

THREE DIMENSIONAL WORLD  
RECONSTRUCTION WITH LEGO BLOCKS

A thesis  
submitted by  
Jose M. Troncoso

In partial fulfillment of the requirements  
for the degree of

Master of Sciences  
in  
Mechanical Engineering

TUFTS UNIVERSITY

November 2002

© Copyright by Jose M. Troncoso, 2002  
ADVISER: CHRIS ROGERS, PHD.

# Abstract

The use of computer vision as an input for machines is becoming a crucial part of industry as machines become more automated. From the assembly process, to testing defective components, image processing allows faster and cheaper manufacturing. Basic computer skills have become a minimum requirement for engineering or technical jobs and a greater knowledge of fields such as image processing is needed in order to become a desired candidate in those markets. Organizations such as the National Science Foundation are making progress in introducing these needed skills at an early age so that students have a greater knowledge of subjects such as image processing.

Robolab has helped to introduce computer programming at the kindergarten and elementary school level. With the use of the vision tools available with this software, we can also teach children of many ages image processing. The purpose of this thesis is to develop a web site that children can visit and, together with their LEGO camera and blocks, be able to understand concepts such as acquiring an image, applying a threshold, and distinguishing between color planes in an RGB image.

The web site will at the same time revolve around a central project which the user can ultimately build after completing a set of challenges. This project consists on rendering a 3 dimensional world with the use of a LEGO camera and a set of LEGO blocks. There are many methods that are currently used to produce 3D images. The goal of this research is to design a reconstruction method that is simple enough for middle school aged children to use and at the same time is able to produce quality models for higher end users. The method that I have developed involves a one-camera view of Lego blocks that are arranged in a certain pattern to represent a building and

its contents. When the pattern is placed in the view of the camera, the software identifies the pieces by their position and color and builds a 3D model in OpenGL on the screen with the specified characteristics. The user can then modify these objects to further refine the model. This software is simplified into icons which accompany certain challenges in the website and which higher end users can explore to get a better understanding of how the code actually works.

# Table of Contents

<b>Abstract</b>	<b>ii</b>
<b>Table of Contents</b>	<b>iv</b>
<b>List of Tables</b>	<b>v</b>
<b>List of Figures</b>	<b>vi</b>
<b>Acknowledgements</b>	<b>x</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Goals . . . . .	4
1.3 Literature Search . . . . .	6
1.4 Approach . . . . .	8
<b>2 Tools Used</b>	<b>14</b>
2.1 Introduction . . . . .	14
2.2 Robolab . . . . .	14
2.3 LEGO RCX . . . . .	15
2.4 LEGO CLI . . . . .	16
2.5 LEGO Sensors . . . . .	16
2.6 LEGO Camera . . . . .	17
2.7 OpenGL VIs . . . . .	19
<b>3 Dot Photogrammetry</b>	<b>21</b>
3.1 The Concept . . . . .	21
3.2 Calibration . . . . .	24
3.3 Set-up . . . . .	25
3.4 Software . . . . .	28

3.5	Results . . . . .	31
<b>4</b>	<b>Stereo Vision</b>	<b>33</b>
4.1	The Concept . . . . .	33
4.2	Set-up . . . . .	34
4.3	Software . . . . .	36
4.4	Results . . . . .	42
<b>5</b>	<b>Plane View</b>	<b>46</b>
5.1	The Concept . . . . .	46
5.2	Set-up . . . . .	48
5.3	Software . . . . .	51
5.4	Results . . . . .	59
<b>6</b>	<b>Design of a Software Tool Set</b>	<b>64</b>
6.1	The Concept . . . . .	64
6.2	Design Approach . . . . .	65
6.3	Tutorial . . . . .	66
6.4	Grabbing an Image . . . . .	68
6.5	Binary Morphology . . . . .	71
6.6	Image Restoration . . . . .	74
6.7	Color Analysis . . . . .	80
<b>7</b>	<b>Conclusion and Future Work</b>	<b>83</b>
7.1	Overview of Work Done . . . . .	83
7.2	Future Enhancements . . . . .	86

# List of Tables

5.1	Average standard deviation for LEGO colored bricks . . . . .	55
-----	--	----

# List of Figures

1.1	Screen shot of a world in Active Worlds . . . . .	2
1.2	LEGO virtual world and user interface at SIGGRAPH 96 [?] . . . . .	8
1.3	The Sojourner rover on the Martian surface [?] . . . . .	9
1.4	Outline of LEGO blocks . . . . .	13
1.5	3D OpenGL representation . . . . .	13
2.1	LEGO programmable brick RCX . . . . .	15
2.2	The LEGO Control Lab Interface . . . . .	16
2.3	LEGO rotation and variable resistance touch sensors . . . . .	17
2.4	LEGO camera . . . . .	18
2.5	Uneven contrast in the image affects processing operations . . . . .	18
3.1	Separate camera views . . . . .	22
3.2	The images from the three cameras combined into one . . . . .	22
3.3	The six displacement parameters of the camera . . . . .	23
3.4	LEGO calibration plate . . . . .	25
3.5	Calibration plate as seen by the center and side cameras . . . . .	26
3.6	The centroids shift slightly to the left because of oval appearance . . . . .	26
3.7	Three camera set-up . . . . .	27
3.8	Dots used for 3D location . . . . .	28
3.9	Points of light as seen by the three cameras . . . . .	29
3.10	Epipolar geometry of an image pair . . . . .	30
3.11	Matlab graph of points on the cylinder . . . . .	31

4.1	Image offset between two cameras attached to each other . . . . .	34
4.2	Two camera set-up . . . . .	35
4.3	Rectangular object used for calibration . . . . .	35
4.4	Wrong correlation from insufficient search image . . . . .	37
4.5	Image with a sobel filter applied . . . . .	38
4.6	The original images and the processed ones with the objects matched	39
4.7	Additional details identified in larger object . . . . .	40
4.8	Picture of an object on an angle . . . . .	41
4.9	LEGO base with ten LEGO people . . . . .	42
4.10	LEGO rover navigating around the LEGO people . . . . .	43
4.11	LEGO people in 3D . . . . .	44
5.1	The one camera set-up with the outline of a building . . . . .	48
5.2	Sample outline for the 3D extrusion method . . . . .	49
5.3	Interior doors in 3D . . . . .	50
5.4	Panel used to arrange objects . . . . .	50
5.5	A building with multiple stories . . . . .	51
5.6	Flow chart of the block recognition process . . . . .	52
5.7	Outline of the building with all the black pieces removed . . . . .	52
5.8	Wall recognition flow chart . . . . .	53
5.9	Complete ID of walls . . . . .	54
5.10	Average standard deviation of a piece of gray LEGO plate . . . . .	54
5.11	The removal of the gray plate from the image . . . . .	56
5.12	Color blocks identified in the image . . . . .	56
5.13	All the blocks have been identified . . . . .	57
5.14	First extrusion example . . . . .	60
5.15	Second extrusion example . . . . .	61
5.16	Third extrusion example . . . . .	62
6.1	Windows Explorer look of web-site tutorial . . . . .	66
6.2	Sample tutorial page . . . . .	67

6.3	Image grabbing folder of tutorial site . . . . .	69
6.4	Using structures in Robolab programming . . . . .	70
6.5	Sample concepts page . . . . .	71
6.6	Sample challenge and front panel answer . . . . .	72
6.7	Diagram for Figure 6.6 challenge . . . . .	72
6.8	Binary Morphology folder of tutorial site . . . . .	73
6.9	Sample concepts page . . . . .	74
6.10	Front panel of sample answer to Global threshold challenge . . . . .	75
6.11	Sample diagram for global threshold challenge . . . . .	75
6.12	Image Restoration folder of tutorial site . . . . .	77
6.13	Sample concepts page . . . . .	78
6.14	Sample answer to challenge 6.1 . . . . .	79
6.15	Color Analysis folder of tutorial site . . . . .	80
6.16	Sample concepts page . . . . .	81
6.17	Sample front panel for color threshold challenge . . . . .	81
6.18	Sample diagram for color threshold challenge . . . . .	82
7.1	Furniture examples . . . . .	87
7.2	Round block identified in outline . . . . .	88

# Acknowledgements

This thesis work would not have been possible without the help and contributions of my advisor Professor Chris Rogers. Since he cannot stay in the same place for more than a couple of weeks, I had to follow him all the way to New Zealand to get his valuable comments and advice. Many thanks go to Professor Alan Mckinnon and Professor Keith Unsworth for their help and hospitality in the southern hemisphere, cheers, as well as Lincoln University for their founding.

I would also like to thank my committee members, Professor Stephen Morrison and Doctor Daniel Groszmann, for their comments and for taking time out of their busy schedule to attend my dissertation. I want to thank that entire TUFTL crowd, especially those who stayed here, Ed, for their help in answering many of my questions as well as for the Ultimate games which prove to be a great stress buster.

Tambien quiero darle las gracias a mis padres por su apoyo y sus consejos, porque la vida da muchas vueltas y nunca sabe uno donde va ir a parar.

And last, but definitely not least, I want to thank my girlfriend Amy for her love and support and for those many delicious meals that I just did not have time to cook.

This material is based upon work supported by the National Science Foundation under Grant No. 9950741 Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

# Chapter 1

## Introduction

### 1.1 Motivation

As computers become more a part of our work and even our daily lives, we look for ways that allow people to interact with them more easily. We have come a long way since the invention of the keyboard. From the development of the mouse to voice recognition, inputting data into a computer has now become a simpler task. But as software become more complex, we require new input devices to provide the computer with more data, without making the user interface more intricate. Three dimensional images in computers have proved to be very useful, both for their realistic appearance, as well as for the amount of information that they provide to the user. Video games have taken a major lead in this aspect of software imaging by creating virtual worlds that are hard to distinguish from real ones. Figure 1.1 shows a screen shot of one of the worlds in Active World, a virtual world system available over the internet. With the payment of an annual fee, the user can easily create worlds with the use of a limited number of available objects [?].

Even though Pacman would always be a classic in the video game realm, people now find 2D displays unrealistic and boring. By adding a third dimension to a piece of

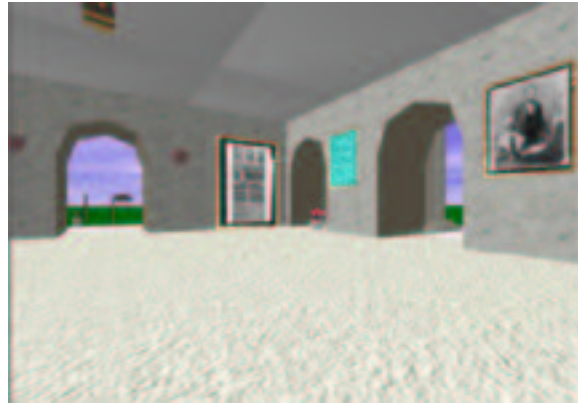


Figure 1.1: Screen shot of a world in Active Worlds

software though, the designer is also adding more complexity to its use, which users do not like. Current input methods (keyboard and mouse) are still constrained to 2D and often require substantial training to use. For this reason, we decided to use LEGO blocks to control some of the 3D building aspects of the software. LEGO blocks are useful not only because a normal person with no training can use them, but also because they are designed with children in mind. We wanted to introduce 3D modeling and its concepts to a high school or college age group: mid teens to early twenties. But the more we simplify the interface, the younger our user group can be. With the use of LEGO blocks, we can therefore allow children to control imaging programs.

It is important to let children interact with imaging software, because it allows them to get an understanding of image processing at a very early stage in their life. Images are intuitive to children, because early in their life, they learn both through what they see and what they hear. They are familiar with cameras and with the pictures they produce. It is therefore more effective to teach them this subject using the camera as a sensor which they can easily operate and experiment with. Our 3D software was designed so that children, hopefully as young as kindergarten, can build

3D worlds. Concepts on how they built the world and how the camera sensor works are then expanded upon to include more complex topics for people that have a better understanding of the subject.

By showing children how to create 3D worlds, we can teach them image processing, a skill that is becoming more heavily used by industry. With the goal of cutting costs and increasing production, many companies are now relying heavily on automation. Computer vision has become a crucial input for these machines which use image processing software for tasks such as assembly or testing of defective components. If we can introduce children to image processing, they will have a greater understanding of the subject and a better opportunity for acquiring a job in the engineering and technical fields.

Standard 3D image processing software, in its current state, is well above the analysis and computer programming capabilities of even the average college graduate, yet the concepts behind image processing are familiar to even a small child. For instance, a two year old can look at an image and identify her parents in the image. To have a computer perform a similar task is very difficult, especially with shadows or if the parents are looking to one side. If we simplify that software enough, we can allow users to perform simple imaging tasks without a lot of programming knowledge. We did this by breaking down the tasks and then building up a software package, or tool set, which is simple to use. The tool set and programming are based on the Robolab programming language.

Robolab is a software package developed here at Tufts University for the LEGO Corporation to introduce computer programming to students ages eight and up [?], but especially children. Robolab is a visual programming language (based on LabVIEW) that allows children and adults to take data measurements and to even make

their own robots. One of the newest additions to this software package is its image processing capability. With the use of a LEGO camera, the user can perform simple tasks such as grabbing an image or finding motion in a pair of images, or more complex ones that include morphology operations and image filters.

To allow anybody in the world to access this tool set, we created a web-based tutorial that explains the concepts behind the image processing tools. The site teaches through the presentation of challenges that build up into a final project: the 3D recognition software built for this thesis. This web site will therefore begin with basic image processing skills so that it can slowly introduce the user to the many aspects of it. As the user gets more comfortable with the basic imaging icons, he can probe them further to understand how each one really works, therefore expanding the level of learning from the very inexperienced to the more knowledgeable. We divided the teaching process into challenges that motivate the student into working towards the final goal or project. At the same time though, with the completion of each challenge, the student is learning a different image processing topic of the many covered in this tutorial.

## 1.2 Goals

The goal of this thesis work is to teach high school and college level students image processing, by developing a web site that introduces and explain this field, and provides them with enough knowledge of the subject to allow them to build a three dimensional rendering software. The web site is based on a major project that would not only attract the child's attention to the subject, but also teach them the concepts that make the algorithm work.

The first idea that we looked at for this project was driving an autonomous LEGO rover through an unknown landscape, by using LEGO cameras as the only input. This proved to be too complex to be taught to someone with no image processing background and therefore had to be dismissed as an option. We then decided to take a different approach to this project. Instead of requiring the computer to identify an unknown world, we would allow the user to build her own on the computer. We wanted to develop a simpler method for entering the data required to build a 3D virtual world into a computer. Currently 3D worlds are built by specifying every aspect of each object in the 3D coordinate plane, such as its location, dimensions, and rotations. For the more experience user, this can seem as a trivial task, but it is well beyond the grasp of the average elementary school student. By combining the real world of LEGO constructions with the virtual world, we developed software that allows the user to quickly build a rough model of what the virtual world looks like. This is helpful for professionals such as architects, to quickly produce models of buildings and landscapes for their clients to look at. The simplicity of this design allows inexperienced users as well to interact with this software.

People have a better grasp of concepts when they can interact with them visually. The image processing web site will allow students to explore imaging concepts with the use of a simple library of Robolab icons that they can download from the Internet. With the use of a LEGO camera and some blocks and pieces, the student can go through a series of challenges that will explain basic image processing principles. Most of this principles form part of the final project, the 3D builder. This process would therefore allow someone with no previous image processing experience, to understand basic image processing and to design a simple image processing algorithm.

### 1.3 Literature Search

There are many sites on the internet that explain image processing concepts to help people get a better understanding of this subject. Some of these sites, such as the HP image processing course [?], are made for adults with a college education, since they require a good knowledge of mathematics. These sites explain the image processing terms with very high detail including the mathematical equations that make them work. This approach might overwhelm an inexperienced user, who would probably give up if the concepts take too much time to understand. We are trying to keep students interested through the LEGO building and challenges, moving from rote learning environment to a hands-on one.

Others, such as the HIPR2 site [?], allow the user to work interactively with the tutorial, providing experiments that the user can run and manipulate to understand the effect of different variables. Our site has a common project that the user builds up to, so that the more the user learns, the more complex the challenges become. Some of these sites that are designed for teachers to use in classrooms require training on the teachers' part. According to Meridian [?], a middle school computer technology journal, on reviewing IPT (Image Processing for Teaching) concluded that "there is a need for additional teacher training in image processing in order for it to be used effectively in the classroom". If the teacher needs previous training, a student would not be able to do this tutorial by himself.

We decided to allow the user to create a physical world before creating the virtual one in the computer, so that inexperienced users could understand spatial relationship of objects in a computer three dimensional image. Building a physical model before creating the virtual one makes the design process less abstract. It also allows for

easier identification of faults or missing objects from the model [?].

There are many methods used to recreate a three dimensional world on a computer screen from one or more images. We wanted to create a 3D rendering software with a simple user interface, so that users with little experience could create 3D models. Gu, J. and Han, J. [?] developed an algorithm to recreate the shape of an object with the use of two cameras and with no calibration target. They do this by first constraining the system and obtaining the depth information explicitly. They then relax the constraints to handle real, more general, situations. This process requires the initial intrinsic and extrinsic camera parameters to be known, which the user has to therefore measure.

