

Scalable and Extensible Augmented Reality with Applications in Civil Infrastructure Systems

By
Suyang Dong

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Civil Engineering)
in The University of Michigan
2012

Doctoral Committee:

Associate Professor Vineet R. Kamat, Chair
Lecturer John G. Everett
Associate Professor Sugih Jamin
Assistant Professor SangHyun Lee

© Suyang Dong 2012
All Rights Reserved

To My Deceased Grandpa,
May he be peaceful in the heaven,
His independent spirit is the beacon in my life.

To My Lovely Parents,
They bestowed the life upon me twice,
Time left wrinkles on their face,
But left their endless love in my heart.

To My Darling,
In happiness or sorrow,
Who held my hands everyday.

Acknowledgments

I would like to give my wholehearted thanks my advisor in the academia and mentor in the life, Professor Vineet R. Kamat, who opened the door of Augmented Reality for me four years ago, guided and supported me to strive for the achievement of my research goals in the past four years, and now encourages me to dive into another challenging chapter in my life. The descriptions about his diligence, wisdom and genuineness are beyond my words, His trust inspires me to explore the unknown; his patience allows me to delve into the core; his experience pulls me out of the pitfall; his sincerity polishes my writings; and his friendship and smile dwells in my heart.

I would like to express my deepest gratitude to Professor John G. Everett, who built up the fundamentals about construction practice in my knowledge pool. He is admired by every student he taught, including me. It was also my privilege and honor to work with him for two years, and his care and patience for his students touched me again and again.

I would like to gratefully and sincerely thank Professor Sugih Jamin, whose sincerity and patience helped me to learn most important concepts of computer graphics in my research. There is so much knowledge that I learnt from his course being applied in my

dissertation, and it would be much harder to make my way towards the achievement of this dissertation without his help.

I am also indebted to Professor SangHyun Lee for his invaluable advice, insightful comments, and constructive criticism to improve the quality of my work, and accomplish the challenge in a timely manner.

I am deeply grateful to my colleagues Chen Feng, Fei Dai, Sanat Talmaki, and Manu Akula. Their thought-provoking suggestions shed light on my research methodologies; their generous help greatly accelerated the accomplishment of my work.

Most importantly, none of this would happen without the love and support of my parents and my wife. My parents, Changqing Dong and Meizhen Wang, may not agree with my decisions, like coming to the States or being an entrepreneur, but their endless love make them be supportive of every goal I go for and every action I take. During the last year of my doctoral study, the most lovely and competent wife in the world, Yao Nie, comes and joins me and takes care of my life—cook, wash, and clean day after day. Thank you, my darling!

Table of Contents

Dedication	ii
Acknowledgements	iii
List of Tables	xi
List of Figures	xii
List of Appendices	xvi
List of Abbreviations	xvii
Abstract	xix
Chapter 1. Introduction	1
1.1 Importance of the Research Activity	2
1.1.1 Previous Work in Augmented Reality	4
1.1.1.1 Manufacturing and Factory Planning	5
1.1.1.2 Medical Image Data Visualization	6
1.1.1.3 Education, Training, and Guidance	7
1.1.1.4 Entertainment, Media, and Commerce	7
1.1.1.5 Mobile Applications	8
1.1.2 Augmented Reality in the Construction Domain	9
1.1.3 Augmented Reality Research in LIVE	12
1.1.4 Challenges for Construction AR Applications	13
1.2 Research Objectives	15
1.3 Research Methodology	18
1.4 Dissertation Outline	20
1.5 References	24
Chapter 2. Robust Mobile Computing Framework for Visualization of Simulated Processes in Augmented Reality	29
2.1 Introduction	29
2.1.1 Importance of the Research	30

2.1.2	Main Contribution	32
2.2	ARMOR Hardware Architecture	33
2.2.1	Tracking Devices	35
2.2.1.1	Orientation Tracking Device—Electronic Compass	35
2.2.1.2	Position Tracking Device — RTK-GPS	36
2.2.2	Input/Output Devices and External Power Supply	38
2.2.2.1	Video Sequence Input: Camera	38
2.2.2.2	Augmented View Output: Head-mounted Display (HMD)	38
2.2.2.3	External Power Supply	39
2.2.2.4	User Command Input: Nintendo Wii Remote	40
2.2.2.5	Load-Bearing Vest	40
2.3	SMART Software Framework	41
2.3.1	Application for Operation-Level Construction Animation	43
2.4	Static Registration	44
2.4.1	Registration Process	44
2.4.1.1	Step 1 – Viewing	45
2.4.1.2	Step 2 – Modeling	47
2.4.1.3	Step 3 and 4 – Viewing Frustum and Projection	48
2.4.2	Registration Validation Experiment	49
2.4.2.1	Calibration of the Mechanical Attitude Discrepancy	49
2.4.2.2	Validation of the Static Registration Algorithm	49
2.5	Resolving the Latency Problem in the Electronic Compass	50
2.5.1	Multi-threading to Reduce Synchronization Latency	51
2.5.2	Experiment to Determine the Latency under Push Mode	52
2.5.3	Filter-induced Latency	55
2.5.4	Half-Window Gaussian Filter	56
2.5.5	Adaptive Latency Compensation Algorithm	57
2.6	Validation	58
2.7	Conclusion and Future Work	61
2.8	References	62

Chapter 3. Real-Time Occlusion Handling for Dynamic Augmented Reality Using Geometric Sensing and Graphical Shading **65**

3.1	Introduction	65
3.2	Depth Buffer Comparison Approach	69
3.2.1	Distance Data Source	69
3.2.2	Two-stage Rendering	70
3.2.3	Challenges with the Depth Buffering Comparison Approach	71
3.3	TOF Camera Raw Data Preprocessing	73
3.3.1	Preprocessing of Depth Map	73
3.3.2	Processing of the Intensity Image	75
3.4	Depth Map and RGB Image Registration	76
3.4.1	Image Registration Between Depth Map and Intensity Image	76
3.4.2	Image Registration Between Depth Map and RGB Image Using Homography	77
3.4.3	Image Registration Between Depth Map and RGB Image Using Stereo Projection	80
3.4.3.1	Assign RGB Value to Depth Map Pixel	81
3.4.3.2	Interpolate RGB Values for Intermediate Depth Map Pixels	82
3.5	Technical Implementation with OpenGL Texture and GLSL	84
3.5.1	Interpolation using OpenGL Texture	84
3.5.1.1	Intensity Image Texture	85
3.5.1.2	Depth Map Texture	86
3.5.2	Homography Registration Using Render to Texture	86
3.5.3	Stereo Registration Using Render to Texture	88
3.6	Validation	89
3.7	Conclusion and Future Work	91
3.8	References	93

Chapter 4. Sensitivity Analysis of Augmented Reality-Assisted	96
Building Damage Reconnaissance Using Virtual Prototyping	
4.1 Introduction	96
4.2 Review of Previous Work	98
4.3 Overview of Reconnaissance Methodology	100
4.4 Technical Approach	103
4.4.1 Reconfigurable Virtual Prototype of Seismically Damaged Building	105
4.4.2 Damage Modeling	106
4.4.3 Camera Modeling	108
4.4.3.1 External Parameters	108
4.4.3.2 Internal Parameters	110
4.4.4 Vertical Edge Detection	111
4.4.4.1 Active Contour Approach	111
4.4.4.2 Line Segment Detection Approach	113
4.4.5 Corner Detection	114
4.4.5.1 Horizontal Edge Detection	114
4.4.5.2 Corner Detection Algorithm	117
4.4.5.3 Interstory Drift Ratio Calculation	119
4.5 Evaluation of Experimental Results	119
4.5.1 Experiment with Ground True Location and Orientation	120
4.5.1.1 Observing Distance	121
4.5.1.2 Observing Angle	122
4.5.1.3 Drift Interval	123
4.5.1.4 Approximate Versus Accurate	124
4.5.1.5 Estimation Step	124
4.5.1.6 Image Resolution	125
4.5.1.7 Automatic versus Manual	126
4.5.2 Experiments with Instrument Error	126
4.6 Conclusion	129
4.7 References	131

Chapter 5. Collaborative Learning of Engineering Processes	135
Using Tabletop Augmented Reality Visual Simulations	
5.1 Introduction	135
5.2 Main Contributions	140
5.3 3D Visualization of Engineering Operations	141
5.3.1 Virtual Reality Visualization of Engineering Operations	141
5.3.2 Augmented Reality Visualization of Engineering Operations	143
5.3.3 Collaborative Learning through 3D Visualization	145
5.4 Technical Implementation of ARVita	149
5.4.1 Model-View-Controller Software Architecture of ARVita	149
5.4.2 Implementation of Model-View-Controller Using OpenSceneGraph	150
5.4.2.1 Model	151
5.4.2.2 View	152
5.4.2.3 Controller	154
5.5 Planar tracking methods for Collaborative AR	155
5.5.1 Principle of Tracking	156
5.5.2 Taxonomy of Tracking Methods	156
5.5.3 Trackers Available in ARVita	157
5.5.3.1 The Fiducial Marker Tracking Method, ARToolkit as an Example	157
5.5.3.2 The Natural Marker Tracking Method, KEG as an Example	158
5.5.3.2.1 Detection-based Method	160
5.5.3.2.2 Tracking-based Method	160
5.5.3.2.3 Global Appearance and Geometric Constraints	161
5.5.3.2.4 The Introduction of AprilTag	162
5.6 Multiple Views in ARVita	163
5.6.1 Technical Implementation of Multiple View	163
5.6.2 Limitations of Multiple Views on a Single Computer	165
5.7 Conclusion and Future Work	165
5.8 References	167

Chapter 6. Conclusion	172
6.1 Significance of the Research	172
6.2 Contributions	174
6.3 Directions for Future Research	180
6.3.1 Augmented Reality with Photorealistic Effect and Interaction Functionality	
180	
6.3.2 Augmented Reality on Mobile Devices	182
6.3.3 Seamless and Collaborative Augmented Reality with Cloud Computing	183
6.4 References	184
Appendices	186

List of Tables

Table 2.1 -	Comparison between the UM-AR-GPS-ROVER and ARMOR configuration.	34
Table 2.2 -	Power voltage demands of different devices.	39
Table 2.3 -	The four steps of registration process.	44
Table 2.4 -	Mechanical attitude calibration result and validation experiment of registration algorithm.	50
Table 3.1 -	The transformation steps applied on the raw TOF depth image.	74
Table 4.1 -	Sensitivity of drift error to observation distance.	121
Table 4.2 -	Sensitivity of drift error to observing angle.	122
Table 4.3 -	Sensitivity of drift error to drift interval.	123
Table 4.4 -	Sensitivity of drift error to accurate and approximate estimation modes.	124
Table 4.5 -	Sensitivity of drift error to estimation step.	125
Table 4.6 -	Sensitivity of drift error to image resolution.	126
Table 4.7 -	Sensitivity of drift error to manual and automatic detection modes.	126
Table 5.1 -	Comparison between two natural marker approaches.	161

List of Figures

Figure 1.1 -	Application examples of AR in different domains.	5
Figure 1.2 -	Total mobile AR market by 2015, split into 7 categories.	9
Figure 1.3 -	Application examples of AR in construction areas.	10
Figure 1.4 -	AR Research for Construction Applications in LIVE.	12
Figure 1.5 -	Research Objectives of this dissertation.	17
Figure 2.1 -	The profile of ARMOR from different perspectives.	41
Figure 2.2 -	SMART framework architecture.	43
Figure 2.3 -	Definition of the world coordinate system.	45
Figure 2.4 -	The relative orientation between the world and eye coordinate systems is described by yaw, pitch, and roll.	46
Figure 2.5 -	The viewing frustum defines the virtual content that can be seen.	48
Figure 2.6 -	Communications stages in the PULL mode and the PUSH mode.	51
Figure 2.7 -	Comparison between the TCM-XB data log and the corresponding recorded image frames. The shaded area highlights the exact instant that the module started swinging.	53
Figure 2.8 -	The data shows that the noise in the raw data increases as the motion accelerates.	54
Figure 2.9 -	The Filter-induced latency when a 32 tap Gaussian filter is used.	56
Figure 2.10 -	Half-window filter latency.	57
Figure 2.11 -	Adaptive Latency Compensation algorithm.	59
Figure 2.12 -	Standard deviation indicates the motion pattern.	61
Figure 2.13 -	Conduit loading procedure, conduits overlaid on Google Earth and field experiment results.	62
Figure 2.14 -	Labeling attribute information and color coding on the underground utilities.	62
Figure 2.15 -	An x-ray view of the underground utilities.	62
Figure 3.1 -	Example of occlusion in an outdoor scene.	67

Figure 3.2 -	Two-stage rendering.	71
Figure 3.3 -	Projection parameters disagreement between the TOF camera and the video camera.	72
Figure 3.4 -	Preprocessing of the TOF intensity and depth image.	76
Figure 3.5 -	Occlusion effects comparison using the TOF camera's intensity image and depth map.	77
Figure 3.6 -	The identical points on two images are used to calculate the homography matrix that registers the RGB image with the TOF depth image.	78
Figure 3.7 -	Occlusion effects comparison using homography mapping between the TOF camera and the RGB camera.	79
Figure 3.8 -	The homography approximation fails to correctly resolve occlusion on the silhouette.	80
Figure 3.9 -	The stereo registration between the TOF and RGB cameras.	81
Figure 3.10 -	Chessboard method used for extrinsic calibration between the TOF and RGB cameras.	82
Figure 3.11 -	Attaching multiple textures to the quad for hardware interpolation.	85
Figure 3.12 -	Using Render to Texture to compute the homography registration on the GPU.	88
Figure 3.13 -	Using Render to Texture to compute the stereo registration on the GPU.	89
Figure 3.14 -	Indoor simulated construction processes with occlusion disabled and enabled.	90
Figure 3.15 -	Outdoor simulated construction processes with occlusion disabled and enabled.	92
Figure 4.1 -	Schematic overview of the proposed AR-assisted assessment methodology.	101
Figure 4.2 -	Major steps to reconstruct the 3D coordinates of key locations on the building.	102
Figure 4.3 -	Graphical discrepancy between the vertical baseline and the detected building edge provides hints about the magnitude of the local damage.	103
Figure 4.4 -	A ten-story graphical building model is constructed as its macro model counterpart.	106
Figure 4.5 -	The OpenGL camera replicates a physical camera to take pictures of the key locations.	107

Figure 4.6 -	Internal structural damage, shift of the vertex, is expressed through the displacement of the texture.	108
Figure 4.7 -	The near plane of the OpenGL camera is the counterpart of the physical camera image sensor chip, a device that converts an optical image into an electronic signal.	110
Figure 4.8 -	LSD (right) outperforms GCBAC (middle) when searching for localized line segments, for example building edges.	112
Figure 4.9 -	A geometric filter plus minimal manual reinforcement can rapidly eliminate most irrelevant line segments.	114
Figure 4.10 -	The detectable gap between the original baseline and real edge enlarges as the camera gets closer to the building.	115
Figure 4.11 -	Alignment of Horizontal Baseline: a) shifts the two ends of the baseline with different distances and costs Θ (N4), while (b) shifts the two ends of the baseline with the same distance and costs Θ (N2).	116
Figure 4.12 -	Flowchart of the proposed corner detection algorithm.	118
Figure 4.13 -	Interstory Drift Ratio Calculation.	120
Figure 4.14 -	Observing angle of the camera.	123
Figure 4.15 -	Sensitivity of computed drift to camera position errors.	127
Figure 4.16 -	Sensitivity of computed drift to camera orientation errors.	128
Figure 4.17 -	Sensitivity of computed drift to camera location and orientation errors.	130
Figure 5.1 -	A survey of undergraduate civil, environmental, and construction engineering students revealed that a large percentage of students support the prospect of reforming current instructional methods.	137
Figure 5.2 -	The view of the real world is used as a readymade backdrop for displaying superimposed information in AR.	140
Figure 5.3 -	Traditional paper-based media is ideal for collaborative work, despite disadvantages such as difficulty in handling, maintaining, and updating.	146
Figure 5.4 -	The software architecture of ARVita conforms to the Model-View-Controller pattern. The arrow indicates a 'belongs to' relationship.	149
Figure 5.5 -	The realization of the Model-View-Controller model with OpenSceneGraph.	151
Figure 5.6 -	osgART's Tracker and Marker updating mechanism.	152
Figure 5.7 -	Two users are observing the animation lying on the table.	154

Figure 5.8 -	Steel erection activities at different timestamps.	155
Figure 5.9 -	The taxonomy of tracking methods.	157
Figure 5.10 -	The natural marker tracking library is flexible on marker visibility.	159
Figure 5.11 -	The algorithm flowchart of the KEG tracker.	162
Figure 5.12 -	All of the views possess their own video, tracker, and marker objects, but point to the same VITASCOPE scene node.	164
Figure 6.1 -	The comparison between the photorealistic AR image and the non-photorealistic one.	181
Figure 6.2 -	Interactions between the synthetic and physical objects in simulated construction processes.	182
Figure D.1 -	ARVita Logo.	301
Figure D.2 -	Camera Selection.	302
Figure D.3 -	Marker Selection.	302
Figure D.4 -	The user interface of ARVita.	305

List of Appendices

Appendix A -	SMART Code Documentation	187
Appendix B -	Occlusion Algorithm Pseudo Code	272
Appendix C -	Aurora Code Documentation	276
Appendix D -	ARVita Quick Start Manual	300
Appendix E -	Biography	306

List of Abbreviations

AR	Augmented Reality
ARMOR	Augmented Reality Mobile Operation platform.
ARVISCOPE	Augmented Reality Visualization of Simulated Construction Operations
ARVita	Augmented Reality VITASCOPE
BRDF	Bidirectional Reflectance Distribution Function
CMR	Compact Measurement Record
CPU	Central Processing Unit
DES	Discrete Event Simulation
FIR	Finite Impulse Response
FPS	Frame Per Second
GLSL	OpenGL Shading Language
GPU	Graphics Processing Unit
GRABC	Graph-Cut Based Active Contour
HMD	Head Mounted Display

IDR	Interstory Drift Ratio
LBS	Location Based Service
LSD	Line Segment Detector
LIVE	Laboratory for Interactive Visualization in Engineering
OpenCV	Open Source Computer Vision Library
OpenGL	Open Graphics Language
OSG	OpenSceneGraph
PPM	Parts Per Million
RMS	Root Mean Square
RTK	Real-time Kinematics
RTT	Render to texture
SMART	Scalable and Modular Augmented Reality Template
SLR	Single Lens Reflex
TOF	Time Of Flight
VITASCOPE	VisualizaTion of Simulated Construction OPERations
VP	Virtual Prototyping
VR	Virtual Reality

Abstract

In Civil Infrastructure System (CIS) applications, the requirement of blending synthetic and physical objects distinguishes Augmented Reality (AR) from other visualization technologies in three aspects: 1) it reinforces the connections between people and objects, and promotes engineers' appreciation about their working context; 2) It allows engineers to perform field tasks with the awareness of both the physical and synthetic environment; 3) It offsets the significant cost of 3D Model Engineering by including the real world background.

The research has successfully overcome several long-standing technical obstacles in AR and investigated technical approaches to address fundamental challenges that prevent the technology from being usefully deployed in CIS applications, such as the alignment of virtual objects with the real environment continuously across time and space; blending of virtual entities with their real background faithfully to create a sustained illusion of co-existence; integrating these methods to a scalable and extensible computing AR framework that is openly accessible to the teaching and research community, and can be readily reused and extended by other researchers and engineers.

The research findings have been evaluated in several challenging CIS applications where the potential of having a significant economic and social impact is high. Examples of validation test beds implemented include an AR visual excavator-utility collision avoidance system that enables spotters to "see" buried utilities hidden under the ground surface, thus helping prevent accidental utility strikes; an AR post-disaster reconnaissance framework that enables building inspectors to rapidly evaluate and quantify structural damage sustained by buildings in seismic events such as earthquakes or blasts; and a tabletop collaborative AR visualization framework that allows multiple users to observe and interact with visual simulations of engineering processes.

Chapter 1

Introduction

The objective of this research was to attempt the mitigation of some long-term Augmented Reality (AR) technical obstacles—such as registration and occlusion—that prevent AR from being usefully deployed in construction and other engineering applications. The visualization approach aimed to achieve two goals: one, to augment the views of engineers and workers with auxiliary graphical information that helps them accurately and rapidly identify critical components in the ongoing operations, and two, to improve safety and efficiency in the conduct of relevant tasks. Such augmented visual information should be presented with stable registration and robust occlusion so that users have high confidence in the augmented graphics instead of being misguided or distracted by visual artifacts. The achieved results must be readily deployable in various construction and other engineering applications that can potentially benefit from AR.

The end result of this research effort is SMART, an acronym for Scalable and Modular Augmented Reality Template. SMART is an extensible AR computing framework that is intended to deliver high-accuracy and convincing augmented graphics that correctly place virtual contents relative to a real scene, and robustly resolve the occlusion relationships between them. Furthermore, the AR computing framework is loosely coupled so that it is independent of any specific engineering application or domain. Instead, it can be readily adapted to an array of engineering applications such as visual collision avoidance of underground facilities, post-disaster reconnaissance of damaged buildings, and visualization of simulated construction processes. In partial fulfillment of this goal, SMART and all of the example applications are made available as open source code to the research and professional community, who can take advantage of the extensible AR framework and concentrate on building the logic component of their own AR applications in the future.

1.1 Importance of the Research Activity

Visualization usually refers to the presentation of 2D or 3D graphics to improve a user's cognition or learning experience. An important variant of visualization is called Virtual Reality, which attempts to replace the physical world with a totally synthetic environment. The user's sensory receptors, like eyes and ears, are isolated from the real physical world and completely immersed in the synthetic environment that replicates the physical world to some extent. The communication inside VR is dual channel. Besides receiving messages from the virtual reality, the user's intent when interacting with the

virtual contents can be translated through hardware like pointing devices or haptic systems. There is a wide array of applications now commonly associated with VR, for example CAD engineering, scientific visualization, visual simulation, animation, computer games, and virtual training.

Even though VR provides a stable, robust, interactive, and immersive experience, the cost of constructing a faithful synthetic environment can be enormous (Brooks, 1999). First, before an involved element or resource can be integrated into the system, its 3D model has to be created, refined, archived, and maintained. This process—also referred to as CAD Model Engineering—accounts for the majority of efforts in a VR-based animation. The acquisition of models typically comes from one of three ways: build models from scratch, inherit or purchase models from others' work, or construct models by sensing real objects. Any way can be costly and labor-intensive. Secondly, as the synthetic environment evolves to be more true to the real world, the increase in models' complexity raises the difficulty of managing and rendering the graphics.

In contrast to the VR philosophy of replicating the real world by faithful synthesis, another variant of visualization approach, called Augmented Reality (AR), attempts to preserve the user's awareness of the real environment by compositing the real world and the virtual contents in a mixed 3D space. In particular, AR refers to the visualization technology that blends virtual objects with the real world (Azuma, et al., 2001). For that purpose, AR must not only maintain a correct and consistent spatial relation between the virtual and real objects, but also sustain the illusion that they co-exist in the augmented space.

The blending effect reinforces the connections between people and objects, promotes people's appreciation about their context, and provides hints for the users to discover their surroundings. Further, the awareness of the real environment and the information conveyed by the virtual objects help users to perform real-world tasks, whereas VR applications are mainly restricted to designing, running simulations, and training (Azuma, 1997). Furthermore, AR offers a promising alternative to the Model Engineering challenge inherent in VR by only including entities that capture the essence of the study (Behzadan and Kamat, 2005). These essential entities usually exist in a complex and dynamic context that is necessary to the model, but costly to replicate in VR. However, reconstructing the context is rarely a problem in AR, where modelers can take full advantage of the real context (terrains and existing structures, for example), and render them as background, thus saving a considerable amount of effort and resources.

1.1.1 Previous Work in Augmented Reality

As early as 1966, Ivan Sutherland invented the first VR and AR head-mounted display. Since the 1990s, along with the maturity of computer graphics software and hardware techniques, AR gained popularity in a variety of applications in different domains (Krevelen and Poelman, 2010). Here, only a few representative areas are briefed: manufacturing, medical treatment, education, and entertainment (Figure 1).



Manufacturing



Medical



Education



Entertainment

Figure 1.1 Application examples of AR in different domains. (Pentenrieder, et al., 2007, Bichlmeier, et al., 2007a, Billinghurst, et al., 2001, Thomas, et al., 2000)

1.1.1.1 Manufacturing and Factory Planning

In the manufacturing domain, particularly in the automotive industry, AR applications have been witnessed in the phases of prototyping, assembly, and maintenance. Klinker, et al. (2002) developed a prototype AR system called Fata Morgana to demonstrate a life-size car model for the design and review process at the BMW workshop. Fiorentino, et al. (2002) addressed the challenge in the aesthetic design of free-form curves and surfaces with the Spacedesign system that features the intuitive sketching and modeling of an automotive design directly in the 3D AR space. Another automotive design example comes from Volkswagen (Nolle and Klinker, 2006), where they used AR to check if the manufactured components comply with the latest version of the design, and if they satisfy

the appropriate precision. AR is not only valuable for engineers in the design phases, but is also helpful for technical staff in their maintenance duties. For example, BMW developed a mechanical guidance system that illustrates the disassembling and reassembling processes step by step in the data goggles worn by technicians (BMW, 2007).

AR has also been considered for use in factory planning and inspection. At Siemens Corporate Research, a fully implemented system called CyliCon (Navab, 2003) enables the user to maneuver and visualize as-built reconstruction models in the real site or on the industrial drawings. Volkswagen puts strict demands on the system accuracy of their assembly line (e.g., interfering edge analysis and aggregation verification), and they found that AR-supported factory planning offers an easy, fast, and affordable means of meeting the challenge (Pentenrieder, et al., 2007).

1.1.1.2 Medical Image Data Visualization

The traditional way of interpreting imaging data—like Magnetic Resonance Imaging (MRI), Computed Tomography scans (CT), or ultrasound imaging—requires intensive medical training, knowledge, and experience. However, with the aid of AR, these imaging data can be analyzed in the context of the patient’s body to provide a surgeon with a natural and direct 3D viewing. Bichlmeier, et al. (2007a) developed an in-situ visualization system that allows for the improved perception of 3D medical imaging data and navigated surgical instruction relative to the patient’s anatomy. They also (Bichlmeier, et al., 2007b) augmented recorded video images from a laparoscopic camera with visualized volumetric CT data plus an augmented surgical instrument and a virtual

mirror. Such augmentation is useful for optimizing the perception of a complex anatomy structure. To smooth the learning curve of the ultrasound image, Blum, et al. (2009) proposed an AR ultrasound simulator where the ultrasound slice inside the body is visualized using contextual in situ techniques.

1.1.1.3 Education, Training, and Guidance

Magic book, where 3D virtual models appear out of physical book pages (Billinghurst, et al., 2001), is one popular application of AR-powered education. Urban planning is another attractive application area, where an application can be indoor tabletop-based, to visualize multi-layered presentations of 2D drawings, 3D physical models, and digital simulations (Ishii, et al., 2002), or even to allow stakeholders to sketch or modify the scene on site (Sareika and Schmalstieg, 2007).

Other applications can be found in training and guidance. For example, Goto, et al. (2010) presented a behavioral support system that uses existing instructional videos as AR contents. Working toward a future driver assistance system, Tonnis, et al. (2007) developed a system that augments a driver's view with spatial sensor data acquired from laser scanners, monitoring cameras, and detected and classified objects. Another assistance system, in the context of a museum, is developed by Miyashita, et al. (2008) for "artwork appreciation" and "guidance" purposes.

1.1.1.4 Entertainment, Media, and Commerce

ARQuake is a pioneering and famous AR game (Thomas, et al., 2000) that allows a user to play a computer-generated first-person shooter game in the real world. Among various

types of AR games, some exemplary entertaining applications include live soccer, where audiences can watch a soccer match from their own favorite perspectives (Inamoto and Saito, 2003, KOYAMA, et al., 2003), like that of a news broadcaster, meaning a narrator can discuss ‘what if’ scenarios by manipulating several AR markers (Woolard, et al., 2003). story-telling supports real time narratives generation and interaction between a user’s own portrait image and synthetic characters (Cavazza, et al., 2003).

In the consumer e-commerce domain, AR plays an active role in branding promotion and digital printing. A pioneering effort is brought by Lego AR Kiosk (Virtualworldlets, 2009); it offers consumers a 3D preview of the model inside a packing box by tracking the color symbol printed on the box’s top. Another product is Vuforia™ from Qualcomm, which drives brand engagement by recognizing 2D/3D visual targets on a retail shelf to start the AR experience (Qualcomm, 2012). Another famous product Layar™; it allows a publisher to add interactive digital content to static print media (Layar, 2012).

1.1.1.5 Mobile Applications

Alongside a lot of sensors becoming compact and accurate enough to be integrated into handheld devices—like cameras, GPS, and electronic compasses—varieties of AR applications have shown up on smart phones in recent years. Juniper (2011) conducted an extensive analysis of the emerging mobile AR market (Figure 1.2). Wikitude (2012) is a representative Location-Based Service (LBS) application that allows users to discover their surroundings. While LBS applications mainly rely on the GPS and electronic compass sensors, other mobile AR applications, like gaming and social networking, are

typically powered by computer vision tracking techniques. Some examples can be found at ardefender (2012)—a marker-based AR shooting game, as well as TAT (2009)—a social network portal that visualizes the digital identities of people and their social network profiles.

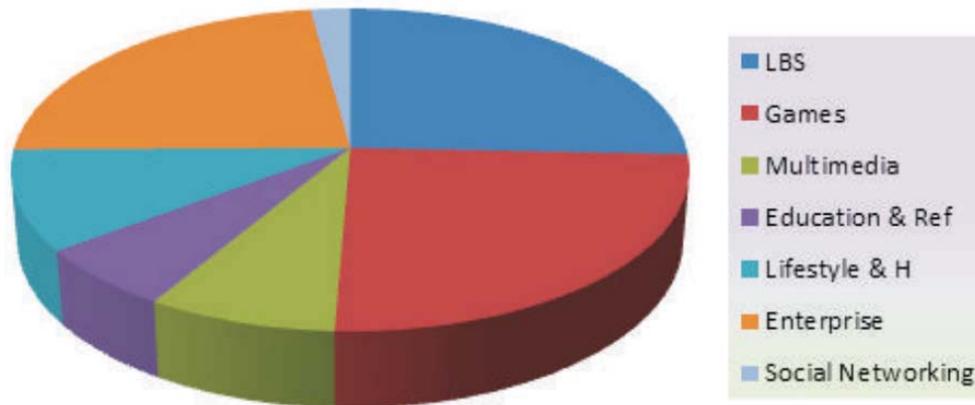
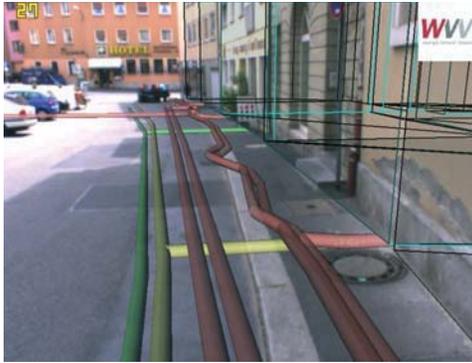


Figure 1.2 Total mobile AR market by 2015, split into 7 categories. (Juniper, 2011)

1.1.2 Augmented Reality in the Construction Domain

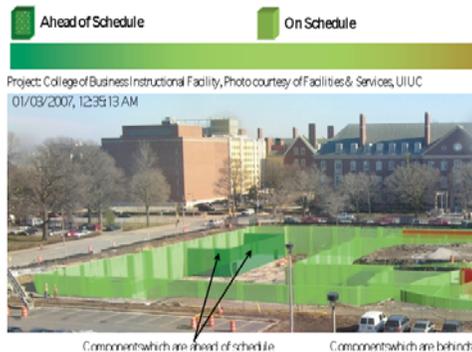
AR has such great potential in the construction industry that Shin and Dunston (2008) presented a comprehensive outline for identifying AR applications in construction. The paper reveals eight work tasks that may potentially benefit from AR (i.e. layout, excavation, positioning, inspection, coordination, supervision, commenting, and strategizing). Based on the classification, some application examples are enumerated below (Figure 1.3).



Excavation



Inspection



Supervision



Strategizing

Figure 1.3 Application examples of AR in construction areas. (Schall, et al., 2010, Georgel, et al., 2007, Golparvar-Fard, et al., 2009, Thomas, et al., 1999)

The first attempt at visualizing underground utilities was made by Roberts, et al. (2002); they looked beneath the ground and inspected the subsurface utilities. Some further exploration can be found in Behzadan and Kamat (2009a) and Schall, et al. (2010), where the work has been extended to improve visual perception and excavation safety.

AR serves as a useful inspection assistance method in the sense that it supplements a user's normal experience with context-related or Georeferenced virtual objects. (Webster, et al., 1996) developed an AR system for improving the inspection and renovation of architectural structures. Users can have x-ray vision and see columns behind a finished wall and re-bars inside the columns. A "discrepancy check" tool has been developed by

Georgel, et al. (2007); it allows users to readily obtain an augmentation in order to find differences between an as-design 3D model and an as-built facility.

Golparvar-Fard, et al. (2009) demonstrated an example of applying AR in the construction supervision. They implemented a system for visualizing performance metrics. It aims to represent progress deviations through the superimposition of 4D as-planned models over time-lapsed real jobsite photographs. Dai, et al. (2011) presented another supervision example of overlaying as-built drawings onto an aboveground site photo for the purpose of continuous quality investigation of a bored pile construction. These works share the characteristic of monitoring the discrepancy in the chronological order, which is different from the ‘discrepancy check’ (Georgel, et al., 2007) mentioned above.

Some examples of coordinating and strategizing are the visualization of construction simulations and architectural designs. Behzadan and Kamat (2007) designed and implemented ARVSCOPE—an AR framework for the visualization of simulated outdoor construction operations to facilitate the verification and validation of the results generated by Discrete Event Simulation (DES). Thomas, et al. (1999)’s work visualized the design of an extension to a building using a mobile AR platform called TINMITH2.

Some other construction tasks excluded by Shin and Dunston (2008), which feature a high complexity of tasks, may also benefit from AR. For example, the quality of welding used to depend on the welders’ experience and skill. Aiteanu, et al. (2003) improved the working conditions for welders, as well as the quality control, by developing a welding

helmet that augments visual information, like paper drawings and online quality assistance, before and during the welding process.

1.1.3 Augmented Reality Research in LIVE

The Laboratory for Interactive Visualization in Engineering (LIVE) has been engaged in several AR applications related to construction operation planning, inspection, safety, and education. These applications are as follows: a visual excavator-collision avoidance system, a rapid reconnaissance for earthquake-induced building damage, and a visualization of operations-level construction processes in both outdoor AR and the collaborative tabletop AR environment (Figure 1.4).



Figure 1.4 AR Research for Construction Applications in LIVE.

The visual collision avoidance system allows excavator operators to persistently “see” what utilities lie buried in the vicinity of a digging machine or a spotter, thus helping prevent accidents caused by utility strikes. With the aid of AR, the rapid post-disaster reconnaissance for building damage system superimposes previously stored building baselines onto the corresponding images of a real structure. The on-site inspectors can thus estimate the damage by evaluating discrepancies between the baselines and the real building edges. Finally, visualization of construction operations in outdoor AR facilitates the verification and validation of the results of simulated construction processes, with minimum efforts spent on model engineering (Behzadan and Kamat, 2009b). Meanwhile, the tabletop collaborative AR version helps to bridge the gap between paper-based static information and computer-based graphical models. It reflects the dynamic nature of a jobsite, and preserves the convenience of face-to-face collaboration.

1.1.4 Challenges for Construction AR Applications

While the aforementioned AR construction applications possess tangible economic and social values, some fundamental technical challenges have to be overcome before AR can be deployed in practical construction applications. The difficulties are associated with two requirements: 1) maintaining a constant spatial alignment between the virtual and real entities, which is also referred to as registration; and 2) creating a sustained illusion that the virtual and real entities co-exist, which is also referred to as occlusion.

The majority of the AR applications in manufacturing, medicine, education, and entertainment favor indoor registration methods, like fiducial marker tracking or feature tracking (Krevelen and Poelman, 2010), with stable registration results and low latency.

Even though some construction applications, like indoor inspection and strategizing, can also benefit from the indoor registration methods, most construction jobsites feature outdoor, unprepared, unconstrained, and dynamically changing environments where the indoor registration methods are inapplicable (Azuma, et al., 1999). It thus increases the difficulties of achieving stable registration effects with low latency and little jittering.

Some of the AR construction research mentioned in Section 1.1.2 puts emphasis on prototyping ideas, but shows little concern for the accuracy requirement when being practiced in a real construction field. In fact, it is a dilemma that one of the major metrics for judging the effectiveness of outdoor AR is whether or not it can provide a faster means of accessing high-accuracy Georeferenced digital information (Shin and Dunston, 2008). In some precision-critical applications, like visual collision-avoidance and building damage assessment, the requirement for position accuracy is at the centimeter or even millimeter level.

Not only can inaccurate positioning cut down engineers' confidence in adopting AR technology, the distracting visual artifacts caused by incorrect occlusion further reduce the credibility of AR graphics. Most current AR construction research adopts the traditional AR rendering approach of always 'superimposing' the virtual content on top of the real scene, regardless of the relative distance between the virtual and real entities to the observer. Therefore, incorrect occlusion manifests itself when the virtual objects should be partially or entirely blocked by the real objects, and further increases the difficulty of correctly interpreting the virtual object's position.

Since most construction AR applications expect to encounter registration and occlusion problems, it is reasonable to solve these challenges and build the solutions into a scalable and extensible AR computing framework that is openly accessible to the community. Researchers who are interested in exploring AR for their specific application in construction or another domain can immediately have access to the core logic components without starting from scratch on developing solutions for the registration and occlusion issues. They would appreciate the existence of such a reusable framework that shortens the lifecycle of developing AR applications. A good example of such a reusable framework is osgART, on which plenty of indoor AR projects have been built (Looser, et al., 2006).

1.2 Research Objectives

Azuma (1997) summarized the properties of a typical AR system: 1) register real and virtual objects in 3D, 2) interact in real time, and 3) real and virtual objects coexist in the augmented space. Each property corresponds to a field of research challenges.

Most of the research challenges are not necessarily new in the AR community, not to mention that they have been carefully studied and that many techniques have been proposed. However, given the difficulty and nature of these problems, all of these research questions remain open. Furthermore, in the context of the proposed work, the challenge lies not only in providing a solution for each question, but also in building a sound scalable and extensible AR computing framework that can be readily adapted to an array of civil engineering applications.

The first and foremost challenge in AR is placing virtual objects in the augmented space with the correct pose, which is called registration. The registration process is difficult because its errors arise from both spatial and temporal domains. Errors in the spatial domain are also referred to as static errors when neither the user nor the virtual objects move (Azuma, 1997). Registration errors become more challenging in the real-time domain, which creates the “swimming” effect.

Even accurate registration does not guarantee an ideal coexistence of virtual and real objects. The result of composing an AR scene without considering the relative depth of the involved real and virtual objects is that the graphical entities in the scene appear to “float” over the real background, rather than blend or co-exist with the real objects in that scene. The occlusion problem is more complicated in outdoor AR, where the user expects to navigate through the space freely, and the relative depth between the involved virtual and real content is changing arbitrarily with time.

The solutions for registration and occlusion were integrated into the SMART computing framework. SMART’s effectiveness was validated in several ongoing AR research projects in LIVE (Section 1.1.3). The specific objectives of this research (Figure 1.5) were thus identified as follows:

- Develop a scalable and extensible AR framework called SMART that features high-accuracy registration and robust occlusion. A rigid and ergonomic carrying harness is built to test the effectiveness of SMART.

- Based on the software and hardware of AR infrastructure, design and build a personnel suite for visualizing underground utilities, which is part of the visual collision avoidance system.
- Design an AR-assisted non-contact method for estimating post-earthquake building damage. This method involves projecting building baselines onto the real site using AR, detecting real building edges, and estimating the position of a building's key locations. Such a reconnaissance system can be deployed on the SMART framework.
- Extend the SMART framework to the tabletop AR environment with indoor registration libraries. Such a tabletop AR system is an interconnecting medium to bridge the gap between computer-based dynamic visualization and paper-based collaboration. It allows multiple users wearing HMDs to observe and interact with dynamic simulated construction activities laid on the surface of a table.

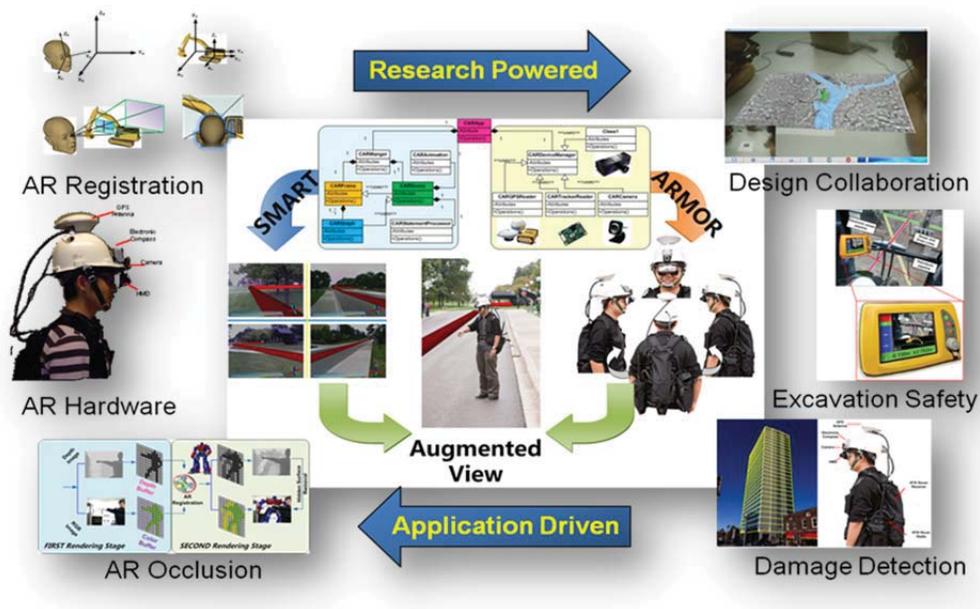


Figure 1.5 Research Objectives of this dissertation.

1.3 Research Methodology

The steps and results enumerated below outline the research methodology:

- Designed a graphics algorithm to place the virtual entities in the augmented scene given the user's geographical position and head orientation.
 - Validated the algorithm's accuracy in the static scenario of simulated construction processes and inspection of underground utilities.
 - Tested stabilization approaches to reduce the jittering response of the orientation tracking data, caused by the electronic compass that is vulnerable to the vibrations associated with walking.
 - Selected high-accuracy but lightweight tracking equipment for building a mobile carrying harness. All of the tracking instruments are placed rigidly on the ergonomic and wearable harness.
- Designed a robust AR occlusion algorithm that uses a real-time Time-of-flight (TOF) camera to resolve the depth of real and virtual objects in real-time.
 - Attempted both homography and stereo image registration approaches to ensure correct alignment and occlusion effect between the TOF depth map and the RGB image.
 - Instrumented the registration algorithms on the GPU for parallel computing using the Render to Texture (RTT) technique.
 - Validated the occlusion algorithm in both the indoor and outdoor experiments of simulated construction processes.

- Studied the convention of marking underground utilities and designed an AR visualization system that is compatible with the existing Michigan MISS DIG system. A spotter can wear the carrying harness and inspect underground utilities.
 - Developed a spotter system for visualizing underground utilities.
 - Developed an anatomy vision for obstructing the view of the ground and rendering the utilities underneath. The augmented view contains attribute and uncertainty information associated with the geometric utilities.
 - Tested the spotter system with the ‘as-designed’ data provided by the DTE Energy Company.

- Developed an AR-assisted non-contact method for estimating the Interstory Drift Ratio (IDR)—the relative drift between consecutive floors divided by the height of the story—which remains the most trustworthy metric for assessing a building’s structural integrity at the story level after an earthquake.
 - Investigated several edge detection algorithms for accurately detecting building edges, and proposed an algorithm for estimating the position of a building’s key locations.
 - Developed a Virtual Prototyping (VP) environment for evaluating the accuracy of the IDR estimation algorithm. Such a VP environment features reconfigurable settings to comprehensively profile the algorithm’s performance.
 - Conducted a sensitivity analysis to evaluate the influence of instrumentation errors on the algorithm in practical use.

- Accommodated the outdoor AR computing framework to the indoor tabletop environment with marker-based tracking libraries. It allows multiple users wearing HMDs to observe and interact with dynamic simulated construction activities laid on the surface of a table.
 - Investigated techniques for rendering multiple videos simultaneously on a single machine so as to synchronize animations across all the users.
 - Implemented a scalable extension interface so that different tracking libraries can be realized as independent plug-ins, and can be flexibly integrated into the system based on the user's choice.

1.4 Dissertation Outline

This dissertation is the result of compiling manuscripts that document the effort involved in realizing SMART and all of its relevant applications. In terms of the content organization, chapters 2 through 5 are stand-alone papers that elaborate details of every phase of the research: review of prior knowledge, research challenges, explored alternatives, proposed methodologies, technical implementation, and validated experiments. The dissertation concludes with chapter 6, which enumerates the contributions and achievements of this research, and posts roadmaps for future work. Since the chapters are written as self-contained papers, some information appears across chapters for the sake of self-completeness.

