A cyan coloured line highlights the edges found in the pictures shown in figure 3.10. A close look at figure 3.10a will show that the contour of the rectangle that makes up the

picture is recognised as a shape. A cyan outline is drawn around that rectangle. In figure 3.10b, where colour correction is applied, the background of the cube is not recognised to be part of the shape. The colour detection can also detect all lines correctly, such as the bottom right line and the right line which are not recognised in the picture on the left. The top right corner of the rectangle in the left picture is missing a few points to connect the lines as well. This is not the case in the picture where colour correction is applied.

Chapter 4

Useful functions for shape detection

4.1 Creating a threshold

This chapter gives a short introduction to functions used in the OpenCV environment. Most of these functions solve complex tasks on only a single line. The functions inputs and outputs are explained to give a short overview of what these functions accomplish. Before using these functions on an image, it is advised to apply the filters mentioned in chapter 3 for a better result.

4.1.1 The threshold function

To find a fixed-level threshold for each array element in the picture, something that needs to be done before applying a contour detecting algorithm, either a OpenCV function called *threshold()* or the Canny Edge Detection algorithm can be used. Both were considered and tested. Since the threshold function only considers the intensity for each pixel value, the Canny Edge Detection algorithm is better suited for more complex images. Although test images for this project were rather simple, it is good practice to lay a groundwork for more complex tasks in the future. The run-time for both methods does not differ much, especially not in this thesis' scope.

```
1 _, threshold = cv2.threshold(threshold_one, 240, 255, cv2.THRESH_BINARY) #creates the
      threshold of the image
```

Listing 4.1: Threshold function

The implementation of the threshold function is shown in listing 4.1. Although this function is not used, it still exists in the current code, but it is commented out. The functions first input is the image we want to process. This image needs to be a grayscale image and it will then be converted by the function into a bi-level binary picture. The objects of interest in the image are white while the background is black. The other inputs are the

values for the threshold. Here the values 240 to 255 are used for binary pictures. The last input is the style of threshold. There are different styles of threshold suited for different kind of situations. Worth mentioning is the adaptive threshold function that OpenCV also provides. This function is suited when the image has varying illumination. This function could be considered for future extensions.

4.1.2 The findContours function

Next up is the *findContours* function [13]. As the name suggests, it finds contours on an image, more precisely it finds white objects on a black background. It is important that the image is in binary format before it is fed as input to this function. The *findContour* function takes three arguments as input. The first being the source image, the second the mode for contour retrieval and the third is the contour approximation method. It also has three outputs; the first is a modified image, the second are the contours found in the image, and the third is the hierarchy of contours found in the image. The last two outputs were very central in finding shapes and their relation to each other. The functions modes and outputs are explained in more detail later in section 5.3.

```
1 _, contours, hierarchy = cv2.findContours(threshold, cv2.RETR_TREE, cv2.
    CHAIN_APPROX_SIMPLE) #finds the contours and the hierarchy of the contours based
    on the template
```

Listing 4.2: Find contour function

4.1.3 The approxPolyDP function

To approximate a polygonal curve OpenCV has a function called *approxPolyDP* which does most of the work [13].

```
1 approx = cv2.approxPolyDP(cnt, epsilon , True)
```

Listing 4.3: Approximation of a polygonal function

The first input for the *approxPolyDP* function is a single contour, a numpy array, of many that were found by using the *findContours* function. Here, by definition, a contour is a curve joining continuous points having around the same colour or intensity. From here on out the word *"contour"* and the word *"model"* will be used interchangeable.

```
1 epsilon = (cv2.getTrackbarPos('epsilon','sliders')/10000)*cv2.arcLength(cnt, True) #
    estimates an epsilon to approximate a polygon
```

Listing 4.4: The epsilon parameter

The next parameter is for specifying the maximum distance between the original curve and the curves approximation. This parameter is often called Epsilon. The epsilon parameter gets its value by multiplying a constant with the arc length of the current contour. The value of Epsilon usually ranges between 1-5% of the original contour perimeter, which was found by using the *arcLength* function on the current contour. Upon running the program, the user can adjust the value to a range between 0,002 and 0,03. This range was found to give the most differences in results. Too low of a limit would on occasion exceed the

programs memory. Below are images showcasing the difference that the value of epsilon makes. The original image is figure 3.4a. while the modified image is shown in figures 4.1 and 4.2. Usually a lower value for epsilon will increase the amount of points, making it useful for complex, unusual structures. For simple shapes, a high epsilon is better suited.

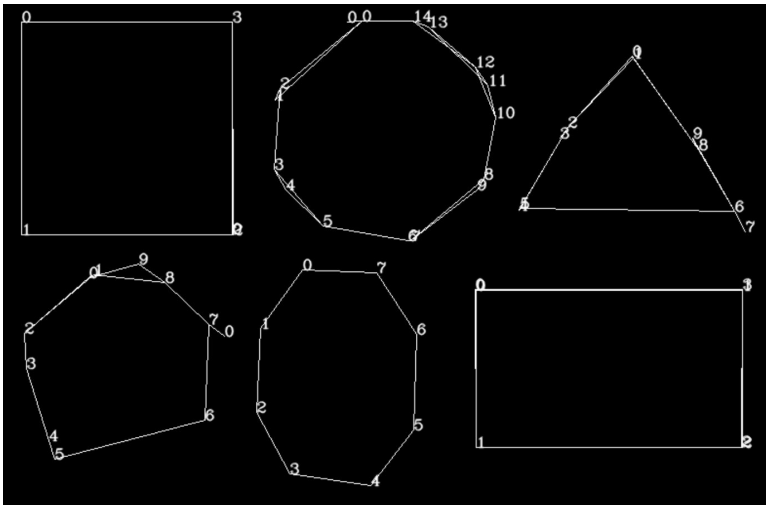


Figure 4.1: Binary image with a low epsilon value

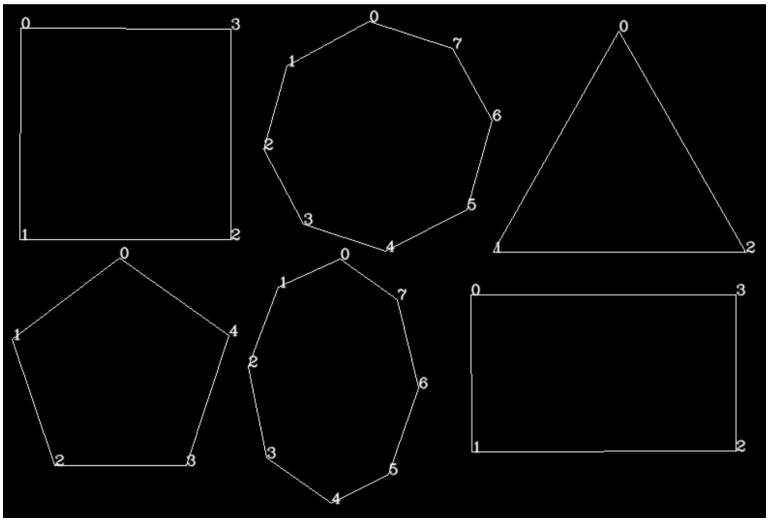


Figure 4.2: Binary image with a high epsilon value

The last variable in the approxPolyDP function is a Boolean. The Boolean specifies that if the approximated curve is closed, the first and last vertices are connected with each other.

4.1.4 The drawContours function

To draw the contours that were found, OpenCV provides a built-in function. It is called *drawContours*, and it can be used to draw a shape if the shapes boundary points are provided [13]. Its first input is the source image. Next is the contour in the form a list, and the third argument is the index of the contour. The last inputs are the colour and thickness of the contour drawn.

```
1 cv2.drawContours(threshold, [approx], 0, (255), 1)
```

Listing 4.5: The drawContour function

With the help of OpenCV, getting the initial coordinates for edges in a shape is relatively easy. The built-in functions are very useful and this is not by mere coincidence. Using Open CV is therefore a very efficient way of doing shape detection.

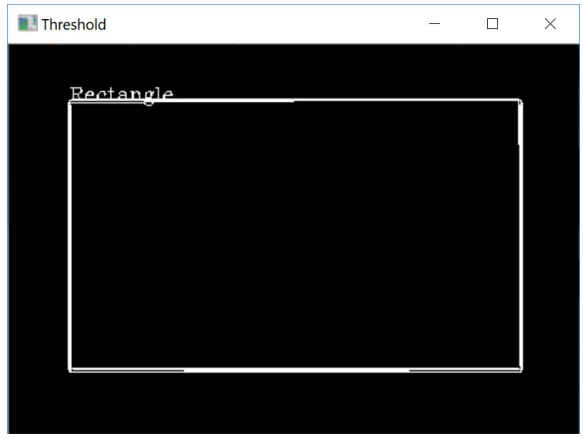


Figure 4.3: The window displaying the threshold image where the contours have been drawn by the drawContours function. This figures source is a simple rectangle, as shown in figure 3.1a

Chapter 5

Detection of shapes

5.1 Communication between the two programs

This project already had a complicated start, since it was impossible to get OpenCV to work with Qt. In theory it should work, and there are many tutorials showcasing how to install them along side each other. But in the end, two separate programs had to be used. Jupyter Notebook with OpenCV and Qt with the GeoMod program.

To be able to let both programs communicate with each other, a write and read from file system was created for both programs. The OpenCV side would detect shapes, manipulate and modify the shapes coordinates and write them to file. The GeoMod side would read the data from file and create models in its environment. This was a tedious way to connect the programs, and in retrospect, more time should have been spent to somehow get OpenCV to work with the Qt application. But after already losing 3 weeks to that fruitless endeavour, a simple read and write from text files solution was implemented.

The read and write functionality is explained in more detail later on in section 5.6. The communication system was not made extravagant on purpose, since future work should consist of porting the python and OpenCV functionality over the Qt side. More on that on the future work section 8.3.

5.2 Implementation of shape detection

Like all other projects that venture out into something untested, starting small and easy is important to not get lost. Shape detection was therefore first applied on simple objects and shapes. Easy to detect rectangles, triangles and hexagons were part of the first iteration of test images.

The functions used to detect outlying shapes were mentioned in chapter 4. Using these functions, it was a rather straight forward task to get the first outline of a shape.

After applying the Canny algorithm, the code started detecting duplicates for every single outline. By summing the value of the outlines coordinates and comparing it to the

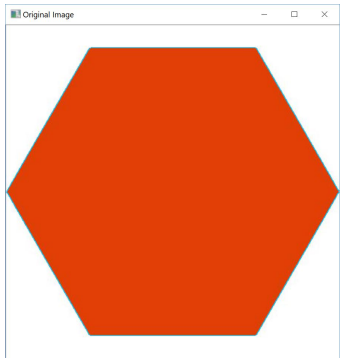
previous shapes summation we could determine if the current shape is a duplicate of the previous one. A threshold of two times the length of the approximation was used to filter out the duplicates. This turned out be a good estimate for almost every model, so a slide for manual user input was not implemented.

```
1 approx = cv2.approxPolyDP(cnt, epsilon , True)
2 #to check of almost duplicates exists with just a variation of a few pixel, we check
      if the sum is almost the same as the next object (possible duplicate)
3 if (not(abs(np.sum(approx.ravel()) - np.sum(lastApprox)) < 2*len(approx))):
4     if (cv2.contourArea(cnt) > (cv2.getTrackbarPos('Area Of Objects','sliders'))): #
      only shows the shapes with an area bigger than what the user chooses on the
      slider
5 #
```

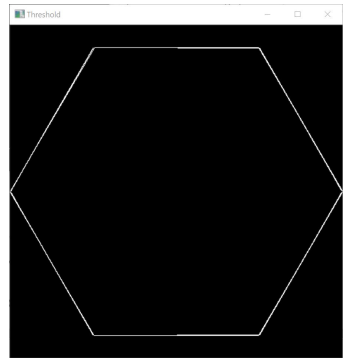
Listing 5.1: Main code for detection of shapes

The code snippet is taken from the part of the code that finds the contours. It can be found inside the loop that edits each contour. Line 3 filters out the duplicates by summing and and taking the absolute value of the current and last approximation of a polygon curve. Using the methods described in the chapter 4 an outer contour has now been found.

At this point an outer shape like in figure 5.1b could be found from a simple test picture.



(a) Original picture of a hexagon shape with a cyan outline highlighting the shape found



(b) The contour detected of that hexagon picture

Figure 5.1: Detection of a simple shape

After successfully finding a outer shape, it was time to write the shapes coordinates to a file, so it could be read it the GeoMod program. Here the keyword *"end"* was used to determine the end of a model to write to file. Before each *"end"* keyword, the x and y coordinates were written to file.

To give a small preview of how the GeoMod program could understand the Python part of the projects, a small part of the read from file method is shown here. This code is from the very early stages of reading from file, and it only considered the outer shape to be read. It is from the GeoMod/Qt environment and it is written in C++. The code was later modified to support a wider form of models. The decision to include it here is because it is a very easy to understand function and hopefully it gives a clear picture of how the reading from file part works.

```
1   while (!in.atEnd()) {
2       QString temp = in.readLine();
3       if (47 < temp.at(0).unicode() && temp.at(0).unicode() < 58) {
4           list.append(temp.toDouble());
5            i++;
6       } else {
7            if (list.isEmpty()) {
8                std::cout  << "No coordinates were read from file , the list is empty
    " << std::endl;
9            }
10           else {
11               list = resizeList(list, 10);
12               if (list.length() == 6) {
13                  qDebug() << "A Triangle was read from file";
14                  model = new generateHexagon(list, "Triangle");
15               } else if (list.length() == 8) {
16                   qDebug() << "A Rectangle was read from file";
17                   model = new generateHexagon(list, "Rectangle");
18               } else if (list.length() == 10) {
19                   qDebug() << "A Pentagon was read from file";
20                   model = new generateHexagon(list, "Pentagon");
21               } else if (list.length() == 12) {
22                   qDebug() << "A Hexagon was read from file";
23                   model = new generateHexagon(list, "Hexagon");
24               } else if (list.length() < 30 && list.length() > 12 ){
25                   qDebug() << "An Ellipse was read from file";
26                   model = new generateHexagon(list, "Ellipse");
27               } else {
28                   qDebug() << "No supported object was found";
29               }
30           }
31           i = 0;
32           list.clear();
33
34      }
35
36  }
```

Listing 5.2: Old version of a read from file method

Line 2 loads each line of the text file into a *QString*. A *QString* is very similar to a normal C++ string, but when used in the Qt Environment, it has some benefits, such as more build-in operations on strings.

After the current line in the text file was read into a *QString* the ASCII value of the first character of the *QString* is compared in an if-sentence. It checks if the ASCII value is between 47 and 58, which means that the the lines first character that was read is a number. If it is a number, the whole number is added to a list. This whole number could consist of multiple characters. Numbers are added into the list until something else than a number is being read from file, like the key word ”end”. If the list is not empty, the list would be send to the class called *genereateHexagon*. This was in the early stages of the projects, so only support for creation of a hexagon was sought initially. This was changed in the later part of the code, which can be seen in the section 6.1 about reading from file. Line 11 performs a re-size of the numbers read from file, making them smaller, so they could more easily be viewed in the GeoMods camera view. This function is also explained in more detail later. The *qDebug()* lines in the code above are simply to let the user know which shapes have been read from file.

5.3 The inner contents of a shape

Now that the outer edge had proven to be detected correctly, it was time to focus inwards. What lies inside a contour? The next step in creating a three dimensional cube from a picture is to take a peek inside the outline of the shape.

From the test images of a cube, it could be assumed that an image of such a cube would have a hexagon outline and that three rectangles make out its interior, like figure 5.4. The algorithm used to find the outermost shape could not account for any holes or shapes inside of the outer shape. The first attempt was to cut out a mask based on the coordinates of the outer most shape. On this mask, usually hexagon shaped, another algorithm for internal search of nodes would be performed. After many tries, this turned out to be an overly complicated endeavour and it was soon abandoned. Mostly because of the difficulty of linking the existing shape, its cut-out and its interior together afterwards.

The search continued for something that could connect the parent shape to its children. Here a function that was introduced earlier comes into play. The *findContours* function takes a mode as its second last argument as input. Four modes are available for use; *"CV_RETR_EXTERNAL"*, *"CV_RETR_LIST"*, *"CV_RETR_CCOMP"* and *"CV_RETR_TREE"*. The first is used for finding extreme outer contours, such as finding the triangle, rectangle and hexagon shapes in the beginning. The *"CV_RETR_LIST"* was the one applied to the mask, it searches for all the inner contours, without establishing a hierarchy of some sort. The last of the modes is the most interesting one, and the one that was ended up being used. It retrieves all the contours in the image and creates a complete hierarchy of the contours that are nested. Each contour found gets its own array of information about its hierarchy. The array can look like this:

```
[ 4 -1  3  1]
```

Figure 5.2: Hierarchy array example

The first number stands for the next contour in the same hierarchy as the current contour [1]. A -1 here represents that no such contour exists. The next number says if there are any previous contours under the same parenthood as the current contour, in other words a sibling. If this number is a -1, no such contour exists. The third number displays the next child of the current contour. Again, a -1 shows that the current contour does not have a child. The last digit stands for the current contours parent. The last two digits are quite important to create a parent and child relationship.

Implementing this algorithm together with a way to keep hold of all the hierarchy turned out to be very complicated. After a few unsuccessful tries, something had to be wrong. It turned out that the duplicates created by the Canny Algorithm, which were filtered out later, messed up the whole hierarchy. Which meant that the whole hierarchical structure was basically useless.

A solution was to convert the hierarchy list into something usable. There were two lists that needed to be kept track off. One containing the coordinates for the contours and one containing the hierarchical data for each contour. This means that both lists should be of the same length. By previously deleting the duplicates, the lists would no longer be of

the same length. Instead of deleting that array of coordinates, an empty array was added. This way an empty array would symbolise a duplicates appearance at that index.

Now a list of contours could look like this:

```
List of Hierachy: [[-1, -1, 0, -1], [], [4, -1, 2, 0], [], [6, 2, 4, 0], [], [-1, 4, 6, 0], []]
```

Figure 5.3: List Of hierarchy with duplicates as empty lists

Next thing was to convert the hierarchy list. This was done by checking if the length of the current contour is zero. Length of zero means that there would have been a duplicate at that index. This means that the index of an occurrence of a duplicate in the hierarchical array of the other contours would have to be replaced by the current contours index, which was the other one of the two duplicates. At the end the pieces were pasted together again, and a correct hierarchical list which can relate correctly to the contours detected in the picture was created.

```
1  def convertListOfEdges(listOfModels, listOfHierarchyOfModels):
2  modelsWithParentAndChildRel = []
3  hierarchyWithoutEmptyLists = []
4  for i in range(len(listOfHierarchyOfModels)-1):
5      #checks if next list is duplicate of current list, removes duplicates and empty
       lists, converts the models number incase the duplicate is deleted
6      if (len(listOfHierarchyOfModels[i]) != 0 and len(listOfHierarchyOfModels[i+1]) ==
       0):
7          for j in range(len(listOfHierarchyOfModels)):
8              if (len(listOfHierarchyOfModels) != 0):
9                  for k in range(len(listOfHierarchyOfModels[j])):
10                     if (listOfHierarchyOfModels[j][k] == (i+1)):
11                         listOfHierarchyOfModels[j][k] = i
12             if (listOfHierarchyOfModels[j] != [] and listOfHierarchyOfModels[j] not in
       hierarchyWithoutEmptyLists ):
13                 hierarchyWithoutEmptyLists.append(listOfHierarchyOfModels[j])
14 if (hierarchyWithoutEmptyLists == []):
15     for i in range(len(listOfHierarchyOfModels)):
16         hierarchyWithoutEmptyLists.append(listOfHierarchyOfModels[i])
```

Listing 5.3: Code to fix index in hierarchical list

Here line 6 is important. It checks if the current contour i has a duplicate $i+1$. If this is the case, all occurrences of the index $i+1$ are replaced by the index i. This replacement happens in line 11. Since a hierarchy exists for every single external contour there is quite a nested hierarchical list to work with.

