University of Amsterdam

Master thesis

Development of a Catadioptric Omnidirectional Camera for the USARSim Environment

Author:
Tijn Schmits

Supervisor:
Dr. Arnoud Visser

June 21, 2008

Acknowledgements

During my thesis there have been many people who have supported me and I, of course, thank them for doing so. I can only imagine how hard it is to write a thesis without support. The following people I would like to thank especially:

First I would like to thank my thesis supervisor, dr. Arnoud Visser. He presented me the opportunity to co-develop for and participate in the Atlanta RoboCup 2007 event. I have many fond memories of Atlanta and it was an unbelievable experience. I would also like to thank him for being very creative where there was room to explore and for being very sharp and precise when focus was required. This powerful combination has made a large contribution to my thesis, for which I am grateful.

Second I would like to thank Bas Terwijn for sharing the Nomad Super Scout II mounted with the catadioptric omnidirectional camera for a day, and for being very patient while I had to transfer 1.4Gb of test data from his laptop to mine with a slow 500Mb USB stick. I would also like to thank Bayu Slamet and Aksel Ethembabaoglu as working with you two has been a real pleasure. I would also like to express my appreciation to all the guys from the Jacobs University team present at the Atlanta Robocup 2007, Stephen Balakirsky, Stefano Carpin and Ben Balaguer. All of you have put a lot of effort into making USARSim what it is today, and I am proud to be part of it.

I would also like to thank Marianne Hoeing as she played in important role during the start of this research project. She helped me to get organised and she helped me to find my way, sometimes just by listening. I am grateful to Michiel Kamermans for proof-reading my thesis, I very much appreciate his willingness to share his impeccable knowledge of the English language.

My special thanks to thank Alfred, Godelieve, Yarah and Yosha. I am glad I decided to write most of my thesis in Kekerdom, I did not only find the peace to stay dedicated to my thesis, I also found a family. Finally, I would like to thank my mom for being there for me. You are, and will always be, my biggest fan.

abstract — The first Urban Search and Rescue operation which involved documented use of robots was at the September 2001 collapse of three towers at the World Trade Center. As researchers rarely have the opportunity to do research in real disaster environments, several standardised simulation methods of USAR situations have been devloped. In this thesis I will discuss USARSim, a 3D simulation environment of mobile USAR robots and environment built on top of a 3D game engine created by Epic Games™and I will describe the development process of a catadioptric omnidirectional camera simulation model for this environment. I will answer the questions whether it is possible to create this simulation model and how much realism is contained in simulated sensor data and thoroughly discuss the method used to create the simulation model. I will show that indeed, it is possible to create simulation model of a catadioptric omnidirectional camera in the USARSim environment and the in terms of image composition it is possible to extrapolate results obtained in the simulation environment using the simulated omnidirectional camera model. This will demonstrated by performing a two phase self localisation method, employing a colour histogram based landmark detection algorithm in combination with an Extended Kalman Filter, in both a real and simulated environment. I will also show that realism of sensor data in terms colour is heavily dependant on the environment in which the camera operates. I will show that, though there is room for improvement of realism in the simulation model, the catadioptric omnidirectional camera is a valuable addition to the USARSim environment. This thesis is a partial fulfilment of the requirements for the title of Master of Science in the field of Artificial Intelligence in accordance with the curriculum set forth by the faculty of Artificial Intelligence of the University of Amsterdam.

Contents

List of Figures

1 Introduction

Extreme natural events happen all around the world; droughts, earthquakes, flooding, hurricanes, landslides, tsunamis, volcano eruptions, wildfires, etc. When these take place at or near inhabitated environments they usually put lives in danger and if they do, these natural events are labelled natural disasters. Disasters do not have to be natural, as massive hazardous material spills, aircraft accidents and catastrophic structure collapses can also create disaster situations by putting lives in danger and doing massive damage to an urban environment. These disaster situations in an urban environment define the background setting of this Master's thesis subject.

The definition for the term 'urban environment' differs amongst nations and they generally include a minimal population density or a specific use of land. Since January first 1998, a 500×500 meter area in the Netherlands is labelled as 'urban environment' by *het Centraal Bureau voor de Statistiek* (the Central Bureau for Statistics, or CBS) if it has a density of 1500 or more addresses per square kilometre[1]. As is generally the case with definitions of urban environment, it defines area as covered with buildings, a means for transportation (i.e. above ground and underground roads and railways) and all other facilities which give a certain quality to living in a high-density population area.

The consequences of a disaster situation in these urban areas generally include structural collapses of aforementioned facilities and, hence, include hazardous areas due to structural instabilities and leaks of flammable substances. As disasters usually affect a much larger area than regular emergency situations, they demand specific skills in rescue operations. Around the world, there is a specific type of organisation which comes into action when disasters occur in urban environments, called Urban Search and Rescue or USAR[1].

[1] For reference: U.S.: http://www.fema.gov/emergency/usr/index.shtm, U.K.: http://www.usar.org.uk/, Netherlands: http://www.usar.nl/,
all websites last accessed: April 30th, 2008.

1.1 Urban Search And Rescue

The term USAR is generally used to describe an operational framework of national rescue facilities which respond to (inter)national disaster situations in urban environments. Many countries have their own version of a USAR team, though the common role of USAR task forces is to support local emergency responders to locate victims and manage recovery operations.

USAR response teams involve personnel from fire brigades, medical services, police brigades and the military, which all work in four areas of specialisation:

- Search - locating victims after a disaster
- Rescue - safely retrieving trapped victims from hazardous environments
- Technical - structural specialists who make rescue operations safe for the rescue teams
- Medical - care for the victims before, during and after a rescue

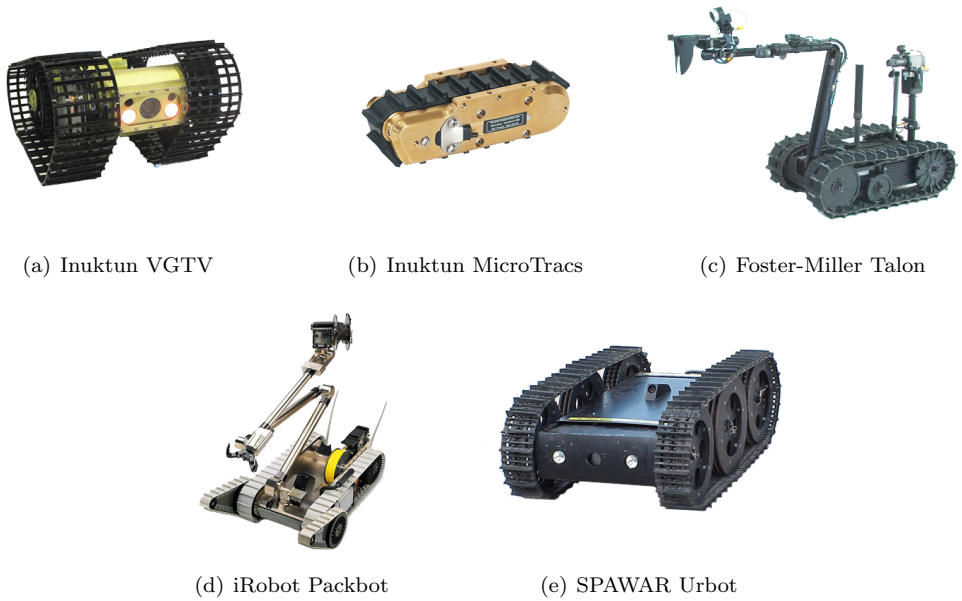
The subject of this thesis mainly focusses on the first area of specialisation: Search. USAR teams need to locate victims that might be trapped in various hazardous situations (i.e. victims trapped under tons of earth and/or constructional debris from collapsed buildings, etc.). Locating these victims is usually done by human rescuers or trained dogs, but in 2001 a new method to aid USAR task forces was introduced: the use of robots.

1.1.1 Robots in USAR Operations

It is known that robots have been used to enter dangerous areas since 1995[2], but the first USAR operation which involved documented use of robots was at the September 2001 collapse of three towers at the World Trade Center [2]. The purpose of the robot deployment was to collect information about the situation and deliver possible victim locations in areas which were impossible or too dangerous for rescue workers to enter.

Robots from Foster-Miller, Inc., iRobot Corporation, Space and Naval War Systems Command and the University of South Florida were used to search the disaster area for damages to surrounding structures, evidence such as the aeroplanes' black boxes, victims and casualties. They were the Inuktun VGTV and MicroTracs, Foster-Miller Talon and Solem, iRobot Packbot, and SPAWAR Urbot (Figure 1.1). During this operation, numerous difficulties and problems arose. The robots were not designed for USAR purposes and therefore suffered greatly from the hazardous environment. Amongst other difficulties, communication was limited, tracks melted due

[2] In 1995, at the Oklahoma City Bombing disaster (Oklahoma, United States of America), it was for the first time a bomb-squad employed the use of a robot.



(a) Inuktun VGTV (b) Inuktun MicroTracs (c) Foster-Miller Talon

(d) iRobot Packbot (e) SPAWAR Urbot

Figure 1.1: Robots involved in 2001 World Trade Center USAR operations.

to extreme temperatures and robots toppled over due to extremely tilted surfaces. In spite of all these difficulties, the use of robots in USAR operations did show promise, and research into this field of expertise was advised [3, 4].

1.1.2 Research in USAR Robotics

The advice was heeded as ever since this introduction of robots to USAR operations, research has been done to overcome known and expected difficulties as well as to improve efficiency and reduce cost. At present day, the collection of USAR robots has been expanded by different types of robots such as marsupial and shape-changing robots [5], hovering reconnaissance types [4] and robots without wheels based on the snake [6]. Currently, there is also a lot of attention given to research into modular reconfigurable multi-agent teams [7]. And not only the robot platform is being expanded, numerous research has been performed to improve the user interfaces by which tele-operators communicate with the remote agents [8, 9].

Disasters happen regularly around the globe, but a disaster is a rare event locally. From almost all points of view this is quite fortunate, though for the research community this fact is slightly unfortunate as this means there is rarely the opportunity to gain access to real-life disaster environments to experiment in. As a result, the research community has to do most of their research using staged field tests and disaster simulations. The U.S. National Institute

of Standards and Technology (NIST) constructed several standardised Reference Test Arenas for Urban Search and Rescue Robots, allowing reproducible challenges allowing researchers to compare implementations and results [10].

It is heavily argued that a staged or simulated USAR response cannot fully simulate all aspects of an unpredicted response (i.e. unforeseen weather and environmental conditions, emotions, and time pressured situations) [3], which will receive some thorough attention in section 5.1. Though regardless of any discussions, simulations have proven to be a valuable asset in research as they provide the basis for testing certain theories in a highly controllable test environment. Simulation environments also provide a means to host multiple competitions, which are a popular method to promote competitive international research cooperation.

1.2 RoboCup/AAAI USAR Competition

In the year 1997 Hiroaki Kitano et al. proposed a Robot World Cup as a new standard problem for AI and robotics research. It was a proposal to use a soccer game as a platform for a wide range of AI and robotics research, such as design principles of autonomous agents, multi-agent collaboration, strategy acquisition, real-time reasoning and sensor-fusion [11]. At the first RoboCup, held in 1997 in Nagoya, 38 teams from 11 countries around the world participated in 6 different leagues. Ever since the first RoboCup, the event has grown in popularity and size. More leagues have created, more attention was received from the press and since the year 2000 an international symposium has been held at the end of every event. Ten years after the first event, at the 2007 RoboCup event held in Atlanta, 321 teams from 39 countries participated in 14 different leagues.

A new league domain beyond the soccer game was introduced in the year 2000 by Satoshi Tadokoro and Hiroaki Kitano. They started the development of RoboCup Rescue League [12], to which they were inspired after the Great Hanshi-Awaji earthquake[3]. At the RoboCup event held in 2002 the RoboCup Rescue League was divided into two main strands, the Rescue Robot League and the Rescue Simulation League.

1.2.1 Rescue Robot League

In the RoboCup Rescue Robot League, teams compete by navigating real robots through *Reference Test Arenas for Autonomous Mobile Robots*, developed by The National Institute of Standards and

[3]It hit Kobe City on the 17th of January 1995 causing more than 6500 casualties and destroying more than 80,000 wooden houses. It affected more than 1 million people and the damage was evaluated at more than 1 billion US dollars.

Figure 1.2: The RoboCup Atlanta 2007 NIST Reference Test Arena for Autonomous Mobile Robots

Figure 1.3: A robot manouvering through the arena.

Technology (NIST), in which points can be scored by achieving certain objectives related to USAR operations. One major goal of this competition is to encourage development of capable robots with intuitive operator interfaces that can negotiate increasingly complex and difficult indoor and outdoor urban environments and within this competition teams and robots with complementary capabilities must collaborate to produce effective systems.

So far within this league robot teams have demonstrated repeated successes against the obstacles posed in the arenas, which makes an annual increase in the level of difficulty possible. By doing so, the arenas will provide a stepping-stone from the laboratory to the real world. The league also provides a platform for direct comparison of robotic approaches, objective performance evaluation, and a public proving ground for robotic systems able to operate in the field.

1.2.2 Rescue Simulation League

In the RoboCup Rescue Simulation League teams compete by navigating agents through simulated disaster environments, in which also points can be scored by achieving certain objectives related to USAR operations. The purpose of the RoboCup Rescue Simulation League is twofold. First, it aims to develop intelligent agents and robots that are given the capabilities of the main actors in a disaster response scenario. Second, it aims to develop simulators which emulate realistic phenomena predominant in disasters. This league in itself is divided into two sub-leagues, the first focusing on agents and infrastructure called the Rescue Agents League and the other focusing on robot simulation called the Virtual Robots League.

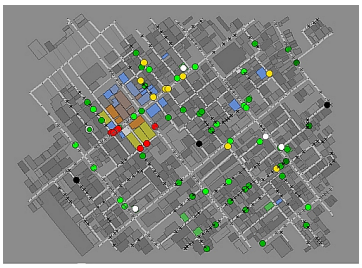


Figure 1.4: A 2D view within the Rescue Agents Simulator

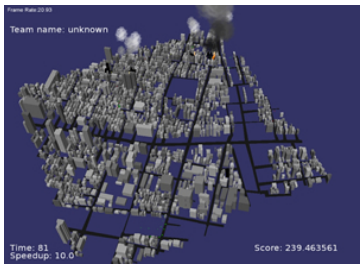


Figure 1.5: A 3D view within the Rescue Agents Simulator

Rescue Agents League

The first sub-league of the Rescue Simulation League focuses on disaster simulation on a macroscopic scale where the purpose is to find a method to manage heterogeneous intelligent agents such as fire fighters, commanders, victims, volunteers, etc. to conduct search and rescue activities in a virtual generic urban disaster simulation environment, constructed to operate on network computers. The simulation environment implements a traffic, fire and civilian simulator interfaced through multiple visualisations, as seen Figure 1.4 and 1.5. Within this competition competitors provide emergency decision support by integration of disaster information, prediction, planning, and human interface, which necessarily requires interdisciplinary research valuable to the USAR community.

The Virtual Robots League

The second sub-league of the Rescue Simulation League focuses on disaster simulation on a more microscopic scale, as it simulates Robots and their direct physical environment in disaster areas. This competition aims to pose as the meeting point between Rescue Robot League and the Rescue Agents League. The purpose of this league is to explore elements related to robot operations such as behaviour strategy, mapping, object recognition, communications and human-robot interaction (HRI). The simulation environment, by the name of USARSim, consists of high fidelity three-dimensional models of disaster situations in which models of existing robots mounted with many types of simulated sensors can be spawned and controlled. The goal within this competition is to use tele-operated robot agents to explore, map, and collect as much information as possible about an unknown and deeply unstructured environment in a set period of time [13].

My supervisor Dr. Arnoud Visser and I participated in the Virtual Robots event at the Atlanta RoboCup 2007, where we used Simultaneous Localisation and Mapping (SLAM) to create maps of

Figure 1.6: An USARSim disaster area simulation as used in the Bremen RoboCup 2006.

the simulated disaster areas, used Histogram based Colour Detection to find victims within these areas and used Wireless Multihop Communication to explore beyond signal strength boundaries. Our team claimed 4th place out of 20 qualified teams. The experiences gained in this event aided in realising this Master thesis project.

1.3 USARSim

USARSim is a 3D simulation environment of mobile USAR robots and environments, intended as a research tool for the study of Human-Robot Interaction (HRI) and multi-robot coordination and designed to be a simulation companion to the National Institute of Standards (NIST) Reference Test Facility for Autonomous Mobile Robots for Urban Search and Rescue [14]. USARSim is a platform-independent open source project built on top of a 3D game engine created by Epic Games™and is based on a client/server architecture. It is a high fidelity simulation environment, which means it utilizes very realistic materials and equipment to represent the task(s) that an agent must perform. The environment supports rigid body simulation (including gravity, inertia, collision and torque) and the engine provides the use of dynamic actors (such as moving victims) and scripted events (such as structural collapses due to pressure). One of the aspects which makes USARSim so accessible is that its socket-based control API allows users to test their own control algorithms and user interfaces without additional programming. A screenshot of a disaster environment simulation by USARSim can be seen in figure 1.6.

1.3.1 Simulator Development

USARSim is constantly in development because one of the main goals of the Rescue Simulation League is to add content to the simulation environment every year. On top of that, USARSim has found its use beyond USAR field and has a devoted community which also adds simulation models and other content. Recent expansions include a new GPS sensor, new Robot Models and the addition of a wireless simulation server (WSS) simulating the consequences of signal loss in wireless communications between agents. Competitions held across the globe result in the annual production of new maps, which either represent disaster areas or testing arenas.

The subject of this thesis is part of this development process, as this thesis is about the creation of a new visual sensor for USARSim.

1.4 Research Questions

At the start of 2007, the USARSim environment did not include a simulation model of a catadioptric omnidirectional camera, in spite of the fact that in the robotics field of science omnidirectional vision has played a major part in past and present research. The omnidirectional camera is frequently employed in the robotics field of science as it has many advantages over a regular camera. The research question which I would like to answer in this thesis is twofold. First, is it possible to create a simulation model of a catadioptric omnidirectional camera in the USARSim environment to be used as a tool for research in methods using this type of vision? Second, if such a simulation model can be created, how much realism does the simulated sensor data present, considering the camera's intended applications?

In this thesis I will describe the process of adding a new simulated visual sensor, the catadioptric omnidirectional camera, to the high-fidelity simulation environment USARSim. First, I will investigate the properties of a real catadioptric omnidirectional camera and then I will investigate technical elements of the simulation environment which could constitute a simulation model of this visual sensor. I will then describe the choices made which eventually led to a simulation model. Finally, I will test and discuss the validity of this simulation model.

1.5 Thesis Structure

Chapter 2 describes the technical background elements which form the basis of this project. It provides background knowledge of omnidirectional vision and some insight into its most important

aspects. This chapter also describes the simulation platform in which the omnidirectional camera model was developed.

Chapter 3 describes all methods employed for simulating and validating the omnidirectional camera. Descriptions of both the real and simulated omnidirectional camera will be given and techniques used for simulating the parabolic mirror surface will be explained. This chapter also describes all UnrealScript programming which needed to be done. Finally, techniques used for validating the simulation model will be defined and explained.

Chapter 4 contains the results of simulation model performance and validation tests. The performance test results played an important role in the final simulation model design. Validation tests are described in which simulation data is compared to real data.

Chapter 5 takes a critical perspective on the omnidirectional camera simulation model and investigates the quality of the simulation model with respect to intended applications. Limitations of the simulation model will be pointed out and possible further developments will be discussed.

Chapter 6 will sum up the project and will describe conclusions which can be drawn from this thesis project.

2 Technical Introduction

This chapter describes the technical background elements which form the basis of this project. The first section is about history, methods, usage and properties of omnidirectional vision. This section should provide fundamental background knowledge of this visual sensing method, as well as give some insight into its most important properties which will play an important role in this thesis. The second section is about the simulation platform in which the omnidirectional camera system is simulated. This section explains all technical elements of the simulation environment which defined the structure for the omnidirectional camera simulation method, discussed in the next chapter.

2.1 Omnidirectional Vision

Agents operating in a complex physical environment more often than not benefit from visual data obtained from their surroundings. The possibilities for obtaining visual information are numerous as one can vary between lenses, imaging devices and accessories. Lenses generally influence a camera's field of view, imaging devices generally determine in which spectrum data will be perceived and accessories can have an influence on both aspects. This section is about using lenses and accessories to increase a camera's field of view (FOV).

A television camera typically has an FOV between 30° and 60° and a photo camera can mount a wide-angle lens with a 75° FOV. It is said the human eye has an FOV of 170° × 125° [15], though these values differ between people and different values can be found in literature. In the 1970's, Donald W. Rees submitted a viewing method to the United States Patent Office, which provided a 360° viewing angle of a person's environment utilising hyperbolic ellipsoidal reflecting optics [16]. Ever since the 1970's many methods have been devised and researched which provide a full

360° view of the camera's surroundings. A system providing a 360° view of its surroundings is called an omnidirectional vision system.