One of the first attempts by the LEGO Corporation to combine virtual reality worlds with its blocks was displayed at SIGGRAPH 96 in New Orleans. The set-up consisted of a pair of virtual reality goggles, that the user would wear in order to navigate through a pre-built 3D LEGO world, combined with dual data gloves to manipulate the environment (see Figure 1.2). This world was a virtual representation of the city of New Orleans and included most of the monuments and landmarks found in this town. The software allowed the user to modifying the buildings with the use of virtual blocks and to interact with machines and objects in the world. The LEGO buildings were constructed with the use of textures, to avoid the need for the thousands of blocks required to build them. This virtual city rendering pushed the limits of both hardware and software running on multiprocessor Silicon Graphics workstations as well as requiring 2 GB of RAM [?, ?].

Currently there are no 3D-extrusion methods that involve the use of LEGO blocks that we know of. There is mention of such an idea in the SIGGRAPH conference of 1998. It was called Wizard and it was supposed to "allow children to create Lego



Figure 1.2: LEGO virtual world and user interface at SIGGRAPH 96 [?]

movies in a virtual world by using physical blocks, cameras, and a PC” [?]. This idea unfortunately, as far as we know, never came through.

## 1.4 Approach

The original goal of this thesis was to drive an autonomous LEGO rover through an unknown landscape. With the use of LEGO cameras, the computer would get a view of the objects around it and it would then identify them in 3D space. With this information, it would then drive a LEGO rover around these objects through the shortest path. This is a similar idea to the NASA’s Mars Pathfinder project which had to navigate the unknown Martian landscape with the use of two cameras (see Figure 1.3).

This project would then become the main focus of an image processing web site, which would allow people with no image processing background to understand what really

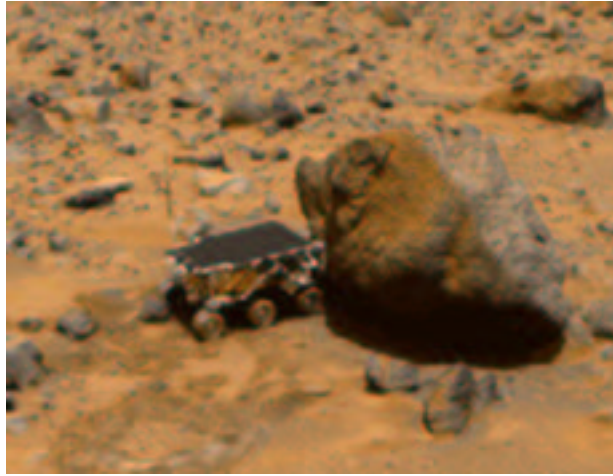


Figure 1.3: The Sojourner rover on the Martian surface [?]

goes on behind such a complex algorithm and to teach them the concepts that drive the program. In order to drive this rover with precision, we needed a camera set-up and algorithm that provides an accurate representation of the unknown landscape. Since the only input guiding the rover is the information from the cameras, we needed a very small error in the rendering of the 3D world, so that the rover did not hit or get caught in any of the objects.

We therefore chose the photogrammetry method, based on the mathematical models from previous research by Mass et al. [?], Nishino et al. [?], and Slama [?] as a means of getting the image information into usable data. We chose photogrammetry because previous work on this topic had already been done in our laboratory to track turbulent particles in 3D space. Photogrammetry involves matching similar points from a set of images from different cameras, in this case three of them. Each camera is focused on roughly the same objects and they each have their own  $x$ ,  $y$ , and  $z$ , locations in space, as well as  $\phi$ ,  $\zeta$ , and  $\beta$  rotations about their centers, giving those objects a different location in each image plane. The algorithm designed for the particle tracking was used to match up points on those objects in the three cameras,

creating point triplets, which provides us with their location in three dimensional space. These points were created by projecting a black image with white dots onto the surface or surfaces that the camera is looking at. This method provides the position of the points in 3D space with very high accuracy, producing an error in the range of one millimeter.

Unfortunately photogrammetry proved to be too complex, not necessarily in the programming context, but on the user interface side. Although photogrammetry is the most detailed of all three processes, it is also more time consuming and less child friendly. Since the cameras have many degrees of freedom, the user has to undergo a calibration process before gathering any data. This process involves moving a calibration plate at predefined intervals and taking a picture of the plate at each interval. The plate contains 24 points of light, which are correlated in all three images. This calibration process would be too intricate for a child to learn and it would also be more prone to errors from an inexperienced user.

The second drawback of the photogrammetry method is the need for a projection of points. In the case of an autonomous rover, projecting points at random objects is not a feasible idea. It would be complicated to carry a projector around with the LEGO set up. This same concern arises when children use this software. They would be able to obtain the LEGO camera and blocks easily, but the projector aspect of the program would complicate the process too much for them as well as making it more expensive for everyone.

Instead, we decided to simplify the set-up by working with only two cameras. The two cameras work like our eyes do providing depth perception. If the X-axes are collinear and their Y-axis and Z-axis are parallel, there exists a disparity in the x direction [?]. As long as they are set up the same way as we had them, side by side,

they don't require any calibration by the user. By comparing the x offset of similar objects in the two images and comparing that offset to a experimentally calculated table, the software can determine the z distance (or depth) of that object. The computer can then use this information to navigate the rover around the obstacles. When an object is blocked by another one in one of the cameras, it only appears in one of the images. The computer cannot therefore correlate that object and it cannot determine its position in 3D space.

The major challenge in the stereo vision method is finding pairs of similar objects. Even though it is obvious for us to distinguish between two LEGO people, the computer does not process information the same way and matches figures that are not the same. We fixed this by allowing the user to select each object in one image and then allowing the software to find that same object in the second image. Unfortunately, when we introduce human input into the software, we also lose the autonomous capabilities of the rover, since it now depends on more than just the images captured by the cameras.

Once again, the autonomous LEGO rover idea failed to be a reality. Since we still wanted to work with the 3 dimensional aspect of image processing, we decided that, instead of letting the software identify an unknown world, we would allow the user to build her own. From then on we started working with the current thesis concept, building a 3D world on the computer with LEGO blocks.

Since we need three spatial co-ordinates, x, y, and z, in order to determine the position in 3D space of the LEGO blocks, we used two cameras: one that images the top of the piece (for the x and z distances) and one on the side of the block to calculate its height or y dimension. The camera that is on the side of the block encounters some calculation problems with the y direction, because the piece seems

to be off the ground as it moves further away from the camera. The software might therefore compute the position of the block in the  $y$  direction as being higher than it actually is.

This method proved to be complicated to learn and it also made the building process time consuming. The user would have to put one piece down at a time and take a picture of it. When the next piece was put down though, there was no grid or index that would tell the user where the previous piece had been, so the matching of several pieces together was sort of a challenge in itself.

Breaking away from the two-camera idea, we decided to simplify the software interaction even more by using just one camera. Unfortunately with one camera we lose the third dimension, in this case the height or  $y$  direction. But it ultimately became the easiest to use for inexperienced users. The one camera set up consists of an outline of the building that the user creates with LEGO blocks. This outline is made up of colored blocks that represent different parts of the house: green are doors, red are windows, etc After the user builds his outline, he grabs a single top view image. At this time the software identifies the position of the pieces and their color, and uses this information, together with a user defined height, to build a 3D representation of the house on the screen (see Figures 1.4 and 1.5).

The user can further modify the building once it is on the screen, by clicking on the desired object and moving it within the navigation window. This set up has proven to be the simplest to use, while still retaining all of the major image processing concepts covered in the web site. In order to introduce these image processing concepts from such a complex algorithm, the code was divided into sub icons that contain critical processing parts of the software and arranged into a software tool set. This tool set was then placed on a web site with a number of challenges for the user to try. Through

this site, the user can learn how each icon works and ultimately combine them all to create a 3 dimensional world.

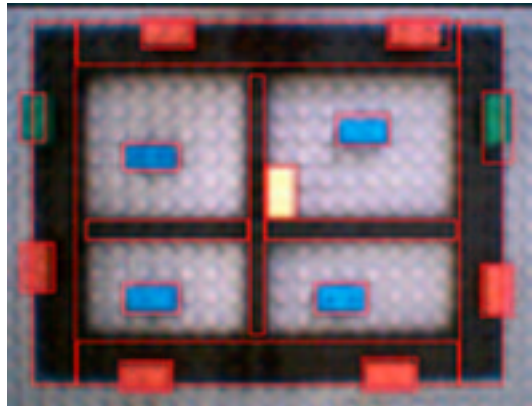


Figure 1.4: Outline of LEGO blocks

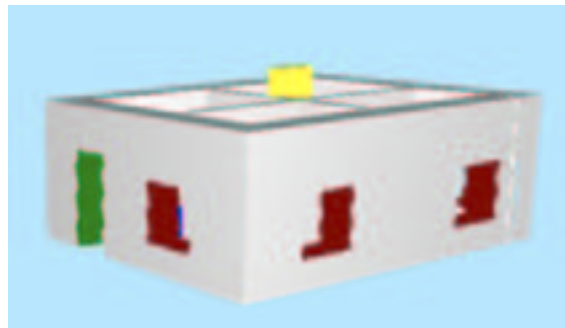


Figure 1.5: 3D OpenGL representation

# Chapter 2

## Tools Used

### 2.1 Introduction

The following chapter explains the capabilities and limitations of the hardware and software that we used for this research. These tools were necessary in order to build the software capable of doing the 3D reconstruction methods described in this Thesis. With the combination of all these methods, we constructed a web based image processing tutorial that will be fully presented in Chapter 5.

### 2.2 Robolab

Robolab is a visual programming language based on LabVIEW, a software product from National Instruments. Robolab was developed at Tufts University for the LEGO Corporation to introduce computer programming to children ages 8 and older [?]. Robolab is an icon based programming language, where the user wires a series of icons to write a code. The latest addition to Robolab software is its image processing capabilities or its multimedia palette. This palette together with the G Code one contain all the necessary icons required for the identification process of our three

dimensional extrusion software.

## 2.3 LEGO RCX

The RCX is a programmable LEGO brick that can be programmed to control motors and gather information from sensors. It is equipped with a 100Hz processor, three nine volt outputs and three 10 bit A/D inputs (see Figure 2.1).



Figure 2.1: LEGO programmable brick RCX

The outputs usually control motors and light bulbs, while the inputs gather data from a large range of sensors including light, temperature, and rotation. The number of ports available was not enough for our three-camera set-up, which used five motors and ten sensors. The user can store and run up to five programs in the RCX. Data is downloaded from the computer to the RCX by an infrared tower, which connects to the computer either via a serial or a USB cable.

## 2.4 LEGO CLI

The control lab interface (CLI) is a product developed by LEGO-dacta that allows the user to control multiple inputs and outputs with the use of Robolab (see Figure 2.2).



Figure 2.2: The LEGO Control Lab Interface

The CLI is composed of eight inputs and eight outputs, plus an additional output that always runs at full power. The CLI does not have any program storage capabilities and needs a continuous connection to the computer to run. It connects to the computer through a serial cable. Just like the RCX, The ports of the CLI also have a maximum output of nine volts.

## 2.5 LEGO Sensors

For our initial set-up, we used touch and rotation sensors to zero out and measure the angle of the cameras before calibrating the set-up (see Figure 2.3).

We used a total of ten sensors in the first set-up: five angle sensors (to determine the rotation about the x and y axis of the side cameras as well as the rotation about



Figure 2.3: LEGO rotation and variable resistance touch sensors

the x axis of the center camera) and five touch sensors to zero out each of those angle sensors. Since we only had eight inputs available with the CLI, we use variable resistance touch sensors. There are three kinds of these sensors, each with a different resistance. We can connect up to three different variable resistance touch sensors to a single input port. By looking at the voltage that goes through when the sensor is pushed, we can determine which of the three has been pushed. The place where the touch sensor activates is not always consistent, varying from slightly pushed in, to pushed in all the way. This was an issue when calibrating the photogrammetry set-up, since the number of rotations for the same angle was not consistent. We also used rotation sensors in our set-up. These sensors have a resolution of 1/16th of a rotation. We further geared each down with a worm gear, making the camera to sensor ratio 1/24.

## 2.6 LEGO Camera

The cameras that we used for all three methods were LEGO PC cameras (see Figure 2.4).



Figure 2.4: LEGO camera

They have a maximum frame rate of 30Hz and a maximum resolution of 352 (horizontally) x 288 (vertically) pixels, although the maximum resolution obtained with the Robolab image grabbing icons is 320 x 240. The actual frame rate acquired with the Robolab VIs varies, but it never reaches 30Hz. It usually ranges around 15Hz and it depends on how large the image displayed is and if there is any processing being done on the image. Another limitation of the camera is its inconsistent lighting throughout the image. The edges of the image are darker than the center portion of it (see Figure 2.5).

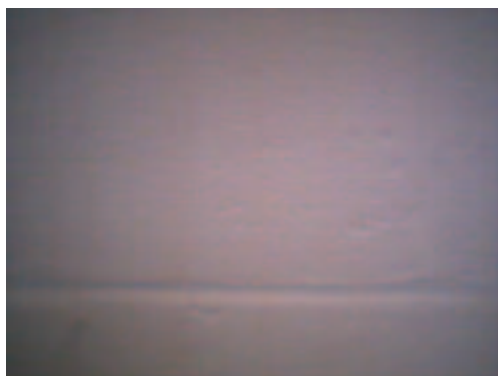


Figure 2.5: Uneven contrast in the image affects processing operations

This poses a problem when applying thresholds to an image, as it includes the darkness from the edges. The camera is also prone to noise, which deteriorates the sharpness of the picture. The camera has a relatively small focusing distance, which means that we had to manually focus the camera when changing the position of the camera just a few LEGO units.

The camera is manufactured by Logitech for the LEGO Corporation and it uses an ST Microelectronics CMOS ST image sensor: STV0610. I could not get the specs on this sensor because it is proprietary information, but it is similar to the CMOS image sensor: STV0680 B which has the following specs: a pixel size of 7.5 x 6.9 mm and an image array size of 2.67 x 2.04 mm.

The LEGO cameras contain a very small lens, which makes it difficult to calculate its projective center. We measured it to be 2.01cm away from the front of the camera. The CCD was about 2.51cm from the front of the camera making the camera constant  $c = 0.5$ .

## 2.7 OpenGL VIs

All the 3D rendering in this software was created with OpenGL VIs. OpenGL is the most widely used 2D and 3D graphics application programming interface in industry [?]. OpenGL supports low-level 3D graphic operations such as polygons rendering and lighting specifications. The user can implement these polygons with the use of point, lines, or other polygons [?]. OpenGL also allows the user to implement texture maps on objects. We used 2D OpenGL rectangles to build 3D boxes that represent our models. We encountered two major difficulties when using this graphics application. The first one was being able to subtract two polygons. We needed to do this operation

in order to build the holes in the walls that represent the windows and doors of the building. Our second obstacle was being able to navigate through the 3D world without the models taking over the entire screen.

# Chapter 3

## Dot Photogrammetry

### 3.1 The Concept

Photogrammetry is an optical triangulation technique used to compute the position of an object in three dimensional space from a set of two dimensional images [?]. These images contain roughly the same information, because the cameras are aimed at the same set of object. Each camera has a unique position in three dimensional space. When we compare the images taken by the three cameras (see Figure 3.1), we can see that there is an offset in the position of the LEGO blocks created by the difference in camera location. If we move from the left camera to right one, we notice that the relative position of the objects change. The closer the objects are to the cameras, the greater the offset between images is. This offset is what makes it possible for us to calculate the position of those objects in 3D space.

To store all three images in one, we assigned each camera to one color plane: the right camera was the red plane, the center camera the green plane, and the left camera the blue plane. When we combine all three planes, we get an image similar to Figure 3.2. We can then extract each plane independently to compute the x and y locations of each object and use those six values to determine the position of that object in 3D

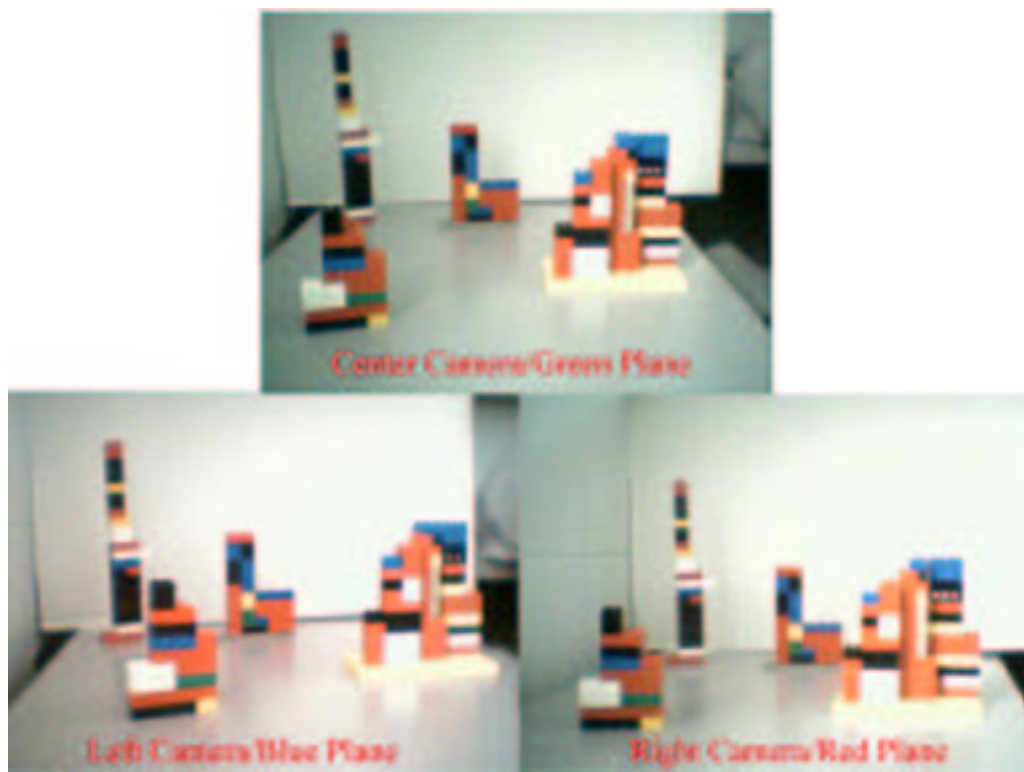


Figure 3.1: Separate camera views



Figure 3.2: The images from the three cameras combined into one

Each camera has an  $x$ ,  $y$ , and  $z$  location, measured from the center of the measuring volume, and a rotation, measured about the center of the each camera's projective center (see Figure 3.3).