Since the hierarchical list is now correct, it can be used to construct parent and child relationships in the form of lists. This code showcases how that was done.

```
1  tempList = []
2  modelsWithParentAndChildRel = []
3  k=0
4  #create a list of parents and their children
5  listsAlreadyAdded = []
6  if (len(hierarchyWithoutEmptyLists) != 0):
7      for i in range(len(listOfModels)):
8          currentModelNumber = i
9          currentModel = listOfModels[i] #we check if this model has any children
10         listOfCompleteFamily = [] #potential children will be added to this list
11         for j in range(len(listOfModels)):
12             possibleChildModel = listOfModels[j] #possible Child of current model
13             childsParent = hierarchyWithoutEmptyLists[j][3] #number of the current
       model
```

```
14              if (childsParent == currentModelNumber): #if childs parent exist and is
    the same as the current model, this will be invoked
15                  if (currentModel not in listOfCompleteFamily): #add the parent to the
    list of the whole family
16                      listOfCompleteFamily.append(currentModel)
17                      listsAlreadyAdded.append(currentModel) #add parent to lists that
    have been checked
18                  listOfCompleteFamily.append(possibleChildModel)
19                  listsAlreadyAdded.append(possibleChildModel)
20          if (len(listOfCompleteFamily) == 0): #if the model doesnt have any children
21              if (currentModel not in listsAlreadyAdded): #and if the currentModel has
    not been added
22                  listOfCompleteFamily.append(currentModel) #current model without
    children gets added to its own list
23          if (len(listOfCompleteFamily) != 0): #making sure we dont add an empty list
24              modelsWithParentAndChildRel.append(listOfCompleteFamily) #we finally add
    the list of the family to the final list
```

Listing 5.4: Code to fix index in hierarchical list

The logic behind this code is simply selecting a model and checking if this model has any children by looping through the rest of the current models hierarchy. This is done for each list of the hierarchy of every model. The tricky part is keeping a memory of models already added as children or checked to be a parent. If a model does not have any contour detected as its children, and it is not a child itself, then the model is added directly to the final list, which here is called *modelsWithParentAndChildRel*. This is checked on lines 20 and 21.

Handling more than one model, with or without children made the code rather complicated. One extra depth could have been avoided if multiple models in one image would not have to be taken into account.

After successfully establishing a hierarchy and parent–child relations, the next step was to determine which points lie close together and combining them into a single point.

5.4 Combining points

The goal of this thesis was to create a three-dimensional model of a cube. Combining two nodes that are close to each other is a vital part of that process. The theory behind it is as follows:

After a hexagon for the external contour was found, the search continued inside its contour. Here hopefully three rectangles will be found. The image 5.4 shows the ideal situation.

Looking closely at the image, multiple nodes are close to other nodes. The top right of the image, where it says rectangle, has a corner where three nodes meet. One node is from the outer contour, two from the rectangles inside. Another point like that is in the middle of the total contour. Here three rectangles meet, and three nodes are close to each other. The goal is to combine those three nodes into a single node, representing all three with a single node. A short summary of the thought behind a function that completes this goal follows below.

At this state of the code we have an array of nodes for each model in the image. Each node has two values determining its position. For convenience these two values are called the x and y values, like they would if they were on a two-dimensional x-y plane. What is

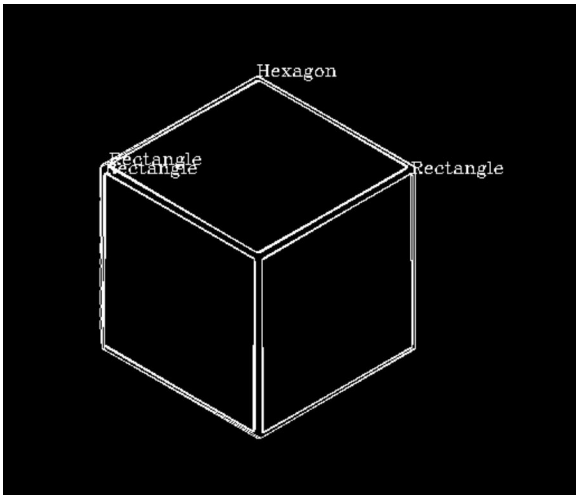


Figure 5.4: Binary picture of a cube where a hexagon as external structure was detected and three rectangles on the inside were detected

needed now is a function that takes a node and compares it to every single other node. In programming terms this means a double loop. One loop to select a node and the second one to go through the whole list of nodes and see if it is close to the node selected. Since multiple models can be detected in an image, every single model needs to go through that process. This means there are suddenly three loops to consider. Three for-loops are not optimal for models that could contain many points. Although run-time was not of great importance for this project, a great deal of time was spent finding the most efficient way to solve tasks. Of course, sometimes only so much optimisation could be done to solve a task.

To check if two points are close together, the x-value of the current node that is selected is subtracted by the x-value of the next node found. Taking the absolute of the value will give a positive value. The same is done for the y-values of the two nodes. If both absolute values are lower than a limiting value chosen by the user, the two nodes will be merged together. The new value will have the average of the x and y value between the two nodes. If the absolute values are not below the limiting value, the next node in the same model is chosen for a comparison. This is then done until the end of the array of nodes has been reached. A new node from the same model is selected and we repeat the whole process. If a model has n number of nodes, $n*(n-1)$ comparisons are done. In the end if nodes were merged into a single node, a new list of fewer nodes that represent a model is available. Here is where the most important part of the code comes into play; the recursive part. A small part about recursive functions was written before in section 2.2.5.

The list of the new nodes after the three loops is compared to how the list of the nodes was before the loops. Here is an example of how this looks like:

Here each number represents the number of nodes in a model. Each number is a model, so that in total there are 7 models in this list. In the beginning, before pasting points together, the first model has 12 nodes, the second model has 6 nodes and so on. After the

```
Length Before [12, 6, 14, 51, 18, 6, 9]
Length After [8, 3, 8, 28, 10, 5, 5]
Length Before [8, 3, 8, 28, 10, 5, 5]
Length After [4, 3, 4, 17, 9, 5, 5]
Length Before [4, 3, 4, 17, 9, 5, 5]
Length After [4, 3, 4, 15, 9, 5, 5]
Length Before [4, 3, 4, 15, 9, 5, 5]
Length After [4, 3, 4, 15, 9, 5, 5]
DONE
```

Figure 5.5: This image shows how nodes are added together. Each index in this array is a contour with x amount of edges

first recursive iteration, after running the three for-loops for the first time, the first model now has 8 nodes and the second model has 3 nodes. This means a few nodes have been pasted together to form a single node. This function is continuously run recursively until the list of length before and the list of length after each recursive call is the same. If they are the same length before and after an iteration that means that no more nodes can be found that are close together. If this happens the function ends and the list of the final nodes for each model is exported to the next function.

Here is the whole code for the recursive function, a more detailed explanation of each line follows below the function.

```
1  def recursiveCheckForPointsCloseTogether(listOfNewEdges, listOfRelations):
2      #recursively finds nodes which are close to eachother and exchanges them by the
        average distance, used for multiple shapes in a picture
3      lengthBefore = []
4      finalEdges = []
5      for k in range(len(listOfNewEdges)):
6          subList = []
7          alreadyCheckedEdges = []
8          updatedlistOfRelations = []
9          lengthBefore.append(len(listOfNewEdges[k]))
10         for i in range(len(listOfNewEdges[k])):
11             count = 0
12             for j in range(len(listOfNewEdges[k])):
13                 if (i != j):
14                     if (i not in alreadyCheckedEdges and j not in alreadyCheckedEdges)
       :
15                         currentX = listOfNewEdges[k][i][0]
16                         currentY = listOfNewEdges[k][i][1]
17                         nextX = listOfNewEdges[k][j][0]
18                         nextY = listOfNewEdges[k][j][1]
19                         pixelTolerance = cv2.getTrackbarPos('sizeTolerance','sliders')
20                         if (abs(currentX - nextX) < pixelTolerance and abs(currentY -
       nextY) < pixelTolerance):
21                             deltaX = int(np.ceil((currentX + nextX)/2))
22                             deltaY = int(np.ceil((currentY + nextY)/2))
23                             subList.append([deltaX, deltaY])
24                             listOfRelations = updateRelationBetweenNodes(
       listOfRelations,[currentX,currentY],[nextX,nextY],[deltaX,deltaY])
25                             alreadyCheckedEdges.append(i)
26                             alreadyCheckedEdges.append(j)
27                             count+=1
28             if (count == 0 and i not in alreadyCheckedEdges):
29                 subList.append([listOfNewEdges[k][i][0],listOfNewEdges[k][i][1]])
30         finalEdges.append(subList)
```

```
31
32      lengthAfter = []
33      for i in range(len(finalEdges)):
34          lengthAfter.append(len(finalEdges[i]))
35      if (lengthAfter != lengthBefore):
36           recursiveCheckForPointsCloseTogether(finalEdges, listOfRelations)
37      else:
38          main(finalEdges, listOfRelations)
39          print ("DONE")
40
41
42      return finalEdges
```

Listing 5.5: Code to add nodes together

This function has three for-loops, one at line 5, 10 and 12. The first for-loop selects a single model in a list of models. Once inside the selected model, a node is selected. This node is then compared to the next node and all the other nodes in the list. The if-sentence at line 13 makes sure that nodes with the same index, the same node, are not compared to each other. The next if-sentence at line 14 filters out the nodes that have been compared with each other before. Inside this if sentence, a value for the pixel tolerance is specified by the user through a slider. Then the nodes are checked if they are close to each other. If both x- and y-values are closer than the pixel tolerance, a new point is created by taking the average of both points. This is done on line 21 and 22. On line 24 a subroutine is called. This subroutine is called *updateRelationBetweenNode* and takes the relation list of nodes, both the two old nodes and the new node which was calculated (*[deltaX, deltaY]*) as input. This subroutine then replaces all the occurrences of the two old nodes in the relation list of nodes with the new node.

At line 35 the check for the need of recursion is done. An if sentence checks if each model in the list that was taken as input, *listOfNewEdges*, has the same length as each of the models in the list after the for-loops, *finalEdges*. Both of the lists are the visualized in figure 5.5. If both lists are not identical, the function is called again recursively on line 36. Its input are the newly calculated list of edges called *finalEdges* and the updated version of the list of relations between nodes called *listOfRelations*. If both lists of length are identical, the else case will be provoked and the main function on line 38 will be called. In the end, the print command printing the word "*Done*" will let the user know that the function has done its calculation.

5.5 Drawing the generated model

After applying the recursive algorithm some lines in the relation list for the nodes may be represented twice. These are deleted by a function with a simple double for-loop check. The code is as follows:

```
1 def deleteDuplicates(inputList):
2     #deletes lines which are represented twice
3     outputList = []
4     for k in range(len(inputList)):
5         subList = []
6         for i in range(len(inputList[k])):
7             if ([inputList[k][i][1],inputList[k][i][0]] not in subList and inputList[k
      ][i] not in subList):
8                 subList.append(inputList[k][i])
```

```
9           outputList.append(subList)
10      return outputList
```

Listing 5.6: Deleting duplicated lines

The next step is to draw up the lines that were found. The list of relations remembers which nodes are connected to which. This means that an edge can be drawn from one point to the next. The list of relations is of the form such as figure 5.6. A simple picture of a rectangle, the figure 3.1a is the source for this list of relation.

```
List of relation: [[[[48, 44], [48, 261]], [[48, 261], [409, 261]], [[409, 261], [409, 44]], [[409, 44], [48, 44]]]]
```

Figure 5.6: A list which keeps hold of which nodes connect to each other

```
1  def drawLinesOnImage(listOfLines, nameOfImage):
2      #draws the lines calculated on a blank, black image
3      lineThickness = 1
4      #calculates the size of the black image
5      height, width = getImageHeightAndWidth()
6      blackImg = np.zeros((height,width,3), np.uint8) #creates a new blank, black image
7      hasText = []
8      for k in range(len(listOfLines)):
9          j = 0
10         for i in range(len(listOfLines[k])):
11             x1 = listOfLines[k][i][0][0]
12             y1 = listOfLines[k][i][0][1]
13             x2 = listOfLines[k][i][1][0]
14             y2 = listOfLines[k][i][1][1]
15             cv2.line(blackImg, (x1, y1), (x2, y2), (255,255,255), lineThickness) #
       creates white lines on a black background
16             if ([x1,y1] not in hasText):#adds the node number to the edges of the
       final picture
17                 cv2.putText(blackImg, str(j),(x1,y1), font, 0.5,(255,255,255),1, cv2.
       LINE_AA)
18                 j+=1
19                 hasText.append([x1,y1])
20     cv2.imshow(nameOfImage, blackImg)
21     return blackImg
```

Listing 5.7: Draw lines onto blank, black image

Each innermost array contains two nodes, each node with a single x- and y-coordinate. First, on line 15, a black image is created of the same size as the original image. This is done by calling the *getImageHeightAndWidth()* function created early on in the code which gives the images height and width. In a for loop a new line is created from one node to the next. After the line is created, a number identification for the origin node is created on the image. This will give an image that looks like figure 5.7.

Before writing the final list of relations to file, another image shape detection is performed. This function is simply called *identifyFinalModel()*. This time a simple shape detection is performed on the image created by the individual lines between the nodes, an image such as figure 5.7. As seen on that figure, it is a simple black and white image. The *findContour()* algorithm is used again, but with the mode "RETR_EXTERNAL" as seen in line 6 for listing 5.9 . As mentioned before, this mode only finds the extreme outer contour of the image. This means, in the image above of the drawn cube, a hexagon shape should be identified. If multiple models are found, each identified model will be written to an array of strings. If a model with an unknown amount of edges is found, this could for

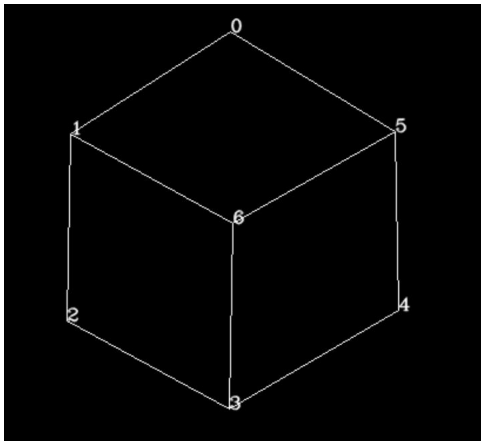


Figure 5.7: This is how the blank black image looks like after lines and the node number have been drawn

example be a star with 15 nodes, the model gets the name *"unknown"*. This can be seen in line 25. This information will be sent back to the main function and will in the end be written to file. Later in the GeoMod part this will be used to filter out which function to use to construct a model.

```
1  def identifyFinalModel(imageOfCalculatedModel):
2      nameOfModelIdentified = []
3      imageOfCalculatedModel = cv2.cvtColor(imageOfCalculatedModel, cv2.COLOR_BGR2GRAY)
4      imageOfCalculatedModel = cv2.convertScaleAbs(imageOfCalculatedModel)
5      _, threshold = cv2.threshold(imageOfCalculatedModel, 240, 255, cv2.THRESH_BINARY)
       #creates the threshold of the image
6      _, contours, hierarchy = cv2.findContours(threshold, cv2.RETR_EXTERNAL, cv2.
       CHAIN_APPROX_SIMPLE) #finds the contours and the hierachy of the contours based
       on the template
7      for cnt in contours:
8          epsilon = (cv2.getTrackbarPos('epsilon','sliders')/10000)*cv2.arcLength(cnt,
       True) #estimates an epsilon to approximate a polygon
9          approx = cv2.approxPolyDP(cnt, epsilon , True)
10         cv2.drawContours(threshold, [approx], 0, (255), 1)
11         x = approx.ravel()[0] #the current x value of the first part of the contour
12         y = approx.ravel()[1] #the current y value of the first part of the contour
13         if (cv2.contourArea(cnt) > (cv2.getTrackbarPos('Area Of Objects','sliders'))):
       #only shows the shapes with an area bigger than what the user chooses on the
       slider
14             if len(approx) == 3:
15                 nameOfModelIdentified.append("triangle")
16             elif len(approx) == 4:
17                 nameOfModelIdentified.append("rectangle")
18             elif len(approx) == 5:
19                 nameOfModelIdentified.append("pentagon")
20             elif len(approx) == 6:
21                 nameOfModelIdentified.append("hexagon")
22             elif 7 < len(approx) < 15:
23                 nameOfModelIdentified.append("ellipse")
24             else:
25                 nameOfModelIdentified.append("unknown")
26     return nameOfModelIdentified
```

Listing 5.8: Draw lines onto blank, black image

5.6 Writing to file

A vital part in this thesis was how the Python side could communicate with the C++/GeoMod side. This has already been mentioned briefly in section 5.1. The function demonstrated in that section is an outdated version. As per the projects end, there are three main write to file functions and one read from file function.

The first is simply called *writeToFile*. It takes in two arguments, the first being the list of relations between the nodes in each model, the second one an array which contains strings represented which model was detected by the second shape detection functions called *identifyFinalModel()*. Since the OpenCV GUI does not support simple file selection explorers, the destination file is always chosen manually by pasting in a string of its location into the *file.open* function. Here the *"a"* mode is chosen, which is the appending mode. It is used to add new data to the end of the file. After the file is open, the *f.truncate()* function is used to delete all of its content. Three for-loops go to the inner most level of the list of relations array. An if-else clause checks if the array on the inner most level either has a length of two or three. If it has a length of three, it means it is in three dimensions. A length of two requires the addition of an extra zero to be added.

Models are created and displayed in the y-z-plane in GeoMod, which means that the x-coordinate in the python codes corresponds to the y-coordinate in the GeoMod program and the y-coordinate corresponds to the z-coordinate in the GeoMod program. If the length is just two, a zero is added for the x-coordinate. After each model, the simple word "end" is written to file. This lets the GeoMod program know that this is the end of a model and it can start constructing this model. When writing the y-coordinate of the python part to file, the images height is subtracted by the nodes y-value. This is due to x-y coordinate system of the image in the Python part having its origin (0,0) in the top left edge of the image. The GeoMod program has its origin in the bottom left like a normal coordinate system. If the y-coordinate would not have been modified, the model would have been upside down in the GeoMod program.

```
1  def writeToFile(listOfLines, nameOfModelIdentified):
2      #writes all lines into the file, ot will then create only one model in GeoMod
3      height, width = getImageHeightAndWidth()
4      f = open('C:/z7mB_GeoMod/Models/coordinatesOfModels.txt', "a")
5      f.truncate(0)
6      for k in range(len(listOfLines)):
7          if (len(listOfLines) == len(nameOfModelIdentified)):
8              f.write(nameOfModelIdentified[k]+"\n")
9          for i in range(len(listOfLines[k])):
10             for j in range(len(listOfLines[k][i])):
11                 if (len(listOfLines[k][i][j]) == 3):
12                     #z-coordinate
13                     f.write(str(listOfLines[k][i][j][2]) + "\n")
14                     #x-coordinate
15                     f.write(str(listOfLines[k][i][j][0]) + "\n")
16                     #flip the y axis
17                     f.write(str(height - listOfLines[k][i][j][1]) + "\n")
18
19                 else:
20                     #z-coordinate
21                     f.write("0" + "\n")
22                     #x-coordinate
23                     f.write(str(listOfLines[k][i][j][0]) + "\n")
24                     #flip the y axis
25                     f.write(str(height -listOfLines[k][i][j][1]) + "\n")
```

```
26
27          f.write("end\n")
28      f.close()
```

Listing 5.9: First write to file function

The next write method is used to write the four points to file which were found by the four-points method. This is a simple modification of the write to file method above. The only real difference is that the final list written to file has only one dimension.