The concept of omnidirectional vision has since also been inspired by biological examples such as the visual system of flying insects. Arthropods have two large compound eyes made up of tiny hexagonal lenses (facets) organised in a honeycomb pattern. Each facet perceives a small part of the environment and all facets combined provide a complete view of the arthropods' environment, providing vision beyond a 90° field of view on two sides of the head. An omnidirectional vision method inspired by this biological example is discussed in Subsection 2.1.2.

In robotics, omnidirectional vision is generally used for navigation and self-localisation. The ISLA group uses omnidirectional vision for appearance based self-localisation using a probabilistic framework [17, 18, 19, 20]. Related to the RoboCup, Heinemann et al. use a novel approach to Monte-Carlo localization based on images from an omnidirectional camera to do soccer robot self-localization and have come with a mutation operator which proves to be a fast and reliable camera calibration method [21, 22]. Self-localisation is also done by Aihara et al. [23], using eigenspaces from autocorrelation images created from omnidirectional images containing global information. 5 Rizzi and Cassinis fed omnidirectional colour image data to a learning subsystem (both neural networks and statistical methods have been tested) to perform self-localisation as well [24]. And again related to the RoboCup, Lima et al. have used a multi-part omnidirectional catadioptric system to develop their own specific mirror shape which they use for soccer robot self-localization [25].

First, an important constraint with respect to visual sensing is described, because satisfying this constraint greatly influences effectiveness of the omnidirectional vision method in certain situations. Second, four methods for obtaining omnidirectional data will be described. After that, the chosen method for simulation will be discussed with respect to the Single Viewpoint Constraint.

2.1.1 The Single Viewpoint Constraint

The single viewpoint constraint (SVC) is a requirement for an (omnidirectional) visual sensor, demanding that the sensor only measures the intensity of light passing through a single point in 3D space [26]. Not all omnidirectional camera methods meet the requirement as some methods capture 360° visual information by measuring light passing through multiple points in 3D space. However, meeting the SVC is desirable for omnidirectional vision sensors as it allows the construction of

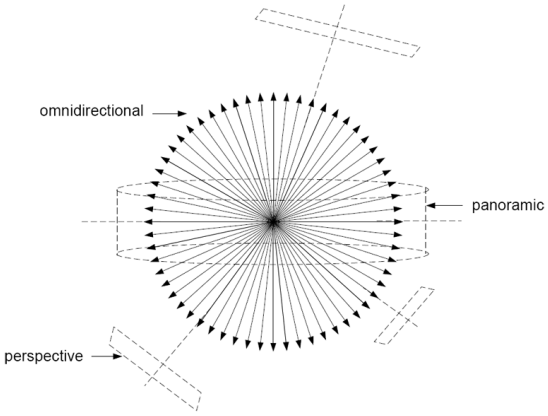


Figure 2.1: The construction of pure perspective images computed by planar projection or a panoramic image computed by cylindrical projection.

geometrically correct perspective images, which can be processed by the enormous amount of vision techniques that assume perspective projection.

The construction of geometrically correct perspective images is founded on the knowledge that, as light passing through a single point in 3D space is measured, all pixels in the captured image are a measurement of the spectral irradiance in a particular direction from that single viewpoint. The particular direction of the measurement is known for each omnidirectional image pixel, as the geometry of the viewing system is known and static. Using these known parameters, a projection from the single viewpoint on a hypothetical planar projection plane can be computed, resulting in a geometrically correct perspective image. Note that panoramic views can also be constructed using these known parameters by projection from the single viewpoint onto a cylindrical projection plane, resulting in a panoramic representation of image data which is more intuitive to a human spectator than the original omnidirectional image data. The construction of pure perspective images is depicted in Figure 2.1.

The SVC turned out to be not just a constraint which the simulated omnidirectional camera should meet, it also proved to be the key concept on which the simulation method was based, though this will be explained further on in Subsection 3.1.3.

2.1.2 Omnidirectional Viewing Methods

Omnidirectional vision can be obtained through many different methods. The difference between these methods lies in the type and cost of required hardware and image post-processing methods. Current omnidirectional vision systems can be divided into four categories, which will be described

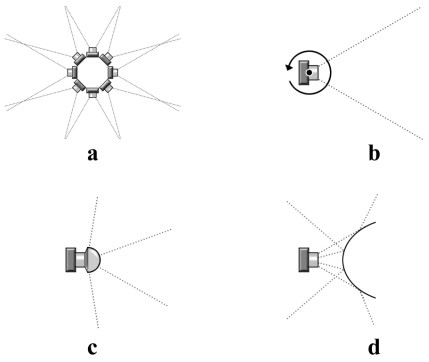


Figure 2.2: Omnidirectional Viewing Methods

next.

Multiple Cameras

The Multiple Camera Method (Figure 2.2(a)) uses, as the name so aptly implies, more than one camera to obtain image data with a 360° FOV. With this setup, multiple cameras are placed on an equal height, rotated around the vertical axis (z). With this setup all 360° viewing angles are seen by at least a single camera. This method produces multiple camera views, which can be stitched together into a full 360° view of the robots surroundings.

The main advantage of this method is the fact that interpretation of the raw camera data is quite intuitive to human spectators, so there is no need for complex post-processing to make the data interpretable by robot tele-operators. Although this is a major advantage over the other methods, this method does have multiple drawbacks. This setup requires the purchase of multiple cameras, which makes it more expensive than most methods. More important however, as the multiple cameras in this setup do not share the same point of view, this setup does not satisfy the SVC. This results in perspective distortion between camera views from different cameras, which makes complex post-processing necessary if certain techniques such as automated localisation are desired.

With respect to simulation, this method is easily reproduced in the USARSim environment as multiple cameras can be mounted on the robot at very precise locations and orientations. The use of multiple cameras has a cost, as it influences the frame rate of the 3D render engine, though most current systems are quite capable of rendering multiple camera views without the framerate dropping below an acceptable amount. As with the real life solution, this solution does not satisfy SVC if proper object collision is taken into account. Virtual cameras can be placed with complete

disregard to realistic collision parameters, so cameras could be placed on a single point in 3D space, hence reducing the complexity of any required image post-processing. It must be pointed out that any simulated physical bodies of each camera would most likely obstruct the view of other cameras, so to make this simulation setup work the rendering of the 3D meshes has to be turned off. However, this solution is not realistic at all and it should therefore not be considered for simulation purposes (see Section 5.1).

Recently, an innovative proposal for an omnidirectional vision sensor which belongs to this category has been introduced. In 2006 Jeong et al. proposed the fabrication of biologically inspired artificial compound eyes. They use artificially created ommatidia with facet microlenses, omnidirectionally arranged along a hemispherical polymer dome such that they provide a wide field of view similar to that of a natural compound eye [27]. As the positioning of the microlenses is customisable, this method is capable of producing highly customizable omnidirectional image data. The microlenses can also be configured in such a way that the sensor satisfies the SVC.

Though belonging to this category of omnidirectional vision methods, this solution uses a number of cameras of a higher order and the microcameras themselves are very different from standard cameras, making it hard to compare to the previously mentioned method. As this method is not yet proven technology and as there is no documented use of this method in USAR operations, this method has not been considered for simulation.

Vertical Axis Rotation

This method (Figure 2.2(b)) is quite similar to the previous method, though instead of using multiple static cameras, it uses a single camera rotating around the vertical axis. This solves the problem of having to buy multiple cameras and, if the camera is rotated around its viewpoint, it satisfies the SVC.

This method is also capable of producing image data which is highly intuitive to a human observer, though it has two major drawbacks, different from the previous method. The first is the fact that the camera is not able to provide a 360° view of the robot's surroundings at a single moment in time. This is not necessarily a drawback in static surroundings, but USAR situations in real life are rarily static. The robot itself must also remain stationary during a complete revolution of the camera if this method is to satisfy the SVC. If the robot moves during a single complete camera revolution, the 360° image data is not obtained from a single viewpoint. The second drawback with this method is blurring of the camera view when the camera moves. Most cameras

in a dynamic situation provide blurred image data (which will receive some more attention in Chapters 2 and 5), and low-cost cameras more often produce lower quality data than the higher priced cameras. This means that the camera rotation speed is inversely related to the image data quality. A low camera rotation speed in a dynamic environment reduces the 'omnidirectional value' of the obtained images as they are most likely not up to date any more. Omnidirectional images obtained by this method thus suffer from a trade-off between image quality value and image omnidirectional value.

Cameras in USARSim do not suffer from blurring due to camera motion (see 2.2.3), which nullifies the omnidirectional value trade-off. Also, the mechanical rotation in USARSim is not limited due to power restrictions or friction, so the camera can be rotated around its viewpoint at unrealistic high speeds without the system braking down. However, as rotation speed should be limited to realistic levels to maintain a certain realism within the simulation, the omnidirectional value of simulation data is limited by the simulation of realistic rotation speeds.

Fisheye Lens

A fisheye lens is a wide-angle lens, featuring FOV angles of 180° to 220°. Placing a camera with a fisheye lens in a vertical position, either upwards or downwards, creates a horizontal 360° view on the camera's surroundings within the vertical 180° - 220° angle.

This method (Figure 2.2(c)) has the major advantage of producing omnidirectional image data from a single viewpoint perspective, using just a single camera without complex mechanical accessories. This method satisfies the SVC and the camera image mapping function is known and fairly simple. This makes the method quite useful for automated applications based on omnidirectional data. The downside to this method is the cost of a fisheye lens, which currently ranges between €400,00 to €950,00. In USAR situations robots more often than not operate in hazardous environments and the cost of a fisheye lens is quite high with respect to the risk of damage or loss. Hence, most robots are mounted with the (cheaper) omnidirectional camera setup described in the next section and for this reason the fisheye lens is not chosen for simulation.

If the fisheye lens is to be simulated in the USARSim environment, it is not trivial. As the sole camera type available in the simulation environment is an ideal perspective camera, a solution must be found to create the transformed fish eye perspective. This could be done by either creating an appropriate `Camera` subclass in Unrealscript (see 2.2.1) or using the methods described in this thesis (chapter 3).

Catadioptrics - the simulated method

The catadioptric method (Figure 2.2(d)) is based on obtaining an omnidirectional view using a camera in combination with a (number of) mirror(s)[1]. The mirror is placed directly in front of the camera and the camera is placed in a vertical position, pointing either directly upwards or downwards. There are a lot of variations in the amount and shapes of mirrors used for obtaining a 360° view, though the most general setup is the use of a single spherical or hyperbolic convex mirror. The idea behind this setup is obtaining a 360° view of the camera's surroundings from the mirror's reflection.

This setup is a relatively low-costs solution compared to the previous solutions and it can provide high quality omnidirectional data. Also, depending on the shape of the mirror it is possible for this method to satisfy the SVC (see next section). This makes this method quite popular in robotics. The downside of this method is the presence of the recording camera in the reflection image, though this is a minor drawback as the camera occupies a relatively small portion of the complete omnidirectional view.

Simulation of this setup in USARSim is not trivial, though with the methods described in this thesis it can be done. I chose to simulate the catadioptric camera as it is a frequently used omnidirectional image sensoring method in robotics and due to its relative low cost it is feasible that this camera is going to be used for USAR operations. A second reason is the fact that the omnidirectional camera at the University of Amsterdam (described in Subsection 3.1.1) belongs to this category, which made verification of a simulation model possible.

2.1.3 Convex Mirrors and The Single Viewpoint Constraint

For a catadioptric system to satisfy the SVC, all irradiance measurements must pass through a single point in space. For example, placing an ideal perspective camera in front of a planar mirror will create a catadioptric system which satisfies the SVC as all irradiance measurements pass through the mirrored camera viewpoint, also called the *effective viewpoint*. Placing the ideal perspective camera in front of two non-aligned mirrors creates a catadioptric system which does not satisfy the SVC as this creates two effective viewpoints at different locations in 3D space.

Baker and Nayar constructed two surface equations which comprise a general solution of the

[1] *Dioptrics.* - the science of refracting elements (lenses). *Catoptrics* - is the science of reflecting surfaces. Hence the name *Catadioptrics*

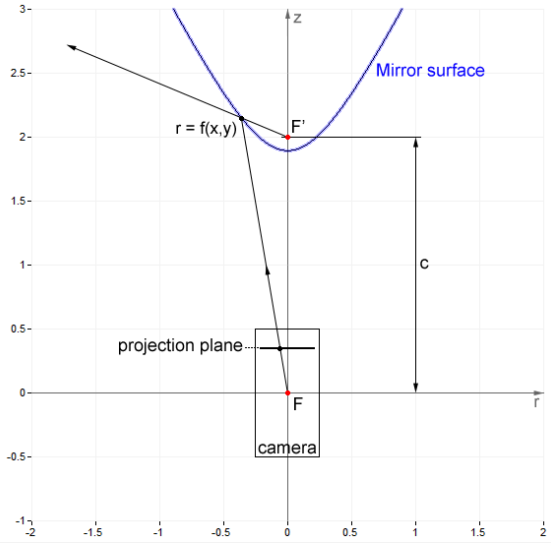


Figure 2.3: Parabolic mirror and camera geometry. Mirror surface is defined by equation 2.1 with $k = 10$ and $c = 2$.

single viewpoint constraint equation [26];

$$\left(z - \frac{c}{2}\right)^2 - r^2\left(\frac{k}{2} - 1\right) = \frac{c^2}{4}\left(\frac{k-2}{k}\right) \quad (k \geq 2) \tag{2.1}$$

$$\left(z - \frac{c}{2}\right)^2 + r^2\left(1 + \frac{c^2}{2k}\right) = \left(\frac{2k + c^2}{4}\right) \quad (k > 0) \tag{2.2}$$

with c denoting the distance between the camera viewpoint F and the effective viewpointF', r is defined by $r = \sqrt{x^2 + y^2}$ and k is a constant . This two-parameter (c and k) solution has five types of solutions: planar, conical, spherical, ellipsoidal and hyperboloidal. The omnidirectional viewing system simulated for the USARSim environment belongs to the hyperboloidal solutions, which are defined by equation (2.1) with $k > 2$ and $c > 0$.

In figure 2.3 the geometry of the hyperbolic catadioptric camera is displayed. In this example, where the mirror surface is defined by equation 2.1 with $k = 10$ and $c = 2$, it is depicted how a measurement by the camera with viewpoint F is reflected in the parabolic mirror such that the effective measurement passes through the effective viewpoint F'. By definition of Baker and Nayar's general solution of the SVC equation, this is true for all measurements done by the camera [26].

2.1.4 Digital Camera Architecture and Image Deformation

In general, digital recording cameras used in omnidirectional systems can be subdivided into three basic elements: an enclosed hollow chamber with a small opening for light to enter, a recording device inside the chamber to capture this light and a lens placed in front of the opening. Images are produced by rays of originating from the object space which pass through the camera perspective centre (the lens) and the aperture (the chamber opening) until they hit the focal plane (the recording device, in most cases a CCD or CMOS array). Movie recordings are created by a shutter opening and closing the aperture at a constant rate, thus allowing a sequence of frames to be captured by the recording device. Due to this camera architecture there several image deformation artefacts that can occur which are able to reduce camera data quality. As some artefacts are relevant to the camera simulation model, their effect and origins need to be described.

In photogrammetry there exists the collinearity model, which assumes that all rays of light from the points in object space to the points on the focal plane are straight. There are four principal sources of departure from collinearity that are physical in nature: radial lens distortion, decentric lens distortion, image plane unflatness, and in-plane image distortion [28]. Both lens distortions can be produced by either misalignment of the lens components or low quality lenses. The other two distortions are produced by inaccurate recording devices which produce a systematic error. Multiple methods for calibrating cameras which allow correction of collinearity deformations have been summarised in a fairly recent article by Remondino and Fraser [29]. Besides collinearity distortion, lenses are also responsible for other image deformations such as lens flares and chromatic abberation.

Another image deformation artefact is motion blur, which occurs when non-identical views due to camera motion form a single camera image. This can have either of two causes: either the camera moves at such a high speed that multiple views are projected on the recording device during a single shutter opening time window or the camera shutter is not synchronised with the CCD image sensor recording frequency. The first cause can be prevented by slower camera movement speeds or higher recording frequencies and the second cause is related to camera design quality. There exist multiple techniques to reduce motion blur, especially since digital recording devices are more prone to motion blur than analog recording devices [30].

2.2 USARSim Architecture and Technical Aspects

Before the omnidirectional camera simulation method can be explained, certain technical aspects of the simulation environment need to be described. As has been mentioned in section 1.3, USARSim is an open source project built on top of a 3D game engine created by Epic Games™, called the Unreal Engine™2.0.

2.2.1 Unreal Engine 2.0

The Unreal Engine™2.0 (UE2.0) is a complete game development framework which initially formed the basis for the Epic Games release Unreal Tournament®2003. UE2.0 is targeted at mainstream PC's, the Microsoft's Xbox game console and Sony's PlayStation 2 and it contains tools and a code base which enables a game development team to author content for a three-dimensional game environment. Next to 3D rendering capabilities, the engine is designed to offer game developers control over elements such as animation, effects, terrain, textures, physics and networking. The UE2.0 development framework also contains the UnrealEd®Content Creation Tool, a program especially designed to make content for UE2.0 based environments.

UnrealScript

UnrealScript is an object-orientated programming language for scripting in-game content for the Unreal Engine™. It was created by Tim Sweeny, founder of Epic Games, to provide Unreal developers with a built-in programming language especially designed for game programming. The Unreal Engine™implements a so called Unreal Virtual Machine, a concept very similar to the Java Virtual Machine. Code written in Unrealscript, with syntax resembling C++ and Javascript, is compiled into an intermediate platform independent byte code that can be executed by the Unreal Engine™.

The UnrealScript language provides automatic support for meta data, garbage collection and profiling. It also supports persistence with file format backwards-compatibility and supports exposing script properties to level designers in the UnrealEd Content Creation Tool.

.Ini-files

.Ini files are plain text files in a specific format, which provide users control over default values of UnrealScript object configuration variables, without the necessity of recompiling UnrealScript objects. The .ini files are read when the UnrealScript class they are associated with is first loaded.

After that, they can only be written to, but the values from their reading remains in memory. Users can change default values of Unreal objects, like the FOV of the Unreal camera, by modifying the desired entry in the appropriate .ini text file. Hence, .ini files are a valuable component in the USARSim environment, as they also provide a means to easily configure the placement of sensors and operators on the simulated robot body framework.

2.2.2 Models in Unreal Engine 2.0

Unreal Models are the definitions of three-dimensional objects in the Unreal environment. Epic Games defined multiple types of models, as some objects are static (i.e. walls, rocks, car chassis, crates, chairs), some are dynamic (i.e. humans, aliens, animals) and others need multiple definitions (a single weapon requires three models: one for first person view, one for world view, and one for opponent view). This subsection will only describe the first type, called Static Meshes, as the other two types of models have not been used in this project.

Static Meshes are a set of 3D points, vertices and textures. 'Static' in Static Mesh refers to the fact that it cannot be animated, which means the relative position and orientation of its vertices is fixed, though the Static Mesh can be moved, rotated and scaled realtime. These static meshes can be created in the content creation tool UnrealEd, by either constructing a mesh out of basic primitives (i.e. cubes, spheres, cylinders, cones) or by importing 3D models from a compatible commercial modelling application.

Static meshes can be defined to be subject to environment physics, which is handled by Epic Games' Karma Engine®. After Static Mesh creation, the UnrealEd tool provides the means for defining collision boundaries and Karma Primitives (i.e. centre of gravity, weight distribution). As the Karma Physics Engine is proprietary, the exact mechanics behind how the engine uses the Karma Parameters is unknown. Fortunately, there is ample online documentation available and the effect of Karma primitive settings have been investigated by Taylor et al. [31].

2.2.3 Cameras in Unreal Engine 2.0

In the Unreal environment, cameras are implemented to act as an ideal perspective camera (IPC), depicted in Figure 2.4. An IPC is a theoretical concept consisting of three basic elements: a viewpoint, a projection plane and a field of view (FOV). The camera view is constructed by projecting all rays of light within the field of view which pass through the viewpoint onto the projection plane, which lies perpendicular to the viewing direction.

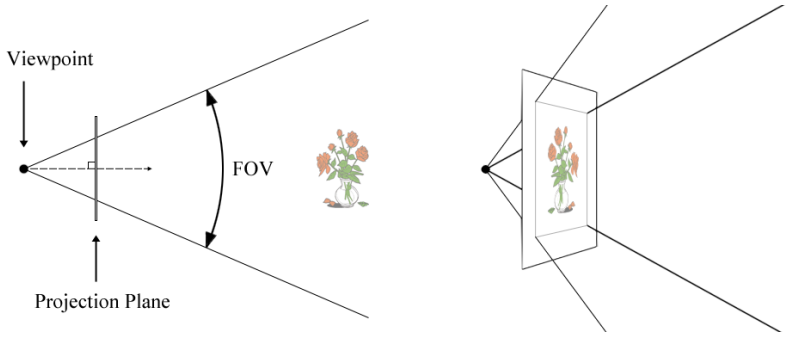


Figure 2.4: A theoretical Ideal Perspective Camera model.

A major difference with the IPC and a real camera is the absence of an optical lens. The aperture of a real camera has a certain diameter allowing multiple rays of light originated from a single point in 3D space to pass through, which requires a lens to converge these multiple rays to a single point on the projection plane to prevent image blurring to occur. As the aperture of the IPC is a single point in space, only a single ray of light can pass through, making the lens obsolete.