These six parameters, together with the focal length, make up the seven extrinsic

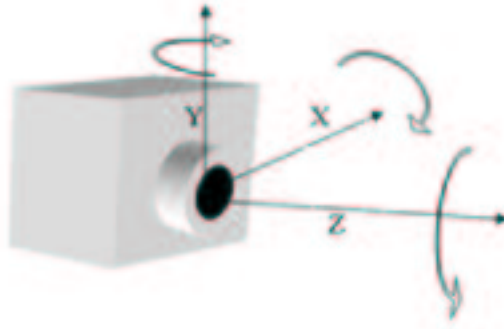


Figure 3.3: The six displacement parameters of the camera

parameters used in our calculations. Because every camera has a unique set of parameters, it is necessary to calibrate the camera arrangement before any calculations are attempted. The accuracy of the system is closely tied to the accuracy of calibration.

We used three LEGO cameras in our setup. Three cameras are more advantageous than two cameras because they reduce the number of ambiguities in the image pairs by one order of magnitude [?]. If only two images are used, all the points lying along one of the lines connecting an object point to its projection in the image plane cannot be distinguished from each other. A third camera significantly increases the possibility of correlating that object, because now, instead of having one line constraint, we would have two lines which would intersect to identify a single point. This also means that the more cameras used, the better the probability of identifying the correct point will be. Every camera added to the two camera set-up would lower the number of ambiguities by one order of magnitude. These ambiguities are basically eliminated when four cameras are used. For our purposes, the accuracy delivered by the three camera set-up was enough.

The photogrammetry software that I used was previously developed at a TUFTL laboratory by Dan Groszmann, who did his PhD. on the tracking of turbulent particles in 3D space [?]. The calibration software contains many parameters, such as

brightness and contrast manipulation that can be adjusted to obtain a very accurate point location. Since our specifications did not require such high accuracy, we simplified some of the more basic program operations, while retaining the core VI's which carry out the complex matrix calculations.

## 3.2 Calibration

Before using the photogrammetry software, the user has to create a calibration file that takes into account sixteen camera parameters (including camera position and rotation). The set-up has to be calibrated and the correct position of the cameras has to be found in order to satisfy the epipolar constraint used to identify the position of the points in space. The calibration file is obtained by taking a series of 5 pictures of a calibration plate at varying depth intervals from the cameras. The spacing should be chosen accordingly, so that all the holes can be seen in all five views. In Grozmann's set-up, the calibration plate was moved in 12.7 mm increments, for a total displacement of 50.8 mm. The plate was mounted onto a linear calibration table with a resolution of 25 microns. The plate used by Grozmann contains 25 holes, or four rows of six plus one hole in the center of the plate. This last hole is used to center the plate before calibration and is later covered during calibration. The holes are spaced at 1 cm intervals. To get a clear image of the holes, we shone a light from behind the plate in a relatively dark room. This process allowed us to threshold everything, except for the holes, out of the image. This plate was CNC machined, which your regular Robolab user would not have access to.

### 3.3 Set-up

The calibration plate that we used for the LEGO set-up was built out of black LEGO blocks. To create the bright holes on the plate, we place light bulbs inside the holes, giving the lights a spacing of 0.8 cm. We used 24 light bulbs, 4 rows of 6 bulbs, and we then drew a white dot with paint on the center block (see Figure 3.4).

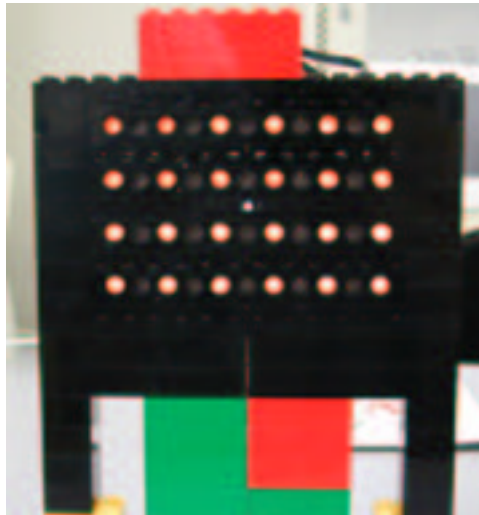


Figure 3.4: LEGO calibration plate

The light bulbs provided a good contrast with the rest of the background, so that everything else could be removed from the image with a threshold. One problem that I ran into with the LEGO light bulbs was that the lights in the calibration plate appear oval instead of round when looked at from an angle. This occurred because the lights are slightly inside the plate. This changes the calibration, because the centroid of the light is not the true centroid anymore. Figure 3.5 shows two images of the calibration plate. The one on the left was taken by the center camera and the one on the right by one of the side cameras. On the side camera image, we can see how the dots of light are slightly oval to the left. When we calculated the centroids

of all the dots, the ones of the side camera were shifted to the left, were as the ones of the center camera were in the middle.



Figure 3.5: Calibration plate as seen by the center and side cameras

Figure 3.6 shows a larger picture of the third row of light bulbs, where we can see how the circle drawn inside the dots on the right image are shifted more to the left than those drawn on the left image. This shift in centroids was not the only cause of error in our calibration process. There is also non-uniformity in the LEGOS, which means that the dots are not exactly equispaced.



Figure 3.6: The centroids shift slightly to the left because of oval appearance

We used a LEGO base onto which we affixed the LEGO calibration plate. LEGOs are molded with high tolerances, allowing us to move the plate in constant LEGO units. We moved the plate by two LEGO point intervals every time, which is equivalent to 15.8 mm.

We experimented with several supports for the cameras. The final one, consisted of three LEGO cameras mounted side by side on independent supports (see figure 3.7).

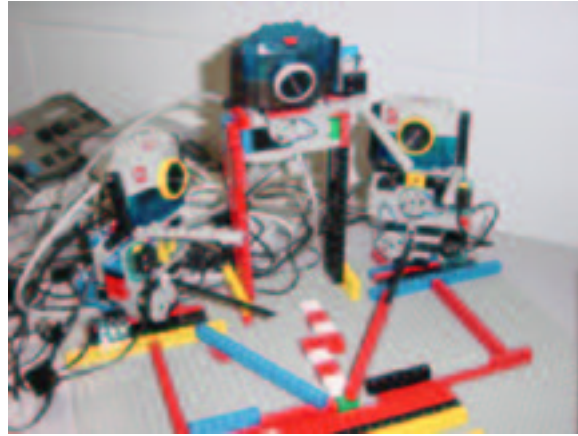
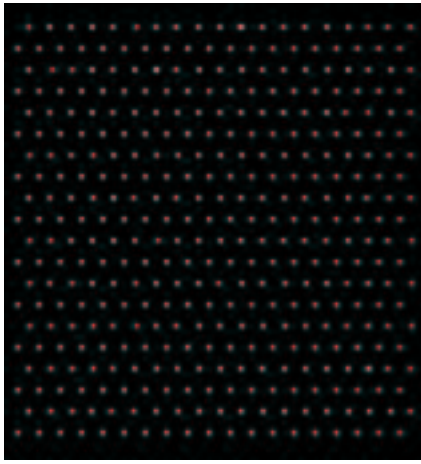


Figure 3.7: Three camera set-up

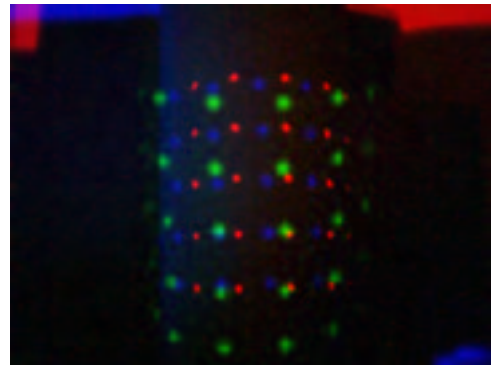
The support for the side cameras allowed them to rotate about their y and x axes. The center camera could move only about its x axis. Since we needed to know the angles at which the cameras were located, we connected rotation sensors to the supports. We geared the angle sensors down to one twenty-four rotations for every one camera rotation to get a better resolution, with the use of a worm gear. To make sure that we started from all angles equal to zero, before measuring them, we built a zeroing mechanism with touch sensors, which moved the cameras to their starting position. The LEGO rotation sensors did not provide a consistent angle measurement because of the slop in the gears and because the zero position varied slightly depending on the response of the touch sensor. Sometimes it stopped when it was pushed in slightly, while other times it needed to be fully pressed in order to activate it. We needed a relatively accurate angle measurement in order to get the iterative calibration method to converge. We therefore resorted to measuring the angles by hand with a protractor. All the motors and sensors used to operate this set-up were connected to a LEGO control lab interface or CLI. We chose the CLI, because the LEGO RCX did not provide sufficient inputs and outputs to connect all our sensors and motors.

As previously mentioned, originally, the pattern of dots required to identify the

shape of an object were projected directly onto the object (see Figure 3.8(a)). To simplify the procedure and need for a data projector, we constructed a cylinder out of black construction paper and we drew white dots on it to make them easily identifiable to the cameras (see Figure 3.8(b)).



(a) Pattern of dots projected onto objects



(b) Construction paper cylinder

Figure 3.8: Dots used for 3D location

### 3.4 Software

The first part of the calibration procedure is grabbing the five images of the plate. The images are grabbed in groups of three that represent each camera view. The three views are combined into a single color image by storing each one of them in one of the three RGB planes (see Figure 3.9). Those images are then loaded one by one and the software performs a series of image processing operations to isolate and locate the centroids of the points of light. First the user masks the region containing the lights. That sub-image is then divided into its three color planes and a threshold

is applied to each one of them to locate the x and y coordinates of their centroids.

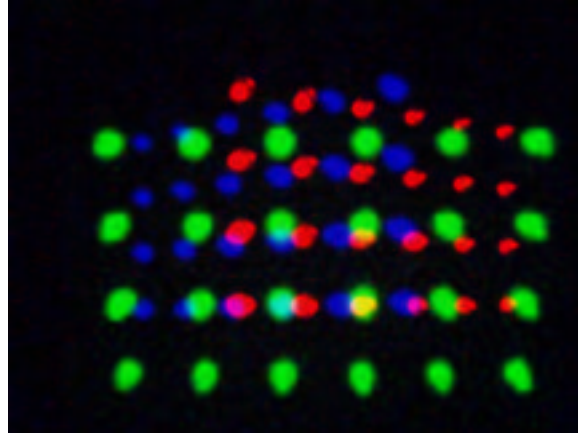


Figure 3.9: Points of light as seen by the three cameras

Those centroids are ordered, starting from the left side of the image and moving across and down and grouped into clusters of three, corresponding to each of the cameras. This information, together with the real world location of those points, is entered into a calibration VI that calculates the camera parameter file. This file defines the sixteen camera parameters for each camera and it is then used as an input for a separate program that identifies the point triplets, and with their locations in 2D space, it computes their real world 3D locations.

The cameras were calibrated through the method of least squares adjustment. This method is used to determine the camera parameters. The camera parameters are first estimated, giving them a one if the value is known to be accurate or a zero if it is not or unknown. A weight matrix is included to scale the accuracy of the measured values. The unknown camera parameters are computed by forcing the calculated position of the calibration points to approach their known measured values. These values are then corrected through an iterative scheme until the error between the computed positions and the known measured positions becomes negligible. We then

use spatial intersection to calculate the object's points once the camera parameters are known [?].

To determine the position of a particle in 3D space, we have to match the same particle in all three image planes by a stereopsis process. Since the camera orientations are already known from the calibration process, we can use an epipolar line constraint to find stereo pairs. Because of the small error introduced in the calibration, the epipolar line has to be given a certain thickness to account for this deviation. We used a stereopsis tolerance of 0.375 cm. Three cameras have the advantage that they produce an epipolar constraint of two intersecting lines, whereas two cameras produce just one epipolar line constraint. Figure 3.10 shows the epipolar geometry of two images. A point  $P_1$  in one image plane has a corresponding point  $P_2$  in the second image constrained to line  $e_2$ , the epipolar line of the second plane [?].

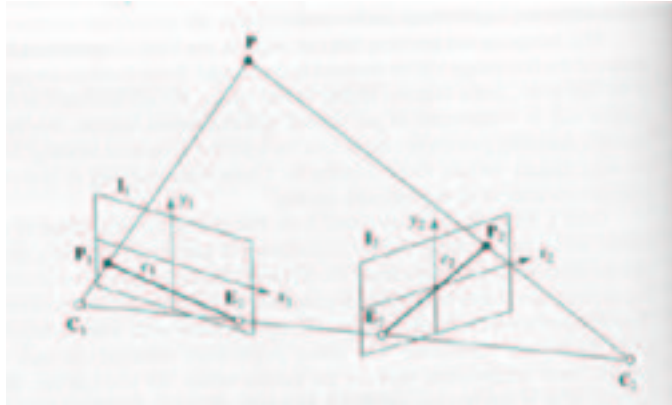


Figure 3.10: Epipolar geometry of an image pair

Once all three pairs of particles have been identified (camera 1 and camera 2, camera 1 and camera 3, camera 2 and camera 3) a spatial intersection determines the particles position in 3D space, by the direct solution of the collinearity equations for each point [?].

### 3.5 Results

With the photogrammetry method, we were able to measure the position of objects with very high accuracy. Our errors on average were 0.38 mm in the x direction, 0.23 mm in the y direction, and 1.5 mm in the z direction. These errors are obtained from the calibration process by comparing the position of the lights in the plate as calculated by the program to known ones that we measured by hand. The root-mean-squared (RMS) deviations between the computed coordinates and the known coordinates provide an estimate of the accuracy of the system [?]. The largest error, which occurs in the z dimension, or the direction away from the camera, is very minimal considering that the objects inside our measuring volume are up to seventy centimeters away from the cameras. Figure 3.11 shows a Matlab graph representation of the front of the cylinder that we built out of construction paper. The cylinder is graphed on its side, with the front side touching the bottom graph plane. The locations of the points drawn on the cylinder were calculated with the photogrammetry software.

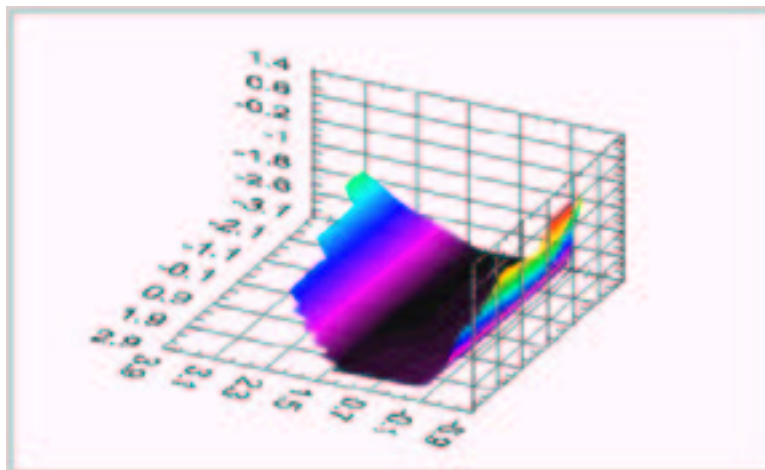


Figure 3.11: Matlab graph of points on the cylinder

The photogrammetry method as seen in the previous figure works. Unfortunately, the setup is too large (with the three cameras and the calibration plate) and the issue of needing to project dots makes it unusable for the kindergartner, who would not understand the concept. Therefore, our next attempt for this project was to simplify the set-up as well as the usability of the software.

# Chapter 4

## Stereo Vision

### 4.1 The Concept

In order to simplify our image grabbing set-up, we decided to remove one camera, and to use the concept of stereo vision instead. The method of stereo vision works similarly to human vision. Alike objects in the two cameras have just an x offset when both cameras are aligned so that their x-axes are collinear and their y-axis and z-axis are parallel [?]. Figure 4.1 shows the x offset with dashed lines. The offset, also known as the disparity [?], increases as the objects get closer to the camera. The solid lines show how the y position barely changes. The small y offset is created by camera imperfections.