The last write to file method is called *writeSingleModelToFile*, and as the name suggests, it writes a single model to file. If the user has determined manually that only one model exists in the image, a decision overwritten by the user, this function will be used to write that single model to file. The reason to have this standalone write function is to absolutely force only the writing of one model to file. To make sure that what is written to file is only one model, we use the *flatten* function, and the *flattenOnce* function. The *flatten* function is shown below, it simply removes the depth of its source list. Here a three-dimensional list will be converted to a one-dimensional array.

```
1  def flatten(lis):
2      #Used to flatten an array into a one dimensional array
3      flattenedList = []
4      for i in range(len(lis)):
5          for j in range(len(lis[i])):
6              for k in range(len(lis[i][j])):
7                  flattenedList.append(lis[i][j][k])
8      return flattenedList
9
10 def flattenOnce(lis):
11     #Used to flatten an array once
12     flattenedList = []
13     for i in range(len(lis)):
14         for j in range(len(lis[i])):
15             flattenedList.append(lis[i][j])
16     return flattenedList
```

Listing 5.10: The flattening functions

The *flattenOnce* function is used to make sure that the list written to file does not have any sort of depth higher than one. It is a simple modification of the *flatten* function. The one-dimensional list can now be written to file.

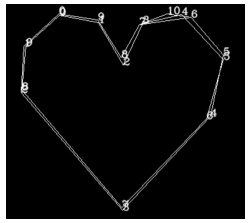
```
1  def writeSingleModelToFile(listOfLines, nameOfModelIdentified):
2      listOfLines_flattened = flatten(listOfLines)
3      listOfLines_onedimensional = flattenOnce(listOfLines_flattened)
4      #writes all lines into the file, it will then create only one model in GeoMod
5      height, width = getImageHeightAndWidth()
6      f = open('C:/z7mB_GeoMod/Models/coordinatesOfModels.txt', "a")
7      f.truncate(0)
8      f.write(nameOfModelIdentified[0] +"\n")
9      for k in range(0,len(listOfLines_onedimensional),2):
10         #z-coordinate
11         f.write("0" + "\n")
12         #x-coordinate
13         f.write(str(listOfLines_onedimensional[k]) + "\n")
14         #flip the y axis
15         f.write(str(height -listOfLines_onedimensional[k+1]) + "\n")
16     f.write("end\n")
17     f.close()
```

Listing 5.11: Writing a single model to file

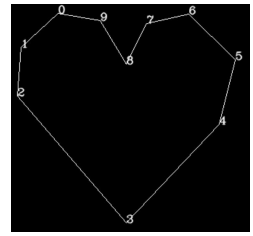
Another benefit of forcing the writing of a single model, is that small contours that have been created by an error in the contour algorithm are becoming a part of the whole, single model. These small contours are usually on the edge of the single models contours, where the probability for error is the highest.

5.7 Difference between a single model and multiple models in an image

Being able to support the detection and processing of multiple models was deemed to be a vital part of this project. But towards the end of the project, it became clear that to assume that only one model is present in the picture could also have its benefits. This is especially true for the run-time and the accuracy of the result. A slider in the control panel lets the user select if only one model is present in the whole picture. By doing so, the accuracy of having unknown, unwanted small shapes is greatly reduced. By adjusting the slider that decides the limit for combining points and increasing or decreasing its value, a lot of holes or unwanted pixels can be suppressed. This holds true for more complicated shapes which require a lot of points to map its contour. Images showing the difference between using the forced single method and letting the program assume the picture can contain multiple models are shown in figure 5.8.



(a) Standard image generated when multiple shapes in one picture are allowed



(b) Forcing the detection of just one model

Figure 5.8: Difference between allowing multiple shapes and not allowing multiple shapes

The recursive method of finding points that are close to each other has been mentioned before. When being able to assume that only one model is presented, using a modified version of that function is possible. This means that shapes that previously were thought to be an independent shape, but that were actually part of the outer edge of a bigger contour, can now be fused together with points from the original contour that are in close proximity.

```
1  def eliminatePointsWhichAreClose(listOfNewEdges, listOfRelations):
2  #     eliminates Points which are close to eachother for single models
3      finalEdges = []
4      alreadyCheckedEdges = []
5      for i in range(len(listOfNewEdges)):
6          count = 0
7          for j in range(len(listOfNewEdges)):
8              currentX = listOfNewEdges[i][0]
9              currentY = listOfNewEdges[i][1]
10             nextX = listOfNewEdges[j][0]
11             nextY = listOfNewEdges[j][1]
```

```
12              if (i != j):
13                  pixelTolerance = cv2.getTrackbarPos('sizeTolerance','sliders')
14                  if (abs(currentX - listOfNewEdges[j][0]) < pixelTolerance and abs(
    currentY - listOfNewEdges[j][1]) < pixelTolerance):
15                      if (currentX not in alreadyCheckedEdges):
16                          deltaX = int(np.ceil((currentX + listOfNewEdges[j][0])/2))
17                          deltaY = int(np.ceil((currentY + listOfNewEdges[j][1])/2))
18                          if ([deltaX,deltaY] not in finalEdges):
19                              finalEdges.append([deltaX, deltaY])
20                          alreadyCheckedEdges.append(currentX)
21                          alreadyCheckedEdges.append(listOfNewEdges[j][0])
22                          listOfRelations = updateRelationBetweenNodes(listOfRelations,[
    currentX,currentY],[nextX,nextY],[deltaX,deltaY])
23                      count+=1
24          if (count == 0):
25              finalEdges.append([listOfNewEdges[i][0],listOfNewEdges[i][1]])
26      if (len(finalEdges) != len(listOfNewEdges)):
27          eliminatePointsWhichAreClose(finalEdges, listOfRelations)
28      else:
29          print ("DONE")
30      return finalEdges, listOfRelations
```

Listing 5.12: Function to fuse together points in close proximity

```
FinalEdg Multi:  [[[200, 15], [148, 63], [144, 132], [295, 307], [426, 167], [445, 76], [397, 20], [324,
37], [258, 24], [293, 91]], [[201, 17], [259, 28], [296, 91], [325, 29], [383, 16], [445, 83], [420, 17
1], [294, 303], [145, 128], [151, 62]]]
```

Figure 5.9: Output of final edges identified when multiple shapes are allowed

```
FinalEdg Single:  [[201, 16], [150, 63], [145, 130], [295, 305], [423, 169], [445, 80], [390, 18], [325,
33], [259, 26], [295, 91]]
```

Figure 5.10: Output of final edges when forcing to detect only one model

The shape detection algorithm is run on the image shown in figure 3.5b, which is a picture of a heart. The figure 5.9 shows what the output looks like when multiple shapes can be detected. The output is taken directly from the figures seen generated in figure 5.8a. Figure 5.10 has a much smaller list with a lot fewer points, the points correspond to the image generated in image 5.8b. This is the result of forcing the program to only consider one model in the picture. Since the program now assumes that only one model is present, it can fuse together points that were thought to be of different models first. The heart shape image is a good example, since the outer shape of the heart is a thick, black line that can confuse the program to detect two independent shapes.

5.8 The Four-Points Method

The four-points method is a simple method that can create a three-dimensional model a of cube based on only 4 edges that were identified from an image of a cube. The four points need to consist of the three points that are connected to the middle node and the middle node. The identification of the middle node is central for this approach to work. Multiple method have been implemented to identify the middle node quite reliable for all the test-images. More on that in section 5.9. Finding the other three nodes should be quite easy.

The list of relation created in this programs stores all the relation nodes have to each other. By plucking out nodes that have a connection to the middle node it should result in finding the last three nodes. This is done in listing 5.13. A write method has been implemented to write the special case of only four nodes to file. A slider in the control panel for the python program gives the user the option to either select the four-points method or not.

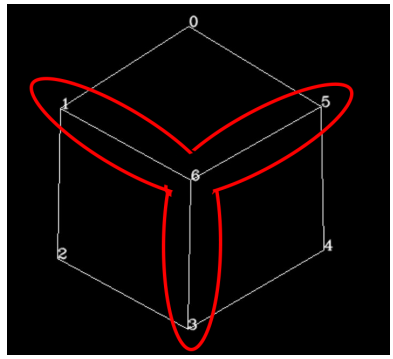


Figure 5.11: The four points needed to use the four-points method

The reason behind creating this method is to avoid the error prone implementation of identifying multiple nodes. Later on when a model of a cube is created, multiple nodes have to be identified. With this method only the middle node needs to be found. Another benefit originally was that the length between all the nodes could be quite error prone, and this method would circumvent that. As of today, the length between nodes does not have a big impact on the creation of models, so that benefit is not as important anymore. The drawing of edges that are not connected in the original shape is another factor that can produce errors in a three dimensional model. These wrongly identified edges can be due to an error in the list of relation node or due to the *findShapes* function.

```
1  def createFourPoints(listOfRelations, finalEdges, middleNode):
2      #finds the three points connected to the middle node
3      listOfRelations = flattenOnce(listOfRelations)
4      finalEdges = flattenOnce(finalEdges)
5      finalFourPoints = []
6      for k in range(len(finalEdges)):
7          if ([finalEdges[k], middleNode] in listOfRelations or [middleNode, finalEdges[
       k]] in listOfRelations):
8              if (finalEdges[k] not in finalFourPoints):
9                  finalFourPoints.append(finalEdges[k])
10     finalFourPoints.append(middleNode)
11     return finalFourPoints
```

Listing 5.13: Four-Points method

5.9 Finding and identifying nodes in contours

This section is about identifying the nodes in a picture of a cube, such as the one in figure 5.7. For modifying a picture of a cube into something three dimensional, locating the middle point was important. If found, the middle point would be able to locate the three

points connected to the middle point. If those point were found, the four-points method could be used and vectors could be applied to calculate an estimate for depth of the cube. Early in this project, being able to locate the middle node consistently was difficult. The first approach was to find every node besides the middle node. That way the only node of the 7 nodes not identified would be the middle node. To make sure it was the middle node was found, it was checked if it was in between the minimum and maximum points of the x- and the y-direction.

This small code snippet shows how minimum and maximum points could be found in the picture of the cube.

```
1  def findNodes(finalEdges):
2      topNode = bottomNode = finalEdges[0][0]
3      nodesOnTheLeft = []
4      nodesOnTheRight = []
5      nodesOnTheLeft.extend([finalEdges[0][0], finalEdges[0][1]]) #fills it
6      nodesOnTheRight.extend([finalEdges[0][0], finalEdges[0][1]]) #fills it
7      if (len(finalEdges[0]) == 7):
8          for i in range(1,len(finalEdges[0])):
9              if (finalEdges[0][i][1] > topNode[1]):
10                 topNode = finalEdges[0][i]
11             if (finalEdges[0][i][1] < bottomNode[1]):
12                 bottomNode = finalEdges[0][i]
13             if (finalEdges[0][i] not in nodesOnTheLeft):
14                 if (finalEdges[0][i][0] < nodesOnTheLeft[0][0] and nodesOnTheLeft
       [0][0] > nodesOnTheLeft[1][0]):
15                     nodesOnTheLeft[0] = finalEdges[0][i]
16                 elif (finalEdges[0][i][0] < nodesOnTheLeft[1][0] and finalEdges[0][i]
       not in nodesOnTheLeft):
17                     nodesOnTheLeft[1] = finalEdges[0][i]
18             if (finalEdges[0][i] not in nodesOnTheRight):
19                 if (finalEdges[0][i][0] > nodesOnTheRight[0][0] and nodesOnTheRight
       [0][0] < nodesOnTheRight[1][0]):
20                     nodesOnTheRight[0] = finalEdges[0][i]
21                 elif (finalEdges[0][i][0] > nodesOnTheRight[1][0]  and finalEdges[0][i
       ] not in nodesOnTheRight):
22                     nodesOnTheRight[1] = finalEdges[0][i]
23
24     if (max(nodesOnTheLeft[0][1],nodesOnTheLeft[1][1]) == nodesOnTheLeft[0][1]):
25         topLeftNode = nodesOnTheLeft[0]
26         bottomLeftNode = nodesOnTheLeft[1]
27     else:
28         topLeftNode = nodesOnTheLeft[1]
29         bottomLeftNode = nodesOnTheLeft[0]
30
31     if (max(nodesOnTheRight[0][1],nodesOnTheRight[1][1]) == nodesOnTheRight[0][1]):
32         topRightNode = nodesOnTheRight[0]
33         bottomRightNode = nodesOnTheRight[1]
34     else:
35         topRightNode = nodesOnTheRight[1]
36         bottomRightNode = nodesOnTheRight[0]
37
38     return topNode, bottomNode, topRightNode, topLeftNode, bottomRightNode,
       bottomLeftNode
```

Listing 5.14: Identifying maximum and minimum values

First the two maximum points both for the left side of the cube , low x-value, and the right side, high x-value were found. This was done from lines 13 to 22. The total maximum value of the cube, the top node is identified in lines 9 to 10, and the bottom on lines 11 to 12.

As can be seen from the logic in this code, this approach was quite error prone. A

single node of the other six could be misidentified and the whole process would be wrong. Errors could be frequent, such as identifying the top-left node both as the top node and the top-left node. Therefore another, better solution was sought. Such solutions would be the four-points method and the Ray Casting algorithm which both are used to identify a node inside a polygon. T

5.9.1 Identifying the middle node

The *Four-Points* method was implemented to find a way to create a cube which was less dependent of identifying which nodes lie where in the contour. Only one node would be needed to be identified correctly for this method to work. This node would for cubes be the *"middle node"*, the node in the middle of the contour. On figure 5.11 it would be node 6. By identifying this node a whole cube could be created. To consistently identify this middle node, two methods were implemented. First the Ray Casting Algorithm which uses a ray to check if a node is inside a polygon. As lack for a name for the second method, it was dubbed *"The alternative method"*. It could be called the *"Inside/Outside"* method, since it alternates a variable when it changes from the inside to the outside of the polygon.

The Ray Casting Algorithm

The Ray Casting Algorithm [7] is a relatively simple algorithm to test if a point in an unknown polygon is inside or outside of the structure. A ray intersects a point that is either inside or outside the polygon. This ray could go in any direction. If the ray intersects the polygons edge an odd number of times it means the point is inside the polygon. If is an even number, the point is outside the polygon. It is a simple but effective method. This

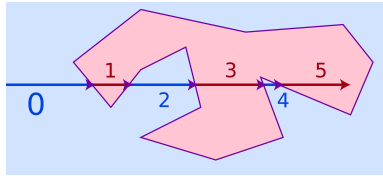


Figure 5.12: Visual representation of how many times a line crosses the shape of a polygon [22]

algorithm was implemented in the Python part of this project. It is based upon the code found on the web page *rosettacode* [8] available under the GNU license and it is modified to work in this thesis' environment.

```
if (not(finalEdges is None)):
    middleNodesCandFromRay = []
    for k in range(len(finalEdges)): #for each node do the check
        poly = Polygon()
        for i in range(len(finalEdges)):
            if (not(finalEdges[i] == finalEdges[k])):
                poly.AddPoint(Point(finalEdges[i][0],finalEdges[i][1]))
        if (PointInPolygon(poly, Point(finalEdges[k][0],finalEdges[k][1]))):
            middleNodesCandFromRay.append(finalEdges[k])
        poly = Polygon()
```

Listing 5.15: Ray Casting implementation in the project

This is only a part of the Ray Casting code, the rest is for the creation of polygons and not that relevant. This part of the code was completely modified to fit the structure of the nodes in the input list. A polygon is created as many times as there are points in the list. Each time, a single point is excluded from the creation of the polygon. This point is then tested against the current polygon to see if it lays outside or inside the polygon. For how the list of relation and the position of nodes inside the input list are placed, the only node inside the polygon should be the middle node. The sequence of nodes in the input list can be wrong, so the wrong nodes can be identified to be the middle node. This didn't happen very often, but to reduce the chance of error another method was implemented as well.

Alternative Method for identifying the middle node

First more out of curiosity another method to find the middle node was implemented. Since the Ray Casting method turned out to be not always 100% reliable, this method served as a second check to identify the middle node. The documentation of the idea behind this code is scarce. The codes origin is by W. Randolph Franklin. He describes the theory behind his code like this:

> "*I run a semi-infinite ray horizontally (increasing x, fixed y) out from the test point, and count how many edges it crosses. At each crossing, the ray switches between inside and outside. This is called the Jordan curve theorem.*" [9]

Another person called Adam Majewski [10] [11] implemented his own version of this code. The code that was implemented in the python part is a modification of that code to fit this projects variables and structure. The benefit with this code was that it is doing the same as the Ray Tracing algorithm, but applying a different logic behind it then using a ray. It also does this with just a few lines, and with a run-time that is significantly lower. Its results were quite reliable, and sometimes even more so than the Ray Casting algorithm. Using them together minimised the chance for an error as well.

```
middleNodesCandFromPath = []
    for i in range(len(finalEdges)):
        tempFinalEdges = []
        for j in range(len(finalEdges)):
            if (not(finalEdges[i] == finalEdges[j])):
                tempFinalEdges.append(finalEdges[j])
        bbPath = mplPath.Path(np.array(tempFinalEdges))
        if (bbPath.contains_point((finalEdges[i][0],finalEdges[i][1]))):
            middleNodesCandFromPath.append(finalEdges[i])
```

Listing 5.16: Alternative method to finding the middle node

This image can give a small understanding of this code. The image is not the basis for the code, but for a modification of the Ray Casting Algorithm, but it is a very similar implementation where the line changes its value from *true* to *false* upon crossing an edge.

