Obstacle detection using stereo vision for unmanned ground vehicles

Martin Olsson

2009-05-06
Obstacle detection using stereo vision for unmanned ground vehicles

Examensarbete utfört i medieteknik vid Tekniska Högskolan vid Linköpings universitet

Martin Olsson

Handledare Daniel Berglund
Examinator Pierangelo Dell'Acqua

Norrköping 2009-05-06
Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kräkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Martin Olsson
Abstract

In recent years, the market for automatized surveillance and use of unmanned ground vehicles (UGVs) has increased considerably. In order for unmanned vehicles to operate autonomously, high level algorithms of artificial intelligence need to be developed and accompanied by some way to make the robots perceive and interpret the environment. The purpose of this work is to investigate methods for real-time obstacle detection using stereo vision and implement these on an existing UGV platform. To reach real-time processing speeds, the algorithms presented in this work are designed for parallel processing architectures and implemented using programmable graphics hardware. The reader will be introduced to the basics of stereo vision and given an overview of the most common real-time stereo algorithms in literature along with possible applications. A novel wide-baseline real-time depth estimation algorithm is presented. The depth estimation is used together with a simple obstacle detection algorithm, producing an occupancy map of the environment allowing for evasion of obstacles and path planning. In addition, a complete system design for autonomous navigation in multi-UGV systems is proposed.
# Table of contents

1 **Introduction** ................................................................................................................... 1
   1.1 Background ..................................................................................................................... 1
   1.2 Motivating scenario ......................................................................................................... 1
   1.3 Aims ............................................................................................................................ 2
   1.4 Method ......................................................................................................................... 2
   1.5 Outline of the report ...................................................................................................... 3

2 **The AURES Platform** ....................................................................................................... 5
   2.1 Rotundus GroundBot ..................................................................................................... 6
   2.2 Auresnet ......................................................................................................................... 6
   2.3 Simulator ....................................................................................................................... 7

3 **The Graphics Processing Unit** ........................................................................................ 8
   3.1 Rendering Pipeline ....................................................................................................... 8
      3.1.1 Vertex Processor ................................................................................................... 9
      3.1.2 Rasterization Unit .............................................................................................. 9
      3.1.3 Fragment Processor ........................................................................................... 9
   3.2 General Purpose Programming .................................................................................... 9
      3.2.1 OpenGL Shading Language ............................................................................... 10

4 **Stereo Vision** ..................................................................................................................... 12
   4.1 Overview ....................................................................................................................... 12
      4.1.1 Basic Theory ....................................................................................................... 12
      4.1.2 Stereo Algorithms ............................................................................................. 14
   4.2 Pre-processing ............................................................................................................. 15
   4.3 Matching Cost Computation ....................................................................................... 16
   4.4 Cost Aggregation ......................................................................................................... 17
      4.4.1 Windows of Fixed Size and Fixed Weights ....................................................... 18
      4.4.2 Shifted Windows ............................................................................................... 18
      4.4.3 Sum of Different Sized Windows .................................................................... 19
      4.4.4 Window with Adaptive Weights ..................................................................... 20
   4.5 Disparity Computation ............................................................................................... 20
   4.6 Post-processing ........................................................................................................... 21
   4.7 Applications................................................................................................................ 22
      4.7.1 Ground Plane Estimation ............................................................................... 22
      4.7.2 Obstacle Detection ......................................................................................... 23

5 **Proposed System** ........................................................................................................... 25
   5.1 Controller .................................................................................................................. 26
   5.2 Video ......................................................................................................................... 27
5.3 Client state ........................................................................................................................... 27

5.4 Perception Algorithm ........................................................................................................... 28
  5.4.1 Depth Estimation ................................................................................................... 28
  5.4.2 Ground Plane Estimation ...................................................................................... 33
  5.4.3 Obstacle Map ........................................................................................................ 34

5.5 Intelligence Algorithm ......................................................................................................... 35

6 Results and Analysis ..................................................................................................................... 36

6.1 Results ................................................................................................................................ 36
  6.1.1 Video Sequence 1 .................................................................................................. 36
  6.1.2 Video Sequence 2 .................................................................................................. 38
  6.1.3 Video Sequence 3 .................................................................................................. 39
  6.1.4 Performance .......................................................................................................... 40

6.2 Conclusion .................................................................................................................... 41

6.3 Discussion and Future Work ............................................................................................... 41

References ........................................................................................................................................ 43
List of figures

Figure 2.1: AURES platform configuration overview ................................................................. 5
Figure 2.2: Image of GroundBot. Image courtesy of Rotundus AB .............................................. 6
Figure 2.3: GroundBot pendulum. Image courtesy of Rotundus AB ............................................. 6
Figure 2.4: Screenshot from the AURES simulator ..................................................................... 7
Figure 3.1: Diagram of the rendering pipeline ............................................................................ 8
Figure 3.2: SIMD diagram .......................................................................................................... 9
Figure 3.3: Class and activity diagram of a shader program ..................................................... 10
Figure 4.1: The pinhole camera model ....................................................................................... 12
Figure 4.2: A basic stereo rig ..................................................................................................... 14
Figure 4.3: Tsukuba stereo set and corresponding ground truth disparity map.......................... 15
Figure 4.4: Slices of the cost volume of the Tsukuba set. Disparity level in order from top left: 19, 32, 43, 56 ........................................................ 16
Figure 4.5: Plane sweep technique with five cameras ................................................................. 17
Figure 4.6: Occlusion problem at object borders ..................................................................... 18
Figure 4.7: Four best of eight supporting windows ................................................................... 19
Figure 4.8: Pyramidal mip map kernel ..................................................................................... 19
Figure 5.1: Component diagram of the complete system ........................................................... 25
Figure 5.2: Controller class diagram ........................................................................................ 26
Figure 5.3: Example of a centralized implementation of the controller ..................................... 26
Figure 5.4: Class diagram of the video library .......................................................................... 27
Figure 5.5: UGV model class diagram ...................................................................................... 28
Figure 5.6: Distance as a function of disparity, plotted for baselines of 15, 30 and 60 cm ............ 29
Figure 5.7: Maximum detection distance plotted against image width .................................... 29
Figure 5.8: Iterative multi-resolution disparity selection ........................................................... 30
Figure 5.9: Comparison of cost aggregation (left) and colour aggregation (right) ..................... 31
Figure 5.10: Pre-processing colour aggregation ....................................................................... 31
Figure 5.11: Aggregated pixels in the R channel for mip map level \( n \) and \( n-1 \) ......................... 32
Figure 5.12: Aggregated pixels and their weighting for the four windows ................................. 32
Figure 5.13: Diagram of the proposed depth estimation algorithm ........................................... 33
Figure 5.14: Illustration of an occupancy map ........................................................................ 34
List of tables

Table 1.1: Definitions of PPS decision-points (left column)[1] and their implementation in this project (right column)......................................................................................................................................... 2

Table 6.1: Measured performance of the GPU implementation........................................................... 41
Chapter 1

1 Introduction

This report documents a master thesis project made during the fall of 2008 at Saab Aerotech in Linköping, Sweden. The thesis project is a part of my Master of Science degree in Media Technology at Linköping University, Campus Norrköping. The reader of this report is expected to have basic knowledge of image processing.

1.1 Background

In recent years the market of automatized surveillance and the use of unmanned vehicles have grown considerably. However there are scenarios where static sensors need to be complemented by more mobile platforms. Additionally, the autonomous capabilities of many unmanned vehicles are still very limited.

Since 2006 FMV (Swedish Defence Materiel Administration) and VINNOVA (The Swedish Governmental Agency for Innovation Systems) are running a research programme called TAIS (Technologies for Autonomous Intelligent Systems). The goal of the programme is to develop enhanced autonomous capabilities and artificially intelligent behaviour for unmanned vehicles. As a part of the TAIS programme, Saab Aerotech together with FOI (Swedish Defence Research Agency) and KTH (Royal Institute of Technology) joined the project group AURES (Autonomous UGV-systems for Reconnaissance and Surveillance). The AURES project group will develop, demonstrate and evaluate autonomous functionality for unmanned ground vehicles used for surveillance and reconnaissance with focus on patrolling, positioning and search-and-secure algorithms.

Saab Aerotech provides the hardware platform that will be used in the AURES project, the Rotundus GroundBot\(^1\), a spherical robot equipped with two cameras. As the focus of AURES lies primarily on the three high level algorithms mentioned above, Saab Aerotech was interested in someone to investigate and add more fundamental or additional autonomous functionality to the robot. Because of my interest in image processing and the capabilities of the robot, one of them being equipped with dual cameras, we decided to focus on computer vision and obstacle avoidance.

1.2 Motivating scenario

A harbour area full of containers and crates is a large and shifting environment with many objects blocking the line of sight. Static surveillance cameras are thereby

\(^1\) See [http://www.rotundus.se](http://www.rotundus.se)
rendered useless. To increase efficiency and reduce costs, the patrolling guards are replaced by a number of small mobile robots. With the position of the cargo containers known, the optimal patrolling routes are calculated by the AURES algorithms. A computer vision system allows the robots to detect and evade any dynamic obstacles such as cars, misplaced crates or humans. New obstacles may be reported to the patrolling algorithm and encounters with possible intruders can be reported to the operator.