Depending on the x offset, the z or depth dimension can be calculated. To simplify the usability of the software, we built up a look up table that compares the x offset between the cameras to a depth value which we experimentally calculated in LEGO units. When the user sets the camera in front of the object, the software computes the difference in the x position of that object in the two images. It then compares this offset to the data in the table to get a depth measurement. Once it has computed the distance in the z direction, the program combines the x and y measurements from

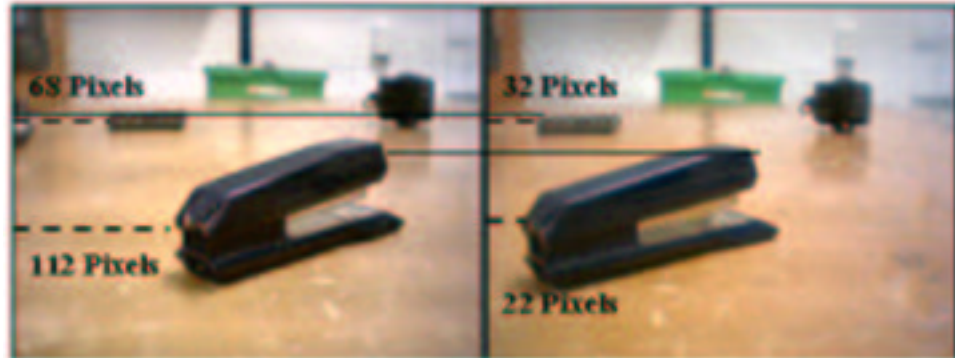


Figure 4.1: Image offset between two cameras attached to each other

the camera on the right to compute the coordinates of the object in 3D space. This procedure is less accurate than the photogrammetry technique, especially because of the low camera resolution, but it is also easier to use. Since there is no need to obtain sub-mm accuracy for children software, this stereo vision concept worked well enough for us.

## 4.2 Set-up

Our LEGO setup consisted of two LEGO cameras connected side by side with black LEGO connectors. They were angled forward slightly to include less background information, which may interfere with the correlation process (see Figure 4.2).

Setting both cameras together simplifies the setup for the user. It also maximizes the shared field of vision between the two cameras. If the cameras are moved away from each other, they have to be turned inwards (rotated about the y axis) in order to focus on objects that are close to them. When the cameras are angled in, they also lose part of their shared field of vision, since they would be looking in slightly



Figure 4.2: Two camera set-up

opposite directions. The closer the cameras are to each other, the smaller the difference between images is. The smaller offset is useful in speeding up the correlation process, because we can estimate where in the second image the same object is. A smaller separation produces less accuracy though, because as the offset gets smaller it is affected more by the measuring error.

With the two cameras attached to each other, we took a series of images of a rectangular object (see Figure 4.3).



Figure 4.3: Rectangular object used for calibration

The images were taken in sets of two: one for the right camera and one for the left

one. Every time a set of pictures was taken, we measured the distance from the block to the LEGO connector closest to the front of the camera. We chose to measure our distances in LEGO connector units to avoid the need for any extra measuring equipment. As we increase the distance from the object to the camera, the offset becomes smaller. We then looked at the pixel location of one of the corners of the object as seen by each camera to calculate their offset. Once we had obtained both values, the depth and the offset, for a significant number of intervals, we built up the look up table. This meant that for any matching points in the two images, we had a corresponding depth measurement.

### 4.3 Software

The first step that we took when designing this new code was to avoid the need for the dots required to identify the location of the object. We therefore decided to look for similarities in both image planes. The process of matching two similar images can be classified in two major approaches: feature-based and area-based [?]. The first code developed for this setup consisted on correlating sections of one image in the second image. We wanted to find an average depth for those sections. Since we did not know how large or small our sub-images had to be, we made their size a user defined input to experiment with different window dimensions. The program would correlate each section of one image with a larger section, also centered about the same point, in the second image. We then compared the x coordinates of the brightest spot in the correlated image, which is the center of the correlation, to the x coordinates of the template or sub-image. The offset between x coordinates would give us an estimate of the depth of the objects in the image. As we decreased the size of the window,

the correlation speed increased. If the window size becomes too small though, the larger second image will not include the objects being correlated, because the offset is greater than the extra window size. Figure 4.4 shows an example of a sub-window that is too small. The rectangle in the search image shows where the match to the template image is in the second image. Because the offset is too large, only half of the LEGO person is visible, and the correlation process fails to identify the right piece as shown by the rectangle in the image on the right.

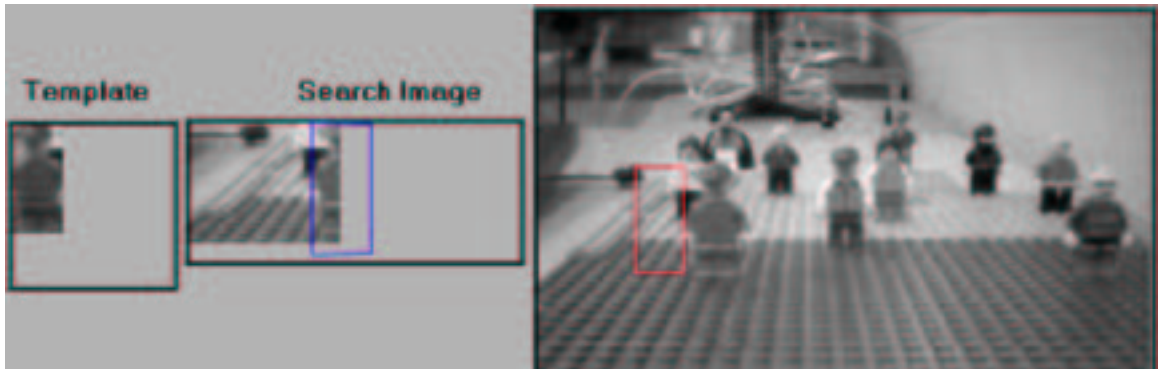


Figure 4.4: Wrong correlation from insufficient search image

We tried a range of window sizes, but the results were inaccurate, because we were including objects at different depths in the same correlating rectangle. We also ran the correlation using edge detection filters (see Figure 4.5).

These edge-detect filters make the process more likely to identify the same object in both images. The answers that we got for offsets did not make much sense though. When we graphed the offsets on an intensity graph to get an image of the relative depths in the picture, they were inconsistent with the original image. The objects are closer to the cameras in the bottom half of the original image, whereas in the intensity graph contains values in the range of zero to fourteen pixel displacements throughout, both in the top and the bottom of the graph, with no apparent pattern.



Figure 4.5: Image with a sobel filter applied

Since this technique was not very accurate, we decided to change our approach by matching similar objects together. Even though it is a simple task for a person to look at two objects and distinguish them, it is not that straight forward for a computer. We tried several methods of defining the objects in the image to correlate them, but more often than not, they would identify the incorrect pair of objects, especially if they were slightly behind the other in one of the camera views.

The code that worked most regularly consisted of matching up objects of similar size. First we found the objects in the image by doing a blob analysis. We used an auto threshold to convert the images into binary. We then applied a blob size filter to the image to get rid of the smaller objects which are usually insignificant. We then sorted the objects by size and matched the ones with similar size together (see Figure 4.6).

In order to use the blob analysis tool, we had to first isolate the blobs in the image. We did this by applying an edge detect filter to the image and then applying a threshold. We used two types of edge detect: a standard deviation (SDV) filter and a sobel filter. The SDV filter provided a thicker edge than the sobel. In some cases this was

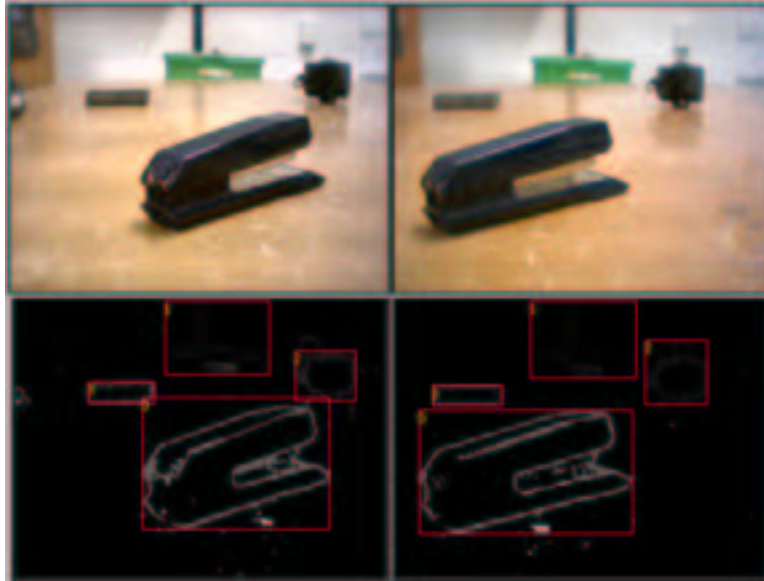


Figure 4.6: The original images and the processed ones with the objects matched preferable because the entire object could be identified, filling in any gaps created by edges that were difficult to isolate. On the other hand, the sobel filter was preferred when objects were closer together, since the SDV filter in that case might combine them together.

The SDV filter works by grabbing clusters of pixels and replacing the cluster values with the standard deviation of all the values in it. This means that if there is a sudden change in intensity, produced by the edge of an object against a lighter background, the standard deviation would be very high, creating a brighter pixel line around that object than inside or outside of it. This was very helpful for our calculations because it helped us to isolate the individual objects. We tried four different cluster sizes: 2x2, 3x3, 4x4, and 5x5. Clusters larger than 4x4 made the image too blurry, reducing the sharpness of the edges and the 2x2 cluster did not provide enough edge detection. We therefore chose 3x3 clusters because they provided the sharpest image with the greatest edge detect. The sobel filter computes the gradient along both axis, x and y, to obtain the edges of the objects where a sharp change in contrast occurs.

We then applied either a manual, with an experimentally defined range, or an auto threshold to the images in order to convert them to binary before using blob ID. With the blob ID we were able to obtain the size of the object and to match them in both images. We also applied a particle filter to get rid of small blobs caused by glare or noise. With this particle filter though, we lost information about the larger objects that might help us to better identify their location. Larger objects can have protrusions or indentations that would be at a different distance from the camera than the edges. We therefore further subdivided the main image into the bounding rectangles of the larger blobs and applied the same edge detect/threshold process to the sub images to obtain more information (see Figure 4.7). Similar work in the stereo matching area was carried out by A. Aguado and E. Montiel who matched stereo pairs through a piecewise linear model. This model proved to be efficient when representing regions containing a uniform disparity or for modelling abrupt changes in depth [?].

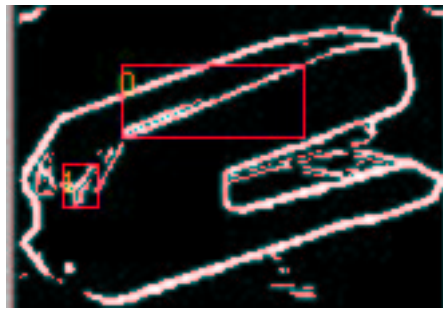


Figure 4.7: Additional details identified in larger object

To calculate the distance of the object from the cameras, we used the x coordinates of the corners of the bounding rectangles in both images and subtracted them to obtain the offset of the object. We used both x coordinates, because if the object is on an angle, one of the edges will be farther away from the camera and therefore the

offset between corners would be different. Even if we used both corners, this method was not accurate enough. It works well when an object is parallel to the plane of view, but once there is an angle to it, the perspective of the object produce a bounding rectangle whose corners are not at the closest and furthest points on the object. If the object is on an angle (see figure 4.8), the offset of the right corner, in this case, would be smaller than that of the left corner, because the right side of the object is further from the camera than the left. The closest point that we can compute is the left corner, which is not actually the closest point, since the corner of the stapler that is touching the bottom edge of the rectangle is closer to the camera.

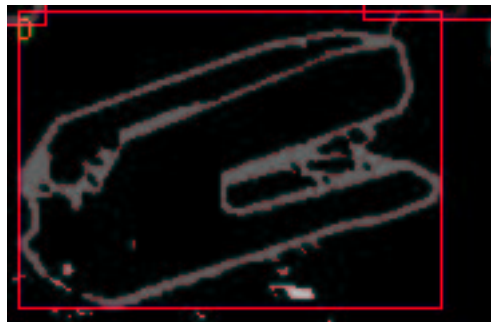


Figure 4.8: Picture of an object on an angle

The blob ID concept is not practical in our application because if two objects of similar color are overlapped in an image, the edge detect won't have enough contrast information to distinguish them apart, and the computer will identify both objects as one blob. This becomes a problem when the computer tries to look for a large blob in the second image where, because of the different camera position, the objects are not overlapping, and each was identified as two smaller ones.

For the final iteration on the two camera method, we decided to allow the user to select each object in one of the images. The software would then find that same object in the second image, using a correlation technique, and it would also compute

the offset between images. To make it easier to build the setup, we decided to use LEGO people. We took about ten of them and spaced them out on a LEGO base (see Figure 4.9).



Figure 4.9: LEGO base with ten LEGO people

The new code would then ask the user to select each of the LEGO people, one at a time, and it would compute their distance from the camera. LEGO people also have a relatively constant depth and no angle, when they are facing forward, which simplifies their location in 3D space. Once the user has identified all of the people, the computer draws up an OpenGL representation of the location of the people in 3D space. With this information, the user could then send a LEGO rover around the people without crashing into any of them as depicted in Figure 4.10. Unfortunately, the rover is no longer autonomous, requiring the input of the user in order to navigate around the obstacles.

## 4.4 Results

The stereo matching set-up has the advantage, over the photogrammetry one, that there is no need for a projection of points onto the objects to recognize their position.

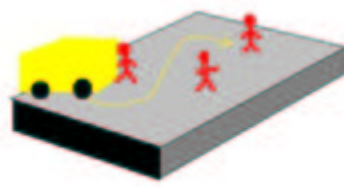


Figure 4.10: LEGO rover navigating around the LEGO people

The bounding rectangles that we used instead of the points though do not provide as much information. This set-up is also simpler since it only requires two LEGO cameras as opposed to three. Reducing the number of cameras lowers the accuracy of the system. The resolution of the LEGO camera is relatively small at 320x240 pixels. As we move the objects farther away, their edges become blurry and it becomes difficult to calculate the offset accurately. At a distance of 100 LEGO units, or 80 centimeters, the depth measurement carries an error of two to three LEGO units, or 1.6 to 2.4 cms. Beyond that the error keeps increasing, becoming greater than 10 LEGO units once it surpasses a distance of 150 LEGO units from the camera. Since this software is designed to correlate objects faster by searching in the same relative area in both images, this specification creates a problem when correlating objects that are very close to the cameras, because the offset becomes larger than the search area.

To account for these two constraints, we kept our objects, in this case LEGO people, in the range of 25 to 58 LEGO units. We can calculate our largest and smallest offset from this range and those offset values allows us to specify the size of the sub image used for correlation, so that we will include the LEGO person that we are looking for. We chose our window size in the x direction to be three widths

larger than the window selected by the user, which should be the width of the LEGO person. Since the cameras are at the same height, there should be no  $y$  offset, and we should be able to use the same  $y$  co-ordinates from one image to the other. Due to variations in the camera construction, we found a 6 pixel difference in the two images and took that into account in the correlations. The correlation process proved to be very consistent. It correlated all ten people in Figure 24 in about one second each. We did run into problems when one of the people was partially blocked by another in one of the images, especially in the template image. The program was able to identify the correct piece when only about one third was occluded. If the view was obstructed more than that, the program would not identify the pair of figures. Stevens and Beveridge [?] designed an algorithm which uses 3D CAD models in order to recognize the shape of occluded objects. They use an iterative method to obtain the shape and orientation that best fits that object. Figure 4.11(a) shows one of the pairs of images taken of the LEGO people and Figure 4.11(b) shows the OpenGL rendering. In our current set-up, the people are represented by rectangles of predefined values. In the future we would like to include the extracted picture of the sub-image as a texture on the corresponding block.



(a) LEGO people set-up



(b) OpenGL rendering of LEGO people

Figure 4.11: LEGO people in 3D

The stereo matching system proved to be simpler to use than the photogrammetry method. It did not require a pattern of dots to identify the objects and consisted of only two cameras connected side by side. Since we did not require the accuracy of the photogrammetry method, the stereo matching proved to be a better alternative, with the drawbacks that the user has to identify each object manually and that occlusion might interfere with the identification process.

# Chapter 5

## Plane View

### 5.1 The Concept

Having tried several methods to identify objects in three dimensional space, we decided to deviate from this concept and instead provide the computer with an outline of the objects that comprise the space and allow the computer to render them in 3D. In the previous chapters, we designed several interfaces with which the user could build a 3D world in the computer by first creating and placing all the physical objects in front of the cameras. The computer would use the location of those objects as well as their shape to recreate the world on the screen. With the plane view concept, the user would only build a basic outline of the world, providing the computer with less visual information than before. Instead, to fill in the missing information, the user employs LEGO blocks of a range of colors and sizes. Using LEGO blocks, the user would be able to construct a building and the objects inside of it fast and easily.