Running both the Ray Casting algorithm and the alternative method on the same data gave a middle node that was consistently correctly identified. In general both methods would output a single middle node as a result, but in some edge cases more than one could be found by the methods. If that was the case, both results were compared, and if a node was part of the result of both methods, and no other node, this node is to be presumed

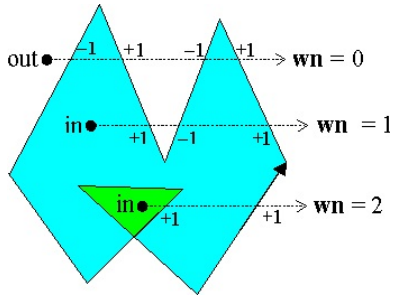


Figure 5.13: Alternative method to finding the middle node with a line changing its value upon crossing an edge [21]

to be the middle node. The figure 5.14 shows an example of how the output could look like. Here the Ray Casting Algorithm had two possible nodes as candidates, and the other method, *path*, had only one. This single node was also present in the result for the Ray Casting algorithm, so this node is presumed to be the middle node. With this middle-node that was now found, it could be used with the *"Four-Points"* method to create a three-dimensional cube.

```
DONE
MiddeNodeRay:  [284, 199]
MiddeNodeRay:  [200, 150]
MiddleNodePath:  [200, 150]
DONE
```

Figure 5.14: The output of both methods. The Ray Casting algorithm has two results, the alternative method only one

5.10 Main method

The main method for this program might be a bit misleading. Generally speaking the main method contains most of the code which should be executed by calling upon functions that implement all the functionality [18]. The first thing that happens upon running the program is that the shape detection is performed when the default value for a trackbar value is set. Each time a trackbar changes its value, the function *"onTrackbarChange"* is executed. This function contains the code to perform the shape detection. This has the benefit that when the user changes a value on the trackbar, the shape detection is performed with this new value. But this also means that the main function of the program is only used after most of the shape detection already is performed. The main function in this program is used to decide which write function is to be used and it performs all the necessary tasks to ensure that the correct data is written to file. This is based on the selection the user makes when using the program.

```
1 def main(finalEdges, listOfRelations):
```

```
2     updatedlistOfRelations = deleteDuplicates(listOfRelations)
3     generatedImage = drawLinesOnImage(listOfRelations, "Plain 2D image")
4     nameOfModelIdentified = identifyFinalModel(generatedImage)
5     if (cv2.getTrackbarPos('Single Model: No/Yes','sliders') == 1 and cv2.
       getTrackbarPos('Use 4 point method? No/Yes','sliders') == 1):
6         middleNode, listOfRelations, finalEdgesModified = middlePointFromRay(
       finalEdges, listOfRelations)
7         finalFourPoints = createFourPoints(listOfRelations, finalEdges, middleNode)
8         writeFourPointsToFile(finalFourPoints, nameOfModelIdentified)
9     elif (cv2.getTrackbarPos('Single Model: No/Yes','sliders') == 1):
10        _,listOfRelations, finalEdgesModified = middlePointFromRay(finalEdges,
       listOfRelations)
11        writeSingleModelToFile(listOfRelations, nameOfModelIdentified)
12    else:
13        cv2.destroyWindow("Plain 2D single shape image")
14        writeToFile(updatedlistOfRelations, nameOfModelIdentified)
```

Listing 5.17: The main method

Line 2 deals with lines that are duplicated in the list of relation, this is to avoid drawing a line twice. The next line in the code uses the *drawLinesOnImage* function, see 5.7, to create a picture of a cube after edges fuse together. Then the name of the final model based on this drawing is generated.

The values of two trackbars are deciding which method for writing to file should be used. If the trackbar *"Single model from file"* has the value 1, *true*, and the trackbar *"Use Four point method"* is also 1 the first if-sentence is entered. This first identifies the middle node with the help of the Ray Tracing algorithm and the alternative algorithm that was implemented. Line 7 then finds the last four points and returns them to a list. Finally, line 8 writes the points to file.

If the else-if is entered on line 9 this means that only one model is pictured on the image, and this is specified by the user. The list of relation is updated and then written to file.

The last part of the main function is the general write to file function and the function used unless specified otherwise by the user. It directly writes the list of relation to file.

5.11 Trackbars and the control panel

When it comes to the GUI, Graphical User Interface, part of OpenCV, this quote from a user on stackoverflow explains it quite well:

> *"OpenCV High(GUI) module is intended for debugging purposes only. If you need some good looking GUI, you should use a GUI library (e.g. Qt). It's pointless to make High(GUI) stuff looking good"* - Miki [19]

The user interface that OpenCV provides leaves a lot to be desired, and is only suitable for the most basic tasks. Since the program relies on some user feedback for an optimal result, sliders, called trackbars in the OpenCV environment, are implemented. Two main types of sliders have been implemented. The first being a slider with a high range of values changing the input for various filters and functions used in the detection of shapes. The other type of slides is a simple *Yes/No* option, where the 0 value stands for *No* and the 1 value stands for *Yes*. An example is when the four-points method should be used instead of

the normal method. If the user chooses to use this method, the slider with the label *"Use Four point method"* needs to have the value 1, for yes.

```
1 if (cv2.getTrackbarPos('Single Model: No/Yes','sliders') == 1 and cv2.getTrackbarPos('
      Use Four point method? No/Yes','sliders') == 1):
```

Listing 5.18: The selection of the user is mirrored by the value of the trackbar that was changed

In total 12 sliders haven been used in the main control panel, with six more in the colour panel changing the value of the HSV format, these six sliders were inspired by an example from [40]. This makes for a not very pretty, but very function oriented first layout of the program.

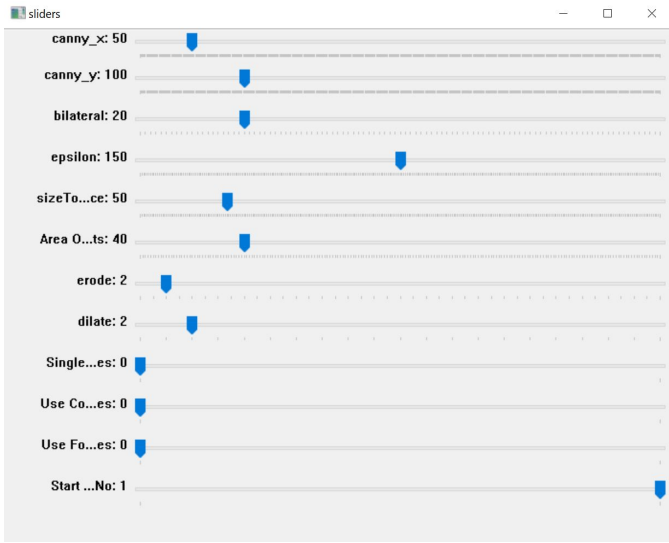


Figure 5.15: Main control panel used to change values for image recognition

The most important sliders, that give the most drastic changes to the result are the *epsilon-* and the "*sizeTo..*" sliders. The "*sizeTo..*" stands for *sizeTolerance*. This slider determines the range that points can be fused together to form a single point. By increasing the value of this slider, a simpler shape is found. Decreasing the value of this slider will allow for more points and a more detailed shape. The epsilon value changes the value of the constant epsilon that is used to approximate a polygonal curve. The difference when changing this value is also quite drastic. For an image with small shapes, or multiple shapes, the slider displayed here as *"Area O.."*, which stands for *"Area Of Objects"*, is important. It specifies a minimal area that objects have to be considered an independent contour. If a small shape is not found in an image, this sliders value has to be lowered to allow for this small models detection.

Below the control panel an image with text shows which models have been identified in the input image. This white image with text is updated each time a value of the control panel changes. If an image contains multiple shapes this is quite useful as it can give the user an overview if some shapes have been misidentified. This picture with text is created

Figure 5.16: Sliders used to manipulate the colours in the image

by printing the value of the shape identified to a blank picture together with a string of the shapes name, such as *triangle*. It might not look very elegant, but it helps the user a bit with helping to select the correct values for some sliders.

```
Objects found: 6
Triangle: 1
Rectangle: 2
Pentagon: 1
Hexagon: 0
Ellipse: 2
Undefined: 0
```

Figure 5.17: Blank, white image giving an overview over shapes identified in a picture

5.12 Runtime of the python program

Run-time has been mentioned multiple times in this project. It was made clear from the beginning that run-time was not a priority, but unnecessary slow functions should be avoided. A picture can be complex to analyse and the built in function to detect a contour can output a lot of points. A lot of points means a lot of calculation. This is definitely true for when nodes are checked to be added together. Using recursion helps a bit with making this function more effective, but in the end still a lot of calculations have to be done. Adding filters to an image greatly reduces the amount of possible contours and points detected, which means it also greatly improves run-time.

Each time the standard value for a slider is set in OpenCV, the *trackbarChange* function is performed, and the picture shape detection together with the combining points function

is run. Since 10 sliders have to get their initial value, the shape detection is run 10 times while the program starts up. At the beginning the value for the filters are not set, and each iteration of setting the initial value for a filter through a slider makes the program runs its shape detection process. Since the filters values are not set, the image detection is run in the first iteration without a single filter, which means it is performed on an raw image, resulting in a lot of contours and calculation. This is very costly, especially for complex models such as the fish shown in figure 3.5a. A solution by adding another slider was implemented. This slider is called *"Start Calculations Yes/No"*. By default no calculations are sure to be run as long as this slider has the value of 0, which is also its default value. After all the initial values for the other sliders have been set, the initial value of 1, *yes*, is set for the *start-calculation* slider. This means that the process of detecting the shape and combining points is only run once, when the final slider gets its value set to one. At that point all other sliders, which means all the filters as well, have their values and shape detection should be rapid. This also starts the program with a good first estimation of detecting the right shapes.

Chapter 6

Creating three-dimensional models in the GeoMod program

6.1 Reading from file

Before it is possible to create models from the shapes found in the previous chapter, these shapes first need to be read from a file into the GeoMod environment.

To read coordinates from a file, a read functionality had to be created. This functionality is implemented in the new class *generateModelsFromFile*. Its implementation and logic has been changed a few times throughout the project. Like the shape recognition part, it was important to start small. A simple rectangle with four edges was the basis for the first version of this read method. After more and more functionality was added the program, the read function had to take account of that. Reading three dimensional coordinates, strings and multiple models from file were the main cause for alternations in the code.

The theory behind the code is simple, but its implementation may not be so straight forward. The programs reads first the name of a model from file, then its coordinates. Next step it collects the users choice if a three-dimensional model should be created. These arguments are then passed on to the *generateModel* class. If multiple models are in a single file, all of them are created during the while loop.

The code to implement that functionality is given below.

```
1  QFile file(pathToPlugin);
2  if (!file.open(QFile::ReadOnly | QFile::Text)) {
3      std::cout << "File not open load" << std::endl;
4  }
5  qDebug() << pathToPlugin;
6  QTextStream in(&file);
7  int i = 0;
8  QString name = "unknown";
9  while (!in.atEnd()) {
10     QString temp = in.readLine();
11     if (!(47 < temp.at(0).unicode() && temp.at(0).unicode() < 58) && temp != "end") {
12         name = temp;
```

```
13      }
14      else if ((47 < temp.at(0).unicode() && temp.at(0).unicode() < 58) || (temp.at(0).
        unicode() == 45)) {
15          list.append(temp.toDouble());
16          i++;
17      } else {
18          if (list.isEmpty()) {
19              std::cout << "No coordinates were read from file, the list is empty" <<
        std::endl;
20          }
21          else {
22              list = resizeList(list, 10); // makes the read in model 10 times
        smaller
23              model = new generateModel(list, name, createMultidimensionalModel);
24          }
25          i = 0;
26          list.clear();
27          name = "unknown";
28      }
29  }
30  qDebug() << "end of file";
31  file.close();
```

Listing 6.1: Reading from file

Line 1 opens the file. The variable *pathToPlugin* is the path to the file chosen through the file browser. After the file is successfully opened, a while loop is initiated. It loops through each line of the text file until it reaches the files end.

First the lines value is added to a temporary variable *temp*. Line 11 checks if the first character of the temp variable is a not a number. If it is not a number and it is not a string with the value ”end”, then it should be the name of the model and the *name* variable is set to the value of *temp*.

The next *else if* at line 14 also checks if the current line is a string or number. If it is a string, this means that the models coordinates have been read and the end has been reached. It collects the numbers that were read into the temp variable and appends them to a *QList* called *list*.

The else statement at line 17 now checks first if the list is empty and prints out a statement to the console if this is the case. If it is not empty, it is almost ready to initialise the constructor of the *generateModel* class. Before initialising the constructor with the list, it has to be re-sized. This is done on line 22, by a method called *resizeList*.

```
1  QList<double> GenerateModelsWidget::resizeList(QList<double> list, int scalingInt) {
2      for (int var = 0; var < list.length(); ++var) {
3          list[var] = list[var]/scalingInt;
4      }
5      return list;
6  }
```

Listing 6.2: Re-sizing the list

This method takes a list as input and an int variable. The int variable specifies the factor for re-sizing. A value of 10 will make the coordinates 10 times smaller. It then returns the re-sized list. The re-sizing of the list is purely to make the model not too big for the camera view of the *GeoMod* program.

The modified list is ready to be send into the *generateModel* constructor. One thing still missing to initialise the constructor is that the *Boolean createMultidimensionalModel* has to get its value. This is done after having chosen a text file from the file browser. A

dialog box will pop up, prompting the user to choose if a 3D model of the coordinates that were read from file is to be created. If the answer is “Yes” it will be saved as true in a the *Boolean*, which will then be send to the *generateModel* class.

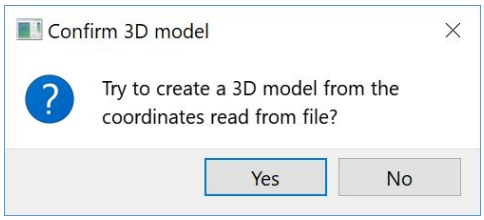


Figure 6.1: Asking the user if a three-dimensional model should be created

All variables for the initialisation of the *generateModel* class have now gotten their proper value and the classes constructor is initiated. This process is done for all the models that are read from the single text file chosen by the user.

6.2 Creating a depth for models

Creating a three-dimensional model from a single picture is no easy task. Extracting a depth from a picture is very difficult. There are many factors that can make this task impossible. Having *”a priori”* knowledge about what the picture depicts is a huge help. This way a depth can be calculated or added to something that appears two-dimensional on a picture. This section takes such a two-dimensional shape and creates an estimation for depth by calculating it through the help of vectors.

If a three-dimensional model is to be created, and the source of shape is not a hexagon, the *createGenericModelDepth* function is called. It takes the list of nodes that were created in the beginning of the constructor as an argument.

```
1 QList<QList<GeomNode*>> generateModel::createGenericModelDepth(QList<GeomNode*>
      listOfPtrNodes ) {
```

Listing 6.3: The createGenericModelDepth function

The function tries to create a depth for models. If a simple rectangle would be its input, its output should be a cube.

After declaring some variables and lists for later use, the function starts to declare points from which vectors can be created. This is done in a for-loop. After selecting the nodes, the cross product between the nodes is calculated. Note, the vectors that are created from the nodes always have one node in common. In this codes case it would always be the node called *node2*.

```
1 for (int i = 0; i < listOfNodesWithoutDuplicates.length();i++) {
2         QList<double> newPoint;
3         if (i == listOfNodesWithoutDuplicates.length()-2) {
4             node1 = listOfNodesWithoutDuplicates[i];
5             node2 = listOfNodesWithoutDuplicates[i+1];
6             node3 = listOfNodesWithoutDuplicates[0];
```

```
7         } else if (i == listOfNodesWithoutDuplicates.length()-1) {
8             node1 = listOfNodesWithoutDuplicates[i];
9             node2 = listOfNodesWithoutDuplicates[0];
10            node3 = listOfNodesWithoutDuplicates[1];
11        } else {
12            node1 = listOfNodesWithoutDuplicates[i];
13            node2 = listOfNodesWithoutDuplicates[i+1];
14            node3 = listOfNodesWithoutDuplicates[i+2];
15        }
```

Listing 6.4: Main code for detection of shapes

Since the list of nodes has the nodes in consecutive order, they will always connect to the next node in the list. The last node in the list also connects to the first node in the list.

6.2.1 Finding the direction and distance of the depth point

Next the direction of the depth needs to be calculated. For a two-dimensional shape that is placed in a known coordinate system consistent of two dimensions, the depth of a model can in theory be set directly. This is done by giving each point a value in the third dimension that is not set. If this is not the case, when a contour is translated or modified, the method with finding the normal vector of two vectors which have one node in common can always be used to find the general depth and direction. To find the normal vector the cross product between the two vectors we create from the three nodes *node1, node2* and *node3* is calculated [14].

$$\begin{aligned} vector1 &= (node2_i - node1_i)i + (node2_j - node1_j)j + (node2_k - node1_k)k \\ vector2 &= (node2_i - node3_i)i + (node2_j - node3_j)j + (node2_k - node3_k)k \end{aligned} \tag{6.1}$$

With the two vectors

$$\vec{a} = vector1, \vec{b} = vector2$$

that were created from the three points, the formula to find the cross product is applied.

$$\begin{aligned} \vec{a} &= a_i i + a_j j + a_k k \\ \vec{b} &= b_i i + b_j j + b_k k \end{aligned} \tag{6.2}$$

In matrix notation

$$\vec{a} \times \vec{b} = \begin{vmatrix} i & i & i \\ a_i & a_j & a_k \\ b_i & b_j & b_k \end{vmatrix} \tag{6.3}$$

More simply this can be written as

$$\vec{a} \times \vec{b} = i(a_j b_k - a_k b_j) - j(a_i b_k - a_k b_i) + k(a_i b_j - a_j b_i) \tag{6.4}$$

The implementation of the cross products is done in this function.

```
1 QList<double> generateModel::getCrossProduct(GeomNode * commonNode, GeomNode* node1,
      GeomNode* node2) {
2     QList<double> normalVector;
3     double vectorAx = commonNode->getModablXyzP()->get_x() - node1->getModablXyzP()->
      get_x();
4     double vectorAy = commonNode->getModablXyzP()->get_y() - node1->getModablXyzP()->
      get_y();
```

```
5     double vectorAz = commonNode->getModablXyzP()->get_z() - node1->getModablXyzP()->
      get_z();
6     double vectorBx = commonNode->getModablXyzP()->get_x() - node2->getModablXyzP()->
      get_x();
7     double vectorBy = commonNode->getModablXyzP()->get_y() - node2->getModablXyzP()->
      get_y();
8     double vectorBz = commonNode->getModablXyzP()->get_z() - node2->getModablXyzP()->
      get_z();
9     normalVector.append(vectorAy*vectorBz - vectorAz*vectorBy);
10    normalVector.append(vectorAz*vectorBx - vectorAx*vectorBz);
11    normalVector.append(vectorAx*vectorBy - vectorAy*vectorBx);
12    return normalVector;
13 }
```

Listing 6.5: Calculating the normal vector from three points

It takes three nodes as input, *commonNode*, which is the *node2* created earlier, and the nodes *node1* and *node2*, they correspond to *node1* and *node3* in the *createGenericModelDepth* function. From line 3 to line 8 the vectors *vector1* and *vector2* are created. Line 9 to 11 then calculate the normalvectors x-, y- and z-value. Each directional value is added to a *QList*. This *QList* is then returned on line 12.

Still inside the for-loop, the distance between the common vector and the two other vectors is calculated.

```
1 lengthOfVector2 = getDistanceBetweenTwoNodes(node2,node3);
2 lengthOfVector1 = getDistanceBetweenTwoNodes(node1,node2);
```

Listing 6.6: The distance between nodes is calculated

The distance is calculated by using the function *getDistanceBetweenNodes*. This function takes two nodes as input and returns a double which is the distance between them.

```
1 double generateModel::getDistanceBetweenTwoNodes(GeomNode* node1, GeomNode* node2) {
2     double xSquared = pow((node2->getModablXyzP()->get_x() - node1->getModablXyzP()->
      get_x()),2);
3     double ySquared = pow((node2->getModablXyzP()->get_y() - node1->getModablXyzP()->
      get_y()),2);
4     double zSquared = pow((node2->getModablXyzP()->get_z() - node1->getModablXyzP()->
      get_z()),2);
5     return sqrt(xSquared + ySquared + zSquared);
6 }
```

Listing 6.7: Function for calculating the distance between two nodes

This is a simple implementation of the function for distance between two points.

$$Distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{6.5}$$

The *"node2->getModablXyzP()->get_x()"* part might be a bit confusing. To get the x-value of a GeomNode in the GeoMod program form a pointer, this has to be done to get the value stored in the pointer address. As seen in the code for the listing 6.7.