The manuscripts in this dissertation cover all aspects of SMART and its extended applications. This includes the explanation about the underlying algorithm and technical

realization. A few program snippets and pseudo-code have also been included in the appendices to give the reader insight into the techniques discussed in the manuscripts. However, given the scientific flavor of the dissertation, comprehensive code felt out of place. Fortunately, the open source spirit adopted by this research guarantees the community open access to every single line of code. The open source projects and relevant instructions can be found at <<http://pathfinder.engin.umich.edu/software.htm>>. To map a blueprint for the remainder of the chapters, their titles and brief introductions are below:

- **Chapter 2 – Robust Mobile Computing Framework for Visualization of Simulated Processes in Augmented Reality**

The visualization of engineering processes can be critical for the validation and communication of simulation models to decision-makers. Augmented Reality (AR) visualization blends real-world information with graphical 3D models to create informative composite views that are difficult to replicate on the computer alone. This chapter presents a robust and general-purpose mobile computing framework that allows users to readily create complex AR visual simulations. The technical challenges of building this framework from the software and hardware perspectives are described. SMART is a generic and loosely-coupled software application framework for creating AR visual simulations with accurate registration and projection algorithms. ARMOR is a modular mobile hardware platform designed for user position and orientation tracking, as well as augmented view display. Together, SMART and ARMOR allow the creation of complex AR visual simulations. The framework has been validated in several case

studies, including the visualization of underground infrastructure for applications in excavation planning and control.

- **Chapter 3 – Real-Time Occlusion Handling for Dynamic Augmented Reality Using Geometric Sensing and Graphical Shading**

The primary challenge in generating convincing Augmented Reality (AR) graphics is to project 3D models onto a user's view of the real world, and to create a temporal and spatial sustained illusion that the virtual and real objects co-exist. Regardless of the spatial relationship between the real and virtual objects, traditional AR graphical engines break the illusion of co-existence by displaying the real world merely as a background, and superimposing virtual objects in the foreground. This research proposes a robust depth-sensing and frame buffer algorithm for handling occlusion problems in ubiquitous AR applications. A high-accuracy Time-of-flight (TOF) camera is used to capture the depth map of the real world in real time. The distance information is processed in parallel using the Graphical Shader and Render to Texture (RTT) techniques. The final processing results are written to the graphics frame buffers, allowing accurate depth resolution and hidden surface removal in composite AR scenes. The designed algorithm is validated in several indoor and outdoor experiments using the SMART AR framework.

- **Chapter 4 – Sensitivity Analysis of Augmented Reality-Assisted Building Damage Reconnaissance Using Virtual Prototyping**

The timely and accurate assessment of the damage sustained by a building during catastrophic events, such as earthquakes or blasts, is critical in determining the building's structural safety and suitability for future occupancy. Among many indicators proposed for measuring structural integrity, especially inelastic deformations, Interstory Drift Ratio

(IDR) remains the most trustworthy and robust metric at the story level. In order to calculate IDR, researchers have proposed several nondestructive measurement methods. Most of these methods rely on pre-installed target panels with known geometric shapes or with an emitting light source. Such target panels are difficult to install and maintain over the lifetime of a building. Thus, while such methods are nondestructive, they are not entirely non-contact. This chapter proposes an Augmented Reality (AR) -assisted non-contact method for estimating IDR that does not require any pre-installed physical infrastructure on a building. The method identifies corner locations in a damaged building by detecting the intersections between horizontal building baselines and vertical building edges. The horizontal baselines are superimposed on the real structure using an AR algorithm, and the building edges are detected via a Line Segment Detection (LSD) approach. The proposed method is evaluated using a Virtual Prototyping (VP) environment that allows the testing of the proposed method in a reconfigurable setting. A sensitivity analysis is also conducted to evaluate the effect of instrumentation errors on the method's practical use. The experimental results demonstrate the potential of the new method to facilitate rapid building damage reconnaissance, and highlight the instrument precision requirements necessary for practical field implementation.

- **Chapter 5 – Collaborative Learning of Engineering Processes Using Tabletop Augmented Reality Visual Simulations**

3D computer visualization has emerged as an advanced problem-solving tool for engineering education. For example in civil engineering, the integration of 3D/4D CAD models in the instruction process helps to minimize the misinterpretation of the spatial, temporal, and logical aspects of construction planning information. Yet despite the

advances made in visualization, the lack of collaborative learning makes for outstanding challenges that need to be addressed before 3D visualization can become widely accepted in the classroom. The ability to smoothly and naturally interact in a shared workspace characterizes a collaborative learning process. This chapter introduces tabletop Augmented Reality to accommodate the need to collaboratively visualize computer-generated models. A software program named ARVita, transformed from the SMART framework, is developed to validate this idea, where multiple users wearing Head-Mounted Displays and sitting around a table can all observe and interact with visual simulations of engineering processes. The applications of collaborative visualization using Augmented Reality are reviewed, the technical implementation is covered, and the program's underlying tracking libraries are presented.

- **Chapter 6 – Conclusion**

The chapter summarizes the achievements and contributions of this research. It briefly describes the features of the SMART framework and the scientific relevance of the applications built on top of the SMART framework. It concludes with suggested directions for future research.

1.5 References

[1] Brooks, F. P. (1999). "What's Real About Virtual Reality?" *Journal of Computer Graphics and Applications*, 19(6), 16-27.

[2] Azuma, R., Baillet, Y., Behringer, R., Feiner, S., Julier, S., and McIntyre, B. (2001). "Recent Advances in Augmented Reality." *Journal of Computer Graphics and Applications*, 34-47.

- [3] Azuma, R. (1997). "A Survey of Augmented Reality." *Teleoperators and Virtual Environments*, 355-385.
- [4] Behzadan, A. H., and Kamat, V. R. "Visualization of Construction Graphics in Outdoor Augmented Reality." *Proc., Proceedings of the 2005 Winter Simulation Conference*, Institute of Electrical and Electronics Engineers (IEEE).
- [5] Krevelen, D. W., and Poelman, R. (2010). "A Survey of Augmented Reality Technologies, Applications and Limitations." *The International Journal of Virtual Reality*, 9(2), 1-20.
- [6] Pentenrieder, K., Bade, C., Doil, F., and Meier, P. "Augmented Reality-based factory planning - an application tailored to industrial needs." *Proc., Proceedings of the 2007 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 1-9.
- [7] Bichlmeier, C., Wimmer, F., Heining, S. M., and Navab, N. (2007a). "Contextual Anatomic Mimesis Hybrid In-Situ Visualization Method for Improving Multi-Sensory Depth Perception in Medical Augmented Reality." *Proceedings of the 2007 IEEE and ACM International Symposium on Mixed and Augmented Reality* Nara, Japan, 129-138.
- [8] Billinghamurst, M., Kato, H., and Poupyrev, I. (2001). "The MagicBook—Moving Seamlessly between Reality and Virtuality." *Journal of Computer Graphics and Applications*, 21(3), 6-8.
- [9] Thomas, B., Close, B., Donoghue, J., Squires, J., Bondi, P. D., Morris, M., and Piekarski, W. "ARQuake: An Outdoor/Indoor Augmented Reality First Person Application." *Proc., Proceedings of the 2000 International Symposium on Wearable Computers*, 75-86.
- [10] Klinker, G., Dutoit, A. H., and Bauer, M. "Fata Morgana -A Presentation System for Product Design." *Proc., Proceedings of the 2002 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 76-85.
- [11] Fiorentino, M., Amicis, R. d., Monno, G., and Stork, A. "Spacedesign: A Mixed Reality Workspace for Aesthetic Industrial Design." *Proc., Proceedings of the 2002 International Symposium on Mixed and Augmented Reality*, 86-90.
- [12] Nolle, S., and Klinker, G. "Augmented Reality as a Comparison Tool in Automotive Industry." *Proc., Proceedings of the 2006 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 249-250.
- [13] BMW (2007). "BMW Augmented Reality." <http://www.bmw.com/com/en/owners/service/augmented_reality_introduction_1.html>.

- [14] Navab, N. "Industrial augmented reality (IAR): challenges in design and commercialization of killer apps." *Proc., Proceedings of the 2003 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 2-6.
- [15] Bichlmeier, C., Wimmer, F., Heining, S. M., and Navab, N. "Laparoscopic Virtual Mirror for Understanding Vessel Structure Evaluation Study by Twelve Surgeons." *Proc., Proceedings of the 2007 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 125-128.
- [16] Blum, T., Heining, S. M., Kutter, O., and Navab, N. "Advanced Training Methods using an Augmented Reality Ultrasound Simulator." *Proc., Proceedings of the 2009 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 177-178.
- [17] Ishii, H., Underkoffler, J., Chak, D., and Piper, B. (2002). "Augmented Urban Planning Workbench: Overlaying Drawings, Physical Models and Digital Simulation." *Proceedings of the International Symposium on Mixed and Augmented Reality*.
- [18] Sareika, M., and Schmalstieg, D. "Urban Sketcher: Mixed Reality on Site for Urban Planning and Architecture." *Proc., Proceedings of the 2007 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 27-30.
- [19] Goto, M., Uematsu, Y., Saito, H., Senda, S., and Iketani, A. "Task Support System by Displaying Instructional Video onto AR Workspace." *Proc., Proceedings of the 2010 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 83-90.
- [20] Tonnis, M., Leonhard Walchshausl, R. L., and Klinker, G. "Visualization of Spatial Sensor Data in the Context of Automotive Environment Perception Systems." *Proc., Proceedings of the 2007 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 115-124.
- [21] Miyashita, T., Meier, P., Tachikawa, T., Orlic, S., Eble, T., Scholz, V., and Gapel, A. "An Augmented Reality Museum Guide." *Proc., Proceedings of 2008 IEEE Conference on Computer Vision and Pattern Recognition*, 103-106.
- [22] Inamoto, N., and Saito, H. "Immersive Observation of Virtualized Soccer Match at Real Stadium Model." *Proc., Proceedings of the 2003 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 188-201.
- [23] KOYAMA, T., Kitahara, I., and Ohta, Y. "Live Mixed-Reality 3D Video in Soccer Stadium." *Proc., Proceedings of the 2003 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 178-186.

- [24] Woolard, A., Lalioti, V., Hedley, N., Carrigan, N., Hammond, M., and Julien, J. "Case Studies in Application of Augmented Reality in Future Media Production." *Proc., Proceedings of the 2003 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 294-295.
- [25] Cavazza, M., Martin, O., Charles, F., Marichal, X., and Mead, S. J. "User Interaction in Mixed Reality Interactive Storytelling." *Proc., Proceedings of the 2003 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 304-305.
- [26] Virtualworldlets (2009). "Lego AR Kiosks: Colour Magic Symbol." <<http://www.virtualworldlets.net/Resources/Hosted/Resource.php?Name=LegoARKiosk>>.
- [27] Qualcomm (2012). "Vuforia." <<http://www.qualcomm.com/solutions/augmented-reality>>.
- [28] Layar (2012). "Layar." <<http://www.layar.com/>>.
- [29] Juniper (2011). "Mobile Augmented Reality-Opportunities, Forecasts & Strategic Analysis 2011-2015."
- [30] Wikitude (2012). "Wikitude." <<http://www.wikitude.com/>>.
- [31] ardefender (2012). "ardefender." <<http://www.ardefender.com/>>.
- [32] TAT (2009). "TAT Augmented ID." <<http://www.tat.se/blog/tat-augmented-id/>>.
- [33] Shin, D. H., and Dunston, P. S. (2008). "Identification of Application Areas for Augmented Reality in Industrial Construction Based on Technology Suitability." *Journal of Automation in Construction*, 17(7), 882-894.
- [34] Schall, G., Sebastian, J., and Schmalstieg, D. (2010). "VIDENTE - 3D Visualization of Underground Infrastructure using Handheld Augmented Reality." *GeoHydroinformatics: Integrating GIS and Water Engineering*.
- [35] Georgel, P., Schroeder, P., Benhimane, S., and Hinterstoisser, S. "An Industrial Augmented Reality Solution For Discrepancy Check." *Proc., Proceedings of the 2007 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 111-115.
- [36] Golparvar-Fard, M., Pena-Mora, F., Arboleda, C. A., and Lee, S. (2009). "Visualization of construction progress monitoring with 4D simulation model overlaid on time-lapsed photographs." *Journal of Computing in Civil Engineering*, 23(6), 391-404.

- [37] Thomas, B., Piekarski, W., and Gunther, B. "Using Augmented Reality to Visualise Architecture Designs in an Outdoor Environment." *Proc., Proceedings of the Design Computing on the Net*.
- [38] Roberts, G., Evans, A., Dodson, A. H., Denby, B., Cooper, S., and Hollands, R. "The Use of Augmented Reality, GPS and INS for Subsurface Data Visualization." *Proc., Proceedings of the 2002 FIG XIII International Congress*, 1-12.
- [39] Behzadan, A. H., and Kamat, V. R. "Interactive Augmented Reality Visualization for Improved Damage Prevention and Maintenance of Underground Infrastructure." *Proc., Proceedings of 2009 Construction Research Congress*, American Society of Civil Engineers, 1214-1222.
- [40] Webster, A., Feiner, S., Macintyre, B., and Massie, W. "Augmented Reality in Architectural Construction, Inspection, and Renovation." *Proc., Proceedings of 1996 ASCE Congress on Computing in Civil Engineering*.
- [41] Dai, F., Lu, M., and Kamat, V. R. (2011). "Analytical Approach to Augmenting Site Photos with 3D Graphics of Underground Infrastructure in Construction Engineering Applications." *Journal of Computing in Civil Engineering*, 25(1), 66-74.
- [42] Behzadan, A. H., and Kamat, V. R. (2007). "Georeferenced Registration of Construction Graphics in Mobile Outdoor Augmented Reality." *Journal of Computing in Civil Engineering*, 21(4), 247-258.
- [43] Aiteanu, D., Hiller, B., and Graser, A. "A Step Forward in Manual Welding: Demonstration of Augmented Reality Helmet." *Proc., Proceedings of the 2003 IEEE and ACM International Symposium on Mixed and Augmented Reality*.
- [44] Behzadan, A. H., and Kamat, V. R. (2009b). "Automated Generation of Operations Level Construction Animations in Outdoor Augmented Reality." *Journal of Computing in Civil Engineering*, 23(6), 405-417.
- [45] Azuma, R., Lee, J. W., and Jiang, B. (1999). "Tracking in Unprepared Environments for Augmented Reality Systems." *Computers & Graphics*, 787-793.
- [46] Looser, J., Grasset, R., Seichter, H., and Billingham, M. (2006). "OSGART - A Pragmatic Approach to MR." *5th IEEE and ACM International Symposium on Mixed and Augmented Reality*.

Chapter 2

Robust Mobile Computing Framework for Visualization of Simulated Processes in Augmented Reality

2.1 Introduction

In a broad sense, Augmented Reality (AR) is a multi-sensory technology that blends virtual contents with the real environment. In particular, AR refers to a visualization technology that superimposes virtual objects on the real world. AR has distinct advantages over other forms of visualization in at least three senses: 1) from the perspective of visualization, the real world can significantly mitigate the efforts to create and render contextual models for virtual objects, and can provide a better perception of the surroundings than both pure virtual reality (e.g., visualization of construction simulations) (Behzadan and Kamat, 2007) and the visualization of architectural designs (Thomas, et al., 1999); 2) from the perspective of information retrieval, AR supplements

a user's normal experience with context-related or Georeferenced virtual objects (e.g., looking through walls to see columns (Webster, et al., 1996), and looking beneath the ground to inspect subsurface utilities (Roberts, et al., 2002)); 3) from the perspective of evaluation, authentic virtual models can be deployed to measure the physical condition of real objects (e.g., evaluation of earthquake-induced building damage (Kamat and El-Tawil, 2007), and automation of construction process monitoring (Golparvar-Fard, et al., 2009)).

A typical AR system should possess the following properties, as concluded by Azuma, et al. (2001): 1) real and virtual objects coexist in the augmented space; 2) the system runs in real time; and 3) the system registers real and virtual objects with each other. Each property corresponds to a field of research challenges (e.g., the coexistence of real and virtual objects leads to occlusion and photorealism problems). This chapter focuses primarily on the challenge of achieving precise registration from both the hardware and software perspectives.

2.1.1 Importance of the Research

The fundamental problem in Augmented Reality is the difficulty of placing virtual objects in the augmented space with the correct pose, the process of which is called registration. The registration process is difficult because its errors arise from both the spatial and temporal domains (Azuma, 1997). Furthermore, different tracking technologies have their own error sources. This chapter focuses on the registration problem of AR in an unprepared environment (i.e., outdoors, where sensor-based AR is thus by far the most reliable tracking method free from constraints put on the user).

Errors in the spatial domain are also referred to as static errors when neither the user nor virtual objects move. The static errors of sensor-based AR include: 1) inaccuracy in the sensor measurement; 2) mechanical misalignments between sensors; and 3) an incorrect registration algorithm. The selection of high-accuracy sensors is crucial, because the errors contained in the measurement cannot be eliminated. The accuracy of measurement can be further compromised by the insecure placement of sensors on the AR backpack and helmet. Some early AR backpack design examples can be found in the touring machine (Feiner, et al., 1997) and Tinmith-Endeavour (Piekarski, et al., 2004), and both are fragile and cumbersome. A more robust and ergonomical version is demonstrated by the Tinmith backpack 2006 version (Stafford, et al., 2006), in which a GPS antenna and an InterSense orientation tracker are anchored on top of a helmet. However, the 50cm accuracy of the GPS receiver is not qualified for a centimeter-level-accuracy AR task.

Static errors are relatively easy to eliminate given high accuracy sensors, rigid placement, and a correct registration algorithm. On the other hand, dynamic errors—errors in temporal domain—are much more unpredictable, and create the “swimming” effect. Noticeable dynamic misregistration is mainly caused by the differences in latency between data streams, which is called relative latency by Jacobs, et al. (1997). Relative latency results from: 1) off-host delay: the duration between the occurrence of a physical event and its arrival on the host; 2) synchronization delay: the time in which data is waiting between stages without being processed; and 3) computational delay: the time elapsed for the processing of data in the host system. Two common mitigation methods for resolving relative latency are: 1) adopting multi-threading programming or scheduling

system latency (Jacobs, et al., 1997); and 2) predicting head motion using a Kalman filter (Liang, et al., 1991, Azuma, et al., 1999).

2.1.2 Main Contribution

The mobile computing framework presented in this chapter provides a complete hardware and software solution for centimeter-level-accuracy AR tasks in both the spatial and temporal domains. The robustness of the framework has been validated with an application for visualizing underground infrastructure as part of an ongoing excavation planning and control project.

The Augmented Reality Mobile Operation platform (ARMOR) evolves from the UM-AR-GPS-ROVER hardware platform (Behzadan, et al., 2008). ARMOR improves the design of the UM-AR-GPS-ROVER in two senses: rigidity and ergonomics. It introduces high-accuracy and lightweight devices, rigidly places all tracking instruments with full calibration, and renovates the carrying harness to make it more wearable.

The Scalable and Modular Augmented Reality Template (SMART) builds on top of the ARVISCOPE software platform (Behzadan and Kamat, 2009). The main motivation of ARVISCOPE is exporting some basic modules communicating with peripheral hardware as dynamic link libraries that can later be imported into other potential AR applications. SMART takes advantage of these modules, and constructs an AR application framework that separates the AR logic from the application-specific logic. This extension essentially creates a standard structured AR development environment.

The in-built registration algorithm of SMART guarantees high-accuracy static alignment between real and virtual objects. Some efforts have also been made to reduce dynamical misregistration, including: 1) in order to reduce synchronization latency, multiple threads are dynamically generated for reading and processing sensor measurement immediately upon the data arrival in the host system; and 2) the Finite Impulse Response (FIR) filter applied on to jittering output of the electronic compass leads to filter-induced latency, therefore an adaptive lag compensation algorithm is designed to eliminate the dynamic misregistration.

2.2 ARMOR Hardware Architecture

As a prototype design, the UM-AR-GPS-ROVER succeeded in reusability and modularity, and produced sufficient results in proof-of-concept simulation animation. However, there are two primary design defects that are inadequately addressed: accuracy and ergonomics. First of all, the insecure placement of tracking devices disqualifies the UM-AR-GPS-ROVER from the centimeter-accuracy-level goal. Secondly, packaging all devices, power panels, and wires into a single backpack makes it impossible to accommodate more equipment like the Real Time Kinematic (RTK) rover radio; the weight of the backpack is also too heavy for even distribution around the body.

ARMOR is a significant upgrade over the UM-AR-GPS-ROVER. The improvements can be broken into four categories: 1) highly accurate tracking devices with rigid placement and full calibration; 2) lightweight selection of input/output and computing devices and external power source; 3) intuitive user command input, and 4) load-bearing

vest to accommodate devices and distribute weight evenly around the body. An overview comparison between ARVISCOPE and ARMOR is listed in Table 2.1.

Table 2.1 - Comparison between the UM-AR-GPS-ROVER and ARMOR configuration.

Name	Meaning	Operation	Expression	Range
Device	UM-AR-GPS-ROVER	ARMOR	Comparison	Device
Location Tracking	Trimble AgGPS 332 using OmniStar XP correction for Differential GPS method	Trimble AgGPS 332 using CMR correction broadcast by a Trimble AgGPS RTK Base 450/900	OmniStar XP provides 10~20cm accuracy. RTK provides 2.5 cm horizontal accuracy, and 3.7cm vertical accuracy	Location Tracking
Orientation Tracking	PNI TCM 5	PNI TCM XB	Same accuracy, but ARMOR places TCM XB rigidly close to camera	Orientation Tracking
Video Camera	Fire-I Digital Firewire Camera	Microsoft LifeCam VX-5000	LifeCam VX-5000 is lightweight, small volume, and less wire	Video Camera
Head-mounted Display	i-Glasses SVGA Pro video see-through HMD	eMagin Z800 3DVisor	Z800 3DVisor is lightweight with stereovision	Head-mounted Display
Laptop	Dell Precision M60 Notebook	ASUS N10J Netbook	ASUS N10J is lightweight, small volume, and equipped with NVIDIA GPU	Laptop
User Command Input	WristPC wearable keyboard and Cirque Smart Cat touchpad	Nintendo Wii Remote	Wii Remote is lightweight and intuitive to use	User Command Input
Power Source	Fedco POWERBASE	Tekkeon myPower MP3750	MP3750 is lightweight and has multiple voltage output charging both GPS receiver and HMD	Power Source
Backpack Apparatus	Kensington Contour Laptop Backpack	Load Bearing Vest	Extensible and easy to access equipment	Backpack Apparatus

2.2.1 Tracking Devices

2.2.1.1 Orientation Tracking Device—Electronic Compass

The TCM XB electronic compass is employed to measure the yaw, pitch, and roll that describe the relative attitude between the eye coordinate system and the world coordinate system. It measures the heading up to a full 360-degree range, and maintains the accuracy of 0.3°rms when the tilt (pitch and roll) is no larger than 65°, the common motion range of the human head.

Theoretically the electronic compass is applied to measure the orientation of the user's head. However, seeing as a user's eyes are obstructed by a Video See-Through Head Mounted Display (HMD) and that the eyes' function is replaced by a video camera, the electronic compass—instead of the eyes—is applied to measure the attitude of the camera.

The UM-AR-GPS-ROVER places the electronic compass on top of the helmet, and thus induces more physical attitude disagreement between the camera and the electronic compass. ARMOR chooses to anchor the electronic compass rigidly close to the camera on the brim of the helmet, and parallel to the line of sight, making the physical discrepancy calibration much easier. The calibration approach is described in Section 2.4.2.1. For safety reasons, the electronic compass is encapsulated in a custom-sized aluminum enclosure that is free of magnetic forces.

Magnetometer calibration also needs to be carried out for the purpose of compensating for distortions to the magnetic field caused by the host system (PNI, 2009).

Given that ARMOR's entire periphery could have magnetic impact on the sensor—for example GPS receivers, HMDs, and web cameras—the TCM XB needs to be mounted within the host system that is moved as a single unit during the calibration. The calibration should also be conducted away from strong magnetic sources that will not appear during the real application, for example rebar in the concrete floor under the carpet.

2.2.1.2 Position Tracking Device — RTK-GPS

The UM-AR-GPS-ROVER selects the AgGPS 332 Receiver with OmniStar XP mode to provide a user's position (i.e., the position of camera center) within 10~20cm accuracy. This level of accuracy is sufficient for creating animations of a construction process simulation, where slight positional displacement may not necessarily compromise the validation purpose. However, for precision critical applications, 10~20cm accuracy will definitely compromise the final result.

The AgGPS 332 Receiver used in ARVISCOPE is upgraded and three principles are followed: 1) The upgraded GPS must be able to produce a centimeter-level output; 2) The hardware upgrade should have minimum impact on the software; and 3) The existing device should be fully utilized given the cost of high-accuracy GPS equipment. Ultimately, the AgGPS RTK Base 450/900 GPS Receiver is chosen for implementing the upgrade for three reasons. First, it utilizes RTK technology to provide 2.5cm horizontal accuracy and 3.7cm vertical accuracy on a continuous real-time basis. The RTK Base 450/900 Receiver is set up as a base station placed at a known point (i.e., control points set up by the government with 1st order accuracy), and tracks the same satellites as an

RTK rover. The carrier phase measurement is used to calculate the real-time differential correction that is sent as a Compact Measurement Record (CMR) through a radio link to the RTK rover within 100km (depending on the radio amplifier and terrain) (Trimble, 2007). The RTK rover applies the correction to the position it receives and generates centimeter-level accuracy output. The second reason this receiver is chosen is that, despite the upgrade, the RTK rover outputs the position data in NMEA format (acronym for National Marine Engineers Association) (NMEA, 2010), which is also used in OmniStar XP. No change therefore applies to the software part. The third and final reason is that the original AgGPS 332 Receiver is retained as an RTK rover with its differential GPS mode being geared from OmniStar XP to RTK. A SiteNet 900 radio works with the AgGPS 332 Receiver to receive the CMR from the base station.

Improvement has also been made to the receiver antenna placement. The UM-AR-GPS-ROVER was mounted to the receiver antenna on a segment of pipe that was tied to the interior of the backpack. This method proved to be inefficient in preventing lateral movement. ARMOR anchors the GPS receiver with a bolt on the summit of the helmet, so that the phase center of the receiver will never shift relative to the camera center. The relative distance between the receiver phase center and the camera center is measured and added as a compensated value to the RTK rover measurement.

ARMOR can work in either Indoor or outdoor mode. Indoor mode does not necessarily imply that the GPS signal is unavailable, but that the qualified GPS signal is absent. The GPS signal quality can be extracted from the \$GGA section of the NMEA string. The fix quality ranges from 0–8, for example 2 means DGPS fix, 4 means Real

Time Kinematic, and 5 means float RTK. The user can define the standard (i.e., which fix quality is deemed as qualified) in the hardware configuration file. When a qualified GPS signal is available, the geographical location is extracted from the \$GPGGA section. Otherwise, a preset pseudo-location is used, and this pseudo-location can be controlled by a keyboard.

2.2.2 Input/Output Devices and External Power Supply

2.2.2.1 Video Sequence Input: Camera

The camera is responsible for capturing the continuous real-time background image. The ideal device should possess properties of high resolution, high-frequency sampling rate, and high-speed connection, with a small volume and light weight. Microsoft LifeCam VX5000 stands out from the mainstream off-the-shelf web cameras for the following reasons—the size is 45cm x 45.6cm and only requires USB2.0 for both data transmission and power supply, and it doesn't compromise on resolution (640 x 480) or connection speed (480Mbps). More importantly, it takes samples at 30Hz, which is the same speed as the electronic compass.

2.2.2.2 Augmented View Output: Head-mounted Display (HMD)

The augmented view generated by the video compositor is eventually presented by the Video See-Through HMD. The eMagin Z800 3DVisor is chosen as the HMD component of ARMOR because it has remarkable performance in all primary factors, including view angle, number of colors, weight, and comfort. Furthermore, stereovision is one of the most important rendering effects valued by domain experts, because it helps the user to

better appreciate the 3D augmented space. Unlike i-Glasses SVGA Pro used by the UM-AR-GPS-ROVER, the Z800 3DVisior provides stereovision when working with an NVIDIA graphics card, which supports two perspectives in frame sequential order (3DVisior, 2010).

2.2.2.3 External Power Supply

External power supplies with variant voltage output are indispensable for powering all devices without integrated internal batteries. Tekkeon myPower ALL MP3750 shows improvements over POWERBASE, which is used by the UM-AR-GPS-ROVER, in four ways: 1) both the volume (17cm x 8cm x 2cm) and weight (0.44kg) of MP3750 are only 1/5 of POWERBASE's volume and weight; 2) the main output voltage varies from 10V to 19V for powering the AgGPS 332 Receiver (12V), and an extra USB output port can charge the HMD (5V) simultaneously (Table 2.2); 3) it features automatic voltage detection with an option for manual voltage selection; and 4) an extended battery pack can be added to double the battery capacity (Tekkeon, 2009).

Table 2.2 – Power voltage demands of different devices.

Component	External power supply	Voltage	Power
Head-mounted Display	MP3750	5V	<1.25W
Electronic Compass	AAA batteries	3.6V~5V	0.1W
Web Camera	Through USB	Unknown	Unknown
RTK Rover Receiver	MP3750	10V~32V	4.2W
RTK Rover Radio	Through RTK Rover Receiver	10.5V~20V	3W
RTK Base Receiver	Integrated Internal battery	7.4V	<8W
User Command Input	AA batteries	3V	Unknown
Laptop	Integrated Internal battery	11.1V	17.76W

2.2.2.4 User Command Input: Nintendo Wii Remote

A domain-related augmented system should be capable of obtaining users' instructions through an intuitive interaction method. For example, the user may want to use the mouse to select objects in the augmented space, to query, to edit, and to update their attribute or spatial information. The Nintendo Wii Remote (Wiimote) has proved its effective user experience not only on the Wii Console but also on PC games because of its Bluetooth connection feature. ARMOR takes advantage of the Wiimote's motion sensing capability that allows the user to interact and manipulate objects on screen via gesture recognition and pointing through the use of an accelerometer. A Programmable Input Emulator GlovePIE is also deployed to map commands or motion of the Wiimote to PC keyboard and mouse events.

2.2.2.5 Load-Bearing Vest

The optimization of all devices in aspects of volume, weight, and rigidity allows the author to compact all components into one load-bearing vest. Figure 2.1 shows the configuration of the backpack and the allocation of hardware. There are three primary pouches: the back pouch accommodates the AgGPS 332 Receiver; the SiteNet 900 is stored in the right side pouch, and the left-side pouch holds the HMD connect interface box to a PC and the MP3750 battery; and the ASUS N10J netbook is securely tied to the inner part of the back. All of the other miscellaneous accessories—like USB to Serial Port hubs, or AAA batteries—are distributed in the auxiliary pouches. The wire lengths are customized to the vest, which minimizes outside exposure.

The configuration of the vest has several advantages over the Kensington Contour Laptop Backpack used by ARVSCOPE: 1) the design of the pouches allows for an even distribution of weight around the body; 2) the separation of devices allows the user to conveniently access and check the condition of certain hardware; and 3) different parts of the loading vest are loosely joined so that the vest can fit any body type, and be worn rapidly even when fully loaded. ARMOR has been tested by several users for outdoor operation that lasts for over 30 continuous minutes, without any interruption or reported discomfort.



Figure 2.1 The profile of ARMOR from different perspectives.

2.3 SMART Software Framework

SMART provides a default application framework for AR tasks, where most of its components are written as generic libraries and can be inherited in specific applications. The framework isolates the domain logic from AR logic, so that the domain developer

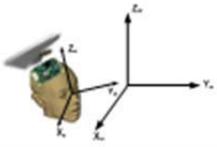
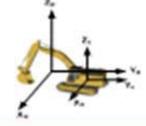
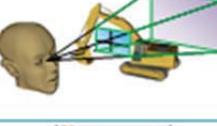
only needs to focus on realizing application-specific functionalities and leaving the AR logic to the SMART framework.

The SMART framework follows the classical model-view-controller (MVC) pattern. Scene-Graph-Controller is the implementation of the MVC pattern in SMART: (1) the model counterpart in SMART is the scene that utilizes application-specific I/O engines to load virtual objects, and that maintains their spatial and attribute status. The update of a virtual object's status is reflected when it is time to refresh the associated graphs; (2) the graph corresponds to the view and reflects the AR registration results for each frame update event. Given the fact that the user's head can be in continuous motion, the graph always invokes callbacks to rebuild the transformation matrix based on the latest position and attitude measurement, and refreshes the background image; (3) the controller—manages all of the UI elements, and responding to a user's commands by invoking delegates' member functions like a scene or a graph.

The framework of SMART that is based on a Scene-Graph-Controller set-up is constructed in the following way (Figure 2.2). The main entry of the program is CARApp, which is in charge of CARSensorForeman and CARSiteForeman. The former initializes and manages all of the tracking devices, like RTK rovers and electronic compasses. The latter one defines the relation among scene, graphs, and controller. After a CARSiteForeman object is initialized, it orchestrates the creation of CARScene, CARController, and CARGraph, and the connection of graphs to the appropriate scene. Applications derived from SMART are Single Document Interface (SDI), therefore there

basic UI elements from CARController, but also adds customized elements for controlling animation like play, pause, continue, and jump.

Table 2.3 - The four steps of registration process.

Step	Task	Illustration	Parameters and Device
Viewing	Position the viewing volume of a user's eyes in the world		Attitude of the camera (Electronic Compass)
Modeling	Position the objects in the world		Location of the world origin (RTK GPS)
Creating Viewing Frustum	Decide the shape of the viewing volume		Lens and aspect ratio of camera (Camera)
Projection	Project the objects onto the image plane		Perspective Projection Matrix

2.4 Static Registration

2.4.1 Registration Process

The registration process of Augmented Reality is very similar to the computer graphics transformation process: 1) positioning the viewing volume of a user's eyes in the world coordinate system; 2) positioning objects in the world coordinate system; 3) determining the shape of the viewing volume; and 4) converting objects from the world coordinate system to the eye coordinate system (Shreiner, et al., 2006). However, unlike computer graphics where parameters needed for step 1 through 3 are coded or manipulated by the

user, Augmented Reality rigidly fulfills these steps according to the 6 degrees of freedom measured by tracking devices and the lens parameter of the real camera. Table 2.3 lists the registration process, the needed parameters, and their measuring devices.

2.4.1.1 Step 1 – Viewing

The origin of the world coordinate system coincides with that of the eye coordinate system, which is the user’s geographical location (Figure 2.3). The world coordinate system uses a right-handed system with the Y-axis pointing in the direction of the true north, the X-axis pointing to the east, and the Z-axis pointing upward. The eye coordinate system complies with the OpenSceneGraph (OSG) (Martz, 2007) default coordinate system, using a right-handed system with the Z-axis as the up vector, and the Y-axis departing from the eye.

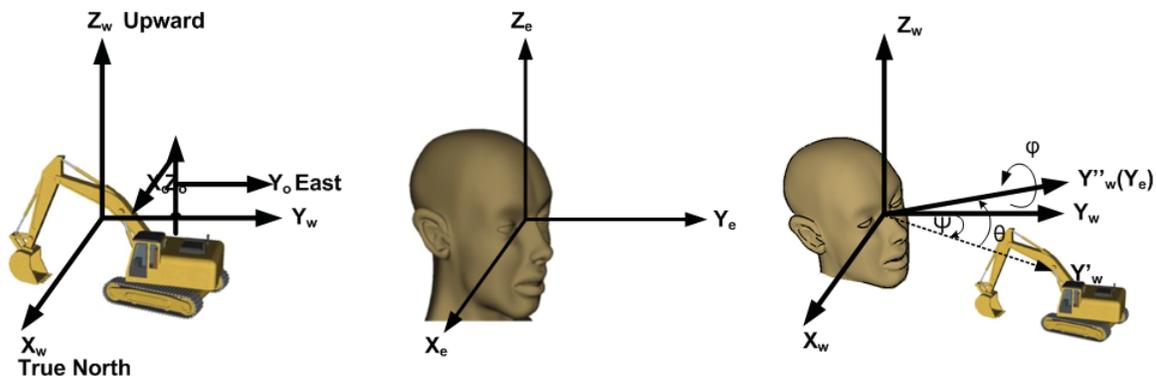


Figure 2.3 Definition of the world coordinate system.

Yaw, pitch, and roll—all measured by the magnetic sensor—are used to describe the relative orientation between the world and eye coordinate systems (Figure 2.4). There are six possibilities of rotating sequences (i.e., xyz, xzy, zxy, zyx, yzx, and yxz), and zxy is picked to construct the transformation matrix between the two coordinate systems. Suppose the eye and world coordinate systems coincide at the beginning; the user’s head

rotates around the Z-axis by yaw angle $\Psi \in (-180, 180]$ to get the new axes, X' and Y' . Since the rotation is clockwise under the right-handed system, the rotation matrix is $R_z(-\Psi)$. Secondly, the head rotates around the X' -axis by pitch angle $\theta \in [-90, +90]$ to get the new axes, Y'' and Z' , with counter-clockwise rotation of $R_{x'}(\theta)$. Finally, the head rotates around the Y'' -axis by roll angle $\phi \in (-180, 180]$ with a counter-clockwise rotation of $R_{y''}(\phi)$ to reach the final attitude.

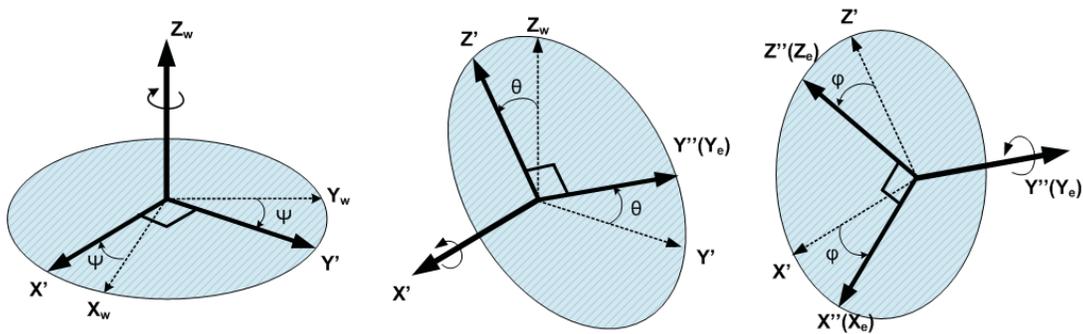


Figure 2.4 The relative orientation between the world and eye coordinate systems is described by yaw, pitch, and roll.

Converting the virtual object from the world coordinate to the eye coordinate is an inverse process of rotating from the world coordinate system to the eye coordinate system, therefore the rotating matrix is written as: $R_z(\Psi) R_{x'}(-\theta) R_{y''}(-\phi)$ or $R_z(\text{yaw}) R_{x'}(-\text{pitch}) R_{y''}(-\text{roll})$ (Equation 2-2). Since OSG provides quaternion, a simple and robust way to express rotation, the rotation matrix is further constructed as quaternion by specifying the rotation axis and angles. The procedure is explained as follows, and its associated equations are listed in sequence from Equation 2-3 to 2-5: rotating around the Y'' -axis by $-\phi$ degree, then rotating around the X' -axis by $-\theta$ degree, and finally rotating around the Z-axis by Ψ degree.

$$\begin{bmatrix} X_e \\ Y_e \\ Z_e \end{bmatrix} = \begin{bmatrix} \cos\Psi & \sin(-\Psi) & 0 \\ \sin\Psi & \cos\Psi & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & \sin(-\theta) & \cos\theta \end{bmatrix} \quad 2-1$$

$$* \begin{bmatrix} \cos\varphi & 0 & \sin(-\varphi) \\ 0 & 1 & 0 \\ \sin\varphi & 0 & \cos\varphi \end{bmatrix} * \begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix}$$

$$P_e = R_z(\Psi) * R_x'(-\theta) * R_y''(-\varphi) * P_w \quad 2-2$$

$$Z - axis = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad 2-3$$

$$X' - axis = \begin{bmatrix} \cos\Psi & \sin\Psi & 0 \\ \sin(-\Psi) & \cos\Psi & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \cos\Psi \\ -\sin\Psi \\ 0 \end{bmatrix} \quad 2-4$$

$$Y'' - axis = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin(-\theta) \\ 0 & \sin\theta & \cos(\theta) \end{bmatrix} * \begin{bmatrix} \cos\Psi & \sin\Psi & 0 \\ \sin(-\Psi) & \cos\Psi & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad 2-5$$

$$= \begin{bmatrix} \sin(\Psi) \\ \cos\theta\cos\Psi \\ \sin\theta\cos\Psi \end{bmatrix}$$

2.4.1.2 Step 2 – Modeling

The definition of the object coordinate system is determined by the drawing software. The origin is fixed to a pivot point on the object with user-specified geographical location. The geographical location of the world coordinate origin is also given by the GPS measurement; the 3D vector between the object and world coordinate origins can thus be calculated. The methods to calculate the distance between geographical coordinates is originally introduced by Vincenty (1975), and Behzadan and Kamat (2007)

proposed a reference point concept to calculate the 3D vector between two geographical locations. SMART adopts the same algorithm to place a virtual object in the world coordinate system using the calculated 3D vector. After that, any further translation, rotation, and scaling operations that are needed are applied on the object.

2.4.1.3 Step 3 and 4 – Viewing Frustum and Projection

The real world is perceived through the perspective projection by the human eye and the web camera. Four parameters are needed to construct a perspective projection matrix: horizontal angle of view, horizontal and vertical aspect ratio, and NEAR plane and FAR plane. All of them together form a viewing frustum and decide the amount of virtual content shown in the augmented space (Figure 2.5). Virtual objects outside of the viewing frustum are either cropped or clipped.

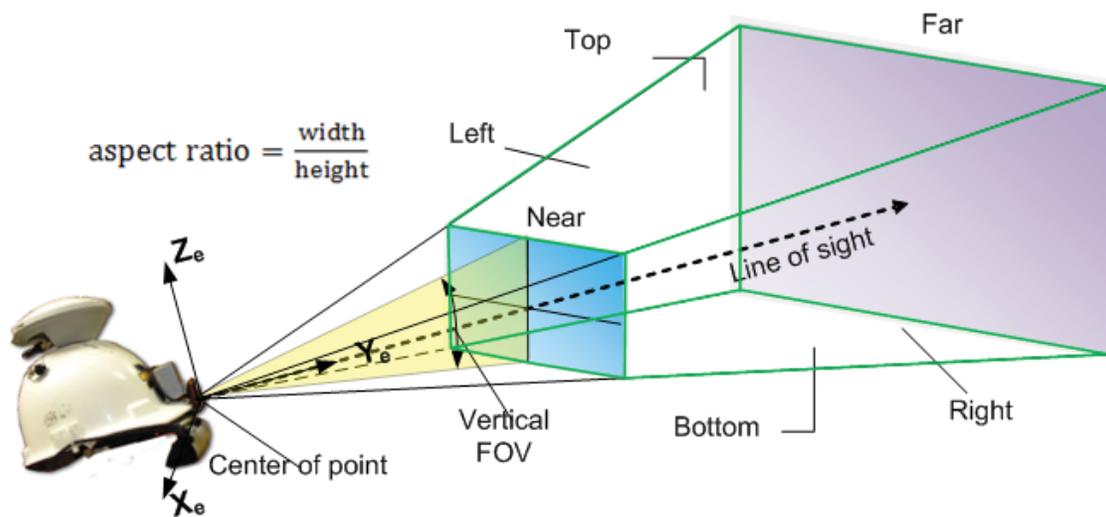


Figure 2.5 The viewing frustum defines the virtual content that can be seen.

The NEAR and FAR planes do not affect how the virtual object appears on the projection plane. However, to maintain a high precision z-buffer, the principle is to keep the NEAR plane as far as possible, and the FAR plane as close as possible. The horizontal and vertical angle of view directly influence the magnification of the projected image and are affected by the focal length and aspect ratio of the camera. In order to ensure consistent perspective projection between the real and virtual camera, both of them need to share the same angle of view.

2.4.2 Registration Validation Experiment

2.4.2.1 Calibration of the Mechanical Attitude Discrepancy

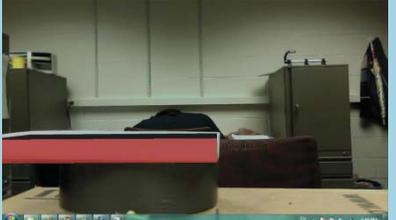
The mechanical attitude discrepancy between the real camera and the sensor needs to be compensated by the following calibration procedure. A real box of size 12cm x 7cm x 2cm (length x width x height) is placed at a known pose. A semi-transparent 3D model of the same size is created and projected onto the real scene, so that the level of alignment can be judged. The virtual box is first projected without adjustments to the attitude measurement, and discrepancy is thus present. The virtual box is then shifted to align with the real one by adding a compensation value to the attitude measurement, as shown in Table 2.4 Row 1.

2.4.2.2 Validation of the Static Registration Algorithm

A series of experiments are performed to validate the agreement between the real and virtual camera; if the static registration algorithm works correctly, the virtual box should coincide with the real box when moved together with 6 degrees of freedom. Overall, the

virtual box really matches the real one in all tested cases, and a selected set of experiments are shown below in Table 2.4 Rows 2~3.

Table 2.4 - Mechanical attitude calibration result and validation experiment of registration algorithm.

<p>Calibration Result Yaw offset: -4.5° Pitch offset: -7.3° Roll offset: -1.0°</p>		
<p>X pos: -0.15m Y pos: 0.30m Z pos: -0.04m</p>		
<p>X pos: -0.05m Y pos: 0.30m Z pos: -0.09m Roll: -22.21°</p>		
<p>X pos: -0.07m Y pos: 0.30m Z pos: -0.09m Pitch: 46.12°</p>		

2.5 Resolving the Latency Problem in the Electronic Compass

Due to the latency induced by the compass module, itself, correct static registration does not guarantee that the user can see the same correct and stable augmented image when in

motion. This section addresses the cause and solution for the dynamic misregistration problem.