Another major difference between the IPC and a real camera is the IPC, unrestricted by the constraints of optical physics, can have its projection plane *in front* of its viewpoint. There are several advantages to this construction. First, by doing so, there is no need to mirror the projection image in post-processing. Second, any non-opaque surfaces positioned between the viewpoint and the projection plane (which is possible due to the fact that the camera has no physical presence) will not be projected on the projection plane. Any object intersecting the projection plane will be observed as being cut in half, an effect which in Computer Graphics is referenced to under the term 'near clipping'.

The IPC, being a mathematical model, does not suffer from the image deformation artefacts described in Subsection 2.1.4. The absence of a lens makes radial lens distortion and decentric lens distortion impossible. Image plane unflatness and in-plane image distortion does not occur as the image plane and light projection is defined by mathematical description. Data produced by an IPC is also void of motion blur as the IPC is capable of capturing image data at a single point in time without the need for an aperture shutter.

2.2.4 UV Texture Mapping in Unreal Engine 2.0

Within the Unreal Engine™, UV Texture Mapping is an important modelling technique for applying images or detail textures to 3D models. The technique could be regarded as a way to precisely

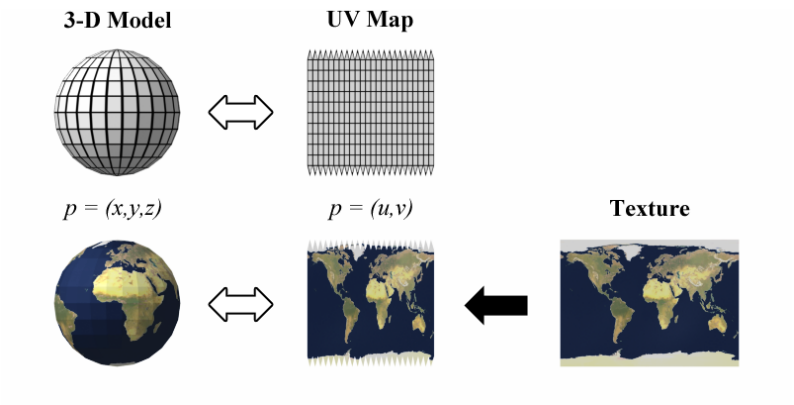


Figure 2.5: UV Texture Mapping Example - mapping a globe.

define how a 2D image should be applied to a 3D surface, such as a map of the world onto a globe.

Figure 2.5 shows an example of how UV Texture mapping works. The term UV Texture Mapping refers to the act of transforming all x, y and z coordinates of a 3D model into 2D u and v coordinates, which will relate to the texture image. This gives each vertex of the 3D model a transformed location on the 2D image, defining what part of the image should be applied to the vertex as a texture in the 3D environment. It also defines what transformation should be applied to the image part, before its projection onto the vertex.

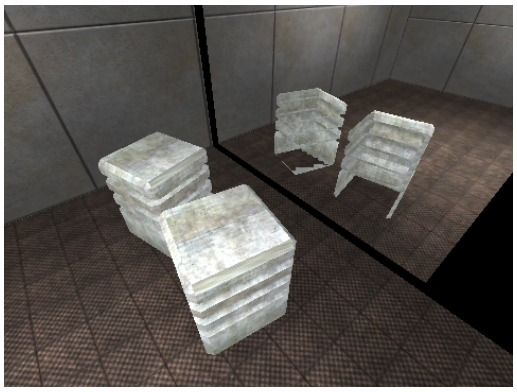
This technique played an important role in creating the omnidirectional mirror reflection image, as will be explained in section 3.1.3.

2.2.5 Mirrors in Unreal Engine 2.0

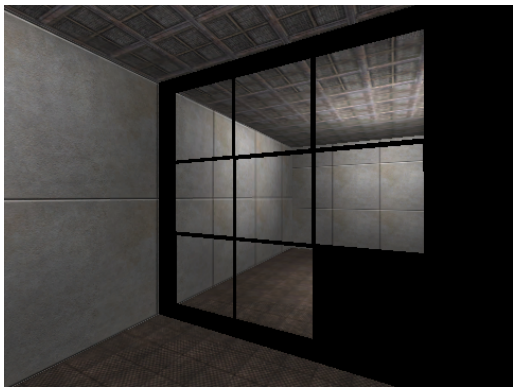
As the catadioptric method is the chosen method for simulation (see subsection 2.1.2), mirrors in the simulation environment are an important aspect of this project. This subsection describes how a mirror can be created in the Unreal environment.

The Unreal Engine™2.0 environment offers two ways to simulate mirrors: The Mirror Surface Property and CameraTextureClients. The name of the first method might suggest otherwise, but neither of these two methods are developed for creating actual perfect mirrors in Unreal Tournament maps. They were included for creating polished surface effects and security camera video displays respectively.

One of the reasons the developers decided not to include actual mirrors in Unreal Tournament maps is related to gameplay. In the Unreal Tournament game, the character mesh of a player is not rendered. The rationale behind this decision is that a player's body is not seen from a first person



(a) Mirrored Static Mesh rendering with inverted z−axis



(b) Eight mirror surfaces rendered simultaneously, while a ninth mirror — present in the lower right corner — is not rendered to prevent system overload.

Figure 2.6: Erronous mirror surface rendering by the Unreal Engine 2.0.

perspective, so the amount of polygons which need to be rendered by the engine can be lowered by not rendering the body at all. This results in the effect that if a player were to stand in front of a mirror, the player would see the reflection of his complete surroundings, but not of his own character's body. Solutions to this could be imagined, but due to computing cost efficiency Epic Games decided not to solve this, but instead discourage the use of mirrors in Unreal Tournament 2004 maps [2].

In the next two sections I describe the two mirror simulation methods, what they were actually designed for and how they relate to the goal of simulating a hyperbolic shaped mirror.

The Mirror Surface Property

Every surface in Unreal Tournament 2004 has a set of properties which define behaviour. One of these is the `Mirror` property, which is generally used to create polished effects on horizontal planar surfaces (i.e. marble floors, waxed wood, still water, etc). The mirror effect is a part of the texture appearance and intended to produce a subtle addition to colour and texture to create the illusion of a polished surface. Though it can be used to create a surface in which mirror images can clearly be seen, the Epic Games development team did not intend this effect to be used in such a way.

This specific mirror effect is based on the assumption that the reflecting surface is completely

[2]There is a way of seeing oneself in a mirror in the Unreal Tournament 2004 game. If the player plays the game from a third person perspective, which can be done by entering the command 'BEHINDVIEW 1' in the Unreal Tournament 2004 console, the mesh will be rendered and visible in the mirror. This is not how the developers intended the game should be played.

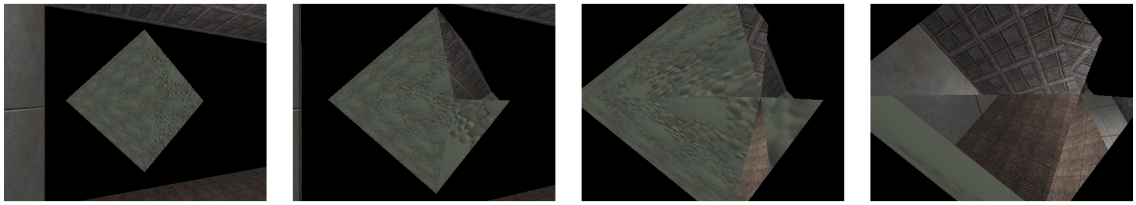


Figure 2.7: A sequence of images displaying the rendering of a mirrored tetrahedron by the Unreal Engine 2.0. In the first image no reflection can be observed and the engine renders the system default texture (green bubbles). As the camera approaches the object, partial mirror reflections are rendered, though the renderings are incorrect and erratic. The last two images show single triangles displaying multiple reflections or a partial single reflection (the rest showing the system default texture). The behaviour is random and highly erratic.

planar: the Unreal Tournament Engine renders the map a second time on the other side of the surface, mirrored in the surface plane normal axis direction.

Surprisingly, reflection rendering of Static Meshes is incorrect. When viewing Static Meshes directly, the Unreal Engine™determines which polygons need to be rendered. If the viewing incidence angle to a polygon is greater than 90° (i.e. the polygon is viewed 'from the back'), the polygon is not rendered. When mirroring Static Meshes, even though the mirrored Static Mesh is viewed from a totally different angle, the Unreal Engine applies the same rendering flags to the polygons in the reflection image. This produces unrealistic results as be seen in Figure 2.6(a), where the 'rear polygons' of the two Static Meshes are not rendered in the mirror reflection, even though these polygons are at the 'front'.

A second limitation is imposed because rendering a map multiple times greatly influences the number of frames per second that the engine is able to render the environment. To prevent system overload, Epic has constrained the amount of simultaneously rendered mirror surfaces to a limit of eight, which can be seen in Figure 2.6(b). A ninth mirror surface is present in the lower right corner, though the Unreal Engine™only renders this surface as soon as one of the other mirrors is completely out of view.

Experimentation with the `Mirror` property on non-planar surfaces resulted in erratic reflection rendering behaviour. A non-planar surface consists of multiple polygons with non-aligned normals and the rendering engine seemingly took a random normal of a single polygon to create the complete reflection. Also, the reflection is not shown on all mirror surface polygons which resulted in undefined texture projections or triangular shaped gaps. Seemingly, this effect also occured when different non-aligned and non-adjacent surfaces belonged to a single object. Figure 2.7 shows the erratic behaviour of the mirror property when applied to the surfaces of a tetrahedron,

where the green bubbled texture is the system default texture in the Unreal Environment.

Because of the effects and severe limitations described above, the `Mirror` surface property cannot be used for simulating the parabolic shaped mirror.

CameraTextureClients

There is another way of simulating mirrors, using the CameraTextureClient class which Epic created for including security cameras and display monitors in Unreal Tournament maps. Within a CameraTextureClient, one can define certain Actors class instances to function as a Virtual Camera Actor, whose camera view will then be projected on a texture which also defined within that same CameraTextureClient instance. Map designers can use this construction to define cameras which can be monitored from a remote location. In this setup the following components are involved:

The Camera Actor - Any instance of the Actor class having a location and orientation can function as a Camera Actor. It is its location and orientation on which the virtual camera view will be based. The preferred Actor for this task is the Emitter, as it is one of the most basic classes in the Unreal Environment that can be pointed in a certain direction.

Scripted Texture - The texture on which the camera view is projected must be a Scripted Texture, because a Scripted Texture is the only type of texture in the Unreal Environment which can be drawn upon in real time. This is necessary as we want to be able to see a real time representation of the Virtual Camera View displayed on this texture.

The CameraTextureClient - The CameraTextureClient functions as the bridge between an Actor used as a Virtual Camera and a Scripted Texture which will display the image. It has the following properties:

- CameraActor Name - A reference to the Actor functioning as camera
- DestTexture Name - A reference to the Scripted Texture on which the camera view is projected
- FOV - Defining the Field Of View of the virtual camera
- RefreshRate - The amount of camera view frames shown per second (fps)

A CameraTextureClient can be used for simulating mirrors by placing an invisible Camera Actor behind the associated Scripted Texture. The virtual camera should mirror the spectator's viewing point in the mirror surface plane, so it can record the reflection which the viewer is supposed to see. Recording the proper reflection is not trivial though, as the distance between an observer to

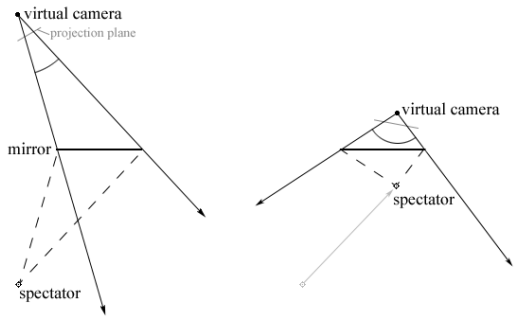


Figure 2.8: A spectator's influence on virtual camera position, FOV and projection plane.

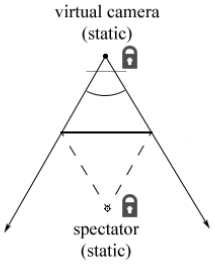


Figure 2.9: A static spectator requires a static virtual camera position, FOV and projection plane.

the mirror surface should influence the required camera FOV and different angles of observation require a different virtual camera locations, as can be seen in figure 2.8. Also, if the spectator view is not perpendicular to the mirror surface, the projection plane of the virtual camera will not be parallel to that mirror surface. This means that the camera image needs to be transformed before it can be projected onto the scripted texture and the appropriate transformation matrix depends on the spectator's location.

Most important to this project is that this method can be used to create a hyperbolic convex shaped mirror for an Omnidirectional Camera. The observation point from which the mirror is seen, which is the position of the recording camera below the mirror facing upwards, is completely static in relation to the mirror. This means that the required location, orientation and FOV of the virtual camera and the UV-mapping of the scripted texture need to be defined only once (see figure 2.9). If it is done in such a way that the proper reflection is displayed, the hyperbolic convex shaped mirror for the catadioptric omnidirectional camera can be simulated. This will be described in subsection 3.1.3.

2.2.6 USARSim Mission Packages

As robots, sensors and actuators in the USARSim environment generally consist of multiple interacting Static Meshes (i.e. a robot arm consists of an arm with a grappling device, all combined at different joints), the USARSim Mission Package (UMP) concept was introduced to organise and control these multiple parts. The omnidirectional camera model is structured as a UMP.

The structure of UMPs is quite simple. A UMP can be seen as a tree-structure of linked Static Meshes. The UMP instance is defined by a MisPkgInfo class, which refers to a .ini-file describing

which Static Mesh lies at the root and what Static Meshes compose the rest of a tree structure. Parents, relative locations and type of relative movement (i.e. revolute, scissor and prismatic) are defined in this .ini-file. For this concept to work, all USARModels which are used in a UMP require an UnrealScript subclass of the MisPkgLinkInfo class, which contains a reference to the .ini-file defining their place in (possibly multiple) UMP's.

Because this organisation of Static Meshes is defined in an .ini-file, one of the great benefits of this concept is the relative low complexity of changing and modifying all these aforementioned properties. Also, partly due to the tree-structure design of UMPs, any Unreal actor can be attached to any link within a UMP. Cameras and other sensors can be attached to movable elements, usually greatly improving their effectiveness. Also, the location of a large amount of sensors linked to a single UMP can be adjusted by simply moving the UMP.

UMP link movement can be controlled real time by sending appropriate string commands to the simulation environment. For example, a robot arm can be lifted at the shoulder by sending a particular string message, which will result in the arm lifting at the shoulder joint until it reaches its maximal revolution angle, defined in the UMP .ini-file. Though the omnidirectional camera does not contain any moving elements, the fact that an UMP makes configuration of multiple static meshes less complex makes it the ideal method to organise the simulation model meshes.

3 Method

This chapter describes all methods employed in simulating and validating the omnidirectional camera. First, a description of the real omnidirectional camera on which the 3D model is based will be given before the 3D model development process in the Unreal Environment will be described. In this section a lot of attention will go to the simulation of the parabolic mirror surface. All elements which had to be programmed in UnrealScript will be discussed as well, revealing the simulation architecture. Finally all techniques used for validating the simulation model are explained for completeness.

3.1 The Simulated Camera in the 3D Environment

The body of the Omnidirectional Camera was modelled for the Unreal Tournament environment as a Static Mesh (Subsection 2.2.2), modelled in the 3D graphics program Lightwave®. The dimensions of the model are defined by size measurements of an existing omnidirectional camera described in the next subsection. The 3D model textures are my own design.

3.1.1 The Real Omnidirectional Camera

The virtual omnidirectional camera model is based on the catadioptric omnidirectional vision system employed by the Intelligent Systems Lab Amsterdam (ISLA) group, University of Amsterdam. It consists of a Dragonfly®2 camera made by Point Grey Research Inc. and a Large Type Panorama Eye®made by Accowle Company, Ltd.

The Dragonfly 2 camera is an OEM style board level camera designed for imaging product development. It offers double a frame rate of up to 30 FPS, auto-iris lens control and on-board colour processing [32]. It has the following specifics:

Figure 3.1: IAS Group Omnidirectional Camera, consisting of a Dragonfly®2 camera and a Panoramic Eye®mirror.

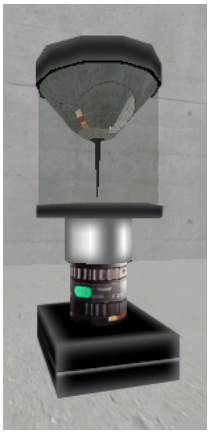


Figure 3.2: Unreal Static Mesh of Simulated Omnidirectional Camera

- Sensor: Sony 1/3" progressive scan CCDs, Color/BW
- Resolution: 648x488, 1036x776 or 1296x964
- Frame Rates: 60 (BW/COL), 30 (HIBW/HICOL), 20FPS (13S2)
- Dimensions: 64x51mm (board), 75x65x25mm (boxed)
- Interface: 6-pin IEEE-1394a 400Mb/s interface

The Large Type Panorama Eye is a 105mm high, 76mm diameter see-through plexiglass cylinder containing a hyperbolic convex shaped mirror[33]. The manufacturer specifies the shape of the mirror to be defined by the equation

$$\frac{r^2}{0.2333} - \frac{z^2}{1.1357} = -1 \tag{3.1}$$

where r defines horizontal distance to the mirror centre and z defines height. This concurs with Baker and Nayar's general solution of the fixed viewpoint constraint (see subsection 2.1.3) as it equals equation 2.1 with $k = 11.546$ and $c = 2.321$, though z values have to be corrected by 1.161 as Point Grey Research Inc. did not use the camera viewpoint as origin for its surface formula.

3.1.2 The 3D Model

The Static Mesh was modelled to scale in Lightwave, created by Newtek®, Inc. and imported to the USARSim environment with the Unreal Editor tool. The mirror itself is modelled as a separate Static Mesh, as multiple mirror surfaces were constructed to explore the influence of the mirror

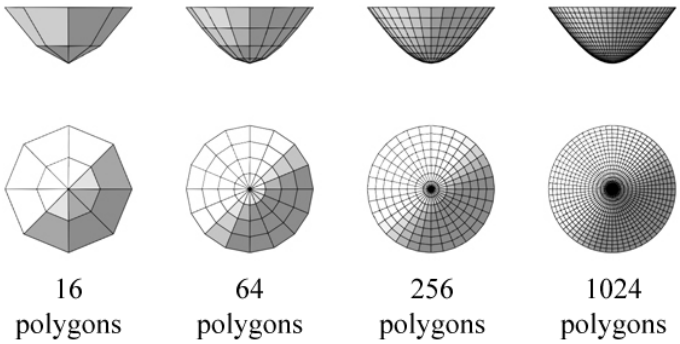


Figure 3.3: Four mirror surfaces with different polygon counts.

polygon[1] count on system performance and simulation quality. Hence, the 3D Omnidirectional Camera simulation is represented by two Static Meshes, a mirror and a single model representing a Dragonfly 2 camera and a Large Type Panorama Eye without the mirror. The mesh is modelled in such a way that the mirror's centre of projection is placed at the origin so that placement of the camera in the USARSim environment is done by reference to this point.

Four mirror Static Meshes were created, defined by a surface based on Equation 3.1, each mirror having a different polygon count. The mirror surface with the smallest number of consists of 16 (2^4) polygons: when viewed from the bottom, the mirror is divided in eight circular sectors, each divided in half. Three additional mirrors were created, doubling the amount of sectors and the amount of divisions with each step, resulting in surfaces consisting of 64 (2^6), 256 (2^8) and 1024 (2^{10}) polygons. All four mirror surfaces are depicted in Figure 3.3.

Importing the mirror surfaces to the USARSim environment increases the number of polygons in the mesh, as the USARSim environment utilises only triangulated polygons. As a result, the Unreal Editor tool requires all polygons with more than 3 vertices to be triangulated before being able to import them. This nearly doubles the amount of polygons in the surface (all 4-vertex-polygons are cut in half to create two three-vertex-polygons). There is a second reason why the Static Mesh has an increased amount of polygons, though this will be discussed in Subsection 3.1.3. The resulting numbers of polygons in the camera housing and the mirror surface Static Meshes are contained in Table 3.1.

[1] In math, a polygon is defined by a plane figure that is bounded by a closed path or circuit, composed of a finite sequence of straight line segments. In computer graphics a polygon is surface face, usually triangular, used to construct an object's surface. Throughout this thesis the latter meaning is intended.