After finding the two distances, we find the lowest value of those two values.

```
1 double distanceBetweenPoints = std::min(lengthOfVector1,lengthOfVector2);
```

Listing 6.8: Minimum of two values

The *distanceBetweenPoints* is the distance that the new point in space is located from the *node2*.

6.2.2 Finding the coordinates of the new depth point

Now that the direction of the point and the distance to the *node2* have been calculated, it is possible to find the coordinates of a new point which gives the model its depth.

The math behind finding the exact position of the new point is simple now that we have the direction and length of the depth. If d is the length of the depth and N is the normal vector. The coordinates for each direction of the new point, *Point*, are calculated.

$$Point_x = \frac{d * \vec{N}_x}{|N|}, Point_y = \frac{d * \vec{N}_y}{|N|}, Point_z = \frac{d * \vec{N}_z}{|N|} \tag{6.6}$$

Equivalently this is done in the C++ code.

```
1 if (lengthOfCrossVector != 0.0) {
2             if (crossProductVector[0] > 0) {
3                  newPoint.append((distanceBetweenPoints/lengthOfCrossVector)*(-1)*
      crossProductVector[0]+node2->getModablXyzP()->get_x()); //we only want the
      negative n-vector here
4             } else {
5                 newPoint.append((distanceBetweenPoints/lengthOfCrossVector)*
      crossProductVector[0]+node2->getModablXyzP()->get_x());
6             }
7             newPoint.append((distanceBetweenPoints/lengthOfCrossVector)*
      crossProductVector[1]+node2->getModablXyzP()->get_y());
8             newPoint.append((distanceBetweenPoints/lengthOfCrossVector)*
      crossProductVector[2]+node2->getModablXyzP()->get_z());
9         }
```

Listing 6.9: Finding the coordinate for the new point

Line 2 is necessary, since it was decided that the depth should always have its direction towards the negative side in the x direction in the GeoMods camera view.

After finding the coordinates for the new point, the node is then added to the models group pointer in line 1 for listing 6.10. In addition it is added to a list of all the new nodes created. The closest node to the new point, which is *node2*, is also added to a list. This way, both the new point and the closest point will have the same index.

```
1 ptrNode = modelGrpP->addNode(newPoint[0], newPoint[1], newPoint[2]);
2 closestPoint.append(node2);
3 newPointsList.append(ptrNode);
```

Listing 6.10: Adding the new point to the model group pointer

This is all still done in the for-loop, so a new depth point is created for every single node in the model. If a rectangular model was given as an input, four new points should have been created.

The final task of the function is to create a region for each side of the new model. If the original shape is a rectangle, in addition to the already existing front region, 5 more regions have to be created, which means 6 regions in total. Here the list with the closest point is very useful. Each region should consist of 4 points, the first being a single node from the closest point list, lets call it *first_node*. It is then connected to the node from the new point list with the same index, this will be the *second_node*. Now a new node, *third_node* from the new point list is connected to the *second_node*. The last node creating a closed region is the node from the closest point list with the same index as the *third_node*. This completes the region.

This code might make the explanation a bit clearer.

```
1  QList<QList<GeomNode*>> listOfRegions;
2      for (int var = 0; var < newPointsList.length(); ++var) {
3          QList<GeomNode*> regionList;
4          QList<GeomNode*> front;
5          QList<GeomNode*> back;
6          for (int j = 0; j < closestPoint.length(); ++j) {
7              front.prepend(closestPoint[j]);
8              back.append(newPointsList[j]);
9          }
10         if (var == newPointsList.length()-1) {
11             regionList.append(closestPoint[0]);
12             regionList.append(newPointsList[0]);
13             regionList.append(newPointsList[var]);
14             regionList.append(closestPoint[var]);
15         } else {
16             regionList.append(closestPoint[var+1]);
17             regionList.append(newPointsList[var+1]);
18             regionList.append(newPointsList[var]);
19             regionList.append(closestPoint[var]);
20         }
21         listOfRegions.append(front);
22         listOfRegions.append(regionList);
23         listOfRegions.append(back);
24     }
```

Listing 6.11: Creating a region

The first if-sentence at line 10 takes care of the edge case when the last node in the list needs to be connected to the first in the list. Line 21 adds the front region to the total region list. The front region consist of the original shape read from file. For a rectangle it is the square that can be seen in the y-z plane of the camera view. The for-loop on line 2 makes sure that every two nodes from the original nodes list create a region with its two depth nodes. The amount of regions created can vary from a simple 6 sides to create a cube shape to a more complex and abstract figure with 20 regions, like a heart shape.

A list full of all the regions created is returned to the constructor. With this list the final model that is seen by the user is created.

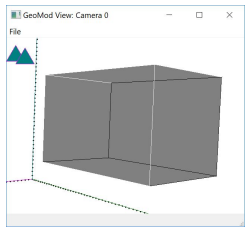
This portion of code inside the classes constructor creates the edges that make up a region, and creates this region for the model by adding it to the final model group pointer.

```
1  if (listOfRegions.length() != 0) {
2      for (int j = 0; j < listOfRegions.length();j++) {
3          QList<GeomEdge*> listOfPtrEdgesForRegion;
4          for (int i = 0; i < listOfRegions[j].length(); i++) {
5              if (i == listOfRegions[j].length()-1) {
6                  ptrEdge = modelGrpP->addEdge(listOfRegions[j][i], listOfRegions[j
       ][0], std::to_string(i));
7              } else {
8                  ptrEdge = modelGrpP->addEdge(listOfRegions[j][i], listOfRegions[j][i
       +1], std::to_string(i));
9              }
10             listOfPtrEdgesForRegion.insert(i,ptrEdge);
11         }
12         GeomRegion* regionP = modelGrpP->addRegion(&listOfPtrEdgesForRegion[0],
       listOfPtrEdgesForRegion.length(), first_against, "region");
13         regionP->setOutColorRGB_byName("grey");
14     }
15 }
```

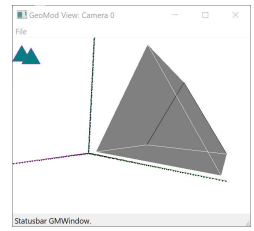
Listing 6.12: Creating a region

First, line 1 checks if regions have been created by checking if the list of regions has any content. Next, a for-loop loops through all the regions in the list. Then, another for-loop goes inside each region. Edges are created from the nodes inside the region. Line 5 takes care of the edge case when the last node in the regions list has to be connected to the first node in the regions list. After all the edges that make up a region have been created, a *GeomRegion* is created from the list of edges that were just created. This is done on line 12, using the built in *GeoMod* function *addRegion* to create the region for the model. The *first_against* variable is a boolean which tells the *addRegion* method wether the first edge is along or against the cycle. In this case it is against the cycle. Line 13 adds a colour to the cubes outer side, here the colour *grey* was chosen.

Running this function on the image of a simple rectangle and a simple triangle gives these results. More images can be seen in the result chapter, chapter 7.



(a) Rectangle created to be three-dimensional



(b) Triangle created to be three-dimensional

Figure 6.2: Generic three-dimensional models of 2D pictures

6.3 Creating a three dimensional cube

This chapter explains the method *createACubeWithDepth* that tries to create a three dimensional model of a cube based on a two-dimensional shape. This is only possible because of the *"a priori"* knowledge behind the shape that is detected by the Python algorithms that are implemented. Knowing an image contains the shape of a cube in a set position makes the extraction of depth possible. Also having the knowledge of how exactly a cube looks like provides the basis for most of the operations performed here to estimate a depth.

This methods design and functionality is very similar to the method called *createGenericModelDepth* that was explained in 6.2. The similarity is due to the method *createACubeWithDepth* being the first generation of the more generalised class *createGenericModelDepth*.

The method starts off with creating a second list of nodes. This list of nodes does not contain any duplicates of any node, a function called *deleteDuplicates* makes sure of that. This means that the list should contain 7 elements, all unique.

This list without duplicates is then used to identify which nodes are placed where. Since the shape is known, this should be rather simple . We divide the nodes into *topLeftNode*, *bottomLeftNode*, *topRightNode*, *bottomRightNode*, *bottomNode*, *topNode* and *middleNode*.

```
GeomNode* topLeftNode = findTopLeftNode(listOfNodesWithoutDuplicates);
GeomNode* bottomLeftNode = findBottomLeftNode(listOfNodesWithoutDuplicates);
GeomNode* topRightNode = findTopRightNode(listOfNodesWithoutDuplicates);
GeomNode* bottomRightNode = findBottomRightNode(listOfNodesWithoutDuplicates);
GeomNode* bottomNode = findBottomNode(listOfNodesWithoutDuplicates);
GeomNode* topNode = findTopNode(listOfNodesWithoutDuplicates);
GeomNode * middleNode = findMiddleNode(listOfNodesWithoutDuplicates, topNode,
    bottomNode, topLeftNode, topRightNode ,bottomLeftNode, bottomRightNode);
```

Listing 6.13: Identifying nodes

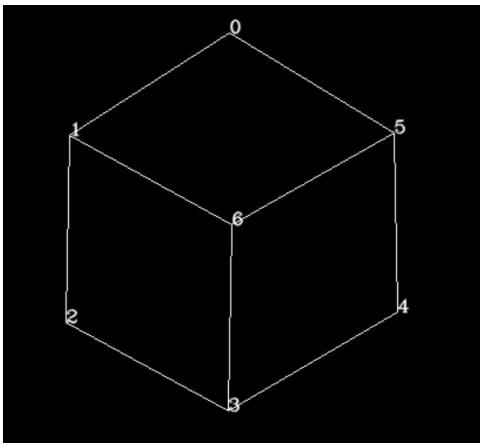


Figure 6.3: This image of a 2D cube is the basis for the 7 nodes, from 0 to 6

Once all of the nodes are identified, the nodes can be manipulated. First, an estimation of depth is done by taking the distance between the bottom-left node and the top-left node and dividing it by two. This depth is of course only an estimation and a start point to find a better, final depth value. The direction of the depth is in the x-direction of the *GeoMod* program, this is done because it is easier to visualise a model in the y-z plane of the camera view.

Re-positioning the middle node and the top node so they are on the same height as the top-left and top-right nodes is the next step. The bottom node is also placed on the same height as the bottom-left and bottom-right node. Now all nodes should be on the same two levels. This is all done in the following code snippet.

```
double xLength = getDistanceBetweenTwoNodes(bottomLeftNode, topLeftNode)/2;
double z_averageTop = (topLeftNode->getModablXyzP()->get_z() + topRightNode->
    getModablXyzP()->get_z())/2;
double z_averageBottom = (bottomLeftNode->getModablXyzP()->get_z() + bottomRightNode->
    getModablXyzP()->get_z())/2;
updateNodesInList(middleNode,xLength,middleNode->getModablXyzP()->get_y(),z_averageTop
    );
updateNodesInList(topNode,-xLength,topNode->getModablXyzP()->get_y(),z_averageTop);
updateNodesInList(bottomNode,xLength,bottomNode->getModablXyzP()->get_y(),
    z_averageBottom);
```

Listing 6.14: Updating the values of some nodes

Line 4 to 6 update the nodes values by using an implemented function called *updateNodesInList*. It takes the node that is going to be updated as first argument, then the new x-, y- and z-values of that node.

6.3.1 Creating right angles between edges

The next step to make the model cube shaped is to create a right angle for the top region of the cube. This region is made up of the middle node, the top-left node and the top-right node. First the angle between the two vectors created from the three nodes is calculated. This is done by using the function *findAngleBetweenThreeCoordinates*. This function takes three nodes as input, one of the nodes has to be connected to both the other nodes. Two vectors are then created, and the angle between the two vectors is calculated by using the dot-product function for vectors.

$$a \cdot b = |a||b|\cos\alpha \tag{6.7}$$

The C++ implementation of this equation in this projects code looks like this.

```
1  double generateModel::findAngleBetweenThreeCoordinates(GeomNode * commonNode, GeomNode
       * node1, GeomNode * node2) {
2  double vectorAx = commonNode->getModablXyzP()->get_x() - node1->getModablXyzP()->get_x
       ();
3  double vectorAy = commonNode->getModablXyzP()->get_y() - node1->getModablXyzP()->get_y
       ();
4  double vectorAz = commonNode->getModablXyzP()->get_z() - node1->getModablXyzP()->get_z
       ();
5  double vectorBx = commonNode->getModablXyzP()->get_x() - node2->getModablXyzP()->get_x
       ();
6  double vectorBy = commonNode->getModablXyzP()->get_y() - node2->getModablXyzP()->get_y
       ();
7  double vectorBz = commonNode->getModablXyzP()->get_z() - node2->getModablXyzP()->get_z
       ();
8  double dot = vectorAx*vectorBx + vectorAy*vectorBy + vectorAz*vectorBz;
9  double lengthA = sqrt(vectorAx*vectorAx + vectorAy*vectorAy + vectorAz*vectorAz);
10 double lengthB = sqrt(vectorBx*vectorBx + vectorBy*vectorBy + vectorBz*vectorBz);
11 double angleInRadians = acos(dot/(lengthA*lengthB));
12 double angle = (180/(atan(1)*4))*angleInRadians;
13 return angle;
14 }
```

Listing 6.15: Finding the angle between three nodes

The function returns the angle in degrees, because visually this was easier to work with. Line 2 to 7 create two vectors called A and B the same way like the theory in 6.1. The implementation is straight forward from the lines 8 to 11. Line 12 converts radians to degrees by using

$$\pi = \frac{180}{(\arctan(1) * 4)} \tag{6.8}$$

To create a right angle, an iterative approach was chosen. By increasing or decreasing the x-value of the middle node with a small value *deltaX* the angle between the top-left node, the middle node and the top-right node will approach a value of 90 degrees. This is done inside a while loop.

```
1  bool stop = true;
2  while(stop) {
3      if (abs(findAngleBetweenThreeCoordinates(middleNode,topLeftNode,topRightNode) -
       90) > 0.5) {
4          if (findAngleBetweenThreeCoordinates(middleNode,topLeftNode,topRightNode) >
       90) {
5              double tempX = middleNode->getModablXyzP()->get_x();
6              updateNodesInList(middleNode,tempX+0.1,middleNode->getModablXyzP()->get_y
       (),middleNode->getModablXyzP()->get_z());
7          } else if (findAngleBetweenThreeCoordinates(middleNode,topLeftNode,
       topRightNode) < 90) {
8              double tempX = middleNode->getModablXyzP()->get_x();
9              updateNodesInList(middleNode,tempX-0.1,middleNode->getModablXyzP()->get_y
       (),middleNode->getModablXyzP()->get_z());
10         }
11     } else {
12             stop = false;
13         }
14 }
```

Listing 6.16: Creating a right angle

Line 3 checks if the angle is in the range of 89.5 and 90.5. If this is the case, the while loop stops and an angle of around 90 degrees is found. If the angle is not in range, the else statement at line 4 checks if the angle is higher than 90 degrees. If it is higher, the x-value of the middle node is increased by a small *deltaX*, 0.1, as seen in line 6. This should decrease the angle. The contrary is done if the angle is below 90 degrees. After a few iterations the value of the angle should reach close to 90 degrees and the while loop stops.

The next step is to find the vector that stands orthogonal on the plane created by the three nodes: middle, top-left and top-right. By taking the cross product between the two vectors created by the three nodes the orthogonal vector is found. The function to find the normal vector with three points has been explained earlier in section 6.2.1.

The distance to the new point below the plane should be the distance between the middle node and the bottom node. The bottom node should be moved only a very short distance to place it directly below the plane along the direction of the planes normal vector.

Doing this gives us three planes that are connected with a right angle. Four nodes are now in the right position to form the final model: the middle node, the top-right node, the top-left node and the bottom node.

Using those 4 nodes, it is now possible to calculate the remaining 4 nodes that can create a full model.

6.3.2 Finding the missing points by calculation

Three of the missing points are points that will complete the three planes that have a right angle. These points have a x-, y- and z-coordinate, all of which are unknown. This means that three equations are needed to solve them. Since it is known that the final shape is a cube, some estimations to set up three equations to solve the problem can be made.

Lets say there a three points, or nodes; A, B and C. The three points create two vectors like shown in the picture below.

The angle BAC is known to be 90 degrees, because of the work that was done in the previous section. The distance from A to the new point, D, is also known. It has to be the

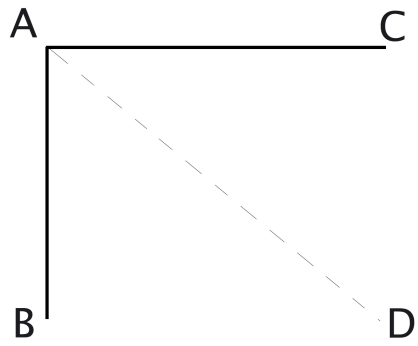


Figure 6.4: Two vectors created from the points AC and AB. The coordinates of point D are unknown

same as the distance from point *B* to point *C*. The angle *DAC* can be calculated by using the dot product. The second angle *BAD* can also be calculated by the dot product, or by taking 90 degrees minus the first angle that was found. The last equation that is needed is the equation for a plane created by the the two vectors.

Three equations can now be set up to find the x-, y- and z-coordinates of point D.

$$\vec{AC} \cdot \vec{AD} = |\vec{AC}||\vec{AD}|\cos\alpha \tag{1}$$

$$\vec{AB} \cdot \vec{AD} = |\vec{AB}||\vec{AD}|\cos\beta \tag{2}$$

$$a(x - x_0) + b(y - y_0) + c(z - z_0) = 0 \tag{3}$$

where we know that

$$|\vec{AD}| = |\vec{BC}| \tag{4}$$

and that in equation 3

$$\vec{n} = \langle a, b, c \rangle \tag{5}$$

The angle α can be found using simple geometry. With this angle, the second angle β can be found. The *N* vector can easily be computed by taking the cross product between the two vectors *AC* and *AB*. Using 4, the length of vector *AD* can be substituted with the length of vector *BC*, which is known. This means that the three unknown variables x, y and z are the only unknown variables present in each of the three equations. Solving these equations by hand is a lengthy process, but not a very difficult one. To solve these linear equations inside of the GeoMod code, the Cramer's rule [16] is implemented. This method uses matrices and determinant to solve the equations. The implementation is explained briefly below.