1.3 Aims

The purpose of this work is to investigate methods for real time computer vision and obstacle detection for unmanned ground vehicles. This report aim to answer the following questions:

- What vision based obstacle detection methods are suitable for UGVs?
- What hardware is required to achieve real time processing speeds?
- How can we integrate these methods into the AURES demonstration platform?

To answer these questions, a literature study will be performed initially. Based on the literature study, a depth estimation and obstacle detection algorithm will be developed and implemented. To allow further developed, the system will be integrated into the AURES demonstration platform.

1.4 Method

Saab Aerotech exercise a project management model called Practical Project Steering (PPS) which was also applied to this thesis project. The base idea of PPS is that a project is managed through a set of decision-points (DP). Additionally a project consists of several processes, outer dependencies, milestones and deliveries. Table 1.1 describes the definition of each decision-point according to PPS Online [1] and their implementations in this project.

| DP1 | “Preparations based on the project directive start; people’s responsibilities and the budget are agreed up to DP2 and/or DP3.” | Initial study of the AURES project and its control documents. Problem description is formulated. |
| DP2 | “Preparations continue or are interrupted. Possible new conditions” and revised budget. | |
| DP3 | “Preparations completed. Project plan, requirement description and solution description frozen.” | An initial time plan is constructed. |
| DP4p | “Part of execution starts (before literature study on computer vision) | |

Table 1.1: Definitions of PPS decision-points (left column)[1] and their implementation in this project (right column).
1.5 Outline of the report

To give the reader an overview of the report, below follows a short summary of each chapter as well as motivation of the disposition:

Chapter 1: Introduction

The first chapter presents the background of this project and specifies the aims. The choice of method is also briefly discussed.
Chapter 2: The AURES Platform

The work carried out in this project is intended to be integrated into the AURES project. Hence, both the hardware (robot) and software of this platform are described in the second chapter.

Chapter 3: The graphics processing unit

Since the image processing algorithms must be performed in real-time, they are often implemented in a parallel computing architecture. I have chosen to implement the algorithm on the graphics processing unit (GPU) of the graphics card. The architecture of the GPU along with how it can be used for general purpose programming such as image processing is discussed in the third chapter.

Chapter 4: Stereo vision

The robot used in the AURES project is equipped with two cameras, enabling stereo vision. In the fourth chapter, the basics of stereo vision is described as well as how it can be used to detect obstacles. This chapter also gives a brief description of the methods used in real-time depth estimation.

Chapter 5: Proposed system

The fifth chapter describes the proposed system for obstacle detection as well as a novel wide-baseline depth estimation algorithm.

Chapter 6: Results and analysis

Finally, the last chapter presents performance results for the proposed system. How well the goals of the project are fulfilled and possible future work is also discussed.
Chapter 2

2 The AURES Platform

The following chapter gives a brief presentation of the architecture of the UGV system used in the AURES project. This is also the platform which the implementations in this project are aimed for. The requirement specification [2] lists the following main components:

- Physical UGVs.
- A simulator for simulation of one or several UGVs.
- User Interface, UI, mainly for presentation, displays UGVs in a map.
- Operator Control, a device to manually control one UGV at a time.
- Algorithms (ALG), the intelligence in the multi-UGV system that produces high-level commands.

The interactions between these components are shown in Figure 2.1 below.

Figure 2.1: AURES platform configuration overview.
2.1 Rotundus GroundBot

The physical UGVs that will be used are the GroundBot™ (Figure 2.2), developed by Rotundus AB. The interior of the shell consists of a pendulum and by moving that in either direction the centre of mass is shifted and the sphere start rolling (Figure 2.3). The sphere has a diameter of 60 cm and because of the large circumference the robot has no problem travelling on uneven ground. The two gyro stabilized camera gimbals mounted on the main horizontal axis provides a solid stereo rig when navigating, as well as the possibility of 360° field of view when on sentry duty. Live video from both cameras are sent as MPEG-4 compressed video streams via RTP (Real-Time Protocol).

![Figure 2.2: Image of GroundBot. Image courtesy of Rotundus AB.](image1)

![Figure 2.3: GroundBot pendulum. Image courtesy of Rotundus AB.](image2)

2.2 Auresnet

The communication interface between the different components is called Auresnet. It is based on the NATO STANAG (Standard Agreement) 4586, an interoperability standard for Unmanned Aerial Vehicles (UAVs). Although STANAG 4586 is intended for UAVs, most of it can also be applied to UGVs. It should be noted that Auresnet is intended as a communication interface for UGV systems only. STANAG 4586 sets out five levels of interoperability [3]:

- Transfer of filtered UAV/UGV data to third party.
- Direct transfer of live data via a ground station to a remote command system.
- Control the onboard systems by commanders in the command system.
- In-flight control by the command system.
- Full flight control by the command system, including take-off and landing.

The above (except take-off and landing obviously as UGVs operate on the ground) is solved by using DLI-messages (Data Link Interface) which are sent over WLAN (Wireless Local Area Network). The messages can be either status information which is broadcasted by the UGVs, or commands which are sent by the operator control or autonomous algorithm to one or several UGVs. There are three ways to control the UGV’s movements through Auresnet:
• Waypoint following - the UGV is following a waypoint trajectory.  
• Direct feedback control - the UGV is following commanded speed and direction.  
• Short Stepwise Translation – the UGV moves (or turns) commanded distance.

2.3 Simulator

In order to be able to perform tests without a physical UGV or simulations with more units than what actually are present, a simulator has been developed. The simulator uses the Auresnet communication interface and the robots can be commanded and behaves identical to their physical counterparts. Additionally, telemetric data from a mission/scenario can be saved and later replayed in the simulator. A screenshot from the simulator is given in Figure 2.4. Most of the data used for image processing in this project are recorded using the simulator.

Figure 2.4: Screenshot from the AURES simulator.

---

1 A waypoint trajectory is a path described by a set of coordinate points (waypoints).
Chapter 3

The Graphics Processing Unit

A graphics processing unit (GPU) is a device dedicated to perform operations regarding graphics rendering, such as floating point vector and matrix operations. It implements several hardware optimized mathematical- and graphics primitive related operations. The GPU may be mounted on a graphics card or integrated into the motherboard. Currently there exists two major manufacturers of GPUs; NVIDIA\(^1\) and AMD Graphics Products Group (former ATI)\(^2\). The following chapter will briefly describe the architecture of a GPU and how it can be used for general purpose programming (In this case image processing rather than just producing 3D-graphics).

3.1 Rendering Pipeline

In short the rendering procedure, called the rendering pipeline, can be divided into three steps which are all run in parallel. Each step takes a data stream as input, processes it and sends it to the next step. The procedure is initialized by the application setting up any textures to be used and sending vectors to the GPU. The vectors may represent for example vertices\(^3\), normals, colour or texture coordinates. The resulting image is then stored in a framebuffer in the graphics card memory. A diagram of the rendering pipeline is displayed in Figure 3.1. Because of the parallelism, optimised hardware, fast busses and high speed memory, the GPU gives a tremendous speed increase as opposed to doing the same things on the CPU.

![Diagram of the rendering pipeline.](image)

---

\(^1\) http://www.nvidia.com

\(^2\) http://www.amd.com

\(^3\) A vertex is a point in space. For example, three vertices are used to form a triangle.
3.1.1 Vertex Processor

The main function of the vertex processor is to transform the incoming vertices. They are first transformed from model space to view space meaning, a transformation from world coordinates to the camera coordinate system. Next a perspective projection is performed by projecting the vertices onto the image plane. After the transformations the vertices are sent to the primitive assembly where they are grouped as graphical primitives. There are three types of primitives; triangles, lines and points. The primitives may then be culled (discarded) due to facing away from camera, or be clipped due to being outside the cameras field of view. In addition to the transformations, some lighting calculations are also done in the vertex processor.

Modern GPUs have programmable vertex processors allowing the user to manually transform the vertex position and other attributes. Only the latest GPUs, implementing Shader Model 3.0, allow sampling of textures in the vertex processor.

3.1.2 Rasterization Unit

The stream of graphical primitives is sent from the vertex processor to the rasterization unit. During rasterization the primitives are divided into fragments. Each fragment represents one pixel in the final image. The attributes from the vertices, such as colours and normals, are interpolated for each fragment. At the time of writing the rasterization unit is not programmable.

3.1.3 Fragment Processor

The purpose of the fragment processor is to colour each fragment based on material properties, lighting and texturing. The fragment processor may only operate on one fragment at a time, but may output several different colours if multiple rendering targets are used. Fragments may also be discarded for different reasons, for example being occluded by another fragment at the same position with a lower depth value. Similar to the vertex processor, the fragment processor is fully programmable.