	Housing	Mirror 16	Mirror 64	Mirror 256	Mirror 1024
points	168	45	117	353	1201
vertices	438	80	256	892	3292
3-vertex-polygons	300	40	144	544	2096
UV Texture Mappings	1	5	5	5	5

Table 3.1: 3D Omnidirectional Camera model specifics

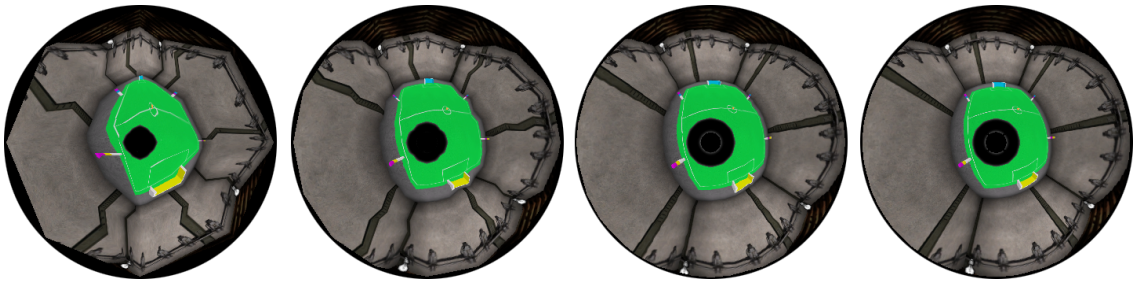


Figure 3.4: 360×360 Pixel mirror surface view comparison of omnidirectional image data obtained in an USARSim map containing an Aibo soccer field. From left to right: 16 polygons, 64 polygons, 256 polygons and 1024 polygons.

Influence of Mirror Polygon Count on Omnidirectional Image Quality

As the UV-texture mapping method linearly interpolates texture placement on a single polygon, improper distortions occur when mirror polygons are too big. Figure 3.4 shows omnidirectional images obtained from mirror surfaces described in Subsection 3.1.2. The first two reflection images, produced by the mirrors based on the 16 and 64 polygon surfaces respectively, show heavy distortion of the reflection image due to the linear interpolation. The data provided by these surfaces does not relate to a proper reflection simulation. The 256 polygon surface seems to provide a proper reflection image, though Figure 3.5 shows a detail of the two surfaces with 256 and 1024 polygons respectively, which depicts a slight quality difference: the 256 polygon mirror still shows minor distortion, jagging of a straight line, which in the 1024 polygon image mirror does not occur.

Physical Properties and Karma Primitives

The physical properties of the simulation model, i.e. collision volume, weight, friction and restitution, are simulated by the Karma Engine®. As calculating the point where two different objects collide is a complex and very demanding task, simplified representations of Static Mesh physical structures, called Karma Primitives, can be defined to speed up the process. For the omnidirectional camera a single eight sided cylinder Karma Primitive was defined which fits tightly around the Static Mesh. The other physical properties are defined in the appropriate UnrealScript Class,

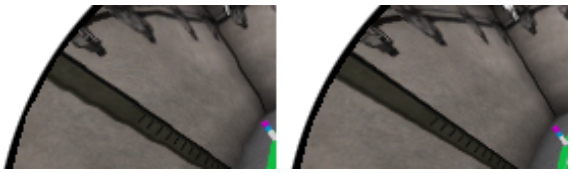


Figure 3.5: Mirror surface detail, left: 256 polygons, right: 1024 polygons. As can be seen the 256 polygon mirror shows distortion of linearity where the 1024 polygon mirror does not.

which will be explained in Section 3.3.1.

3.1.3 Mirror Simulating Method

Based on the Single Viewpoint Constraint (subsection 2.1.1), and the possibility of using CameraTextureClients (subsection 2.2.5) and UV Texture Mapping (subsection 2.2.4), a virtual parabolic convex mirror can be simulated. First, I will explain the basics of the simulation technique before explaining what elements are necessary for the simulation. Finally, I will explain why UV-mapping is needed to achieve a proper simulation of the mirror and how the UV-coordinates were deduced.

Virtual Cameras and the Single Viewpoint Constraint

As explained in 2.1.1, the SVC defines projection images which would be equivalent to images acquired by a perspective camera with its aperture placed in the centre of projection of the vision system. Using CameraTextureClients the same effect can be achieved: Virtual perspective cameras can be placed with their effective viewpoint in the centre of projection of the mirror and the images they acquire can be projected on Scripted Textures which are UV mapped onto the mirror.

Note that when a virtual camera is placed behind a mirror surface, it does not see the mirror as the mirror vertices are single-sided. The term single-sided means that a vertex is only rendered when the viewing angle lies between $0°$ and $90°$. As the virtual camera placed behind the mirror is looking at the its surface at an angle higher than $90°$, the vertices are not rendered and the camera sees 'through' the mirror.

Placement of the Virtual Cameras

To obtain image data for the complete mirror surface, five virtual cameras with a $90°$ FOV are necessary. They all need to be placed on the Centre of Projection in $90°$ angles of each other. Four cameras are rotated parallel to the horizontal plane, one in each wind direction N, E, S, W and the final camera is aimed directly downwards, parallel to the normal line of the horizontal plane.

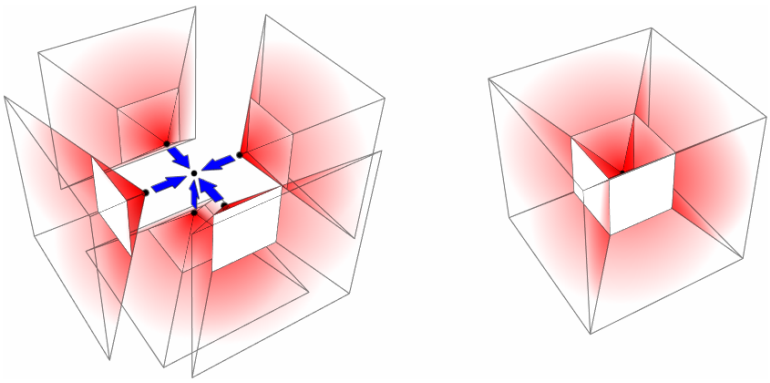


Figure 3.6: Placement of the 5 virtual cameras.

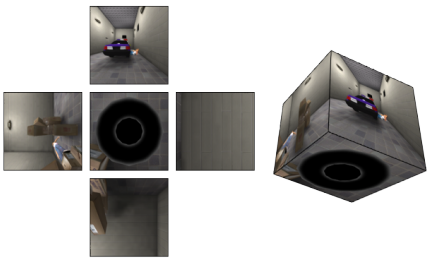


Figure 3.7: A five camera cube mapping of an environment.

By placing them in such a fashion, the projection planes of the virtual cameras form five sides to a cube, as depicted in figure 3.6. Being placed in such a fashion, there is no gap between the edges of the projection planes nor do they overlap. This means that everything not located at a angle above the horizontal plane higher than 45° from location of the cameras is perceived, and it is perceived by a single camera. Projecting these five camera views on five sides of a cube, results in what is called a cube mapping of an environment, depicted in 3.7.

A sixth virtual camera could be added to obtain image data in the area beyond the 45° upward angle, though there were multiple reasons not to do so. First, catadioptric omnidirectional cameras rarely provide image data for this region. The Super-Wide-View type Panorama Eye mirrors provide the largest angle of any mirrors made by Accowle Company, Ltd., which approximates 46° . Second, the image data obtained in these angles is in many cases quite less valuable compared to the image data obtained from the more lower angles, so considering computer performance I decided not to include another CameraTextureClient, virtual camera and Scripted Texture.

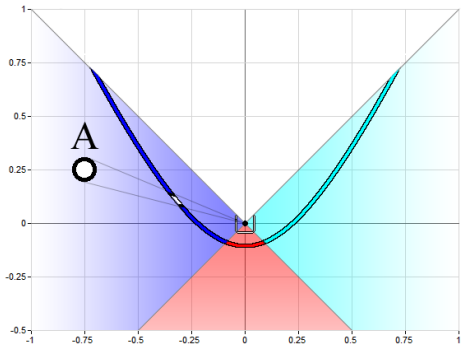


Figure 3.8: 2D visualisation of a point-projection of three camera projection planes onto the mirror surface. A point-projection of object A is depicted as well.

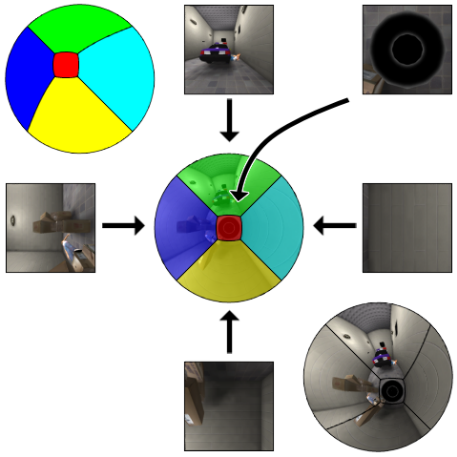


Figure 3.9: UV mappings for the virtual camera views.

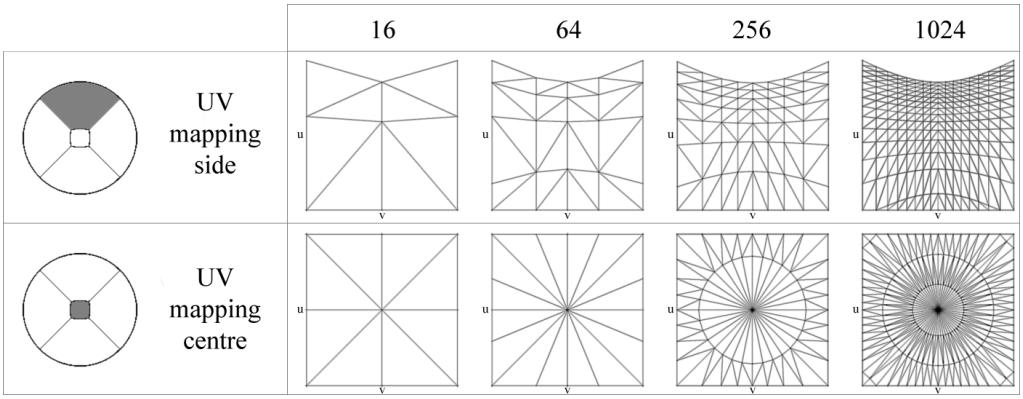


Figure 3.10: UV mappings of the mirror surfaces, resulting in a point-projection of camera views from the centre of projection.

UV Texture Mapping the Scripted Textures

Considering a virtual camera view, the mirror surface on which it is projected and the relative positions of both, a proper UV-mapping was deduced. In this setup the camera is directed at the back of the mirror surface and this mirror surface should exactly display what the camera would see 'through' the mirror. To achieve this, the camera view image captured from the camera projection plane must be point-projected on the mirror surface from the virtual camera location. A two-dimensional representation of this concept is depicted in Figure 3.8. When the proper UV-mappings are deduced, the camera views can be applied to the mirror surface as depicted in Figure 3.9.

Obtaining the UV projection mappings was done by backprojecting the mirror surfaces on the camera projection plane. The UV projection mappings obtained by this method in the Lightwave

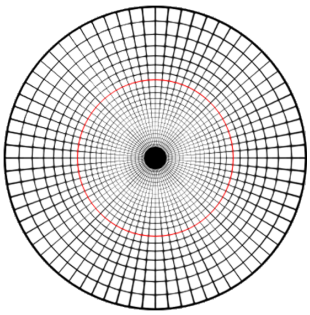


Figure 3.11: A 360×360 pixel simulation mirror rendering of incidence angles with a five degree difference. The red line depicts the 90° incidence angle.

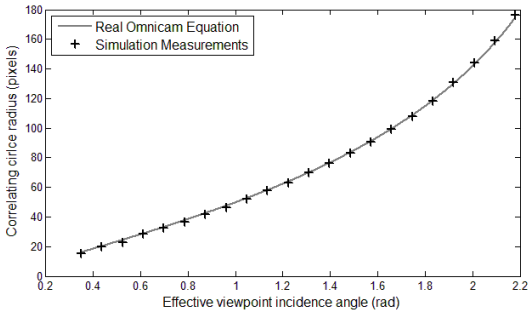


Figure 3.12: A plot of the real omnidirectional camera Equation 3.2 and the simulation circle radii measurements in Figure 3.11.

graphics tool are depicted in Figure 3.10.

In this setup, the mirror surface is mapped with five different textures of which the location has to be precise. In the Unreal environment, a triangulated polygon can only be wholly mapped by a texture, not partially. As the initial mirror surface mesh polygon edges created in Lightwave do not completely coincide with the required texture mappings, some polygons needed to be subdivided. This is the second reason why the actual Static Mesh polygon count is higher than the initial mirror surface design meshes created in Lightwave.

3.2 UV Texture Mapping Verification

To verify the UV mapping, omnidirectional image data is compared to omnidirectional image transformation theory described in [34]. For a real catadioptric omnidirectional camera satisfying the SVC using a parabolic mirror, the relation between the effective single viewpoint incidence angle θ and the radius of the correlating circle in the omnidirectional image r_i is known to be defined by the following equation

$$r_i = sin(\theta)\frac{h}{(1 + cos(\theta))} \tag{3.2}$$

where the origin of the image coincides with the 0° incidence angle and h defines the radius of the 90° circle. As the shape of the mirror defines the location of this 90° circle, h is directly related to c and k in Equation 2.1.

The proper UV mapping of the simulated mirror surface produces omnidirectional image data

displaying the same relation between θ and r_i as defined by Equation 3.2. A simulation rendering of the relation between θ and r_i is depicted in Figure 3.11. The circle radii in this figure were compared to Equation 3.2, plotted in Figure 3.12, demonstrating a mean squared error of less than a single pixel.

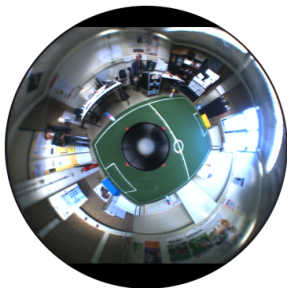
3.2.1 Image Transformations

Omnidirectional image data provided by a parabolic catadioptric omnidirectional camera is hardly intuitive to a human observer as the image data shows the robot's surroundings upside down, which could makes recognition more complicated, and due to reflection the environment is mirrored. The image also shows counter-intuitive perspective distortion. It has already been shown that image post-processing can greatly improve the readability of omnidirectional data to a human observer. Also in the Artificial Intelligence field of science a large amount of automated vision techniques assume perspective projection, which makes the transformation of omnidirectional image data to geometrically correct perspective images highly desirable in some cases. Two methods used to transform real omnidirectional image data are described next: the first to create geometrically correct perspective images and the other to improve omnidirectional data readability to a human observer.

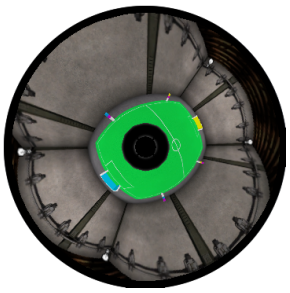
Based on the equations described in this subsection omnidirectional image data can be transformed or projected into more intuitive and illustrative image data. To demonstrate this, the omnidirectional views in Figure 3.13 have been transformed. Figure 3.13(a) shows a real view of the ISLA Aibo soccer field, while the other three show a simulated omnidirectional view. Note that, as the system configuration is static over time, a lookup table for each transformation can be computed beforehand, reducing the image transformation computational costs during an application of real-time image conversion.

Bird-Eye View

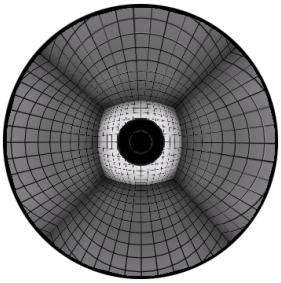
Shree K. Nayar describes a direct relation between a location in a 3D environment and the location in the omnidirectional image where this point can be seen if nothing obstructs the view [34]. The correspondence between a pixel in the omnidirectional image, $p_{omn} = (x_{omn}, y_{omn})$, and a location



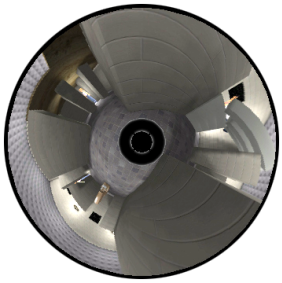
(a) ISLA Aibo soccer field



(b) the SPQRSoccer2006 map



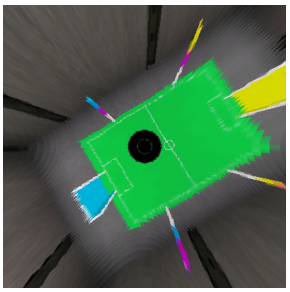
(c) a tiled room



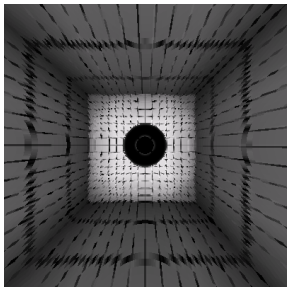
(d) the Mapping map

Figure 3.13: Omnidirectional views obtained at ISLA and in the USARSim environment.

(a) translation of Figure 3.13(a)



(b) translation of Figure 3.13(b)



(c) translation of Figure 3.13(c)

(d) translation of Figure 3.13(d)

Figure 3.14: 400×400 pixel bird-eye translations of Figures 3.13 with $z_w = 40$

in the 3D world, $p_w = (x_w, y_w, z_w)$, also defined in pixels, is defined by the following equations

$$\theta = \arccos \frac{z_w}{\sqrt{x_w^2 + y_w^2 + z_w^2}}, \quad \phi = \arctan \frac{y_w}{x_w} \tag{3.3}$$

$$\rho = \frac{h}{1 + \cos\theta} \tag{3.4}$$

$$x_{omn} = \rho \sin\theta \cos\phi, \quad y_{omn} = \rho \sin\theta \sin\phi \tag{3.5}$$

where h is the radius of the circle describing the 90° incidence angle on the omnidirectional camera effective viewpoint. These equations can be used to construct perspectively correct images based on omnidirectional camera data by translating 3D projection plane pixel locations to omnidirectional pixel locations.

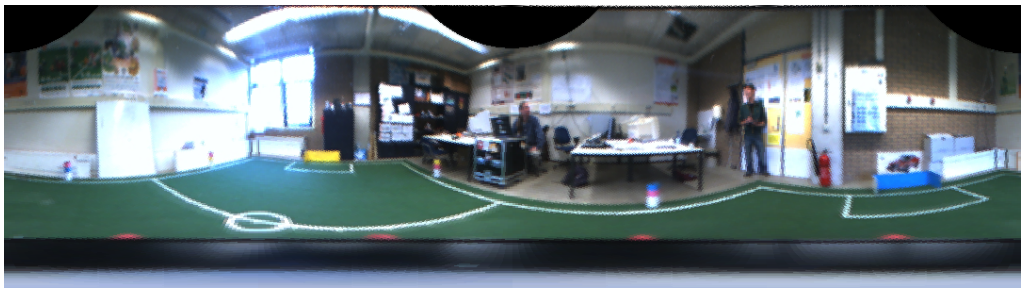
A frequently used omnidirectional data projection is the perspectively correct bird-eye view. This method projects the omnidirectional image data onto a plane perpendicular to the vertical axis of the omnidirectional system. As this plane is parallel to the ground supporting the robot, this view instantly provides an overview of the robot's surroundings, seen top down, hence the name of this view.

Figure 3.14 shows 400 × 400 pixel bird-eye images obtained using Equations 3.3 to 3.5 with $z = 40$. Figure 3.14(c) demonstrates correct perspective, showing parallel straight lines on the floor and all four walls. The usefulness of a bird-eye view is quite appearant in Figure 3.14(a) and 3.14(b), as this transformation very effecively shows the precise location and orientation of the robot on the Aibo soccer field. These figures also show that there is close agreement between the real and simulated sensor.

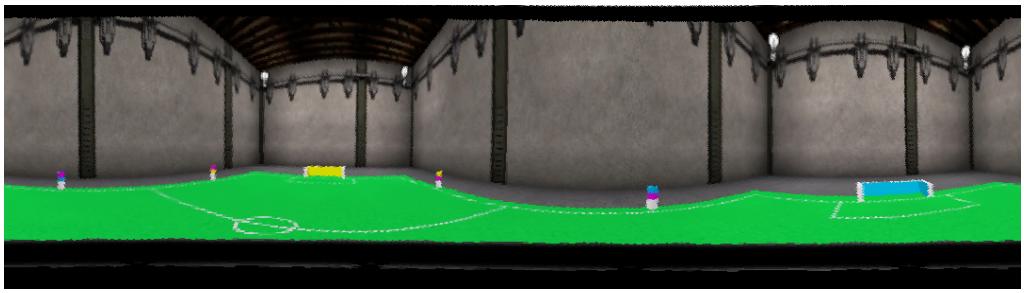
Panoramic View

Grassi and Okamoto describe a simple method to create panoramic views by directly mapping pixels from polar to rectangular coordinates [35]. This method does not relate to a linear projection of image data on a 3D surface thus it creates perspectively incorrect panoramic images. Even though they are perspectively incorrect, in Subsection 3.2.1 it is shown that these panoramic images can be very useful nonetheless.