2.5.1 Multi-threading to Reduce Synchronization Latency

There are two options for communicating with the compass module: PULL and PUSH mode. PULL is a passive output mode for the compass module, and is used by the UM-AR-GPS-ROVER to pull data out of the module. Since the UM-AR-GPS-ROVER does not separate I/O communication from the electronic compass as a background task, the main function has to be suspended when the program requests orientation data from the module. One loop of the pulling request is 70ms on average, and significantly slows down program performance. Thus the maximum frames per second (fps) for ARVISCOPE is 15, causing noticeable discontinuity.

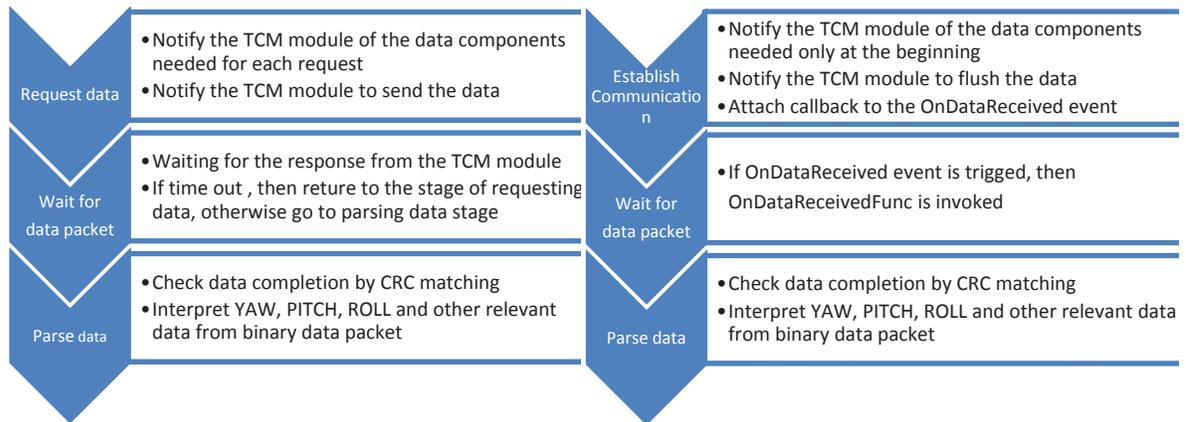


Figure 2.6 Communication stages in the PULL Mode (left) and the PUSH Mode (right).

The PUSH mode is an active output mode for the compass module. SMART selects the PUSH mode as its data communication method to increase the program efficiency. If the PUSH mode is selected, the module outputs the data at a fixed rate set by the host

system (Figure 2.6). If the fixed rate is set to 0, which is done by SMART, the module will flush the next data packet as soon as the previous one is sent out. The sampling and flushing happens at proximately 30 to 32 Hz. The biggest advantage of choosing the PUSH mode is that, once the initial communication is successfully established, and no FIR filtering is carried on in the hardware, the host system can acquire the observed orientation data in only 5ms on average.

However, disadvantages of choosing the PUSH mode also exist. Since the data packet arrives at faster than 30Hz, if the software is not capable of handling the data queue at the same rate, it will cause the rapid accumulation of the data packet in the buffer. Not only will this introduce latency to the view updating, but will also overflow the buffer and eventually crash the program. Therefore, SMART adopts an event-based asynchronous pattern to manage high frequency data packet arrival. When SMART detects that a character is received and placed in the buffer, a DataReceived event is triggered, and the data parsing function that was registered with this event beforehand is invoked and proceeds on a separate thread in the background without interrupting the main loop. This multi-threaded processing accelerates the main function, rendering a speed up to 60 fps, and also reduces the synchronization latency to the minimum possible value.

2.5.2 Experiment to Determine the Latency under Push Mode

A similar experiment has been done by Liang, et al. (1991) using a video camera to track the periodic motion of a pendulum to which the Polhemus Isotrack magnetic sensor was attached. The author's experiment was performed on a DELL INSPIRON machine with

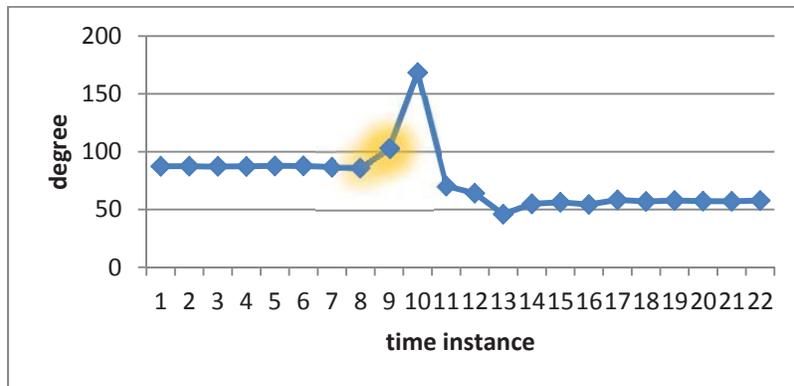
an Intel Core(TM) 2 Duo CPU T6600 2.2GHz and 64bit Windows Operating System. In order to minimize the transmission latency between the camera and the host system, an integrated camera was used and the resolution was adjusted to the minimum option of 160 x 120. Both the camera and the TCM module ran at approximately 30Hz. The camera update function was written as a callback and executed at every frame. The system time was recorded when each new frame is captured. The TCM module update function was written as a delegate and registered with the OnDataReceived event when a new data packet was placed in the buffer. The system time stamp was also assigned to the angular data each time the event is triggered.



(a) Static

(b) Begin to swing

(c) Second frame of swing



(d) Recorded data log

Figure 2.7 Comparison between the TCM-XB data log and the corresponding recorded image frames. The shaded area highlights the exact instant that the module started swinging.

The TCM module was held static at the beginning, and then rapidly swung to one side at the speed of about $150^\circ / \text{sec}$. Later, the exact instant that the module started swinging was identified from the recorded image frames and the TCM module angular data, along with their corresponding time stamps (Figure 2.7). In this way, the time stamps were compared to find the lag of the TCM module PUSH mode. Six groups of experiments were carried out and the delay was the PUSH mode relative to the web camera is 5 ms on average. This implies that the communication delay in the PUSH mode was small enough to be neglected. Another set of experiments was carried out on the SMART system where the external MS LifeCam VX5000 web camera was used instead of the machine-integrated camera. Because of the data transmission delay between the external camera and the computer through USB 2.0, the external camera fell behind the TCM module by 40ms on average.

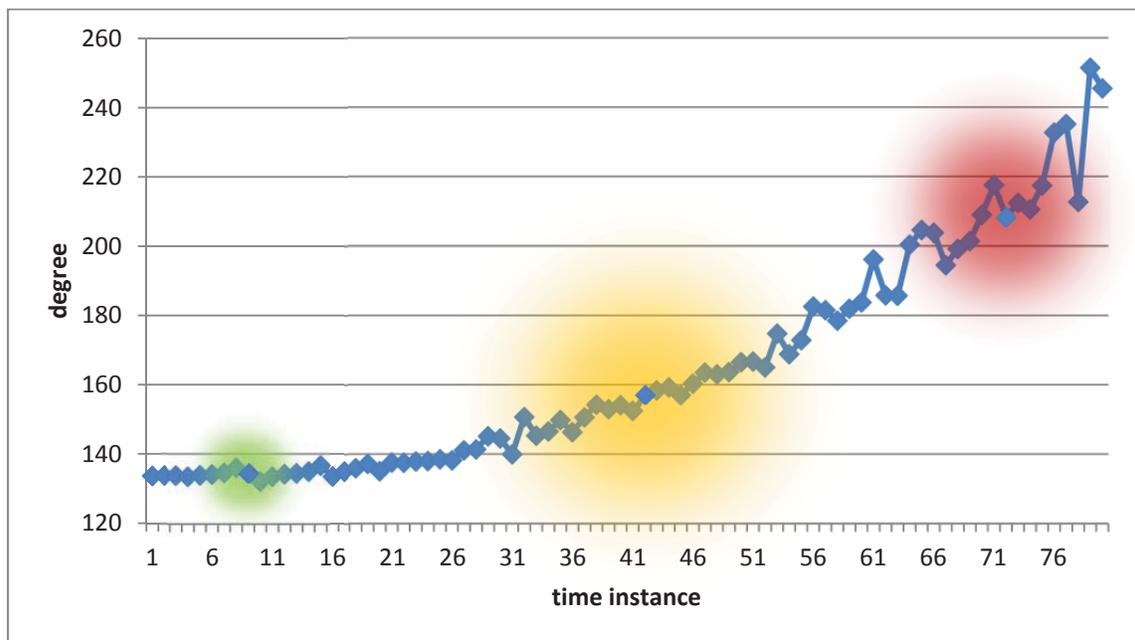


Figure 2.8 The data shows that the noise in the raw data increases as the motion accelerates.

2.5.3 Filter-induced Latency

Even though the PUSH mode is free of synchronization delay, there is still significant latency if the FIR filter is switched on inside of the compass module. This section explains the reason for this phenomenon. The calibration of the magnetometer can compensate for a local static magnetic source within the vicinity of the compass module. However, dynamic magnetic distortion still impacts the module in motion, and the noise magnification depends on the acceleration of the module. Usually the faster the acceleration is, the higher the noise is (Figure 2.8). Among the three degrees of freedom, heading is the most sensitive to the noise.

Except the high frequency vibration noise, other types of noise can be removed by a FIR Gaussian filter. The compass module comes with 5 options for filtering: 32, 16, 8, 4, and 0 tap filter. The higher the number is, the more stable the output, but the longer the expected latency. Consider the case of selecting a 32 tap filter (Figure 2.9). When it is time to send out estimated data at time instant A, the module adds a new sample A to the end of the queue with the first one being dropped, and applies a Gaussian filter to the queue. However, the filtered result actually reflects the estimated value at time instant (A-15). Since the module samples at approximately 30–32 Hz, it induces a 0.5 second delay for a 32 tap filter; a 0.25 second delay for 16 tap filter, and so on. This is called filter-induced latency, and it applies to both PULL and PUSH mode. A 0 tap filter implies no filtering, but significant jittering.

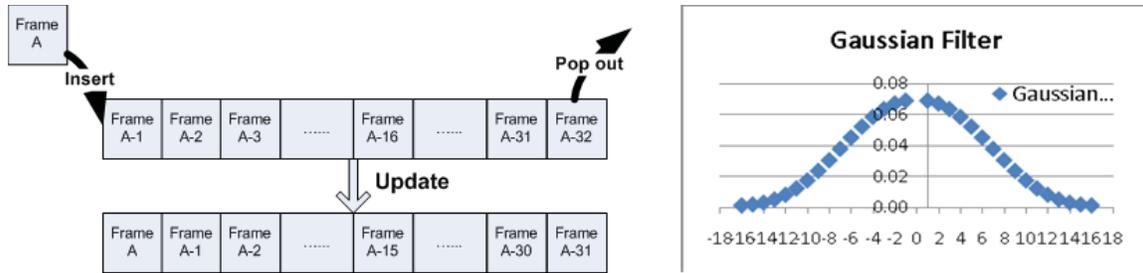


Figure 2.9 The Filter-induced latency when a 32 tap Gaussian filter is used.

2.5.4 Half-Window Gaussian Filter

In order to avoid the filter-induced latency, the Gaussian FIR filter is moved from the hardware to the software, but with only half window size applied. For example, if a complete Gaussian window is used, it is not until time instant $A+15$ that the estimated value can be available for time instant A . However, half window replicates the past data from time instant $A-15$ to time instant A as the future data from time instant $A+1$ to $A+16$, and generates an estimated value for time instant A (Figure 2.10). Nevertheless, as is shown in the graph chart, half window still causes 4-5 frames of latency on average. Depending on the speed of module movement, the faster the speed, the longer latency it presents. We address this kind of latency as half window induced latency.

Because the half window Gaussian filter puts more emphasis on the current frame, it makes the estimated result more sensitive to noise contained in the current frame, and consequently there is more jittering than in the estimated result of the complete window Gaussian filter. Therefore, a second half window Gaussian is applied on the first filtered result for smoothing purposes, but this introduces 1-2 extra frames of latency (Figure 2.11). However, this additional latency can be discounted because it does not exceed the original latency—the one between the half window Gaussian filter and the complete

window Gaussian filter. Therefore, double the additional latency is subtracted from the Twice Gaussian filter result, and this makes the estimation closer to the actual data than the half window Gaussian filter result. Unfortunately, this approach fails during the transition state, and leads to overshooting during change of direction, and during transitions from dynamic to static states.

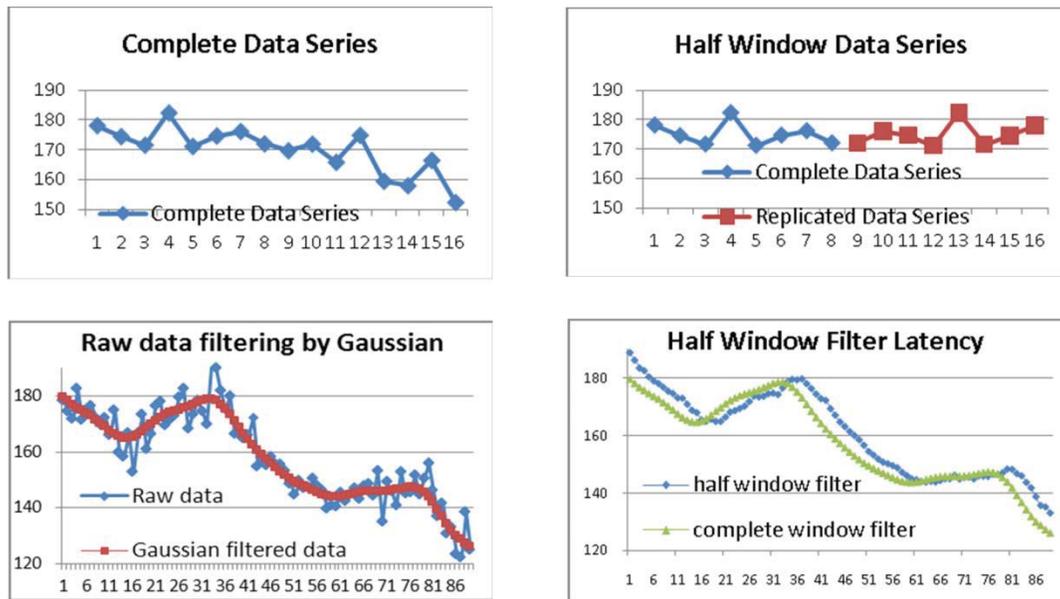


Figure 2.10 Half-window filter latency.

2.5.5 Adaptive Latency Compensation Algorithm

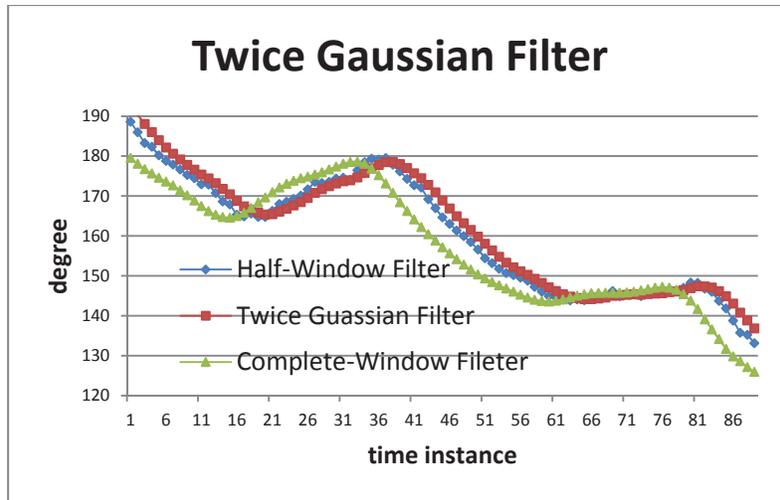
In order to resolve the overshooting problem, the estimated result needs to be forced to the observed data when the module comes to a stop. This is possible because the observed data is very stable and close to the actual value when the module is static. Large collections of observed value show standard deviation is a good indicator of dynamic and static status of the sensor; when the standard deviation is larger than 6, the heading component of the module is in motion, otherwise it is in static or on the way to stopping

(Figure 2.12).. Therefore the adaptive algorithm computes the latency compensated value in the following way: when the standard deviation is no larger than 6, the compensated value is double of the subtraction of the Twice Gaussian filter by the Half-Window Gaussian filter result; otherwise the compensated value is equal to the subtraction of the Twice Gaussian Filter by the observed data.

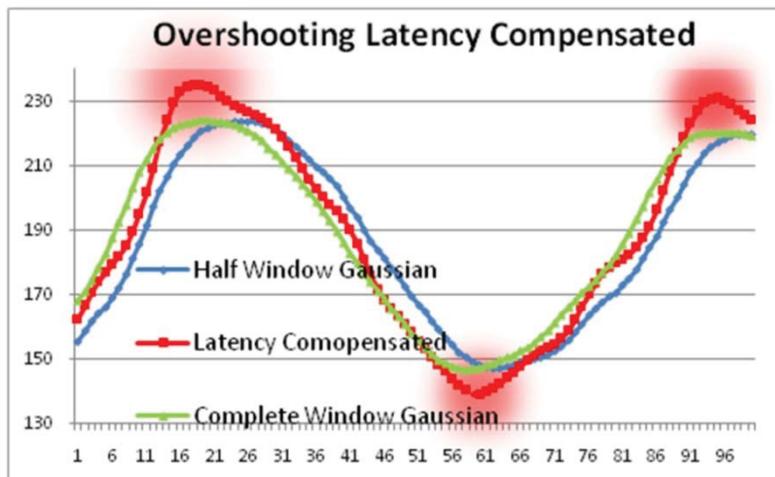
2.6 Validation

The robustness of ARMOR and the SMART framework have been tested in an ongoing excavation collision avoidance project. Electricity conduits in the vicinity of the G.G. Brown Building at the University of Michigan were exported as KML (Keyhole Modeling Language) files from a Geodatabase provided by the DTE Energy Company. The following procedure interprets KML files and builds conduit models (Figure 2.13):

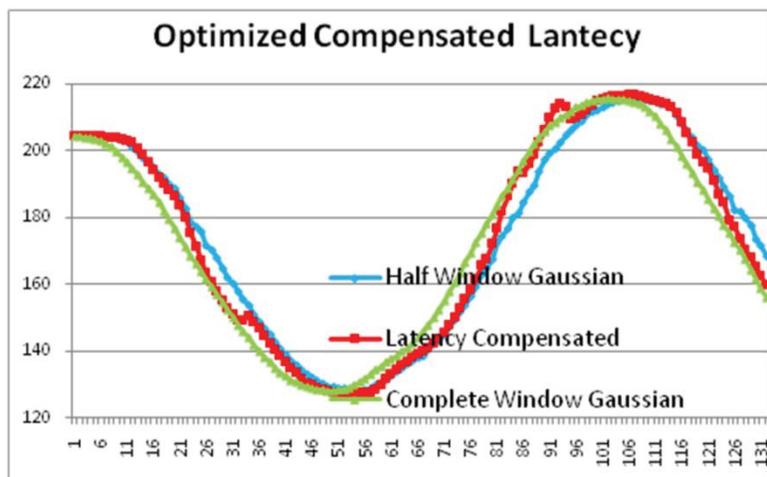
- (1) Extract the spatial and attribute information of pipelines from the KML file using libkml, a library for parsing, generating, and operating in KML (Google, 2008). For example, the geographical location of pipelines is recorded under the Geometry element as “LineString” (Google, 2012). A cursor is thus designed to iterate through the KML file, locate “LineString” elements, and extract the geographical locations.



(A) Additional Latency



(B) Overshooting Problem



(C) Adaptive latency compensation

Figure 2.11 Adaptive Latency Compensation algorithm.

(2) Convert consecutive vertices within one “LineString” from the geographical coordinate to the local coordinate in order to raise computational efficiency during the registration routine. The first vertex on the line string is chosen as the origin of the local coordinate system, and the local coordinates of the remaining vertices are determined by calculating the relative 3D vector between the rest of the vertices and the first one, using the Vincenty algorithm.

(3) In order to save storage memory, a unit cylinder is shared by all pipeline segments as primitive geometry upon which the transformation matrix is built.

(4) Scale, rotate, and translate the primitive cylinder to the correct size, attitude, and position. For simplicity, the normalized vector between two successive vertices is named as the pipeline vector. First, the primitive cylinder is scaled along the X- and Y-axis by the radius of the true pipeline, and then scaled along the Z-axis by the distance between two successive vertices. Secondly, the scaled cylinder is rotated along the axis—formed by the cross product between vector $\langle 0, 0, 1 \rangle$ and the pipeline vector—by the angle of the dot product between vector $\langle 0, 0, 1 \rangle$ and the pipeline vector. Finally, the center of the rotated pipeline is translated to the midpoint between two successive vertices. This step is applied to each pair of two successive vertices.

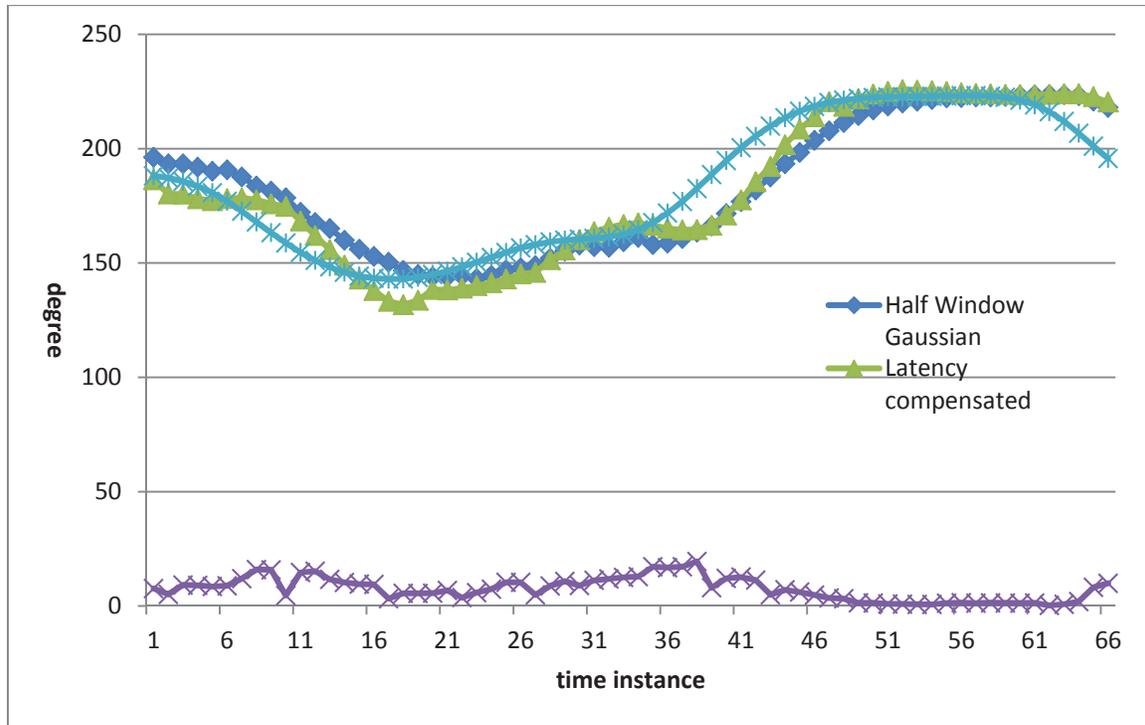


Figure 2.12 Standard deviation indicates the motion pattern.

2.7 Conclusion and Future Work

This chapter has presented the design and implementation of a robust mobile computing platform composed of the rigid hardware platform ARMOR and the application framework SMART. Targeting AR tasks at centimeter-level accuracy, algorithms for both static and dynamic registration have been introduced. Dynamic misregistration continues to be an open research problem and is still under investigation by the author. Several efforts are being made: 1) synchronizing the captured image and sensor measurements; and 2) optimizing the adaptive latency compensation algorithm with image processing techniques (e.g., optical flow can afford more information about the angular speed).

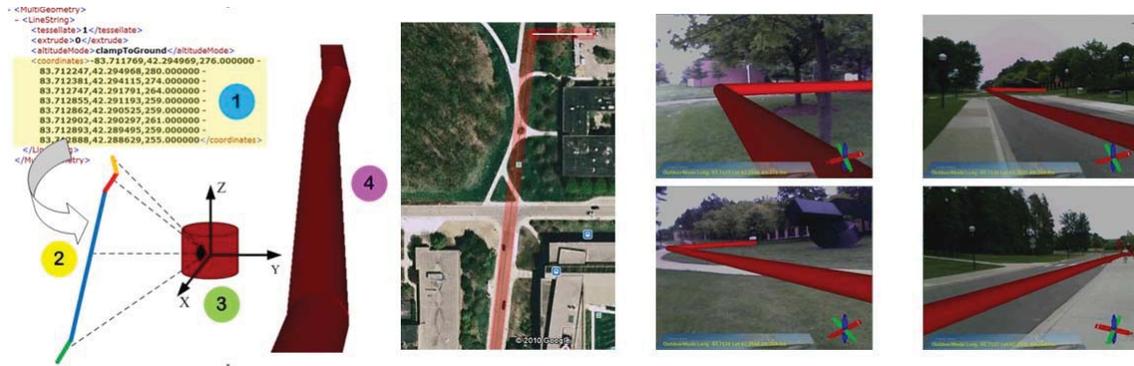


Figure 2.13 Conduit loading procedure, conduits overlaid on Google Earth and field experiment results.



Figure 2.14 Labeling attribute information and color coding on the underground utilities.

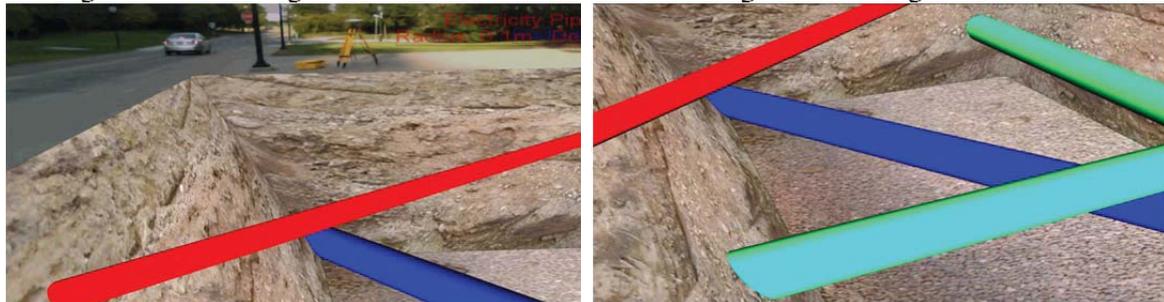


Figure 2.15 An x-ray view of the underground utilities.

2.8 References

- [1] Behzadan, A. H., and Kamat, V. R. (2007). "Georeferenced Registration of Construction Graphics in Mobile Outdoor Augmented Reality." *Journal of Computing in Civil Engineering*, 21(4), 247-258.

- [2] Thomas, B., Piekarski, W., and Gunther, B. "Using Augmented Reality to Visualise Architecture Designs in an Outdoor Environment." *Proc., Proceedings of the Design Computing on the Net*.
- [3] Webster, A., Feiner, S., Macintyre, B., and Massie, W. "Augmented Reality in Architectural Construction, Inspection, and Renovation." *Proc., Proceedings of 1996 ASCE Congress on Computing in Civil Engineering*.
- [4] Roberts, G., Evans, A., Dodson, A. H., Denby, B., Cooper, S., and Hollands, R. "The Use of Augmented Reality, GPS and INS for Subsurface Data Visualization." *Proc., Proceedings of the 2002 FIG XIII International Congress*, 1-12.
- [5] Kamat, V. R., and El-Tawil, S. (2007). "Evaluation of Augmented Reality for Rapid Assessment of Earthquake-Induced Building Damage." *Journal of Computing in Civil Engineering*, 21(5), 303-310.
- [6] Golparvar-Fard, M., Pena-Mora, F., Arboleda, C. A., and Lee, S. (2009). "Visualization of construction progress monitoring with 4D simulation model overlaid on time-lapsed photographs." *Journal of Computing in Civil Engineering*, 23(6), 391-404.
- [7] Azuma, R., Baillet, Y., Behringer, R., Feiner, S., Julier, S., and MacIntyre, B. (2001). "Recent Advances in Augmented Reality." *Journal of Computer Graphics and Applications*, 34-47.
- [8] Azuma, R. (1997). "A Survey of Augmented Reality." *Teleoperators and Virtual Environments*, 355-385.
- [9] Feiner, S., Macintyre, B., and Hollerer, T. "A Touring Machine: Prototyping 3D Mobile Augmented Reality Systems for Exploring the Urban Environment." *Proc., Proceedings of 1997 International Symposium on Wearable Computers*, 74-81.
- [10] Piekarski, W., Smith, R., and Thomas, B. H. "Designing Backpacks for High Fidelity Mobile Outdoor Augmented Reality." *Proc., Proceedings of the 2004 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 280-281.
- [11] Stafford, A., Piekarski, W., and Thomas, H. B. "Implementation of God-like Interaction Techniques for Supporting Collaboration Between Outdoor AR and Indoor Tabletop Users." *Proc., Proceedings of the 2006 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 165-172.
- [12] Jacobs, M. C., Livingston, M. A., and State, A. "Managing Latency in Complex Augmented Reality Systems." *Proc., Proceedings of the 1997 Symposium on Interactives 3D Graphics*, 49-54.

- [13] Liang, O., Shaw, C., and Green, M. "On temporal-spatial realism in the virtual reality environment." *Proc., Proceeding of 1991 Symposium on User Interface Software and Technology*.
- [14] Azuma, R., Hoff, B., Neely, H., and Sarfaty, R. "A Motion-Stabilized Outdoor Augmented Reality." *Proc., Proceedings of the Virtual Reality*, IEEE.
- [15] Behzadan, A. H., Timm, B. W., and Kamat, V. R. (2008). "General-purpose modular hardware and software framework for mobile outdoor augmented reality applications in engineering." *Advances Engineering Informatics*, 22, 90-105.
- [16] Behzadan, A. H., and Kamat, V. R. (2009). "Automated Generation of Operations Level Construction Animations in Outdoor Augmented Reality." *Journal of Computing in Civil Engineering*, 23(6), 405-417.
- [17] PNI (2009). "User Manual Field Force TCM XB."
- [18] Trimble (2007). "AgGPS RTK Base 900 and 450 receivers.", Trimble.
- [19] NMEA (2010). "NMEA data." <<http://www.gpsinformation.org/dale/nmea.htm>>.
- [20] 3DVisior (2010). "Z800 3DVisior User's Manual."
- [21] Tekkeon (2009). "MP3450i/MP3450/MP3750 datasheets."
- [22] Shreiner, D., Woo, M., Neider, J., and Davis, T. (2006). *OpenGL Programming Guide*, Pearson Education.
- [23] Martz, P. (2007). *OpenSceneGraph Quick Start Guide*.
- [24] Vincenty, T. (1975). "Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations." *Survey Reviews*.
- [25] Google (2008). "Introducing libkml: a library for reading, writing, and manipulating KML." <<http://google-opensource.blogspot.com/2008/03/introducing-libkml-library-for-reading.html>>.
- [26] Google (2012). "KML Documentation Introduction." <<https://developers.google.com/kml/documentation/>>.

Chapter 3

Real-Time Occlusion Handling for Dynamic Augmented Reality Using Geometric Sensing and Graphical Shading

3.1 Introduction

As a novel visualization technology, Augmented Reality (AR) has gained widespread attention and seen prototype applications in multiple engineering disciplines for expressing simulation results, visualizing operations design, and conducting inspections, among others. For example, by blending real-world elements with virtual reality, AR helps to alleviate the extra burden of creating complex contextual environments for visual simulations (Behzadan and Kamat, 2009). As an information supplement to the real environment, AR has also been shown to be useful in appending Georeferenced information to a real scene to inspect earthquake-induced building damage (Kamat and

El-Tawil, 2007), or in the estimation of construction progress (Golparvar-Fard, et al., 2009). In both cases, the composite AR view is comprised of two distinct groups of virtual and real objects, and they are merged together by a set of AR graphical algorithms.

Spatial accuracy and graphical credibility are the two keys in the implementation of successful AR graphical algorithms, and the primary focus of this research is exploring a robust occlusion algorithm for enhancing graphical credibility in ubiquitous AR environments. In an ideal scenario, AR graphical algorithms should be capable of intelligently blending real and virtual objects in all three dimensions, instead of superimposing all virtual objects on top of a real-world background, as is the case in most current AR approaches. The result of composing an AR scene without considering the relative depth of the involved real and virtual objects is that the graphical entities in the scene appear to “float” over the real background, rather than blend or co-exist with real objects in that scene.

The occlusion problem is more complicated in outdoor AR where the user expects to navigate the space freely, and where the relative depth between the involved virtual and real content changes dynamically with time. Figure 3.1 (Behzadan and Kamat, 2010) presents a snapshot of a simulated construction operation, where two real objects (tree and truck) are closer than the virtual excavator model to the viewpoint, and should be consequently blocked by the real objects. The right-side image shows visually correct occlusion where the boom and bucket are partially hidden from the scene. However, the

left-side image shows the scene in absence of occlusion, producing an incorrect illusion that the excavator was in front of the tree and truck.

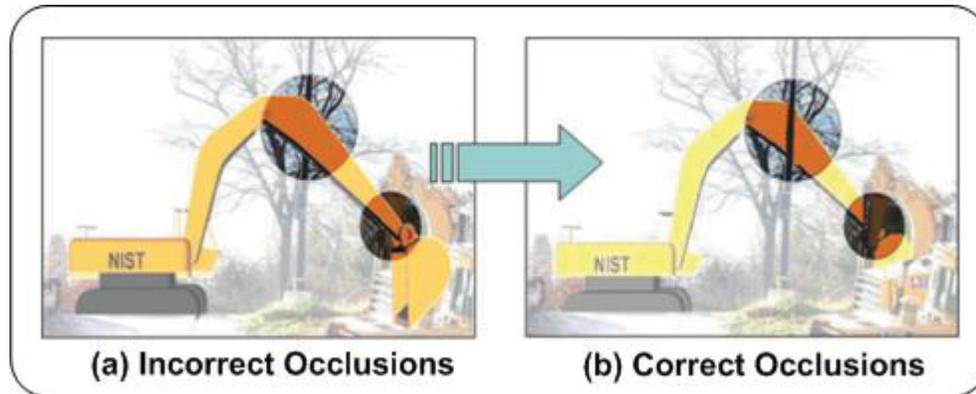


Figure 3.1 Example of occlusion in an outdoor scene. (Behzadan and Kamat, 2010)

Several researchers have explored the AR occlusion problem from different perspectives. Wloka and Anderson (1995) implemented a fast-speed stereo matching algorithm that infers depth maps from a stereo pair of intensity bitmaps. However, random gross errors blink virtual objects on and off and turn out to be very distracting. Berger (1997) proposed a contour-based approach, but with the major limitation that the contours need to be seen from frame to frame. Lepetit and Berger (2000) refined the previous method with a semi-automated approach that requires the user to outline the occluding objects in the key-views, and then the system automatically detects these occluding objects and handles uncertainties on the computed motion between two key frames. Despite the visual improvements, the semi-automated method is only appropriate for post-processing. Fortin and Hebert (2006) studied both a model-based approach using a bounding box, and a depth-based approach using a stereo camera. The former works only with a static viewpoint, and the latter is subject to low-textured areas. Ryu, et al. (2010), Louis and Martinez (2012) tried to increase the accuracy of the depth map by a

region of interest extraction method using background subtraction and stereo depth algorithms; however, only simple background examples were demonstrated. (Tian, et al., 2010) also designed an interactive segmentation and object tracking method for real-time occlusion, but their algorithm fails in the situation where virtual objects are in front of real objects.

In this chapter, the author propose a robust AR occlusion algorithm that uses a real-time Time-of-flight (TOF) camera, an RGB video camera, OpenGL Shading Language (GLSL), and Render to Texture (RTT) techniques to correctly resolve the depth of real and virtual objects in real-time AR visual simulations. Compared with previous work, this approach enables improvements in three ways: 1) Ubiquity: the TOF camera is capable of suppressing the background illumination and enables the designed algorithm to work in both indoor and outdoor environments. It puts the least limitation on context and conditions compared with any previous approach; 2) Robustness: Thanks to the OpenGL depth-buffering method, this method can work regardless of the spatial relationship among involved virtual and real objects; 3) Speed: The author take advantage of the GLSL fragment shader and the RTT technique to parallelize the processing and sampling of the depth map. A recent publication (Koch, et al., 2009) describes a parallel research effort that adopted a similar approach for TV production in indoor environments with a 3D model constructed beforehand. Its main goal is to segment the moving actor from the background.

3.2 Depth Buffer Comparison Approach

This section previews the methodology and computing paradigm for resolving incorrect occlusion with the OpenGL depth buffer on a two-stage rendering basis. The challenges are also briefly described.

3.2.1 Distance Data Source

Getting an accurate measurement of the distance from the virtual and real object to the eye is a fundamental step for correct occlusion. In an outdoor AR environment, the distance from the virtual object to the viewpoint is calculated using the Vincenty algorithm (Vincenty, 1975). This algorithm interprets the metric distance based on the geographical locations of the virtual object and the user. The locations of the virtual objects are predefined by the program. In a simulated construction operation, for example, the geographical locations of virtual building components and equipment are extracted from the engineering drawings. Meanwhile, the location of the user/viewpoint is tracked by GPS. The ARMOR (Augmented Reality Mobile Operating platform) (Dong and Kamat, 2010) utilizes the Trimble AgGPS 332 receiver along with the Trimble AgGPS RTK (Real-time Kinematic) Base 450/900 to continuously track the user's position up to centimeter-level accuracy.

Meanwhile, a TOF camera estimates the distance from the real object to the eye with the help of the time-of-flight principle, which measures the time that a signal travels, with well-defined speed, from the transmitter to the receiver (Beder, et al., 2007). Specifically, the TOF camera measures Radio Frequency (RF)-modulated light sources with phase

detectors. The modulated outgoing beam is sent out with an RF carrier, and the phase shift of that carrier is measured on the receiver side to compute the distance (Gokturk, et al., 2010). Compared with traditional LIDAR scanners and stereo vision, the TOF camera features real-time feedback with high accuracy. It is capable of capturing a complete scene with one shot, and with speeds of up to 40 frames per second (fps). However, common TOF cameras are vulnerable to background light that generates electrons—like artificial lighting and the sun—as this confuses the receiver. Fortunately, the Suppression of Background Illumination (SBI) technology allows the TOF camera used in this project to work flexibly in both indoor and outdoor environments (PMD, 2010).

3.2.2 Two-stage Rendering

Depth buffering, also known as z-buffering, is the solution for hidden-surface elimination in OpenGL, and is usually done efficiently in the Graphics Processing Unit (GPU). A depth buffer is a two-dimensional array that shares the same resolution with the color buffer and the viewport. If enabled in the OpenGL drawing stage, the depth buffer keeps record of the closest depth value to the observer for each pixel. For an incoming fragment at a certain pixel, the fragment will not be drawn unless its corresponding depth value is smaller than the previous one. If it is drawn, then the corresponding depth value in the depth buffer is replaced by the smaller one. In this way, after the entire scene has been drawn, only those fragments that were not obscured by any others remain visible.

Depth buffering thus provides a promising approach for solving the AR occlusion problem. Figure 3.2 shows a two-stage rendering method. In the first rendering stage, the background of the real scene is drawn as usual, but with the depth map retrieved from the

TOF camera written into the depth buffer at the same time. In the second stage, the virtual objects are drawn with depth buffer testing enabled. Consequently, the invisible part of virtual object, either hidden by a real object or another virtual one, will be correctly occluded. The technical implementation is explained in detail in Section 3.5.

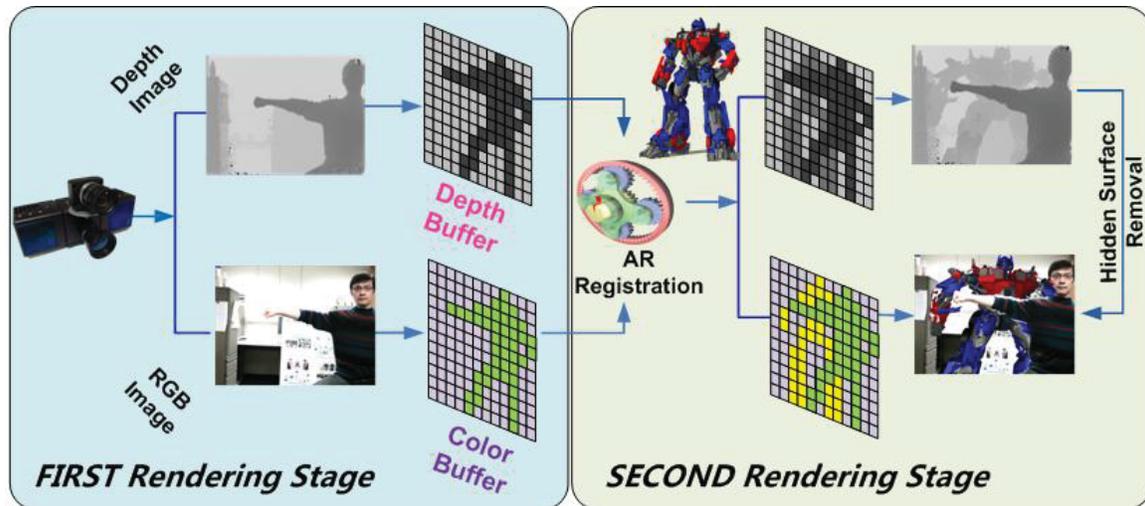


Figure 3.2 Two-stage rendering.

3.2.3 Challenges with the Depth Buffering Comparison Approach

Despite the simple and straightforward approach of depth buffering, there are several challenges when padding the depth buffer with the depth map from the TOF camera.

1) After being processed through the OpenGL graphics pipeline and written into the depth buffer, the distance between the OpenGL camera and the virtual object is no longer the physical distance (Shreiner, et al., 2006). The transformation model is explained in Section 3.3.1. Therefore, the distance for each pixel from the real object to the viewpoint recorded by the TOF camera has to be processed by the same transformation model, before it is written into the depth buffer for comparison.

2) There are three cameras for rendering an AR space: a video camera, a TOF camera, and an OpenGL camera. The video camera captures RGB or intensity values of the real scene as the background, and its result is written into the color buffer. The TOF camera acquires the depth map of the real scene, and its result is written into the depth buffer. The OpenGL camera projects virtual objects on top of real scenes, with its result being written into both the color and depth buffers. In order to ensure correct alignment and occlusion, ideally all cameras share the same projection parameters—the principle points and focal lengths. Even though the chosen TOF camera provides an integrated intensity image that can be aligned with the depth map by itself, the monocular color channel compromises the visual credibility. On the other hand, if an external video camera is used, then the intrinsic and extrinsic parameters of the video camera and TOF camera do not agree (i.e., different principle points, focal lengths, and distortions) (Figure 3.3). Therefore, some image registration methods are required to find the correspondence between the depth map and the RGB image. Two methods are discussed in Section 3.4. Finally, the projection parameters of OpenGL camera are adjustable and can accommodate either an RGB or TOF camera.

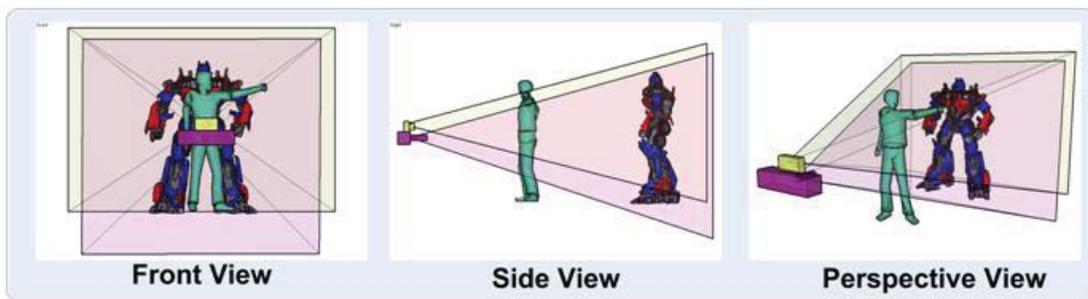


Figure 3.3 Projection parameters disagreement between the TOF camera (bottom) and the video camera (top).

3) Traditional OpenGL pixel-drawing commands can be extremely slow when writing a two-dimensional array (i.e., the depth map) into the frame buffer. Section 3.5 introduces an alternative and efficient approach using OpenGL texture and GLSL.

4) The resolution of the TOF depth map is fixed as 200 x 200, while that of the depth buffer can be arbitrary, depending on the resolution of the viewport. This implies the necessity of interpolation between the TOF depth map and the depth buffer. Furthermore, image registration demands an expensive computation budget if a high-resolution viewport is defined. The method described in Section 3.5 takes advantage of the RTT technique to carry out the interpolation and registration computation in parallel.

3.3 TOF Camera Raw Data Preprocessing

3.3.1 Preprocessing of Depth Map

In OpenGL, the viewing volume is normalized so that all vertices' coordinates lie within the range $[-1, 1]$, in all three dimensions. This is usually referred to as the 'canonical view volume' (Shreiner, et al., 2006), and it prepares the vertices to be transformed from 3D eye space to 2D screen space. Consequently, the Z value (depth value) of a vertex is no longer the physical distance after being normalized. In order to make the distance recorded by the TOF camera comparable with that computed by the OpenGL camera, the same normalization has to be applied to the depth map from the TOF camera.