Plane view extrusion works by constructing or extruding a 3D image from a 2D picture. With the extrusion program, the user can build a 3D house on the screen without any previous programming knowledge. All she needs is a set of LEGO blocks and a camera and the computer does the rest by identifying the blocks and replacing

them on the screen with corresponding virtual object. We wanted to design something simple to use so that kids could also build their own 3D models. Our set-up as well as our software is designed with children in mind. We built our set-ups with LEGO blocks that children can easily assemble and programmed our code in Robolab software which is designed to introduce children ages eight and up to programming. At the same time, we also wanted to create software that could be used by professionals, such as architects, to render more complex buildings, by providing them with textures and objects that they can insert into their design. Over all, the interface had to be very user friendly, so that the models could be designed in very little time.

Our program works by identifying blocks of certain sizes and colors and matching that information to a pre-programmed list of available choices. The current code can recognize five colors, each of which represents a specific virtual object. Red, for example, represents a window. The user can therefore build the outline of the house with LEGOs, see its 3D representation on the screen, and quickly change the design by adding or moving pieces around. The other four colors green, blue, black, and yellow represent doors, furniture, walls, and stairs respectively. OpenGL is widely used all over the world and there are many pre-built models that the user can download from the web. With this software, the user would have the option of downloading a set of object libraries from the web to personalize his design. This libraries would include furniture models to replace the LEGO blocks that the program uses to build the virtual model.

Current methods of building 3D models require the user to work with 3D images in a 2D environment. It is much simpler to understand the relationship between objects when we see them in an actual three dimensional set-up, in this case a physical LEGO one, than on a 2D screen. With current 3D software, the user has to either construct

the shapes by specifying the size and orientation parameters, or click and drag from a list of pre-defined ones, all of this with current input methods, such as the mouse or keyboard, that are constrained to a two dimensional motion.

## 5.2 Set-up

Instead of using the two camera set-up we decided to simplify the interface by reducing it to just one camera. The one camera would take a picture of the LEGO blocks from the top, losing therefore one dimension: the y or height (see Figure 5.1).

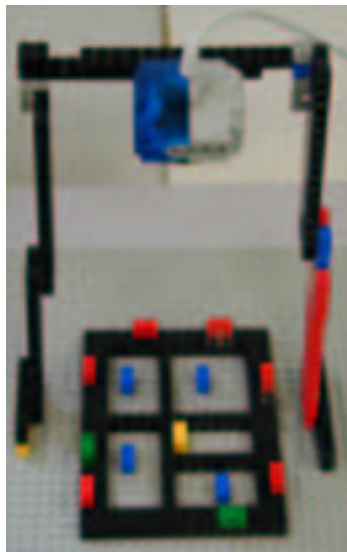


Figure 5.1: The one camera set-up with the outline of a building

To account for this lost dimension we had to make a default height, which later can be changed by the user, for the different objects comprising the building. The program became more versatile and easier to use. The user would build up an outline of the house with LEGO blocks. Black blocks that are two LEGO sizes wide became the outside walls and the single LEGO size the inside walls. The user can then place

windows, doors, furniture, and stairs in the house which are 2x1 LEGO blocks colored red, green, blue, and yellow respectively. For the outside walls, the doors and windows are placed on top of the black blocks on the outside edge (see Figure 5.2).

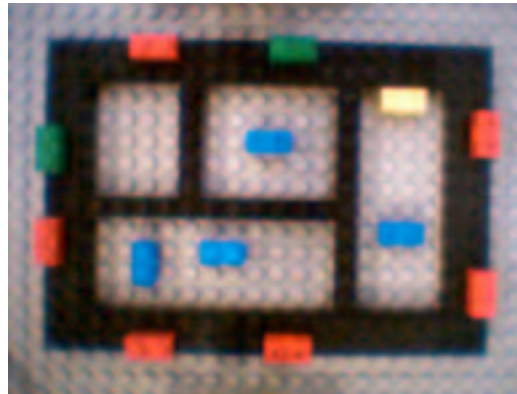


Figure 5.2: Sample outline for the 3D extrusion method

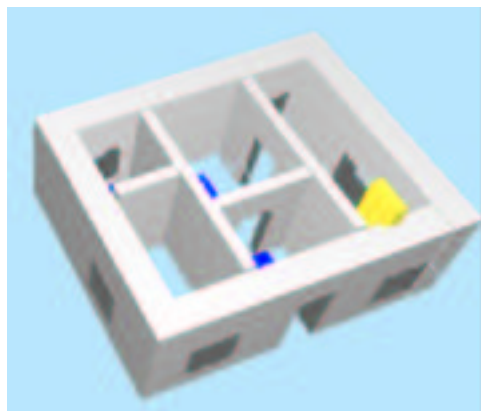
Since the inside walls are only one LEGO size wide, we had to place the doors next to the wall so that the program could still recognize the complete wall. If we placed the block on top, the software would identify the wall as two pieces instead of one. The inside walls are also limited to forming rectangular rooms, which means that a room cannot take on an L shape. Figure 5.3(a) shows a sample outline that includes two interior doors and one exterior one. The OpenGL representation of that same outline can be seen next to it in Figure 5.3(b).

Once all the software has identified and placed the blocks on the screen, the user can modify their position by clicking on the appropriate object in the camera view. The user can move the doors and windows within the plane of the wall and the furniture and stairs within the plane of the floor using a simple arrow interface (see figure 5.4).

The user can also add multiple stories to the building (see Figure 5.5). Once all the modifications are done in the current level, the user can press the next level



(a) Sample outline with interior walls



(b) OpenGL representation of Figure 5.3(a)

Figure 5.3: Interior doors in 3D

button and take a second image of the same or a different set-up to create the next floor. After that, the user can modify the position of the objects in the current floor accordingly.

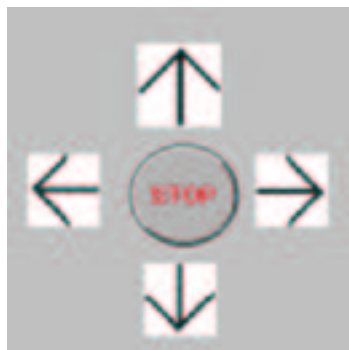


Figure 5.4: Panel used to arrange objects

Finally, as the house is being built on the screen, the user can walk around in the virtual world by either rotating the view or by navigating through the view. The rotation option allows the user to rotate the building about the x or y axes. The user can return to the zero rotation view by holding down the shift button and clicking on the screen. To navigate through the image, the user would select the navigation option and use his mouse on the screen to change the x position by moving the mouse

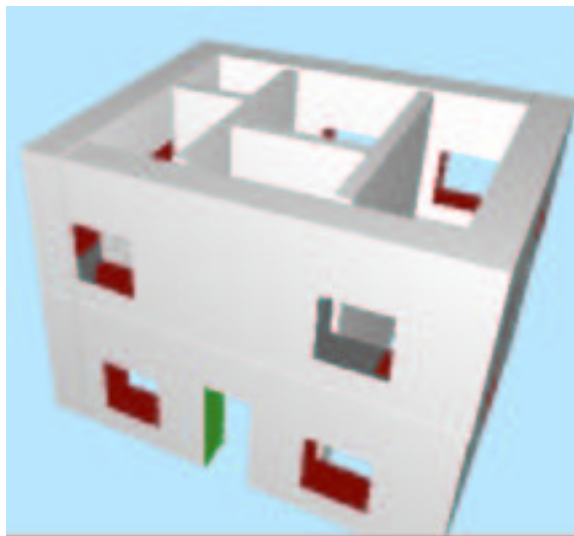


Figure 5.5: A building with multiple stories

left or right or zoom in and out in the  $z$  direction by moving the mouse up or down respectively. We use the  $x$  and  $y$  locations of the mouse pointer inside the picture screen to determine the navigation. This method of navigation is far inferior to a joystick or some sort of 3D mouse, which could easily be added to the software.

### 5.3 Software

This program works by identifying each part of the building separately. A flow chart of the steps taken by the program is shown in Figure 5.6.

First of all, the software identifies all the black in the image. This includes the outside and inside walls. It does this by setting to zero all the pixel values that are less than a critical number in the red, green, and blue planes (see figure 5.7). This critical value was determined experimentally and varies depending on the intensity of the ambient light. It is usually around 90 (out of 255), since the intensity of black in any of the three color planes would be very low.

## Recognition Process



Figure 5.6: Flow chart of the block recognition process

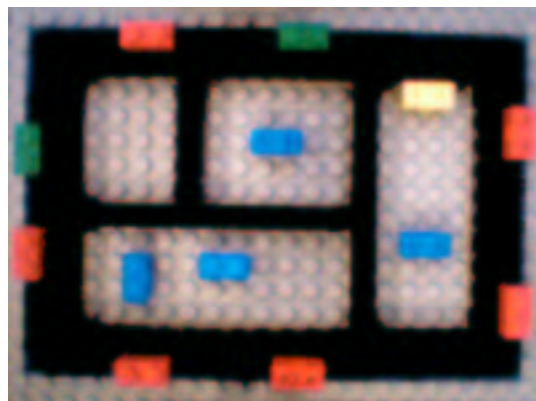


Figure 5.7: Outline of the building with all the black pieces removed

Glare on the block should be avoided since this is seen by the camera as high intensity and therefore not recognized as black, but merely as a void. We then find the walls by following the steps outlined in Figure 5.8. We first apply a threshold to the image to find the largest blob which gives us the bounding rectangle of the entire base of the house. The thickness of the wall is defined by the program so that shadows or other errors in the image won't create very thick or very thin walls.

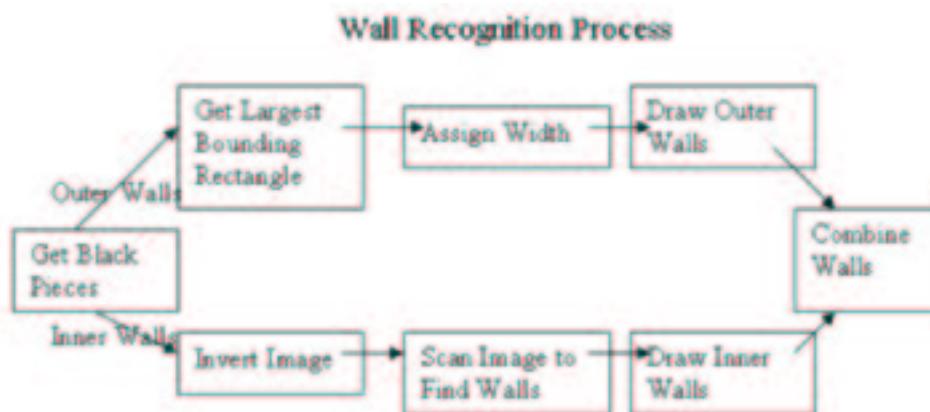


Figure 5.8: Wall recognition flow chart

At the same time the software also identifies the inside walls. It does so by inverting the image of the walls and finding the gaps between the rooms, which are equivalent to the inside walls. Once it has made a separate image with just the inside walls, it scans the image horizontally and vertically to find each wall, deleting them after they are found, and looping until there are no more walls left in the image. Figure 5.9 shows the outline of a building with all the walls identified.

After the program converts all the black pieces to zero pixel value, it continues by removing the gray background. We used a large gray LEGO base to build the outline of the building and this interferes with the recognition of the colored blocks. When graphed on an RGB plot, gray appears at about the same intensities in all axes. To

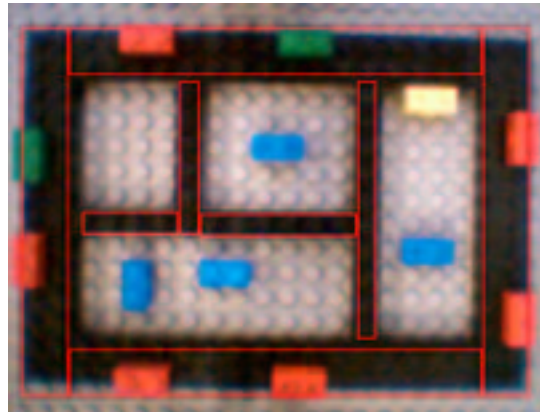


Figure 5.9: Complete ID of walls

label a pixel as gray, the standard deviation of that pixel value in all three planes had to be less than an experimentally defined value which we choose to be thirty (see Figure 5.10).



Figure 5.10: Average standard deviation of a piece of gray LEGO plate

The standard deviation in this figure is much lower than thirty, at 6.83, but we found that if we used such a low number we had too many small particles left over that were recognized by the program as blocks. Once again, this value varies depending on the ambient light. The lower the intensity of the light, the more gray will look like black and the smaller the standard deviation will be. Thirty is a safe value because

it is relatively small. Of the colors that we chose, red and blue have a much higher standard deviation than gray, so they stand out very well when the gray is removed. The image sensor of the LEGO camera is not very sensitive to the color green and therefore this color appears relatively dark even in its own plane. Since its intensity is barely over one hundred, its standard deviation is also small. For this reason, it is sometimes partly removed during the gray removal phase. An average standard deviation of the LEGO blocks is shown in Table 5.1.

Color	Average Standard Deviation
Red	59
Green	32
Blue	74
Yellow	20

Table 5.1: Average standard deviation for LEGO colored bricks

The second criterion to label a pixel as gray was that the mean of the pixel value in all three planes had to be less than a defined value. This value, which varies with light intensity, just like everything else, is 235. The purpose of this criterion is to keep the yellow blocks from disappearing. High contrast objects, such as yellow blocks, have a high pixel value in all planes, making the standard deviation small, but the mean large. Unfortunately, when we keep the yellow blocks, we also keep glare that sometimes appears in the rooms. The image on the left in Figure 5.11 shows the gray plate before it is removed, while the image on the right shows how most of the gray has been identified and removed by the labeling process.

To delete the objects created by the glare, we then use an auto median morphology filter, which creates simpler particles with fewer details, deleting small noise and splitting larger noise into particles that could be deleted with a particle filter. We therefore used a particle filter to delete all particles under 200 pixels in area.

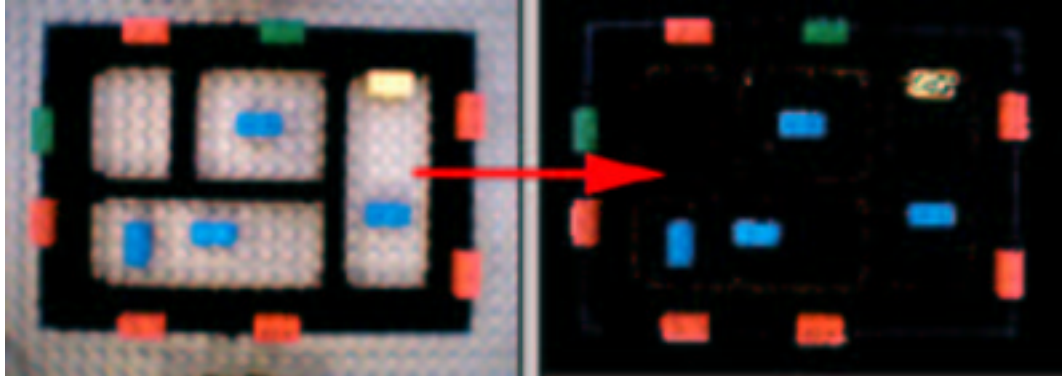


Figure 5.11: The removal of the gray plate from the image

The last part of the code is identifying the five colors of the blocks. Once all the colored blocks have been identified, we extract each one of them with the information of their bounding rectangles. We then compare the mean of those pixel values in the RGB planes. The red, blue, and green pieces are straight forward in identifying, since they will show up brighter in their color plane. To identify the yellow, we chose only means that were greater than 120 in both the red and the green plane. Figure 5.12 shows all the colored blocks from the outline of the building.



Figure 5.12: Color blocks identified in the image

Once all the walls and objects have been identified, we build up an array with their

information which we use to construct the OpenGL objects. Figure 5.13 shows all the blocks that have been identified with a bounding rectangle.

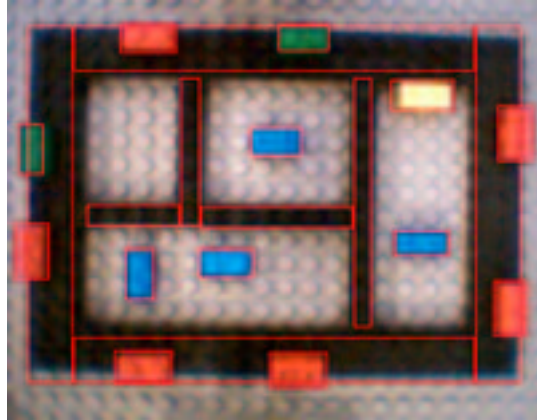


Figure 5.13: All the blocks have been identified

The OpenGL objects are build in a similar fashion to the previous codes. The height of the objects is set, since the camera cannot get that information from its two dimensional view. Currently all objects are represented by rectangles and the only rotation is computed in ninety degrees angle increments. The objects are also assigned a color to match the blocks on the picture. If the user decides to change the location of one of the objects, or nudge it, she would click on the corresponding one in the image of the LEGO outline, and the computer would identifies the x and y position of the click. It then matches up this position, with the block in the OpenGL picture that corresponds to it. Once the right blocks has been identified, the program moves it up/down, forward/backwards, left/right, depending on the arrow that the user clicks on and the type of object it is. The update occurs in real time since the changes are applied to a global VI.