3.2 General Purpose Programming

With the introduction of programmable vertex and fragment processors, programmers are able to utilize the computing power of GPUs for other tasks than graphics rendering. This technique is often referred to as GPGPU (General Purpose computing on Graphic Processing Units). Because of the parallel architecture of the rendering pipeline, GPGPU offers the possibility to increase the computing speed of many algorithms, but there are also many limitations. As each vertex and fragment is processed independently, GPGPU works best for stream-processing and SIMD (Single Instruction Multiple Data) techniques. In stream-processing, parallelism is achieved by applying a series of operations (kernels) to each element in a data stream. In SIMD, multiple...
processing units share the same instruction pool and may thereby operate on several elements in a data stream simultaneously, see Figure 3.2. The GPU uses SIMD for the built in vector and matrix functions.

To alter the functionality of the vertex and fragment processors, small computer programs called shaders are uploaded and executed on the GPU. The shaders were initially written in assembly language but several high level programming languages have been developed to increase the ease of use and portability. Each shader program can be considered as a kernel and each vertex and fragment is processed independently and simultaneously in SIMD manner. To create an algorithm using several kernels the programmer can render to an off-screen buffer and use that as an input texture to the next pass using a different shader. In GPGPU most of the work is usually done in the fragment shader operating on a 2D grid of fragments. This is done by rendering a single rectangle covering the entire screen and by using one or several textures as input data. This applies to most GPGPU applications such as matrix operations, image processing and physical simulations. The input data to the shaders, and dataflow between the vertex and fragment shaders, are in floating point format, or converted to floating points if the input is an integer texture. A fragment shader may output images with a depth of one, two, three or four colour components using integer or floating point format.

3.2.1 OpenGL Shading Language

The OpenGL Shading Language (GLSL) is a high-level C-like programming language defined by Kessenich et al. [4]. It is used to modify the fixed functionality of the vertex and fragment pipelines through shader programs. As hinted in the name, GLSL does not support stand-alone applications but must be used through the OpenGL API (Application Programming Interface). The functionality to use shader programs are defined in three OpenGL extensions [5],[6][7] and are part of the OpenGL Specification 2.0 [8]. The programs are uploaded to the GPU through the OpenGL entry points as text strings and are compiled and linked using drivers provided by the GPU vendor. A class and activity diagram of a shader program is given in Figure 3.3. GLSL contains all the standard C operators except bit operations. It also has implementations of many common mathematical functions and allows looping and branching.

![Figure 3.3: Class and activity diagram of a shader program.](image)

In addition to GLSL, which only can be used with OpenGL, there exist two other major high-level shading languages; HLSL (High Level Shading Language) which can be used only with Direct3D, and Cg (C for graphics) which can be used with both
OpenGL and Direct3D. Furthermore, both NVIDIA and AMD have each developed technologies specifically aimed towards GPGPU and stream-processing, thus enabling even more hardware functionality and increased performance over the other shader programs which are initially intended for graphics rendering. These technologies are CUDA\(^1\) by NVIDIA and ATI Stream SDK\(^2\) by AMD. The drawback of these two technologies is that they are only compatible with the respective vendors’ GPUs. This drawback will however be nullified in the future as both NVIDIA [9] and AMD [10] have announced that their technologies will fully support OpenCL (Open Computing Language)\(^3\). OpenCL is a framework for general purpose parallel programming on heterogeneous platforms such as CPUs, GPUs and DSPs (Digital Signal Processors) and the API specification of version 1.0 was released in December of 2008. The OpenCL language is based on writing kernels, which is similar to the concept of GPGPU utilization of shader programs.

I have chosen to work with OpenGL and GLSL in this project because of the portability of OpenGL and personal preferences of GLSL over Cg. OpenCL will definitely be a strong candidate in all stream-processing applications in the future.

---

Chapter 4

4 Stereo Vision

The robot used in the AURES project is equipped with two cameras allowing stereo vision. An effective stereo algorithm would give the robot the ability of depth perception and make it able to detect obstacles. Stereo vision is the technique of using two or more images taken from different positions to estimate the distance to an object. Humans do this continuously and effortlessly in the vision system with the images from the left and right eyes. Simulating this in a computer is a considerably more difficult task. However, stereo vision or depth estimation is an ongoing research topic in computer vision and a huge number of stereo algorithms have been developed over the past decade. There is a large variety of algorithms in terms of performance. Some are complex and gives very exact depth estimates whereas others are simpler and aims towards applications where processing time is preferred over quality. This chapter will give an overview of the general methods used in stereo vision algorithms as well as discuss what qualities of the algorithms would be preferred in this project.

4.1 Overview

4.1.1 Basic Theory

Depth estimation in computer vision is about solving the so called correspondence problem: “For a given point in image A, find the corresponding point in image B”. With the problem formulation applied to the general stereo case of two cameras, A and B represents the images from the left and right camera respectively. The distance from the camera to the object is related to the disparity between the corresponding points in image A and B. The disparity can be described as the change in location on the image plane for these points. The most common approach is to use two cameras which are vertically aligned. The corresponding points will then lie on the same scan line and the distance to the object is calculated from their horizontal disparity.

For the transformation between the scene coordinates and the image coordinates, the pinhole camera model is often used, also known as perspective projection. The pinhole camera model has proven to be an accurate enough approximation of real world cameras.

Figure 4.1: The pinhole camera model.
when used in most applications. It can be seen as a box of length $f$ (from now on referred to as focal length) with a tiny hole, see Figure 4.1. All light in the photographed scene passes through this hole called centre of projection, and is projected onto the back of the box. This results in a horizontally and vertically flipped image of the scene. Consider the scene coordinate system $(X \ Y \ Z)$ and the image coordinate system $(u \ v)$ with the centre of projection in origin and $f$ along the negative Z-axis. In the synthetic camera model we can place a virtual image plane at distance $f$ on the positive Z-axis to avoid the inversion of the image. This gives the following relations:

$$\frac{f}{Z} = \frac{u}{X} = \frac{v}{Y}$$

and thereby $u = \frac{fX}{Z}, v = \frac{fY}{Z}$ (4.1)

The perspective projection of a 3D scene point to 2D image coordinates can also be written as a matrix operation using homogenous coordinates$^1$:

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} P$$ (4.2)

In this equation, $(u \ v \ w)$ are the homogenous image coordinates where their 2D counterparts, $(x, y)$, is described by \( x = u/w \) and \( y = v/w \). $P$ is a $3 \times 4$ matrix containing the intrinsic and extrinsic camera parameters. The extrinsic parameters specify the camera orientation while the intrinsic specify the camera’s focal length and aspect ratio. Another camera parameter which is often mentioned in perspective projection is the field of view (FoV) which is related to the focal length and the width and height of the film or image.

**Horizontal field of view**

$$\text{Horizontal field of view} = 2 \tan^{-1} \left( \frac{\text{width}}{2f} \right)$$ (4.3 a)

**Vertical field of view**

$$\text{Vertical field of view} = 2 \tan^{-1} \left( \frac{\text{height}}{2f} \right)$$ (4.3 b)

If width and height are given in pixels the focal length must be given in pixels as well. The focal length in pixels can be obtained by multiplying the focal length in meters by the sensor’s pixel density.

Figure 4.2 shows two identical cameras with focal length $f$, which are positioned at $C_L$ and $C_R$ and are vertically aligned at $Y = 0$. The cameras are separated along the $X$-axis by the baseline distance $b$. The object point, located at $(0 \ 0 \ Z)^T$, is projected at the two locations $P_L$ and $P_R$ in the left and right image respectively. The points are related by their horizontal disparity $d$ where $P_R = P_L + d$. Basic trigonometry yields the following distance relations:

---

$^1$ Homogenous coordinates is used to allow affine transformations to be represented by a matrix.
The following relation between baseline, focal length, distance to object and disparity can be derived from equations 4.4:

\[ d = \frac{fb}{Z} \iff Z = \frac{fb}{d} \]  

(4.5)

4.1.2 Stereo Algorithms

Scharstein and Szeliski [11] have made an evaluation of the most common stereo algorithms where they aim to assess the different components and design decisions made in different stereo algorithms. They present a taxonomy for stereo algorithms where the algorithms generally can be divided into three steps:

1. Matching cost computation
2. Cost aggregation
3. Disparity computation

The term cost is used for the similarity measure of two pixels from the left and right images. In addition to these steps, many algorithms also perform some form of pre- and post-processing. The stereo algorithms can also be divided into three categories; local, global and scan line-based. Local algorithms operate on each pixel individually, trying to find the best disparity match based on a small area around the target pixel. The global algorithms on the other hand aim to minimize the global energy (cost) and in that way find the correct disparity for each pixel. A 2D-optimizing problem of that kind is shown to be NP-hard\(^1\). To overcome this there are scan line\(^1\)-based algorithms

 aggressively solves a similar energy-minimizing problem for each scan line individually. Common for all algorithms is that they either implicitly or explicitly make assumptions that all surfaces in the images are Lambertian reflectors\(^2\) which are piecewise smooth. This assumption means that the reflected intensity of a point is the same for all viewing directions. Additionally, it means that all surfaces can be described by a plane. This assumption is clearly false but has proven to work well enough. The output of a stereo algorithm is essentially a disparity map where the pixels of the input image have been assigned a disparity value that is related to the distance from the camera according to equation 4.5. Figure 4.3 shows the famous Tsukuba stereo pair used in [11] (top) and its corresponding ground truth disparity map (bottom). High intensities represent pixels located close to the camera.