The distance value recorded by the TOF camera is called z_e (the measured distances from vertices to the viewer in the viewing direction). Table 3.1 summarizes all of the major transformation steps applied to z_e before it is written to the depth buffer, and more details are available from McCreynolds and Blythe (2005): 1) clip coordinate z_c (distance values in the clip space where objects outside of the view volume are clipped away) is the result of transforming z_e by the camera projection matrix; 2) z_c divided by w_c (homogenous component in the clip space) is called a perspective divide that generates normalized coordinate z_{cvv} ; 3) since the range of z_{cvv} (distance values in the CVV space) is $[-1,1]$, it needs to be biased and scaled to the depth buffer range $[0,1]$ before it is sent there (Figure 3.4).

Table 3.1 - The transformation steps applied on the raw TOF depth image.

Name	Meaning	Operation	Expression	Range
Z_e	Distance to the viewpoint	Acquired by TOF camera		$(0, +\infty)$
Z_c	Clip coordinate after projection transformation	$M_{ortho} * M_{perspective} *$ $[X_e \ Y_e \ Z_e \ W_e]^T$	$= \frac{Z_e * (-n) - W_e}{-2 * n * W_e}$ n and f is the near and far plane, W_e is the homogenous component in eye coordinate, and is usually equal to 1	$[-n, f]$
Z_{cvv}	Canonical View Volumn	Z_c / W_c ($W_c = Z_e$, and is the homogenous component in clip coordinate)	$= \frac{Z_e}{W_e} - \frac{2 * n * W_e}{W_e * (-n)}$	$[-1, 1]$
Z_d	Value sent to depth buffer	$(Z_{ndc} + 1) / 2$	$= \frac{Z_{cvv} + 1}{2 * (-n)} - \frac{W_e}{W_e * (-n)} + 0.5$	$[0, 1]$

3.3.2 Processing of the Intensity Image

This step is only necessary if the integrated intensity image provided by the TOF camera is used. The intensity image's raw color data format is incompatible with the data format supported by OpenGL. Furthermore, the intensity distribution is biased and needs to be redistributed for better visual effect.

$$P(v) = \frac{CDF(v) - CDFmin}{(width * height) - CDFmin} * (Level - 1) \quad 3-1$$

Since an unsigned byte (8 bits represents 0 ~ 255) is the data type used in OpenGL for showing intensity values, the raw TOF intensity image needs to be refined in two ways. First, the raw intensity values are spread from 1,000 to 20,000, and thus have to be scaled to [0,255]; second, the intensity is represented by close contrast values. In other words, the most frequent color values occupy only a small bandwidth in the entire spectrum. Therefore, histogram equalization is helpful in spreading out the most frequent intensity values on the spectrum for improved visual appeal (Figure 3.4). The basic idea of equalization is to linearize the cumulative distribution function (CDF) across the spectrum from 0 to 255. The transformation can be described by Equation 3-1 (Acharya and Ray, 2005). CDF is the cumulative distribution function of a given intensity image; v is the original intensity value of a given pixel, $P(v)$ is the intensity value after equalization for that pixel; $Level$ is the total number of gray scale after equalization, 256 in OpenGL's case. An optimized equalization algorithm is presented in Appendix B.1.

3.4 Depth Map and RGB Image Registration

3.4.1 Image Registration Between Depth Map and Intensity Image

Given that the depth map and the intensity image are interpreted from the homogeneous raw data source of the TOF camera, these two images have the same intrinsic and extrinsic parameters. Pixels in these two images thus have a one-to-one mapping relation. In other words, no explicit registration step is required. After the preprocessing step (mentioned in Section 3.3) and the interpolation for higher resolution (described in Section 3.5.1), the depth map and the intensity image are ready to be used in the two-stage rendering. Figure 3.5 shows comparisons between occlusion-enabled and -disabled modes. The TOF camera is positioned approximately 7m away from the wall of a building, which is surrounded by a flowerbed. A mini excavator model is positioned about 3m away from the TOF camera, and a person stands in front of the excavator.

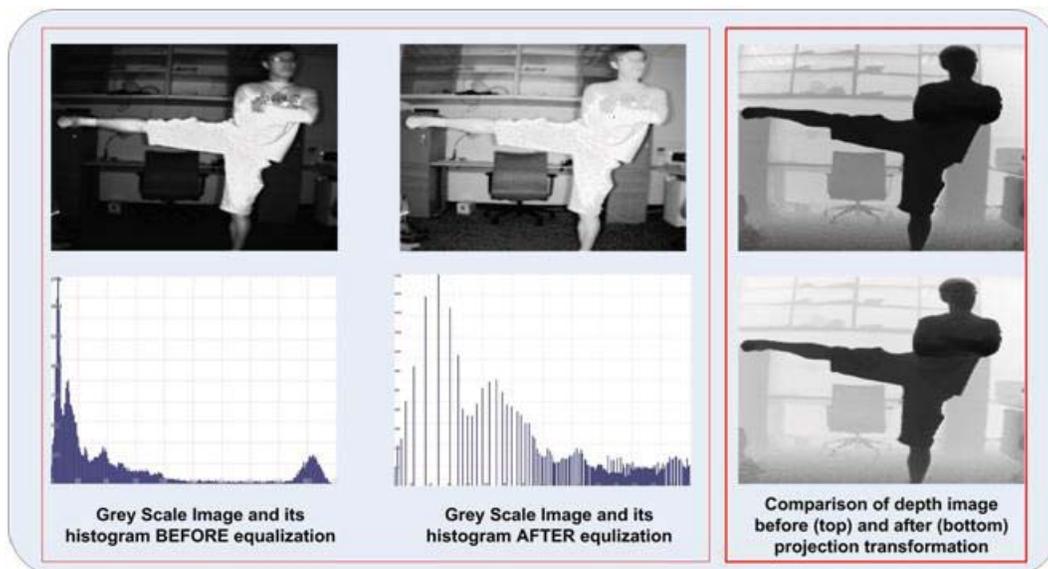


Figure 3.4 Preprocessing of the TOF intensity and depth image.

3.4.2 Image Registration Between Depth Map and RGB Image Using Homography

Homography is the invertible mapping of points and lines from plane to plane, so that three points lying on the same line have collinear mapped points. A homography matrix is 3×3 with 8 unknowns, and its mathematical definition can be found in Hartley and Zisserman (2003). Theoretically, homography requires two images to be taken by cameras that have only pure rotation relative to each other. In other words, ideally no relative translation should be involved. However, given that the RGB camera has a small translation to the TOF camera (Figure 3.6), the image registration can be approximated by the homography model.

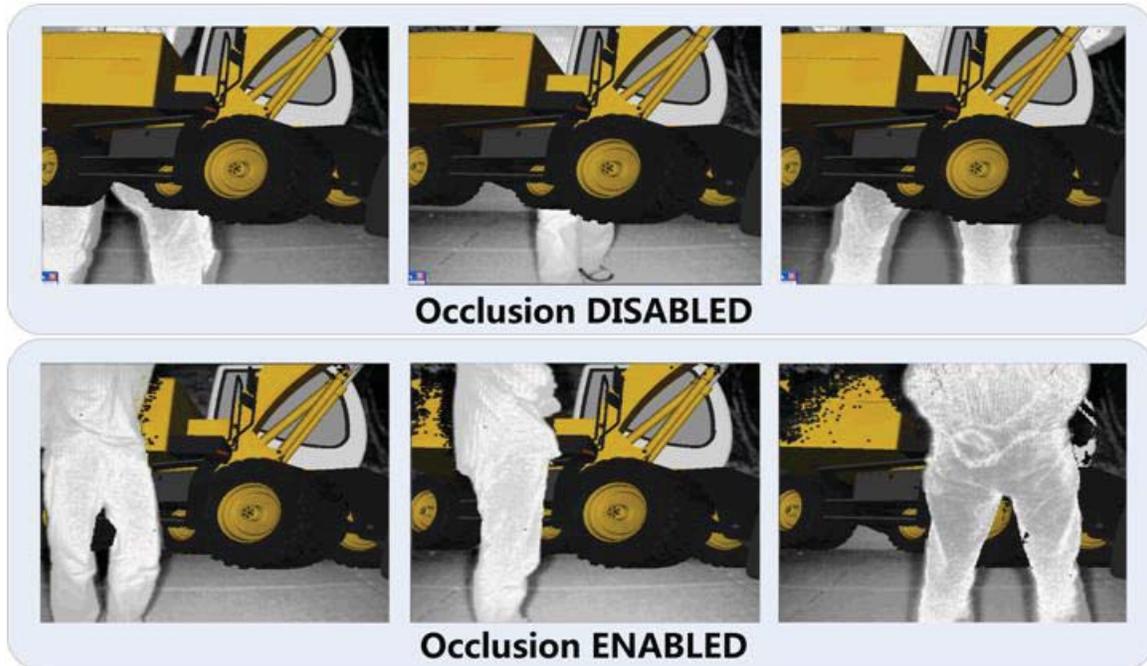


Figure 3.5 Occlusion effects comparison using the TOF camera's intensity image and depth map.

The estimation of homography is typically derived by the Direct Linear Transformation (DLT) algorithm given a sufficient set of point correspondence. Solving 8 unknowns requires 4 point correspondences to be available. If more than 4 pairs are available, then a more accurate estimation can be acquired by minimizing certain cost functions (Dubrofsky, 2007). Here we choose a non-linear homography estimation implementation from (Lourakis, 2011). Figure 3.6 shows the registration results using homography, where the RGB image is transformed to the TOF depth image coordinate frame. Since it is difficult to find identical points using the depth map, we instead use the intensity image recorded by the TOF camera that has a one-to-one mapping to the depth map.

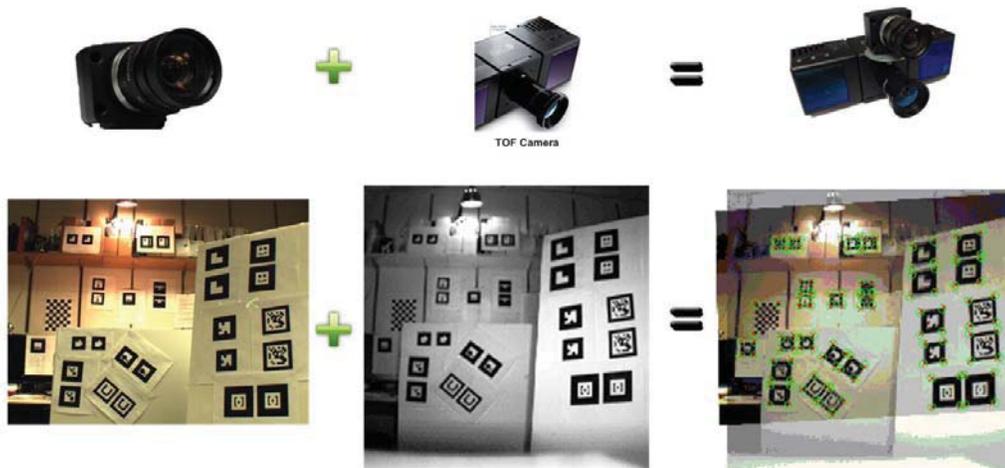


Figure 3.6 The identical points on two images are used to calculate the homography matrix that registers the RGB image with the TOF depth image.

It is computationally expensive to multiply each RGB image point with the homography matrix in real time. In order to accelerate the process, the mapping relationship from RGB to TOF image points is pre-computed and stored as a look-up table. The depth value is then bilinearly interpolated for the corresponding RGB image

points on the fly. Section 3.5.2 will discuss how we parallelize the interpolation process using the RRT technique.

A similar experimental setting to the one in the previous section is conducted, and Figure 3.7 shows the comparison between scenarios when occlusion is enabled and disabled. It is obvious that occlusion provides much better spatial cues and realism. The effect is also superior to the previous monocular intensity image in terms of visual appeal.



Figure 3.7 Occlusion effects comparison using homography mapping between the TOF camera and the RGB camera.

3.4.3 Image Registration Between Depth Map and RGB Image Using Stereo Projection

Even though homography mapping yields good occlusion results in most cases, the approximation cannot always faithfully resolve occlusion. Figure 3.8 shows one scenario where the blank gap between the virtual and real object is wide enough to be visually identified. In order to be able to always assign the correct RGB value to each depth map pixel, a stereo registration method is needed. This means each 2D point on the depth map is back-projected to the 3D space, and then re-projected onto the RGB image plane. Intrinsic and extrinsic parameters of both cameras need to be calibrated for the stereo registration.

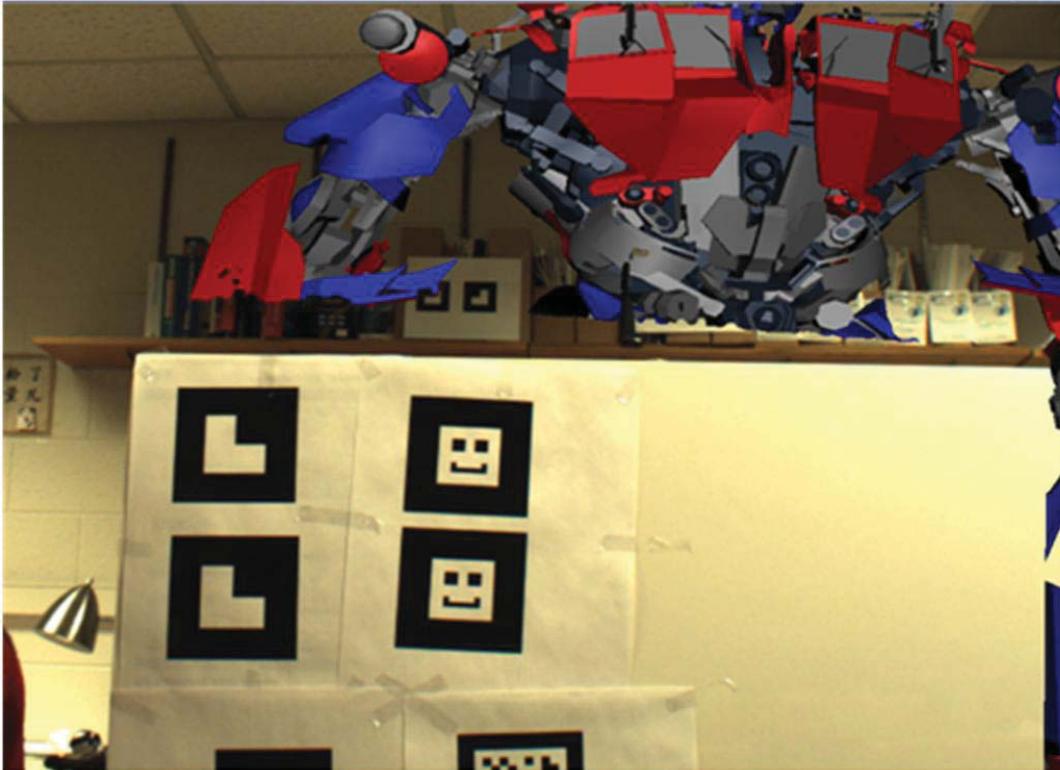


Figure 3.8 The homography approximation fails to correctly resolve occlusion on the silhouette.

3.4.3.1 Assign RGB Value to Depth Map Pixel

As illustrated in Figure 3.9, X^{2D_TOF} is one pixel on the depth map. In order to find its corresponding point X^{2D_RGB} on the RGB image, X^{2D_TOF} is first back-projected to the 3D space as X_{3D_TOF} given K^{TOF} , the intrinsic matrix of the TOF camera. Since X_{3D_TOF} is expressed in the TOF camera's coordinate system, it needs to be transformed to X^{3D_RGB} in the RGB camera's coordinate system using the extrinsic matrix $[R, T]$ between the RGB and TOF cameras. R and T represent the relative rotation and translation from the TOF to the RGB image. R and T represent the relative rotation and translation from the TOF to the RGB image. Finally X^{3D_RGB} is projected onto the RGB image plane as X^{2D_RGB} using K^{RGB} , the intrinsic matrix of the RGB camera. Hartley and Zisserman (2003) mathematically describe this process via Equation 3-2.

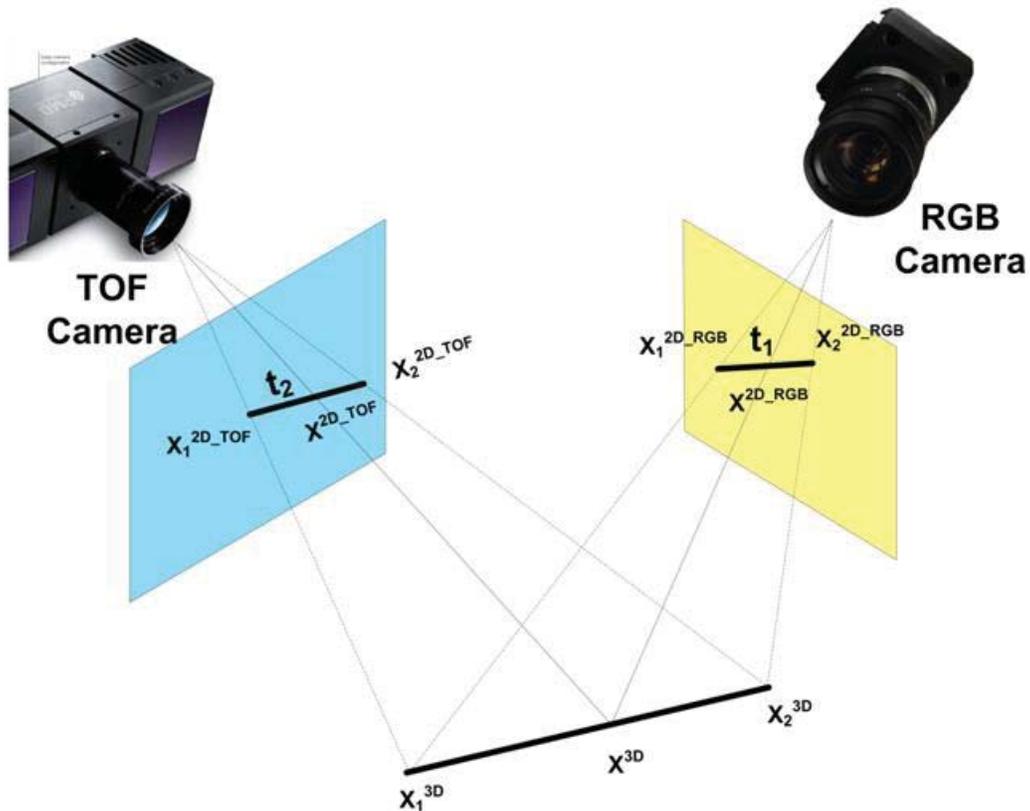


Figure 3.9 The stereo registration between the TOF and RGB cameras.

$$X^{2D_RGB} = K^{RGB} R K^{TOF^{-1}} X^{2D_TOF} + K^{TOF} T / Z(X^{2D_TOF}) \quad 3-2$$

The Z value is the physical distance measured by the TOF camera, which is known for each pixel on the TOF depth map. K^{RGB} , K^{TOF} , R, and T are pre-calibrated. The successful implementation of the above model highly relies on the accurate intrinsic and extrinsic calibrations. The author take advantage of the 'camera_calibration' and 'stereo_calibration' provided by the OpenCV library (OpenCV, 2012). Figure 3.10 shows one pair of images for the extrinsic calibration.

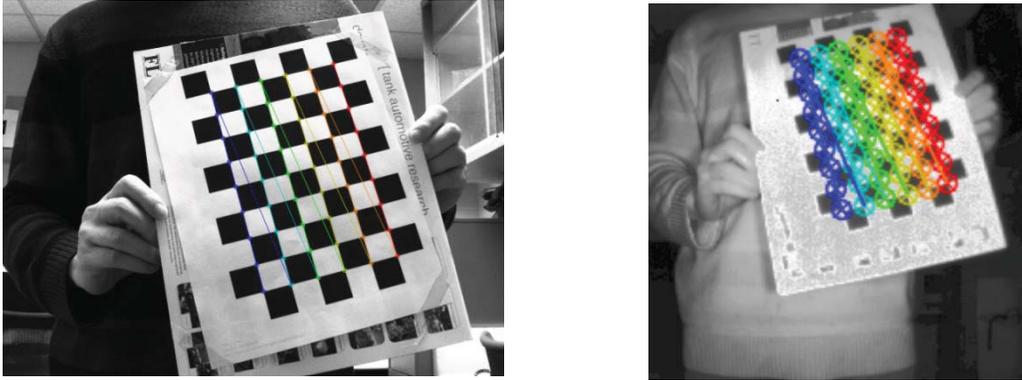


Figure 3.10 Chessboard method used for extrinsic calibration between the TOF and RGB cameras.

3.4.3.2 Interpolate RGB Values for Intermediate Depth Map Pixels

Given the low resolution of the TOF camera (200 x 200), interpolation is indispensable if a higher-resolution augmented image needs to be rendered. However, due to the perspective projection involved, it is incorrect to render a 200 x 200 augmented image, and then leave it to the texture to (bi)linearly interpolate to a higher resolution, like the case in Section 3.4.1. The reason can be inferred from Figure 3.9. This time, X^{2D_TOF} is an intermediate point for interpolation between $X_1^{2D_TOF}$ and $X_2^{2D_TOF}$, whose corresponding

points are $X_1^{2D_RGB}$ and $X_2^{2D_RGB}$ on the RGB image. In order to correctly assign the RGB value for X^{2D_TOF} , its corresponding point should be linearly interpolated on the RGB image via Equation 3-3. Due to the fact that $t_1 \neq t_2$, RGB value for X^{2D_TOF} would be incorrectly assigned if it were linearly interpolated via Equation 3-4.

Lindner, et al. (2007) adopt the projective texture mapping approach (Segal, et al., 1992) to interpolate the RGB value correctly on the depth image. Here we propose a more straightforward approach: the depth value is first interpolated for each intermediate pixel on the depth map, then the RGB value for all of the original and intermediate depth map pixels are interpolated in the same way as discussed in Section 3.4.3.1. The computational cost is higher than that of projective texture mapping on the CPU, but negligible on the GPU using RTT.

$$RGB(X^{2D_RGB}) = (1 - t_1) * RGB(X_1^{2D_RGB}) + t_1 * RGB(X_2^{2D_RGB}) \quad 3-3$$

$$RGB(X^{2D_TOF}) = (1 - t_2) * RGB(X_1^{2D_TOF}) + t_2 * RGB(X_2^{2D_TOF}) \quad 3-4$$

As discussed by Low (2002), the depth value of the intermediate point cannot be linearly interpolated on the depth map for the same reason (i.e. perspective projection). However, Low (2002) has proved that it is possible to obtain perspective correct results by linearly interpolating the reciprocal of the depth value via Equation 3-5.

$$Z(X^{2D_TOF}) = \frac{1}{(1-t) * \frac{1}{Z(X_1^{2D_TOF})} + \frac{1}{Z(X_2^{2D_TOF})}} \quad 3-5$$

3.5 Technical Implementation with OpenGL Texture and GLSL

The interpolation and transformation operations that have been discussed in Section 3.4 are very costly if computed on the CPU in a serial way. However, by using OpenGL texture and GLSL, these operations can be conducted efficiently on the GPU and achieve interactive frame rates.

3.5.1 Interpolation using OpenGL Texture

This section describes how we efficiently render the interpolated result to the frame buffer. This step is needed by all three image registration methods. Here we take the method described in Section 3.4.1 as an example to demonstrate the technical implementation.

After preprocessing, the depth map and intensity image are ready to be written into the depth buffer and color buffer, respectively. However, a challenging issue is how to write to the frame buffers fast enough so that real-time rendering is possible. This is a two-part issue: 1) the arbitrary size of the frame buffers requires interpolation of the original 200 x 200 images. While software interpolation can be very slow, texture filtering presents a hardware solution, since texture sampling is so common that most GPUs implement it very efficiently; and 2) even though OpenGL command `glDrawPixels()` (with `GL_DEPTH_COMPONENT` and `GL_RGBA` parameter) provides an option for writing array into frame buffers, no modern OpenGL implementation can

efficiently accomplish this. This is because, for every single frame, data is passed from the main memory to OpenGL, then to the graphics card.

Texturing a quad and manipulating its depth and color values in the GLSL fragment shader is an alternative approach, and can be very efficient. Texture is a container of one or more images in OpenGL (Shreiner, et al., 2006), and is usually bound to a geometry. Moreover, geometry can be associated with multiple textures. Here the OpenGL geometric primitive type `GL_QUADS` is chosen as a binding target, and two 2D textures are pasted on it: an intensity image texture and a depth map texture (Figure 3.11). The quad is projected orthogonally. During the rasterization stage, the textures coordinates of each fragment (pixel) are (bi)linearly interpolated according to the size of the OpenGL frame buffers.

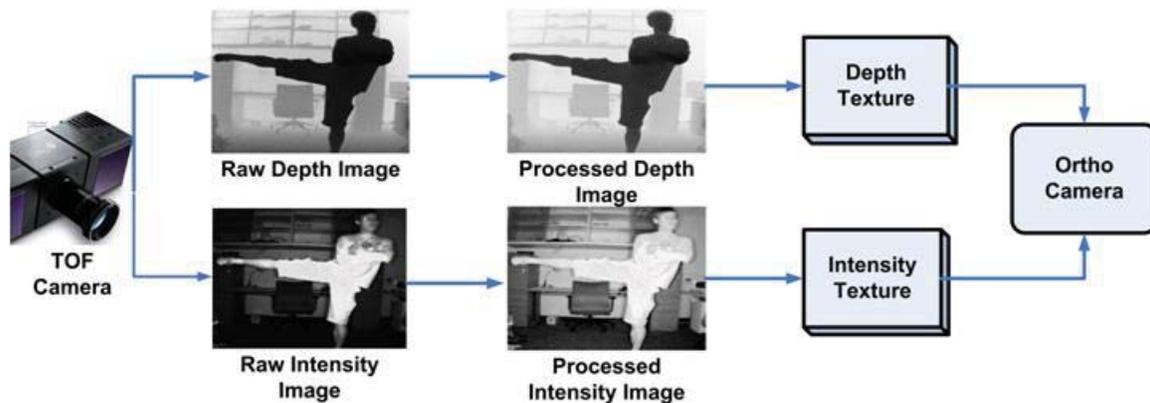


Figure 3.11 Attaching multiple textures to the quad for hardware interpolation.

3.5.1.1 Intensity Image Texture

Since modifying the existing texture object on a fixed region of the GPU is computationally cheaper than allocating a new one for each frame, it is better to use `glTexSubImage2D()` to repeatedly replace the texture data with new captured intensity

images (Shreiner, et al., 2006). However, the intensity image must be loaded to an initial, larger texture with size in both directions set to the next biggest power of two than its resolution (i.e., 256 x 256). Accordingly, the texture coordinates are assigned as (0, 0), (200/256, 0), (200/256, 200/256), and (0, 200/256) in counterclockwise order of the quad.

3.5.1.2 Depth Map Texture

The same sub image replacement strategy is applied to depth map texture. However, even though the internal format of the texture is set to `GL_DEPTH_COMPONENT`, the depth value written into the depth buffer is not the depth map texture value, but the actual depth value of the quad geometry. Therefore the depth value of the quad needs to be manipulated in the fragment shader according to the depth map texture value. A fragment shader operates on every fragment that is spawned by the rasterization phase in the OpenGL pipeline. One input for the fragment processor is interpolated texture coordinates, and the common end result of the fragment processor is a color value and a depth for that fragment (Randi, et al., 2006). These features make it possible to alternate polygon depth values so that the TOF depth map can be written into the depth buffer. The GLSL fragment shader is listed in Appendix B.2.

3.5.2 Homography Registration Using Render to Texture

In the homography registration implementation, one extra step beyond the method described in Section 3.5.1 is interpolating a depth map that has a one-to-one mapping relationship with the RGB image at the pixel level. As discussed in Section 3.4.2, given a pixel on the RGB image, its corresponding point on the depth map can be found through the pre-computed homography look-up table, and the depth value is (bi)linearly

interpolated from the closest four points. However, since the RGB camera used for this experiment comes with a resolution of 1280 x 960, it implies that there is 1280 x 960 loops for each rendered frame if computed on the CPU. This dramatically slows down the rendering frame rate.

Since this kind of pixel-by-pixel computation is highly parallel, the computation performance can be boosted if it is carried out on the GPU. In order to enable computation on the GPU, the procedure must be comprised of 1) uploading data to the GPU; 2) interpolation; and 3) transferring the result to the depth texture. Here we choose the Render to Texture (RTT) technique to fully optimize this procedure. The basic idea of RTT is that it renders a frame as usual, but the rendering results are written to texture(s) instead of to frame buffers (OpenGL-Tutorial, 2012). RTT is a common and useful technique that has been widely applied in generating projective texture, shadow mapping, multi-sampling, etc.

As illustrated in Figure 3.12, the homography look-up table (1280 x 960) and the raw depth map (200 x 200) are uploaded as textures. A high-resolution depth map (1280 x 960) is also interpolated in the fragment shader and written to the depth texture. The depth texture and the RGB texture are then pasted to the quad geometry, and rendered to the frame buffers, as discussed in Section 3.5.1. On a Toshiba Qosmio X505 with Intel M450 and NVIDIA GeForce GTS 360M, the frame rate with major computation load on the GPU is 15fps, which is 1.5 times faster than the case with a major load on the CPU (acronym for Central Processing Unit). In fact, when the computation load is on the GPU,

the frame rate is mainly limited by the speed of retrieving images from the TOF and RGB cameras. The GLSL fragment shader is listed in B.3.

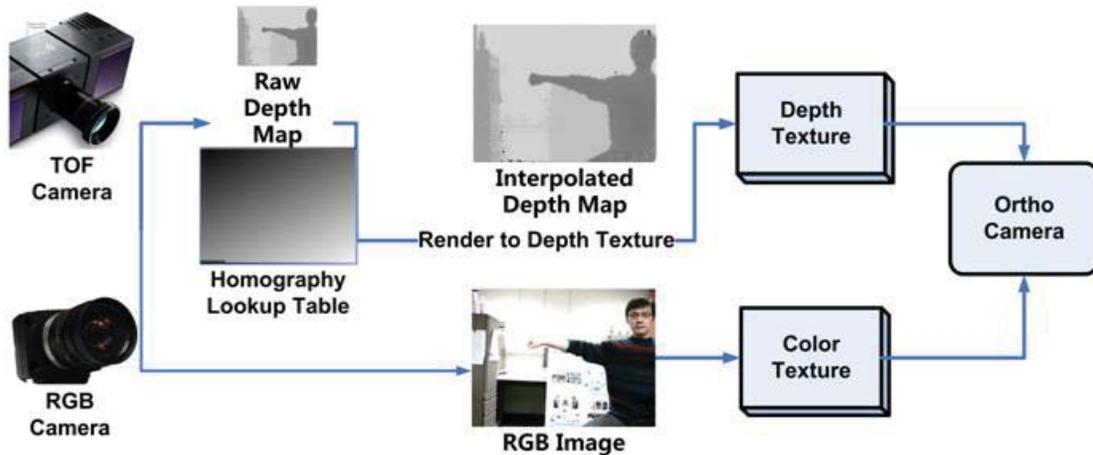


Figure 3.12 Using Render to Texture to compute the homography registration on the GPU.

3.5.3 Stereo Registration Using Render to Texture

A similar RTT technique is used to assign an RGB value to each pixel on the depth map through stereo registration. The only difference is that, instead of writing to one texture, this time RTT writes to two targets simultaneously: the depth texture and the RGB texture.

As illustrated in Figure 3.13, the raw depth image (200 x 200) and the raw RGB image (1280 x 960) are uploaded as textures. For each fragment, its depth value is linearly interpolated, as discussed in Section 3.4.3.2, and the associated RGB value is identified through stereo registration, as described in Section 3.4.3.1. The former result is pushed to the depth texture, the latter to the RGB texture. Finally, these two textures are rendered to the frame buffers. Since the TOF camera has a wider field of view than that

of the RGB camera in the vertical direction, the bottom part of the depth map cannot be matched with any valid RGB value. This problem is solved by clipping the invalid region of the RGB and depth textures. For example, 1000 x 1000 depth and RGB texture are generated by RTT, and only the regions [20, 980] x [20, 560] on both textures are sent to the frame buffers. The GLSL fragment shader is listed in Appendix B.4.

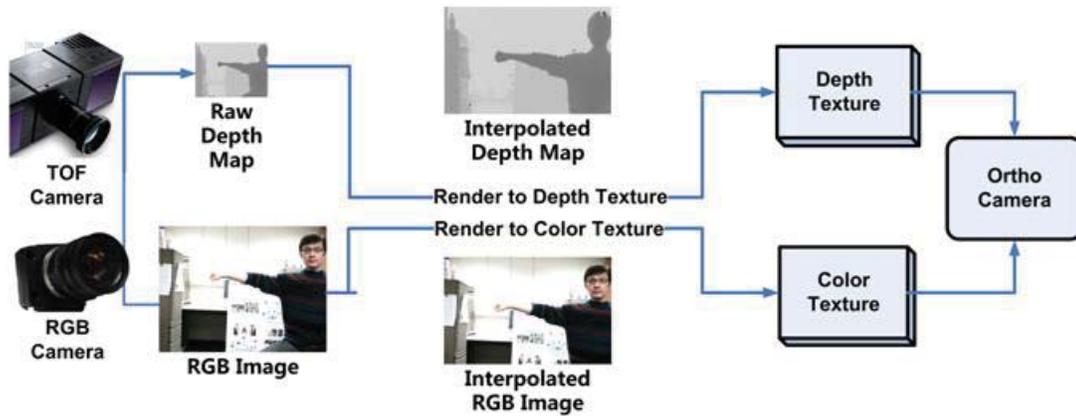


Figure 3.13 Using Render to Texture to compute the stereo registration on the GPU.

3.6 Validation

Despite the outstanding performance of the TOF camera in speed and accuracy, the biggest technical challenge it faces is modular error, since the receiver decides the distance by measuring the phase offset of the carrier. The maximum valid range is limited by the RF carrier wavelength. For instance, the standard measurement range of CamCube3.0 is 7m (PMD, 2010). If an object happens to be 8m away from the camera, its distance is represented as 0.5m ($8 \bmod 7.5$) on the depth map, instead of 8m. This can create incorrect occlusion in outdoor conditions, where ranges can easily go beyond 7m. The author have been exploring object detection and segmentation as possible options to

mitigate this limitation. In the presented research, the experiment range is intentionally restricted to within 7m.



Figure 3.14(a) Indoor simulated construction processes with occlusion disabled.



Figure 3.14(b) Indoor simulated construction processes with occlusion enabled.

Two sets of validation experiments have been conducted in both indoor and outdoor environments. In both cases, the TOF camera is positioned about 7.5m away, facing the wall. Demo videos of both experiments can be found on the website <<http://pathfinder.engin.umich.edu/videos.htm>>. All of the virtual models are courtesy of the Google 3D Warehouse community.

In the indoor experiment, a forklift picks up a virtual piece of cardboard in front of the virtual stack, and maneuvers to put it on top of a physical piece of cardboard. In the

meantime, a construction worker passes by with a buggy, and then puts a physical bottle beside the virtual cardboard. Figure 3.14 shows the results of indoor snapshots to validate the occlusion correctness.

In the outdoor experiment, a construction worker stands on a virtual scissor lift and paints the wall. She then jumps off of the scissor lift, pushes the debris to the virtual pile of dirt with a physical shovel, and operates the virtual mini-dozer. Figure 3.15 shows the results of outdoor snapshots to validate the occlusion correctness. It is clear from the composite visualization that occlusion provides much better spatial cues and realism for outdoor AR visual simulation.

3.7 Conclusion and Future Work

This chapter described research that designed and implemented an innovative approach to resolve AR occlusion in ubiquitous environments using real-time TOF camera distance data and the OpenGL frame buffers. Sets of experimental results demonstrated promising depth visual cues and realism in AR visual simulations. However, several challenging issues remain and are currently being investigated by the author. For example, the stereo registration algorithm is sometimes subject to phantom effects on the silhouette, whose causes might be due to the imperfect intrinsic calibration of the TOF camera, and the extrinsic calibration of the TOF and RGB camera poses. Phantom effect implies over-occluding virtual objects and leaving blank gaps between virtual and real objects.



Figure 3.15(a) Outdoor simulated construction processes with occlusion disabled.



Figure 3.15(b) Outdoor simulated construction processes with occlusion enabled.

The author also acknowledge a second challenging issue—that the current 7.5m average operational range of the TOF camera puts a limitation on full-scale outdoor simulation visualization. However, the occlusion algorithm designed here is generic and scalable so that future hardware with improved range and accuracy can be plugged into the current AR visualization framework with little modification to the core algorithm. The author are also studying the feasibility of implementing hybrid methods, like stereovision and object detection, to mitigate this limitation. This occlusion algorithm has been built into the SMART framework (acronym for Scalable and Modular Augmented Reality Template). The current version of the occlusion source code, which includes

SMART framework, can be found on the website:

<http://pathfinder.engin.umich.edu/software.htm>.

3.8 References

[1] Behzadan, A. H., and Kamat, V. R. (2009). "Automated Generation of Operations Level Construction Animations in Outdoor Augmented Reality." *Journal of Computing in Civil Engineering*, 23(6), 405-417.

[2] Kamat, V. R., and El-Tawil, S. (2007). "Evaluation of Augmented Reality for Rapid Assessment of Earthquake-Induced Building Damage." *Journal of Computing in Civil Engineering*, 21(5), 303-310.

[3] Golparvar-Fard, M., Pena-Mora, F., Arboleda, C. A., and Lee, S. (2009). "Visualization of construction progress monitoring with 4D simulation model overlaid on time-lapsed photographs." *Journal of Computing in Civil Engineering*, 23(6), 391-404.

[4] Behzadan, A. H., and Kamat, V. R. (2010). "Scalable Algorithm for Resolving Incorrect Occlusion in Dynamic Augmented Reality Engineering Environments." *Journal of Computer-Aided Civil and Infrastructure Engineering*, 25(1), 3-19.

[5] Wloka, M. M., and Anderson, B. G. "Resolving Occlusion in Augmented Reality." *Proc., Proceedings of Symposium on Interactive 3D Graphics*, 5-12.

[6] Berger, M.-O. "Resolving Occlusion in Augmented Reality : a Contour Based Approach without 3D Reconstruction." *Proc., Proceedings of 1997 IEEE Conference on Computer Vision and Pattern Recognition*.

[7] Lepetit, V., and Berger, M.-O. "A Semi-Automatic Method for Resolving Occlusion in Augmented Reality." *Proc., Proceedings of 2000 IEEE Conference on Computer Vision and Pattern Recognition*.

[8] Fortin, P.-A., and Hebert, P. "Handling Occlusions in Real-time Augmented Reality : Dealing with Movable Real and Virtual Objects." *Proc., Proceedings of the 2006 Canadian Conference on Computer and Robot Vision*, 54-62.

[9] Ryu, S.-W., Han, J.-H., Jeong, J., Lee, S. H., and Park, J. I. "Real-Time Occlusion Culling for Augmented Reality." *Proc., Proceedings of the 2010 Korea-Japan Joint Workshop on Frontiers of Computer Vision*, 498-503.

- [10] Louis, J., and Martinez, J. (2012). "Rendering Stereoscopic Augmented Reality Scenes with Occlusions using Depth from Stereo and Texture Mapping." *Construction Research Congress 2012*.
- [11] Tian, Y., Guan, T., and Wang, C. (2010). "Real-Time Occlusion Handling in Augmented Reality Based on an Object Tracking Approach." *Sensors*, 2885-2900.
- [12] Koch, R., Schiller, I., and Bartczak, B. (2009). "MixIn3D: 3D Mixed Reality with ToF-Camera." *DYNAMIC 3D IMAGING*, Spinger.
- [13] Vincenty, T. (1975). "Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations." *Survey Reviews*.
- [14] Dong, S., and Kamat, V. R. "Robust Mobile Computing Framework for Visualization of Simulated Processes in Augmented Reality." *Proc., Proceedings of the 2010 Winter Simulation Conference*, Institute of Electrical and Electronics Engineers, 3111-3122.
- [15] Beder, C., Bartczak, B., and Koch, R. "A Comparison of PMD-Cameras and Stereo-Vision for the Task of Surface Reconstruction using Patchlets." *Proc., Proceedings of 2007 IEEE Conference on Computer Vision and Pattern Recognition*.
- [16] Gokturk, B. S., Yalcin, H., and Bamji, C. "A Time-Of-Flight Depth Sensor - System Description, Issues and Solutions." *Proc., Proceedings of 2010 IEEE Conference on Computer Vision and Pattern Recognition Workshop*.
- [17] PMD (2010). "PMD CamCube 3.0."
- [18] Shreiner, D., Woo, M., Neider, J., and Davis, T. (2006). *OpenGL Programming Guide*, Pearson Education.
- [19] McCreynolds, T., and Blythe, D. (2005). *Advanced Graphics Programming Using OpenGL*, Elsevier Inc, San Francisco, CA.
- [20] Acharya, T., and Ray, A. K. (2005). *Image processing : principles and applications*, John Wiley & Sons, Inc.
- [21] Hartley, R., and Zisserman, A. (2003). *Multiple View Geometry in Computer Vision*, Cambridge University Press.
- [22] Dubrofsky, E. (2007). "Homography Estimation." Master of Science, Carleton University.

[23] Lourakis, M. (2011). "homest: A C/C++ Library for Robust, Non-linear Homography Estimation." <<http://www.ics.forth.gr/~lourakis/homest/>>.

[24] OpenCV (2012). "OpenCV ", <<http://opencv.willowgarage.com/wiki/>>.

[25] Lindner, M., Kolb, A., and Hartmann, K. (2007). "Data-Fusion of PMD-Based Distance-Information and High-Resolution RGB-Images." *International Symposium on Signals, Circuits and Systems, 2007. ISSCS 2007*.

[26] Segal, M., Korobkin, C., Widenfelt, R. V., Foran, J., and Haeberli, P. "Fast shadows and lighting effects using texture mapping." *Proc., Proceedings of the 1992 annual conference on Computer graphics and interactive techniques*, 249-252.

[27] Low, K.-L. (2002). "Perspective-Correct Interpolation."

[28] Randi, R. J., Licea-Kane, B. M., Ginburg, D., Kessenich, J., Lichtenbelt, B., Malan, H., and Weiblen, M. (2006). *OpenGL Shading Language*.

[29] OpenGL-Tutorial (2012). "Tutorial 14 : Render To Texture." <<http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-14-render-to-texture/>>.

Chapter 4

Sensitivity Analysis of Augmented Reality-Assisted Building Damage Reconnaissance Using Virtual Prototyping

4.1 Introduction

Rapid and accurate evaluation approaches are essential for determining a building's structural integrity for future occupancy following a major seismic event. The elapsed time in the evaluation process could translate into private financial loss or even a public welfare crisis. Current inspection practices usually conform to the ATC-20 post-earthquake safety evaluation field manual and its addendum, which provide procedures and guidelines for making on-site evaluations (Rojah, 2005). Responders—such as ATC-20 trained inspectors, structural engineers, and other specialists—conduct visual inspections and designate affected buildings as green (apparently safe), yellow (limited

entry), or red (unsafe) for immediate occupancy (Chock, 2006). The assessment procedure can vary from minutes to days depending on the purpose of the evaluation (Vidal, et al., 2009). However, it has been pointed out by researchers (Tubbesing, 1989, Kamat and El-Tawil, 2007) that this approach is subjective and thus may sometimes suffer from misinterpretation, especially given that building inspectors do not have enough opportunities to conduct building safety assessments and verify their judgments, as earthquakes are infrequent.

Despite the de-facto national standard of the ATC-20 convention, researchers have been proposing quantitative measurements for a more effective and reliable assessment of structural hazards. Most of these approaches, especially non-contact, build on the premise that significant local structural damage manifests itself as translational displacement between consecutive floors, which is called interstory drift (Miranda, et al., 2006). The interstory drift ratio, which is the interstory drift divided by the height of the story, is a critical structural performance indicator that correlates the exterior deformation with the internal structural damage. The larger the ratio is, the higher the likelihood of damage. For example, a peak interstory drift ratio larger than 0.025 signals the possibility of a serious threat to human safety, and values larger than 0.06 translate to severe damage (Krishnan, 2006).

This research proposes a new approach for estimating IDR using an Augmented Reality (AR) -assisted non-contact method. AR superimposes computer-generated graphics on top of a real scene, and provides contextual information for decision-making purposes. AR has been shown to have several potential applications in the civil

infrastructure domain, such as inspection, supervision, and strategizing (Shin and Dunston, 2008). AR-assisted building damage detection is a specific type of inspection.

4.2 Review of Previous Work

So far the most commonly accepted approach for obtaining IDR is via contact methods, specifically the double integration of acceleration. This method is used most commonly because of its robustness and widespread availability in the world's seismically active regions. However, Skolnik and Wallace (2010) identified the vulnerability of double integration to nonlinear response. It has been suspected that sparse instrumentation or subjective choices of signal processing filters led to these problems.

Another school of obtaining IDR is non-contact methods. Wahbeh, et al. (2003) demonstrated a vision-based approach—tracking an LED reference system with a high fidelity camera. Ji (2010) instead applied feature markers as reference points for vision reconstruction. Similar target tracking vision-based approaches have also been studied in Hutchinson and Kuester (2004) and Lee and Shinozuka (2006). However, all of them require the pre-installation of a target panel or emitting light source, and such infrastructure is not widely available and is subject to damage during long-term maintenance, since it is located on the exterior of the structure. Fukuda, et al. (2010) tried to eliminate the use of target panels by using an object recognition algorithm, for instance orientation code matching. They performed comparison experiments by tracking a target panel and existing features on bridges, such as bolts, and achieved satisfactory agreement

between the two test sets. However, it is not clear whether this approach works in the scenario of monitoring a building's structure, as building surfaces are usually featureless.