The plane view method provides 3D renditions of the LEGO outline in a very small amount of time. The software does have some limitations that limit its interface

flexibility. Originally, we took a picture of the outline holding the camera right over blocks with our hand. If the camera was not in line with the blocks and the image appeared slightly skewed, the program would identify the walls incorrectly, because the bounding rectangle of the major blob does not comprise the edges of the walls anymore. We therefore built a support for the camera to ensure that the walls were aligned with the image edges. The support forms several right angles in order to hold the camera in the same plane as the base. Since some angles are difficult to build with LEGO blocks, the support is also not as straight as we would have hoped by using LEGOs. Even now that we use the support, the plane of the camera might be at a slight angle with the plane of the outline, causing uneven walls that produce extra internal walls in the 3D picture. The user is given the option of accepting the recognition of the blocks before they are built on the screen in case something is recognized incorrectly.

When a threshold is applied to the image, the walls do not show up as straight edges due to uneven lighting and shadows. We therefore could not assign the thickness of the walls as computed with the blob ID bounding rectangle to the virtual walls. In order to make the walls even, we assigned them a constant thickness, and sub-divided the global bounding rectangle so that the walls are just touching. Because of the way the search algorithm works when defining the internal walls, the user can only build rectangular rooms.

Other limitations of this software include being susceptible to glare problems. Fortunately, the light right above my desk does not work, so most of the times I did not have to worry about blocking the light from the set-up. When I did move it around though, any light that is reflected by the blocks, especially the black ones, will make the program to incorrectly identify them. The user should therefore place some sort

of shield (a LEGO base works well) right on top of the camera to ensure a consistent 3D world rendering. The identification of green pieces is also readily affected by the light. Whereas glare is a major problem for black blocks, too little light is also a problem for green blocks. Green LEGO blocks have relatively low intensity values in all three planes, with an average of just over 100 (out of 255) in its own. For this reason, if too much light is blocked from these pieces, the program might just consider them as black pieces and it will not therefore build the doors that they are supposed to represent.

## 5.4 Results

Our block identifications are very dependent on the ambient lighting. We have designed the software to operate under relatively bright conditions, comparable to the lighting in an office, with fluorescent ceiling bulbs. Care should be taken so that no lights shine directly on the LEGO pieces creating glare. The glare can pose several problems. When it shines on the black pieces, it increases their intensity and the program does not identify the block as a wall. If the color black is not removed at the beginning of the ID process, it might affect the dimensions of the color pieces, or it might just look like a separate block. Since light black resembles gray, the black that is not removed with the walls because of glare, could be processed out of the image with the remove gray option. We can do this by increasing the minimum variance required to classify something as gray. The problem with that approach is that we also begin to remove part of the yellow blocks, since they have similar intensity in all three planes. We could also remove the glare created on the gray base by increasing the maximum mean value required to classify a color as gray. But again, the yellow

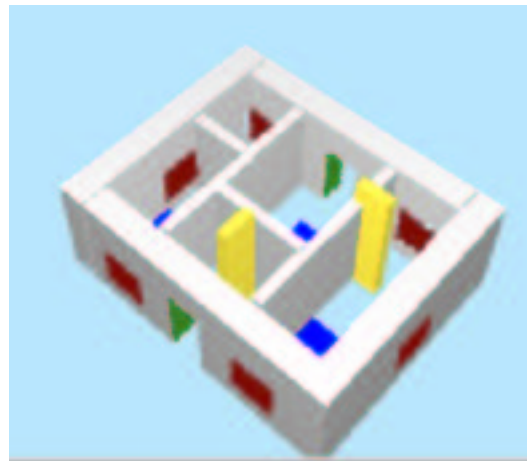
mean is slightly below that of white, and by deleting the bright spots, we also delete the yellow blocks.

This type of set-up, building a LEGO outline of an entire floor, limits the user to the amount of detail that she can use. When there are multiple rooms as in the pictures shown, the user does not have very much space to place blue blocks, furniture, inside of those rooms; they quickly take over all the space available. The rooms can be made larger by increasing the size of the outline over all and moving the camera farther away from the set-up. If the camera is moved too far from the blocks, they will appear too small to be recognized by this program, which uses particle filters to remove small blobs that are caused by glare, shadows, and camera imperfections.

Figures 5.14(a) through 5.16(b) show sample outline set-ups that can be built with the current system.



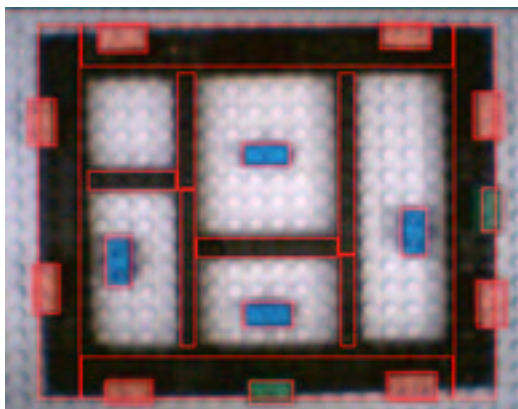
(a) Sample LEGO outline



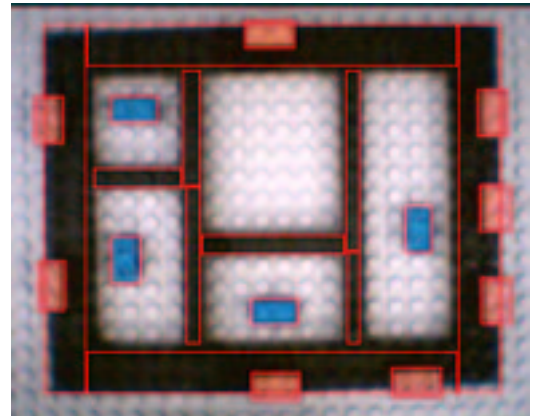
(b) Corresponding OpenGL rendering for Figure 5.14(a)

Figure 5.14: First extrusion example

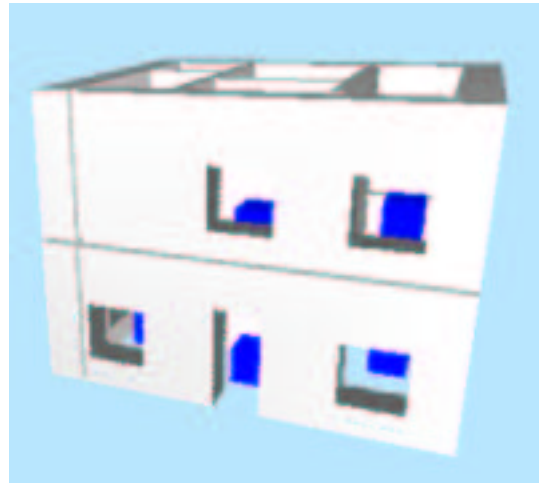
There is still much to develop with this extrusion method. We have a few ideas for the future that will hopefully be implemented to extend the capabilities of this



(a) Bottom floor

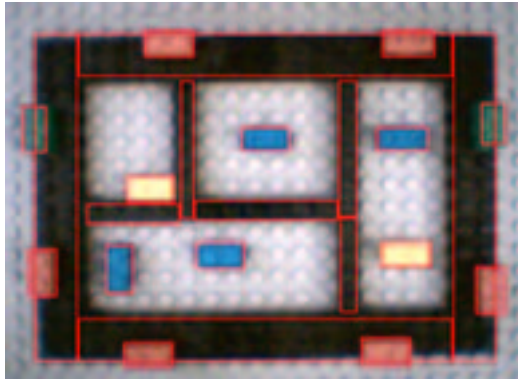


(b) Top floor

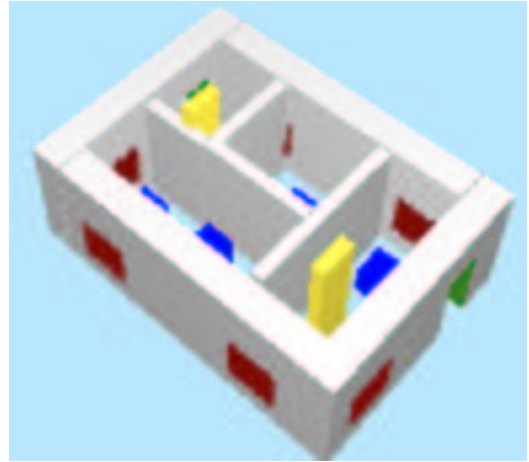


(c) OpenGL rendering of Figures 5.15(a) and 5.15(b)

Figure 5.15: Second extrusion example



(a) LEGO outline



(b) OpenGL rendering of Figure 5.16(a)

Figure 5.16: Third extrusion example

software. Right now, the user is able to see the 3D model from different angles and positions by rotating or translating it with the use of a mouse. In the future we would like to implement a fly-thru capability, where the user can make a movie of the set-up by specifying a certain number of positions that the camera has to follow. When the user plays the movie, the software would design a path to join the points that the user has specified and would then proceed to navigate or fly through the model.

We would also like to offer ways to add more detail to the 3D model. Right now, the space available to the user is very limited. The camera can be moved further away from the outline to increase the building area. Unfortunately, because of the small resolution of the LEGO camera, the colored blocks are difficult to identify if the distance from the camera to the base is too large. To overcome this problem, we would give the user the option of building each room individually. The user would either take a picture of the entire floor, as it is done now and then click on each room to replace it with an image of just that room, or he would take a picture of each room, and then decide the placement of each of them. By increasing the building area, the

user has more space to place furniture, therefore giving the building a more detailed and personalized look.

# Chapter 6

## Design of a Software Tool Set

### 6.1 The Concept

Image processing is a complex subject that is becoming more and more popular in jobs, especially as we move towards a more automated industry. The goal of this thesis is to provide an introduction to this subject, to younger people, who might not have the opportunity or the option of taking a class about it. Robolab already contains most of the tools needed to build the 3D world building software on which this thesis is based. These tools are designed for more experienced users who already have some background in image processing. In order to simplify the learning process, we designed a software tool- set that works through Robolab, by grouping many of these functions together to explain simple image processing concepts, such as grabbing an image and applying a threshold to one of its color planes.

The tool-set is then used in a web-based tutorial that anyone with an Internet connection can access anywhere in the world. This tutorial is based on challenges used for a college level introductory image processing class. The challenges are distributed into four categories:

1. Grabbing an image
2. Binary morphology
3. Image restoration
4. Color analysis

The final challenge of this tutorial is to write the code for a LEGO 3D world builder similar to the plane view extrusion designed for this thesis. The web site would guide the user through the different categories, providing them with hints, and also a sample solution which they can download on to their computer.

## 6.2 Design Approach

Initially, we were going to build a web site that contained a set of LEGO explorations that would teach the user image processing concepts. Each exploration would explain one or more of these concepts with the use of a LEGO model that the user would build. The models ranged from a simple dice dot counter to the more complex LEGO block sorter, which classifies blocks in respective bins, according to their color and size. All these set-ups used the LEGO camera as a sensor. One of the models that we created was a target shooter. The target consisted of a green LEGO brick inside a red wall. This example explains the concept of color planes, which is needed in order to distinguish the target colors. The user had to build a LEGO device that would scan the wall to find the brightest spot, or the target, and shoot a LEGO projectile to hit it. We thought that building models would be a fun idea for people in general, because building with LEGOs is fun. Unfortunately with this approach,

the user would spend too much time building the models, making the learning curve very slow. We wanted to focus more on the image based concepts and not on the building ones. We therefore decided to simplify the building requirements for the

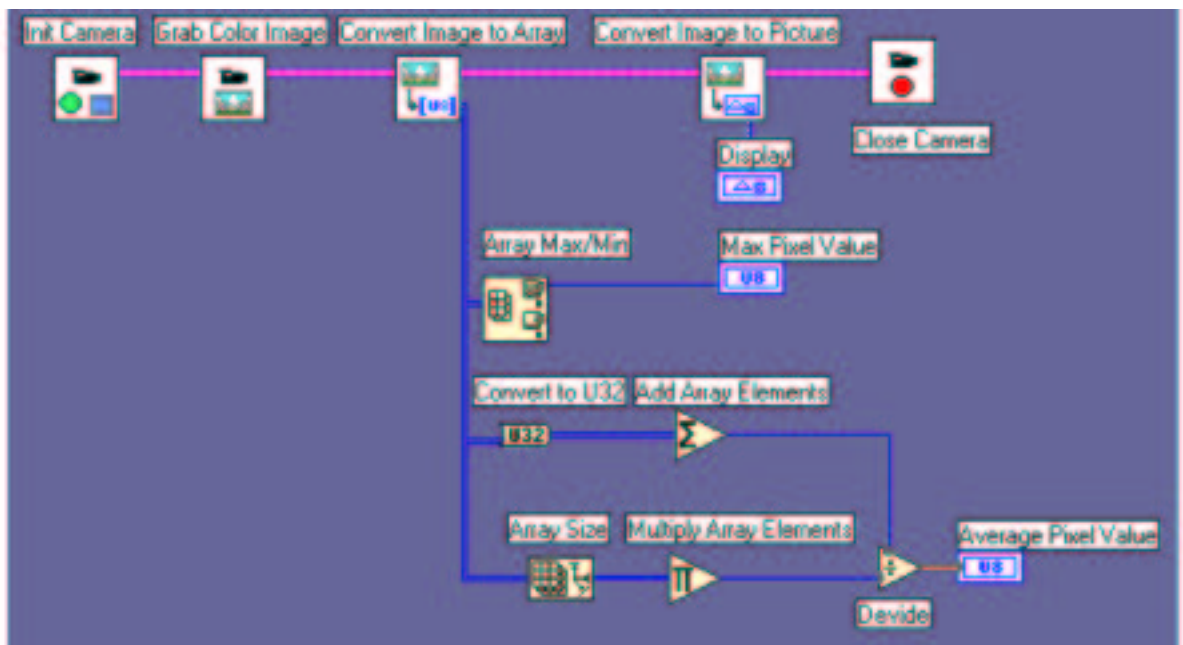


Figure 6.1: Windows Explorer look of web-site tutorial

We begin our tutorial by introducing the user to the LEGO camera and some of its more basic capabilities with Vision Center. Vision Center is a Robolab control panel that allows the user to experiment with functions, such as color planes, and to define sensors using image processing functions. This whole process is done by

pointing and clicking with the mouse and selecting from drop down menus. The user can get a feel for the power (or difficulties) of image processing without the need to program anything. The capabilities of Vision Center are some what limited, because the user can only experiment with what has already been programmed.

Next, we introduce the user to image grabbing and manipulation with the Robolab software. This tutorial does not include a section on using Robolab. We assume that the user already has enough experience with this software, that they know such basic



This program captures an image and computes the max pixel value and the average pixel value.

First, you have to initialize the camera, grab a color image and convert the image into an array. Compute the max pixel value in the array. Compute the average pixel value by: adding all the elements in the array and dividing that sum by the number of elements. Convert the image into a picture and display it. Close the camera.

**Hint:** Array processing icons are under G Code / Array palette menu.

Figure 6.2: Sample tutorial page

The picture is not as comprehensive as the actual program, because the user cannot

click on the different icons and structures to see how everything works. The image grabbing section includes the topics of color planes, look-up tables, histograms, and image arithmetic. The binary morphology section covers applying different kinds of thresholds, blob analysis, and common morphology operations such as edge detect and eroding. In the image restoration part of the site, we introduce the user to spatial and non-linear filters, filtering in wave space, correlation, and noise reduction. We end the tutorial with color analysis techniques which cover HSI and RGB planes, color thresholding, and color manipulation. The 3D world building example consists mainly on color analysis. We therefore left this topic until the end, so that the next step of the user would be to write the 3D world software. Finally, the current web pages also have a brief introduction to some of the image processing terminology. Before each challenge there is an introduction page that explains some of the concepts before the student attempts to complete the challenge.

## 6.4 Grabbing an Image

Our tutorial begins with the introduction of the most basic, but also the most important, concept of image processing, grabbing an image. The topics covered in the Image Grabbing chapter are seen in Figure 6.3.