Given that there is this set of linear equations

$$\begin{cases} a_1x + b_1y + c_1z = d_1 \\ a_2x + b_2y + c_2z = d_2 \\ a_3x + b_3y + c_3z = d_3 \end{cases} \tag{6.9}$$

In matrix format this is

$$\begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix} \tag{6.10}$$

The three unknown values for the new point with the coordinates *(x,y,z)* can now be found like this

$$x = \frac{\begin{vmatrix} d_1 & b_1 & c_1 \\ d_2 & b_2 & c_2 \\ d_3 & b_3 & c_3 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}, y = \frac{\begin{vmatrix} a_1 & d_1 & c_1 \\ a_2 & d_2 & c_2 \\ a_3 & d_3 & c_3 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}}, z = \frac{\begin{vmatrix} a_1 & b_1 & d_1 \\ a_2 & b_2 & d_2 \\ a_3 & b_3 & d_3 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}} \tag{6.11}$$

The Cramer's rule and the three equations needed to solve the three unknowns are implemented like this in the C++ code section.

```
QList<double> generateModel::cramersRule(GeomNode* A, GeomNode* B, GeomNode* C) {
//    solves three linear equations to find the coordinates of a new point
    QList<double> NVector = getCrossProduct(A,B,C); //first finding the normal vector
    double lengthAB = getDistanceBetweenTwoNodes(A,B);
    double lengthAC = getDistanceBetweenTwoNodes(A,C);
    double lengthBC = getDistanceBetweenTwoNodes(B,C);
    double angle1 = acos(lengthAC/lengthBC)*(180/(atan(1)*4)); //finding the angle by
     simple cosinus math
    double angle2 = 90-angle1;
    double a1 = C->getModablXyzP()->get_x()-A->getModablXyzP()->get_x(); //ACx
    double b1 = C->getModablXyzP()->get_y()-A->getModablXyzP()->get_y(); //ACy
    double c1 = C->getModablXyzP()->get_z()-A->getModablXyzP()->get_z(); //ACz
    double a2 = B->getModablXyzP()->get_x()-A->getModablXyzP()->get_x(); //ABx
    double b2 = B->getModablXyzP()->get_y()-A->getModablXyzP()->get_y(); //ABy
    double c2 = B->getModablXyzP()->get_z()-A->getModablXyzP()->get_z(); //ABz
    double a3 = NVector[0];
    double b3 = NVector[1];
    double c3 = NVector[2];
    double d1 = -cos(angle1*((atan(1)*4)/180))*lengthAC*lengthBC -
            A->getModablXyzP()->get_x()*a1 - A->getModablXyzP()->get_y()*b1 - A->
     getModablXyzP()->get_z()*c1;
    double d2 = -cos(angle2*((atan(1)*4)/180))*lengthAB*lengthBC -
            A->getModablXyzP()->get_x()*a2 - A->getModablXyzP()->get_y()*b2 - A->
     getModablXyzP()->get_z()*c2;
    double d3 = - NVector[0]*A->getModablXyzP()->get_x() - NVector[1]*A->getModablXyzP
     ()->get_y() -
            NVector[2]*A->getModablXyzP()->get_z();

    double D = (a1*b2*c3+b1*a3*c2+c1*a2*b3)-(a1*c2*b3+b1*a2*c3+c1*b2*a3);
    double x = ((b1*c3*d2+c1*b2*d3+d1*c2*b3)-(b1*c2*d3+c1*b3*d2+d1*b2*c3))/D;
    double y = ((a1*c2*d3+c1*a3*d2+d1*a2*c3)-(a1*c3*d2+c1*a2*d3+d1*c2*a3))/D;
    double z = ((a1*b3*d2+b1*a2*d3+d1*b2*a3)-(a1*b2*d3+b1*a3*d2+d1*a2*b3))/D;

```

```
30      QList<double> newPointCoord;
31      newPointCoord.append(x); newPointCoord.append(y); newPointCoord.append(z);
32      return newPointCoord;
33  }
```

Listing 6.17: The implementation of solving three linear equations

Line 3 calculates the N vector between the vectors *AD* and *AB*. Line 2 to 8 calculates the angle between the two vectors by simple cosines geometry. Then each direction of the vector *AD* and *AB* get their value. Line 18 defines the value of the constant *d1* by using equation 1. The same for line 20, where *d2* gets its value by equation 2. Line 23 is a bit more tricky, this constant *d3* uses the equation for a plane 3 to find its value.

After initialising all values used in the Cramer's rule, it is implemented in lines 25 to 28. Line 25 first determines the determinant of the matrix with all the known constants. The other three lines can then compute the values for x, y and z. The function then appends the values for the new point to a list at line 31 and returns this list on line 32.

A new point has now be found. This method is then repeated three more times.

```
1  QList<double> newPointBottomLeft = cramersRule(middleNode,topLeftNode,bottomNode);
2  updateNodesInList(bottomLeftNode,newPointBottomLeft[0],newPointBottomLeft[1],
       newPointBottomLeft[2]);
3
4  QList<double> newPointTop = cramersRule(middleNode,topRightNode, topLeftNode);
5  updateNodesInList(topNode,newPointTop[0],newPointTop[1], newPointTop[2]);
6
7  QList<double> newPointBottomRight = cramersRule(middleNode,bottomNode, topRightNode);
8  updateNodesInList(bottomRightNode,newPointBottomRight[0],newPointBottomRight[1],
       newPointBottomRight[2]);
9
10 QList<double> finalPointList = cramersRule(topLeftNode,bottomLeftNode,topNode);
11 GeomNode* finalPoint = modelGrpP->addNode(finalPointList[0],finalPointList[1],
       finalPointList[2]);
```

Listing 6.18: Finding new points by cramers rule

6.3.3 Creating regions

From the original four points, four additional points were calculated. This means that all points to create a complete cube have been found. Since all points needed are present, it is time to create regions. A cube needs six regions. To create for example the top region of the cube, the code looks like this.

```
1  QList<GeomNode*> region3;
2  region3.append(middleNode);
3  region3.append(topLeftNode);
4  region3.append(topNode);
5  region3.append(topRightNode);
```

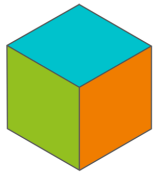
Listing 6.19: Creating a region

This is then done for the other 5 regions. Finally, the regions are collected into a list, and the list of regions is then returned to the constructor. Inside the constructor, the function at 6.12 creates the model and a cube shaped model should now appear in the camera view.

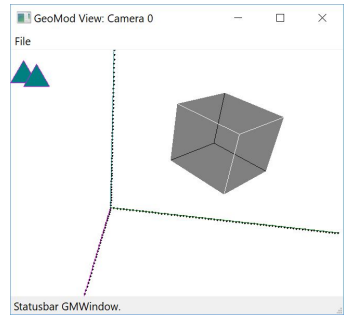
```
QList<QList<GeomNode*>> listOfRegions;
listOfRegions.append(region1);
listOfRegions.append(region2);
listOfRegions.append(region3);
listOfRegions.append(region4);
listOfRegions.append(region5);
listOfRegions.append(region6);
```

Listing 6.20: Collecting all the regions to a list

Below the result of running this function is shown. More images are shown in chapter 7.



(a) Source image of the cube created



(b) The cube created from a single 2D picture

Figure 6.5: A 2D image has become three-dimensional

6.4 The four-points method

The four-points method has been mentioned earlier in section 5.8. This is the C++ implementation that deals with the four nodes that were found in the python part.

This method was implemented to experiment if it was possible to create a cube from just 4 points, no matter what the orientation the cube has. The four points the method needs are read from the text file and originate from the calculations from the python code. Here, the first three nodes read from the file are the nodes that are connected to the middle node. The last node in the file has to be the middle node. This middle node is found in the python part of the code, using both the ray-tracing algorithm and the theory behind W. R. Franklins method. Using the combination of those two implementations, the middle node could be identified quite reliably, and the four points should be correct. A benefit of using this technique is that only one node needs to be identified, instead of the seven nodes like the implementation in section 6.3. This should should make the four-points method less error prone, and that is the reason it originated in the first place.

The four-points methods implementation is quite similar to the implementation in section 6.3. But the first difference can already be spotted in the beginning of the code. To align the middle node to the same height as two of the other nodes, vector calculation is used again. The four nodes are *node1*, *node2*, *node3* and *middleNode*. A vector is created using the nodes *node1* and *node2*. To elevate the middle node to the same height as *node1*

and *node2* we place it on to the vector that the nodes create. Since the *middleNode* does not have any depth yet, the first value of depth is to be estimated to be half the distance between *node1* and *node2*.

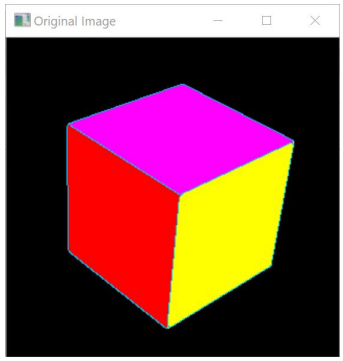
```
1  GeomNode* node1 = listOfPtrNodes[0];
2  GeomNode* node2 = listOfPtrNodes[1];
3  GeomNode* node3 = listOfPtrNodes[2];
4  GeomNode* middleNode = listOfPtrNodes[3];
5  double d = getDistanceBetweenTwoNodes(middleNode,node3);
6  double d2 = getDistanceBetweenTwoNodes(node1,node2);
7  double vectorAy = node2->getModablXyzP()->get_y() - node1->getModablXyzP()->get_y();
8  double vectorAz = node2->getModablXyzP()->get_z() - node1->getModablXyzP()->get_z();
9
10 updateNodesInList(middleNode,d2/2, node1->getModablXyzP()->get_y() + vectorAy/2,node1
       ->getModablXyzP()->get_z() + vectorAz/2);
```

Listing 6.21: Aligning the *middleNode* with the other nodes

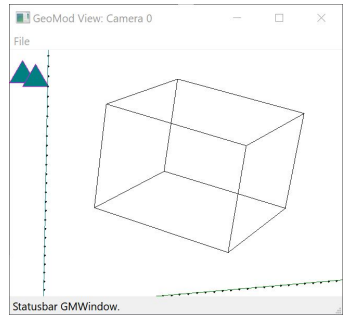
To maintain the distance, *d*, from the *middleNode* to *node3*, the original distance was first saved to a variable, line 5. After that a vector from the *middleNode* to the *node3* is created. To preserve the direction of *middleNode* and *node3*, *node3* is moved along the vector between the two nodes until its distance to the *middleNode* is as far as the original distance *d*.

Next, a modification of the function to iteratively determine a right angle between two nodes is used. This modification is due to determining which direction should be increased by a small value *delta* to create a right angle. In the previous section it was always clear that the small *delta* had to be added to the value in the x-direction. This time that was also very likely, but just to keep the created cube more like its original shape, a function called *checkWhichOrientationNeedsToBeChanged* was created to determine which direction needed to be increased to create a right angle. This function returns an int telling an if sentence which direction needs to be changed. As per the test-images for this projects, the answer would always be the x-direction, but the function was tested for edge-cases and should hold up for more complicated tasks in the future.

After determining a depth value for the *middleNode*, this depth value was taken to update the value of *node3*. At this point there are still 4 points missing, and these are calculated the same way as in section 6.3.2, using Cramer's rule. Regions are also created the same way as in section 6.3.3. The results of using this function to create a cube are very similar to the results in the last section. The length of the sides of the cube may differ in length, but both functions are very reliable to create a cube. The big difference is that in this section only 4 points are needed to create a cube, and it can more reliably create cubes without edge errors.



(a) The original image



(b) Model created by the four-points method

Figure 6.6: Using the four-points method

6.5 Estimating the volume for a polygon

An estimation of the volume for a polygon was implemented to give an approximation of the volume of the model shown in GeoMod. The reason behind implementing this is showcasing a potential use in the real world. More on that in section 8.1 about potential application in the real world. The function to approximate the area for a simple, non-intersecting polygon [20] is given by :

$$A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i), \text{ where } x_n = x_0 \text{ and } y_n = y_0 \tag{6.12}$$

Its implementation looks as follows:

```
1  double generateModel::calculateAreaOfSurface(QList<GeomNode*> listOfNodes) {
2      for (int i = 0; i < listOfNodes.length(); i++) {
3          if (i == listOfNodes.length()-1) {
4              area += 0.5*(listOfNodes[i]->getModablXyzP()->get_y()*listOfNodes[0]->
       getModablXyzP()->get_z()
5                      - listOfNodes[0]->getModablXyzP()->get_y()*listOfNodes[i]->
       getModablXyzP()->get_z());
6          } else {
7              area += 0.5*(listOfNodes[i]->getModablXyzP()->get_y()*listOfNodes[i+1]->
       getModablXyzP()->get_z()
8                      - listOfNodes[i+1]->getModablXyzP()->get_y()*listOfNodes[i]->
       getModablXyzP()->get_z());
9          }
10     }
11     return area;
12 }
```

Listing 6.22: Calculating the area of a polygon

Here, only the area in the y-z coordinate orientation is calculated. Once a model gets its depth, the average depth is calculated for the complete model. The volume is then found by taking the models area and multiplying it by the average depth. This gives a good estimation of the volume, which of course is more correct for simpler models and

less correct the more advanced the polygon is. The area and volume is printed to the console after the creation of the model.

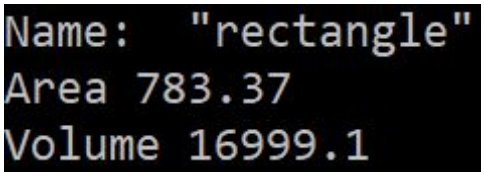


Figure 6.7: This is what the output looks like. These values were found for the simple rectangle in figure 3.1a

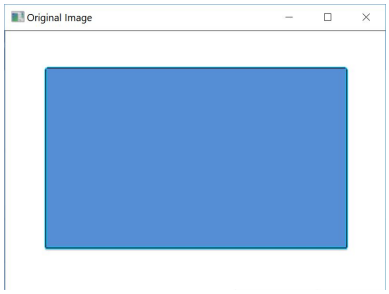
Chapter 7

Results

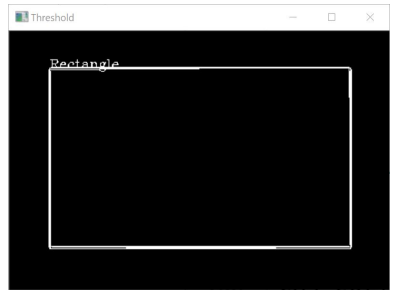
This chapter presents the results found during this thesis. For each method a few pictures show the process from a normal two-dimensional picture to its three-dimensional model. More results are shown in the appendix, section 1.4.

7.1 Results of creating generic models

Here the results of the *createGenericModelDepth* function are presented. This function takes a general shape as input and creates a depth using the normal vector of two vectors on the same plane as the shape. The models depth is decided by the length of the neighbouring edge. This usually gives a good estimate for simple forms such as rectangles, pentagons and ellipses. The depth of an advanced form such as a fish is less realistic. The presentation is in chronological order. The first image is the original image, the second one shows the threshold created from that image with all filters applied. The third image is the calculated image where duplicated nodes and lines have been eliminated. The last images are the three-dimensional approximations of the two-dimensional shape.

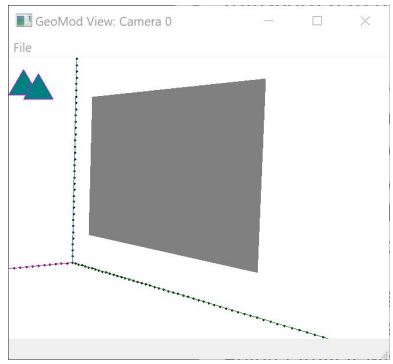


(a) Original Picture

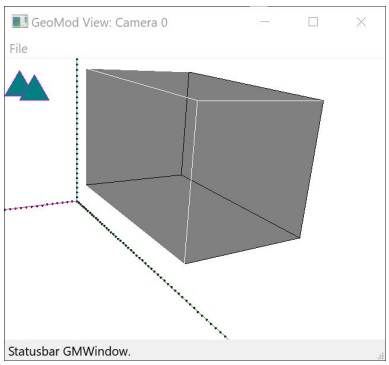


(b) Threshold

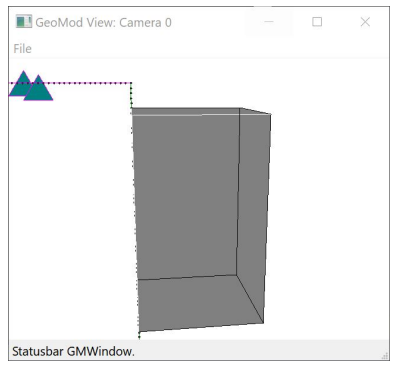
(c) Calculated Image



(d) Two Dimensional

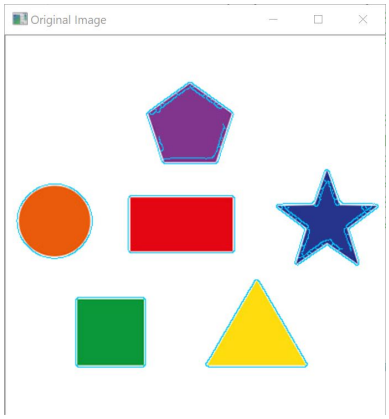


(e) Three dimensional front

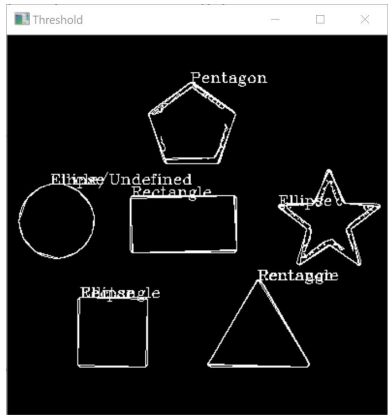


(f) Three dimensional top

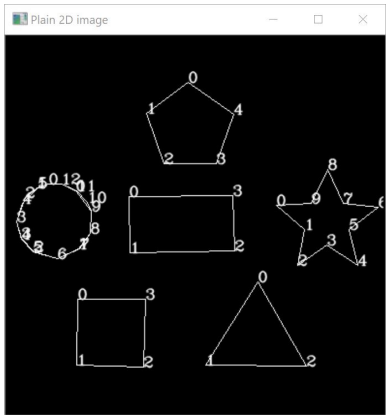
Figure 7.1: A normal rectangular shape



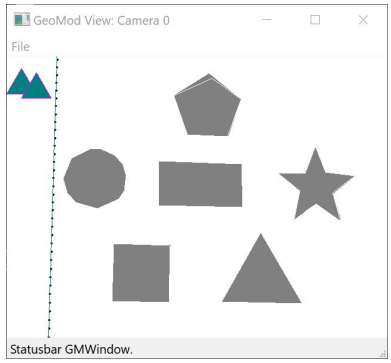
(a) Original Picture



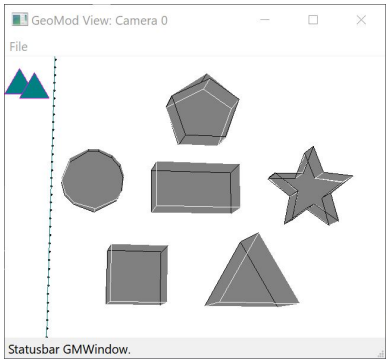
(b) Threshold



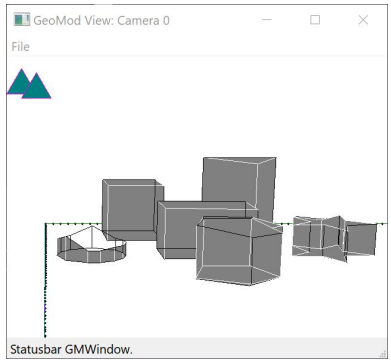
(c) Calculated Image



(d) Two Dimensional



(e) Three dimensional front

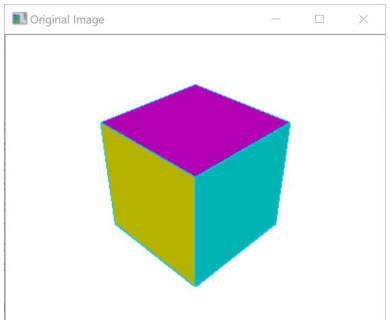


(f) Three dimensional top

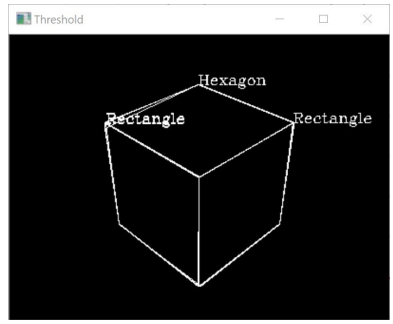
Figure 7.2: Multiple shapes in a single image

7.2 Three dimensional cubes

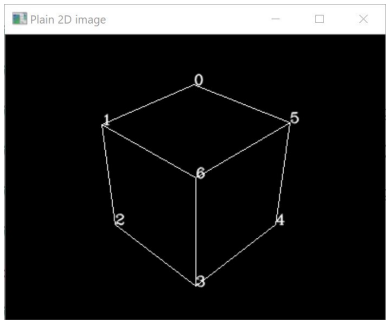
Here the function *createACubeWithDepth* is used to create three-dimensional cubes. The order of the images is chronological and the same as section 7.1. The first image is of a simple cube with three differently coloured sides. The last is of a cube that has one uniform colour, red.