Researchers also utilized terrestrial laser scanning technology in non-contact methods for continuous or periodic structural monitoring (Alba, et al., 2006) (Park, et al., 2007). In spite of the high accuracy of such systems, the equipment volume and the large collected dataset put these methods at a disadvantage for rapid evaluation scenarios.

Kamat and El-Tawil (2007) first proposed the approach of projecting the previously stored building baseline onto the real structure, and using a quantitative method to count the pixel offset between the augmented baseline and the building edge. In spite of the stability of this approach, which has been tested at the University of Michigan's Structural Engineering Laboratory with large-scale shear walls, it required a carefully aligned perpendicular line of sight from the camera to the wall for pixel counting. Such orthogonal alignment becomes unrealistic for high-rise buildings, since it demands the camera and the wall be at the same height.

(Dai, et al., 2011) removed the premise of orthogonality using a photogrammetry-assisted quantification method, which established a projection relation between 2D photo images and the 3D object space. They validated this approach with experiments that were conducted with a two-story reconfigurable aluminum building frame whose edge could be shifted by displacing the connecting bolts. The experimental results were in favor of the adoption of consumer-grade digital cameras and photogrammetry-assisted concepts. However, the issue of automatic edge detection and the feasibility of deploying such a method at large scales, for example with high-rise buildings, have not been addressed.

This chapter specifically addresses the above limitations and proposes a new algorithm called line segment detector for automating edge extraction, as well as a new computational framework automating the damage detection procedure. In order to verify the approach's effectiveness, a synthetic Virtual Prototyping (VP) environment has been designed to profile the detection algorithm's sensitivity to errors inherent in the used tracking devices.

4.3 Overview of Reconnaissance Methodology

Figure 4.1 shows the schematic overview of measuring earthquake-induced damage manifested as a detectable drift in a building's façade. The previously stored building information is retrieved and superimposed as a baseline wireframe image on the real building structure after the damage. Then the sustained damage can be evaluated by comparing the key differences between the augmented baseline and the actual drifting building edge. Figure 4.1 also demonstrates a hardware prototype called ARMOR (Dong and Kamat, 2010) on which the developed application can potentially be deployed. The inspector wears a GPS antenna and an RTK (acronym for Real Time Kinematic) radio that communicates with the RTK base station. Together they can track the inspector's position up to centimeter-accuracy level. As is discussed in Section 4.5.2, position and orientation tracking accuracy greatly influence the effectiveness of the estimation algorithm. Meanwhile, the estimation procedure and the final results can be shown in the HMD (acronym for Head Mounted Display) in front of the inspector.



Figure 4.1 Schematic overview of the proposed AR-assisted assessment methodology.

The evaluation procedure is further illustrated in Figure 4.2. The first step is for the camera to take pictures of the building. The orientation and location information about the camera needs to be recorded for 3D to 2D projection, as well as for 2D to 3D triangulation. The second step is to extract edges in the captured photo frames. A line segment detector extracts the vertical building edge, and an estimation method is used to represent the horizontal edge with the baseline. The last step involves the triangulation of the 3D coordinate at the key location from multiple corresponding 2D intersections between the vertical and horizontal edges. IDR is subsequently computed by comparing the key difference between two consecutive building floors divided by the story height. The accuracy of IDR calculation thus depends on the accuracy of internal and external camera parameters, the accurate detection of the vertical edge, and the estimation of the horizontal edge.

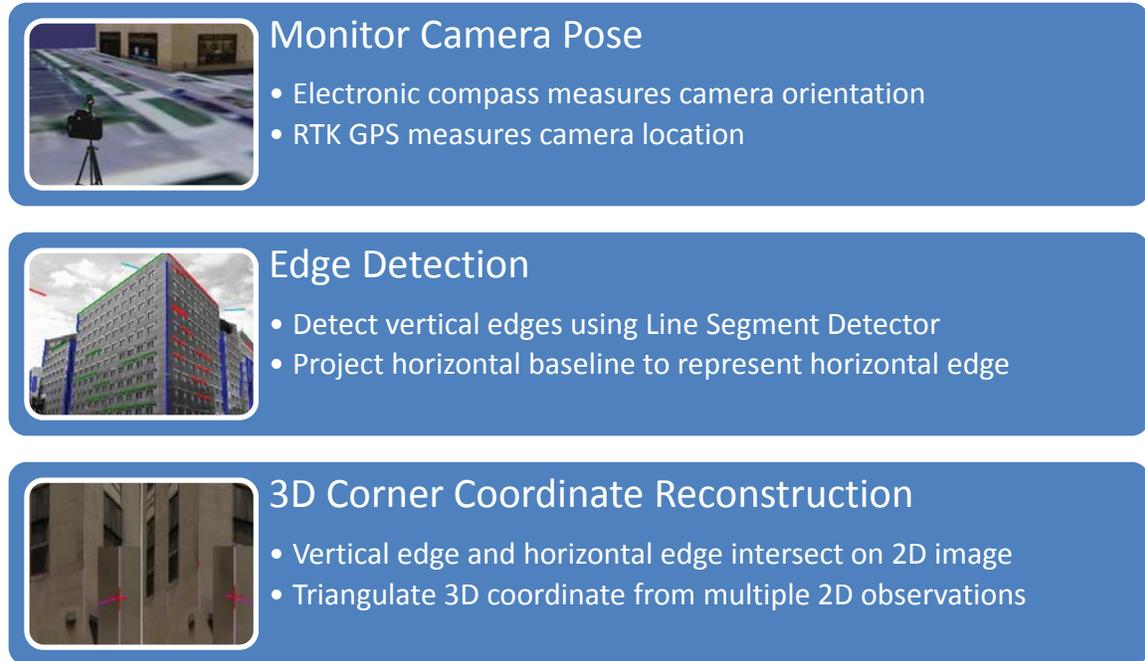


Figure 4.2 Major steps to reconstruct the 3D coordinates of key locations on the building.

Besides being a quantitative means of providing reliable damage estimation results, the vertical baseline of the building structure is also a qualitative alternative for visual inspection of local damage. By observing the graphical discrepancy between the vertical baseline and the real building edge, the on-site reconnaissance team can approximately, but quickly, assess how severe the local damage is in the neighborhood of the visual field. In other words, the larger the graphical discrepancy is, the more severe the damage. Figure 4.3 (a) and (b) focus on different key locations of the building, but they are taken from the same angle (i.e., direction). The right-bottom window on each image is a zoom-in view of the key location. The two vertical lines in the zoom-in window represent the detected edge and the vertical baseline, respectively. The fact that the gap between the detected edge and the vertical baseline on Figure 4.3 (a) is smaller than that on Figure 4.3

(b), indicates that the key location on Figure 4.3 (b) suffered more local damage than that on Figure 4.3 (a).

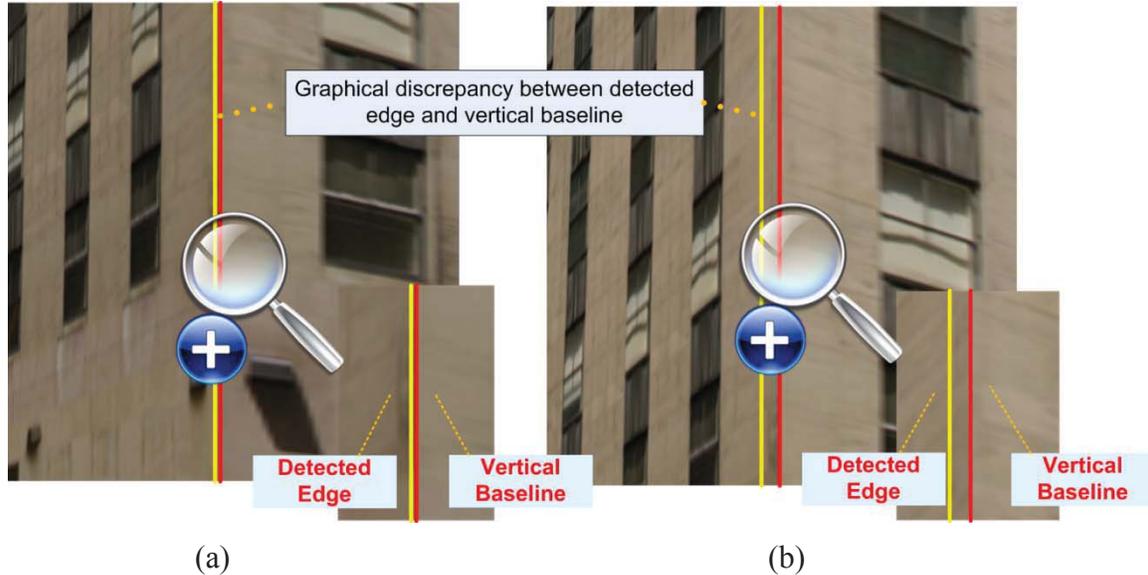


Figure 4.3 Graphical discrepancy between the vertical baseline and the detected building edge provides hints about the magnitude of the local damage.

4.4 Technical Approach

The objective of this research was to design, demonstrate, and evaluate a new AR-assisted non-contact method for rapidly estimating the IDR in buildings that manifest residual drift from seismic damage. In particular, the research objectives included the verification of the developed algorithms, and the evaluation of the sensitivity of computed drift to measurement errors inherent in the used tracking devices. Access to a damaged high-rise building is rare. Moreover, such a test bed offers no possibility of inducing specific amounts of drift in the building stories for calibration or evaluation purposes.

In addition, an experimental plan conducted to understand the designed algorithm's sensitivity to ambient conditions and instrument uncertainty requires a controlled test bed environment. In order to demonstrate and evaluate the developed computational framework, this research designed a synthetic 3D environment based on Virtual Prototyping (VP) principles for verifying the developed algorithms, and for conducting the sensitivity analysis. A Virtual Prototype, or digital mock-up, can be defined as a computer simulation of a physical counterpart that can be observed, analyzed, and tested from life-cycle perspectives, such as design and service, as if it were a real physical model. The creation and evaluation of such a Virtual Prototype is known as Virtual Prototyping (VP) (Wang, et al., 2002). By using a digital model instead of a physical prototype, VP can alleviate several shortcomings in the design and evaluation process.

Virtual Reality (VR) is a related concept and is specifically defined as a computer simulation of a real or imaginary system that enables a user to perform operations on the simulated system, and shows the effects in real time (American Heritage Dictionary, 2009). VR is thus of significant value to VP because it can facilitate the visual understanding of a virtual product during the design and evaluation process (Jayaram, et al., 1998). In effect, VR can support the analysis required for demonstrating and evaluating a proposed design by offering the possibility of immersing end-users in the virtual environment to perform specific tasks (Bauer, et al., 1998). VP using VR principles thus emerged as a clear choice for demonstrating and evaluating the proposed computational framework.

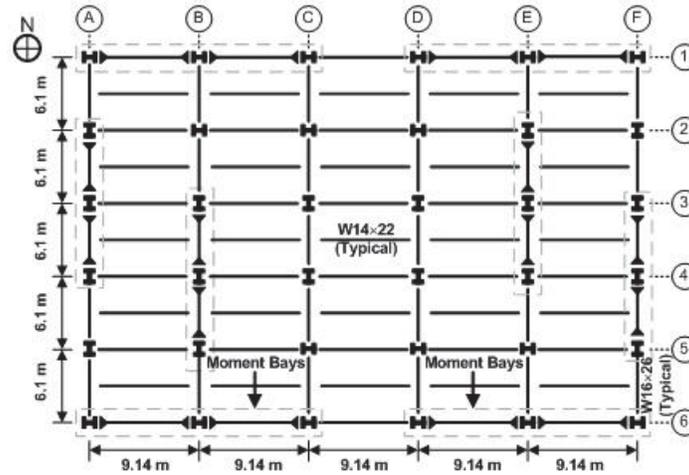
4.4.1 Reconfigurable Virtual Prototype of Seismically Damaged Building

The simulated VP environment contains a ten-story graphical and structural building model constructed as plans, as shown in Figure 4.4. The graphical model is entirely reconfigurable and capable of manifesting any level of internal damage on its façade in the form of residual drift, so that the IDR can be extrapolated for each floor. Given the input IDR, the structural macro model predicts the potential for structural collapse and the mode of collapse, should failure occur. The remainder of this chapter focuses on the graphical model behavior and the underlying algorithm of extrapolating the IDR.

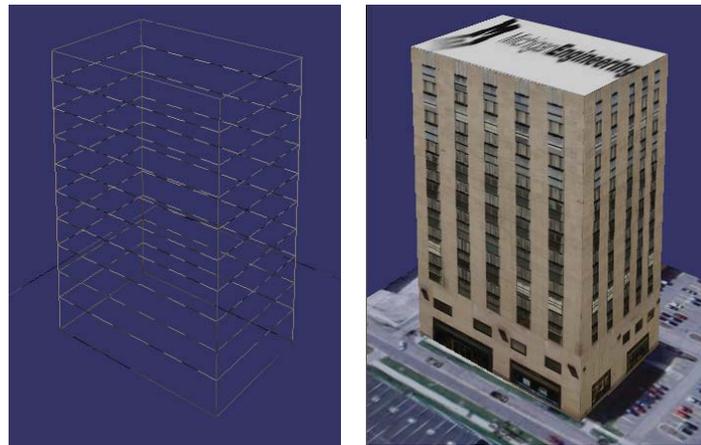
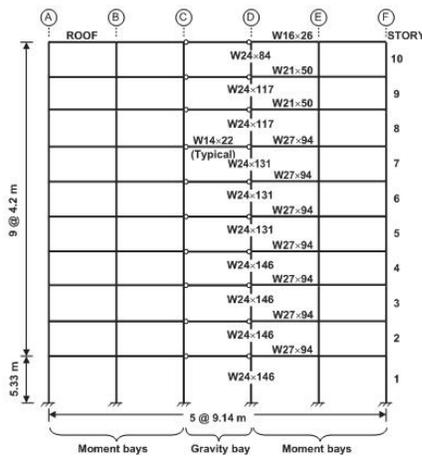
The residual drift is represented by translating the joints of the wireframe model that have been superimposed with a high-resolution façade texture. The drift is further manifested through the displaced edges on the surface texture that can be extracted using a Line Segment Detector (LSD). Subsequently, the 2D intersections between extracted edges and projected horizontal baselines are used to triangulate the 3D spatial coordinates at key locations on the building.

The 2D image, where extracted edges and baselines are visible, is taken by the OpenGL camera that is set up at specified corners in the vicinity of the building (Figure 4. 5). At each corner, the camera's orientation (i.e., pitch) is adjusted to take a snapshot of each floor in sequence, and project the corresponding horizontal baseline. In reality, the location of the camera may be tracked by a Real-Time Kinematic GPS (RTK-GPS) sensor, and its orientation may be monitored with an electronic compass. In the simulated VP environment, the location and orientation of the camera are known and can be

controlled via its software interface. Random errors can thus be introduced to simulate the effects of systemic tracking uncertainty or jitter expected in a field implementation.



(a) Plan View



(b) Elevation View

Figure 4.4 A ten-story graphical building model is constructed as its macro model counterpart.

4.4.2 Damage Modeling

The drift with uniform distribution is applied on each joint of a building to imitate the structural damage sustained after the disaster. The damage model is justified by the

following statistics. The requirement on inelastic IDR is commonly limited within 2.5% by building codes, and it is occasionally relaxed to 3% for tall buildings (Hart, 2008). Given that the height of a building story is, on average, 3m~4m, the maximum allowable displacement between two consecutive floors is 0.09m~0.12m when using the most relaxed IDR of 3%. The drift of the corner's position is modeled as uniform distribution in both the X and Y directions. The distribution interval is limited within $[-0.06\text{m}, 0.06\text{m}]$, so that the difference between consecutive floors in either the X or Y direction is less than 0.12m. In the experiment, a reasonable assumption is made that, unless the internal columns buckle or collapse, the height of the building remains the same after the damage. Since the column buckling or collapse situation is not modeled in the simulation, the Z value of the corner coordinate does not change.

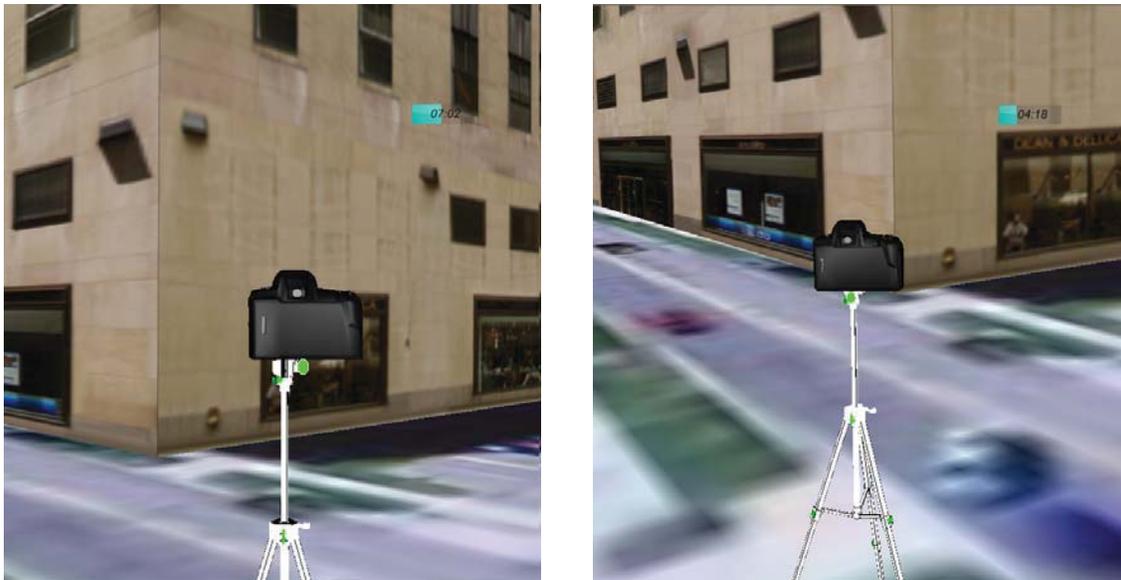


Figure 4.5 The OpenGL camera replicates a physical camera to take pictures of the key locations.

A high-resolution façade texture was acquired for the modeled building from the Google Warehouse (Sardinas, 2010). The texture is taken by a physical camera and

rectified into orthogonal perspective so that it can be superimposed directly on the façade of the wireframe model. Each polygon vertex is assigned a 2D texture coordinate, and the associated clipped texture is pasted onto the surface of the wall (Figure 4.6). The texture can thus displace with the drifting vertex in the 3D space, with the goal of estimating the vertex deformation through the displaced texture.

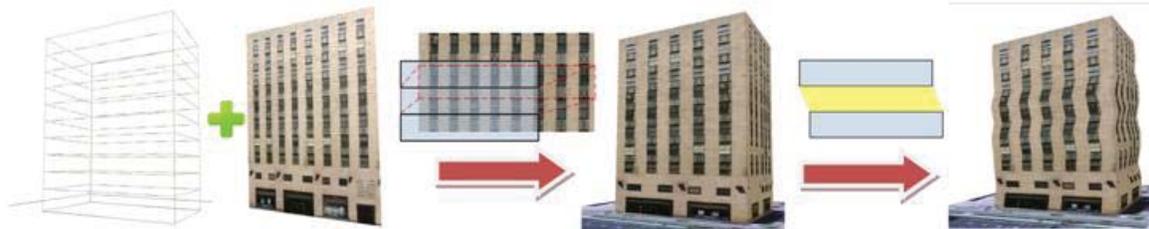


Figure 4.6 Internal structural damage, shift of the vertex, is expressed through the displacement of the texture.

4.4.3 Camera Modeling

In order to achieve parity with a practical field implementation, the OpenGL camera in the simulated environment is configured with the specifics of a real digital SLR (Single Lens Reflex) camera that may be used by an inspector to take pictures of a real building. This section describes how the external and internal parameters of the OpenGL camera were modeled to achieve such parity.

4.4.3.1 External Parameters

The external parameters describe the position and orientation of the camera in the world coordinate system. In the OpenGL environment, the origin and unit of the world coordinate is arbitrarily specified by the programmer, and the pose of the camera can be known and error-free. However in the real field application, the position and orientation

of the camera may be tracked by GPS and 3D electronic compass, respectively, whose measurements are subject to instrument uncertainty.

The Laboratory for Interactive Visualization in Engineering (LIVE) is equipped with an RTK (Real Time Kinematics) GPS with a manufacturer-specified accuracy of 2.5 cm + 2 ppm RMS (Root Mean Square) horizontal, and 3.7 cm + 2 ppm vertical (Trimble, 2007). The parts per million (ppm) error is dependent on the distance between the base and rover receiver. For example, if the distance is 10km, a 2ppm error equals 20mm. Once warmed up, the RTK-GPS yields a measurement reading in seconds. Better accuracy can be achieved with higher-ranking RTK equipment and no significant compromise on collecting time. For example, manufacturers report 3mm + 0.1ppm RMS horizontal accuracy with the Fast GNSS survey (Trimble, 2009).

The 3-axis digital compass used in LIVE for outdoor angular measurements measures yaw, pitch, and roll with a resolution of 0.01° as the manufacturer-specified accuracy. The static accuracy for 3 axes is 0.3° (RMS) when the tilting (i.e., pitch and roll) is smaller than 65° . The accuracy is slightly compromised when the tilting range goes beyond 65° (PNI, 2009).

In order to verify the developed algorithms and computational framework in ideal conditions, the simulated experiments are first performed with a ground true position and orientation readings (i.e., assuming perfect tracking of the camera's position and orientation). Subsequently, to investigate the practicality of the method in the field implementations, the same experiments are conducted with the introduction of the instrument uncertainty in a certain range.

4.4.3.2 Internal Parameters

In the OpenGL camera, the internal parameters can be represented by left, right, top, bottom, near, and far plane values, which form a viewing frustum (Figure 4.7). The physical counterpart of the near plane inside the digital camera is the sensor chip. Given the sensor chip parameters of the mainstream off-the-shelf camera, and without a loss of generality, the left and right values are set to $\pm 11.15\text{mm}$, and the bottom and top values to $\pm 7.45\text{mm}$ (The Digital Picture, 2012). The focal length of the camera is approximately equivalent to the near plane distance, and can be flexibly adjusted from 20mm to 200mm. The lens with a similar range is off-the-shelf available and economically affordable. In the simulated experiments, usually the maximum focal length is selected for best performance.

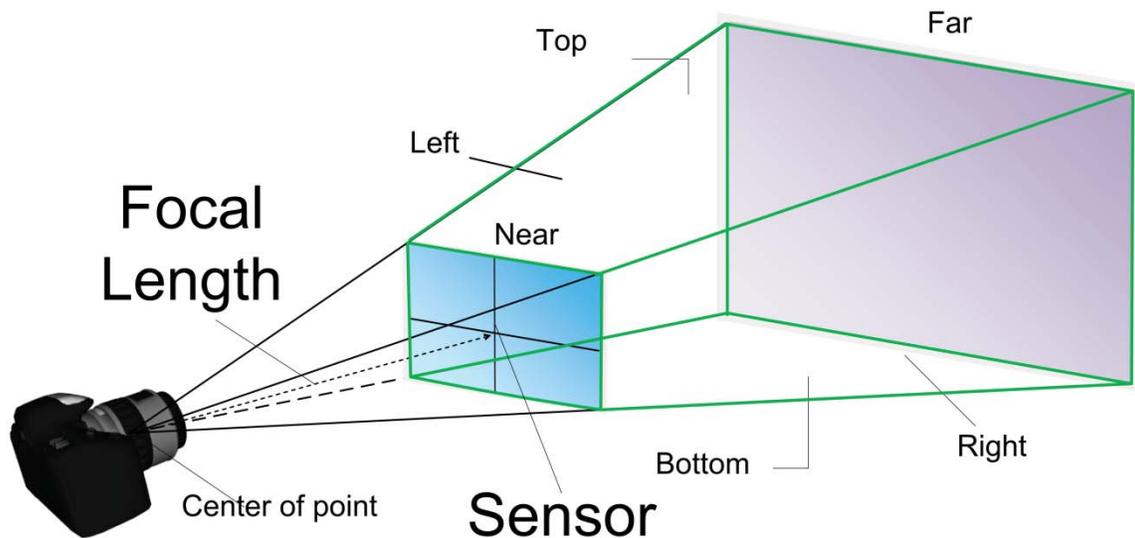


Figure 4.7 The near plane of the OpenGL camera is the counterpart of the physical camera image sensor chip, a device that converts an optical image into an electronic signal.

Theoretically, the internal parameter errors are consistent due to being induced by the imperfection of the camera lens and its internal mechanics. There are two major elements

comprising the induced camera's systematical error (i.e. the lens distortion): (1) an aggregate of the radial distortion, and the decentering distortion, and (2) the approximation of the focal length distance. Unlike the external errors that can be dynamically affected by environmental variables—like visibility of the sky and dynamic magnetic field—the internal errors are relatively stable and can be systematically compensated, beforehand, by camera calibration (Dai, et al., 2011).

4.4.4 Vertical Edge Detection

Vertical Edge detection of the building wall is the most critical step for locating the key point on the 2D image plane, which happens to be a fundamental problem in image processing and computer vision domains, as well. Many algorithms for edge detection exist and most of them use the Canny Edge Detector (Canny, 1986) and Hough Transformation (Duda and Hart, 1972) as a benchmark. However, standard algorithms are subject to two main limitations. First, they face threshold dependency. Edge detection algorithms contain a number of adjustable parameters that influence their effectiveness. The tuning of parameters can yield significant overhead for on-site reconnaissance inspectors, and can compromise assessment efficiency and detection accuracy. Second, standard algorithms face false positives and negatives. They either detect too many irrelevant small line segments or fail to interpret the desirable line segments. False positives and negatives are highly related to the threshold tuning.

4.4.4.1 Active Contour Approach

The author's first attempt was to apply the Graph-Cut Based Active Contour (GCBAC) algorithm developed by (Xu and Bansal, 2003). Traditional active contour is an energy-

minimizing spline guided by external forces and influenced by image forces (Terzopoulos, et al., 1988). By introducing the concept of contour neighborhood, GCBAC alleviates the local minima trapping problem suffered by traditional active contour—the energy-minimizing spline could be trapped by objects in their neighborhood with higher gradient zones, which means instead of detecting edge with global minimized energy, the edge with local minimized energy turns out to be the converged result (detected edge).

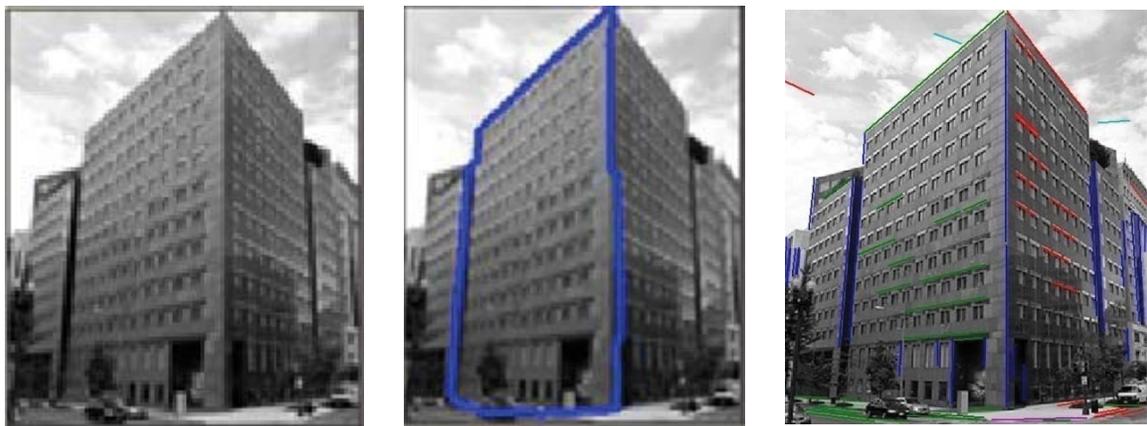


Figure 4.8 LSD (right) outperforms GCBAC (middle) when searching for localized line segments, for example building edges.

GCBAC requires manual specification of the initial contour and contour neighborhood width, quantities that are arbitrary and subjective. However, optimization can be achieved by using the original baseline of the damaged building to numerically calculate both the initial contour and neighborhood width. GCBAC works best when the image covers the entire outline of the building that is not applicable in the real applications (Figure 4.8). Unfortunately, the coverage of the entire high-rise building surface inevitably results in lower-resolution details. Moreover, frequent partial occlusion from trees and other buildings can compromise detection accuracy.

4.4.4.2 Line Segment Detection Approach

The second attempt was a linear-time line segment detector that gives accurate results, a controlled number of false detections, and—most importantly—requires no parameter tuning (von Gioi, et al., 2010). LSD combines the advantages of (Burns, et al., 1986) and (Moisan and Morel, 2000) methods, and gracefully overcomes their drawbacks. The Burns' algorithm innovatively ignores gradient magnitudes and uses only gradient, which yields a well-localized result. It is linear-time but subject to the threshold problem. The threshold question was thoroughly studied in Desolneux's algorithm. It is based on a general perception principle that an observed geometric structure is perceptually meaningful when its expectation in noise is less than one. The principle guarantees the lack of false positives and no false negative. Unfortunately, the method is exhaustive and has an $O(N^4)$ complexity. Consequently, the innovative combination of these two approaches is a linear-time LSD that requires no parameter tuning and gives accurate results.

LSD outperforms GCBAC in searching for localized line segments. However, there are still multiple line segment candidates in the neighborhood of the actual edge of the building wall (Figure 4.9). A filter is used to eliminate those line segments whose slope and boundary deviate significantly from the original baseline. In initial attempts, the author proposed fully automating the edge detection procedure by choosing the detected line with the closest distance to the original baseline. It will be shown that this approach is problematic and was thus found to be unfeasible. Manual selection, on the other hand, was identified to be much more accurate and can be completed in a few seconds. If the

LSD fails to locate a desirable edge, the user can manually transpose the closest line segment to the desirable position in a short amount of time.



Figure 4.9 A geometric filter plus minimal manual reinforcement can rapidly eliminate most irrelevant line segments.

4.4.5 Corner Detection

4.4.5.1 Horizontal Edge Detection

Besides the vertical edge detection, the horizontal edge detection also plays an essential role in deciding the 2D coordinate of the drifting corner. If the horizontal frames of windows roughly match with the physical floors separating stories, then the horizontal edge can also be graphically detected by LSD as windows' bottom frames (Figure 4.9). However, since such an assumption is not universally true, we choose to numerically project the horizontal baseline that physically separates stories on the damaged building surface to represent the horizontal edge. Such an approach is more generic than the graphical detection. However since a floor is allowed to drift within the XY plane, its horizontal baseline has to be shifted accordingly before it is projected onto the 2D image so as to match the real horizontal edge. If the 2D projected horizontal edge does not strictly match with the real horizontal edge, then a gap is detectable, and it enlarges as the

camera moves closer to the building. Furthermore, the drift between the horizontal baseline and the horizontal edge contains both parallel and perpendicular components (in the x and y directions), and the detectable gap is caused exclusively by the perpendicular component. The drift on the z coordinate is not considered here since internal column buckling or collapse is not considered in the damage model.

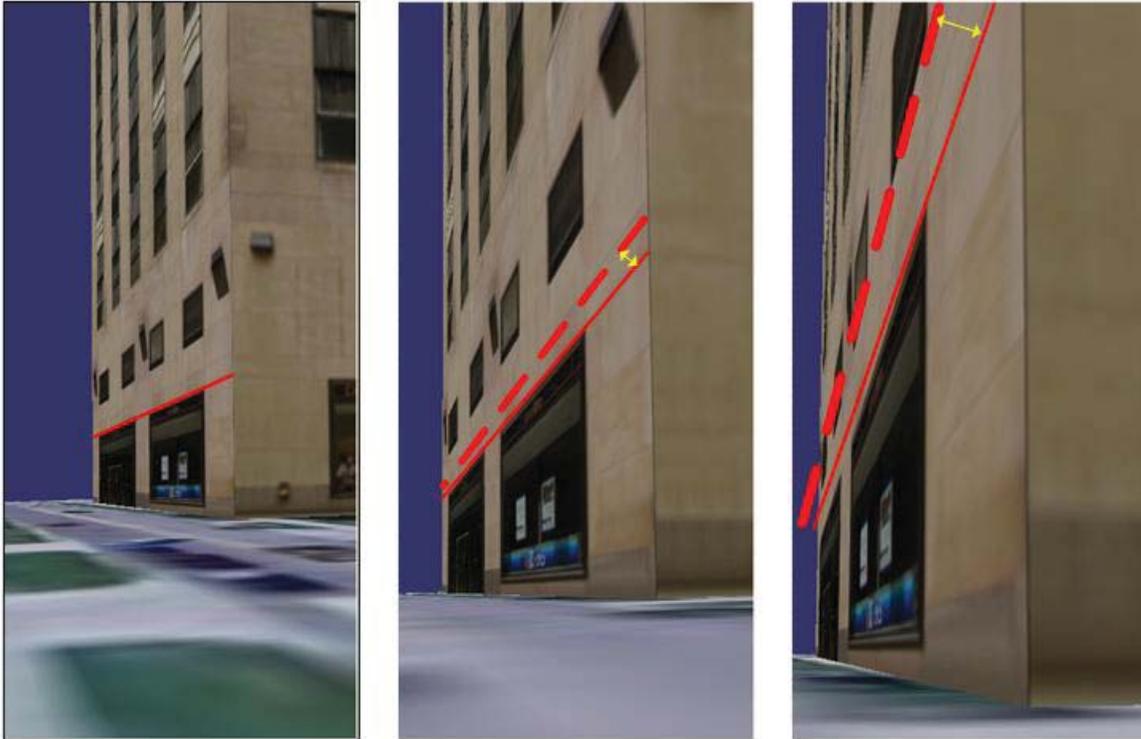


Figure 4.10 The detectable gap between the original baseline and real edge enlarges as the camera gets closer to the building.

Unless the drift is known, it is impractical to deterministically position the edge in the XY plane. Therefore, the proposed solution is to exhaustively test all possible drifting configurations, with a computation complexity of $\Theta(N^4)$. This happens because iterating through all the possible shift configurations of two endpoints on one line segment costs $\Theta(N^2)$, given that only the perpendicular drift component between the edge and the baseline is considered. The union of two line segments needed in the triangulation has $\Theta(N^4)$

complexity, (Figure 4.11 (a)) where N is equal to the uniform distribution interval divided by the estimation step. For example, if the uniform distribution interval is $[-0.6, 0.6]$, and the joint position is shifted from -0.6 to 0.6 by 0.1 at one step, then N is equal to 12.

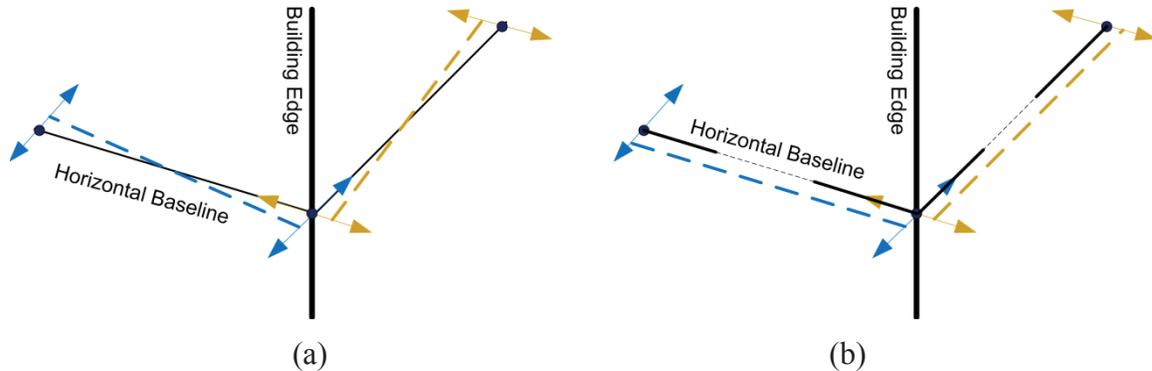


Figure 4.11 Alignment of Horizontal Baseline: a) shifts the two ends of the baseline with different distances and costs $\Theta(N^4)$, while (b) shifts the two ends of the baseline with the same distance and costs $\Theta(N^2)$.

Furthermore, a simple approximation can reduce the complexity from $\Theta(N^4)$ to $\Theta(N^2)$ without compromising accuracy. Since the intersection between the baseline and the edge is close to one endpoint of the line segment, only the (x,y) of that endpoint dominates the intersection accuracy, and the impact of the other end diminishes significantly given the ratio of the drift magnitude over the distance between two endpoints. Therefore the two points on one line segment can share the same tested shifting value with $\Theta(N)$ complexity, and subsequently the complexity of two line segments decreases to $\Theta(N^2)$ (Figure 4.11 (b)).

There are two edges on the adjacent walls intersecting with the edge, and the one with the lower slope (absolute value) should be chosen for calculating the 2D intersection. This is because how close the projected baseline is to the actual edge on the 2D image is

not only affected by its 3D coordinate, but also the perspective projection. For example, imagine the camera is initially placed at an infinite point; regardless of the displacement between the horizontal baseline and the building edge, their projections overlap on the 2D image. Then the camera is moved toward the building and eventually placed beneath the baseline. In this case, if the camera looks straight up, the gap between their projections on the 2D image is equal to the displacement in the 3D space. Figure 4.10 backs up this observation. In general, the lower the slope of the edge, the less the gap between the baseline and edge projected on the 2D image. Additionally, if only one side of the building is covered in the image, the edge on the visible side is chosen (Figure 4.13 (c,d)).

4.4.5.2 Corner Detection Algorithm

The next challenge is to select the best estimation from the N^2 candidates in the aforementioned iteration test. Each pair of tested shift $(\Delta x, \Delta y)$ of the baselines corresponds to an estimated 3D corner position (x', y', z') . If the actual 3D corner position is (x, y, z) , and if the height of the building remains the same after the damage, an intuitive judgment for the confidence of the estimation is $\min(z-z')$.

A better judgment also takes (x', y') into account. Say the original 3D corner position is (x_0, y_0, z_0) , a proper tested shift $(\Delta x, \Delta y)$ should be close to the estimated shift $(x'-x_0, y'-y_0)$. In other words, $(x'-x_0-\Delta x, y'-y_0-\Delta y)$ should be minimized.

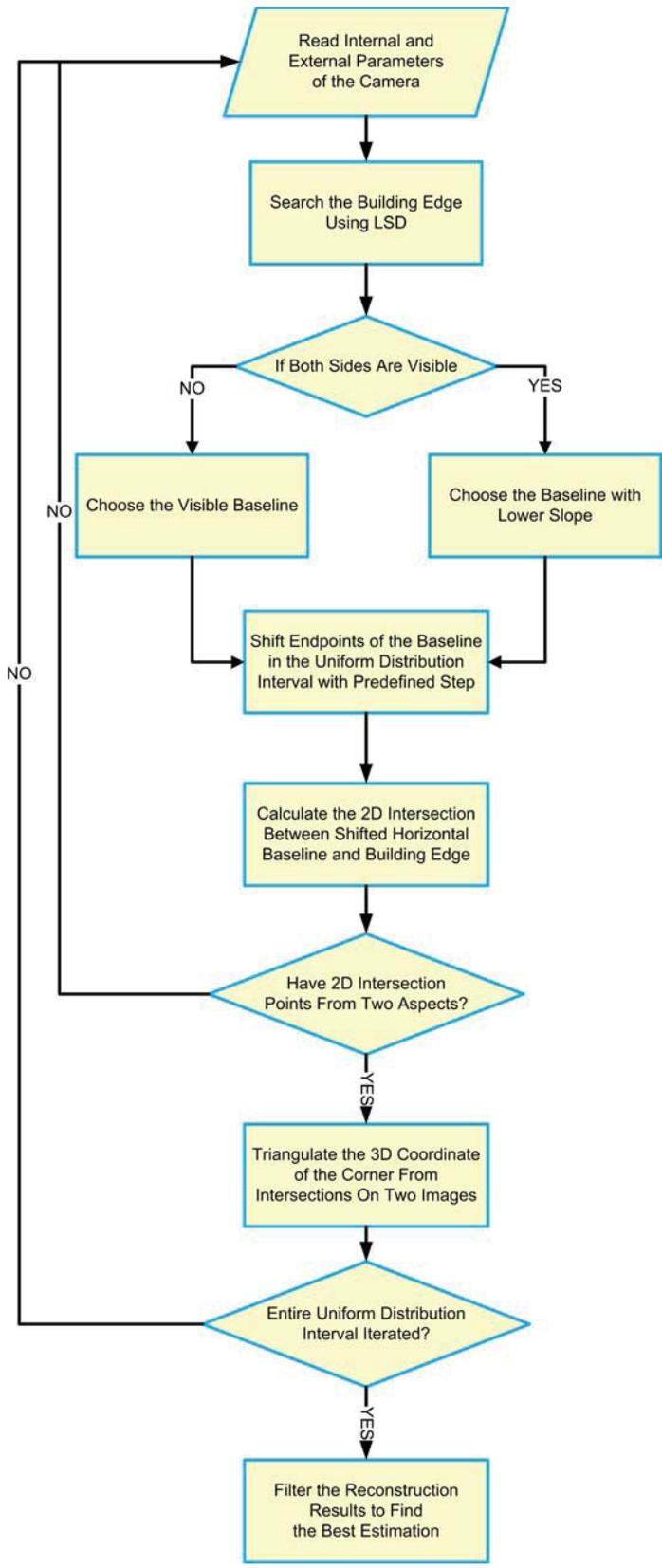


Figure 4.12 Flowchart of the proposed corner detection algorithm.

Based on the hypothesis above, there are two filters proposed for selecting the estimated corner coordinate. The first one minimizes the square root of $(x'-x_0-\Delta x, y'-y_0-\Delta y, z'-z_0)$. The second one sets thresholds for $(x'-x_0-\Delta x, y'-y_0-\Delta y, z'-z_0)$, and selects the one with the smallest $(|\Delta x|, |\Delta y|)$ among the filtering results. Based on the experiment results, there is no major performance gain of one over the other. The algorithm is described as a flow chart in Figure 4.12.

4.4.5.3 Interstory Drift Ratio Calculation

Once the corner's position is estimated, the calculation of the Interstory Drift Ratio for each story is straightforward given its definition, which is the interstory drift divided by the height of that story. For example, the right image in Figure 4.13 shows a zoom-in view of a certain story. The IDR of that story by the side facing the reader can be calculated as $(P_1 - P_2)/h$. The horizontal movement of the floor relative to the ceiling is denoted as $P_1 - P_2$, and h is the height of the story.

4.5 Evaluation of Experimental Results

In order to understand the best performance that the computational framework can achieve in the ideal situation, this section starts with studying algorithm performance in a series of controlled comparison experiments with ground true camera tracking data. Later on, the experiment is extended to situations where instrumental errors are included to profile algorithm sensitivity.

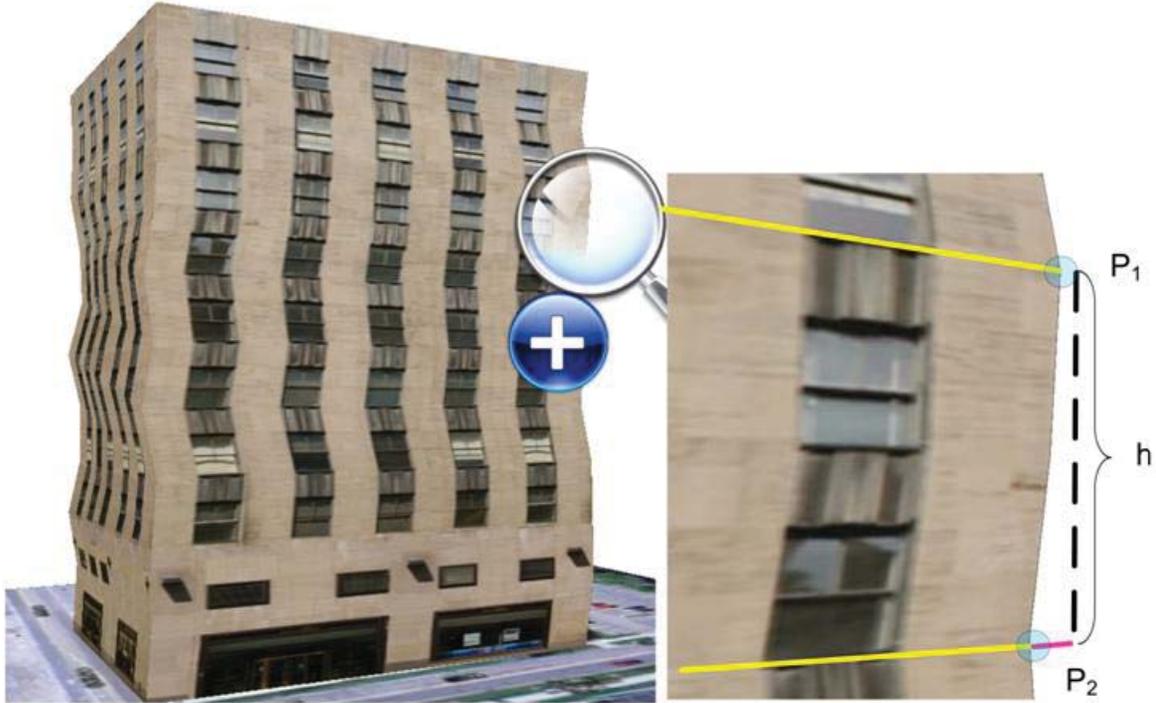


Figure 4.13 Interstory Drift Ratio Calculation.