The next five sections of this chapter will describe the three steps of stereo algorithms as well as methods for pre- and post-processing.

Figure 4.3: Tsukuba stereo set and corresponding ground truth disparity map. Image courtesy of [11].

4.2 Pre-processing

As mentioned earlier the problem consists of finding the corresponding points in two or more images. An assumption is made that the points have the exact same intensities, which is obviously not the case when dealing with reflective surfaces. In addition the cameras may not have the exact same sensitivity function resulting in the

---

\(^1\) A scan line is a horizontal row of pixels in an image.

\(^2\) A Lambertian reflector reflects light independently of viewing direction, i.e. it causes no reflections or specular highlights.
two points having different intensities. To counter this, the images can be filtered with a LoG-filter (Laplacian of Gaussian) or be normalized by subtraction of a median filtered version of themselves, thus only keeping the relative intensity differences [12].

If the algorithm assumes the corresponding points to be lying on the same scan line, the cameras need to be very carefully calibrated. If this is not the case, the images need to be rectified. The rectification processes consists of a linear transformation where the parameters can be calculated at initialization. Hartley presents a method for image rectification in [13].

4.3 Matching Cost Computation

The disparity is estimated in a discrete number of steps called depth hypotheses. The matching cost is computed using one of two methods; either using rectified images or by an image projection technique called plane-sweep. In both cases, the cost, or similarity, is measured pixel per pixel by the intensity differences between the left and right images for each depth hypothesis. This results in a cost volume $C(x, y, d)$, where $x$ and $y$ are the pixel coordinates and $d$ the depth hypothesis index.

In the case of rectified images one of the images is chosen as the reference frame. The resulting disparity map will have the same point of view as this image. The other image is displaced along the horizontal axis a number of pixels equal to the disparity level. The cost for a pixel at a given disparity is:

$$C(x, y, d) = L(x, y) - R(x - d, y)$$

(4.6)

Figure 4.4 illustrates slices of the cost volume from the Tsukuba stereo set at disparities, or depth hypothesis index, of 19, 32, 43 and 56. Dark values indicate high similarity.

---

**Figure 4.4:** Slices of the cost volume of the Tsukuba set. Disparity level in order from top left; 19, 32, 43, 56.
In Figure 4.4, one can notice how the background pixels all have high similarity at disparity level 19 (top left), while the pixels belonging to the lamp have high similarity at disparity level 56 (bottom right). It is also notable how the table, which is quite poorly textured, seems to have good matches at several disparities.

An alternative method of cost computation is the plane-sweep technique used in [14] and [15]. The images are projected onto a common plane for each depth hypothesis. If the plane is located at the same depth as the recorded scene, pixels from all images should have the same intensities. In the standard case of two fronto-parallel\(^1\) cameras, the planes should be placed at a distance according to equation (4.5); \(Z = fb/d\). The cost is calculated as the pixel intensity difference of the projected images:

\[
C(x, y, d) = L'_d (x, y) - R'_d (x, y)
\]  

(4.7)

The advantage of the plane-sweep technique is that images do not need to be rectified since that is handled by the projection. Additionally, projective texture mapping is a built-in functionality in most 3D graphics libraries, thus when using a GPU to accelerate the stereo algorithm, the image projection step can be performed automatically in hardware. Figure 4.5 illustrates a plane-sweep approach with a setup of five cameras. The red dot represents the resulting view point.

To avoid negative cost values, the cost volume is normalized using either absolute (AD) or square difference (SD) values.

\[
C_{AD}(x, y, d) = |C(x, y, d)|
\]

(4.8 a)

\[
C_{SD}(x, y, d) = C(x, y, d)^2
\]

(4.8 b)

4.4 Cost Aggregation

In section 4.3, a cost or similarity measure has been calculated for every pixel at each depth hypothesis. This point sampling is very sensitive to sensor noise and in addition a single pixel may have several good matches, especially in low-textured areas or in areas with repetitive texture. To make the matching more robust, the cost is usually aggregated by summing or averaging the cost values over a support area around each pixel. This is based on the assumption that most pixels in a small area will belong to the same object and therefore have similar depth value. This statement can

---

\(^1\) Cameras aligned with parallel optical axis.
be considered true everywhere except at object boundaries. In the general case, this can be explained as that each depth hypothesis slice in the cost volume, \( C_d \), is convolved with a window of weights, \( W \), according to:

\[
C'_{d}(x, y) = (C_{d} \ast W)(x, y) \quad \text{(4.9)}
\]

\[
Rightarrow C'_{d}(x, y) = \sum_{i=-\text{width}/2}^{\text{width}/2} \sum_{j=-\text{height}/2}^{\text{height}/2} C_d(x + i, y + j) \cdot W(\text{width}/2 + i, \text{height}/2 + j)
\]

where \text{width} and \text{height} are the dimensions of the window.

Care must be taken when choosing the size and weights of the window. A large window will provide an overall robust matching but will perform badly at object boundaries and completely remove thin objects. A small window on the other hand will provide a detailed depth map but is subject to noise and ambiguous matches. Figure 4.6 shows the problem of occlusion at object borders, the large window \( X \) will not find any good match for background object \( B \) in the right image while a smaller window \( Y \) would correctly make a match.

Based on the convolution window approach, many methods for cost aggregation have been developed. Below follows a list of some of them:

**4.4.1 Windows of Fixed Size and Fixed Weights**

The most basic cost aggregation is just an averaging window and is also the one most commonly used, for example in the algorithm presented by Di Stefano et al. [12]. The convolution with a window with homogenous weights can be optimized and computed very efficiently on the CPU. By reusing values from previous calculated image rows or columns as shown in [12], the algorithm becomes independent of the window size. This optimization can however not be done in a GPU because of its parallel architecture. On the other hand, a GPU supports hardware functionality for bilinear texture sampling. Sampling a texture in the middle of four pixels returns an average of those four pixels’ intensities, meaning effectively a 2 x 2 averaging window. A 2 x 2 window is unfortunately still too small to be robust at many times but the functionality can still be exploited.

**4.4.2 Shifted Windows**

As mentioned above and illustrated in Figure 4.6, large windows introduces matching errors at object borders. This may result in smeared boundaries and halo effects.
Hirchmüller et al. [16] as well as Woetzel and Koch [17] presents an assumption which states that at least some part of a large window should be able to correctly match an non occluded object. This is the case in Figure 4.6 where Y, which is a part of the larger window X, correctly matches the partly occluded object B.

Hirchmüller et al. [16] implements this by in addition to using a small window centered on the pixel of interest, calculates the cost of a number of neighbouring small windows. Out of the neighbouring windows, the ones with the best score are added to the centre window. They propose configurations of 4, 8 and 24 supporting windows where the best, 2, 4 and 12 respectively, are added to the cost. Figure 4.7 shows a setup of 8 supporting windows around a blue centre window. The four windows with the best costs are used (marked in red). Selecting the \( n \) best matches requires a sorting algorithm which can be quite costly, especially for the configurations with 8 and 24 supporting windows.

Woetzel and Koch [17] present a similar approach where they are utilizing the GPU functionality of bilinear texture sampling. By sampling the texture at 1/2 pixel offset in the four diagonal directions, they are using four 2 x 2 windows overlapping a 3 x 3 area. Instead of using a centre window with a number of supporting windows, they are just choosing the best one out of the four candidates. The small support area is sufficient for their algorithm as they are using a multi-camera setup of four cameras. The small area would probably not provide enough support in a standard two camera configuration.

**4.4.3 Sum of Different Sized Windows**

Yang and Pollefeys [14] presents another method of countering the window size problem. By studying cost curves \( C(d) \) for different image points, they observe that large windows usually have one strong cost minimum in the neighbourhood of the true depth while smaller windows often have multiple minima where one of them is very well localized. Summing the cost curves for different sized windows usually results in a curve with a strong minimum well localized at the true depth.

This is implemented by a *mip mapping* technique. In the mip mapping process, images are iteratively scaled down by a factor of two. This means a mip map of level \( i \) will have a width and height equal to \( 1/2^i \) of their original value. Consequently, a pixel in a mip map of level \( i \) has the intensity of the average of it’s corresponding four pixels in mip map level \( i-1 \). This result in that a mip map of level 1 equals a 2 x 2 averaging window, at level 2 it becomes a 4 x 4 window, and so on. Mip map generation is a part of most GPU’s built in functionality and can be calculated very fast. Additionally, this method only introduces a 33% increase in computing time from matching cost and cost aggregation compared to just using a 1 x 1 window, this because the total...
number of pixels of the mip maps is related to the original width, \(W\), and height, \(H\), according to:

\[
\sum_{i=0}^{n} \frac{W \cdot H}{2^i} \approx 0.33 \cdot W \cdot H \quad n \to \infty
\]

By summing the cost of different mip map levels, this method can be illustrated as using a large window with pyramidal shaped weights where pixels closer to the centre have greater influence. Figure 4.8 shows an illustration of a three level mip map kernel where the height represents the weight. It can also be seen as an approximation of a Gaussian kernel convolved with the original sized cost slice.