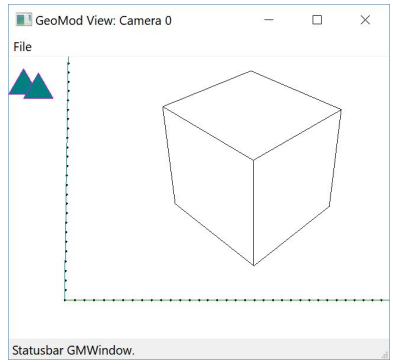
(a) Original Picture



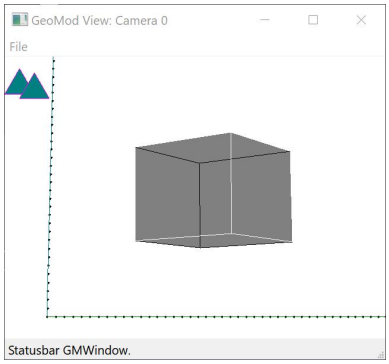
(b) Threshold



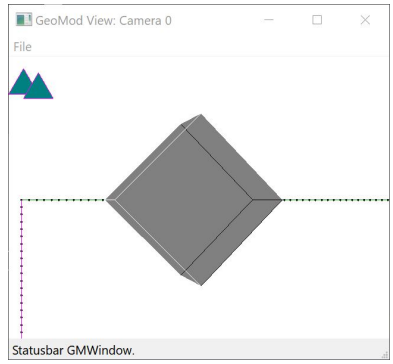
(c) Calculated Image



(d) Two Dimensional

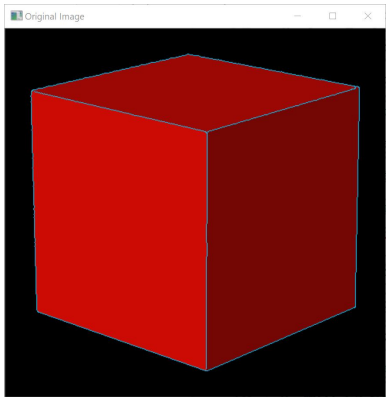


(e) Three dimensional front

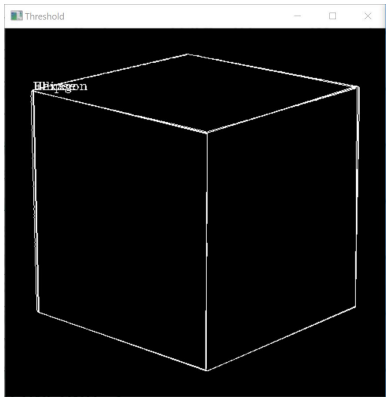


(f) Three dimensional top

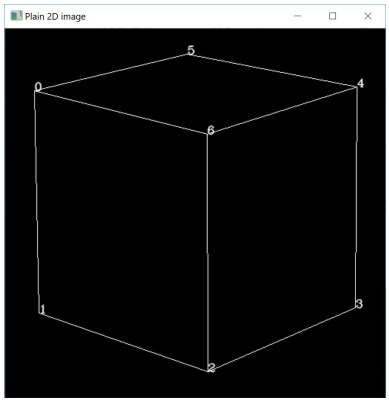
Figure 7.3: Process of creating a three-dimensional cube-shaped model out of a simple cube



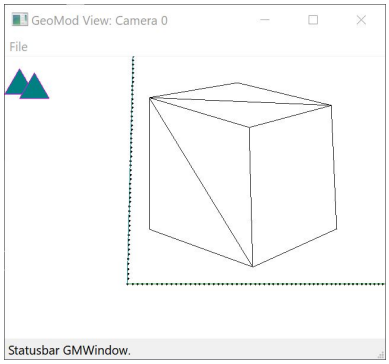
(a) Original Picture



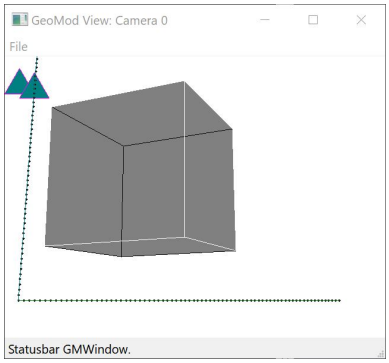
(b) Threshold



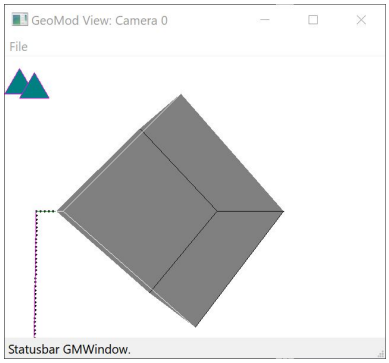
(c) Calculated Image



(d) Two-dimensional



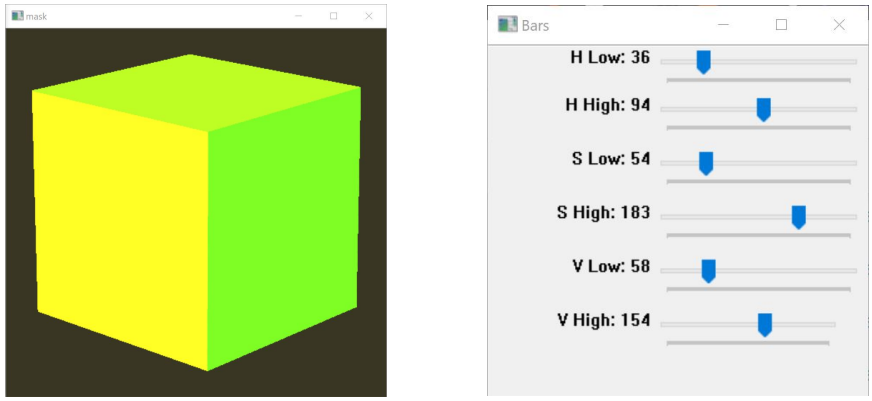
(e) Three-dimensional front



(f) Three-dimensional top

Figure 7.4: A red cube is transformed into a three-dimensional mode

For this image of a red cube, colour modification had to be used to find all the edges of the cube correctly. This also helps mark an object against its background. The values for the sliders used to modify the pictures colours is shown in figure 7.5b.



(a) Cube with colours changed to help identify edges

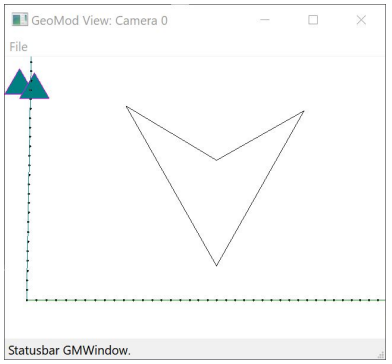
(b) Values used to create the colour modified version

Figure 7.5: Colour modification used to find the edges of the cube

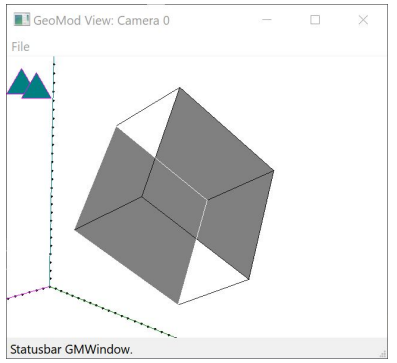
As seen in figure 7.4d two lines were not drawn correctly in the GeoMod view. This is due to the list of relations from the python program having an error and writing the nodes in the wrong sequence to file. This means that the for-loop drawing the edges makes errors. The faulty edges have no effect on the final model and the cube will still be created correctly.

7.3 Cubes created by the four-points method

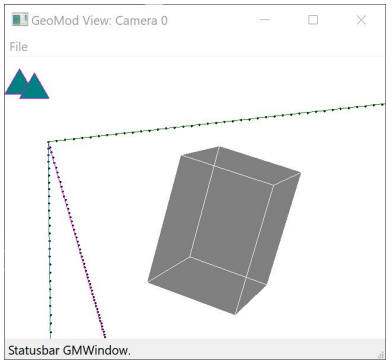
Another few results using the four-points method are shown in this section. Figure 7.6 shows the same cube as in figure 7.3, using the same threshold and calculation basis.



(a) Two dimensional model of the four points



(b) Calculated three-dimensional model using the four-point method



(c) Top view of the model

Figure 7.6: Pictures showing the result of the four-points method

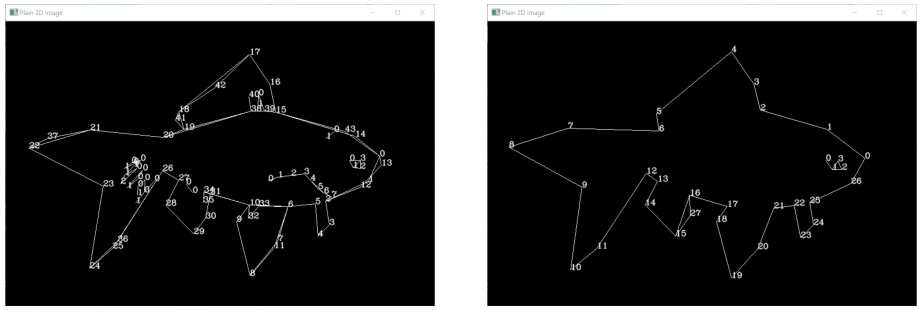
The figure 7.6b shows the front side of the cube. The distortion is due to the camera view, not the model. The white sides are because those sides are being created inside out during the for-loop, which means they go in the opposite direction of the drawing direction defined in the program. The more rectangular shaped sides of the model are due to varying distance between the nodes. A distortion free angle is shown of the top view in figure 7.6c.

7.4 More complex shapes

After the method to create general models in the GeoMod environment was completed it was exciting to see what it was capable of. More complex images where chosen to create two-dimensional shapes with great results. Here the values of the filters are important.

Using the sliders to adjust these values can give very different results. Choosing a low value for epsilon resulted in a lot of points along the outline of the model. Choosing a too low value for epsilon could crash the program. Therefore the lower limit for epsilon was increased to prevent future crashes. These pictures of a fish shows a outline with low epsilon and low *"Area of Objects"* value, giving a great magnitude of points. Increasing the *"Area of Objects"* value will eliminate small shapes often found inside the outer contour. This model is also very costly to produce in the GeoMod environment.

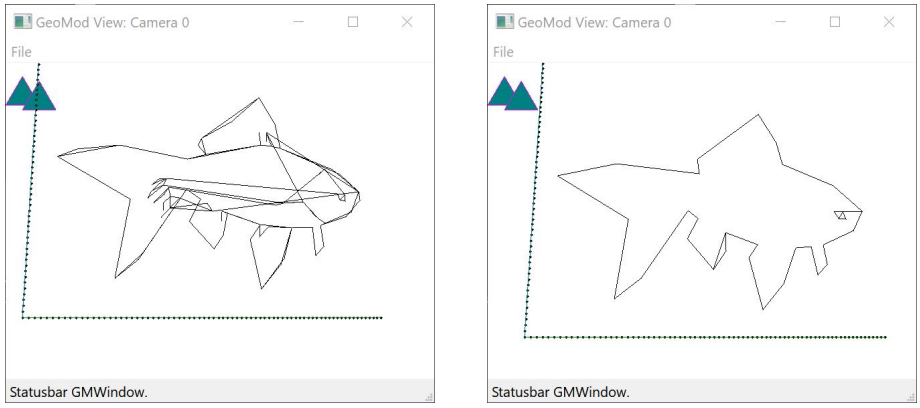
After the *createGenericModelDepths* method was developed, it was applied to these images as well. The results were not true to real life, but they were still quite exciting. The result of a depth image of the fish can be seen in figure 8.1b.



(a) Calculated image of the fish with low epsilon and low "Area of Objects"

(b) Calculated image of the fish with higher epsilon and high erosion (5)

Figure 7.7: The fish image with different slider values



(a) Many points inside the outer contour give multiple errors when creating edges

(b) Fewer points and less points inside the outer contour create a more correct outline of the fish

Figure 7.8: The image on the right is preferred over the one on the left

7.5 Multiple models in one image

One of the benefits of dynamic linking is to be able to link in multiple models with the same class during run-time. This means that multiple images can be used to detect shapes and the results of that detection can be linked into the camera view. Here is a figure depicting how the view looks like when multiple models originating from different images have been linked into the view. Here two cubes and a heart shape have been linked during the same run-time.

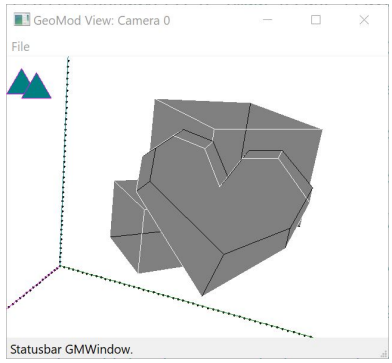


Figure 7.9: Multiple models loaded into the camera view

This is also extremely useful for navigation. The camera view can keep hold of models it passes during navigation, and un-link and link them when needed. Once a model has been linked, it can be tracked and moved relatively to the movement of the vehicle.

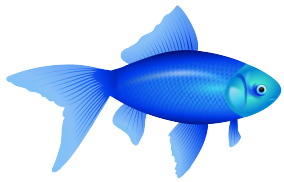
Chapter 8

Application, challenges and future work

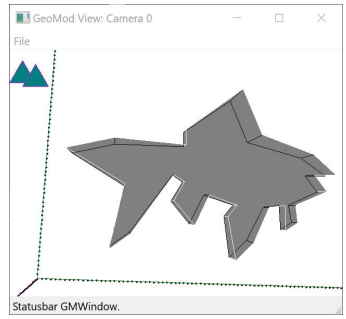
8.1 Potential applications in the real world

As with most projects and thesis', using the thesis' product and result to improve something that exists in the real word is very exciting. What could model estimates of shapes possibly be used for? Since this thesis is written for the Norwegian University of Science and Technology and has strong Norwegian influences, the fish farming industry was an immediate thought. It's not more than a thought, but some concrete application for this kind of software are given below.

An idea would be to use cameras in combination with a 3D model construction to evaluate the weight, size and area of fish in fishing farms. Submerging the camera below the surface of the water and taking pictures or translating a video feed directly into 3D models which can then estimate the weight and size of the fish. This will give direct information of the health of the fish, when it can be processed, and if there are edge case fish that must be monitored. This will require good knowledge of the fish's shape and layout, and an automated method to find important points in the fish's shape. Just out of curiosity a fish was modelled in the GeoMod program and an approximation of its volume was taken. This was done by calculating its area and multiplying it by its depth, as shown in section 6.5. For such a complex model the result is of course not completely accurate, but it is a good estimation. For simpler model such as a cube or a simpler polygon the numbers are quite accurate. Since it was shown that an approximation for cubes in three dimensions could be made in this program, recreating a fish in three-dimensions should be possible.



(a) Original picture of a fish



(b) Estimated and plotted model of a fish

Figure 8.1: Generic three-dimensional models of 2D pictures

The thought of fish modelling was directly related to this thesis environment, the sub-sea. In addition to hindrances and fish, also other objects floating under the water surface could be modelled and logged in real time. Even plastic pollution in oceans can in theory be documented and mapped with a modification of this program.

Another real-life application for a thesis such as this, is in mass productions. Products transported down a product line can be detected by a single camera. This camera can then rapidly decide where to sort this product by changing its lane. This way also faulty products can be detected.

8.2 Challenges

As one can imagine with a pure programming project, everything did not go as smooth as it should. A lot of trial and error have resulted in evolving the code to be what it is now.

The first challenge was already met in the beginning of the project, when OpenCV was supposed to be implemented in Qt. The program Qt has support for the OpenCV environment, and there are guides explaining how to install them alongside each other [38]. Although most of these guides were outdated, the optimism was high to be able to get OpenCV to work with the GeoMod program and the Qt environment. For some reason this turned out to be impossible, both on the main computer used for this thesis and a backup computer, both running Windows. The OS was important to be Windows, since that is the operating system which NTNU seems to have adopted. After many frustrating weeks, it was decided that OpenCV should be installed on an IDE running python. This was simple and took around 10 minutes. So, in the end, two programs had to do the tasks that the thesis was supposed to implement. This resulted in a bit sub-optimal communication system. The benefit of that was that they could run independently. This means that images could be recognised and transported to the other program while GeoMod was running in the background.

Using a python environment for coding allowed the use of the TensorFlow API [3]. Since machine learning, and everything that seems to be using the key word AI, is very popular right now, an approach using something of that order was tried to be implemented. After running a few examples, such as the *"cat and dogs'"* example in figure 8.2, it was

found that although potentially useful, it would be too much work to implement it to work with GeoMod and use it to recognise simple shapes. It would also be too dependable on the TensorFlow API, and since updates to that program were very frequent, the thesis' code would soon be outdated and would need constant updates.

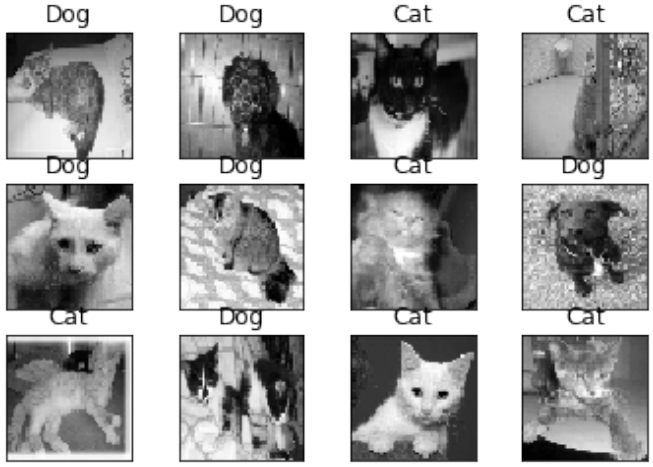


Figure 8.2: An example run in TensorFlow that tries to differentiate between a cat and a dog in a picture. Not every picture was recognised perfectly. The basis were 10000 pictures of cats and dogs to train the model

The next challenge was the unfamiliarity of the languages used for this thesis. Both Python and C++ were unfamiliar programming languages and much of the languages syntax and general rules had to be learned from scratch. Being familiar with a lot of other programming languages helped a lot. Some work had previously been done with the GeoMod program. It is quite a complicated project and a lot of time was used to get to know the program. For example, at least a week could have been saved if the implementation of regions would not have been thought to be different than it turned out to be.

8.3 Future Work

After the completion of this thesis, some future work can be done to improve and extend the functionality of the program, these are listed in no particular order below.

A great basis for a future semester thesis would be to port the OpenCV functionality to the Qt environment. Since the functions available in the OpenCV environment for C++ are very much alike to those implemented in the python code, the porting should not be too challenging. Porting the function for finding the middle node would be top priority, making the GeoMod less dependant on the result of the python part.