4.5.1 Experiment with Ground True Location and Orientation

The goal of this subsection is to test the best performance that the algorithm can achieve with ground true camera pose tracking data. Even given the ground true tracking data, the estimation accuracy can still be affected by many factors. Therefore a series of comparison experiments are conducted to find the influence magnitude of each factor. For each group of comparisons, the statistics show the average, standard deviation, and maximum of the square root of the x , y coordinate error. The minimum—generally smaller than 1mm—is not included because it is not essential in judging the accuracy. There are 10 stories and four building edges, and thus 40 corner coordinate samples are included in each experiment group.

4.5.1.1 Observing Distance

The purpose of this set of experiments is to understand the impact of observing distance on estimation accuracy. The observing distance is the projection of the vector between the camera and building corner on the XY plane. Three experiments are conducted with the same internal camera parameters (6 Mega Pixels), damage model ($\pm 0.04\text{m}$ shifting range), and 0.01m estimation step, except that the camera is moving farther away from the building.

Table 4.1 - Sensitivity of drift error to observation distance.

Average Distance	10m	20m	35m
Ave Error	0.0079m	0.0065m	0.0048m
StdDev	0.0047m	0.0038m	0.0029m
Max Error	0.0165m	0.0133m	0.0126m

As evidenced in Table 4.1, the accuracy improves when the camera moves away from the building. As mentioned in Section 4.4.5.1, in general, the lower the slope, the less the projected error, because increasing distance helps to approximate orthogonal perspective. Therefore, increasing the distance of the camera from the building has the effect of lowering the slope and attenuating the error. Approximate orthogonal perspective can also be achieved if the camera is at the same height as the horizontal baseline. This is supported by the fact that the estimation error is always insignificant for the first floor, where the height of the floor is similar to that of the camera, and the slope is close to zero.

4.5.1.2 Observing Angle

This experiment tries to understand whether the observing angle could affect the accuracy. The observing angle is formed by the line of sight between two cameras. In the first group, two images from two perspectives cover both sides of the building wall (Figure 4.14 (a,b)). In this case, the observing angle is closer to a right angle. In the second group, one image covers both sides, while the other one covers only one side; in the third group, both images cover only one side of the building wall (Figure 4. 14 (c,d)). In the latter two cases, the observing angle is closer to 180° . All of the other environmental parameters are controlled as follows: 6 Mega Pixels, ± 0.04 shifting range, 35m observing distance, and 0.01m estimation step.

Table 4.2 - Sensitivity of drift error to observing angle.

	Both cover two sides	One covers two sides	Both cover one side
Ave Error	0.0048m	0.0111m	0.0238m
StdDev	0.0029m	0.0084m	0.0141m
Max Error	0.0126m	0.0287m	0.0525m

It has been shown in Table 4.2 that the accuracy degenerates significantly when covering only one side of the wall. This indicates that the detection error is minimized when the angle formed by two lines is close to a right angle, and magnified when the angle is either acute or obtuse.

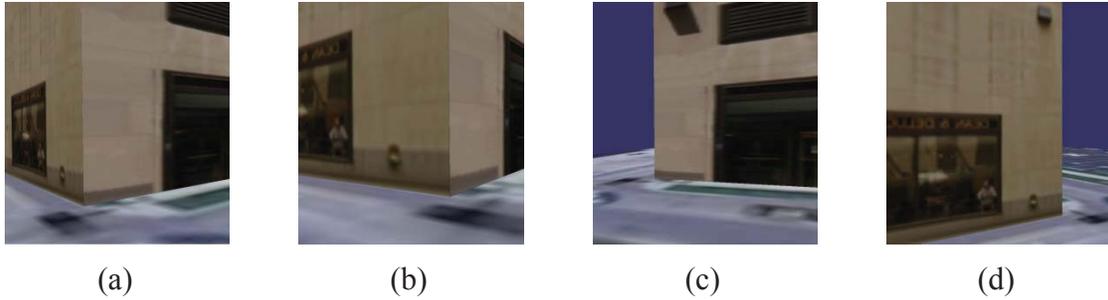


Figure 4.14 Observing angle of the camera.

4.5.1.3 Drift Interval

Here we want to understand whether the estimation accuracy could be affected by the assumed uniform distribution interval. The three tested intervals are $\pm 0.04\text{m}$, $\pm 0.05\text{m}$, and $\pm 0.06\text{m}$. The camera distance is fixed at 35m, the image covers both sides of the building wall with 6 Mega Pixels, and the estimation step is 0.01m.

Table 4.3 - Sensitivity of drift error to drift interval.

Drifting Range	[-0.04m, 0.04m]	[-0.05m, 0.05m]	[-0.06m, 0.06m]
Ave Error	0.0048m	0.0046m	0.0049m
StdDev	0.0029m	0.0034m	0.0041m
Max Error	0.0126m	0.0159m	0.0153m

As evidenced in Table 4.3, the increase of the drifting range slightly deteriorates the accuracy, especially for standard deviation. The increasing difficulty of estimating the larger gap between the baseline and floor outline is probably responsible for the degeneration of the accuracy. However, as will be shown in Section 4.5.1.6, the increase of image resolution can mitigate accuracy loss.

4.5.1.4 Approximate Versus Accurate

The author claimed in Section 4.4.5.1 that the approximate mode can achieve the same accuracy level as the detailed iterative mode, but with less computational expense. This is supported by the following experiment group. The chosen test bed is the same one as in 5.1.3. The first group uses the approximated method of $\Theta(N^2)$ complexity, and the second one uses the accurate method of $\Theta(N^4)$ complexity.

Table 4.4 - Sensitivity of drift error to accurate and approximate estimation modes.

$\Theta(N^2)$ $\Theta(N^4)$	[-0.04m, 0.04m]		[-0.05m, 0.05m]		[-0.06m, 0.06m]	
Ave Error	0.0048m	0.0056m	0.0046m	0.0051m	0.0049m	0.0053m
StdDev	0.0029m	0.0037m	0.0034m	0.0036m	0.0041m	0.0039m
Max Error	0.0126m	0.0149m	0.0159m	0.0161m	0.0153m	0.0153m

Table 4.4 shows that, even though the average error of $\Theta(N^2)$ is slightly smaller than the average error of $\Theta(N^4)$, the standard deviation and maximum error have more or less the same accuracy level. Therefore, $\Theta(N^2)$ is a good approximation of $\Theta(N^4)$ without accuracy loss, and with a significant gain in computational time.

4.5.1.5 Estimation Step

As mentioned in Section 4.4.5.1, the computation cost is decided by the uniform interval and the discrete estimation step. This experiment group tries to identify the optimal estimation step. The controlled experiment condition with $\pm 0.04m$ interval in 5.1.3 is chosen as a benchmark here. The compared estimation steps are 0.01m, 0.005m, and 0.0025m.

As evidenced by Table 4.5, smaller intervals can hardly increase the accuracy, therefore the 0.01m interval is recommended for field applications.

Table 4.5 - Sensitivity of drift error to estimation step.

Estimation Step	0.01m	0.005m	0.0025m
Ave Error	0.0048m	0.0048m	0.0048m
StdDev	0.0029m	0.0029m	0.0029m
Max Error	0.0126m	0.0120m	0.0120m

4.5.1.6 Image Resolution

Image resolution is one of the most important factors of characterizing a camera. Unfortunately, the resolution of the OpenGL camera is limited by that of the monitor (1900 x 1200) of the Dell Precision M60 laptop that was used in this research. An alternative is to use a telephoto lens. This can be achieved in OpenGL by pushing the near plane farther away without changing its size. For example, pushing the near plane two times away is equivalent to magnifying the resolution by a factor of four.

Table 4.6 - Sensitivity of drift error to image resolution.

Resolution	6 Mega Pixels	10 Mega Pixels	18 Mega Pixels
Ave Error	0.0048m	0.0026m	0.0019m
StdDev	0.0029m	0.0016m	0.0011m
Max Error	0.0126m	0.0066m	0.0055m

Table 4.6 indicates that a higher image resolution apparently helps promote the accuracy of line segment detection, which in turn increases the overall accuracy. Given the statistics from Sections 4.5.1.4 and 4.5.1.5, where no improvement is observed, it can

be concluded that the accuracy of line segment detection is the bottleneck of the algorithm given ground true tracking data.

4.5.1.7 Automatic versus Manual

As mentioned in Section 4.4.4.2, the automatic detection filters the line segments by their distance to the original vertical baseline. The closest one is preserved. This experiment demonstrates that such a heuristic is problematic. Similar experimental conditions to the ones in 5.1.3 are chosen as a test bench.

Table 4.7 - Sensitivity of drift error to manual and automatic detection modes.

Auto Manual	[-0.04m, 0.04m]		[-0.05m, 0.05m]		[-0.06m, 0.06m]	
Ave Error	0.0109m	0.0048m	0.0199m	0.0046m	0.0144m	0.0049m
StdDev	0.0119m	0.0029m	0.0193m	0.0034m	0.0140m	0.0041m
Max Error	0.0583m	0.0126m	0.0800m	0.0159m	0.0822m	0.0153m

As shown in Table 4.7, the current automatic selection of the line segment detection is not robust enough to achieve the same level of accuracy as the manual selection. However, these observations do not preclude the existence of other possible heuristics.

4.5.2 Experiments with Instrument Error

The previous section analyzed the algorithm's performance with ground true sensor readings. In this subsection, we design three groups of comparison experiments to test the robustness of this method in the presence of instrument errors. The experiments are conducted with the best configuration, as found in the previous section (i.e., the camera

of 18 mega pixels is located about 35m away from the building with its photos covering both sides of the building).

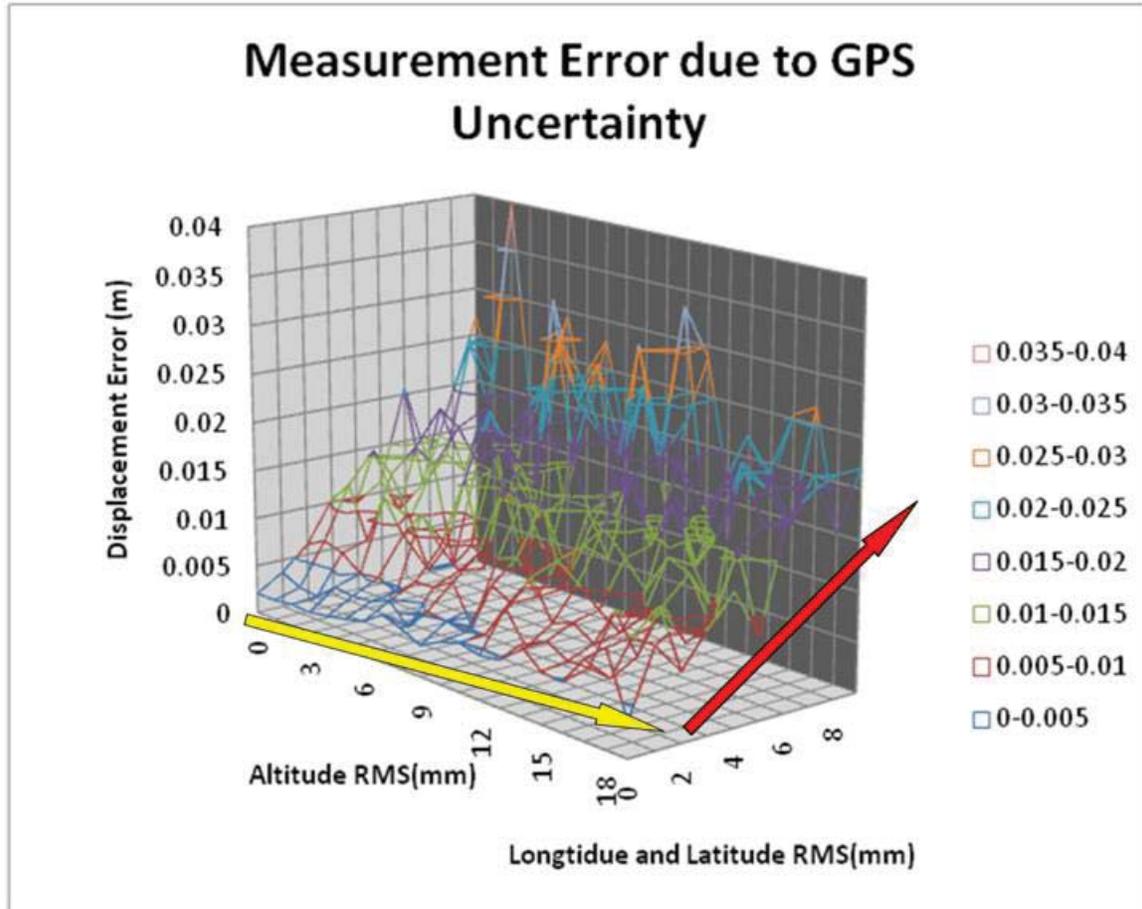


Figure 4.15 Sensitivity of computed drift to camera position errors.

This first experiment assumes ground truth orientation data, and only introduces error to location. In Figure 4.15, the Z-axis shows the average estimation error with the unit of meter. The altitude RMS-axis shows the accuracy response to the change in RTK-GPS altitude measurement uncertainty, and the longitude and latitude RMS-axis shows the accuracy response to the change in both RTK-GPS longitude and latitude measurements' uncertainty. The result indicates that uncertainty on longitude and latitude has a bigger impact on the displacement error than it does on altitude, as indicated by the diagonal

arrow. The result also indicates that longitudes and latitudes smaller than 3mm can achieve the measurement accuracy of 5mm, as indicated by the left to right arrow. Given that the displacement error is linear to the GPS location accuracy, state of the art RTK-GPS can meet the precision requirement. For example, manufacturer-specified accuracy reports uncertainty of 1mm (RMS) on the latitude and longitude, and 2mm~3mm (RMS) on the altitude, in which case displacement error stays below 5mm (Trimble, 2009).

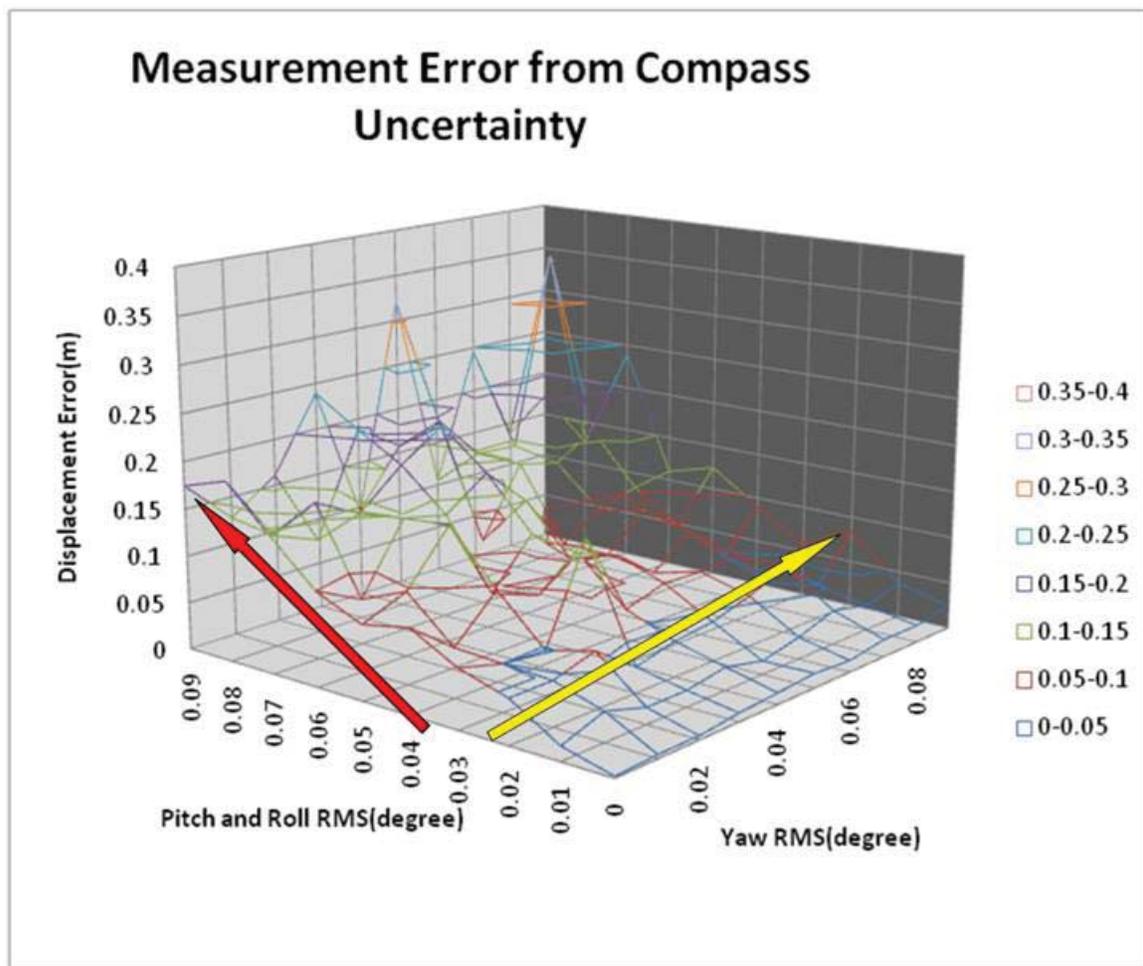


Figure 4.16 Sensitivity of computed drift to camera orientation errors.

The second experiment assumes ground truth location data, and only introduces error to orientation. In Figure 4.16, the Z-axis shows the average estimation error with the unit

of meter. The Pitch and Roll RMS-axis shows the accuracy response to the change in electronic compass pitch and roll readings uncertainty, and the Yaw RMS-axis shows the accuracy response to the change in electronic compass yaw reading uncertainty. The result shows that uncertainty on pitch and roll has a more adverse impact on the displacement error than the yaw does, as indicated by the right to left arrow. Furthermore, a precision of 0.01 degrees (RMS) on all three axes is required to keep the displacement error in the useful range, as indicated by the left to right arrow. Unfortunately, to the author's best knowledge, a state-of-the-art electronic compass cannot satisfy this precision requirement. Most off-the-shelf electronic compasses report uncertainty bigger than 0.1degrees (RMS), thus suggesting the need for survey-grade line-of-sight tracking methods for monitoring the camera's orientation.

The third experiment considers comprehensive errors from both location and orientation readings (Figure 4.17). It again proves that uncertainty from an electronic compass becomes the critical source of error in the methodology.

4.6 Conclusion

This chapter described a simulated Virtual Prototyping test bed to evaluate the feasibility of deploying an Augmented Reality-assisted non-contact building damage reconnaissance method in the field. The research demonstrated the effectiveness of VR-assisted Virtual Prototyping in evaluating and demonstrating new reconnaissance methods where full-scale physical test bed experimentation is impractical. The experimental plan constructed a ten-story graphical building capable of expressing internal structural damage through

texture displacement on the surface. LSD can detect the shifted building edge on the captured building image, and the final corner coordinate is triangulated through the intersections between the detected vertical building edge and the projected horizontal baseline.

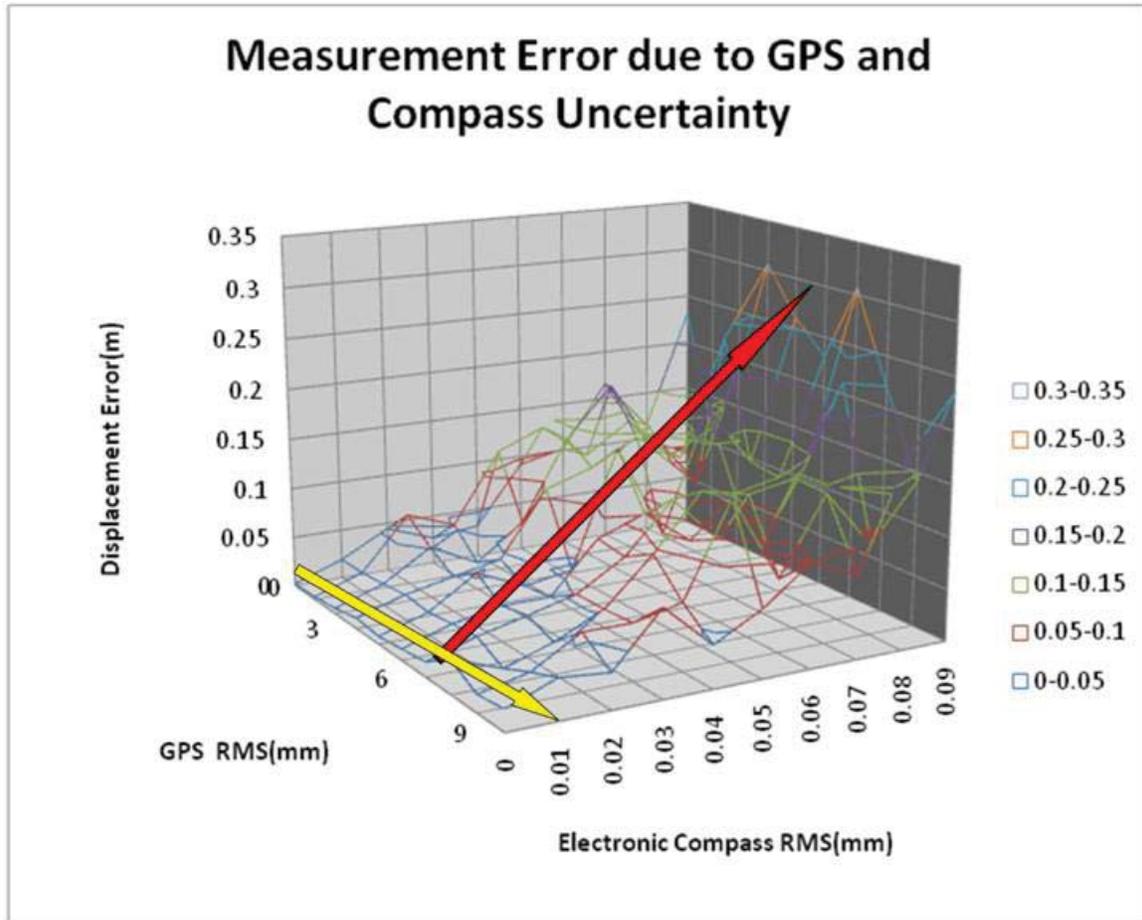


Figure 4.17 Sensitivity of computed drift to camera location and orientation errors.

The experimental results with ground true location and orientation data are satisfactory for damage detection requirements. The results also highlight the conditions for achieving the ideal measurement accuracy, for example observing distance, angle, and image resolution. The experimental results with instrumental errors reveal the bottleneck

for the proposed method in the field implementation conditions. While the state-of-the-art RTK-GPS can meet the location accuracy requirement, the electronic compass is not accurate enough to supply qualified measurement data, suggesting that alternative survey-grade orientation measurement methods must be identified to replace electronic compasses. The conducted sensitivity analysis developed a clear matrix revealing the relationship between instrument accuracy and accuracy of computed drift, so the proposed method's practical implementation can evolve with choices made for higher-accuracy instruments than the ones tested. The author acknowledges that the sensitivity matrix developed from the virtual prototyping may have limitations, and needs to be further validated in a real environment setting. For example, the dynamic illumination may bring challenges to the edge detection. Furthermore, the estimation method assumes ground true geometric building information is available. It is possible that, in reality, such information contains uncertainty or is possibly unavailable for older buildings. The current virtual prototyping has not modeled such data uncertainty. The open source code for the virtual prototyping and its sensitivity analysis is available at <http://pathfinder.engin.umich.edu/software.htm>.

4.7 References

- [1] Rojahn, C. (2005). "ATC-20-1 Field Manual: Postearthquake Safety Evaluation of Buildings." Applied Technology Council.
- [2] Chock, G. (2006). "ATC-20 Post-Earthquake Building Safety Evaluations Performed after the October 15 , 2006 Hawaii Earthquakes Summary and Recommendations for Improvements (updated)." HAWAII STRUCTURAL ENGINEERS ASSOCIATION.

- [3] Vidal, F., Feriche, M., and Ontiveros, A. "Basic Techniques for Quick and Rapid Post-Earthquake Assessments of Building Safety." *Proc., Proceedings of the 2009 International Workshop on Seismic Microzoning and Risk Reduction*, 1-10.
- [4] Tubbesing, S. K. (1989). "The Loma Prieta, California, Earthquake of October 17, 1989-Loss Estimation and Procedures."
- [5] Kamat, V. R., and El-Tawil, S. (2007). "Evaluation of Augmented Reality for Rapid Assessment of Earthquake-Induced Building Damage." *Journal of Computing in Civil Engineering*, 21(5), 303-310.
- [6] Miranda, E., Asce, M., and Akkar, S. D. (2006). "Generalized Interstory Drift Spectrum." *Journal of Structural Engineering*, 132(6), 840-852.
- [7] Krishnan, S. (2006). "Case Studies of Damage to Tall Steel Moment-Frame Buildings in Southern California during Large San Andreas Earthquakes." *Bulletin of the Seismological Society of America*, 96(4A), 1523-1537.
- [8] Shin, D. H., and Dunston, P. S. (2008). "Identification of Application Areas for Augmented Reality in Industrial Construction Based on Technology Suitability." *Journal of Automation in Construction*, 17(7), 882-894.
- [9] Skolnik, D. A., and Wallace, J. W. (2010). "Critical Assessment of Interstory Drift Measurements." *Journal of Structural Engineering*, 136(12), 1574-1584.
- [10] Wahbeh, A. M., Caffrey, J. P., and Masri, S. F. (2003). "A vision-based approach for the direct measurement of displacements in vibrating systems." *Smart Materials and Structures*, 12(5), 785-794.
- [11] Ji, Y. (2010). "A computer vision-based approach for structural displacement measurement." *Sensors and Smart Structures Technologies for Civil, Mechanical, and Aerospace Systems*, 43(7), 642-647.
- [12] Hutchinson, T. C., and Kuester, F. (2004). "Monitoring global earthquake-induced demands using vision-based sensors." *IEEE Transactions on Instrumentation and Measurement*, 53(1), 31-36.
- [13] Lee, J., and Shinozuka, M. (2006). "A vision-based system for remote sensing of bridge displacement." *Journal of NDT E International*, 39(5), 425-431.
- [14] Fukuda, Y., Feng, M., Narita, Y., Kaneko, S., and Tanaka, T. (2010). "Vision-based displacement sensor for monitoring dynamic response using robust object search algorithm." *IEEE Sensors*, 1928-1931.

- [15] Alba, M., Fregonese, L., Prandi, F., Scaioni, M., Valgoi, P., and Monitoring, D. "STRUCTURAL MONITORING OF A LARGE DAM BY TERRESTRIAL LASER." *Proc., Proceedings of the 2006 ISPRS Commission Symposium*.
- [16] Park, H. S., Lee, H. M., Adeli, H., and Lee, I. (2007). "A New Approach for Health Monitoring of Structures: Terrestrial Laser Scanning." *Journal of Computer Aided Civil and Infrastructure Engineering*, 22(1), 19-30.
- [17] Dai, F., Lu, M., and Kamat, V. R. (2011). "Analytical Approach to Augmenting Site Photos with 3D Graphics of Underground Infrastructure in Construction Engineering Applications." *Journal of Computing in Civil Engineering*, 25(1), 66-74.
- [18] Dong, S., and Kamat, V. R. "Robust Mobile Computing Framework for Visualization of Simulated Processes in Augmented Reality." *Proc., Proceedings of the 2010 Winter Simulation Conference*, Institute of Electrical and Electronics Engineers, 3111-3122.
- [19] Wang, Z., McKinley, K. S., Rosenberg, A. L., and Weems, C. C. "Using the compiler to improve cache replacement decisions." *Proc., Proceedings 2002 International Conference on Parallel Architectures and Compilation Techniques*.
- [20] American Heritage Dictionary (2009). *The American Heritage Dictionary of the English Language*, American Heritage Publishing, Boston, MA.
- [21] Jayaram, S., Angster, S. R., Gowda, S., Jayaram, U., and Kreitzer, R. R. (1998). "An Architecture for VR based Virtual Prototyping of Human-Operated Systems." *Proceedings of the 1998 ASME Design Technical Conference and Computer in Engineering Conference*.
- [22] Bauer, M. D., Siddique, Z., and Rosen, D. W. "A Virtual Prototyping System for Design for Assembly, Disassembly, and Service." *Proc., Proceedings of the 1998 ASME Design Technical Conference and Computers in Engineering Conference*.
- [23] Hart, G. C. (2008). "AN ALTERNATIVE PROCEDURE FOR SEISMIC ANALYSIS AND DESIGN OF TALL BUILDINGS LOCATED IN THE LOS ANGELES REGION AN ALTERNATIVE PROCEDURE FOR SEISMIC ANALYSIS AND DESIGN OF TALL BUILDINGS LOCATED IN THE LOS ANGELES REGION." Los Angeles Tall Buildings Structural Design Council
- [24] Sardinas, C. (2010). "New York - Empire State Building." CS3Design.
- [25] Trimble (2007). "AgGPS RTK Base 900 and 450 receivers.", Trimble.

- [26] Trimble (2009). "Trimble R8 GNSS Datasheet."
- [27] PNI (2009). "User Manual Field Force TCM XB."
- [28] The Digital Picture (2012). "Field of View Crop Factor (Focal Length Multiplier)." <<http://www.the-digital-picture.com/canon-lenses/field-of-view-crop-factor.aspx>>.
- [29] Canny, J. (1986). "A computational approach to edge detection." *IEEE Transactions Pattern Analysis and Machine Intelligence*, PAMI-8(6), 679-698.
- [30] Duda, R. O., and Hart, P. E. (1972). "Use of the Hough Transformation to Detect Lines and Curves in Pictures." *Communications of the ACM*, 15(1), 11-15.
- [31] Xu, N., and Bansal, R. "Object Segmentation Using Graph Cuts Based Active Contours." *Proc., Proceedings of 2003 IEEE Conference on Computer Vision and Pattern Recognition*, 46-53.
- [32] Terzopoulos, D., Witkin, A., and Michael, K. (1988). "Constraints on deformable models: Recovering 3D shape and nonrigid motion." *Journal of Artificial Intelligence*, 36(1), 91-123.
- [33] von Gioi, R., Jakubowicz, J., Morel, J.-M., and Randall, G. (2010). "LSD: A Fast Line Segment Detector." *IEEE Transactions Pattern Analysis and Machine Intelligence*, 722-732.
- [34] Burns, B. J., Hanson, A. R., and Riseman, E. M. (1986). "Extracting Straight Lines." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(4), 425-455.
- [35] Moisan, L., and Morel, J.-m. (2000). "Meaningful Alignments." *International Journal of Computer Vision*, 40(1), 7-23.

Chapter 5

Collaborative Learning of Engineering Processes Using Tabletop Augmented Reality Visual Simulations

5.1 Introduction

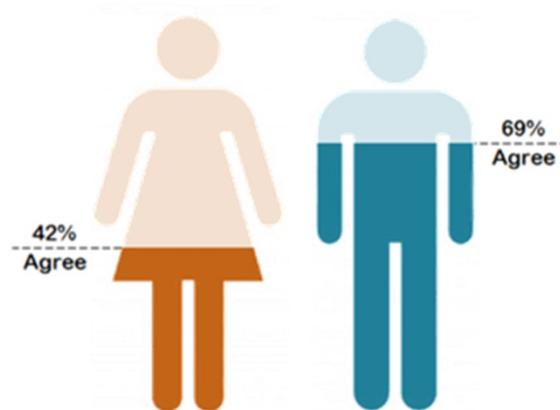
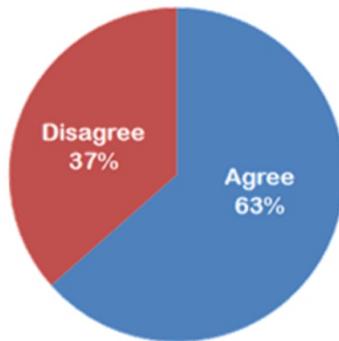
During the past several years, engineering systems have been rapidly growing in terms of complexity, scale, uncertainty, and interdisciplinarity. In this regard, construction systems and projects are no exception. Most construction projects involve parallel assembly-like complex processes that interact in a dynamic environment. The circumstances in which these processes take place often become more complicated due to unforeseen conditions, deviations from project plans, change orders, and legal issues. Despite this, figures show that many construction and civil engineering students have historically lacked a comprehensive knowledge of onsite construction tasks and the dynamics and complexities involved in a typical construction project (Arditi and Polat, 2010).

Nonetheless, the curricula of most construction and civil engineering programs do not fully convey the knowledge and skills required by future field workers and project engineers to effectively face and resolve these challenges.

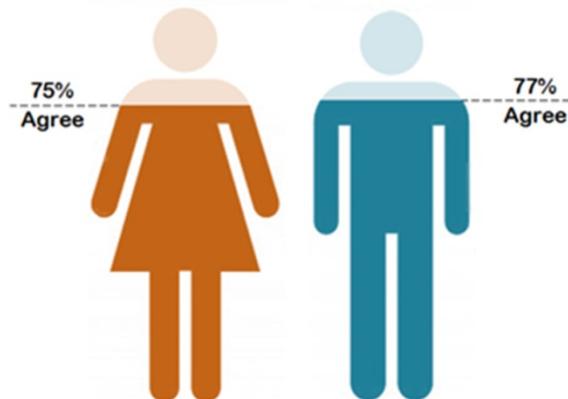
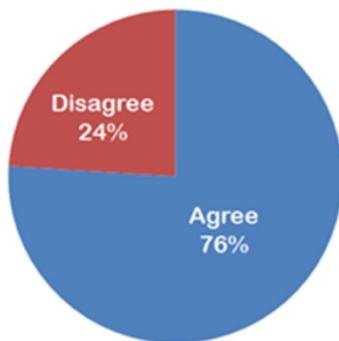
The continued emphasis on traditional information delivery methods—including the use of chalkboards, handouts, and lecture-style presentations—coupled with focusing mainly on simplistic approaches and unrealistic assumptions to formulate and solve complicated engineering problems, can potentially result in construction and civil engineering students falling behind in applying what they learn in the classroom to practical scenarios in the field (Tener, 1996). In particular, students complain that little effort is put into educating them with the latest trends of emerging technologies and advanced problem-solving tools. Figure 5.1 shows the results of a recent survey of 63 undergraduate students in civil, environmental, and construction engineering at the University of Central Florida (Behzadan and Kamat, 2012). The survey indicated that a solid majority of students believed that, compared to other engineering disciplines, they were exposed to fewer technology advancements in the classroom.

Engineering students need to pick up the social and technical skills (e.g. critical thinking, decision making, collaboration, and leadership) that they need in order to be competent in the digital age. Bowie (2010), Mills and Treagust (2003) discussed how most students are graduating with a decent knowledge of fundamental engineering science, but that they don't know how to apply that knowledge in practice.

Compared to other engineering disciplines, instructors in civil and construction engineering use less technology in classroom.



I learn better when working in a collaborative setting where I can play a role in the learning process (e.g. teamwork).



I learn better when the instructor uses 3D representations or visualization to teach engineering concepts and theories.

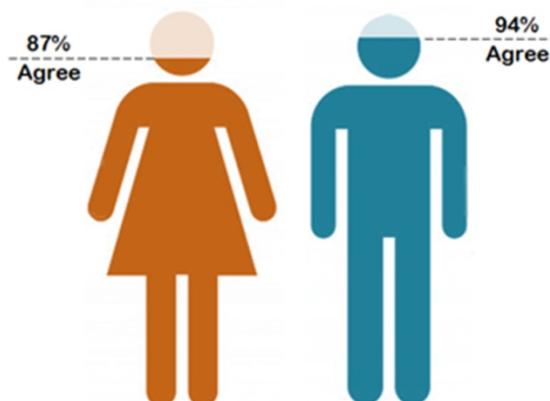
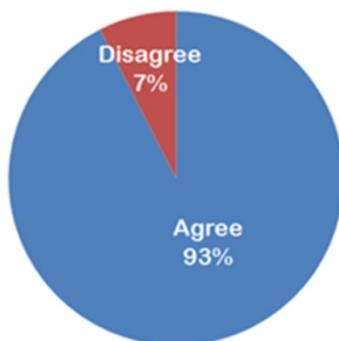


Figure 5.1 A survey of undergraduate civil, environmental, and construction engineering students revealed that a large percentage of students support the prospect of reforming current instructional methods.

One of the fastest emerging technologies in engineering education is visualization. According to the same student survey (Figure 5.1), more than 90% of those who responded indicated that they learn better when the instructor uses 3D representations or visualization to teach engineering concepts and theories. Although instructional methods that take advantage of visualization techniques have been around for several decades, these methods still rely on traditional media and tools. For example, students who take a course in construction planning may use drawings, scheduling bar charts, sand table models, and more recently, 3D CAD models. However, none of these techniques are capable of effectively conveying information on every aspect of a project. For instance, 2D or 3D models do not reflect temporal progress, while scheduling bar charts do not demonstrate the corresponding spatial layout.

More recently, some studies have been conducted on linking 3D CAD models with construction schedules, so as to exploit the dynamic 3D nature of construction at the project level. This class of visualization technique is commonly known as 4D CAD, where based upon the planned work sequence (i.e. project schedule), individual CAD components are added to the target facilities as time advances. At the project level, 4D CAD modeling proves its value in minimizing the misinterpretation of a project sequence by integrating the spatial, temporal, and logical aspects of construction planning information (Koo and Fischer, 2000).

Unlike project-level visualization in which only major time-consuming processes are animated, operations-level visualization explicitly represents the interaction between equipment, labor, materials, and space (e.g. laying bricks, lifting columns). This

visualization approach is especially powerful when there is a need to elaborate on operational details such as the maneuverability of trucks and backhoes in excavation areas, and the deployment of cranes and materials in steel erection. Such tasks require careful and detailed planning and validation, so as to maximize resource utilization and to identify hidden spatial collision and temporal conflicts. Therefore, this visualization paradigm can help engineers in validating and verifying operational concepts, checking for design interferences, and estimating overall constructability (Kamat and Martinez, 2003).

There are two major categories of 3D visualization that can be used to reconstruct engineering operations for education and training purposes, and to facilitate the study of environments in which such operations take place: Augmented Reality (AR) and Virtual Reality (VR). AR is the superimposition of computer-generated information over a user's view of the real world. By presenting contextual information in a textual or graphical format, the user's view of the real world is enhanced or augmented beyond the normal experience (Behzadan and Kamat, 2005). The addition of such contextual information spatially located relative to the user can assist in the performance of several scientific and engineering tasks. For this reason, AR-enabling technologies have been researched in an increasing number of studies during recent years. AR is different from VR, a visualization technology that has been around for several decades. As shown in Figure 5.2, unlike VR, AR does not completely replace the real world, rather the real world is supplemented with relevant synthetic information, and thus real and virtual objects coexist in an augmented space (Azuma, 1997).

The real advantage of AR is that the view of the real world is used as a readymade backdrop for displaying superimposed graphics or information. This allows AR users to create and overlay only the information that needs to be augmented onto the real-world view, and as a result, recreating the whole surrounding environment—which often proves to be a time-consuming and computing-intensive task—is no longer a concern. In addition, the very fact that a human observer of an AR scene is part of the real surrounding world enables the creation of immersive augmented environments where the observer can essentially interact with both real and virtual objects. Collaborative AR takes this one step further by allowing multiple users to access a shared space populated by virtual objects (Kaufmann and Schmalstieg, 2003).

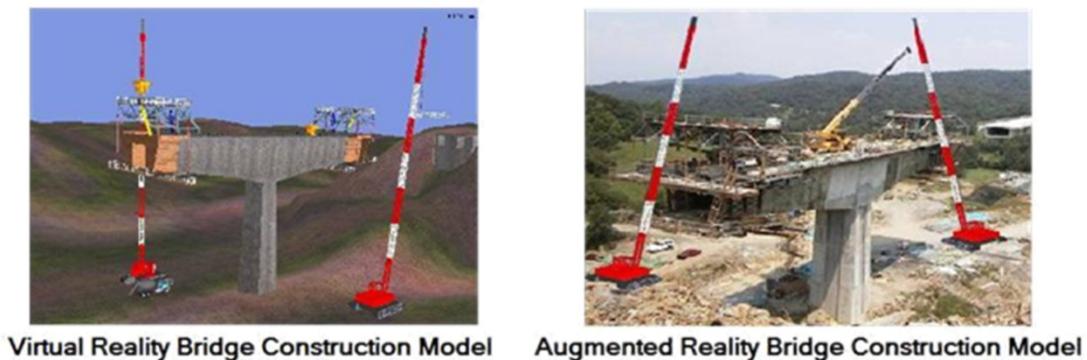


Figure 5.2 The view of the real world is used as a readymade backdrop for displaying superimposed information in AR.

5.2 Main Contributions

3D computer visualization has gained significant attention in engineering education. From the point of view of collaborative learning, however, there are still outstanding

challenges that need to be addressed before 3D visualization can fully be implemented in a classroom setting. These challenges and the associated knowledge gaps have been the major motivation for the presented research. The author attempt to identify an interconnecting media to bridge the gap between computer-based dynamic visualization and paper-based collaboratively shared workspace. AR is one of the most promising candidates because it blends computer-generated graphics with real-scene backgrounds, using real-time registration algorithms. Users can work across the table face-to-face, shift the focus of shared workspace instantly, and jointly analyze dynamic engineering scenarios. This idea is developed and implemented in ARVita (acronym for Augmented Reality Vitascope), in which multiple users wearing HMDs can observe and interact with dynamic simulated construction activities laid on the surface of a table. In the following section, a summary of previous efforts to incorporate 3D visualization techniques into construction and civil engineering education will be presented. In each case, the methodological benefits and limitations will also be described in an effort to highlight the significance of the presented work.

5.3 3D Visualization of Engineering Operations

5.3.1 Virtual Reality Visualization of Engineering Operations

Researchers have previously investigated the potential of VR visualization in animating simulated construction and civil engineering operations. Kamat and Martinez (2001) designed the VITASCOPE (acronym for VISualizaTION of Simulated Construction OPERations) visualization system for articulating operations-level construction activities

using a comprehensive and extensible authoring language for depicting modeled processes along a simulated project timeline. The VITASCOPE authoring language is an abstract layer to make the visualization engine (VE) independent of any particular driving processes (e.g., a specific simulation system). Incoming operation events and data generated by simulation models, hardware controls, or real-time sensors can be adapted to conform to the VITASCOPE syntax, and can be fed into the dynamic VE (Kamat and Martinez, 2003).

The VE is built on top of a scene graph architecture and the frame updating algorithms (Kamat and Martinez, 2002), and it is used to interpret the instruction sets and render the depicted activities sequentially in the virtual environment. VITASCOPE is capable of visualizing simulated construction operations in smooth, continuous, and animated 3D virtual worlds. Nonetheless, from the interaction perspective, VITASCOPE is—by its nature—a post-processing animation engine; it thus does not allow users to dynamically alter the course of a simulation.

VITASCOPE thus degenerates the validation confidence to one single realization of the simulation, while other possible cases are not discovered and analyzed. This constraint can be somewhat reduced if the simulation and animation can run concurrently, meaning users observe the subsequent changes based on their interaction with the animation. This idea motivated the design and implementation of VITASCOPE++ to extend the existing capabilities of VITASCOPE to include message-based architecture. VITASCOPE++ enables users to communicate from inside of the

running animation with the state of simulation that drives it, and consequently allows users to affect the remaining course of the simulation (Rekapall and Martinez, 2007).

5.3.2 Augmented Reality Visualization of Engineering Operations

From the modeling perspective, recreating the construction jobsite in a fully virtual environment always involves quantities of modeling work (e.g. terrain and existing facilities). Even though many such modeling elements are not part of the simulation process, their existence is necessary to realistically represent the jobsite context. Such 3D CAD modeling engineering demands a significant amount of effort in acquiring, creating, and maintaining the models (Brooks, 1999). In order to overcome this challenge, Behzadan and Kamat (2009a) previously created ARVISCOPE (acronym for Augmented Reality VISualization of Simulated Construction OPERations), a visualization system that shares the same capabilities with VITASCOPE in creating dynamic, smooth, and continuous construction activity animations, while cutting off the effort potentially needed for context modeling (Behzadan and Kamat, 2009b).

This is essentially achieved by blending simulated graphics with real scenes. The backbone of this system is a robust georeferencing registration algorithm that applies the user's geographical position tracked by a GPS receiver, and the 3D orientation of the user's head tracked by an electronic compass unit to calculate the correct pose of 3D construction graphics in outdoor AR environments. Compared to VR, AR can enhance the traditional learning experience since:

- The ability to learn concepts and ideas through interacting with a scene and building one's own knowledge (constructivism learning) facilitates the generation of knowledge and skills that could otherwise take too long to accumulate.
- Traditional methods of learning spatially related content by viewing 2D diagrams or images create a cognitive filter. This filter exists even when working with 3D objects on a computer screen because the manipulation of objects in space is done through mouse clicks. By using 3D-immersive AR, a more direct cognitive path toward understanding the content is possible.
- Making mistakes during the learning process will have literally no real consequence for the educator, whereas in traditional learning, the failure to follow certain rules or precautions while operating machinery or handling a hazardous material could lead to serious safety and health-related problems.
- AR supports discovery-based learning—an instructional technique in which students take control of their own learning process, acquire information, and use that information in order to experience scenarios that may not be feasible in reality given the time and space constraints of a typical engineering project.
- An important objective of all academic curricula is to promote social interaction among students, and to teach them to listen, respect, influence, and act. By providing multiple students with access to a shared augmented space populated with real and virtual objects, they are encouraged to become involved in teamwork and brainstorming activities to solve a problem, which simultaneously helps them improve their communication skills.

5.3.3 Collaborative Learning through 3D Visualization

Collaborative learning in which individuals are the cornerstones of the learning process has proven to be one of the most effective instructional methods. This is also evident from Figure 5.1 where, on average, 76% of students surveyed named collaborative learning as their most preferred method of learning. As far as collaborative learning is concerned, the convenience of traditional paper-based discussion is somewhat lost in computer-based VR environments, where users' discussion is restricted to the scale of the screen. On the other hand, as shown in Figure 5.3, even though paper-based media is difficult to handle, maintain, and update, it is a natural collaboration platform that allows people to promptly exchange ideas.