### 4.4.4 Window with Adaptive Weights

As mentioned above, cost aggregation is based on the assumption that neighbouring pixels belong to the same surface and consequently should have the same depth. This is not true at object boundaries and methods to overcome this, such as the use of multiple small windows, have been discussed. Wang et al [18] extends the aggregation assumption by saying that that neighbouring pixels that has a similar colour is more likely to belong to the same surface than pixels with a different colour. This is implemented as a window with adaptive weights based on the colour difference and the Euclidean distance\(^1\) on the image plane to the current pixel. “Given a pixel \(p\), and a pixel \(l\) in its support region, the matching cost from \(l\) is weighted by the colour difference \(\Delta c_{pl}\) between \(p\) and \(l\), and the Euclidean distance \(\Delta g_{pl}\) between \(p\) and \(l\) on the image plane.” [18] The weight, \(w\), is calculated according to:

\[
w(p,l) = e^{\left(\frac{\Delta c_{pl}}{\gamma_c} + \frac{\Delta g_{pl}}{\gamma_g}\right)},\quad (4.10)
\]

where \(\gamma_c\) and \(\gamma_g\) are constants. The adaptive weights need to be calculated independently for all pixels, thus making this aggregation very computationally heavy. To speed up the aggregation, Wang et al proposes the use of a \(n \times 1\) sized vertical window. This works well when using scan line-based methods for the disparity computation step. For local WTA algorithms (more on disparity computation methods in section 4.5), they propose a way to approximate a square window by using a two-pass aggregation. The cost is first aggregated in the vertical direction by a \(n \times 1\) window followed by a second pass where the cost is aggregated in the horizontal direction by a \(1 \times n\) window. Wang et al notes that the square window is not separable in theory but their two-pass approximation works well in practice [18].

### 4.5 Disparity Computation

Disparity computation is the process of assigning a final disparity value to each pixel based on the computed, and possibly aggregated, cost volume. Local algorithms,

---

\(^1\) Euclidean distance is the shortest path between two points. In the 2D case, this is equivalent of \(\sqrt{\Delta x^2 + \Delta y^2}\).
which usually perform most of the work in the aggregation step, uses a winner-take-all (WTA) approach where they simply select the disparity with the lowest accumulated cost.

Global algorithms usually perform little or no cost aggregation. They are instead focusing on the disparity computation. By minimizing a global energy function, a disparity can be found for every pixel. As mentioned in [11] and [18] the energy function is usually formulated by:

\[ E(d) = E_{\text{data}}(d) + E_{\text{smooth}}(d) \]  

(4.11)

In equation 4.11 \( E_{\text{data}} \) is the similarity measure between the images, i.e. the cost volume \( C(x, y, d) \). \( E_{\text{smooth}} \) is a function enforcing the smoothness constraint based on the assumption that surfaces are piecewise smooth. The function is penalising disparity jumps between neighbouring pixels. The \( E_{\text{smooth}} \) may also be formulated to depend on both disparity difference and colour difference, thus allowing disparity jumps when two pixels don’t have similar colour. As mentioned in section 4.1, the 2D-optimizing problem is NP-hard but max-flow min-cut algorithms among other can be used to find a local minimum. These methods perform very well but are also computationally heavy making real-time implementations impossible.

A more popular use of the energy minimizing approach is to optimize along each scan line individually and thereby makes it possible to reach real-time performance. This can be done with so called dynamic programming which is described in [18] and [19]. The optimizing algorithms are using a lot of branching and looping, thus making them inappropriate for a GPU or SIMD implementation. However, a GPU can be used for the cost calculation and aggregation and then have the cost volume read back from the graphics card memory to the application for dynamic programming. This is done in the algorithm proposed by Wang et al [18].

### 4.6 Post-processing

When the final disparity map has been computed, illustrating the distance between the camera and the objects in the scene, it will most likely have a significant amount of mismatched pixels, especially when using a local WTA algorithm. This may be due to noise, ambiguous matches or occluded areas which obviously cannot be matched correctly at all. There are several post-processing methods which can enhance the quality of the disparity map. Some of them aim to correct mismatches whereas other simply invalidates pixels whose match is not considered to be reliable.

Mismatches in the form of salt and pepper noise can be corrected by applying a median filter to the disparity map. Another method similar to this is the use of a min-filter. By min-filtering, a pixel’s disparity is replaced by the disparity of a pixel in it’s neighbourhood with the lowest accumulated cost.

A frequently used technique of invalidating occluded pixels is the left-right consistency check. Disparities are calculated twice, using both the left and right images as reference frames and then crosschecked. Pixels that do not have the same disparity in the left and right reference frame, are considered to be occluded and are invalidated. This is an effective method to remove halo-effects on near objects where the borders suffer much occlusion. This however comes with a major increase in the
number of computations, as the best matching disparity for every pixel has to be searched twice. Di Stefano et al [12] propose an alternative consistency enforcing method which only needs one matching pass. The cost volume is searched from left to right for each scan line. When the best match for a pixel in the left image is found it is assigned to the corresponding pixel in the right image:

\[ L(x, y) \rightarrow R(x', y) \] (4.12)

Another pixel may then find a best match corresponding to the same pixel in the right images. If the cost for the new match is lower than the old match the assignment is updated and the old assigned pixel is invalidated. This algorithm “allows for recovering from possible previous matching errors” [12]. Unfortunately the sequential nature of this algorithm makes it hard to implement in a parallel computing scheme, although similar to dynamic programming, the cost volume could be calculated on a GPU and then read back to the application for disparity selection using the CPU.

### 4.7 Applications

#### 4.7.1 Ground Plane Estimation

Ground plane estimation is the process of determining the location and curvature of the ground, based on a previously calculated disparity map. This is commonly used in applications of autonomous vehicles. van der Mark and Gavrila states that “ground plane estimation is required to distinguish obstacles such as other cars and pedestrians in the disparity image from the road surface” [20]. Labayrade et al [21] proposes a real-time algorithm based on the assumption that a majority of the pixels in an disparity image will belong to the ground, and that ground plane pixels along a certain scan line will have the same disparity. The algorithm transforms the disparity map into a “V-disparity” image by computing the disparity histogram for each scan line. The V-disparity image is then searched for straight lines with the Hough transform. The found lines can then be used to form a piecewise linear curve of the height profile of the ground. Given the height profile of the ground, obstacles can be detected based on their height relative to the ground. The algorithm is aimed towards obstacle detection in highway traffic and is very effective in this environment. However, its performance in situations where the road does not cover the most part of the image due to nearby obstacles or its off-road performance, is not presented. It should also be noted that a reasonably high quality disparity map seems to be needed for this algorithm.

An alternative to explicitly estimate the ground plane is to model the whole environment as a height map or a digital elevation model (DEM). A method of doing this is presented by Thompson and Kagami in [22]. Each pixel in a disparity image is mapped to a grid cell in the height map. Each cell serves as a Kalman filter, continuously updating the height estimate. Kalman filters have proved to be effective estimators when dealing with noisy measurements. In the algorithm, the height map is also converted into a gradient or slope map. When the robot passes over a cell with estimated height, the gradient can be used to calculate the robot’s attitude, and thereby be used to improve the mapping from the disparity map to the height map. The gradient map can also be used for navigation to calculate an optimal route.
4.7 Applications

4.7.2 Obstacle Detection

Obstacle avoidance for autonomous vehicles requires that a map of all obstacles is calculated in real-time. For vehicles operating indoors or on structured surfaces, this is often done by marking points that extend above the ground plane as obstacles. The ground plane can be estimated for example by the algorithm described in [21]. Many algorithms also assume a flat ground.

Matthies et al [23] proposes a fast and simple obstacle detection algorithm. For each scan line, given the position of the intersection with a flat ground plane, the vertical offset, $O(y;H)$, to a hypothesized obstacle of height $H$ is calculated. For any pixel $p_1$ in the disparity map, its height in the camera coordinate system is compared with the height of its higher partner $p_2$. If the change in height is greater than a threshold, all pixels in the column between $p_1$ and $p_2$ will be marked as obstacles. The intersection between the obstacle and the ground plane and transformation between image and camera coordinates can be calculated using equations 4.4a and 4.5.

While algorithms like [23] operate by analysing image columns, other algorithms such as the ones proposed by Manduchi [24] and van der Mark et al [25] works with the 3-D point clouds which is produced by transforming the disparity map to camera coordinates. Neighbouring points are clustered together and marked as obstacles if the line connecting them forms an angle with the horizontal plane greater than a certain threshold.