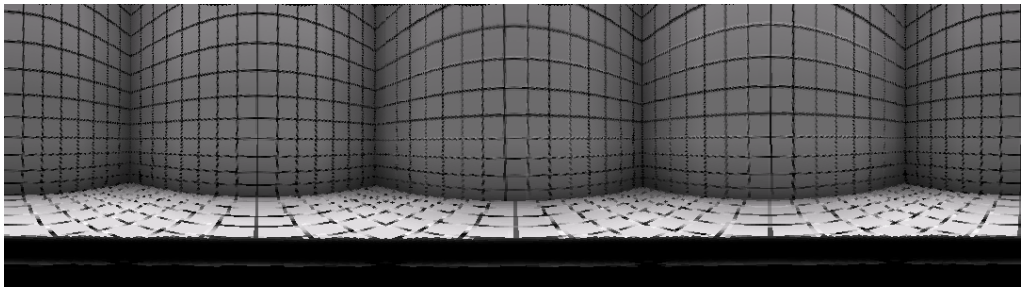
The polar to rectangular coordinates correspondence between a pixel in the omnidirectional image $p_{omn} = (x_{omn}, y_{omn})$ and a pixel in the panoramic image $p_{pan} = (x_{pan}, y_{pan})$ is defined by



(a) translation of Figure 3.13(a)



(b) translation of Figure 3.13(b)



(c) translation of Figure 3.13(c)

(d) translation of Figure 3.13(d)

Figure 3.15: Panoramic translations of Figures 3.13

the following equations

$$x_{omn} = \left(r_{omn} \cdot \frac{x_{pan}}{X_{pan}}\right) \cos\left(2\pi \cdot \frac{y_{pan}}{Y_{pan}}\right) \tag{3.6}$$

$$y_{omn} = \left(r_{omn} \cdot \frac{x_{pan}}{X_{pan}}\right) \sin\left(2\pi \cdot \frac{y_{pan}}{Y_{pan}}\right) \tag{3.7}$$

where r_{omn} is defined by the radius of the omnidirectional data in pixels and where X_{pan} and Y_{pan} define the width and height of the desired panoramic image respectively.

Figure 3.15 shows 1080 × 300 pixel panoramic images produced using Equations 3.6 and 3.7. As can be seen in Figure 3.15(c), all lines in the environment parallel to the vertical axis of the omnidirectional camera appear as straight vertical lines in the panoramic image. Figure 3.15(c) also demonstrates the expected vertical perspective distortion as the horizontal lines on the walls appear unevenly spaced in the panoramic image while they are evenly spaced in the environment. Again, Figures 3.15(a) and 3.15(b) show that there is close agreement between the real and simulated sensor. Regarding the usefulness of this image transformation with respect to USAR operations, Figure 3.15(d) shows three victims (indicated by circles) and their horizontal location in the panoramic image directly correlates to the azimuth angle at which the victims are situated in relation to the robot. Locating these victims can either be done by a human spectator or by using automated techniques. One of these methods, which I have implemented for the Atlanta RoboCup Rescue League as part of the UvA Rescue Team 2007 [36], is a colour histogram based pixel classifier. As this method is used to detect landmarks on the Aibo soccer field for validating the simulation model, the pixel classifier will be explained in Subsection 3.4.1. Note that in this particular situation, a regular camera would not be capable of providing information about 3 victims within a single image because of its limited FOV.

3.3 UnrealScript Programming

In order to integrate the simulation design into the USARSim environment, some programming needed to be done in the Unreal Engine™scripting language UnrealScript. Separate classes for each Static mesh needed to be implemented and new classes had to be created for simulating the mirror surface. Finally, the USARSim KRobot class had to be modified so it was able to mount these new classes whenever a robot is spawned. This section describes what classes were developed and discusses the choices made during the design process.

3.3.1 USARModel Classes

The USARModel classes define the objects in the USARSim environment which relate to a Static Mesh. As has explained, the 3D Omnidirectional Camera object is defined by two Static Meshes, a mirror and a single model representing the Dragonfly 2 camera and a Large Type Panorama

Eye. As each Static Mesh requires its own UnrealScript class, in total five USARModel classes were created: four classes for each mirror Static Mesh and one class for the Dragonfly/Panoramic Eye Static Mesh. Within these USARModel classes certain properties (i.e. scaling, whether it should cast a shadow, Karma properties, etc.) can be defined.

For the mirror classes a property called `bunlit` was set to `true`, as special lighting conditions should be applied to a mirror. An uneven or granular surface is visible to an observer because of the diffuse reflection of light (reflected light will be evenly spread over the hemisphere surrounding the surface). Diffuse reflection is also the reason for the occurrence of shadows when something non-opaque comes between the surface and the source of light. However, if the surface is a perfect mirror no shadows can be cast because of specular reflection (the angle of incidence equals the angle of reflection). As the simulated mirror surface is a Scripted Texture, the effect of a shadow cast upon this texture is undesirable: the mirror should 'reflect' light with the same intensity as it is perceived by the Virtual Camera. Fortunately, this effect can be achieved by setting the `bunlit` property of the mirror classes to `true`, as by doing so the mirror mesh will always be rendered as if fully lit and is therefore void of shadows.

The Karma properties of the Static Mesh are defined here as well. As has been said in Subsection 2.2.2, the exact mechanics behind how the engine uses the Karma Parameters is not known. Online Karma documentation describes the Karma property `KMass` to define mass density[2], though M. Zaratti published an online report demonstrating that this is not the case: `KMass` is demonstrated to define total object mass[3]. Due to the lack of a clear description between the real camera mass and the `KMass` variable, this property of the omnidirectional camera is based on `KMass` values of other USARModel sensor and actuator classes and the test results from [31].

As the camera is almost completely non-elastic the `KRestitution` (defining object elasticity) property is set to zero. The `KFriction` (defining surface friction) property was more difficult to determine as the omnidirectional camera has both smooth plexiglass and rougher polymers as surfaces. The most common value encountered in a large amount of actuator classes, 0.5, is chosen to be used.

3.3.2 The USARCameraTextureClient Class

CameraTextureClients are an important part of the simulation method (Subsection 2.2.5), though the standard CameraTextureClient class of the Unreal environment could not be used for this

[2] http://wiki.beyondunreal.com/Legacy:KarmaParams, last accessed: May 30th, 2008.
[3] http://digilander.libero.it/windflow/eng/Reports/mass_test.htm, last accessed: May 30th, 2008.

simulation due to two major limitations. For this project five USARCameraTextureClient classes, subclasses of the standard CameraTextureClient, had to be created.

The first limitation of a standard CameraTextureClient is the fact that it cannot be dynamically spawned in, or deleted from a running Unreal environment. As robots (and therefore omnidirectional cameras mounted on these robots) spawn dynamically, this limitation was removed by setting USARCameraTextureClient properties `bNoDelete` and `bStatic` to `false`. The second limitation is the quite surprising fact that a Scripted Texture, onto which the CameraTextureClient will project the camera images, cannot be assigned dynamically to a CameraTextureClient. As five Scripted Textures are required for a proper simulation (see Subsection 3.1.3), five USARCameraTextureClient classes had to be created with each their own default `DestTexture` property, defining their appropriate Scripted Textures.

3.3.3 The USAREmitter class

The standard Emitter class in the Unreal environment is used as a container class for multiple Particle Emitter classes, which are responsible for creating a large variety of effects like smoke, fire, falling water and sparks. As the Emitter class itself only acts as an invisible container it is one of the most simple classes in the Unreal environment. It is possible to assign a location and an orientation to the Emitter, making it the ideal candidate for taking up the role as a Virtual Camera Actor for a CameraTextureClient. The standard Emitter class has no orientation by default and it also cannot be dynamically spawned in, or deleted from a running Unreal environment. Therefore a USAREmitter class was created with default properties `bDirectional` set to `true` and, as with the USARCameraTextureClient, `bNoDelete` and `bStatic` set to `false`.

3.3.4 The OmniCam USAR Mission Package

Considering the fact that the simulated omnidirectional camera consists of multiple elements, the camera is designed to be mounted as an UMP (Subsection 2.2.6). A subclass of the MisPkgLinkInfo class was created for all five developed USARModel classes and a single UMP was defined within an appropriate .ini-file. Note that by doing so, one of the four mirror static meshes can be selected by simply modifying the .ini-file in which the UMP is defined. Also, all the elements involved in the omnidirectional camera simulation process can be linked to the Omnidirectional Camera UMP, which makes (modifications to) the placement of the complete omnidirectional camera a simple task.

3.3.5 KRobot Class Modifications

The USARSim KRobot class, the class defining the general robot actor in USARSim, had to be modified such that the new USARCameraTextureClient and USAREmitter classes could be mounted onto a robot. Code was added which is able to spawn and delete these classes in a similar fashion like the KRobot class was already able to do with sensors, cameras and UMPs. For integration of the omnidirectional camera simulation implementation to the KRobot class no complex design decisions had to be made.

3.4 Validation of the Simulation Model

If a simulation model is used for scientific or educational purposes, it is important that the model is validated (in Section 5.1 more attention will be given to this claim). Validation is not to be confused with verification, as verification is a quality check ensuring that programming and implementation operates as intended. Validation, however, is usually defined to mean substantiation that a computerized model within its domain of applicability possesses a satisfactory range of realism consistent with the intended application of the model [37]. Related to USARSim, Carpin et al. chose to do a quantitative validation of simulation accuracy, with the goal to provide quantitative indices that indicate to which degree it is possible to extrapolate results obtained in simulation. They have implemented four different image processing algorithms that require fine tuning of several parameters. The parameter fine tuning phase has been performed exclusively in simulation and then the same algorithms have been run on real world images, to outline similarities and differences in performance [38].

For validation of the omnidirectional camera simulation model, the following setup has been chosen: similar to Carpin's approach, an assessment of sensor accuracy is given, indicating to which degree it is possible to extrapolate results obtained in simulation. I have implemented a two-phase self localisation algorithm which requires the fine tuning of multiple parameters. The parameter fine tuning phase has been performed exclusively in simulation and then the same algorithms have been run on real world omnidirectional image data, to outline similarities and differences in performance.

3.4.1 The Colour Histogram Pixel Classifier

The first phase of the self localisation algorithm is image based landmark detection. Landmark detection was performed using a very common statistical model, the colour histogram, to create a pixel classifier. Jones and Rehg demonstrate a successful application of this method for skin detection in the RGB colour space, arguing that histogram classifiers compare favourably to the more expensive but widely-used Gaussian mixture densities [39].

The histogram classifier requires two training datasets for a single classifier: one containing images displaying the desired object and another containing images in which the object is absent, called the positive dataset and the negative dataset respectively. Using these two training sets two colour models are created, using a three-dimensional histogram with a certain amount of bins per channel in the RGB colour space. The histogram counts were then converted into a discrete probability distribution $P(\cdot)$:

$$P(rgb) = \frac{c[rgb]}{T_c} \tag{3.8}$$

where $c[rgb]$ gives the count in the histogram bin associated with the RGB colour triplet $\{r, g, b, \}$ and T_c is the total count obtained by summing the counts in all of the bins. Given the positive and negative histograms, the probability that a given colour value belongs to the positive and negative classes can be computed:

$$P(rgb|positive) = \frac{p[rgb]}{T_p}, \quad P(rgb|negative) = \frac{n[rgb]}{T_n} \tag{3.9}$$

where $p[rgb]$ is the pixel count contained in bin rgb of the positive histogram, $n[rgb]$ is the count from the negative histogram, and T_p and T_n are the total counts contained in the positive and negative histograms respectively. Given positive and negative histogram models a positive pixel classifier was constructed, defined through the standard likelihood ratio approach. A particular rgb value is labelled positive if

$$\frac{P(rgb|positive)}{P(rgb|negative)} \geq \Theta \tag{3.10}$$

where $0 \leq \Theta \leq 1$ is a threshold which can be adjusted to trade-off between correct detections and false positives. Using this positive pixel classifier, images were scanned for the presence of the desired coloured object. For each class, all pixels in the image are labelled either positive or negative, resulting in a $m \times n$ matrix of 0 and 1 values. For research performed in this thesis, three pixel classifiers were constructed for image based landmark detection as landmarks on the 2006

RoboCup soccer fields were defined by the three colour coding using cyan, magenta and yellow.

If the location of a landmark in the omnidirectional image defined by an (x_{omn}, y_{omn}) pixel position is known, the relative position of a landmark to the omnidirectional camera can be determined. Given the known height of landmark colourings, the inverse of Equations 3.3 to 3.5 can be used to determine the distance to the landmark defined by $\sqrt{x_w^2 + y_w^2}$ while the relative angle is identical to the angle of the landmark position in the omnidirectional camera image.

3.4.2 The Kalman Filter

The second phase of the realistic application of the omnidirectional camera uses the Kalman Filter for self localisation. In 1960 Rudolf E. Kálmán published a paper describing a recursive solution to the discrete data linear filtering problem in which he presented a recursive filter able to estimate states of a dynamic system based on noisy or incomplete measurements [40]. This solution is frequently used for a fundamental problem in control theory, providing accurate and continuous information about the location and velocity of an object based on scarce observations containing a zero mean error. If the filter is used to estimate a non-linear process, it is referred to as an Extended Kalman Filter (EKF).

For this thesis, the location and orientation of the robot on a Aibo soccer field is estimated using the EKF based on landmark measurements, both in the real and simulated setting. The process to be estimated is formulated by

$$\mathbf{x}_t = g(u_t, x_{t-1}) = \begin{pmatrix} x_{t-1} + v_t \cos(\theta_{t-1} + \omega) \\ y_{t-1} + v_t \sin(\theta_{t-1} + \omega) \\ \theta_{t-1} + \omega \end{pmatrix} \tag{3.11}$$

$$z_t = h(x_t, l) = \begin{pmatrix} \sqrt{(l_x - x_t)^2 + (l_y - y_t)^2} \\ \arctan 2(l_y - y_t, l_x - x_t) - \theta_t \end{pmatrix} \tag{3.12}$$

where $\mathbf{x}_t \in \Re^3 \leftarrow (x_t, y_t, \theta_t)$ defines the robot's two-dimensional location and orientation at time t, $u_t \in \Re^2 \leftarrow (v_t, \omega_t)$ defines user velocity and steering input at time t and $z_t \in \Re^2 \leftarrow (d_t, \alpha_t)$ defines a landmark distance and orientation measurement at time t based on the robot location and the location of landmark l.

The estimated robot position μ_t is recursively estimated by the EKF using two phases. The

first phase, prediction, is defined by the following equations

$$\overline{\mu}_t = g(u_t, \mu_{t-1}) \tag{3.13}$$

$$\overline{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + V_t M V_t^T \tag{3.14}$$

where $M \in \Re^2$ defines expected motion noise and G and V are defined by

$$G_t = \frac{\delta g(u_t, \mu_{t-1})}{\delta x_{t-1}} = \begin{pmatrix} 1 & 0 & -v_t \sin(\theta_t + \omega_t) \\ 0 & 1 & -v_t \cos(\theta_t + \omega_t) \\ 0 & 0 & 1 \end{pmatrix} \tag{3.15}$$

$$V_t = \frac{\delta g(u_t, \mu_{t-1})}{\delta u_{t-1}} = \begin{pmatrix} \cos(\theta_t + \omega_t) & -v_t \sin(\theta_t + \omega_t) \\ \sin(\theta_t + \omega_t) & -v_t \cos(\theta_t + \omega_t) \\ 0 & 1 \end{pmatrix} \tag{3.16}$$

being the Jacobians with respect to location and user input respectively.

The second phase of the EKF algorithm starts with computing the predicted measurement mean

$$\hat{z}_t = h(\overline{\mu}_t, l) = \begin{pmatrix} \sqrt{(l_x - \overline{\mu}_{t,x})^2 + (l_y - \overline{\mu}_{t,y})^2} \\ \arctan 2(l_y - \overline{\mu}_{t,y}, l_x - \overline{\mu}_{t,x}) - \overline{\mu}_{t,\theta} \end{pmatrix} \tag{3.17}$$

and determining the Jacobian of h with respect to robot location

$$\begin{aligned} H_t &= \frac{\delta h(\overline{\mu}_t, l)}{\delta x_t} \\ &= \begin{pmatrix} -\frac{(l_x - \overline{\mu}_{t,x})}{\sqrt{l_x^2 - 2 l_x \overline{\mu}_{t,x} + \overline{\mu}_{t,x}^2 + l_y^2 - 2 l_y \overline{\mu}_{t,y} + \overline{\mu}_{t,y}^2}} & -\frac{(l_y - \overline{\mu}_{t,y})}{\sqrt{l_x^2 - 2 l_x \overline{\mu}_{t,x} + \overline{\mu}_{t,x}^2 + l_y^2 - 2 l_y \overline{\mu}_{t,y} + \overline{\mu}_{t,y}^2}} & 0 \\ \frac{(l_y - \overline{\mu}_{t,y})}{l_x^2 - 2 l_x \overline{\mu}_{t,x} + \overline{\mu}_{t,x}^2 + l_y^2 - 2 l_y \overline{\mu}_{t,y} + \overline{\mu}_{t,y}^2} & -\frac{(l_x - \overline{\mu}_{t,x})}{l_x^2 - 2 l_x \overline{\mu}_{t,x} + \overline{\mu}_{t,x}^2 + l_y^2 - 2 * l_y \overline{\mu}_{t,y} + \overline{\mu}_{t,y}^2} & -1 \end{pmatrix} \end{aligned} \tag{3.18}$$

which gives way to the predicted measurement covariance S_t and the Kalman gain K_t

$$S_t = H_t \overline{\Sigma}_t H_t^T + Q_t \tag{3.19}$$

$$K_t = \overline{\Sigma}_t H_t^T S_t^{-1} \tag{3.20}$$

where Q_t defines the expected measurement noise.

In localisation there is a common problem called the data association problem, which refers to

the problem of measurements not correlating to the appropriate feature, putting the estimation process at risk of braking down. Kristensen and Jensfelt describe a common method to tackle the data association problem, called the *validation gate*, defined by

$$\rho = (z_t - \overline{z}_t)S_t(z_t - \overline{z}_t)^T < \gamma \tag{3.21}$$

where ρ defines the Mahalanobis distance between the measurement and the map feature, and γ gives the size of the gate [41]. If $\rho < \gamma$ the measurement is considered to be valid and the mean and covariance are updated according to

$$\mu_t = \overline{\mu}_t + K_t(z_t - \hat{z}_t) \tag{3.22}$$

$$\Sigma_t = (I - K_t H_t)\overline{\Sigma}_t \tag{3.23}$$

resulting in an accurate estimation of the robot location over time with the following estimates having a zero mean error

$$\mathbf{E}[\mathbf{x}_t - \mu_t] = 0 \tag{3.24}$$

$$\mathbf{E}[z_t - \overline{z}_t] = 0 \tag{3.25}$$

assuming the model definitions are correct.

With the EKF robot location and orientation could be accurately estimated over time based on noisy and occasionally absent landmark measurements, which will be demonstrated in the next chapter.

4 Results

This chapter contains the results of the camera simulation model performance and validation tests. First, performance of the different mirror surfaces is shown as these results decided which mirror surface was used for the final simulation model. Second, the results of performing a two phase robot self localisation method based on omnidirectional visual data in both a real and simulated setting will be compared, indicating to which degree it is possible to extrapolate results obtained in simulation.

4.1 Mirror Polygon Count and System Performance

To determine which of the four mirror surfaces should be used in the final simulation model, the trade-off between sensor data quality and system performance was analysed. The influence of the four mirror surfaces on the amount of frames per second (FPS) rendered by the Unreal Engine™2.0 was recorded and plotted in Figure 4.1. All four mirror surfaces were mounted on a P2AT robot model and spawned in two RoboCup maps, `DM-spqrSoccer2006_250.utx` and

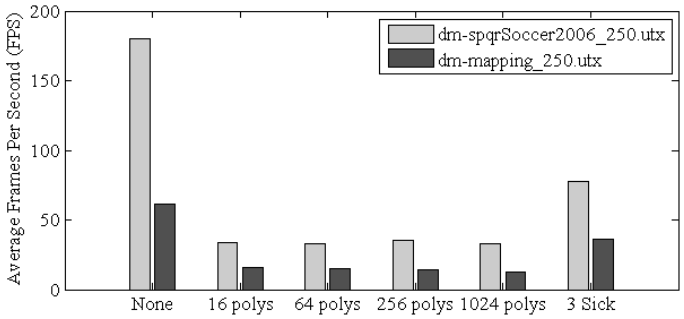


Figure 4.1: Mirror surface polygon count related to USARSim performance.

`DM-Mapping_250.utx`, which vary greatly in size. The exact polygon count of these maps could not be determined, but the first map is roughly one twentieth the size of the second. The test was run on a dual core 1.73 Ghz Pentium processor 1014 MB RAM system with a 224 MB Intel GMA 950 video processor.

It is clear that FPS is influenced by the presence of an omnidirectional camera, though this influence does not depend on the number of mirror polygons. No discernible differences between rendered FPS were observed, which led to the conclusion that the mirror surface with the highest polygon count, providing the highest quality sensor data, is best suited for the final simulation model. All simulated omnidirectional camera sensor data presented in this thesis is provided by the highest polygon count mirror surface, unless stated otherwise.

For reference, the omnidirectional camera system load is roughly twice as demanding as an other commonly used sensor simulation model, that of the LMS200 SICK laser scanner.

4.2 Self Localisation Using Omnidirectional Data

To compare the performance of the simulated omnidirectional camera to the real omnidirectional system on which it was based, a realistic application of the omnidirectional camera was performed in both a simulated and real setting and the results were compared. Colour histogram based landmark detection was performed on 100 omnidirectional camera images data obtained in these two environments:

- on a RoboCup 2006 Four-Legged League soccer field of ISLA, using the omnidirectional camera described in Subsection 3.1.1;
- in the USARSim `DM-spqrSoccer2006_250.utx` map [42], using the developed omnidirectional camera simulation.