Group discussion cultivates face-to-face conversation, where there is a dynamic and easy interchange of focus between the shared workspace and speakers' interpersonal space. The shared workspace is the common task area between collaborators, while the interpersonal space is the common communication space. The former is usually a subset of the latter (Billinghurst and Kato, 1999). Educators can use a variety of non-verbal cues to quickly shift the focus of a shared workspace accordingly, and thus work more efficiently. Compared to VR, AR by definition better supports the prospect of collaborative learning and discussion. The collaborative features of AR visualization have been previously explored. In this context, researchers have investigated the potential of AR-based education and training in three areas: localized collaborative AR, remote collaborative AR, and industrial collaborative AR.

Some early work in localized collaborative AR is found in Billinghamurst and Kato (1999), Rekimoto (1996), and Szalavári, et al. (1997). The TRANSVISION system developed by Rekimoto (Rekimoto, 1996) is a pioneering work in collaborative AR. In it, multiple participants use palmtop handheld displays to share computer-generated graphics on a table. Collaborative Web Space (Billinghurst and Kato, 1999) is an interface for people in the same location to view and interact with virtual world wide web pages floating around them in real space. The Studierstube system (Szalavári, et al., 1997) mainly targets presentations and education. Each viewer wears a magnetically tracked see-through HMD, and walks around to observe 3D scientific data.



Figure 5.3 Traditional paper-based media is ideal for collaborative work, despite disadvantages such as difficulty in handling, maintaining, and updating.

Other related work, like collaborative AR game- and task-oriented collaboration, then followed this trend. The Art of Defense (Huynh, et al., 2009) is a typical AR board game, in which gamers use handheld devices to play social games with physical game pieces on a tabletop. Nilsson et al. (Nilsson, et al., 2009) did a comparison experiment on cross-organizational collaboration in dynamic emergency response tasks. Actors hold positive attitudes toward AR, and would like to use it for real tasks. Besides the traditional Head Mounted Display (HMD) and Hand Held Display (HHD), a number of other AR media exist (e.g. projection table and multi-touch table). The augmented urban planning workbench (Ishii, et al., 2002) is a multi-layered luminous table for hybrid presentations

like 2D drawings, 3D physical models, and digital simulation overlaid onto the table. The system was used in a graduate course supporting the urban design process. Multi-Touch Mixed Reality (Wei, et al., 2010) allows designers to interact with a multi-touch tabletop interface with 2D models, while 3D models are projected onto their 2D counterparts.

In remote collaborative AR systems, avatars are the most necessary elements of the visualization environment. WearCom (Billinghurst and Kato, 1999) enables a user to see remote collaborators as virtual avatars in multi-party face-to-face AR conferencing (Minatani, et al., 2007). The system recreates each participant's facial appearance in real time, and represents each participant's upper body and hands above the table as a deformed billboard (Stafford, et al., 2006), thus inventing an interactive metaphor, termed "god-like," for improving the communications of situational and navigational information between outdoor and indoor AR users. The gestures of indoor users are captured by video-based tracking and shown as "god-like" style guidance to the outdoor users.

Industrial collaborative AR is mainly used in product design and factory planning. The MagicMeeting system (Regenbrecht, et al., 2006) is used in concrete test cases in which experts from the automotive industry meet to discuss the design of car parts. The collaboration is powered by a tangible AR interface. Fata Morgana (Klinker, et al., 2002), on the other hand, also demonstrates car design, but uses a life-sized model in a BMW show room. At Siemens Corporate Research, a fully implemented system called CyliCon (Navab, 2003) enables users to move around the environment and visualize as-built reconstruction on real sites and in industrial drawings. The Roivis project is another successful example of factory design and planning at Volkswagen Group Research

(Pentenrieder, et al., 2007). This project applied AR in interfering edge analysis and aggregation verification, and insisted on strict standards for the system's accuracy.

AR has also been widely studied in construction in areas including but not limited to operations visualization, computer-aided operations, project schedule supervision, and component inspection. However, there are few examples in the collaborative AR domain. For instance, Wang and Dunston (Wang and Dunston, 2008) developed an AR face-to-face design review prototype and conducted test cases for collaboratively performing error detection. Hammad et al. (Hammad, et al., 2009) applied distributed AR for visualizing collaborative construction tasks (e.g., crane operations) to check spatial and engineering constraints in outdoor jobsites.

To the author's best knowledge, none of the previous work in this domain allows users to validate simulated processes and learn from the results by collaboratively observing animations of dynamic operations. In the following sections, the design requirement, technical implementation, and capabilities of ARVita—a collaborative AR-based learning system for visualizing and studying dynamic construction operations—will be described. The rest of this chapter includes descriptions and discussions about the design and technical implementation of ARVita and is organized as follows: i) software scheme (Section 5.4.1), ii) realization of software scheme in OpenSceneGraph (OSG) (Section 5.4.2), iii) tracking libraries (Section 5.5), and iv) multiple views and their limitations (Section 5.6).

5.4 Technical Implementation of ARVita

5.4.1 Model-View-Controller Software Architecture of ARVita

The software architecture of ARVita conforms to the classical Model-View-Controller (MVC) pattern (Figure 5.4).

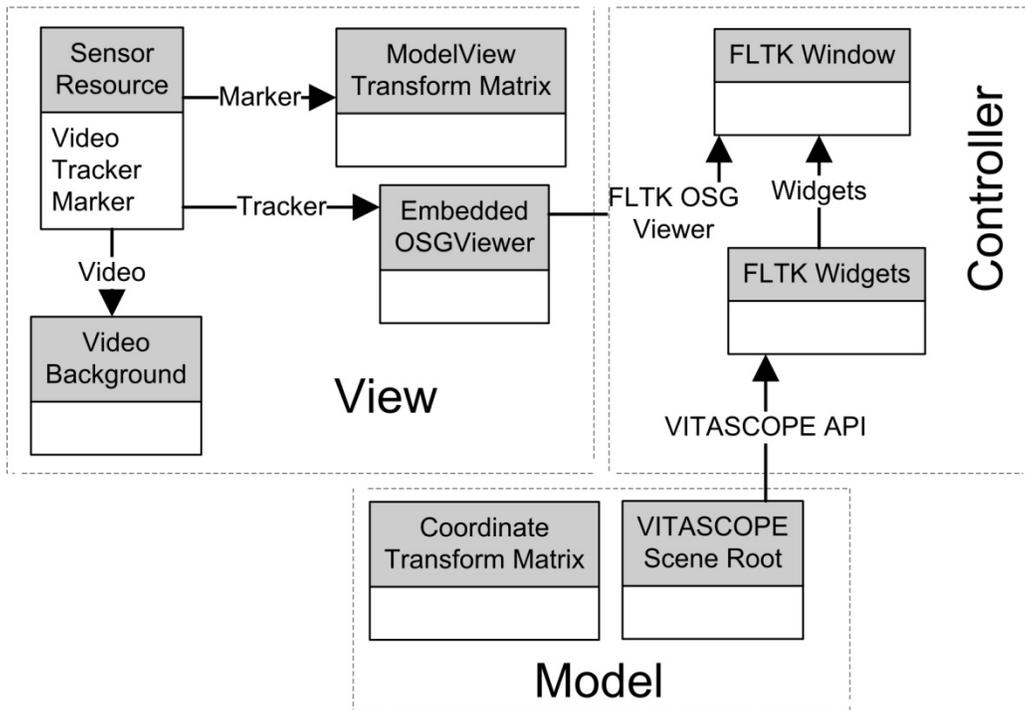


Figure 5.4 The software architecture of ARVita conforms to the Model-View-Controller pattern. The arrow indicates a ‘belongs to’ relationship.

A Model is responsible for initializing, archiving, and updating the VITASCOPE scene node. The manipulation of the VITASCOPE scene node is possible because the VITASCOPE visualization engine exposed a list of APIs (Application Programming Interface), granting developers full control of the underlying animation process (e.g., open and close files, start and pause animation, and alter animation speed and

timestamp). A Controller communicates users' interaction/input commands to the VITASCOPE API wrapped inside the Model. The communication channel is powered by FLTK (acronym for Fast Light Toolkit) that translates and dispatches mouse/key messages to the Model and the View. The View displays the updated Model content, and this is based on the premise that it can correctly set up projection and ModelView matrices of the OpenGL cameras. First, a camera projection matrix is populated at the start of the program based on the camera calibration result; this is to make sure that the OpenGL virtual camera and real camera share a consistent view volume. Second, the ModelView matrix—the pose of the OpenGL camera—is updated every frame based on the marker tracking results so that CAD models are transformed to the correct stance relative to the camera.

5.4.2 Implementation of Model-View-Controller Using OpenSceneGraph

OSG is chosen for the implementation of the MVC pattern described above. OSG uses an acyclic directional graph (tree) to express the scene storing geometry, state, and transformation nodes. The graph is traversed at each frame for updating, drawing, and culling purposes (Martz, 2007). More importantly, its update and event callbacks mechanism makes it convenient for driving the simulated construction operations and performing the tracking procedure (Figure 5.6). For example, tracking and transforming logic can be predefined as callbacks, and then executed upon tree traversal at every frame.

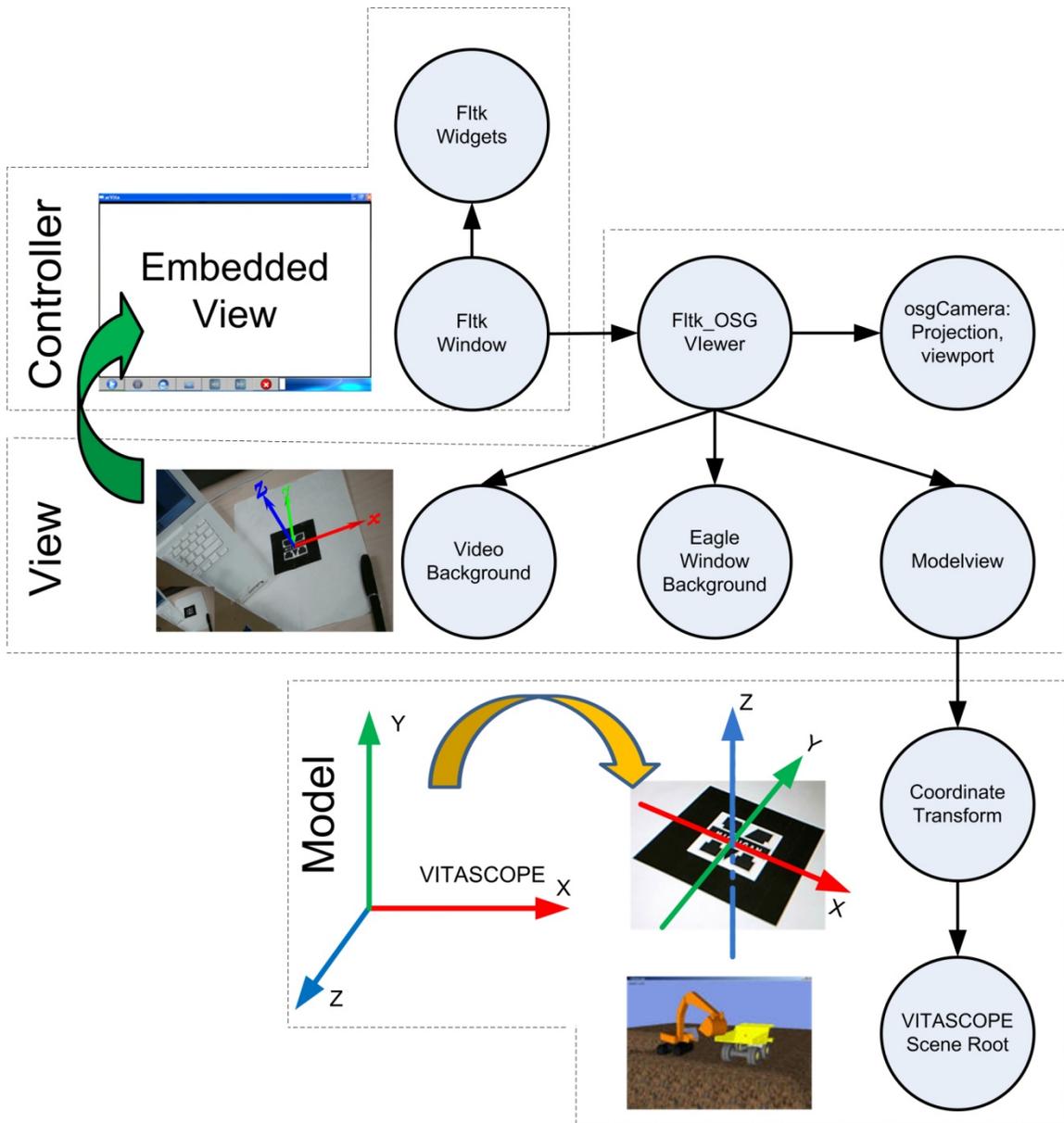


Figure 5.5 The realization of the Model-View-Controller model with OpenSceneGraph.

5.4.2.1 Model

The VITASCOPE scene node—the core logic of the model—resides at the bottom of the tree (Figure 5.5). The `vitaProcessTraceFile()` function is called up every frame to update the animation logic. Above the scene node is a coordinate transformation node. Since all of the tracking algorithms used in ARVita assume the Z-axis is up and use the right-hand

coordinate system, this transformation converts VITASCOPE's Y-axis to be up, and converts the right-hand coordinate system to ARVita's default system, so that the jobsite model is laid horizontally above the marker.

5.4.2.2 View

The core of the View is the FLTK_OSGViewer node that inherits methods from both the FLTK window class and osgViewer class, and thus functions as the glue between the FLTK and OSG. Under its hood are the ModelView transformation node and video stream display nodes. The ModelView matrix is updated frame to frame by the tracking event callbacks. This approach follows osgART's example (OSG ARToolkit) that uses the 'Tracker and Marker' updating mechanism to bundle the tracking procedure, e.g. ARToolkit, and OSG together. Both Tracker and Marker are attached as event callbacks to the respective node in the graph (Figure 5.6).

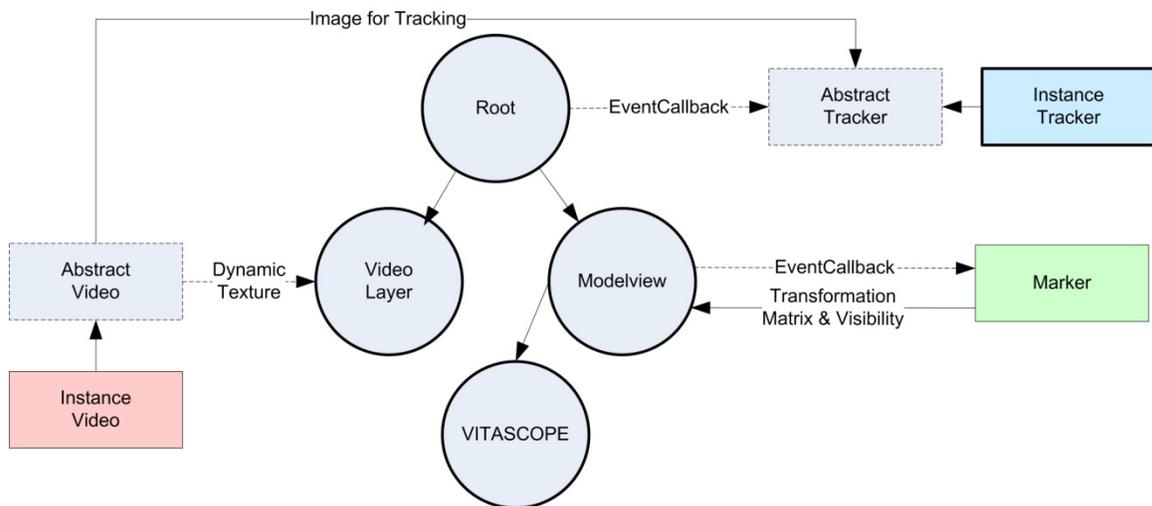


Figure 5.6 osgART's Tracker and Marker updating mechanism.

The Tracker reads updated video frames and stores the detected physical marker descriptor in the Marker. Consequently, the Marker calculates the camera's pose in the world coordinate system based on the descriptor, and updates the ModelView transformation node. ARVita chooses to comply with this 'Tracker and Marker' mechanism because it is an abstract layer to separate the tracking and rendering logic. For example, as will be shown later, this mechanism is versatile in accommodating new tracking procedures, and making the change in the Tracker transparent to the rest of the software structure.

The video resource is pasted as dynamic texture on the background and the eagle window. Despite the stability of the trackers used in ARVita, all of them require the majority of the marker, or even the entire marker, to be visible in the video frame. For example, in the ARToolkit, the CAD models could immediately disappear as soon as a tiny corner of the marker is lost from the camera's sight. This limitation is much more severe when the animated jobsite covers the majority of the screen, which makes it very difficult to cover the marker within the camera's view volume. The eagle window is thus valuable for mitigating these flaws. It can be toggled on and off by the user. When the user moves the camera to look for a vantage point, the eagle window can be toggled such that the user is aware of the visibility of the marker. When the camera is set to static, and the user is paying attention to the animation, the eagle window can be toggled off so it won't affect the observation.

5.4.2.3 Controller

FLTK plays the role of the Controller in ARVita, and it translates and dispatches a user's interaction/input messages to the system. The motivation behind ARVita is to allow multiple users to observe the animation from different perspectives, and promptly shift the focus of the shared working space in a natural approach. These natural interactions include rotating the marker to find vantage points, and pointing at the model to attract others' attention (Figure 5.7). Given that the scale of the model may prevent users from getting close enough to interesting regions, ARVita provides users with basic zooming and panning functionalities.

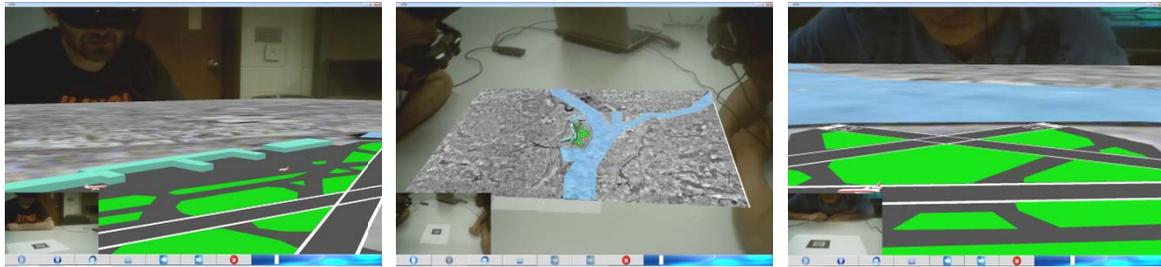


Figure 5.7 Two users are observing the animation lying on the table.

The focus of the shared working space can not only be switched spatially, but also temporally. Users can choose to observe the animation at different speeds, or jump instantaneously along the timeline (Figure 5.8). The ARVita Controller wraps the existing VITASCOPE APIs, like `vitaExecuteViewRatioChange()` and `vitaExecuteTimeJump()`, in a user-friendly interface as most media players do—using fast-forward buttons, a progress bar, etc.

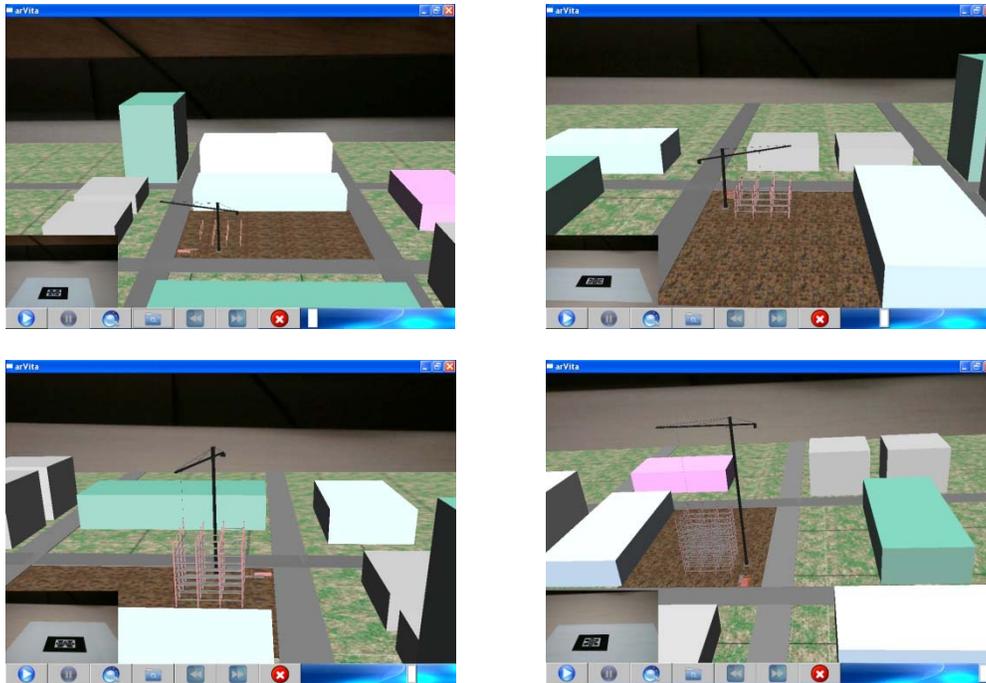


Figure 5.8 Steel erection activities at different timestamps.

5.5 Planar tracking methods for Collaborative AR

As noted earlier, the ‘Tracker and Marker’ mechanism makes ARVita versatile in accommodating different tracking procedures. It currently comes with two available trackers. The first one, ARToolkit (Hirokazu and Billinghurst, 1999), is a widely used fiducial marker tracking library. The second one, KEG (Feng and Kamat, 2012), was developed at the University of Michigan, and is a natural marker tracking library. This section will articulate the importance of tracking in AR, and will describe mainstream tracking approaches and those used in ARVita. It will also highlight the KEG tracking algorithm.

5.5.1 Principle of Tracking

Tracking, which is also referred to as registration in the AR community, is the procedure of aligning real and virtual objects properly in an augmented world to create the illusion that they coexist across time and space. Specifically, since real objects are shown by the physical camera, and virtual objects are shown by the OpenGL virtual camera, the coexistence of heterogeneous objects implies that these two cameras share the same pose. Pose is the translation and rotation of the camera relative to the origin of the world coordinate system. A physical camera's pose needs to be tracked continuously and used to alter the ModelView matrix of the OpenGL camera accordingly. Tracking is acknowledged as the fundamental challenge in the AR community, and as such, has been well studied for decades (Azuma, 1997).

5.5.2 Taxonomy of Tracking Methods

There are a variety of tracking methodologies depending on the application. For example, in the outdoor environment, a GPS and electronic compass are usually employed together to track a camera's position and orientation (Feiner, et al., 1997). However in the indoor environment, where a GPS signal is blocked, visual tracking methods are usually preferred (Figure 5.9). Based on visual tracking libraries' assumptions about the environment, they can be classified as known and unknown environment tracking methods. SLAM (acronym for simultaneous location and mapping) (Klein and Murray, 2007) is the representative of an unknown environment tracking method that imposes few assumptions about the environment. The other school of tracking method assumes a known environment and can be further divided into non-planar and planar environments.

The former works with 3D structures that have known visual features (usually CAD models) (Drummond and Cipolla, 2002). Despite both SLAM and non-planar methods requiring a high computational budget, they can be listed as promising candidates for the future version of ARVita, as their loose restriction on the environment could grant users more flexibility when observing animations.

The planar marker tracking method is simpler compared to the previous tracking methods, but is sufficient given the application context of ARVita, where the working space is usually a large flat table laid with markers; users can seat themselves around the table and search for vantage points. The planar marker’s branches consist of a fiducial marker and natural marker tracking method, and tracker options in ARVita include both.

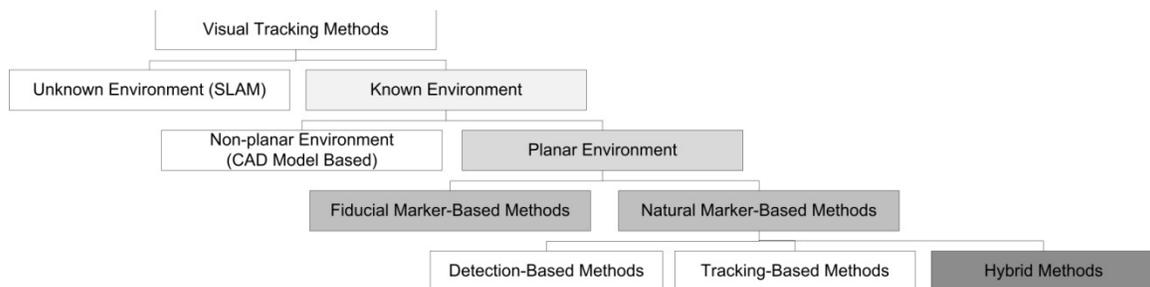


Figure 5.9 The taxonomy of tracking methods.

5.5.3 Trackers Available in ARVita

5.5.3.1 The Fiducial Marker Tracking Method, ARToolkit as an Example

The fiducial marker tracking method is efficient and fast. This is because the fiducial marker is usually composed as an artificial picture that contains ‘easy to extract’ visual features like a set of black and white patterns. The extraction of these patterns—straight lines, sharp corners, and circles, for example—is fast and reliable. ARToolkit is one of

the oldest fiducial marker tracking methods, and is widely considered a benchmark in the AR community. ARToolkit's fiducial marker is a logo bounded by a thick black frame (Figure 5.5). The four corners of the frame are used to compute the camera pose, and the center logo is used to interpret the identity of the marker. Because of its simplicity and fast tracking speed, ARToolkit has been popular for over a decade in numerous applications. However, it also suffers from the common shortcomings of a fiducial marker, and requires all four of its corners to be visible to the camera so that the camera pose can be computed. This can cause frustration when the user navigates through the animated jobsite only to find the animated graphics blinking on and off due to loss of tracking. This disadvantage motivated the author to look into natural markers, which are more flexible with regard to the marker's coverage.

5.5.3.2 The Natural Marker Tracking Method, KEG as an Example

Besides the advantage of partial coverage (Figure 5.10), the natural marker offers the advantage of not depending on special predefined visual features, like those found in the fiducial marker. In other words, the features can be points, corners, edges, and blobs that appear in a natural image. The extraction of these features is vital in establishing correspondence between the observed image by the camera and the marker image, and in estimating the camera's pose. Therefore, robust estimation usually requires the establishment of ample correspondence, which is a challenging issue.

There are two schools for tracking the natural markers, depending on whether they treat observed images independently or consecutively. The former is referred to as a detection-based method, such as SIFT (Scale Invariant Feature Transform) (Lowe, 2004)

or FERNs (Ozuysal, et al., 2007). The latter is addressed as a tracking-based method, such as Kanade-Lucas-Tomasi (KLT) (Lucas and Kanade, 1981, Shi and Tomasi, 1994). Each school has its own pros and cons. A robust detection-based method often demands a high computational budget, and can hardly meet real-time frame rates. On the other hand, accumulated errors in the tracking-based method can be carried forward along consecutive frames, and thus compromise the tracking quality. The KEG tracker used in ARVita inherits merits from both the detection-based and tracking-based methods. The following sections will briefly cover these two methods, and how they are combined by the proposed global appearance and geometric constraints.



Figure 5.10 The natural marker tracking library is flexible on marker visibility.

5.5.3.2.1 Detection-based Method

The correspondence relation between the marker image and the captured image can be expressed as a transformation matrix, called homography, that maps points on the observed image to their corresponding points on the marker image.

H is a 3×3 matrix that represents the homography matrix, and s is a scale factor. p' (x', y') and p (x, y) is a certain pair of corresponding points on the marker and observed images. Furthermore, H encodes the camera's pose (i.e., rotation and translation) because (x, y) can also be transformed to (x', y') through rotation (R), translation (T), and camera calibration (K) matrices. In other words, if K can be found through camera calibration, R and T can be calculated via H .

The above definition can be expressed as $p'_i \times H p_i = 0$, and thus H can be solved given multiple matching pairs of $\langle p'_i, p_i \rangle$. Because most existing algorithms for finding matching points like SIFT and FERNs only inspect the local appearance, mismatching is inevitable in some cases. Accounting for the existence of outliers/mismatching, RANSAC (RANDOM Sample Consensus) (Fischler and Bolles, 1981) is used to estimate parameters of H . In the KEG tracker, one of RANSAC's variants (Simon, et al., 2000) is used.

5.5.3.2.2 Tracking-based Method

A tracking-based method is based on the premise that the difference between two consecutive observed images is small, and thus the feature points can be tracked by checking the points' local neighborhood patches. Apparently this approach costs less than

the detection-based method, where prior knowledge is discarded in comparing the current captured image with the marker image. Once the correspondence is established at the initial stage by using one of the detection methods mentioned in 4.3.2.1, those feature points can be tracked on the following observed images, and the initial detection cost is amortized in the tracking stage. The tracking library used in KEG is Kanade-Lucas-Tomasi (KLT).

Table 5.1 - Comparison between two natural marker approaches.

Approach	Detection-Based	Tracking-Based
Principle	Identify matching feature points on each new frame	Follow up the existing feature points from frame to frame
Relation between consecutive frame	Independent	Current frame is correlated with previous one
Pros	Failure of estimation in one frame will in no way affect the next frame	Fast
Cons	Time-consuming	Error of estimation in one frame will be carried forward, and the accumulated error will eventually lead to loss of tracking

5.5.3.2.3 Global Appearance and Geometric Constraints

Table 5.1 summarizes the profiles of detection-based and tracking-based methods. The framework of the KEG tracker integrates these two methods and enhances the overall performance by using global appearances and geometric constraints. In Figure 5.11, black boxes that imply feature points on the marker image are identified only once when the program starts, and grey boxes and dashed lines mean the detection-based method is only employed at the initial phase or the loss of track case. Once matching feature points are found, the path is shifted to the tracking-based method (solid lines), where the known

feature points are tracked by KLT. The estimated homography using RANSAC is addressed as coarse H, and may still contain systematic and random errors.

The first enhancement introduced by KEG is called the global appearance constraint, which processes coarse H under ESM (Benhimane and Malis, 2004) as a second-order approximation. The yield refined H can thus rectify the captured image similar to the appearance of the marker image. The second enhancement is called the global geometric constraint. Since the refined H is assumed to be theoretically free of systematic and random errors, the updated feature points \bar{x}_{updated} to be tracked at the next frame do not inherit the accumulated errors. Here x_{marker} refers to the feature points found on the marker image. These two enhancements not only boost the tracking accuracy and stability, but also increase the tracking speed, which is attributed to the global refinement that reduces the iterations required by the KLT algorithm.

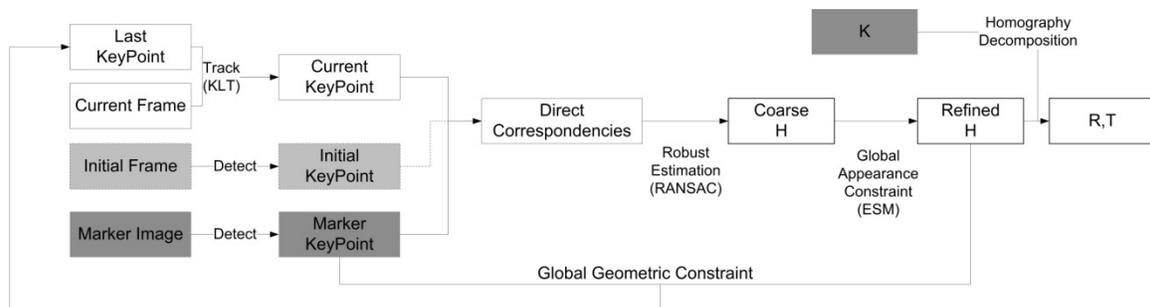


Figure 5.11 The algorithm flowchart of the KEG tracker.

5.5.3.2.4 The Introduction of AprilTag

Besides tracking the pose of the camera, another necessity is to recognize the identity of the natural marker. Currently, KEG takes advantage of the coding system in the AprilTags (Olson, 2011) tracking library to identify the associated information with a

certain marker. Even though AprilTags, itself, is part of the fiducial tracking family, it does not need to be fully covered by the camera once the identity of the marker is confirmed (Figure 5.10).

5.6 Multiple Views in ARVita

5.6.1 Technical Implementation of Multiple View

The `OSGCompositeViewer` class is the key to upgrading the single-view version of ARVita to the multiple-views version. Composite Viewers is a container of multiple views; it keeps the views synchronized correctly and threaded safely. Each view plays the same role as the `FLTK_OSGViewer` does in Figure 5.5, and independently maintains its own video, tracker, marker resources, and `ModelView` matrix. However, these views share the same `VITASCOPE` scene node (Figure 5.12 (b)) for two reasons: 1) to synchronize animation across different views; and 2) to save memory space by maintaining only one copy of each scene node.

The number of instance views depends on how many video capture devices are available. Based on their system device ID, ARVita presents users with a list of available web cameras as the program starts, and lets the users choose the number of views and corresponding webcams (Figure 5.12 (a)). When one user interacts with the model—rotating the marker, zooming, or dragging the progress bar—all of these spatial or temporal updates will be reflected in all of the other users' augmented spaces, so that a consistent dynamic model is shared among all users.

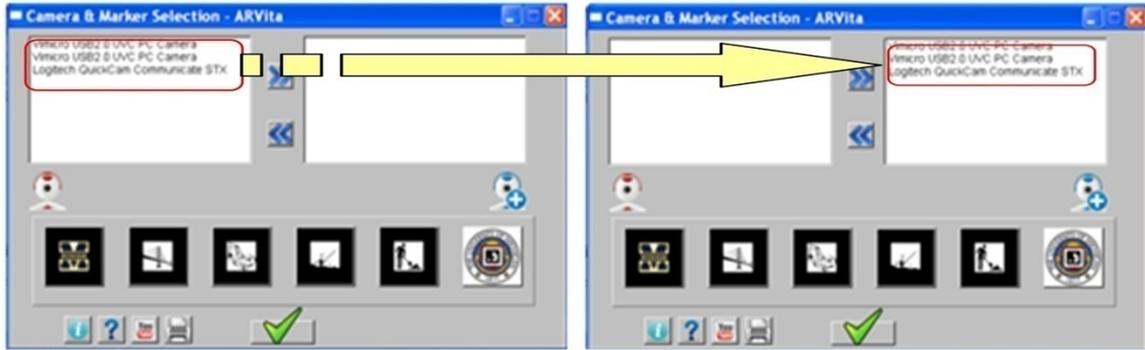


Figure 5.12(a) Users can select multiple cameras that are detected by ARVita as the program starts.

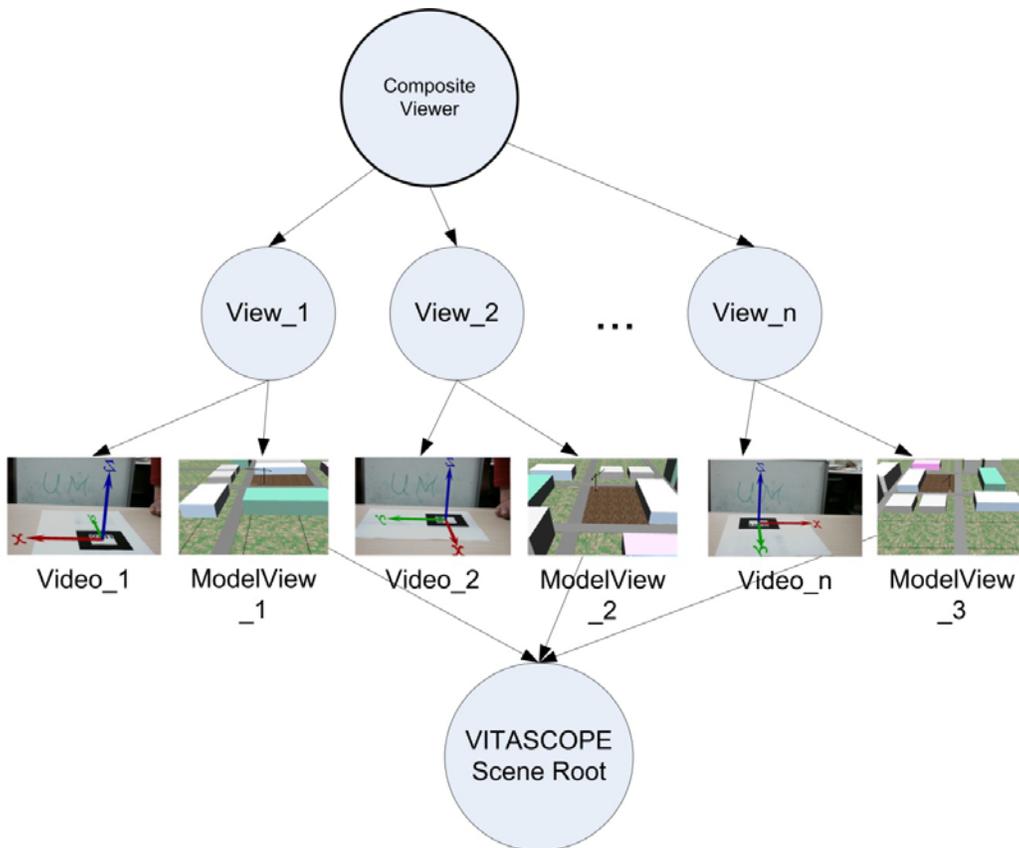


Figure 5.12(b) All of the views possess their own video, tracker, and marker objects, but point to the same VITASCOPE scene node.

5.6.2 Limitations of Multiple Views on a Single Computer

The current version of ARVita supports running multiple views on a single computer, indirectly limiting the maximum number of participants. As more “viewers” join, the computer quickly gets overloaded by maintaining too many video resources and tracking procedures. The author are currently pursuing an alternative distributed computing approach to overcome this limitation. As a generic architecture for distributed computer simulation systems, HLA (acronym for High Level Architecture) can not only integrate heterogeneous simulation software and data sources, but also communicate between computers, even platforms. HLA thus presents itself as a promising solution for a distributed ARVita. However, having multiple views on one computer is still useful. For example, in the multi-view version of ARVita, one can observe animation from different viewpoints simultaneously, and thus acquire a broader comprehension of the whole simulated processes.

5.7 Conclusion and Future Work

3D visualization is one of the most effective tools for exposing engineering students to the latest trends of emerging technology advancements in the classroom. Statistics also indicate that students learn better when their instructors use 3D representations or visualization to teach engineering concepts and theories. Even though 3D visualization methods in the form of 3D/4D CAD models have existed for decades, they have not yet largely replaced traditional media and tools. One reason for this is that traditional visualization tools offer less opportunity for collaboration.

As an effort to enable collaborative learning through 3D visualization, we introduce a software program named ARVita for collaboratively visualizing dynamic 3D simulated construction operations. Users sitting across a table can have a face-to-face discussion about 3D animations “laid on” the table surface. Interaction functionalities are designed to assist users in moving smoothly between the focus of shared workspaces. A video demo of ARVita can be found on the website: <http://pathfinder.engin.umich.edu/videos.htm>.

The next generation of ARVita will focus on offering students more flexibility and space when they observe a 3D animation model. Two thrusts of improvements can be made. The first is to enable the tracking library to function in the unknown environment (i.e., SLAM) so that a user’s observation range is no longer limited by the visibility of the marker. In other words, any flat table with an ample amount of features could be a tracking plane. The second area for improvement relates to the efforts that are being made to make ARVita comply with the rules of HLA. This compliance will allow ARVita to be distributed and synchronized across computers. When this happens, students can run multiple instances of ARVita on their own computers, but still collaborate on the synchronized model. The current version of ARVita software and its source code can be found on the website: <http://pathfinder.engin.umich.edu/software.htm>.

5.8 References

[1] Arditi, D., and Polat, G. (2010). "Graduate Education in Construction Management." *Journal of Professional Issues in Engineering Education and Practice*, 3, 175-179.

[2] Tener, R. K. (1996). "Industry-University Partnerships for Construction Engineering Education." *Journal of Professional Issues in Engineering Education and Practice*, 122(4), 156-162.

[3] Behzadan, A., and Kamat, R. V. "A Framework for Utilizing Context-Aware Augmented Reality Visualization in Engineering Education." *Proc., Proceedings of the 2012 International Conference on Construction Applications of Virtual Reality (CONVR)*.

[4] Bowie, J. (2010). "Enhancing classroom instruction with collaborative technologies." <<http://www.eschoolnews.com/2010/12/20/enhancing-classroom-instruction-with-collaborative-technologies/>>.

[5] Mills, J. E., and Treagust, D. F. (2003). "Engineering Education - Is problem-based or project-based learning the answer?" *Australasian Journal of Engineering Education*, 1-16.

[6] Koo, B., and Fischer, M. (2000). "Feasibility Study of 4D CAD in Commercial Construction." *Journal of Construction Engineering and Management*, 251-260.

[7] Kamat, V. R., and Martinez, J. C. (2003). "Automated Generation of Dynamic, Operations Level Virtual Construction Scenarios." *Electronic Journal of Information Technology in Construction (ITcon)*, 8, 65-84.

[8] Behzadan, A. H., and Kamat, V. R. "Visualization of Construction Graphics in Outdoor Augmented Reality." *Proc., Proceedings of the 2005 Winter Simulation Conference*, Institute of Electrical and Electronics Engineers (IEEE).

[9] Azuma, R. (1997). "A Survey of Augmented Reality." *Teleoperators and Virtual Environments*, 355-385.

[10] Kaufmann, H., and Schmalstieg, D. "Mathematics and Geometry Education with Collaborative Augmented Reality." *Proc., Journal of Computers & Graphics*, 37-41.

[11] Kamat, V. R., and Martinez, J. C. (2001). "Visualizing Simulated Construction Operations in 3D." *Journal of Computing in Civil Engineering*, 15(4), 329-337.

- [12] Kamat, V. R., and Martinez, J. C. (2002). "Scene Graph and Frame Update Algorithms for Smooth and Scalable 3D Visualization of Simulated Construction Operations." *Journal of Computer-Aided Civil and Infrastructure Engineering*, 17(4), 228-245.
- [13] Rekapall, P., and Martinez, J. "A message-based architecture to enable runtime user interaction on concurrent simulation-animations of construction operations." *Proc., Proceedings of the 2007 Winter Simulation Conference*, 2028-2031.
- [14] Brooks, F. P. (1999). "What's Real About Virtual Reality?" *Journal of Computer Graphics and Applications*, 19(6), 16-27.
- [15] Behzadan, A. H., and Kamat, V. R. (2009a). "Automated Generation of Operations Level Construction Animations in Outdoor Augmented Reality." *Journal of Computing in Civil Engineering*, 23(6), 405-417.
- [16] Behzadan, A. H., and Kamat, V. R. "Interactive Augmented Reality Visualization for Improved Damage Prevention and Maintenance of Underground Infrastructure." *Proc., Proceedings of 2009 Construction Research Congress*, American Society of Civil Engineers, 1214-1222.
- [17] Billingham, M., and Kato, H. "Collaborative Mixed Reality." *Proc., Proceedings of the 1999 International Symposium on Mixed Reality*.
- [18] Rekimoto, J. "Transvision: A Hand-held Augmented Reality System for Collaborative Design." *Proc., Proceeding of 1996 Virtual Systems and Multimedia*.
- [19] Szalavári, Z., Schmalstieg, D., Fuhrmann, A., and Gervautz, M. (1997). "'Studierstube' An Environment for Collaboration in Augmented Reality."
- [20] Huynh, T. D., Raveendran, K., Xu, Y., Spreen, K., and MacIntyre, B. (2009). "Art of Defense: A Collaborative Handheld Augmented Reality Board Game." *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, 135-142.
- [21] Nilsson, S., Johansson, B., and Jonsson, A. "Using AR to support cross-organisational collaboration in dynamic tasks." *Proc., Proceedings of the 2009 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 3-12.
- [22] Ishii, H., Underkoffler, J., Chak, D., and Piper, B. (2002). "Augmented Urban Planning Workbench: Overlaying Drawings, Physical Models and Digital Simulation." *Proceedings of the International Symposium on Mixed and Augmented Reality*.

- [23] Wei, D., Zhou, Z. S., and Xie, D. "MTMR: A conceptual interior design framework integrating Mixed Reality with the Multi-Touch tabletop interface." *Proc., Proceedings of the 2010 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 279-280.
- [24] Minatani, S., Kitahara, I., Kameda, Y., and Ohta, Y. "Face-to-Face Tabletop Remote Collaboration in Mixed Reality." *Proc., Proceedings of 2007 IEEE Conference on Computer Vision and Pattern Recognition*, 43-46.
- [25] Stafford, A., Piekarski, W., and Thomas, H. B. "Implementation of God-like Interaction Techniques for Supporting Collaboration Between Outdoor AR and Indoor Tabletop Users." *Proc., Proceedings of the 2006 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 165-172.
- [26] Regenbrecht, H. T., Wagner, M. T., and Baratoff, G. (2006). "MagicMeeting: A Collaborative Tangible Augmented Reality System." *Virtual Reality*, SpringerLink.
- [27] Klinker, G., Dutoit, A. H., and Bauer, M. "Fata Morgana -A Presentation System for Product Design." *Proc., Proceedings of the 2002 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 76-85.
- [28] Navab, N. "Industrial augmented reality (IAR): challenges in design and commercialization of killer apps." *Proc., Proceedings of the 2003 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 2-6.
- [29] Pentenrieder, K., Bade, C., Doil, F., and Meier, P. "Augmented Reality-based factory planning - an application tailored to industrial needs." *Proc., Proceedings of the 2007 IEEE and ACM International Symposium on Mixed and Augmented Reality*, 1-9.
- [30] Wang, X., and Dunston, P. (2008). "User Perspectives on Mixed Reality Tabletop Visualization for Face-to-Face Collaborative Design Review." *Journal of Automation in Construction*, 17(4), 399-412.
- [31] Hammad, A., Wang, H., and Mudur, S. P. (2009). "Distributed Augmented Reality for Visualizing Collaborative Construction Tasks." *Journal of Computing in Civil Engineering*, 23(6), 418-427.
- [32] Martz, P. (2007). *OpenSceneGraph Quick Start Guide*.
- [33] Hirokazu, K., and Billinghurst, M. "Marker Tracking and HMD Calibration for a video-based Augmented Reality Conferencing System." *Proc., Proceedings of the 1999 IEEE and ACM International Workshop on Augmented Reality (IWAR 99)*.