So far, only dense depth maps, i.e. maps where all pixels are assigned a disparity value, have been discussed. For simpler obstacle detection algorithms, a sparse depth map can be sufficient and is often preferred because it is much faster to produce. In sparse depth maps, only some of the pixels will be assigned a disparity value, these pixels may be areas of special interest such as edges. For the calculation of a sparse depth map, all the methods discussed in chapter 4 regarding local dense depth maps are still applicable with the exception that there are less possible matches for each pixel. In addition, some pre-processing has to be done in order to find the areas of interest. Sun et al [26] and Franke and Heinrich [27] present obstacle detection algorithms where feature based depth maps based on edges in the images are used to speed up the computations.

In the algorithm presented in [26], pixels are first classified with a class index based on the colour difference between the current pixel and its four neighbours. An edge pixel in the left images is then matched against an edge pixel on the same scan line in the right image with the same class index. A disparity histogram is calculated and possible obstacles are detected by finding peaks in the histogram. Bikes and pedestrians are detected by template matching from the locations of the possible obstacles. Sun et al mentions that “the sparse depth map seems to be not enough to construct a complete position for vehicles.” [26] Instead, vehicles are detected by finding horizontal edges and additional pattern matching. Although this algorithm can successfully detect bikes, pedestrians and vehicles in traffic situations, and the fast production of the sparse depth map is interesting, it seems not to be useful for arbitrary obstacle detection in autonomous navigation.

Similar to the previous discussed algorithm, the one presented by Franke and Heinrich [27] is aimed towards obstacle detection in urban traffic situations and produces a sparse depth map based on edge features. To calculate the depth map, the
algorithm uses an interesting multi-resolution approach where correspondences are found at a course level and then recursively refined. A mip map pyramid of $N$ levels is constructed for the left and right images. The matching begins by calculating a disparity map for mip map level $N$. For the next levels, $N-1$ to $0$, the disparity map is iteratively fine-tuned using a small search area of $\pm 1$ pixel from the current disparity selection. “If $D$ is the maximum searched disparity at level zero, it reduces to $D/2^n$ at level $n$. At level 2 this corresponds to a saving of computational burden of about 90% compared to a direct computation at level zero.” [27] The drawback of this method is that mismatches at the first search propagate down through the rest of the levels.
Chapter 5

5 Proposed System

The following chapter describes the proposed complete system of obstacle detection for unmanned ground vehicles aimed towards the AURES platform. Chapter 2 presents the AURES platform which is the foundation of this project. In chapter 3, the basics of using programmable GPUs for general purpose programming to speed up computationally demanding applications such as image processing were described. In chapter 4, different methods for depth estimation and obstacle detection have been discussed. The system is designed using individually distinct components, or modules. This is because different modules such as the depth estimation algorithm should be easily replaced. It also allows us to start out with a quite crude system and iteratively make it more sophisticated. Figure 5.1 shows a component diagram of the complete system.

Figure 5.1: Component diagram of the complete system.
In addition to the modular definition of the system, the components are also grouped under different nodes representing different areas of responsibility. If we consider the UGV as a human being, the video node can be seen as the eyes providing the brain with images of the environment. The perception algorithm is how the brain processes this information and constructs a mental image of the environment. The intelligence algorithm is the process of reasoning, given the perceived environment, find the best route to a target and avoid running into obstacles. The client state is the robot’s body awareness, dealing with position, speed and heading as well as roll and pitch motions. From a programming point of view, one can consider the nodes as different threads. The system is tested against a set of simulated video sequences recorded from the simulator described in section 2.3.

## 5.1 Controller

The controller is the communication interface between the physical UGV, the model and the algorithms. It uses the Auresnet communication API, described in section 2.2, through which the algorithms may control the robots. Additionally, the controller updates the models with information received through status messages from the UGVs. Consequently, the main purpose of the controller is to keep a list of models for all UGVs in action, update the models’ status and forward command messages from the algorithms to the robots. Figure 5.2 shows a class diagram of the proposed implementation. In a multi-UGV system such as the AURES project, the controller along with the models and any cooperative algorithms should preferably be implemented as a centralized unit. Any unit specific algorithms such as the obstacle detection should be implemented on separate computers as shown in Figure 5.3, or mounted directly on the robots.

---

**Figure 5.2:** Controller class diagram.

**Figure 5.3:** Example of a centralized implementation of the controller.
5.2 Video

Video from the robot’s sensors is sent as MPEG 4-compressed video streams via RTP. A library for receiving RTP packets and decoding video frames has been implemented. To set up an RTP session and receive RTP packets, the library uses Jori’s JRTPLIB [28], a cross-platform library that offers support for the Real-time Transport Protocol defined in RFC 3550 [29]. The Xvid video codec [30] is used to decode the video frames.

The video library consists of three major classes; the RTPSession handling the data packets and buffering, the Decoder handling the actual video decoding and the abstract class FrameHandler which defines what to do with the video frames. The Decoder provides the FrameHandler with raw pixel data, letting the user define what to do with it, or even if every frame should be processed. In this project, a FrameHandler has been defined that uploads the video frame to the graphics card memory as an OpenGL texture. This allows for further GPGPU image processing as described in chapter 3. An overview of the video library can be seen in the class diagram in Figure 5.4.

In GPGPU, reading from and writing to the GPU texture memory is a major bottleneck. This is somewhat overcome by the introduction of PCI Express connected graphics cards but it is still a problem. To speed up the streaming texture updates, OpenGL Pixel Buffer Objects (PBO) are used. As an introduction to the Pixel buffer object extension [31] the reader is advised to the article by Ahn [32]. “The main advantage of PBO are fast pixel data transfer to and from a graphics card through DMA (Direct Memory Access) without involving CPU cycles.” [32] Additionally, OpenGL can schedule the DMA transfer for later execution meaning the copy operation call will return immediately and the CPU can perform other tasks without waiting for the pixel transfer to complete. This is utilized in the streaming texture updates using two PBOs. When a video frame \( n \) has been decoded, the CPU updates the texture source to PBO 1 while PBO 2 simultaneously uploads frame \( n-1 \) to the GPU. This causes a latency of one frame but is speeding up the overall process.

5.3 Client state

In order to make accurate depth estimations and correctly update the obstacle map according to the robot’s movements, full knowledge of the robot’s state is required. This includes more static variables such as the placement and focal length of the cameras and also highly dynamic variables such as the robot’s position, orientation and speed. This information is broadcasted by the robots over Auresnet. The Interface Control Document for AURES [33] defines the DLI message \( X3 \) which should contain the following information:
Proposed System

- Latitude position
- Longitude position
- Heading
- Altitude
- Pitch
- Roll
- Payload information for both cameras

According to the control document [33], the DLI message is sent with a frequency of 5 Hz. As this system is aimed at operating at near PAL video rate (25 Hz), it requires prediction of the position and heading for updating the obstacle map. Additionally, it is questionable if the roll values in these messages can be trusted and used in the algorithms as the values are momentarily and the roll motions of the spherical robot will probably change very rapidly.

The client state is generalized to be compatible with any UGV that is using Auresnet and is equipped with two cameras (a stereo rig). A class diagram of the UGV model is shown in Figure 5.5.

5.4 Perception Algorithm

5.4.1 Depth Estimation

As discussed in chapter 4, there are basically two possible approaches for GPU accelerated dense depth estimation algorithms. The first is the local WTA approach computed entirely on the GPU such as the algorithm by Yang and Pollefeys [14]. The other alternative is a GPU-CPU cooperative scan line optimizing approach such as the algorithm by Wang et al [18]. Although the algorithms using dynamic programming like [18] produces very good results and are relatively fast, we want to use as few CPU cycles as possible as the depth estimation algorithm only represents a part of the whole system. Video decoding and obstacle detection are other computationally demanding processes which need to be done for each frame. A local WTA algorithm computed entirely on the GPU is therefore preferred. Furthermore, van der Mark and Gavrila states in their evaluation of different stereo algorithms that errors made by algorithms which use optimization steps such as dynamic programming, affect the ground plane estimation more than simpler algorithms [20].

In order to accomplish real-time processing speeds for the depth estimation algorithm, it is preferable to use as few operations as possible, while maintaining enough quality and precision to support the obstacle detection algorithm. Most algorithms in literature do this by scaling down the input images and by searching a very limited disparity range. This is possible because these algorithms are intended for scenes with a very short depth span. They are also using a short baseline (distance between the two cameras) resulting in near objects having lower disparity, and in that way reducing the possible number of disparities. For obvious reasons, we want the detection range span to be as large a possible, preferably ranging from just in front of

![Figure 5.5: UGV model class diagram](image)
the robot to the horizon. The robot in the AURES project is also stuck with a relatively wide baseline of 60 cm. We start of by examining what influence the stereo baseline, image width and disparity range has on the detection distance. In Figure 5.6, equation 4.5 is used to plot the distance, $Z$, as a function of $d$, where $d$ is the disparity in percent of the image width. The function is plotted for baselines of 15, 30 and 60 cm. In the calculation, a focal length equal of 60 degree field of view is used.