As mentioned previously, we introduce the user to image processing through vision center, a Robolab control panel that allows the user to get an idea of the power (or difficulties) of image processing without the need to program. Vision center also allows the user to get to know the LEGO camera. The user can easily tell if the camera is plugged in right, if the right camera is selected, or if it is showing an image. These are questions that are easily answered with Vision Center, but require

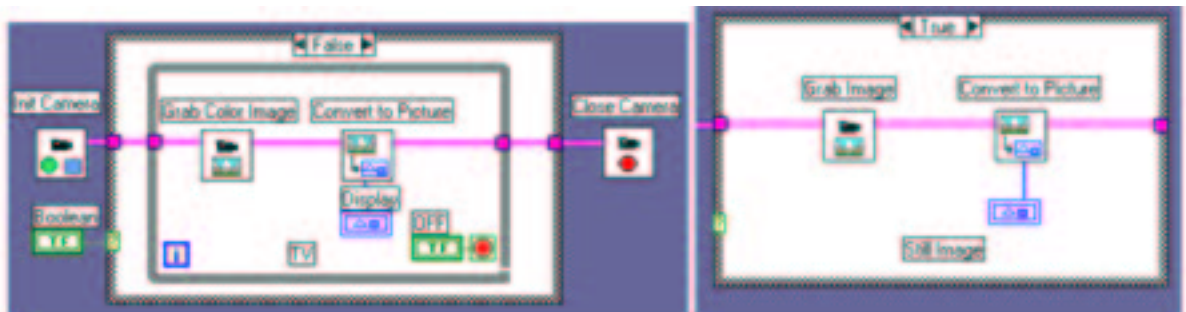
## Image Grabbing

This chapter shows you how to grab an image from a camera and how to then find certain information about that image. When you finish this section you should be able to:

1. Grab an image.
  1. [Overview](#)
  2. [Depth and resolution](#)
  3. [Color planes](#)
2. Learn basic programming concepts.
  1. [Structures](#)
  2. [Image storage](#)
  3. [For loops](#)
  4. [Graphs](#)
3. Change the appearance of the image.
  1. [LUT display](#)
  2. [LUT](#)
  3. [LUT transformation](#)
  4. [LUT histogram](#)
4. Apply arithmetic operations.
  1. [Histogram equalization](#)
  2. [Image math](#)
  3. [Subtraction](#)
  4. [Averaging](#)

Figure 6.3: Image grabbing folder of tutorial site

more knowledge when tested through Robolab. The tutorial then covers grabbing an image with the use of the Robolab icons. This tutorial assumes a working knowledge of Robolab, although it also provides hints on the location and usage of functions that the user might not have seen before. These functions are not image processing related, but they are needed in order to expand the capabilities of the vision software. One such example is While loops. In order to continuously grab an image, the user



This program allows the user to either grab continuously or just a single image.

First, you have to initialize the camera and allow the user to choose either TV or Still mode. In TV mode you continuously grab an image with a while loop, convert the image to a picture and display it. In still mode you grab an image, convert the image to a picture and display it. At the end close the camera.

**Hint:** The Case and While Loop are under the G Code / Structures palette menu. To use them you need to click and drag to place it on the diagram and then put the icons inside of it.

[Download this example](#)

Figure 6.4: Using structures in Robolab programming

The Image Grabbing section continues by showing the user what kind of information is in an image. First, it covers maximum and average pixel values. This section allows the user to see an image not as a picture, but as thousands of little pieces of information (pixels), through which the computer is able to carry out all its processing tasks. We then cover histograms, which will become very useful later on in this section, when defining look up tables (LUT), and later in the Binary Morphology chapter. We teach the user how to change the looks of an image and how to only see specific parts, by introducing her to LUTs. We end this section with image

## Log Transformation

Logarithmic look up tables are used when pixel intensity distributions are skewed to the left. This LUT would therefore make the pixel values larger, more quickly, as we move away from the origin.

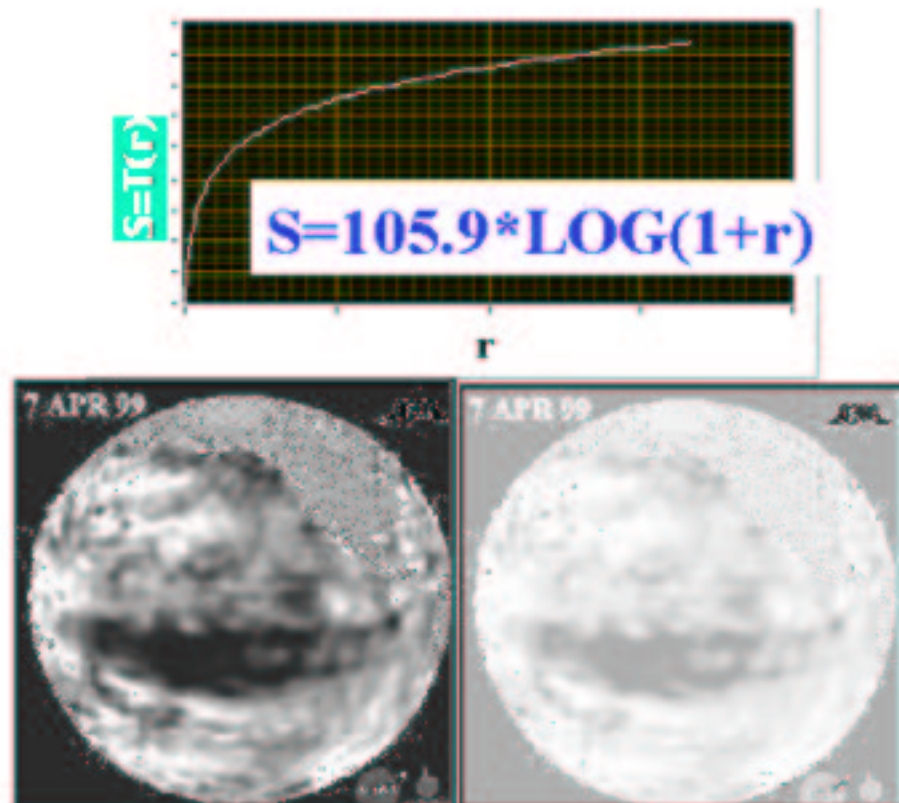


Figure 6.5: Sample concepts page

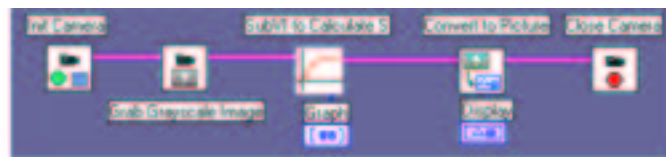
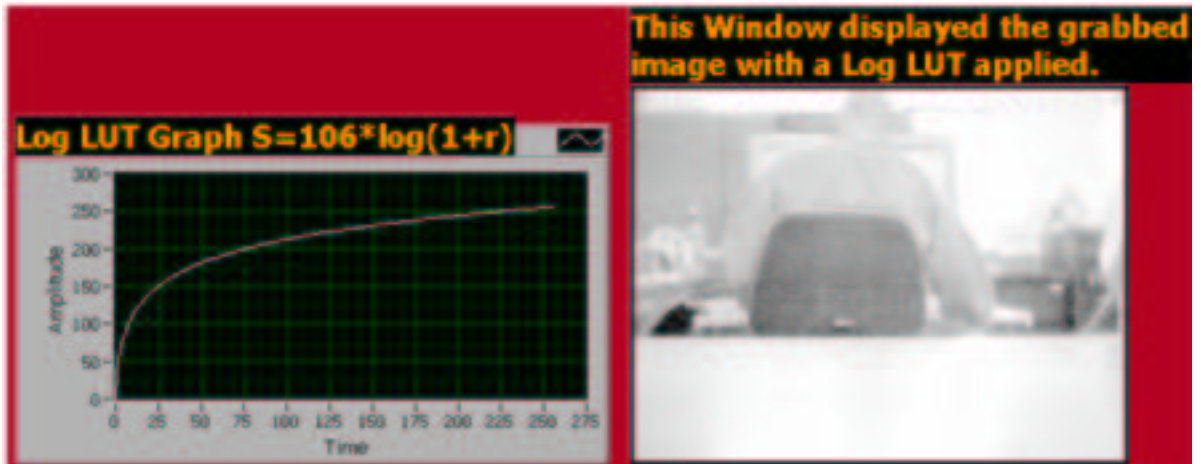
## 6.5 Binary Morphology

After teaching the user how to grab and change the appearance of an image, we introduce the topic of binary morphology. The concepts covered in this section can be seen outlined as it appears in the web site in Figure 6.8.

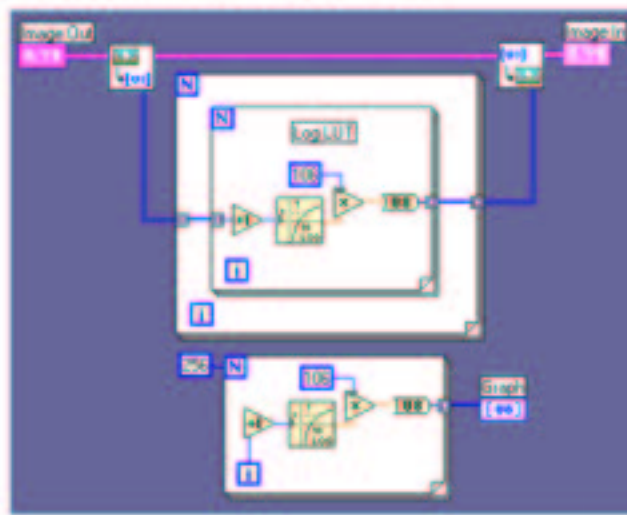
Once again, we begin this section with the Vision Center control panel. As mentioned

To complete this challenge you have to write a program that constructs a LOG LUT and graphs  $S$ ,  $S=106*\text{LOG}(1+r)$ .

Sample solution:



This program grabs a grayscale image and displays it on an intensity graph which uses a logarithmic Look Up Table for its color table. First you have to initialize the camera, grab a grayscale image and compute the Log LUT with a Sub VI. Then, display the Log function on a waveform graph, convert the image into a picture and display it. Close the camera.



This Sub VI computes the function  $S=106*\text{Log}(1+r)$ , by first converting the image into an array, isolating each pixel value, applying the above formula to each one and converting the array back into an image.

Figure 6.7: Diagram for Figure 6.6 challenge

## Binary Morphology

This chapter teaches you basic morphology operations. When you finish this section you should be able to:

1. Use Vision Center to control the RCX
  1. [RCX & Camera](#)
2. Apply a threshold to an image.
  1. [Threshold](#)
  2. [Multiple threshold](#)
3. Use basic morphology operations to process that image:
  1. [Erosion](#)
  2. [Dilation - Hit-miss](#)
  3. [Edge](#)
4. Identify and filter blobs.
  1. [Blob analysis](#)
  2. [Blob measurements](#)

Figure 6.8: Binary Morphology folder of tutorial site

earlier, Vision Center becomes a very powerful tool when we use its pre-program commands. In this challenge, the user designs a Vision Center sensor that finds the brightest blob in the image, getting introduced at the same time to concepts such as applying a threshold and blob ID. We then return to the Robolab icons, showing the user how he can program those same tasks performed by Vision Center. First we teach them how to apply a binary threshold, which is necessary in order to perform the binary morphology operations covered in the next set of challenges, such as erode, dilate, and open. All of the morphology operations described in this section can be carried out with the simple Robolab icons. We have also included challenges for higher end users who want to build their own image processing icons. One such challenge is the global threshold VI, where the user builds his own auto threshold code with the use of a histogram. A sample answer to this challenge, as shown in the web site, along

with the concepts that explain the challenge can be found in Figures 6.9 through 6.11.

## Threshold: Auto

Auto thresholding is useful when you do not want to depend on the user to enter a threshold range.

### Global Threshold

- Pick  $T$
- $G1 = \text{avg}(\text{all pts } P_{ij} < T)$
- $G2 = \text{avg}(\text{all pts } P_{ij} > T)$
- $T' = 1/2 (G1 + G2)$
- Iterate until  $|T - T'| < \epsilon$

### Local Threshold

- $P_{ij} = \text{Illumination} \times \text{reflectance}$
- If illumination varies, then segment image



Figure 6.9: Sample concepts page

We end this chapter of the tutorial with blob analysis. With the last two challenges, the user learns how to isolate objects or blobs in an image as well as how to filter them out by size and other criteria.

## 6.6 Image Restoration

The longest chapter in our tutorial covers the topic of image restoration. Apart from being a very broad topic, we devoted a large part of the image tutorial to it because,

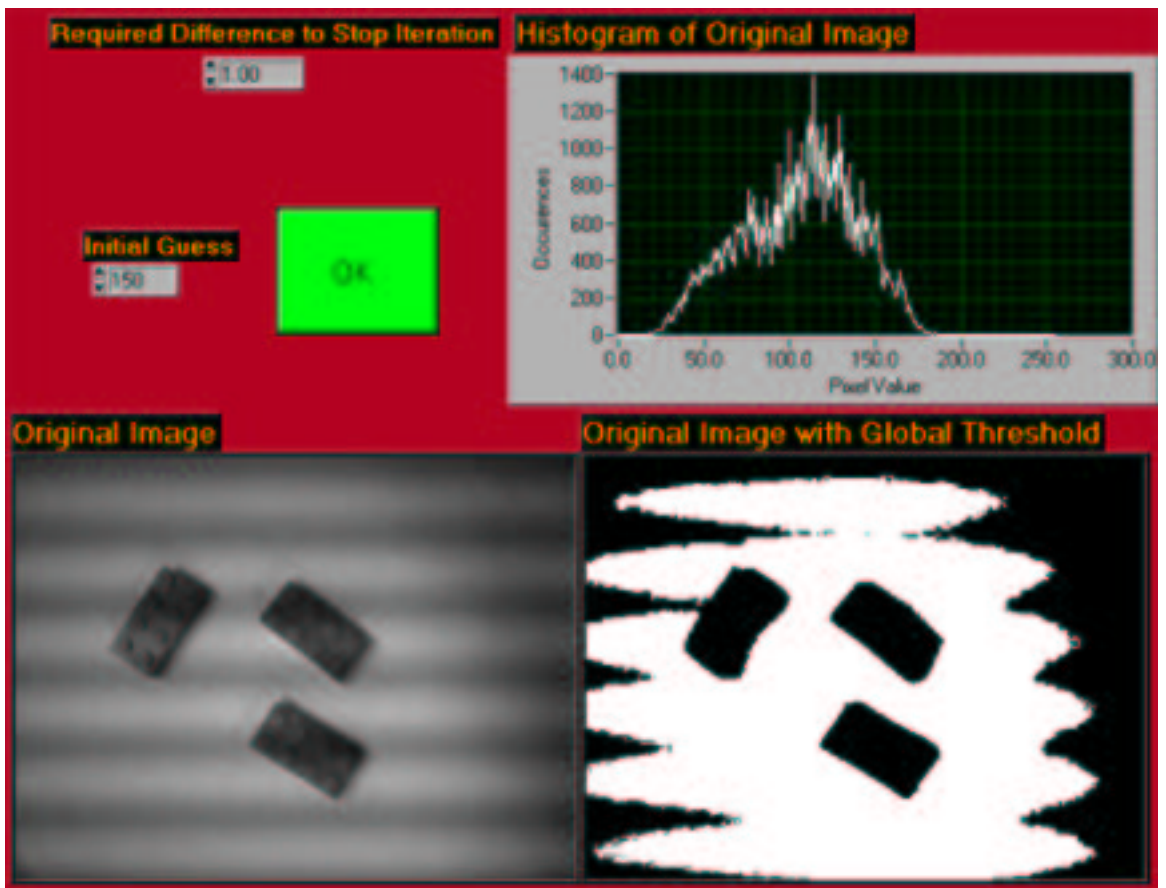


Figure 6.10: Front panel of sample answer to Global threshold challenge

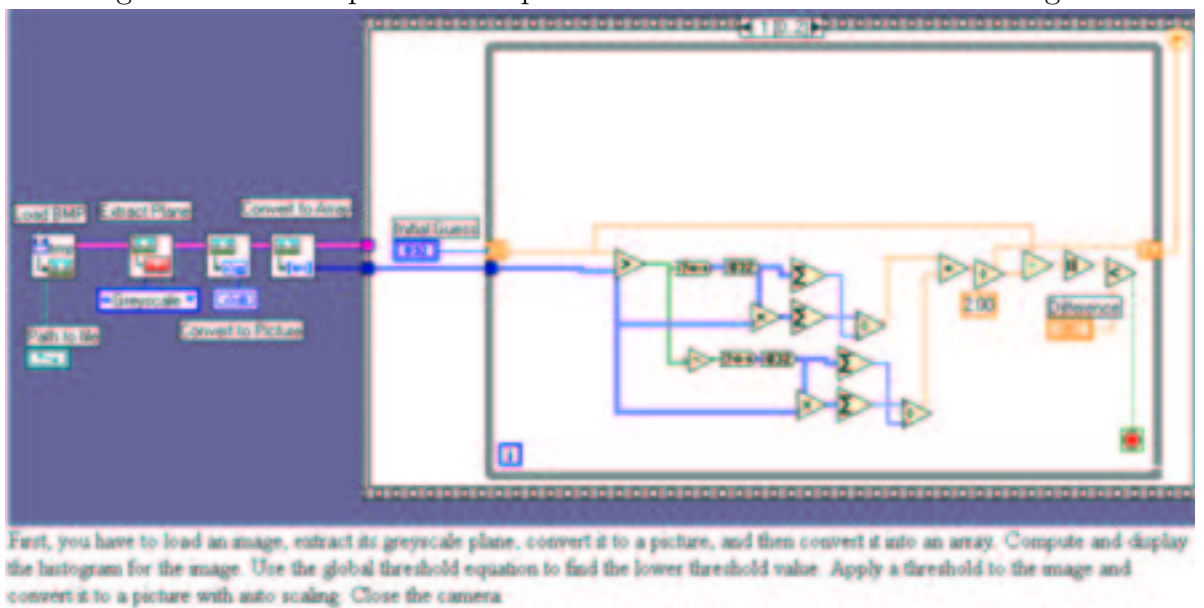


Figure 6.11: Sample diagram for global threshold challenge

of all the processing done to images, people probably spend the most time trying to make them look clearer. Images are constantly taken by the media where a stable camera support is not always available. The moving parts of machines produce a lot of noise that filters into cameras. For this reasons and many more, images have to be filtered and processed to obtain a clear image from which information can be obtained. Figure 6.12 shows the topics covered in the image restoration folder.