Robot self-localization based on obtained landmark localisation measurements has been performed using the Extended Kalman Filter.

4.2.1 Test Setup

In the simulated environment, a P2DX robot model mounted with the 1024 polygon mirror omnidirectional camera was spawned at the lower right corner of the soccer field and it was driven on a clockwise path along the edges of the field until it reached its starting point, where it then crossed the field diagonally heading towards the opposite corner. In the real setting a Nomad Super Scout

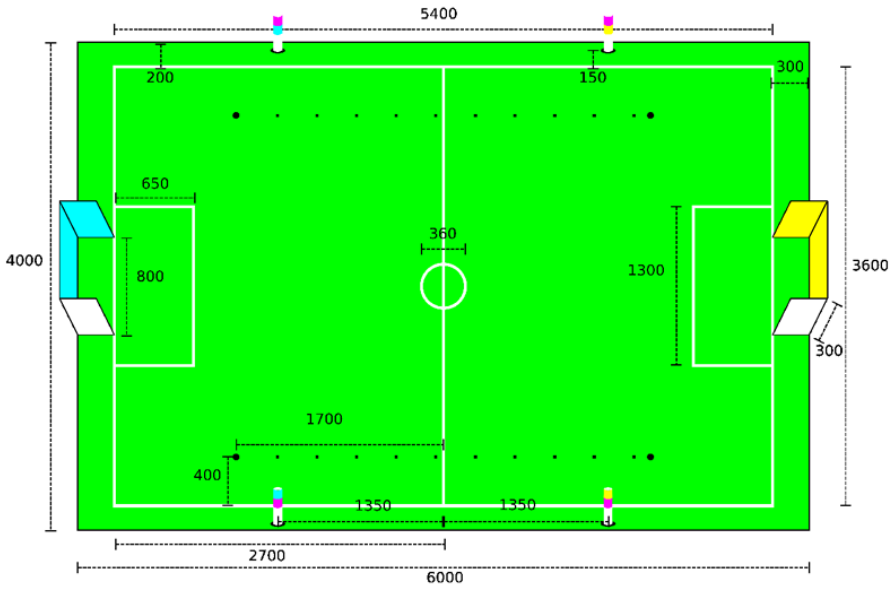


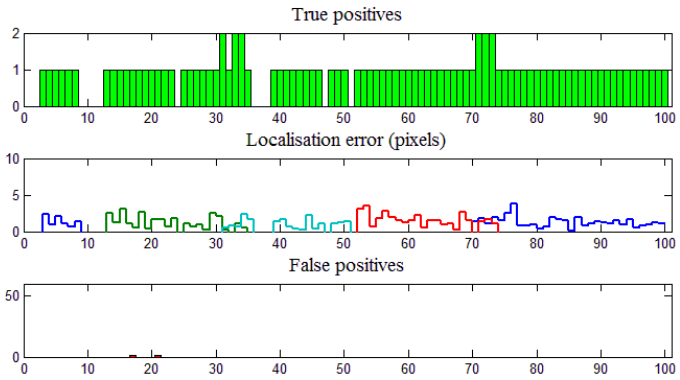
Figure 4.2: RoboCup 2006 Four-Legged League soccer field rules.

II was placed at the same lower right corner where the simulated P2DX model started, mounted with the omnidirectional vision system described in Subsection 3.1.1, where it followed a path identical to the one followed in the simulation run. The soccer field in the real setting was set up according the RoboCup 2006 Four-Legged League rules depicted in Figure 4.2, so that it would correspond to the soccer field represented in the simulation environment, which complies with these rules by design.

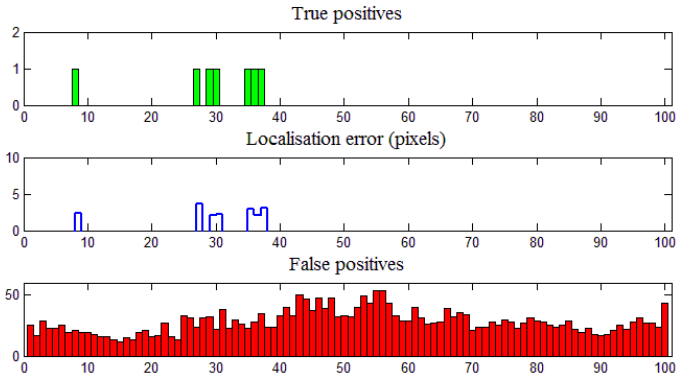
4.2.2 Colour Histogram Based Landmark Detection Results

To compare the accuracy of the colour histogram based landmark detection algorithm in both environments, landmark detection results were compared to landmark localisations obtained by hand. Because camera data is omnidirectional and there are no objects present on the soccer field which could obstruct the view on a landmark, all four landmarks are constantly in view. This makes the representation of missed landmarks (or false negatives) obsolete as this result is defined by four minus the number of true positives. The results are shown in the graphs contained in Figure 4.3.

Test results show the colour based landmark detection algorithm parameters in the simulated environment can hardly be extrapolated to the realistic setting. The statistical colour models



(a) USARSim test results

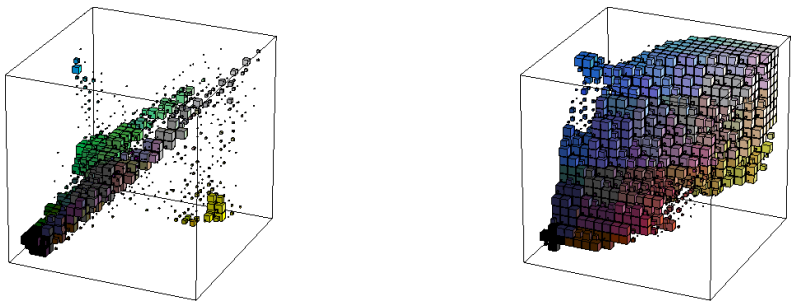


(b) ISLA test results

Figure 4.3: Colour histogram based landmark detection test results.

of the pixel classifier which have been constructed using datasets obtained in the simulated environment are not realistic enough to demonstrate identical performance in the real setting. In the USARSim map, a total of 94 landmarks were detected in 100 omnidirectional images with an average localisation error of 1.3 pixels, and only two false positives were detected in frame 17 and 21, resulting in $\frac{true}{false} = 47.0$ (Figure 4.3(a)). In the real ISLA setting however, these high quality performance results have not been measured. On the ISLA soccer field a total of 7 landmarks were detected in frames 8, 27, 29, 30, 35, 36 and 37 with an average localisation error of 2.6, and a total of 2801 false positives were detected in 100 omnidirectional images (Figure 4.3(b)). This results in a $\frac{true}{false} = 0.0025$ ratio, which obviously is unacceptable performance.

These results are as expected, as the colour histogram based pixel classifier is not expected to perform well in an environment which differs greatly from the environment in which training data is obtained. To depict the difference between the amount and spectrum of colours observed in both



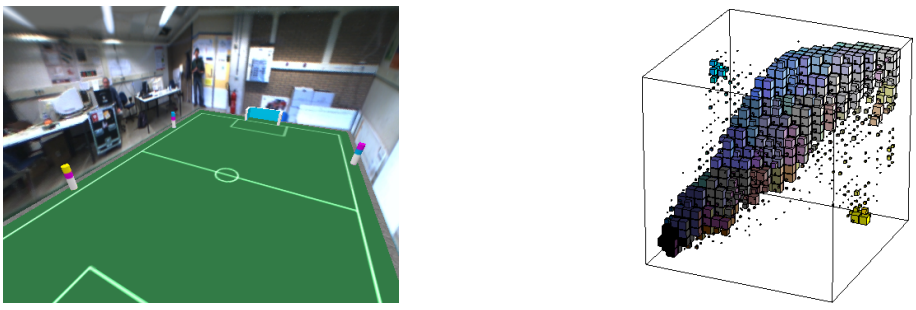
(a) USARSim environment

(b) ISLA environment

Figure 4.4: A rendering of 3D colour histograms created from an omnidirectional image obtained in two environments.

environments, a depiction of 3D colour histograms created from an omnidirectional image obtained in both environments is shown in Figure 4.4, in which the sizes of a coloured cubes within the 3D histogram cube space represents the calculated $P(rgb)$ value using Equation 3.8. As can be seen the $P(rgb)$ values in Figure 4.4(a) are more scarcely distributed, suggesting a smaller set of colours is present in data obtained in USARSim environment. Because of this difference, the statistical models which have been created by training data obtained in the USARSim environment do not represent the statistical probabilities a certain pixel belongs to a class in the ISLA environment, which clearly diminishes the quality of the pixel classifier in the latter environment.

To determine whether it is possible to obtain results in the simulated environment which approximate results obtained in the ISLA environment, a modified `DM-spqrSoccer2006_250.utx` map was created, where the walls were textured with planar projections of omnidirectional sensor data obtained at ISLA, depicted in Figure 4.5(a). Figure 4.5(b) depicts a 3D colour histogram



(a) view of the USARSim map

(b) 3D colour histogram of omnidirectional data obtained in this map

Figure 4.5: A simulation environment textured with planar projections of real omnidirectional camera data.

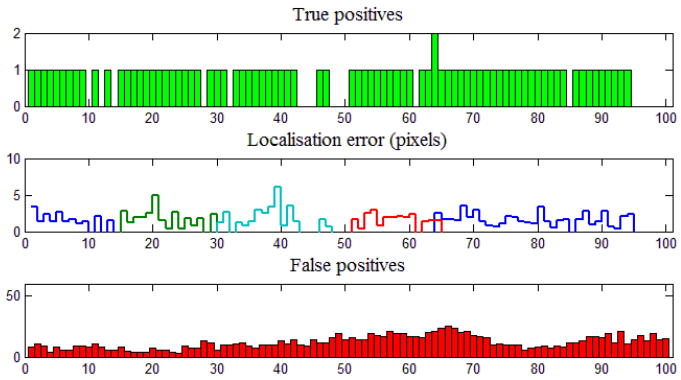


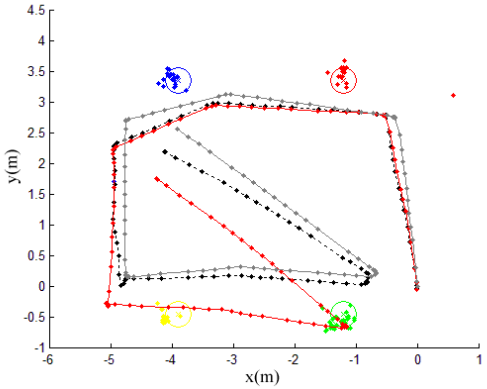
Figure 4.6: Colour histogram based landmark detection test results in the ISLA textured USARSim map.

created from omnidirectional data obtained in this modified map and, as can be observed, the spectrum of colours available in the modified approximates the colours present in omnidirectional data obtained in the ISLA environment.

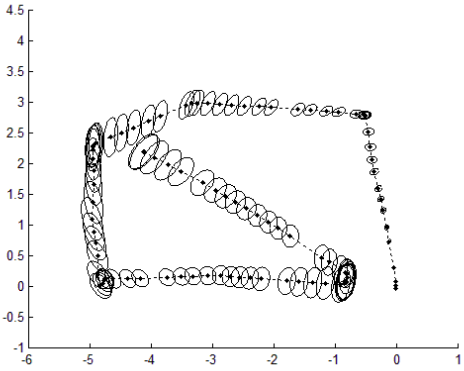
Figure 4.6 shows colour histogram based landmark results obtained in the modified map. In this map a total of 82 landmarks were detected in 100 omnidirectional images with an average localisation error of 1.8354, and 1208 false positives were detected, resulting in $\frac{true}{false} = 0.0679$. The amount of true positives is comparable to the amount of true positives found in the initial USARSim run, as landmarks are not modified in the second USARSim map and are hence detected by the pixel classifiers. However, the $\frac{true}{false}$ ratio resembles the ratio observed in the ISLA environment as the ISLA textures provide an average of 121 false positives per image. Appearantly, adding realistic textures to the simulated environment adds realism to the simulated omnidirectional camera data.

Another factor influencing the colours perceived by an omnidirectional camera in a realistic environment not simulated by the `DM-spqrSoccer2006_250.utx` map is dynamic lighting conditions. In the real setting perceived colours are highly dependable on the relative location of the camera to different light sources. In the simulated environment this is not the case as the Unreal Engine™2.0 uses vertex lighting and precomputed light maps, which are not influenced by camera position during runtime. Though this can be simulated by the Unreal Engine™, current available USARSim maps do not contain this functionality and hence the simulated omnidirectional camera is not subject to this influence.

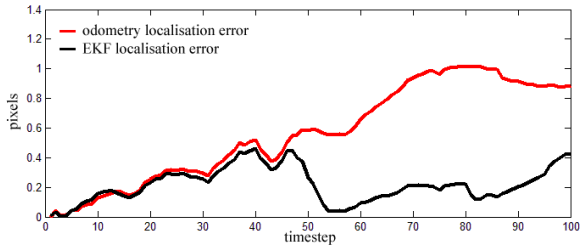
Test results obtained in the modified `DM-spqrSoccer2006_250.utx` map demonstrate that the



(a) Landmark measurements and robot self localisation. Red path: noisy odometry sensor data, black path: EKF estimated path, grey path: actual robot path.



(b) EKF localisation uncertainty covariance plot.



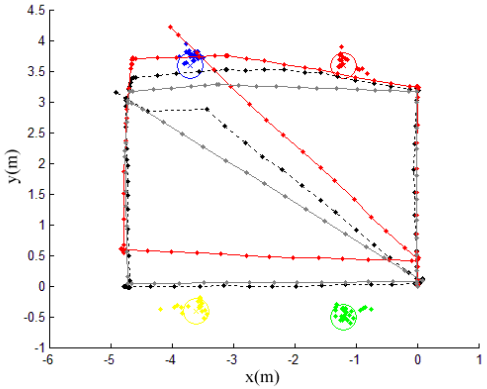
(c) Localisation error in metres.

Figure 4.7: EKF run in the simulated USARSim environment.

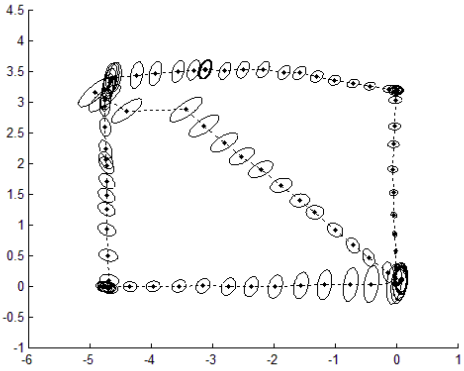
realism of colour information provided by the omnidirectional camera simulation model is heavily dependant on the environment. The virtual IPC cameras responsible for creating the reflection image project the colours they observe on the scripted textures. If the environment does not provide realistic colour information, the omnidirectional camera will not do so as well. However, if realism in colour information is added to the environment either by adding realistic textures or implementing dynamic lighting conditions, the omnidirectional camera simulation model data will become more realistic as well.

4.2.3 Extended Kalman Filter Results

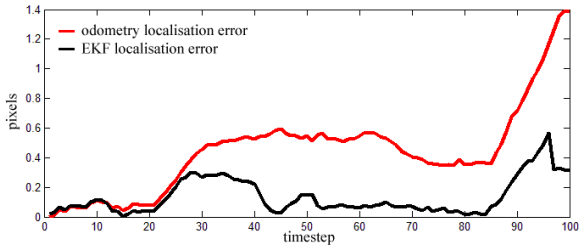
Based on the equations decribed in Subsection 3.4.2, Robot self localisation can be performed. User input u_t is defined by noisy odometry sensor data and the measured distance and relative orientation of landmark measurements is based on results obtained in the previous subsection. As automated landmark detection in the ISLA environment was not efficient enough to produce accurate measurements, landmark detection in the ISLA obtained omnidirectional images was



(a) Landmark measurements and robot self localisation. Red path: noisy odometry sensor data, black path: EKF estimated path, grey path: actual robot path.



(b) EKF localisation uncertainty covariance plot.



(c) Localisation error in metres.

Figure 4.8: EKF run on sensor data obtained in the real ISLA environment.

done by hand, such that the measurements reflect the accuracy depicted in Figure 4.3(a). The results of the test run in the simulated environment are depicted in Figure 4.7.

As can be seen in Figure 4.7(a), landmark measurements remain well within the expected measurement noise parameters in the first 45 timesteps, resulting in little to no Kalman Gain correction on self localisation estimation values. After the 45th timestep, measurements of the lower left landmark result in correction of self localisation estimations, lowering the estimation error to well below 10cm within 8 timesteps. As the robot progresses, the EKF self localisation error does not rise above 20cm until step 87, while the noisy input based self localisation error reaches roughly three times as much, as can be seen in Figure 4.7(c). Figure 4.7(b) shows the localisation uncertainty covariance, depicting a lower uncertainty in the corners. This results from the robot being stationary in the lateral sense while rotating around its vertical axis. The multiple measurements of landmarks combined with the robot remaining at the same location results in less localisation uncertainty.

Figure 4.8 shows the results obtained by performing EKF self localisation on sensor data

obtained in the real ISLA environment. Quite noticeable in this test run is the improvement of self localisation by the EKF algorithm from the 27th step on. The improvement can be observed in Figure 4.8(a) in between the top two landmarks. This improvement results from the occurrence of two simultaneous measurements of a landmark, increasing the Kalman gain. After the robot has left the top left corner to head downwards to the lower left corner, EKF localisation error drops well below 20cm. The estimated robot path along the bottom from the lower left corner to the lower right defined by timesteps 52 to 86 is very accurate as it remains within a 10cm estimation error. However, when the robot leaves the lower right corner it turns out the estimated robot orientation is off by 10 degrees. Measurements continue to remain within expected noise parameters until timestep 97, where the Kalman gain based on measurements of the top left landmark improves self localisation estimation by about 20cm. As with the run in the simulation environment, the self localisation uncertainty lowers in the corners due to identical causes.

Both runs display comparable improvement in self localisation by the EKF algorithm using completely identical parameters. The average self localisation improvements over 100 timesteps are 30.35cm and 35.11cm in the simulated and real environment respectively. Also, both runs also demonstrate a lowering of self localisation uncertainty in the corners and, in general, show comparable behaviour. These two runs demonstrate that is possible to extrapolate EKF self localisation results obtained in the simulation environment using the simulated omnidirectional camera model. As the EKF algorithm uses sensor measurements based on omnidirectional image composition, it is assumed that it is possible to extrapolate results from other methods based on omnidirectional image composition.

5 Discussion and Further Work

Now that the simulation model has been developed and tested, it is time to take a critical perspective on the omnidirectional camera simulation model and asses the realism of the simulation model. Limitations shall be pointed out and possible further developments will be discussed.

5.1 Simulation Requirements

A crude way to describe one of the foremost simulation requirements is stating that the simulation should be as realistic as possible. However, a simulation of a system is defined by the operation of a model of that system, which is by definition a simplified representation. The realistic quality of this simplification heavily depends on the purpose of the intended usage, and to properly analyse this simulation requirement one cannot regard it without an intended use. It is also important to analyse intended use as it defines other simulation requirements such as reproducibility. It must be said that general simulation requirements are found to vary greatly in literature, hence an effort has been made to address simulation requirements in this section which are specifically related to USAR simulations.

USARSim was developed to be a tool for some aspects of Human-Robot Interaction (HRI) research. Physical robotics is somewhat inappropriate for this type of research as those involved in HRI research generally operate outside computer science or mechanical engineering departments, making physical robots quite expensive and unreliable. A more appropriate solution is the use of a simulator, which should the costs and should offer a more secure research environment. Regarding the fact that the general object of study in HRI research is human behaviour, a simulation should offer full control of the robot and the environment in which the robot operates to make large

samples, repeated trials and controlled varied conditions a possible. Also, both high quality graphics and accurate physics are needed for HRI research because the operators tasks depend strongly on remote perception [43].

Modern behaviour-based robotics research currently investigates a wide spectrum of design philosophies. To make a classical disjunction, on one side of the spectrum it is regarded that effective systems can best be implemented as stimulus-response systems that are void of any internal representations of the environment in which the agent operates, while on the other side of the spectrum high level planning algorithms are employed, enabling agents to plot goal-directed paths in a complex environment. Though these general tenets differ in essence, for research in both directions it is required that a good simulation must simultaneously supply an accurate model of the robots geometry and kinematics, accurate models of sensors, an accurate model of the environment and an accurate model of the robots interaction with that environment so that simulation results may be generalizable to real robots [13].

Regarding educational purposes, simulation is used to provide a low cost platform for instructors to teach robotics. A simulation should provide the means for students and instructors to explore concepts employed in physical robotics as well devise new concepts. Hence, simulation should impose emphasis on realism of its models so that users are assured that the encountered problems and the devised solutions relate to physical robotics and are not artefacts of the simulation platform.