![Figure 5.6: Distance as a function of disparity, plotted for baselines of 15, 30 and 60 cm.](image1)

One can notice how an object at 1 meter distance would have an disparity of 50% of the image width in our configuration (60 cm baseline), while the disparity would only be 12,5% of the image width in a configuration with a baseline of 15 cm. It is also notable how steep the slope of the curve is at low disparities, as $Z \to \infty$ when $d \to 0$. Using discrete disparity values introduces a distance uncertainty that is related to the sampling density, i.e. the image width. We define the change in distance, $\Delta Z$, between two discrete disparity steps as:

$$\Delta Z(d) = Z(d) - Z(d + 1) = \frac{fb}{d} - \frac{fb}{d + 1}$$

(5.1)

![Figure 5.7: Maximum detection distance plotted against image width.](image2)
To make the obstacle detection robust, only disparities that correspond to \( \Delta Z \leq 1 \) meter is said to be trusted. This results in a maximum detection distance based on image width. In Figure 5.7, this maximum detection distance is plotted against image width. A baseline of 60 cm and a focal length equal of 60 degree field of view are used in the calculation.

To acquire the large detection distance span we want, it is obvious that a large number of disparities have to be searched over a wide image. To achieve real-time processing speeds at the same time, this proposed algorithm adopts the mip mapping approach presented by Franke and Heinrich [27] and applies it on dense depth map generation. In their algorithm, the best disparity is searched at mip map level \( N \). For the remaining levels, \( N-1 \) to \( 0 \), the disparity selection is fine-tuned by searching a small area of \( \pm 1 \) pixel. To somewhat mend the problem of mismatches at the first search propagating through the rest of the mip map levels, in this algorithm, the four best disparities are instead searched and passed on for each level. This multi-resolution approach is also supported by Yang and Pollefeys [14], whose algorithm is discussed in section 4.4.3. They state that the sum of different sized cost aggregation windows results in strong and well localized cost minimum at the true depth.

As mentioned in section 5.2, the decoded video frames are stored as textures on the graphics card memory. At the start of the depth estimation algorithm, the images are converted to greyscale and a mip map pyramid of textures is created by repeatedly scaling down the input images with a factor of two. Figure 5.8 illustrates an example of the iterative disparity search process. Suppose the four best disparities, \( d_i \) \((1 \leq i \leq 4)\), are found at mip map level \( N \). These, along with their respective cost, \( c_i \), are passed on to the next level. At this level, for each of the four inputs \( d_i \), the cost is calculated for disparities \( 2d_i - 1 \) to \( 2d_i + 1 \) and the cost from the previous step is added to the result. This results in twelve \( (4 \cdot 3) \) disparity calculations. Out of these twelve, the four with the lowest accumulated cost is chosen and passed on to the next level. After the last selection step is completed, the final disparity is simply chosen to be the one with lowest accumulated cost out of the remaining four. When using a mip map pyramid ranging from 16 to 256 pixels wide images and 128 final disparity levels, this method results in a computational reduction of almost 90% compared to searching 128 disparity levels in the original 256 pixels wide image. This because: 
\[
1 - \frac{12 \cdot 1.33}{128} \approx 0.875 ,
\]
where the 1.33 is the 33% increase in computations from a complete mip map pyramid.

As discussed in section 4.4, calculating the cost based on single pixel samples will not be enough to produce a depth map of acceptable quality. Even though the mip mapping can be seen as an averaging window in itself, some additional cost aggregation is required. This is of major importance in our configuration since the...
wide baseline also increases the number of occluded areas. A cost aggregation method which is proven to be robust and works well at object borders, is the use of several shifted aggregation windows. This proposed algorithm uses a method similar to the one presented by Woetzel and Koch [17], see section 4.4.2. Woetzel and Koch get away with using small 2 x 2 windows because of their multi-camera configuration. In this algorithm however, larger support windows are needed.

When cost computation and aggregation were described earlier, the aggregation was performed by convolving a window $W$ with a complete cost slice $C_d$. In our selective search approach this is not applicable since complete cost slices for each depth hypothesis is not calculated. The cost aggregation is instead approximated by aggregating the colour, or intensity if using greyscale images, of the input images. By rewriting the equations 4.6, 4.8a and 4.9, the cost aggregation can be expressed as equation 5.2 a, while the approximation by colour aggregation is expressed as equation 5.2 b.

\[
C_d^* = ((L_d - R_d) * W)(x, y) \tag{5.2 a}
\]

\[
C_d^* = ((L_d * W)(x, y) - (R_d * W)(x, y)) \tag{5.2 b}
\]

A comparison of these two methods can be seen in Figure 5.9 where they are applied on a cost slice from the Tsukuba set using an averaging window. Cost aggregation is used in the left image and colour aggregation is used in the right image. While the two methods produce similar result for most pixels, the colour aggregation also introduces some artefacts which unfortunately may lead to additional ambiguous matches.

As a pre-processing step applied to each mip map level, four small support windows are “created” by sampling the greyscale texture at 1/2 and 3/2 pixels offset in four different directions, as shown in Figure 5.10. The bilinear texture sampling results in each of the four samples being the average of a 2 x 2 area. The samples are then stored in the texture’s four colour channels; red (R), green (G), blue (B) and alpha (A). Consider the two mip map textures of level $n$ and $n-1$. In Figure 5.11, the red square in $a$ represents a pixel at level $n$ and its aggregated pixels in the R channel. By sampling the same area at level $n-1$, an average of the four pixels in $b$ is returned. The aggregated
pixels in the R channel will be weighted according to the labelled pixels in \( b \). The final aggregated pixel value, \( \bar{P}_n^* \), is defined as the sum of the samples from level \( n \) and \( n-1 \) according to:

\[
\bar{P}_n^* = 2\bar{P}_n + \bar{P}_{n-1}
\]

(5.3)

where \( \bar{P}_n^* \) is a vector of 4 components. The aggregated pixels and their corresponding weighting for each of the components R, G, B and A are illustrated in Figure 5.12. The cost for a pixel at a given disparity is calculated as the absolute difference of the left and right aggregated pixel values according to:

\[
C_n = \min \left( \| \bar{P}_L^* - \bar{P}_R^* \| \right)
\]

(5.4)

Similar to Woetzel's and Koch's algorithm [17], the window with the lowest cost out of the four is used.

![Figure 5.11: Aggregated pixels in the R channel for mip map level \( n \) and \( n-1 \).](image)

Since the wide baseline of our stereo configuration causes many occluded areas and the fact that local WTA algorithms are prone to make mismatches, some kind of error correction is needed. This proposed algorithm uses the consistency check where disparity maps are calculated to both left and right reference frames and then cross-checked. Given the relation between disparity, \( d \), and left and right horizontal position being:

\[
x_R = x_L - d
\]

(5.5)

the consistency check is performed by invalidating pixels based on the following criteria:

\[
|D_L(x,y) - D_R(x - D_L(x,y))| > \lambda
\]

(5.6)

In equation 5.6, \( D_L \) and \( D_R \) is the disparity maps from left and right reference frames respectively and \( \lambda \) is a threshold constant. For low values of \( \lambda \), lots of pixels will be invalidated due to ambiguous matches. However, one can be quite sure that the remaining pixels will represent the true depth. Additionally, by introducing the assumption that the depth of the scene varies slowly over time, the disparity map from the previous video frame can be used to replace the invalidated pixels. This will introduce streaking artefacts when the assumption doesn’t hold such as for fast moving objects, but it is definitely improving the overall quality of the algorithm. As
a last post-processing step, a median filter is also applied to the final disparity map to reduce noise.

To summarize this proposed depth estimation algorithm, a diagram describing the algorithm is shown in Figure 5.13. The different steps are depicted as diamonds. They represent a specific *kernel*, or in this GPU implementation, a fragment shader. The squares in the diagram represents storage areas, or textures. The algorithm is computed entirely on the GPU. In addition, it can be directly ported to any other stream processing architecture with the exception for the additional implementation of the GPU built-in functionality of bilinear texture sampling.

![Diagram of the proposed depth estimation algorithm.](image)

**Figure 5.13:** Diagram of the proposed depth estimation algorithm.

### 5.4.2 Ground Plane Estimation

Due to lack of time and real-world video data from non-planar terrain, no algorithm for ground plane estimation has been implemented. A planar terrain where the cameras are aligned parallel with the ground plane is therefore assumed in the obstacle detection method. This assumption is also true for the simulated video sequences. In addition, for obstacle detection algorithms such as the one by van der Mark et al [25], where 3-D point clouds are examined based on their vertical angle against the vehicle, no explicit ground plane estimation is needed.
5.4.3 Obstacle Map

The depth map calculated in 5.4.1 is not of much use by itself. But it can be used to map the environment or detect obstacles in the line of sight. Different obstacle detection algorithms have been briefly discussed in section 4.7.2. For autonomous navigation, the obstacles would preferably be presented either as a digital elevation model (DEM) or as an occupancy map. Gradients, or slopes, can be calculated from the DEM and possible routes can thereby be determined. An occupancy map is often presented as a binary grid map indicating if each cell is occupied by an obstacle or not. Figure 5.14 shows a hypothetical occupancy map (right) for the scene (left). The red dot represents the point of view and white pixels indicate the presence of an obstacle. The gray areas represent the actual coverage of the obstacles.