This section begins by introducing spatial filters and it also explains the kernels that make it work. It covers such spatial filters as smoothing, sharpening, and Laplacian. We also show the user how to implement LabVIEW IMAQ icons into a Robolab diagram. IMAQ is a more powerful image processing tool that would be of interest to the experienced user. We then cover non-linear filters, such as prewitt, sobel, and gradient, which are used for edge detection. Before introducing the user to Fourier transforms, we have a challenge that involves taking the power spectrum of a sound sample. The LEGO cameras are equipped with a microphone that can be used with a set of icons included inside the multimedia palette to capture sound. It is simpler to understand the concept of frequencies when one looks at sound than at an image, because people usually relate the word frequency to sound. After introducing FFTs and DFTs, we cover filtering images in wave space, before moving on to the correlation of images. This chapter ends with noise types and filters, before processing a few template images that we provided with a noise pattern already included. The user has to clean the image up by using the appropriate technique. Figures 6.13 and ?? show a concepts page and its related challenge. In this challenge, a mean filter is used to clean up salt and pepper noise in an image.

---

## Image Restoration

In this chapter you will learn how to apply a number of filters and techniques in order to clean-up an image. When you finish this section you should be able to:

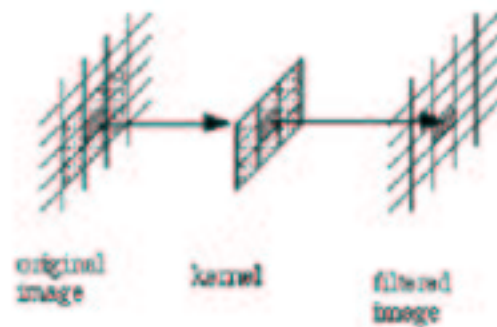
1. Apply spatial filters.
    1. [1D & 2D filters overview](#)
    2. [Smoothing / Sharpening / Laplacian](#)
    3. [Directional filters](#)
  2. Apply non-linear filters to an image.
    1. [Median filter](#)
    2. [Edge detect](#)
    3. [Nth order](#)
  3. Manipulate images in frequency space
    1. [FFT overview](#)
    2. [FFT vs. spectrum](#)
    3. [Frequency filtering](#)
  4. Filter images in wave space
    1. [DFT & notch filter](#)
    2. [Butterworth / Gaussian](#)
    3. [Kernels & padding](#)
  5. Correlate two images.
    1. [Matching objects](#)
    2. [Applications](#)
  6. Remove noise from an image.
    1. [Overview of noise types](#)
    2. [Adaptive & ordering filters](#)
  7. Restore an image.
    1. [Noise reduction](#)
    2. [Camera degradation](#)
    3. [Turbulence](#)
- 

Figure 6.12: Image Restoration folder of tutorial site

## Median Filter

- Gets rid of random noise
- Median - pixel value in the middle

$P_{i-1,j-1}$	$P_{i,j-1}$	$P_{i+1,j-1}$
$P_{i-1,j}$	$P_{i,j}$	$P_{i+1,j}$
$P_{i-1,j+1}$	$P_{i,j+1}$	$P_{i+1,j+1}$



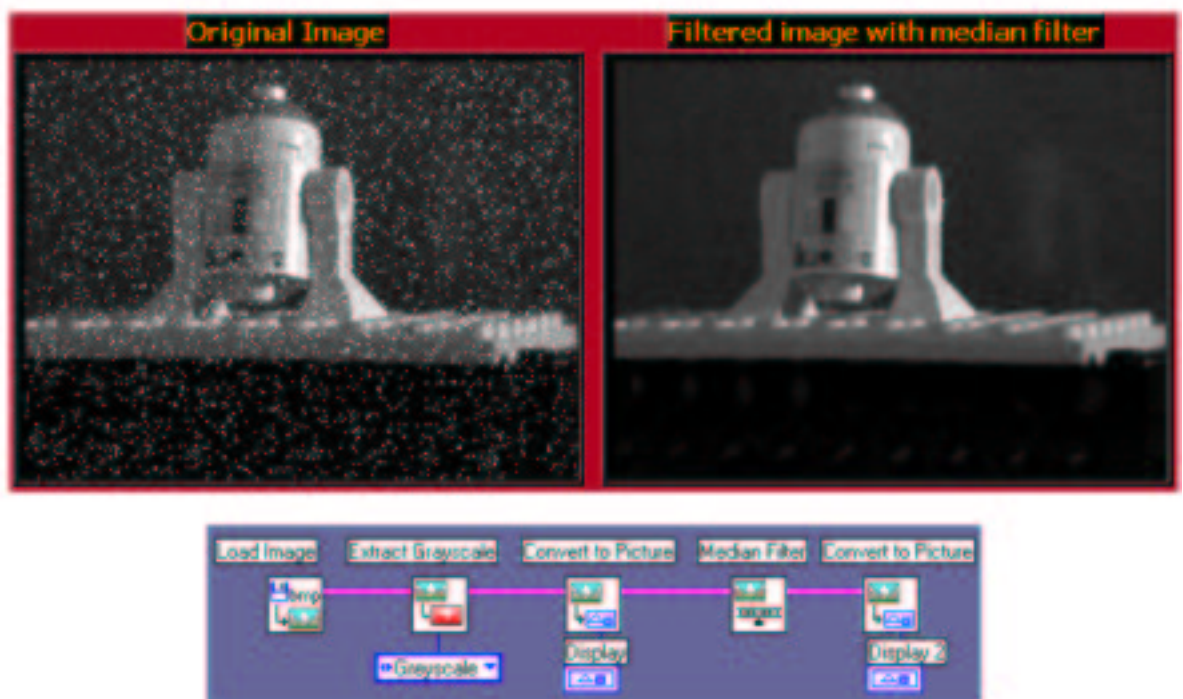
$$P_{i,j} = \text{median}[P_{i,j}, P_{i+1,j}, P_{i-1,j}, P_{i,j+1}, P_{i,j-1}, P_{i-1,j-1}, P_{i+1,j+1}, P_{i-1,j+1}, P_{i+1,j-1}]$$

Figure 6.13: Sample concepts page

## Lecture 6 - Challenge 6.1

To complete this challenge you need to build a program that reads in [R2D2.bmp](#), removes the dots with a median filter, and show both to screen.

Sample solution:



This program loads an image, displays it, applies a median filter to it, and then displays the filtered image. First, you have to load an image, extract the grayscale plane, convert the image into a picture, and display it. Then, apply a median filter to get rid of the noise and convert the image into a picture and display it.

Figure 6.14: Sample answer to challenge 6.1

## 6.7 Color Analysis

The last chapter of our tutorial covers color analysis. This chapter is relatively short, with only three challenges, but these challenges cover the topics that we think are necessary to get a basic understanding of color images. Figure 6.15 shows the concepts covered in this chapter.

### Color Analysis

This chapter explains how to process color images. It teaches you what kind of information is available in them. When you finish this section you should be able to:

1. Understand the concept of color in RGB and HSI space.
  1. [Color analysis](#)
  2. [RGB - HSI](#)
2. Color threshold an image.
  1. [Color threshold](#)
3. Apply look up tables to a color image.
  1. [Color manipulation](#)

Figure 6.15: Color Analysis folder of tutorial site

We begin this part of the tutorial by introducing the concept of color and how they are created. This includes the RGB color box and the HSI color wheel. We then challenge the user to create a color threshold program and to apply look up tables to the HSI planes. With these challenges, the user can get a good working knowledge of how each color plane works and what kind of information they carry. Figures 6.16 through 6.18 show a sample concepts page for the color threshold program mentioned above and the corresponding front panel and diagram of a solution.

## Color Thresholding or color slicing

- HSI
  - Take angle subset
  - Sometimes need subset in S and I
- RGB
  - Works better
  - Grab volume in RGB cube

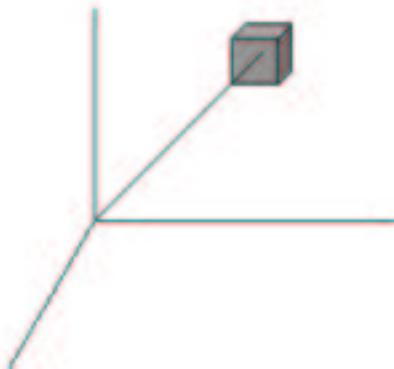


Figure 6.16: Sample concepts page



Figure 6.17: Sample front panel for color threshold challenge

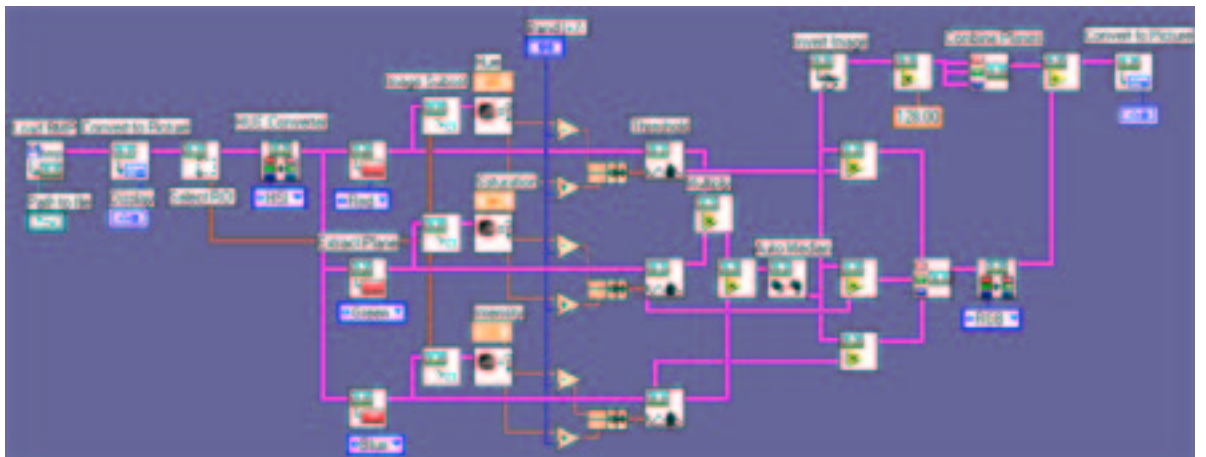


Figure 6.18: Sample diagram for color threshold challenge

# Chapter 7

## Conclusion and Future Work

### 7.1 Overview of Work Done

In this thesis work, we have developed a web site tutorial to teach students of many ages around the world the important concepts of image processing. This tutorial was based on the lectures of a college level introductory image processing course. By designing challenges around the topics covered in this course, we can make the learning process more comprehensive and hopefully also more enjoyable. Our goal at the same time was to teach students concepts in 3D vision. We therefore had to develop a set of examples that would merge the skills of the course with creating a 3D world on the computer. The methods that we choose to explain the concepts of three dimensional imaging did not always prove to be the simplest way of getting the concepts through to image processing novices.

The photogrammetry method proved to be too complex both in the set-up aspect of it, since we needed three cameras with their corresponding supports, and also in the user interface part of it, because it required a pre-image acquisition calibration. The camera set-up consisted of many moving parts, because we needed to focus all three cameras correctly on the calibration plate. This meant that the user had to perform

a large number of measurements, to identify the correct location of all three cameras, before being able to run the software. The biggest limitation of this software was the need for a pattern of point on the objects that needed to be identified. The need for points on the objects made the set-up too complex for children and the idea of drawing dots on all the objects just was not practical. The photogrammetry method did prove to be the most accurate one when identifying the correct location of objects in 3D space.

Even though the stereoscopic set-up was simpler than the three camera one, it proved to be less accurate as well as less consistent. The biggest challenge with this approach was pairing up similar objects. The two automatic methods that we tried did not consistently identify the correct set of objects. By sub dividing images and correlating them, we combined objects that were at different depths, therefore creating an average distance from the camera, not an object specific one. The correlation of individual objects proved to be inconsistent. If objects of similar colors overlapped in one of the images, the blob ID process could not identify them as two, but rather as one larger blob. We therefore decided to allow the user to select the objects herself and to let the computer correlate the image pairs and compute the distance from that information. This approach required too much user interaction and time and we wanted to design an example that did not require very much knowledge from the user in order to understand.

We then decided to experiment with a different approach to 3D imaging. Instead of having the user build the physical 3D world, then having the cameras identify their position, and lastly having the computer draw up the virtual world, we came up with the 3D plane view extrusion that requires less input from the user at a loss of accuracy of physical information from the computer. With the extrusion software,

the user can build a virtual world on the computer faster and without the need to know how the system actually works. The 3D plane view extrusion software designed for this thesis has proven to be reliable and consistent when tested under different lighting conditions. A uniform lighting is always preferred, since the shadows cast by the blocks can make the dimensions of the walls different from what the user tried to create. Glare will also affect the recognition of the pieces and should be avoided by shielding the blocks from a direct light source. Color recognition is also affected very readily by the lighting conditions. Of all the colors used, the computer has the most trouble identifying green, because its intensity is very low in all three planes, and the software might confuse it with black. Since black is removed at the beginning of the identification process, the green piece might also be removed, and it will therefore not show up when the small blocks are identified. In the future we could implement a piece identification process that would select the pieces and determine their color before anything else is removed from the image.

The 3D plane view project was useful in explaining many of the topics in the tutorial. By following the challenges in the website, the user can build the LEGO block recognition part of the software, which tells the OpenGL VIs what kinds of objects and where to build them. The student can utilize concepts such as applying a threshold and cleaning up the binary image with morphology operations. He can then use blob identification and filtering techniques to isolate the correct pieces. The major challenge of this software is designing the color recognition algorithm. The last chapter of the tutorial explains to the user the concepts of color planes and their information. Once the user is able to manipulate this information, he can identify the blocks and their meaning to build the OpenGL model.

## 7.2 Future Enhancements

In the future, we would like to incorporate other features into our software which would make it more versatile as well as more fun to use. We would like to include textures for our walls and objects to increase the amount of detail that the user can add to the building. We would also provide the user with object libraries that can be used to personalize the house. The lack of space is a problem that the 3D extrusion software has when the user wants to add a considerable amount of detail to the rooms. In the future, we want to allow the user to include a greater amount of objects inside the outline. To make the design experience more enjoyable and understanding, we would create a movie option to allow the user to navigate through the 3D model.

By applying textures to the walls and objects we can make them look more realistic. The user could either load up a file with the texture that has to be applied to the OpenGL blocks, or she could just load one from an example folder that we would provide. This is fun for children who want to create castles or practical for architects who want to show the available building options to their clients. To increase the amount of detail that the user can add to the house or building, we want to add a set of object libraries that would contain parts of the house, pre-built for OpenGL, such as sofas, tables, and other kinds of furniture that the user could place in the building (see Figures 7.1(a) and 7.1(b)).

The user would assign shapes to the different furniture blocks by clicking on the corresponding square that represents that block and then selecting the shape from a pull down menu. This would follow the same process as the nudging option in the current software.

To solve the limited amount of space problem that the user encounters when



(a) Sample 3D sofa



(b) Sample 3D chair

Figure 7.1: Furniture examples

placing objects in a room, the program can be modified so that each room is built separately. The user would either built the entire floor with the current method and then select each room individually to add more detail or she would build each room independently from the beginning and then combine them together to build the entire building.

To get the user more involved in the design, we also want to create a movie option that would allow the user to navigate through the building. Part of this concept has already been developed. The user would use a round, black LEGO block which is identified by the camera with a shape matching VI. The user would place the block in the rooms that he wants to visit, in the order that the movie should follow it (see Figure 7.2). By taking a picture of the round block in each room, the software can identify the position of the block and it can then trace a path for the camera to follow. In order to move the camera we have to change its position in 3D space, which would involve modifying the x and z position values, as well as the rotation of the camera to allow the user to look around the room. The problem that we encountered when we moved the camera is that when we come closer to a certain object, this object



Figure 7.2: Round block identified in outline

takes over the whole screen because the viewing volume does not change add more?. In order to overcome this, we have to change the position of the viewing volume by writing a new algorithm.

The path of the camera has to be mathematically computed by some sort of spline fit. If we just use a linear connection between points, the user may end up navigating through walls. We could set a limitation on the software so that the user has to place the round block in a sequence that follows a straight line navigation through the doors of the house. This limitation would in turn limit the age of the user, since the user would have to understand the straight line navigation concept. Once the user outlines the route that the camera has to follow in the movie, he can play the movie and take a virtual tour of the building.

This thesis provides basis for people interested in learning more about the subject of image processing. The web tutorial that we designed takes the user through a set of challenges, introducing him to many of the concepts of image processing. With the help of this tutorial, we will hopefully see an increased awareness in computer imaging which will allow people to learn science and mathematics through simpler image interaction.