A simulation platform for USAR responses, providing a means to test the usefulness or efficiency of using a robot in a disaster situation, the requirements are more complex. For example, it is heavily argued by those having experience with real USAR responses that a simulated response should be able to simulate all aspects of an unpredicted response (i.e. unforeseen weather and environmental conditions, emotions, material transport and time pressured situations) [3]. Proper analysis of using robots in a USAR response depends not only on the extent to which a robot is able to perform a certain task, but also the diversity of situations in which it is able to perform its task and the amount in which robotic systems are able to adapt to unforeseen circumstances. A proper simulation environment for a general assessment of using robots in a USAR response should provide such a means through Unforeseen Event Modelling.

5.1.1 Simulation Requirements and USARSim

USARSim has proven itself to be a valuable research tool and its validation has been extensively been addressed in [13, 38, 44]. In relation to HRI research, Wang et al. have shown that operator control behaviour in a USARSim simulated setting shows great similarities to behaviour when dealing with a real robot. Similarities between learning rates, terrain effects and task completion times have been observed. They also show that by comparing robot operation in both a real NIST yellow arena and a USARSim map based on that arena, that there is close agreement between data obtained from real and simulated sensors. They positioned robots in roughly identical locations so that range sensor readings overlapped. Then they compared Hough transforms of retrieved sensor data, which correlated to a high extent [44].

It has also been shown that in behaviour-based robotics certain test results obtained in USARSim are generalizable to a real setting without modifying the algorithms employed. The University of Amsterdam has shown that a SLAM algorithm tested and developed in the USARSim environment could be integrated into a real robot without modification and produce results with identical accuracy [45]. This shows that USARSim robot geometry, kinematics and the models of the range finding sensors are accurate.

The USARSim environment currently does not simulate aspects required for a proper USAR response assessment testing environment. Existing USARSim environments representing disaster areas are currently extremely static when compared to real disaster situations. For example, USARSim environments do not represent the presence of dynamic intelligent rescue workers operating at the scene. Even though the design of USARSim does provide a basis for these elements to be developed, a lot of work needs to be done before USARSim will meet the requirements set by those having experience with real USAR responses.

5.1.2 Requirements and the Simulated Omnidirectional Camera

Now that USARSim and specific simulation requirements have been discussed, the two questions which need to be asked here are:

- Does the catadioptric omnidirectional camera simulation model meet the previously stated simulation requirements? and
- how to what extent does the simulated sensor data present realism considering the camera's intended applications?

Meeting the Requirements

In this section several simulation requirements have been described from the perspective of simulation usage, but given the fact that mentioned requirements overlap each other the question whether or not the camera model meets a requirement is addressed per requirement.

Sensor Data Realism - One of the most important theoretical aspects underlying the omnidirectional view of the catadioptric camera is the Single Viewpoint Constraint, and, as has been discussed in Chapters 3, the sensor meets this Single Viewpoint Constraint by design. All perceived rays of light pass through a single point in space and UV texture mapping has been verified to correlate to real mirror reflection theory. In Chapter 3 it is verified that image transformation techniques produce accurate results.

When comparing simulated sensor data to real sensor data, great colour discrepancies are apparent. If the USARSim environment does not contain textures that show the same spectrum of colours perceivable in real environments it means the omnidirectional camera is not capable of producing realistic colour depth. In Chapter 4 it is shown that the introduction of realistic camera textures improves the realism of the sensor data. However, the behaviour of light in realistic settings is far more complex than in the Unreal environment as the Unreal Engine™2.0 uses vertex lighting and precomputed light maps, lowering the realism of simulated lighting conditions and sensor data. Because of this, colour detection algorithms developed in the simulation environment are not generalisable to a real application, as could be seen in Chapter 4. It must be pointed out that the regular camera simulation model already available in the USARSim environment also does not simulate realistic colours and thus the omnidirectional camera data is just as realistic as data provided by the standard available camera in USARSim.

The standard USARSim camera nor the omnidirectional camera simulate realistic image deformations such as any of the four sources responsible for collinearity departure (Subsection 2.1.4), because sensor data is provided by an Ideal Perspective Camera. Another element not simulated by these cameras is motion blur. As has been described in Subsection 2.1.4, within a real camera the time taken for the composition of an image based on light registered by the charge-coupled device does not always coincide with the time a shutter opens up for the light to pass and if a shutter opens up twice or too long during a single image composition, possibly non-identical views are merged into a single image. When these views are different due to relative camera movement, the occurrence of this event causes movement blur. As the simulation models do not include simulation implementations of either shutters or lenses they currently do not simulate motion

blur.

What the omnidirectional camera model does simulate, however, is that in terms of image composition the simulated omnidirectional camera model does provide an omnidirectional view which is correct when compared to a real omnidirectional camera. In Chapter 4 it can be seen that the omnidirectional camera creates image data with a perspective transformation identical to that of the real catadioptric omnidirectional camera. The algorithms used for determining relative locations of identified Landmarks are generalisable to a real application and the KF localisation algorithm produces results with comparable accuracy. This makes the camera model accurate enough for use in HRI research, robotics research and educational activities.

Geometry and Physics Accuracy - The geometry of the real omnidirectional catadioptric camera was measured and the simulated camera was then modelled to those measurements, in a proper scale related to all other objects in USARSim. The mirror surface is modelled to the surface equation defined by Accowle Company, Ltd. making the complete model geometrically correct. Physically, some major properties of the real catadioptric system have not been simulated. Though the Static Mesh is a single entity, the real catadioptric camera consists of many parts that can move in relation to each other. Prior to using a real omnidirectional cameras, calibration must be performed because even minor misalignments within the catadioptric system require major modifications to perspectival transformation algorithm settings. Excessive bumping or shaking during runtime could also influence relative positions in a realistic setting, which could cause a major shift in sensor data content. These effects are not yet simulated by the omnidirectional camera simulation model.

Unforeseen Event Modelling - Real omnidirectional cameras suffer from unforeseen events. Though analysing the unforeseen might seem like a paradox, note that a developer could include behaviour which users do not foresee, like camera malfunctioning due to physical damage or randomised minor data anomalies. These aspects have not been included into the simulation model as the simulation environment as a whole does marginally provides accurate implementation of the unforeseen. Modelling unforeseen events for the omnidirectional camera while the environment as a whole does not yet sufficiently support this seems superfluous.

Parameter Control - Though not directly related to simulation data realism, simulation reproducibility and the possibility for repeated trials are important aspects in many fields of research, hence control over parameters is a simulation requirement as well. The omnidirectional camera is a static object and the properties of the simulation model are mostly defined. There are

currently no customisable properties to the Static Mesh besides a location where it is mounted on the robot. This location of the camera is pivoted at its effective viewpoint (Subsection 3.1.2) and can be accurately instantiated in the .ini-files. As the omnidirectional image data is captured by a regular USARSim camera, settings such as image size, image quality and frame rates can be fully controlled in a manner previously possible and described in the USARSim manual [46].

USARSim Contribution

Vision is one of the richest perceptual sources for both autonomous and remotely operated robots. A realistic simulator cannot therefore omit a realistic and quantitatively precise video simulation component [38]. As the catadioptric omnidirectional camera is a very common visual sensor in the current robotics field, the addition of this sensor to the USARSim environment should be a very valuable addition.

In HRI research different types of use of the omnidirectional camera can be explored and compared. As the omnidirectional camera model is capable of producing accurate perspectival transformed images of its complete surroundings, optimal strategies for applying visualisation techniques can be researched. Full and easily accessible control of camera placement and image quality settings allows even researchers operating outside computer science or mechanical engineering departments to work with the system. In behaviour-based robotics the omnidirectional camera model can be used for research in methods to expand the autonomy of robots as the validation techniques described in this thesis demonstrate this possibility. Research described in Section 2.1 can now benefit from this simulation environment and improve their existing techniques without the use of real and costly testing setups. The quality of the omnidirectional camera model is also sufficient for educational purposes, allowing students and teachers to explore its use and to devise new applications.

The technical committee of the RoboCup Rescue Simulation League has allowed the use of the omnidirectional camera simulation model to be used in the upcoming RoboCup 2008 held in China. This will hopefully improve the scores of participating teams, demonstrating the usefulness of the omnidirectional camera in USAR operations.

5.2 Further Work

The simulation model is complete and ready for use, though there remain elements which could be expanded. There are technical limitations that could be resolved and improvement is always a

possibility. Some suggestions are given next.

5.2.1 Multiple Cameras

The current implementation method imposes an important limitation which should be resolved, as currently only one instance of the omnidirectional camera simulation is able to operate in a single environment. As explained in Subsection 3.1.3, the mirror is simulated using five CameraTextureClients which send Virtual Camera views to the five appropriate Scripted Textures. Spawning a second omnidirectional camera in the same environment creates five new CameraTextureClients and a second Static Mesh textured with the same five Scripted Textures. As CameraTextureClients draw their Virtual Camera Views on all Scripted Textures with the appropriate label, every CameraTextureClients draws their view on two instances of the Scripted Texture, one per camera. As, by implementation, the Unreal Engine™2.0 deals with CameraTextureClients in spawning order, this results in the new CameraTextureClients drawing their Virtual Camera Views over the old ones and both cameras will show the identical omnidirectional image data: that of the second camera.

To solve this, an extended and more complex USARCamerTextureClient needs to be developed. This USARCameraTextureClients should be able to do two things:

- allow assignment of a Scripted Texture during runtime, and
- discern between multiple occurrences of a single Scripted Texture present in the environment.

If this is implemented, different mirror Scripted Texture instances can dynamically be assigned to appropriate USARCameraTextureClients after a new robot is spawned, allowing multiple cameras to be simultaneously active in the same environment. This USARCameraTextureClient also improves the quality of the complete model implementation as a single USARCameraTextureClient class could operate where five are required in the current implementation (see Subsection 3.3.2).

5.2.2 Simulation Improvement

In the previous section some possibilities for improving the realism of the simulation model have been addressed. Modifying the physical properties of the omnidirectional camera Static Mesh could improve the correctness of simulation data with respect to calibration necessity, though this requires some complex adjustments to the complete simulation model. First, the Static Mesh needs to be subdivided in multiple meshes, so that minor variations in system composition can

be realised. The composition could then variate whenever the system is spawned to simulate the misalignments occurring in real catadioptric omnidirectional systems. However, simulating a real-time correct mirror reflection is extremely complicated and requires heavy modification and thorough research. The current simulation method is based on a static position of the recording camera relative to the mirror surface, on which the position of the Virtual Cameras and the placement of the Scripted Textures is based. In a real setting, a modified position of the recording camera relative to the mirror changes the position of effective viewpoint and the simulation should show identical behaviour. The solution to the complex equation defining the effective viewpoint can be pre-computed, lowering computational costs, though the main problem does not lie with the placement of the Virtual cameras, but with texture mapping the mirror surface. The UV texture mapping of the five Scripted Textures is defined by the position of the effective viewpoint, and UV texture mapping cannot be modified real-time. As simulation of real-time configuration changes is desired and as creating different Static Meshes for each possible effective viewpoint position is extremely inefficient, another solution needs to be found. A possibility might be to develop an extended CameraTextureClient class especially designed to create non planar projections on a Scripted Surface, though this requires solid Unreal Engine™2.0 expertise.

Simulating motion blur can easily be realised in USARSim, as long the framerate of an Unreal Client is higher than the recording frequency of the simulated recording camera. As mentioned in Subsection 2.1.4, motion blur occurs when non-identical views compose a single camera image, so if the framerate of an Unreal Client is higher than the framerate of the recording camera, multiple views in the Unreal Client can be used to compose a single camera image. The quality of the camera can be defined by defining how often motion blur occurs and the amount of blur can be defined by the amount of Unreal Client views used to construct a single camera image. As image data obtained from recording cameras in the USARSim environment are obtained from the Image Server through the network, a possible location where motion blurring could be implemented is within the image server itself. Messages can be used to communicate unique behaviour per client, so that the image server is capable of delivering camera data of non-identical quality.

6 Conclusions

Urban Search And Rescue is a term describing an operational framework of national rescue facilities which respond to (inter)national disaster situations in urban environments to support local emergency responders to locate victims and manage recovery operations. Since the September 2001 collapse of three towers at the World Trade Center robots have been used to aid USAR task forces in locating victims in hazardous environments. As scientists rarely have the opportunity to experiment in real disaster situations, other means have been developed to perform research in USAR robotics.

USARSim is a high fidelity 3D simulation environment of mobile USAR robots and environments, used for research in the fields of HRI, Robotics and Artificial Intelligence. The simulator has a devoted community which adds simulation models and other content every year and the subject of this thesis is the development of a simulation model of a parabolic catadioptric omnidirectional camera.

The questions which were asked at the start of this thesis and were:

- Is it possible to create a simulation model of a catadioptric omnidirectional camera in the USARSim environment which can be used as a tool for research in methods using this type of vision?
- If a simulation model can be created, does the simulation model meet simulation validation requirements which are based on the camera's intended use?

In the course of this thesis, these two questions have been answered.

Exploring different methods for obtaining omnidirectional vision data, the choice has been made to simulate the parabolic omnidirectional camera as it satisfies the Single Viewpoint Constraint (SVC) and is a common sensor in robotics and AI research. The SVC is an important

requirement for omnidirectional vision systems as meeting this requirement allows geometrically correct perspective images to be constructed.

The simulation is modelled after the vision system employed by the Intelligent Systems Lab Amsterdam (ISLA) group, University of Amsterdam, consisting of a Dragonfly®2 camera made by Point Grey Research Inc. and a Large Type Panorama Eye®made by Accowle Company, Ltd. The 3D camera simulation was modelled to scale in the Lightwave graphics content creation tool, the mirror surface defined by the general solution of mirror surfaces meeting the SVC. The most important elements contributing to the proper simulation model are

- CameraTextureClients — actors in the Unreal environment which allow camera views to be projected on dynamic textures. These actors are used to simulate reflection on the mirror surface. Virtual cameras were placed in the single effective viewpoint based on the SVC.
- verified proper UV Mapping — the method to precisely define how a 2D image should be applied to a 3D surface, properly applied on the mirror surface
- Karma definitions — simulated camera rigid body physics
- UnrealScript programming — integration of the simulation model in the Unreal environment

Research was done to obtain the mirror surface best suited for the simulation model and model validation tests have been performed. Comparisons between image transformations of real and simulated data show that there is close agreement between the real and simulated sensor images. Performance comparison in a real life application setting produced results which are not completely identical, though highly comparable. Again, there is close agreement between real and simulated variations. This answers the first question as it shows that a simulation model of a catadioptric omnidirectional camera in the USARSim environment can be used as a tool for research in methods using this type of vision.

In answer to the second question, it must first pointed out that realism of simulated sensor data is highly dependable on the environment in which the camera operates. In Subsection 4.2.2 it is shown that the degree to which it is possible to extrapolate results obtained in simulation is highly dependable on realism contained in the colours provided by the environment. If realism in colour information is added to the environment by adding realistic textures, it is shown that the omnidirectional camera simulation model data will demonstrate more realism than was initially perceived in the map provided by USARSim community. However, in terms of image composition the simulated data presents a high degree of realism independent of the environment. In Subsection 4.2.3 it is shown that it is possible to extrapolate EKF self localisation results obtained in the

simulation environment using the simulated omnidirectional camera model. As the EKF algorithm uses sensor measurements based on omnidirectional image composition, it is assumed that it is possible to extrapolate results from other methods based on omnidirectional image composition.

The model also meets simulation requirements dictated by multiple uses as much as the standard available camera in USARSim. Besides demonstrating a high degree of realism in terms of image composition, the model also allows complete parameter control for reproducibility and repeated trials. The model is also geometrically correct, though physically some major properties of the real catadioptric system have not been simulated. Neither does the model simulate unforeseen events, though it must be said that the environment as a whole marginally supports implementation of unforeseen events. Further work can be done to improve the realism of the camera and as USARSim is constantly in development I believe the omnidirectional camera realism will improve over time.

A lot of time and effort went into creating a proper simulation model of an catadioptric omnidirectional camera for the USARSim environment. I believe I have achieved to do so, making it a very valuable addition to the USARSim environment. I hope a lot of interesting and valuable research will be done with the use of this simulation model, so I will be keeping an eye on coming RoboCup Rescue competitions!

Appendices

A

USARSim Design and Usage

USARSim in itself is a set of Unreal models and classes defining the simulation of robots, sensors and actuators. Note that though robots seem dynamic, within USARSim all robots, sensors and actuators are (collections of) Static Meshes (subsection 2.2.1). As the internal structure of the Unreal Engine™2.0 is proprietary, USARSim is built on top of the Gamebots project, which is started at the University of Southern California's Information Sciences Institute to turn the Unreal Tournament game into a domain for research in artificial intelligence [47].

A.1 USARSim Design

The USARSim system uses a client/server architecture with three main components that construct the system:

1. the Unreal engine that makes the role of server
2. the Gamebots that communicates between the server and the client
3. the Control client that controls the robots on the simulator

The client side includes the Unreal client and the controller or user applications. The Unreal client renders the simulated environment, so a view of the robot in the environment is obtained here. All the clients exchange data with the server through the network. The Unreal server encompasses the Unreal Engine™, the Gamebots, the map and the models. It maintains the states of all the objects in the simulator and it responds to the data from the clients by changing the objects states and sends back data to both Unreal clients and the user side controllers. A schematic of the system architecture can be seen in figure A.1.

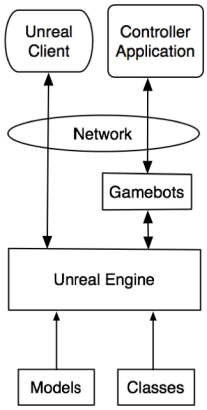


Figure A.1: USARSim architecture.

A.2 USARSim Usage

Initially, the Unreal Server must be started up. It loads a set of geometrical models describing all the objects in the environment and a set of classes of compiled scripts that govern the behaviour of loaded models. After this start-up phase is completed, the system is ready to accept connections from client applications. Users can start controller applications to control robots inside the simulated environment. Any language capable of reading and writing a TCP socket can be used. The format used to exchange information is a simple text based protocol documented in [46] or use available Player, Pyro or MOAST compliant wrappers.

Users can also connect using a standard Unreal Tournament Client to render the simulated environment. The functionality of this client is twofold:

- If a camera is mounted on a robot, the client can be set to visualize the camera view. The Unreal Client can then be attached to a controller application, so the application can work with camera image data. The application needs to capture the back buffer of DirectX 8.x and store it as a raw picture in a block of shared memory.

  The USARSim package also provides an Image Server, which is an alternative way of obtaining camera images. The Image Server starts its own Unreal Client and starts capturing images. These images are sent through the network to any client requesting these images.
- The client can also be set in Spectator Mode, in which a disembodied observer can move within the simulator without interacting with the robots, thus allowing a close observation of the performance from different points of view [13].

B

Unrealscript Classes

To integrate the omnidirectional camera simulation model into the Unreal environment, several classes had to be created in UnrealScript.

B.1 The OmniCamBase class

As explained in Subsection 3.1.2, five Static Meshes were created for the omnidirectional camera, four mirrors and one base housing mesh. For each Static Mesh an UnrealScript class had to be created in which object behaviour properties can be defined.

As is shown in UnrealScriptCode B.1, physical properties are stored in the `KarmaParamsRBFull` object. `KMass` defines static mesh mass, line 17-22 define standard rotational inertia. All other variable declarations are also Karma default values.

B.2 The OmniCam1024Mirror class

UnrealScriptCode B.2, shows the OmniCam1024Mirror class which is defined to have one third the KMass of the OmniCamBase class. `KFriction` is lowered as a mirror provides little friction resistance to other onbjects.

B.3 The CameraTextureClient Classes

As can be seen in in UnrealScript B.3, the USARCameraTextureClientN class extends the CameraTextureClient class. In the third line, '`config (USARBot)`' dictates that default values of variables can also be defined or modified in the USARBot.ini file (see section 2.2.1).

UnrealScript B.1 The OmniCamBase Class

```
 1 class OmniCamBase extends KDPart;
 2
 3 defaultproperties
 4 {
 5   DrawScale=4.762
 6   StaticMesh=StaticMesh'USARSim_VehicleParts_Meshes.OmniCam.CatOmniCam_base'
 7
 8   Begin Object Class=KarmaParamsRBFull Name=KParams0
 9     KMass=0.015
10     KActorGravScale=0.00
11     KStartEnabled=True
12     bKDoubleTickRate=True
13     bHighDetailOnly=False
14     bClientOnly=False
15     KMaxAngularSpeed=100
16     KMaxSpeed=25000
17     KInertiaTensor(0)=0.4
18     KInertiaTensor(1)=0
19     KInertiaTensor(2)=0
20     KInertiaTensor(3)=0.4
21     KInertiaTensor(4)=0
22     KInertiaTensor(5)=0.4
23     KLinearDamping=0.0
24     KAngularDamping=0.0
25     KFriction=0.5
26     KRestitution=0.0
27   End Object
28   KParams=KarmaParamsRBFull'USARModels.OmniCamBase.KParams0'
29 }
```

The first two default variables, `bNoDelete` and `bStatic`, need to be set to `false`, otherwise the class cannot be dynamically spawned by Unreal's `Spawn()` function. When `bNoDelete` is set to `false`, the class can dynamically be deleted. When `bStatic` is set to `false`, the class will dynamically be updated during runtime.