![Image of an occupancy map](image)

Figure 5.14: Illustration of an occupancy map.

For the main purpose of testing the quality of the depth estimation, a simple obstacle detection algorithm has been implemented. Based on the assumption of a planar ground and that the cameras are always aligned parallel to the ground, points are marked as obstacles if they are located above the ground plane. The pixels in the depth map are first converted to the camera coordinate system \((X \ Y \ Z)\) using equations 4.1 and 4.5 according to:

\[
Z = \frac{fb}{d}, \quad X = \frac{uZ}{f}, \quad Y = \frac{vZ}{f}
\]

where \((u \ v)\) is the image coordinates \((-0.5width \leq u \leq 0.5width, -0.5height \leq v \leq 0.5height)\), \(f\) is the focal length in pixels, \(d\) is the disparity and \(b\) is the stereo baseline. As the cameras are mounted on the main horizontal axis of the spherical robot shell of 60 cm diameter, this means that the assumed flat ground plane is located at \(Y = -0.3\) meters. The algorithm then simply marks all pixels whose \(Y\)-component is greater than -0.3 as possible obstacles. An occupancy map is then created by mapping the obstacle points to a grid in the \(X-Z\) plane. Each cell is given a value equal to the number of obstacle points belonging to that particular cell. A high value indicates a high probability that the cell is occupied by an obstacle. Furthermore, the calculation of the obstacle map, \(O\), is supported by the result from previous frames according to:

\[
O_t = o_t + \lambda O'_{t-1}
\]

where \(o_t\) is the detected obstacles for the current frame. \(O'_{t-1}\) is the previous obstacle map which has been rotated and translated according to the robot’s movements since...
the last video frame. $\lambda$ is a constant that is $< 1$ used to suppress old values. This result in that only a local occupancy map is generated. A local occupancy map is chosen in this implementation because it is intended for obstacle avoidance primarily.

5.5 Intelligence Algorithm

The intelligence algorithm is where the perceived environment, the obstacle map, is analysed. It is a decision making algorithm, determining if any evasive action is needed and is responsible of re-planning the route if necessary. Due to lack of time and low availability of the robot, no actual intelligence algorithm has yet been implemented. However, such an algorithm would preferably examine the occupancy map after each video frame has been processed. Each of the cells that lie within a certain radius of the previously planned waypoint trajectory is investigated. If any of the cells are occupied, a new route needs to be calculated. Finding the shortest path through an occupancy grid can be done with for example the A* ("A star") search algorithm$^1$.

---

Chapter 6

6 Results and Analysis

6.1 Results

The algorithms presented in this report have been developed and tested against three video sequences of simulated stereo data. Two of the sequences are recorded using the simulator described in section 2.3, using a baseline of 60 cm and a 45 degree field of view. The third sequence is produced by van der Mark and Gavrila and is used in [20]. It is configured with a baseline of 25 cm and a 45 degree field of view. Unfortunately, no live video data from the robot has been available during this work.

The following three subsections present the results from a few frames of the video sequences. The images in order from left to right represent:

1. Image from the left camera.
2. Resulting disparity map from the depth estimation algorithm.
3. Pixels marked as obstacles by the obstacle detection algorithm.
4. Resulting occupancy map where white pixels indicate obstacles. The position of the robot is located in the centre of the bottom row of the image. The height of the occupancy map represent 15 meters in video sequence 1 and 2, and 10 meters in sequence 3.

The depth estimation algorithm is configured using mip maps ranging from 16 to 256 pixels wide images, which results in 128 disparity levels and a 256 x 256 size disparity map. The occupancy map for the first and second video sequences displays a maximum detection distance of 15 meters while the occupancy map for the third video sequence uses a maximum detection distance of 10 meters. It is notable how the shorter baseline in the third sequence clearly affects the resolution of the occupancy map. Furthermore, the camera motion for the third sequence was not known. Updating the occupancy map based on vehicle speed therefore had to be done using estimated values.

6.1.1 Video Sequence 1
6.1.2 Video Sequence 2
6.1 Results

6.1.3 Video Sequence 3
6.1.4 Performance

Although no quantitative analysis of the output data have been done, when examining the resulting occupancy map and detected obstacle points, two questions should be considered:

1. Are there any false negative obstacle detections that would make the robot run into undetected obstacles?
2. Are there any false positive obstacle detections that would make the robot re-plan the route even when there are no obstacles?

False negative detection means that a cell in the occupancy map is marked as free when it is really occupied by an obstacle. False positive detections mean that cells are marked as obstacles when they are free. When studying the results from the obstacle detection algorithm, the 3\textsuperscript{rd} and 4\textsuperscript{th} columns in the image sequences above, one can notice both false negative and false positive errors to some extent. However, the false positives can be eliminated either by down sampling the occupancy map or by additional noise reduction filtering. The number of false negatives is so few that no obstacle in the test sequences would be overlooked by the robot.

The images presented here are produced by a Matlab implementation of the proposed algorithm. An identical algorithm has also been implemented in C++ and GLSL where the depth estimation is computed on the GPU. Table 6.1 presents the measured mean processing times of the different parts of the algorithm. The tests are performed on a PC with an Intel\textsuperscript{®} Core\textsuperscript{TM}2 Quad 6600 2.4GHz CPU and an ATI Radeon HD 2400 Pro graphics card. Additional numbers presented are frames per second (FPS) and million disparity evaluations per second (MDE/sec) for the whole system. MDE/sec is a measurement often presented by researchers of real-time depth estimation algorithms. Our MDE/sec measurement puts us well in line with or above most other real-time stereo algorithms in terms of performance speed. It is also interesting to note that while running, the proposed system uses only about 10% of the CPU capacity.
### Table 6.1: Measured performance of the GPU implementation.

<table>
<thead>
<tr>
<th></th>
<th>Time (ms)</th>
<th>FPS</th>
<th>MDE/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode video</td>
<td>3.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copy images to GPU memory</td>
<td>1.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disparity estimation + read back from GPU</td>
<td>22.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Obstacle detection</td>
<td>0.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>28.4</td>
<td>35.2</td>
<td>295</td>
</tr>
</tbody>
</table>

### 6.2 Conclusion

The purpose of this work was to investigate methods for real-time computer vision and obstacle detection for unmanned ground vehicles and determine how these could be integrated into the AURES platform. In order to do this, an extensive literature study was initially performed. The study showed that although there are many well performing methods for depth estimation and obstacle detection around, none of them really had the requirements and hardware configuration similar to the AURES system. The main difference is the configuration of the stereo rig. Most algorithms use a short baseline and mount the cameras relatively high above the ground. This configuration is ideal for detecting near obstacles, such as a collision warning system, using a low number of disparity levels. The low number of disparity levels allows for fast computation of the depth estimation algorithm. The robot in the AURES project however, has a relatively wide baseline of 60 cm and the cameras are mounted only 30 cm above the ground. Our system needs to be able to detect obstacles at short distances and preferably also mapping the environment accurately at larger distances. The first is to avoid having the robot running into for example humans or other robots. The later is to provide a basis for efficient route planning and autonomous navigation. To achieve this with the given configuration, a high number of disparity levels needs to be calculated for a relatively high resolution depth map. This applied to many conventional stereo algorithms would result in a computational burden many times higher than what most real-time algorithms can handle. To achieve real-time processing speeds and to meet the resolution requirement of the depth estimates, a novel wide-baseline depth estimation algorithm has been developed. The algorithm uses a multi-resolution approach with approximate cost aggregation of multiple shifted windows. The quality of the depth estimation is proven to be high enough by producing acceptable occupancy maps even from a very simple obstacle detection algorithm. In addition, the algorithm also passes the real-time performance requirement of being able to operate at PAL-video frame rate.

### 6.3 Discussion and Future Work

Because of low availability of the robot and lack of live video data, the proposed algorithms have not been tested and evaluated against the live system. However, our hopes are that the simulations are realistically looking enough to provide reliable results. One thing that should be mentioned is that priori rectification of the images have not been needed for the simulated video sequences while it will probably be
needed for real video images. For static stereo rigs, a calibration matrix for the image rectification is usually computed at initialization. Since the camera gimbals mounted on the GroundBot are individually stabilized, the standard rectification process is unfortunately not applicable. How much the two cameras differ in attitude during movement has not been determined.

Another thing that could be discussed is the need of having a stationary PC with a relatively high-end graphics card for each robot. The algorithms proposed here are therefore designed to be implemented in any parallel computing architecture. Although the topic of autonomous navigation were not fully addressed and the fact that the algorithms have not been tested against the live system, this report along with the implemented algorithms and applications should provide a solid ground for future development.
References


[33] Persson M. Interface Control Document for AURES; Issue 0.6. Saab Aerotech; 2008-02-12, AURES TAIS-GS07-xxx.