Qt offers a huge advantage over the python part in that it supports much better implementation of a Graphical User Interface. This should make the implementation of changing the value for the filters much nicer and simpler to work with. A file browser should

also be implemented for the selection of source images used for the detection of shapes.

The most important difference an implementation in Qt would make is the communication between the shape detection and the shape modelling. Shapes detected could be send directly into the camera view of the GeoMod program and a three-dimensional estimation could also be created directly after identifying a shape.

An implementation that would have been really fun to do for this project, but that was unfortunately outside of this projects scope, is the implementation of recognising shapes from a video feed. A video feed could be linked directly into OpenCV environment by using for example a web-cam. Supporting video streams is really exciting when it comes to movement of objects. Imagine a camera mounted on a drone, the camera feeding the program with pictures and information. The program creates a real time, three-dimensional model of what is around it. This is still very far off, but this project could be a step in the right direction.

Implementing the adaptive threshold function could improve future results and make automation of shape detection easier. It is after all automation that is sought to be reached in this program.

To not implement a form of gravity that affects the linked in models in the program was done on purpose. Using the control panel of the GeoMod program, models can be moved around in the camera view. This was deemed sufficient, and the models were left floating in space when being loaded in dynamically.

Automation can further be implemented by eliminating the need for the sliders that change the filters and the functions input. This can be done by trying out multiple scenarios of images and see what filters work best for them. This knowledge is then saved. If a scenario is recognised, the settings from previous tests can be applied and the shape detection can now be performed on the image. An example is if a dark shape or scene is present on the image this can be recognised by looking at the RGB values of the pixels. If a predetermined setting is available for dark objects, like when the dark cube was used in figure 3.10, it can then be applied and improve the shape detection.

Allowing a custom depth for general models is a feature that could be implemented. This would also make estimation of volume more correct. The possibility to re-size a model is already implemented, but a slider in the GUI to adjust this value would be a recommended implementation.

Chapter 9

Conclusion

This project was about shape detection. It showed how to detect shapes and port them over to a different program. The project started with detecting very simple shapes and forms. By using read and write functions, these simple shapes could be identified in a python environment and then they could be drawn in the view environment of the GeoMod program. More complex shapes were supported by adding more points to a model. The detection of multiple shapes in an image was also added to the program.

The main focus was on making cubes three-dimensional based on a set view of a cube in two dimensions. A four-points method was implemented that could draw cubes based on four points. This method could make the program less error prone than the alternative method that was implemented, but distances between edges could be less like the length in the original two-dimensional image. A way to make normal two-dimensional shapes three dimensional was implemented. The depths for general shapes such as rectangles and pentagons were found by using normal vectors. This gives the models a three-dimensional shape in space. The results presented in chapter 7 show that the code is working well for the test-images chosen. Cubes are created quite accurately, with both methods that are implemented. Other shapes, such as triangles and pentagons can be drawn in two dimensions or a general depth can be given to the contour to make the model three-dimensional. This works very well with such simple shapes. More complex shapes like a heart or a fish can be drawn in two dimensions quite accurately. When creating a three-dimensional estimation, complex shapes can have a various degree of depth for different points, which makes the three-dimensional approximation not very life-like.

Bibliography

[1] Opencv: Contours hierarchy. https://docs.opencv.org/3.4/d9/d8b/tutorial_py_contours_hierarchy.html.

[2] Pep 20 – the zen of python — python.org. https://www.python.org/dev/peps/pep-0020/.

[3] Tensorflow. https://www.tensorflow.org/.

[4] Recursive function definition. https://techterms.com/definition/recursivefunction.

[5] Fibonacci numbers. https://www.ics.uci.edu/~eppstein/161/960109.html.

[6] Point in polygon - wikipedia. https://en.wikipedia.org/wiki/Point_in_polygon#Ray_casting_algorithm.

[7] What is the ray-casting algorithm? - quora. https://www.quora.com/What-is-the-ray-casting-algorithm.

[8] Ray-casting algorithm - rosetta code. https://rosettacode.org/wiki/Ray-casting_algorithm#Python.

[9] Pnpoly - point inclusion in polygon test - wr franklin (wrf). https://wrf.ecse.rpi.edu//Research/Short_Notes/pnpoly.html.

[10] File:point in polygon problem.svg - wikimedia commons. https://commons.wikimedia.org/wiki/File:Point_in_polygon_problem.svg.

[11] performance - how can i determine whether a 2d point is within a polygon? - stack overflow. https://stackoverflow.com/questions/217578/how-can-i-determine-whether-a-2d-point-is-within-a-polygon.

[12] Numpy — numpy. http://www.numpy.org/.

[13] Opencv: Contours : Getting started. https://docs.opencv.org/3.4.2/d4/d73/tutorial_py_contours_begin.html.

[14] Advanced engineering mathematics - dennis g. zill, michael r. cullen - google books. https://books.google.de/books?id=x7uWk8lxVNYC&pg=PA324&redir_esc=y&hl=en#v=onepage&q&f=false.

[15] Dot product. https://www.mathsisfun.com/algebra/vectors-dot-product.html.

[16] Cramer's rule – from wolfram mathworld. http://mathworld.wolfram.com/CramersRule.html.

[17] Cramer's rule - wikipedia. https://en.wikipedia.org/wiki/Cramer%27s_rule.

[18] Understanding the main method of python - stack overflow. https://stackoverflow.com/questions/22492162/understanding-the-main-method-of-python.

[19] c++ - size/length of a trackbar name in opencv - stack overflow. https://stackoverflow.com/questions/39845274/size-length-of-a-trackbar-name-in-opencv.

[20] Polygon area and centroid - polygon area and centroid.pdf. chrome-extension://oemmndcbldboiebfnladdacbdfmadadm/https://www.seas.upenn.edu/~sys502/extra_materials/Polygon%20Area%20and%20Centroid.pdf.

[21] Picture: Inclusion of a point in a polygon. http://geomalgorithms.com/a03-_inclusion.html.

[22] https://upload.wikimedia.org/wikipedia/commons/c/c9/recursiveevenpolygon.svg. https://upload.wikimedia.org/wikipedia/commons/c/c9/RecursiveEvenPolygon.svg.

[23] Normal cube. http://imagine.inrialpes.fr/people/Francois.Faure/htmlCourses/WebGL/cube36.png.

[24] Red cube. https://www.freeiconspng.com/uploads/red-box-png-3d-cube-picture-24.png.

[25] Black cube. https://www.freeiconspng.com/uploads/black-cube-box-png-5.png.

[26] Xhensila Poda and University of Tirana Olti Qirici. Shape detection and classification using opencvand arduino uno. chrome-extension://oemmndcbldboiebfnladdacbdfmadadm/http://ceur-ws.org/Vol-2280/paper-19.pdf.

[27] Libraries & apis, tools and ide — qt. https://www.qt.io/qt-features-libraries-apis-tools-and-ide/.

[28] Project jupyter — home. https://jupyter.org/.

[29] Opencv library. https://opencv.org/.

[30] Marc Pollefeys and Luc Van Gool. From images to 3d models. Communications of the ACM July 2002/Vol. 45, No.

[31] Deva Ramanan Mohsen Hejrati. Analyzing 3d objects in cluttered images.

[32] http://www.cplusplus.com/doc/tutorial/pointers/. C++ pointers.

[33] https://www.tutorialspoint.com/cplusplus/cpp_overloading.html. C++ inheritance.

[34] Is there any method to extract 3d shapes/volumes from single 2d grayscale image? thread. https://www.researchgate.net/post/Is_there_any_method_to_extract_3d_shapes_volumes_from_single_2d_grayscale_image.

[35] c++ - opencv edge/border detection based on color - stack overflow. https://stackoverflow.com/questions/29156091/opencv-edge-border-detection-based-on-color.

[36] Canny edge detection — opencv-python tutorials 1 documentation. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_canny/py_canny.html#canny.

[37] How to create an image classifier using qt, opencv and tensorflow - amin. http://amin-ahmadi.com/2018/03/15/how-to-create-an-image-classifier-using-qt-opencv-and-tensorflow/.

[38] Qt and opencv. http://qtandopencv.blogspot.com/.

[39] Choosing the correct upper and lower hsv boundaries for color detection inrange (opencv) - stack overflow. https://stackoverflow.com/questions/10948589/choosing-the-correct-upper-and-lower-hsv-boundaries-for-color-detec

[40] opencv hsv mask/filter with trackbar · github. https://gist.github.com/yingminc/68bf81a79b3bd87070b364d1764e6c70.

[41] Picture of blue fish. https://kascomarine.com/wp-content/uploads/2017/06/blue-fish-png-image-18.png.

[42] Picture of heart. https://dumielauxepices.net/sites/default/files/drawn-shapes-heart-571379-6515246.png.

[43] Picture of many simple shapes. http://diysolarpanelsv.com/images/preschool-shape-monster-clipart-23.jpg.

[44] Picture of many, more complex shapes. https://www.amvplaygrounds.co.uk/pub/media/catalog/product/a/m/amv_f4-pm-020-shapes-circle-square-rectangle-star-triangle-500mm-2-sq-3jpg.

[45] Picture of simple rectangle. https://upload.wikimedia.org/wikipedia/commons/c/cc/Rectangle_.png.

[46] Welcome to python.org. https://www.python.org/.

[47] History of c++ - c++ information. http://www.cplusplus.com/info/history/.

[48] A control system for autonomous vehicles, three-dimensional geometric models from pictures. Tony Gjendahl, June 2017.

[49] Image processing - how is gaussian blur implemented? https://computergraphics.stackexchange.com/questions/39/how-is-gaussian-blur-implemented.

[50] Smoothing images — opencv 2.4.13.7 documentation. https://docs.opencv.org/2.4.13.7/doc/tutorials/imgproc/gausian_median_blur_bilateral_filter/gausian_median_blur_bilateral_filter.html.

[51] Eroding and dilating — opencv 2.4.13.7 documentation. https://docs.opencv.org/2.4/doc/tutorials/imgproc/erosion_dilatation/erosion_dilatation.html.

Appendix

1.1 Risk assessment

Generally a risk assessment is required when handing in a master thesis, but for theoretical thesis' a risk assessment is not required. Since this thesis only involves software development it was not deemed necessary to include a risk assessment.

1.2 Saving last read path to file

A save to file system was implemented to remember the path of the previously selected file. This made navigating when using a file-browser easier. Previously, upon opening a file-browser, the user would have to navigate through multiple levels to find a file to for example link into the GeoMod program dynamically. With the save-to-file system, it opens the file-browser on the level that the last file was loaded from. This saves a lot of time, since this is done very frequently.

1.3 Installation Guide for Qt Creator (5.9.1) and Visual Studio 2017

Note: In this installation guide Qt Version 5.9.1 and Windows Visual Studio 2017 were used.

This Guide is divided into three parts. The first part goes through installing the correct version of Qt Creator to be able to run this project. The second part helps you install Windows Visual studio. The third part is the installation of a QT plug-in for Visual Studio, so that a Qt project can be run in the Windows Visual Studio environment.

Running the project can be done in both Qt Creator and Visual Studio, so the preferred editor can be chosen freely.

1.3.1 Part 1: Installing Qt Creator

First we will need to download **Qt Creator**. This can be done by either googling "Qt creator Community Edition" or following this link:

- `https://www.qt.io/download-open-source/?hsCtaTracking=f977210e-de67-475f-a32b-65cec207fd03%7Cd62710cd-e1db-46aa-8d4d-2f1c1ffdacea`

This version of the Qt Creator is the Community version, which means it is free to use.

The installation of the Qt Creator edition is quite simple. When asked to log in or to create an account, you can simply use the skip button to skip this step.

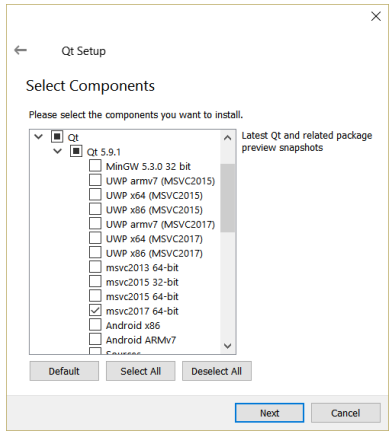


Figure 1.1: Qt version

When selecting what components of Qt you want to install, choose the newest version which supports Visual Studio 2017. This can be seen by going on the different versions of Qt, such as *5.9.1, 5.9.0* and *5.8*. This is important if you later on want to install Visual Studio and import the Qt libraries.

Follow the hierarchy downwards and you will need to find a box with the name *msvc2017 64 bit*. This box has to be checked, and then you can press *Next* on the installer.

When you come to the License Agreement window, simply choose the Qt License Agreement, accept the license agreement, and you are done.

Running the project now without a windows SDK installed will result in the following error:

Figure 1.2: Run error

1.3.2 Part 2: Installing Windows Visual Studio and a QT plug-in

Next up we will need to install **Windows Visual Studio Community** Edition from the Visual Studio web page. This is done by either navigating to the official web page of the software or following this link:

- `https://www.visualstudio.com/downloads/`

The file you downloaded will be your ”Command Center” to edit, install and modify your Visual Studio Software. This program is a standalone installer which will let you

install and modify missing components later on. On the first time install, you will have to select what components to install. If you navigate to the *Individual Components* tab at the top, you will have to select the the *Windows 8.1 SDK* under the SDK menu to get it to work with Qt.

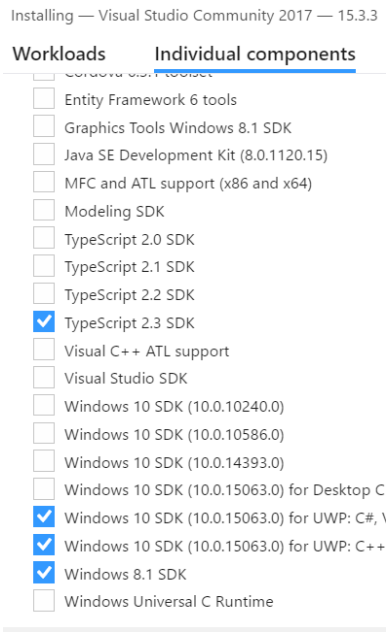


Figure 1.3: SDK selection

When it comes to the license part you will need to enter your student email, this will redirect you to the log in page of your University. After log in is completed, your license should have been registered and you can continue with the installation.

1.3.3 Part 3: Installing the Qt plug-in in Windows Visual Studio

You will need a plugin to make Qt work together with the Visual Studio platform. This plug-in is called QtvsTools and can either be downloaded by googling said name, or by navigating to this link:

- `http://download.qt.io/development_releases/vsaddin/`

After downloading it, execute the program, and wait for it to be done.

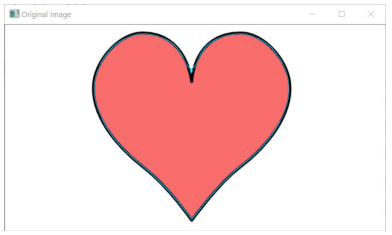
Last up you will need to open up your Visual Studio 2017 version. After installing the plug-in, there should now be a Qt VS Tools drop down menu at the top of the window. Open the menu, go to Qt Options. Here we will need to add a new path to the Qt library. Press *Add* and navigate to your Qt installation folder. In this folder, select the version you've installed (5.9.1). Select this version folder as you path. The version name field

should be named the same as the version of your Qt program (5.9.1). Once the path has been added, your program should be fully functional with Qt Creator.

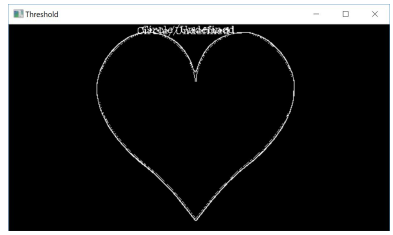
(Note: Some work PC's from NTNU might not be able to use the Qt framework through Visual Studio, as those programs communicate through their registers. In the windows register one can find all the necessary information, settings, options, and other values for programs and hardware installed on all versions of Microsoft Windows operating systems. These settings usually not accessible on work computers.)

After following this guide, you should be able to use either Qt or Visual Studio for your work. You additionally will need to install OpenCV and OpenGL for this program to work.

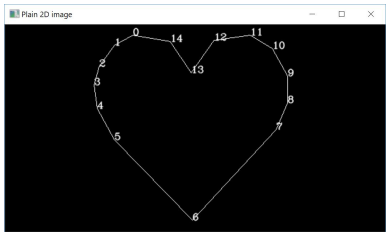
1.4 More results



(a) Original Picture



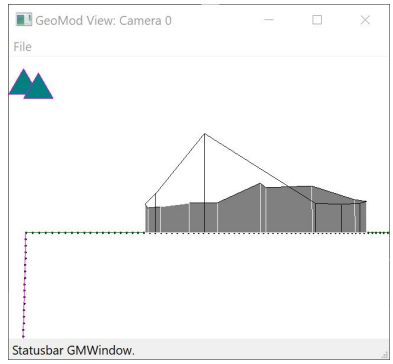
(b) Threshold



(c) Calculated Image

(d) Two Dimensional

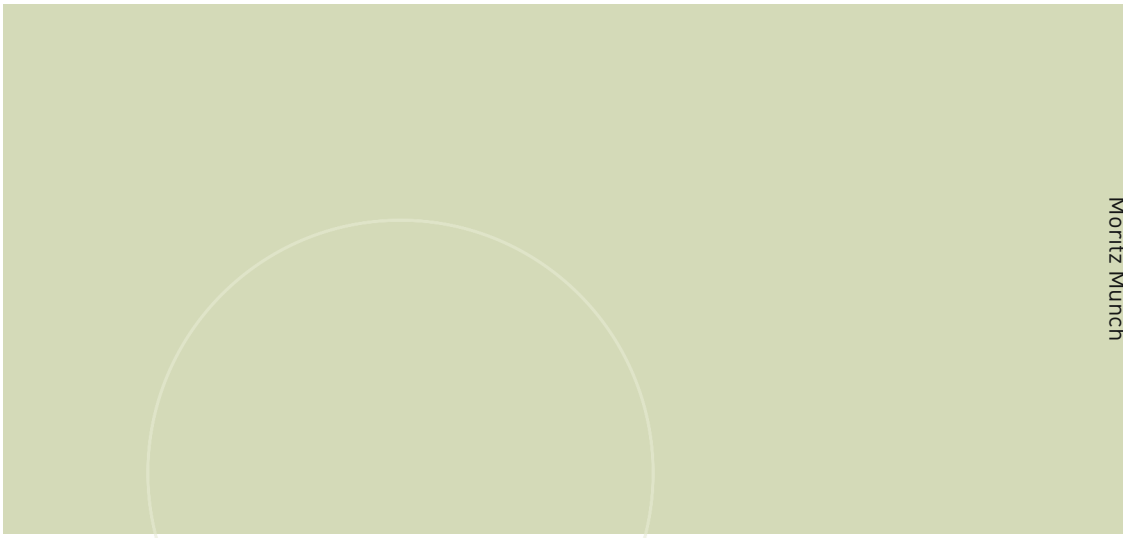
(e) Three dimensional front



(f) Three dimensional top

Figure 1.4: A more complex shape of a heart

The figure 1.4 depicts a heart which goes from being a normal picture of a heart to being a model loaded into the GeoMod program. The estimation of depth for the heart is based upon the lowest value of the distance between two nodes that are next to each other. This means that the depth for the lowest point of the heart is really deep. Its deepness is exactly the length of one of the long lines on that go down to the bottom. This can very easily been seen in image 1.4f.



Moritz Munch

Three-dimensional Geometric Models from Pictures

NTNU
Norwegian University of
Science and Technology