The third variable, `DestTexture`, is the main reason why I had to create five different classes for the USARCameraTextureClients. I was able to assign the appropriate Scripted Textures to this variable dynamically, but for unknown reasons the Scripted Texture then did not display the virtual camera view. If I declared the `DestTexture` variable as a default property in its class, it show the virtual camera view. Unable to solve this any other way, I created 5 different classes with their appropriate `DestTexture` declarations.

The `RemoteRole` variable declaration '`ROLE_SimulatedProxy`' defines that the actor is replicated and may execute simulated functions and simulated state code on the remote side in a client-server setup, which is required for the USARSim architecture. The `CameraTag` variable

UnrealScript B.2 The OmniCam1024Mirror Class

```
1 class OmniCam1024Mirror extends KDPart;
2
3 defaultproperties
4 {
5   DrawScale=4.762
6   StaticMesh=StaticMesh'USARSim_VehicleParts_Meshes.OmniCam.CatOmniCam_Mirror1024'
7   bUnlit=true
8
9     Begin Object Class=KarmaParamsRBFull Name=KParams0
10    KMass=0.005
11    KActorGravScale=0.00
12    KStartEnabled=True
13    bKDoubleTickRate=True
14    bHighDetailOnly=False
15    bClientOnly=False
16    KMaxAngularSpeed=100
17    KMaxSpeed=25000
18    KInertiaTensor(0)=0.4
19    KInertiaTensor(1)=0
20    KInertiaTensor(2)=0
21    KInertiaTensor(3)=0.4
22    KInertiaTensor(4)=0
23    KInertiaTensor(5)=0.4
24    KLinearDamping=0.0
25    KAngularDamping=0.0
26    KFriction=0.1
27    KRestitution=0.0
28     End Object
29     KParams=KarmaParamsRBFull'USARModels.OmniCam1024Mirror.KParams0'
30 }
```

UnrealScript B.3 The USARCameraTextureClientN Class

```
1 #exec OBJ LOAD FILE=..\Textures\USARSim_VehicleParts_Textures.utx
2
3 class USARCameraTextureClientC extends CameraTextureClient config(USARBot)
4  placeable;
5
6 defaultproperties
7 {
8  bNoDelete=false
9  bStatic=false
10   DestTexture=ScriptedTexture'USARSim_VehicleParts_Textures.OmniCam.North'
11   RemoteRole=ROLE_SimulatedProxy
12   CameraTag=USAREmitter
13   FOV = 90.000000
14 }
```

defines the name, or tag, of the Actor which provides this CameraTextureClient with a location and direction on which it can base its virtual camera view. The final variable, `FOV`, defines the

UnrealScript B.4 The USAREmitter Class

```
1  class USAREmitter extends Emitter
2      placeable;
3
4  defaultproperties
5  {
6      bStatic = false
7      bNoDelete = false
8      bDirectional=true
9  }
```

Field Of View of the virtual camera.

B.4 The Emitter Class

As can be seen in in UnrealScript B.4, the USAREmitter class extends the Emitter class.

The Emitter class is part of Epic's particle emitter system, used for creating visual effects such as fire, water fountains or smoke. An Emitter itself is merely a container for one or more instances of the ParticleEmitter class, which create the effects. The USAREmitter class has `bNoDelete` and `bStatic` set to `false` to be able to spawn this class dynamically (see section B.3). The variable `bDirectional` is set to `true` to give the USAREmitter an orientation. This is necessary as it will act as Camera Actor for USARCameraTextureClients.

B.5 The Krobot Class

UnrealScript B.5 displays an excerpt of the `PostNetBeginPlay()` function in the KRobot class, in which the spawning of the USAREmitter and USARCameraTextureClientN classes is displayed.

The struct `CamTexActors` and its members can be declared in the USARBot.ini file. This declaration is explained in the next section. If any Camera Texture Actors are declared in the ini-file, the program will eventually `Spawn` an USAREmitter, of which the reference is stored in the `myEmitterNorth` variable. The `Spawn()` function takes the following five arguments: the class to instantiate, the 'owner' of the object, a 'spawn tag', a location and a direction vector. As can be seen on line 1004 and 1006, the third and fifth arguments are not defined when the `Spawn()` method is called. The USAREmitter does not need a spawn tag and its direction is defined after its base is set.

At line 1007 the USAREmitter's `bHardAttach` property is set to `true`, so that the emitter

UnrealScript B.5 Excerpt of the KRobot Class

```
      ...

 755  simulated function PostNetBeginPlay()
 756  {

        ...

 996    // Mount OmniCam elements
 997    if (CamTexActors.ItemName!="") {
 998        Parent = FindPart(CamTexActors.Parent);
 999        if (Parent!=None) {
1000            GetAxes(Parent.Rotation,RotX,RotY,RotZ);
1001
1002            // ***** spawn Camera Actor Emitter North
1003            myEmitterNorth = USAREmitter(spawn( ...
                            ... CamTexActors.ItemClass,
1004                    Parent,,
1005                    Parent.Location + ...
                        ... CamTexActors.Position.X * RotX + ...
                        ... CamTexActors.Position.Y * RotY + ...
                        ... CamTexActors.Position.Z * RotZ,
1006                ));
1007            myEmitterNorth.bHardAttach=true;
1008            myEmitterNorth.SetBase(Parent);
1019            myEmitterNorth.SetRelativeRotation( ...
                    ... CamTexActors.uuDirection);
1010
1011            // Spawn CameraTextureClient North
1012            myCamTexClientNorth = spawn( ...
                  ... class'USARBot.USARCameraTextureClientN',
1013                    Parent,,
1014                    Parent.Location + ...
                        ... CamTexActors.Position.X * RotX + ...
                        ... CamTexActors.Position.Y * RotY + ...
                        ... CamTexActors.Position.Z * RotZ,
1015                );
1016            myCamTexClientNorth.CameraActor = myEmitterNorth;
1017            myCamTexClientNorth.SetBase(Parent);

            ...

1107    }

      ...
```

cannot move or rotate relative to its base due to Unreal events. If this Actor variable is not set to true, these Emitters will not mimic the x or y orientation of its base, resulting in unwanted effects if the camera pitches or rolls. Setting this variable as a default property in the USAREmitter class

does not result in hard attachment, therefore this must be done during runtime in the KRobot class.

After the orientation of the Emitter is set at line 1019, an instance of the USARCameraTextureClientN class is spawned at line 1012 in the same manner as the Emitter. Its orientation has no effect on its purpose, therefore it does not need to be hard attached or rotated. The only property which needs to be defined is the `CameraActor` variable, to which the reference of the USAREmitter is assigned.

As five USAREmitters and USARCameraTextureClients are needed the section from line 1002 to 1017 is repeated 4 times in the KRobot lines 1019 to 1093. The only difference with this section, besides classes used, is the `SetRelativeRotation()` function for the USAREmitters. East, south, west and center have modified rotations, according to the description in section 3.1.3.

B.6 The USARBot.ini File

UnrealScript B.6 displays an excerpt of the USARBot.ini file, in which config variables for UnrealScript classes can defined (how this works is mentioned in section 2.2.1). This section defines three configuration variables: `MisPkg`, `Cameras` and `CamTexActors` for any descendant of the KRobot class.

UnrealScript B.6 Excerpt of USARBot.ini defining use of the omnidirectional camera

```
p   MisPkgs=(PkgName="OmniCamera",Location=(Y=0.0,X=0.0,Z=0.0), ...
          ... PkgClass=Class'USARMisPkg.OmniCamera')

q   Cameras=(ItemClass=class'USARBot.RobotCamera', ...
          ... ItemName="Camera",Parent="OmniCamera_Link0", ...
          ... Position=(Y=0.0,X=0.0,Z=0.09), ...
          ... Direction=(Y=1.5707964,Z=-3.1415927,X=0.0))

r   CamTexActors=(ItemClass=class'USARBot.USAREmitter', ...
          ... ItemName="OmniCamEmitter", ...
          ... Parent="OmniCamera_Link0", ...
          ... Position=(Y=0.0,X=0.0,Z=0.0), ...
          ... Direction=(Y=0.0,Z=0.0,X=0.0))
```

As can be seen in UnrealScript B.5, the KRobot class contains a variable called `CamTexActors`, which is a struct with several members. Line `q` in USARBot.ini shows how the members of this struct are assigned.

Line `p` in USARBot.ini will make the KRobot class spawn the Omnidirectional Camera Static Mesh. The `Position` vector defines the precise location of the mirror's centre of projection, as

the origin of the Static Mesh lies on that point. The reason for this is twofold. First, by placing the origin at the centre of projection, it minimises the possibility of erroneous placement of the USAREmitters elements. Second, by being able to define the exact location of the centre of projection, one is able to define one of the most important parameters which influence any visual methods using the Omnidirectional Camera data as an input. In USARBot.ini the MisPkg is placed at `(Y=0.0,X=0.0,Z=0.0)`.

The `Cameras` variable declaration places the actual recording camera at the proper position and direction for it to record the omnidirectional view. Its relative `Position` to the Static Mesh position should be `(X=+0.0,Y=+0.0,Z=+0.09)` so it is at the right distance from the mirror surface. Its relative `Direction` should be `(Y=+1.5707964,Z=-3.1415927,X=+0.0)` so it is aimed directly at the mirror. This rotation also places the `Direction` of the Static Mesh at the top of the resulting omnidirectional image.

B.7 The MisPkg.ini File

UnrealScript B.7 displays an excerpt of the USARMisPkg.ini file, in which mission packages are defined and configured, as mentioned in section 2.2.1).

In this this excerpt it can be seen that mission package definitions is divided in two phases. First, the links within the tree structure are configured[1], in which mount points are positioned and joint types are defined. Second, the complete package tree structure is defined as links are numbered and connected.

[1]Though the term *links* suggests they form a *chain* structure, they are as a matter of fact, technically speaking, *nodes* in a *tree* structure. The naming convention is slightly misleading.

UnrealScript B.7 Excerpt of MiskPkg.ini defining the structure of the camera mission package.

```
[USARMisPkg.OmniCamBase]
MountPoints=(Name="A", ...
         ... JointType="Revolute", ...
         ... Location=(X=0.0,Y=0.0,Z=0.0), ...
         ... Orientation=(X=0,Y=0,Z=0))
MountPoints=(Name="B", ...
         ... JointType="Revolute", ...
         ... Location=(X=0.0,X=0.0,Z=0.0), ...
         ... Orientation=(X=0,Y=0,Z=0))
MinRange=0
MaxRange=0

[USARMisPkg.OmniCam1024Mirror]
MountPoints=(Name="A", ...
         ... JointType="Revolute", ...
         ... Location=(X=0.0,Y=0.0,Z=0.0), ...
         ... Orientation=(X=0,Y=0,Z=0))
MinRange=0
MaxRange=0

...

[USARMisPkg.OmniCamera]
Links=(LinkNumber=0,LinkClass=Class'USARMisPkg.OmniCamBase', ...
   ... DrawScale3D=(X=1.0,Y=1.0,Z=1.0), ...
   ... ParentLinkNumber=-1, ...
   ... SelfMount="A")
Links=(LinkNumber=1,LinkClass=Class'USARMisPkg.OmniCam1024Mirror', ...
   ... DrawScale3D=(X=1.0,Y=1.0,Z=1.0), ...
   ... ParentLinkNumber=0, ...
   ... ParentMount="B",SelfMount="A")
```

Bibliography

[1] voor de Statistiek, C.B.: Cbs - begrippen. On-line. Available from Internet, http://www.cbs.nl/nl-NL/menu/methoden/begrippen/default.htm?conceptid=2384 (2008) accessed: January 28th, 2008.

[2] Gibson, L.: Usf robots sought survivors in world trade center rubble. On-line. Available from Internet, http://www.sptimes.com/News/092801/Hillsborough/USF_robots_sought_sur.shtml (2001) accessed: January 28th, 2008.

[3] Casper, J.: Human-robot interactions during the robot-assisted urban search and rescue response at the world trade center. Master's thesis, Computer Science and Engineering, University- of South Florida (2002)

[4] Greer, D., McKerrow, P., Abrantes, J.: Robots in urban search and rescue operations. In: ACRA, Aukland (2002) 25–30

[5] Murphy, R., Casper, J., Micire, M., Hyams, J.: Assessment of the nist standard test bed for urban search and rescue competitions (2000)

[6] Wolf, A., Choset, H.H., Jr., B.H.B., Casciola, R.W.: Design and control of a mobile hyper-redundant urban search and rescue robot. Advanced Robotics **21** (2005) 221–248

[7] Ren, J., McIsaac, K.A., Patel, R.V.: A novel hybrid navigation scheme for reconfigurable multi-agent teams. Int. J. Robot. Autom. **21** (2006) 100–109

[8] Gauthier, M., Anderson, J.: Peer instruction for a teleautonomous usar system. In: Proceedings of the Third International Conference on Computational Intelligence, Robotics, and Autonomous Systems (CIRAS), Singapore (2005)

[9] Wegner, R., Anderson, J.: Balancing robotic teleoperation and autonomy for urban search and rescue environments. In: Canadian Conference on AI. Volume 3060 of Lecture Notes in Computer Science., Springer (2004) 16–30

[10] Jacoff, A., Messina, E., Weiss, B., Tadokoro, S., Nakagawa, Y.: Test arenas and performance metrics for urban search and rescue robots. Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on **4** (2003)

[11] Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E.: Robocup: The robot world cup initiative. In Johnson, W.L., Hayes-Roth, B., eds.: Proceedings of the First International Conference on Autonomous Agents (Agents'97), New York, ACM Press (1997) 340–347

[12] Tadokoro, S., Kitano, H., Takahashi, T., Noda, I., Matsubara, H., Shinjoh, A., Koto, T., Takeuchi, I., Takahashi, H., Matsuno, F., Hatayama, M., Nobe, J., Shimada, S.: The robocup-rescue project: A robotic approach to the disaster mitigation problem. In: ICRA. (2000) 4089–4094

[13] Carpin, S., Lewis, M., Wang, J., Balakirsky, S., Scrapper, C.: Usarsim: a robot simulator for research and education. In: ICRA. (2007) 1400–1405

[14] Jacoff, A., Messina, E., Evans, J.: A standard test course for urban search and rescue robots. In: Proceedings of the Performance Metrics for Intelligent Systems Workshop. (2000)

[15] Deering, M.: The limits of human vision. 2nd International Immersive Projection Technology Workshop (1998)

[16] Rees, D.: Panoramic television viewing system. In: US Patent. (1970) http://www.visionbib.com/bibliography/compute71.html#TT3955, accessed January 8 2007.

[17] Bunschoten, R., Kröse, B.: Robust scene reconstruction from an omnidirectional vision system. IEEE Transactions on Robotics and Automation **19** (2003) 351–357

[18] Bunschoten, R., Kröse, B.: Visual odometry from an omnidirectional vision system. In: Proceedings of the International Conference on Robotics and Automation ICRA'03, Taipei, Taiwan (2003) 577–583 ISBN 0-7803-7737-0.

[19] Kröse, B., Bunschoten, R., Vlassis, N., Motomura, Y.: Appearance based robot localization. In Kraetzschmar, G., ed.: Proc. IJCAI-99 Workshop on adaptive spatial representations of dynamic environments. (1999) 53–58

[20] Kröse, B., Vlassis, N., Bunschoten, R.: Omnidirectional vision for appearance-based robot localization. In Hagar, G., Cristensen, H., Bunke, H., Klein, R., eds.: Sensor Based Intelligent Robots: International Workshop, Dagstuhl Castle, Germany, October 2000, Selected Revised Papers. Number 2238 in Lecture Notes in Computer Science. Springer (2002) 39–50

[21] Heinemann, P., Streichert, F., Sehnke, F., Zell, A.: Automatic calibration of camera to world mapping in robocup using evolutionary algorithms. In: Proceedings of the IEEE International Congress on Evolutionary Computing (CEC 2006). (2006) 1316 – 1323

[22] Heinemann, P., Haass, J., Zell, A.: A combined monte-carlo localization and tracking algorithm for robocup. In: Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '06). (2006) 1535–1540

[23] Aihara, N., Iwasa, H., Yokoya, N., Takemura, H.: Memory-based self-localization using omnidirectional images. Systems and Computers in Japan **34** (2003) 56–68

[24] Rizzi, A., Cassinis, R.: A robot self-localization system based on omnidirectional color images. Robotics and Autonomous Systems **34** (2001) 23–38

[25] Lima, P., Bonarini, A., Machado, C., Marchese, F.M., Marques, C., Ribeiro, F., Sorrenti, D.G.: Omnidirectional catadioptric vision for soccer robots. Journal of Robotics and Autonomous Systems **36** (2001) 87–102

[26] Baker, S., Nayar, S.K.: A theory of single-viewpoint catadioptric image formation. International Journal of Computer Vision **35** (1999) 1 – 22

[27] Jeong, K., Kim, J., Lee, L.: Biologically Inspired Artificial Compound Eyes. Science **312** (2006) 557–561

[28] Fraser, C.: Digital camera self-calibration. ISPRS Journal of Photogrammetry and Remote Sensing **52** (1997) 149–159

[29] Remondino, F., Fraser, C.: Digital camera calibration methods: considerations and comparisons. Proc. ISPRS Commission V Symposium (2006) 266–272

[30] Har-Noy, S., Nguyen, T.: A Deconvolution Method for LCD Motion Blur Reduction. Image Processing, 2006 IEEE International Conference on (2006) 629–632

[31] Taylor, B.K., Balakirsky, S.B., Messina, E.R., Quinn, R.D.: Design and validation of a whegs robot in usarsim. In: Performance Metrics for Intelligent Systems (PerMIS) 2007. (2007)

[32] Inc., P.G.R.: Imaging products - dragonfly2. On-line. Available from Internet, http://www.ptgrey.com/products/dragonfly2/index.asp (2007) accessed: December 20th, 2007.

[33] Seiwapro Co. Formerly named Accowle Company, L.: Ltd. panorama eye. On-line. Available from Internet, http://www.accowle.com/english/index.html. (2004) accessed: December 22th, 2007.

[34] Nayar, S.: Omnidirectional vision. In: Proc. of Eighth International Symposium on Robotics Research (ISRR). (1997) http://citeseer.ist.psu.edu/article/nayar97omnidirectional.html, accessed: July 13 2007.

[35] Jr., V.G., Jr., J.O.: Development of an omnidirectional vision system. Journal of the Brazilian Society of Mechanical Sciences and Engineering **28** (2006) 58 – 68

[36] Visser, A., Slamet, B., Schmits, T., González Jaime, L.A., Ethembabaoglu, A.: Design decisions of the UvA Rescue 2007 Team on the Challenges of the Virtual Robot competition. In: Proc. 4th International Workshop on Synthetic Simulation and Robotics to Mitigate Earthquake Disaster. (2007) 20–26

[37] Schlesinger, S., et al.: Terminology for model credibility. Simulation **32** (1979) 103–104

[38] Carpin, S., Stoyanov, T., Nevatia, Y., Lewis, M., Wang, J.: Quantitative assessments of usarsim accuracy. In: PerMIS 2006. (2006)

[39] Jones, M., Rehg, J.: Statistical Color Models with Application to Skin Detection. International Journal of Computer Vision **46** (2002) 81–96

[40] Kalman, R.: A new approach to linear filtering and prediction problems. Journal of Basic Engineering **82** (1960) 35–45

[41] Kristensen, S., Jensfelt, P.: An experimental comparison of localisation methods, the MHL sessions. Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on **1** (2003)

[42] Zaratti, M., Fratarcangeli, M., Iocchi, L.: A 3D Simulator of Multiple Legged Robots Based on USARSim. In Lakemeyer, G., Sklar, E., Sorrenti, D., Takahashi, T., eds.: RoboCup 2006: Robot Soccer World Cup X. Volume 4434 of Lecture Notes on Artificial Intelligence., Berlin Heidelberg New York, Springer (2007) 13–24

[43] Woods, D., Tittle, J., Feil, M., Roesler, A.: Envisioning human-robot coordination in future operations. Systems, Man and Cybernetics, Part C, IEEE Transactions on **34** (2004) 210–218

[44] Wang, J., Lewis, M., Hughes, S., Koes, M., Carpin, S.: Validating usarsim for use in hri research. In: Proceedings of the 49th Annual Meeting of the Human Factors and Ergonomics Society. (2005)

[45] Slamet, B., Pfingsthorn, M.: ManifoldSLAM: a Multi-Agent Simultaneous Localization and Mapping System for the RoboCup Rescue Virtual Robots Competition. PhD thesis, (Masters thesis, Universiteit van Amsterdam, December 2006)

[46] Wang, J., Balarski, S.: Usarsim v3.1.1 - a game-based simulation of mobile robots. On-line. Available from Internet, http://downloads.sourceforge.net/usarsim/USARsim-manual_3.1.1.pdf (2007) accessed: Januari 18th, 2008.

[47] Kaminka, G., Veloso, M., Schaffer, S., Sollitto, C., Adobbati, R., Marshall, A., Scholer, A., Tejada, S.: Gamebots: a flexible test bed for multiagent team research. Commun. ACM **45** (2002) 43–45