- [34] Feng, C., and Kamat, V. R. (2012). "KEG Plane Tracker for AEC Automation Applications." *The 2012 International Symposium on Automation and Robotics in Construction, Mining and Petroleum*.
- [35] Feiner, S., Macintyre, B., and Hollerer, T. "A Touring Machine: Prototyping 3D Mobile Augmented Reality Systems for Exploring the Urban Environment." *Proc., Proceedings of 1997 International Symposium on Wearable Computers*, 74-81.
- [36] Klein, G., and Murray, D. "Parallel Tracking and Mapping for Small AR Workspaces." *Proc., Proceedings of the 2007 IEEE and ACM International Symposium on Mixed and Augmented Reality*.
- [37] Drummond, T., and Cipolla, R. (2002). "Real-Time Visual Tracking of Complex Structures." *IEEE Transactions Pattern Analysis and Machine Intelligence*, 24(7), 932-946.
- [38] Lowe, D. G. (2004). "Distinctive Image Features from Scale-Invariant Keypoints." *International Journal of Computer Vision*.
- [39] Ozuysal, M., Fua, P., and Lepetit, V. "Fast Keypoint Recognition in Ten Lines of Code." *Proc., Proceedings of 2007 IEEE Conference on Computer Vision and Pattern Recognition*, 1-8.
- [40] Lucas, B. D., and Kanade, T. (1981). "An Iterative Image Registration Technique with an Application to Stereo Vision." *Proceedings of the 7th International Joint Conference on Artificial Intelligence*.
- [41] Shi, J., and Tomasi, C. (1994). "Good features to track." *Proceedings of 1994 IEEE Conference on Computer Vision and Pattern Recognition*, 593-600.
- [42] Fischler, M., and Bolles, R. (1981). "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography." *Communications of the ACM*, 24(6), 381-395.
- [43] Simon, G., Fitzgibbon, A. W., and Zisserman, A. "Markerless Tracking using Planar Structures in the Scene." *Proc., Proceedings of the 2000 International Symposium on Augmented Reality*, 120-128.
- [44] Benhimane, S., and Malis, E. "Real-time image-based tracking of planes using efficient second-order minimization." *Proc., Proceedings of 2004 International Conference on Intelligent Robots and Systems*, IEEE, 943-948.

[45] Olson, E. "AprilTag: A robust and flexible visual fiducial system." *Proc., Proceedings of the {IEEE} International Conference on Robotics and Automation (ICRA)*.

Chapter 6

Conclusion

6.1 Significance of the Research

Augmented Reality does not simply blend real and virtual objects together, its underlying value is rooted in its potential to promote people's appreciation of their contexts, and encourage people to discover their surroundings. In the construction domain, a long-term challenge is to map the information from drawings and specifications correctly onto real jobsites, and vice versa. For example, workers try to lay the pipelines at the location labeled on the 'as-design' plans, and managers try to reflect workers' progress by linking the 'as-built' on the jobsite to the schedule. These mapping processes are challenging because they require people to first interpolate abstract knowledge and then relate it to reality based on their experiences.

However, the visual auxiliary information presented by AR links the spatial-temporal –dependent data with the real jobsite, and helps the managers and workers rapidly and accurately establish the mapping relation in their minds. In other words, thanks to AR, ‘what they see is what they get.’ AR has such great potential in the construction industry that Shin and Dunston (2008) identified eight areas in construction where AR can be useful (i.e., layout, excavation, positioning, inspection, coordination, supervision, commenting, and strategizing).

Similar to its applications in many other domains, the usefulness of AR in the construction domain is highly related to the questions of how accurately the virtual models can be aligned with real jobsites, and to how well the illusion can be sustained that virtual and real entities coexist in the augmented space. The answers depend on different environment conditions (Krevelen and Poelman, 2010), and can be more challenging in outdoor, unprepared, and unrestricted environments, like construction sites. A failure to adequately address these challenges compromises the quality of auxiliary information, and lowers people’s confidence in the augmented graphics. Furthermore, it prevents the AR technology from being well received in the construction field.

This research successfully investigated high-accuracy user position and orientation tracking techniques for AR registration, and built a securing harness with high-accuracy Real-Time Kinematics GPS (RTK-GPS), electronic compass, and other supporting peripheral devices. It also instrumented a ubiquitous occlusion-handling algorithm that helps to sustain a spatial and temporal illusion that the virtual and real objects being blended in the augmented space coexist. Furthermore, these solutions have been

integrated into a loosely coupled and extensible AR computing framework called SMART with open access to the community. Researchers who are interested in improving the AR graphical algorithms or developing new AR construction applications can take advantage of the existing AR framework instrumented by this research, and can prototype their ideas in a much shorter lifecycle.

In addition to the tracking and occlusion solutions, this research has also implemented several real-life construction applications based on the AR framework. These applications not only serve as the validation experiments to the AR framework, but also possess huge economic and social impacts. For example, the AR visual collision avoidance system allows spotters to ‘see’ buried utilities hidden under the surface, thus helping prevent utility strike accidents and so reducing losses of life, cost, and time. The rapid AR post-disaster reconnaissance for building damage speeds up the process of quantifying building safety and suitability for the future occupancy, thus shortening the unoccupied period that otherwise translates to economic losses for both owners and the public. The collaborative tabletop AR visualization system bridges the gap between paper-based static drawings and computer-generated dynamic models, thus enabling users to collaborate on discussing and strategizing about project progress.

6.2 Contributions

This research contributes to construction operations by increasing the job safety and productivity with the novel and stable AR visualization technology. For engineers and contractors, this will transform the traditional means of accessing computer-aided

information, and thus improve their spatial awareness and decision-making capabilities. This research also contributes to the enhancement of the educational experience of construction engineering students by exposing them to the novel AR applications in construction engineering, and by shaping their ability to explore AR potential in their future careers.

The following bullet points summarize the individual research challenges successfully investigated and overcome in the preceding chapters:

- A general-purpose, loosely coupled, and extensible AR software framework called SMART with built-in accurate static registration, dynamic misregistration compensation, and occlusion-handling algorithms.
- A rigid and ergonomic hardware platform called ARMOR that can be deployed in precision-engineering applications.
- A ubiquitous occlusion-handling algorithm to present high-credibility AR graphics.
- A visual collision avoidance system for inspecting underground utilities and minimizing the chance of adversely striking these utilities.
- A building damage reconnaissance system for rapidly and accurately interpolating the interstory drift ratio (which is the critical metric for building damage assessment).
- A tabletop AR software called ARVita for collaboratively visualizing simulated processes with various tracking libraries.

- Open source projects available to both the research community and the construction industry.

Chapter 2 presented the details of the static registration principles and the dynamic misregistration mitigation methodologies. Moreover, the software architecture of the SMART framework and the design of the ARMOR platform are discussed and illustrated. Comparisons have been made between SMART and its predecessor ARVISCOPÉ, and between ARMOR and its predecessor UM-AR-GPS-ROVER, all in order to highlight the improvements. The visual excavation-collision avoidance system as an extension to SMART and ARMOR was introduced in detail: extraction of geometric and attribute information; generation of the graphical pipelines; and rendering of the soil environment. *The primary contribution of the research presented in Chapter 2 was the designed scalable and extensible SMART framework and the ergonomic and robust ARMOR platform. Together they serve as the infrastructure to precision-critical AR applications. One of the good examples is the instrumented visual collision-avoidance system.*

Chapter 3 covered the details of a ubiquitous real-time occlusion-handling algorithm for solving the depth-hidden problems between real and virtual objects. An RGB-TOF hybrid camera was built to capture the color illumination and depth image at the same time. The image registration between the color and depth image is achieved by using either homography or stereo projection. Both algorithms have been implemented using the GLSL graphical shader and the Render to Texture technique on the GPU for parallel computing. The algorithms, along with the hybrid camera, have been validated under both indoor and outdoor construction processes simulations. *The primary contribution of*

the research presented in Chapter 3 was the robust and generic AR occlusion-handling algorithm for correctly resolving occlusion effects in real time and delivering credible AR graphics with strong spatial clues. Even though the current operation range is limited to 7.5m by the depth-sensing device, the algorithm is designed to be generic enough to be easily adapted to future products with a wider operational range and higher level of accuracy.

Chapter 4 introduced an AR-assisted non-contact reconnaissance method for estimating buildings' Interstory Drift Ratio. IDR is by far the most trustworthy and robust metric for assessing structural integrity at the story level. In this algorithm, the Line Segment Detector was used to estimate the vertical building edges, and an exhaustively numerical testing algorithm was developed to estimate the horizontal building edges. Eventually, the 3D coordinates at key locations were triangulated from the 2D intersections between the horizontal and vertical edges. A virtual prototyping environment called Aurora was built to test the algorithm's sensitivity to a variety of environment ambient and the instrument accuracy. *The primary contribution of the research presented in Chapter 4 was the rapid and accurate AR-assisted approach for assessing post-earthquake building damage. The sensitivity analysis in the reconfigurable VP environment mapped a clear matrix for the relation between the algorithm performance and the instrument accuracy. While the initial purpose was for assessing post-earthquake building damage, if the precision allows, such a method can also be applied to the inspection of regular buildings or even buildings under construction.*

Chapter 5 discussed the development of a collaborative tabletop AR visualization system called ARVita. ARVita was instrumented as an extension to the outdoor SMART framework on the tabletop environment with fiducial and natural tracking libraries. Furthermore, the extensibility of the ‘Tracker and Marker’ mechanism makes ARVita versatile in accommodating new tracking methodologies. Right now the system allows multiple users wearing HMDs to sit around the table, observing and interacting with dynamic construction simulated processes. *The primary contribution of the research presented in Chapter 5 was the collaborative tabletop AR system that serves as an interconnecting medium that bridges the gap between static paper-based drawings and dynamic computer-generated models. The system allows construction managers to have rapid and accurate access to digital information retrieval for decision making, and to preserve the nature of collaborative convenience.*

In summary, it is expected that the results from this research will spawn influential advances in construction operation and inspection, construction education, and safety training. Further, the results should accelerate future AR discoveries made by other researchers.

Success in the robust AR framework powers precision-critical engineering applications with accurate registration methods and credible graphics rendering. This can potentially transform the practice in construction and other engineering domains related to safety and productivity. The ability to visualize invisible information is such an innovation because attributes and spatial data that are otherwise printed on specifications and drawings, or roughly estimated by the eyes, can now be represented in front of field

operators and inspectors. Such transformation revamps the media representation approach and speeds up the process of information retrieval. Personnel assisted by the system will no longer rely on their subjective memory or judgment, but rather accurate and real-time AR visual assistance.

In addition, the framework is designed as a scalable and extensible paradigm in the hope that researchers interested in testing the software system with their own tracking devices can simply replace the system's I/O interface with minimum impact on other components. Likewise, domain experts conceiving of applications can integrate the logic components into their systems without worrying about the underlying registration and rendering procedures. In general, the replacement of registration, rendering, or application modules is transparent to other components, and precious research resources can be dedicated to either the AR graphical engine or the application logic, instead of being repeatedly wasted on assembling pieces.

In addition, the work in this dissertation contributes to the research community in the sense that all pertinent code is released under common open source license at <http://pathfinder.engin.umich.edu/software.htm> >. Researchers will better appreciate the principles behind the manuscripts by referring to the well-documented code. End users interested in implementing their own application can quickly get their hands on the integration, and may do so by referring to the application examples of visual collision avoidance, building damage reconnaissance, and tabletop collaborative AR visualization.

6.3 Directions for Future Research

This research explored techniques that feature high-precision tracking methods and robust occlusion-handling algorithms, and that have been built into an extensible and scalable AR computing framework. However, certain limitations mentioned in the preceding manuscripts translate into interesting future research challenges. Some of these initiatives have been mentioned in the conclusion section of each individual chapter: 1) robust dynamic registration algorithm; 2) wider operational range for the occlusion algorithm; 3) excavator-cabin view of the underground utilities; 4) higher-level of orientation tracking accuracy for the building damage reconnaissance; and 5) marker-less tracking for tabletop AR collaboration. In addition, the following subsections unfold some specific broader research directions as potential valuable extensions to SMART.

6.3.1 Augmented Reality with Photorealistic Effect and Interaction Functionality

While this research emphasized the tracking precision and occlusion handling, there are some other desirable features, if added, that can strengthen the credibility of the AR graphics. First of all, despite the convincing spatial clue delivered by the correct occlusion effect, other kinds of artifacts in the AR graphics are still detectable by human eyes and compromise the user's confidence in the AR visual output. For example, in contrast to the real objects whose illuminations and shadows are dynamically affected by the ambience, artificial light sources, or even sunshine, the shading effects of the virtual objects are often static and make the virtual objects distinguishable from reality (Haller, 2004). The suboptimal rendering shading effect separates the artificial and natural

entities, and breaks the seamless integration between them. Therefore, it is desirable to build intelligent virtual objects that are capable of sensing the global illumination conditions, and adjust its own visual response—like illuminations and shadows—according to the environment’s lighting conditions (Figure 6.1). Several interesting computer graphics concepts can be explored in this case, like Image-Based Lighting and Bidirectional Reflectance Distribution Function (BRDF) (Saulo, et al., 2010).



Figure 6.1 The comparison between the photorealistic AR image (left) and the non-photorealistic one (right) (Aittala, 2010).

Even though this research has the philosophy of ‘what you see is what you get,’ AR can be more useful if the synthetic objects interact with the real environment. There are two kinds of interactions. In the first case, a synthetic object actively alters its own status as a reaction to the change in the real environment; for example, an artificial car brakes in front of a stop sign and yields to a physical car that passes by (Figure 6.2 (a)). In the second case, a synthetic object’s status is passively affected by the interference from the real environment; for example, an artificial backhoe is operated by a worker to control the simulated excavation activity (Figure 6.2 (b)). Such interactions not only make AR

graphics more believable (Beaney and Namee, 2009), but also provide real-time feedback in ‘human-in-the-loop’ simulations.



(a) (Behzadan and Kamat, 2008)

(b)

Figure 6.2 Interactions between the synthetic and physical objects in simulated construction processes.

6.3.2 Augmented Reality on Mobile Devices

In Chapter 1, the dissertation author claimed that the usefulness of AR heavily depends on its registration accuracy in the precision-critical construction applications like building damage reconnaissance and visual excavation-collision avoidance. This assertion leads to the design of ARMOR targeting high-precision construction tasks. Despite the ergonomic features of ARMOR, its cost is expensive and its launch is non-trivial. Therefore, ARMOR is best operated by professionals in certain precision-critical jobs, rather than by normal users in daily routines. However, in other types of applications like strategizing, coordinating, and commenting, the registration accuracy can be relaxed to meter-level without compromising the mission. More importantly, these applications should be made available to normal users in their daily routines with convenient access. The deployed

devices must thus be light-weight, rigid, and economical to be well received in the construction community.

The advent of mobile devices, particularly the smart phone and the tablet PC, brings the most promising platform to reality. The integrated GPS, electronic compass, and camera contain all of the necessary sensors for the AR applications in both indoor and outdoor environments. In addition, the upcoming Google Project Glass (Levy, 2012) can further free the user from the device, and let them retrieve the relevant information hands-free while performing their tasks. The research challenge is not only about transplanting SMART from the PC end to the mobile device end, but enriching the human-machine interaction experience by taking advantage of the gesture control and voice command input unique to the mobile device.

6.3.3 Seamless and Collaborative Augmented Reality with Cloud Computing

From strategizing in the office, to coordinating in the trailer, to commenting and monitoring on the jobsite, the footprint of AR can be identified in most phases of the lifecycle of a construction project. Currently, research activities develop individual AR software for each phase without a complete solution for the whole lifecycle. This phenomenon translates into fragmentation that blocks the information flow, raises the barrier of interoperability, and makes it hard for the contractors to keep track of updates. Therefore, a complete and seamless AR solution that keeps the information flow in a closed loop will build a unified construction activities stream inbox, increasing the efficacy of the communication channel.

Furthermore, given the nature of construction, constant and effective communication about the standards, designs, and progress is essential to the success of a project, thus the collaboration functionality among groups of workers is a must-have feature. For example, the coordination decision made by the contractors in the trailer should be instantly reflected as AR graphics in front of the workers. Likewise, site photos taken by the workers should be automatically assembled as 3D reconstructions on an AR-powered table so the contractors can keep track of the activity's progress. In addition, the advent of cloud computing technology serves as an ideal infrastructure for the collaboration function in the sense that it makes up for the limited computing power and storage capacity of mobile devices. Cloud computing also enables real-time synchronization across multiple machines and platforms. More importantly, its scalability and economy makes it affordable to a variety of engineering and construction users regardless of their business size.

6.4 References

- [1] Shin, D. H., and Dunston, P. S. (2008). "Identification of Application Areas for Augmented Reality in Industrial Construction Based on Technology Suitability." *Journal of Automation in Construction*, 17(7), 882-894.
- [2] Krevelen, D. W., and Poelman, R. (2010). "A Survey of Augmented Reality Technologies, Applications and Limitations." *The International Journal of Virtual Reality*, 9(2), 1-20.
- [3] Haller, M. "Photorealism or/and Non-Photorealism in Augmented Reality." *Proc., Proceedings of the 2004 ACM SIGGRAPH* 189-196.

[4] Saulo, P., Guilherme, M., Joao, L., and Veronica, T. (2010). "Photorealistic Rendering for Augmented Reality: A Global Illumination and BRDF Solution." *Virtual Reality*, 20-24.

[5] Aittala, M. (2010). "Inverse lighting and photorealistic rendering for augmented reality." *The Visual Computer* 26, 669-678.

[6] Beaney, D., and Namee, B. M. "Forked! A Demonstration of Physics Realism in Augmented Reality." *Proc., Proceedings of the 2009 IEEE and ACM International Symposium on Mixed and Augmented Reality*.

[7] Behzadan, A. H., and Kamat, V. R. "Simulation and Visualization of Traffic Operations in Augmented Reality for Improved Planning and Design of Road Construction Projects." *Proc., Proceedings of the 2008 Winter Simulation Conference*, 2447-2454.

[8] Levy, S. (2012). "Google Glass Team: Wearable Computing Will Be the Norm'." <http://www.wired.com/gadgetlab/2012/06/clear-glass-leaders-googles-wearable-computing-breakthrough-explain-it-all-for-you/all/>.

Appendices

Appendix A

SMART Code Documentation

Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

▼ N SmartCrew	
C CARClass	
C CARObject	
▼ N SmartForeman	
C CARPluginForeman	
C PluginProxy	
C CARSensorForeman	
C CARSiteForeman	
▼ N SmartMotion	
C CARMotionTracker	CARMotionTracker computes the modelview transformation based on the location and orientation
C CARCalibration	
C CARTrackerCallback	Update the modelview rotation matrix and objects' location
▼ N SmartSensor	
C CARLocation	
C CAROrientation	
C CDeviceDelegate	Delegate to notify the unexpected disconnection of physical location or orientation tracking device
C CARCompassSimulator	
C CARGPSReader	
C CARGPSSimulator	
C CARTCMReader	
C timer	Tracking time ticking
▼ N SmartSite	
C CARController	
C ARCameraConfig	Virtual and real video camera settings used in the AR
C CARGraph	
C CARScene	
▼ N SmartVideo	
C CARPGRSource	
C CARRGBSource	
C CARTOFSource	
C CARVideoCamera	
C VideoConfiguration	The web camera setting information, like size, pixel format, device factory configuration

C CARVideoGeode	
C CARVideoSource	
C CResizeEventHandler	Resize the camera's viewport when window size changes
C PreCameraPostDrawCallback	
C VideoCameraPostDrawCallback	

SmartCrew Namespace Reference

Classes

- class [CARClass](#)
 - class [CARObject](#)
-

Detailed Description

SmartCrew contains [CARClass](#) that archives digital models and [CARObject](#) that are instances of the [CARClass](#).

SmartCrew::CARClass Class Reference

Public Member Functions

	CARClass (const char *szClsName, std::string strModelFileName)
	CARClass (const char *szClsName, osg::Node *pLoadedObj)
	CARClass ()
	A default constructor.
virtual	~CARClass (void)
	A default destructor.
osg::MatrixTransform *	GetClsRoot ()
	Get the pointer to the entity class's root.
void	SetClsName (const char *szClsName)
const char *	GetClsName ()
bool	OrientClsModel (std::string strAxis, double dRotAmt)
	Rotate the model around the strAxis by dRotAmt.

Protected Attributes

std::string	m_strClsName
	Name of the entity class.
osg::ref_ptr < osg::MatrixTransform >	m_pClsRoot
	The loaded model from disk is directly attached to the root.
osg::ref_ptr < osg::MatrixTransform >	m_pXRotTransform
	Specify the entity class's rotation around X axis.
osg::ref_ptr < osg::MatrixTransform >	m_pYRotTransform
	Specify the entity class's rotation around Y axis.
osg::ref_ptr < osg::MatrixTransform >	m_pZRotTransform
	Specify the entity class's rotation around Z axis.

Detailed Description

CARClass contains the basic methods for loading models and maintains the transformation node of the models.

Constructor & Destructor Documentation

```
SmartCrew::CARClass::CARClass ( const char * szClsName,  
                                std::string strModelFileName  
                                )
```

A constructor that initializes the entity class by loading the model file from the disk.

Parameters

szClsName The entity class's name.
strModelFileName The model's filename on the disk.

```
SmartCrew::CARClass::CARClass ( const char * szClsName,  
                                osg::Node * pLoadedObj  
                                )
```

A Constructor that initializes the entity class with the given model.

Parameters

szClsName The entity class's name.
pLoadedObj The pointer to the existing model in the memory.

SmartCrew::CARObject Class Reference

Public Member Functions

	CARObject (void)	A default constructor.
virtual	~CARObject (void)	A default destructor.
	CARObject (const char *szObjName, SmartCrew::CARClass *pObjCls)	A constructor that initializes with CARObjet's name and the entity class which it is affiliated.
void	SetObjName (const char *szObjName)	Attribute operation of the name of the object instance.
const char *	GetObjName () const	
void	SetCurVertRotn (double dNewVertRotn)	Set the rotation degree around the Y axis as pitch.
double	GetCurVertRotn () const	
void	SetCurHorRotn (double dNewHorRotn)	Set the rotation degree around the Z axis as yaw.
double	GetCurHorRotn () const	
void	SetCurSideRotn (double dNewSideRotn)	Set the rotation degree around the X axis as roll.
double	GetCurSideRotn () const	
void	SetCurPosn (osg::Vec3 vNewPosn)	Set the object instance's position in its parent's local coordinate system. (not the geographical coordinate system).
osg::Vec3	GetCurPosn () const	
void	SetCurScale (osg::Vec3 vNewScale)	Set the (x, y, z) scale of the instance object.
osg::Vec3	GetCurScale () const	
void	SetGlobalPosn (osg::Vec3 vGlobalPosn)	Set the object instance's geographical location.
osg::Vec3	GetGlobalPosn () const	
bool	GetIsPlaced () const	Whether the object instance has been placed in the scene.
void	SetIsPlaced (bool blsPlaced)	
bool	GetIsAttached () const	Whether this object has been attached to another parent object.
void	SetIsAttached (bool blsAttached)	

bool	GetIsAttachedNS () const	Whether this object has been attached to another parent object.
void	SetIsAttachedNS (bool blsAttachedNS)	
void	SetMyParentObj (CARObject *pParentObj)	Set the parent object to which the object instance is attached.
CARObject *	GetMyParentObj () const	
void	AddOneChildObj (CARObject *pNewChildObj)	Add one more child to the object's child list.
void	RemoveOneChildObj (CARObject *pOldChildObj)	Remove the specified child from the object's child list.
std::list< CARObject * > &	GetMyChildObjs ()	Get the object's child list.
void	SetPivotPoint (osg::Vec3 pNewPivotPoint)	Set the object's position in its immediate parent's local coordinate system.
const osg::Vec3 &	GetMyPivotPoint () const	
SmartCrew::CARClass *	GetMyClass () const	Get the pointer to the object's class model.
bool	ChangeObjCls (SmartCrew::CARClass *pObjCls)	Change the object's current class model with a new class model.
bool	GetIsVisibleInScene () const	Whether this object is currently visible in the scene and is being drawn at each frame.
osg::MatrixTransform *	GetObjRoot ()	
osg::ref_ptr < osg::MatrixTransform >	GetHorOrntTransform () const	
osg::ref_ptr < osg::MatrixTransform >	GetVertOrntTransform () const	
osg::ref_ptr < osg::MatrixTransform >	GetScalingTransform () const	Attach children object to ScalingTransform if the scale of the children does changes with the parent.
osg::ref_ptr < osg::MatrixTransform >	GetAttachNoScaleTransform () const	Attach children object to NoScaleTransform if the scale of the children does not changes with the parent.
osg::ref_ptr < osg::MatrixTransform >	GetSideOrntTransform () const	
void	setAnnotation (std::string txt)	Set the text annotation for the object.

Protected Attributes

std::string **m_strObjName**

	The name of the object instance.
bool	m_blsPlaced
	Whether the object has been placed in the scene.
bool	m_blsAttached
	Whether the object has been attached to a parent object.
bool	m_blsAttachedNS
	Whether the object has been attached to a parent object.
osg::Vec3	m_vPivotPoint
	The position in the local coordinate system of the immediate parent, where the children object is attached.
SmartCrew::CARClass *	m_pCls
	The pointer to the object's class.
CARObject *	m_pParentObj
	The pointer to its parent's object.
std::list< CARObject * >	m_listChildObjs
	The list contains the pointers to its children.
osg::ref_ptr < osg::MatrixTransform >	m_pObjRootTransform
osg::ref_ptr < osg::MatrixTransform >	m_pHorOrntTransform
osg::ref_ptr < osg::MatrixTransform >	m_pVertOrntTransform
osg::ref_ptr < osg::MatrixTransform >	m_pSideOrntTransform
osg::ref_ptr < osg::MatrixTransform >	m_pScalingTransform
osg::ref_ptr < osg::MatrixTransform >	m_pAttachNoScaleTransform
double	m_dCurHorRotn
	Rotation degree around z axis (yaw).
double	m_dCurVertRotn
	Rotation degree around x axis (pitch).
double	m_dCurSideRotn
	Rotation degree around y axis (roll).
osg::Vec3	m_vCurPosn
	The object instance's position in its parent's local coordinate system. (not the geographical coordinate system)
osg::Vec3	m_vCurScale
osg::Vec3	m_vGlobalPosn
	The global/geographical position in the world coordinate system.

Detailed Description

CARObject contains information about the entity class with which it is affiliated, maintenance of its parents and children, control about its transformation node.

Constructor & Destructor Documentation

```
SmartCrew::CARObject::CARObject ( const char *          szObjName,  
                                  SmartCrew::CARClass * pObjCls  
                                  )
```

A constructor that initializes with CARObject's name and the entity class which it is affiliated.

Parameters

szObjName The name of **CARObject**.

pObjCls The pointer to the entity class.

Member Function Documentation

```
bool SmartCrew::CARObject::GetIsAttached ( ) const
```

[inline](#)

Whether this object has been attached to another parent object.

When its parent changes scale, this object changes scale as well.

See Also

[GetIsAttachedNS\(...\)](#)

```
bool SmartCrew::CARObject::GetIsAttachedNS ( ) const
```

[inline](#)

Whether this object has been attached to another parent object.

When its parent changes scale, this object does not change scale.

See Also

[GetIsAttached\(...\)](#)

bool SmartCrew::CARObject::GetIsPlaced () const

inline

Whether the object instance has been placed in the scene.

Whether a valid location has been assigned to the object.

bool SmartCrew::CARObject::GetIsVisibleInScene () const

Whether this object is currently visible in the scene and is being drawn at each frame.

This is only true when: (1) Either the object is placed globally in the scene OR (2) The topmost parent in the attachment hierarchy is placed in the scene.

void SmartCrew::CARObject::SetCurPosn (osg::Vec3 vNewPosn)

inline

Set the object instance's position in its parent's local coordinate system. (not the geographical coordinate system).

If the object has no parent, then the position is in the world coordinate system. If the object has parent, grandparent and etc., then the position is relative to its immediate parent's local coordinate system.

void SmartCrew::CARObject::SetGlobalPosn (osg::Vec3 vGlobalPosn)

inline

Set the object instance's geographical location.

Parameters

vGlobalPosn x is longitude; y is latitude; z is altitude.

Member Data Documentation

bool SmartCrew::CARObject::m_bIsAttached

protected

Whether the object has been attached to a parent object.

The object's scale changes when its parent changes.

See Also

[m_bIsAttachedNS](#)

bool SmartCrew::CARObject::m_bIsAttachedNS

protected

Whether the object has been attached to a parent object.

The object's scale doesn't change when its parent changes.

See Also

[m_bIsAttached](#)

osg::Vec3 SmartCrew::CARObject::m_vCurPosn

protected

The object instance's position in its parent's local coordinate system. (not the geographical coordinate system)

If the object has no parent, then the position is in the world coordinate system. If the object has parent, grandparent and etc., then the position is relative to its immediate parent's local coordinate system.

SmartForeman Namespace Reference

Classes

class [CARPluginForeman](#)

class [PluginProxy](#)

class [CARSensorForeman](#)

class [CARSiteForeman](#)

Detailed Description

The **SmartForeman** is in charge of building `CARMotionTracker` with `CARLocation` and `CAROrientation` instances, and handle the connection with device and the status change. The `SiteForeman` is in charge of coordinating `CARGraph`, `CARScene` and `CARControl`. The `PluginForeman` is borrowed from `osgART` to allow the third party to build their own dlls to replace the existing `GPSReader` and `TCMReader` provided by SMART as default.

SmartForeman::CARPluginForeman Class Reference

Public Member Functions

```
int load (const std::string &pluginname, bool resolveName=true)
osg::Referenced * get (int id)
osg::Referenced * operator[] (int id)
int add (const std::string &name, osg::Referenced *plugin)
```

Static Public Member Functions

```
static CARPluginForeman * instance (bool erase=false)
```

Protected Types

```
typedef std::deque
    < osg::ref_ptr
    < osgDB::DynamicLibrary > > PluginArray
typedef std::map< int,
    osg::ref_ptr< osg::Referenced > > PluginInterfaceMap
typedef std::map< int,
    std::string > PluginTagMap
```

Protected Attributes

```
PluginArray m_plugins
PluginInterfaceMap m_plugininterfaces
PluginTagMap m_plugintags
```

Private Attributes

```
int m_id
```

Detailed Description

CARPluginForeman realizes an auto discovery capable plugin system in order to load and unload plugins on the fly. It is loosely based on the implementation of `osgDB::Registry`.

Member Function Documentation

```
int SmartForeman::CARPluginForeman::add ( const std::string & name,  
                                           osg::Referenced * plugin  
                                           )
```

Add a new instance of a dynamically loaded class.

Parameters

name Identifier to retrieve the object.
plugin Pointer to the instantiated class.

Returns

id of the loaded plugin.

```
osg::Referenced * SmartForeman::CARPluginForeman::get ( int id )
```

Return a pointer to the plugin.

Parameters

id Identifier to get the plugin instance.

Returns

0 if unsuccessfully or a pointer to the plugin.

```
CARPluginForeman *
```

```
SmartForeman::CARPluginForeman::instance
```

```
( bool erase = false ) static
```

Returns an instance of a plugin manager as it is singleton.

Parameters

erase Explicitly deletes the instance of the plugin foreman.

Returns

Instance of the **CARPluginForeman** or 0 if erase was true.

```
int SmartForeman::CARPluginForeman::load ( const std::string & pluginname,  
                                           bool resolveName = true  
                                           )
```

Load an external plugin. The plugin will be loaded and stays in memory unless explicitly unloaded.

Parameters

pluginname Filename short version or complete path (use **resolveName** with false).

Returns

-1 if failed, otherwise this is the ID of the tracker loaded.

```
osg::Referenced * SmartForeman::CARPluginForeman::operator[] ( int id )
```

Shorthand for the get method.

Parameters

id Identifier to get the plugin instance.

Returns

0 if unsuccessfully or a pointer to the plugin.

SmartForeman::PluginProxy< T > Class Template Reference

Public Member Functions

PluginProxy (const std::string &name)

Protected Attributes

osg::ref_ptr< osg::Referenced > **_plugininterface**

Detailed Description

template<typename T>
class SmartForeman::PluginProxy< T >

PluginProxy is a helper to dynamically instantiate and register plugins in the libraries **CARPluginForeman**.

Constructor & Destructor Documentation

```
template<typename T >
```

```
SmartForeman::PluginProxy< T >::PluginProxy ( const std::string & name )
```

inline

Constructor for the dynamic loaded object.

Parameters

name Identifier to retrieve an instance.

Member Data Documentation

```
template<typename T >
```

```
osg::ref_ptr<osg::Referenced> SmartForeman::PluginProxy< T >::_plugininterface
```

protected

Stored instance of the plugin.

SmartForeman::CARSensorForeman Class Reference

Public Member Functions

CARSensorForeman (const std::string &strSensorConfig, std::map< std::string, osg::ref_ptr< **SmartCrew::CARObject** > > &objMap)

A constructor that reads the sensor configuration file.

~CARSensorForeman ()

A default destructor..

void **InitTrackerCallback** (osg::Node *root, osg::Node *transform)

Attach corresponding tracker to the root and transform nodes for updating and retrieving modelview quaternion.

virtual void **HandlePhysicalLoecnDeviceDisabled** (const std::string &msg)

Override handling the emergency of location tracking device disconnection, it usually calls the corresponding simulator.

virtual void **HandlePhysicalLoecnDeviceEnabled** (const std::string &msg)

Override handling the location device back online event (the quality reaches the preferred quality).

virtual void **HandlePhysicalOritrnDeviceDisabled** (const std::string &msg)

Override handling the emergency of orientation tracking device disconnection, it usually calls the corresponding simulator.

bool **SetGeoLon** (float lon)

Set the simulated geographical longitude.

bool **SetGeoLat** (float lat)

Set the simulated geographical latitude.

bool **SetGeoAlt** (float alt)

Set the simulated geographical altitude.

bool **SetYaw** (float yaw)

Set the simulated yaw.

bool **SetPitch** (float pitch)

Set the simulated pitch.

bool **SetRoll** (float roll)

Set the simulated roll.

void **GetGeoLoecn** (osg::Vec3 &vGeoLoecn, struct tm ×tamp)

void **GetOritrn** (osg::Vec3 &vOritrn, struct tm ×tamp)

Protected Attributes

bool **m_bPhysicalLoecnFail**

	The connection with the physical location tracking device fails.
bool	m_bPhysicalOritnFail
	The connection with the physical orientation tracking device fails.
bool	m_bPreferPhysicalLocn
	Whether to use the physical location tracking device specified by the sensor configuration.
bool	m_bPreferPhysicalOritn
	Whether to use the physical orientation tracking device specified by the sensor configuration.
osg::ref_ptr < SmartSensor::CARLocation >	m_pPhysicalLocnReader
	Pointer to the instance of physical location reader.
osg::ref_ptr < SmartSensor::CAROrientation >	m_pPhysicalOritnReader
	Pointer to the instance of the physical orientation reader.
osg::ref_ptr < SmartSensor::CARLocation >	m_pLocnSimulator
	Pointer to the instance of the location simulator.
osg::ref_ptr < SmartSensor::CAROrientation >	m_pOritnSimulator
	Pointer to the instance of the orientation simulator.
osg::ref_ptr< osg::Node >	m_pMVUpdateNode
	The root node that invokes callback to update the modelview matrix.
osg::ref_ptr< osg::Node >	m_pMVRetrieveNode
	The transformation node that invokes callback to retrieve the modelview matrix.
std::map< std::string, osg::ref_ptr < SmartCrew::CARObject > > &	m_objMap
	Reference to the object map.
osg::ref_ptr < SmartMotion::CARTrackerCallback >	m_pMVUpdateCallback
	Callback to update the modelview matrix.
osg::ref_ptr < SmartMotion::CARTrackerCallback >	m_pMVRetrieveCallback
	Callback to update the transformation node based on the modelview matrix.
osg::ref_ptr < SmartMotion::CARMotionTracker >	m_pMotionTracker

Pointer to the motion tracker made from either physical device or simulator.

cv::FileStorage **m_configFS**

The handler of the sensor configuration file.

Private Member Functions

void **BuildTracker** ()

Build CARMotionTracker instance based on the chosen CARLocation and CAROrientation instances.

void **InitLocation** ()

Initialize the CARLocation based on the preference set in the sensor configuration file.

void **InitOrientation** ()

Initialize the CARLocation based on the preference set in the sensor configuration file.

Detailed Description

CARSensorForeman initializes CARLocation and CAROrientation based on the sensor configuration, and builds them into the CARMotionTracker which is invoked by the CARTrackerCallback. It also handles the situation when the connection status with the physical sensors changes.

Member Function Documentation

void **SmartForeman::CARSensorForeman::BuildTracker** ()

private

Build CARMotionTracker instance based on the chosen CARLocation and CAROrientation instances.

The selection is based on the preference set in the sensor configuration file and the connection status with the physical device.

See Also

m_bPhysicalLocnFail **m_bPhysicalOritnFail** **m_bPreferPhysicalLocn**
m_bPreferPhysicalOritn

void **SmartForeman::CARSensorForeman::InitLocation** ()

private

Initialize the CARLocation based on the preference set in the sensor configuration file.

If prefer the physical device but fails to launch, then it will fall back to simulator if the user agrees.

void SmartForeman::CARSensorForeman::InitOrientation ()

private

Initialize the CARLocation based on the preference set in the sensor configuration file.

If prefer the physical device but fails to launch, then it will fall back to simulator if the user agrees.

bool SmartForeman::CARSensorForeman::SetGeoAlt (float alt)

Set the simulated geographical altitude.

Returns

True if in simulation mode, false if not.

bool SmartForeman::CARSensorForeman::SetGeoLat (float lat)

Set the simulated geographical latitude.

Returns

True if in simulation mode, false if not.

bool SmartForeman::CARSensorForeman::SetGeoLon (float lon)

Set the simulated geographical longitude.

Returns

True if in simulation mode, false if not.

bool SmartForeman::CARSensorForeman::SetPitch (float pitch)

Set the simulated pitch.

Returns

True if in simulation mode, false if not.

bool SmartForeman::CARSensorForeman::SetRoll (float roll)

Set the simulated roll.

Returns

True if in simulation mode, false if not.

bool SmartForeman::CARSensorForeman::SetYaw (float yaw)

Set the simulated yaw.

Returns

True if in simulation mode, false if not.

Member Data Documentation

osg::ref_ptr<SmartMotion::CARMotionTracker>

SmartForeman::CARSensorForeman::m_pMotionTracker

protected

Pointer to the motion tracker made from either physical device or simulator.

It will be dynamically reconstructed if a failure message from the physical sensor is received.

osg::ref_ptr<SmartMotion::CARTrackerCallback>

SmartForeman::CARSensorForeman::m_pMVRetrieveCallback

protected

Callback to update the transformation node based on the modelview matrix.

It will be dynamically reconstructed if a failure message from the physical sensor is received.

osg::ref_ptr<SmartMotion::CARTrackerCallback>

SmartForeman::CARSensorForeman::m_pMVUpdateCallback

protected

Callback to update the modelview matrix.

It will be dynamically reconstructed if a connection failure message from the physical sensor is received.

SmartForeman::CARSiteForeman Class Reference

Public Member Functions

	CARSiteForeman (const char *szARConfigFile)
	A constructor that initializes with default scene/graph/controller provided by SMART..
	CARSiteForeman (SmartSite::CARScene *pScene, SmartSite::CARGraph *pGraph, SmartSite::CARController *pCtrl)
	A constructor that assembles the scene/graph/controller's children class.
virtual	~CARSiteForeman ()
	A default destructor.
virtual void	Run ()
	Launch the scene/graph/controller.
SmartSite::CARGraph *	GetGraph ()
SmartSite::CARScene *	GetScene ()
SmartSite::CARController *	GetController ()

Protected Attributes

osg::ref_ptr< SmartSite::CARGraph >	m_pGraph
osg::ref_ptr< SmartSite::CARScene >	m_pScene
osg::ref_ptr < SmartSite::CARController >	m_pController

Detailed Description

CARSensorForeman initiates CARLocation and CAROrientation based on the sensor configuration, and builds them into the CARMotionTracker which are integrated into the CARTrackerCallback. It also handles the situation when the connection with the physical sensors status changes

SmartMotion Namespace Reference

Classes

class **CARMotionTracker**

CARMotionTracker computes the modelview transformation matrix based on the geographical location and orientation. [More...](#)

class **CARCalibration**

class **CARTrackerCallback**

Update the modelview rotation matrix and objects' location. [More...](#)

Functions

osg::Vec3 **InterpDisVec** (const osg::Vec3 Pt1, const osg::Vec3 Pt2)

Returns the offset VECTOR between two points.

double **InterpDis** (const osg::Vec3 Pt1, const osg::Vec3 Pt2)

Returns the DISTANCE between two points.

osg::Vec3d **InterpUnitDis** (osg::Vec3d GlobalPoint)

Return the unit distance based on the given object global/user position.

Detailed Description

SmartMotion contains **CARMotionTracker** that yields modelview transformation matrix based on the readings provided by the CARLocation and CAROrientation; **CARTrackerCallback** that updates the transform node. Vincenty that converts geographical location to metric location; **CARCalibration** that generates the virtual camera's projection matrix based on the camera calibration results.

Function Documentation

```
SMARTMOTION_API double SmartMotion::InterpDis ( const osg::Vec3 Pt1,
                                                const osg::Vec3 Pt2
                                                )
```

Returns the DISTANCE between two points.

Parameters

Pt1 Equivalent to the User position.

Pt2 Equivalent to the Object Position.

```
SMARTMOTION_API osg::Vec3 SmartMotion::InterpDisVec ( const osg::Vec3 Pt1,  
                                                    const osg::Vec3 Pt2  
                                                    )
```

Returns the offset VECTOR between two points.

Parameters

Pt1 Equivalent to the User position

Pt2 Equivalent to the Object Position

SmartMotion::CARMotionTracker Class Reference

CARMotionTracker computes the modelview transformation matrix based on the geographical location and orientation. [More...](#)

Public Member Functions

	CARMotionTracker (CARLocation *pARLocn, CAROrientation *pAROritn)
	Construct a CARMotionTracker with instances of CARLocation and CAROrientation.
	~CARMotionTracker (void)
	A default destructor.
void	GetOritn (osg::Vec3 &vOritn, struct tm &oritnTimestamp)
	Get orientation and its associated timestamp.
void	GetGeoLocn (osg::Vec3 &vGeoLocn, struct tm &locnTimestamp)
	Get geographical location and its associated timestamp.
const osg::Quat &	GetModelViewQuat ()
	The modelview quaternion that expresses the relative rotation from the eye coordinate system to the world coordinate system.
virtual void	update (osg::NodeVisitor *nv)
	Called during the OSG update traversal to retrieve the geographical location and the orientation, and compute the modelview quaternion.

Protected Attributes

osg::Vec3	m_vOritn	The latest retrieved yaw, pitch and roll.
osg::Vec3	m_vGeoLocn	The latest retrieved geographical location.
struct tm	m_sOritnTimestamp	The timestamp associated with the latest orientation readings.
struct tm	m_sLocnTimestamp	The timestamp associated with the latest geographical location readings.
osg::Quat	m_mvTransform	
osg::ref_ptr< CARLocation >	m_pARLocn	Pointer to the CARLocation instance.
osg::ref_ptr< CAROrientation >	m_pAROritn	Pointer to the CAROrientation instance.

Detailed Description

CARMotionTracker computes the modelview transformation matrix based on the geographical location and orientation.

Member Function Documentation

```
void SmartMotion::CARMotionTracker::GetGeoLocn ( osg::Vec3 & vGeoLocn,  
                                                struct tm & locnTimestamp  
                                                )
```

inline

Get geographical location and its associated timestamp.

If the connection with the physical device breaks down, then the latest geographical location readings will be fed into the simulator as the initial value.

Parameters

vGeoLocn Stores longitude, latitude and altitude in sequence.

timestamp Stores UTC year, month, day in a month, hour, minute, and second.

```
void SmartMotion::CARMotionTracker::GetOritn ( osg::Vec3 & vOritn,  
                                               struct tm & oritnTimestamp  
                                               )
```

inline

Get orientation and its associated timestamp.

If the connection with the physical device breaks down, then the latest orientation readings is fed into the simulator as the initial value.

Parameters

vOritn Stores yaw, pitch, and roll in sequence.

timestamp Stores UTC year, month, day in a month, hour, minute, and second.

Member Data Documentation

```
osg::Quat SmartMotion::CARMotionTracker::m_mvTransform
```

protected

The modelview quaternion that expresses the relative rotation from the eye coordinate system to the world coordinate system.

SmartMotion::CARCalibration Class Reference

Public Member Functions

CARCalibration (const std::string &calibFilename)

Constructor loads the camera calibration file.

bool **LoadCalibrationMatrix** (const std::string &filename)

Load the camera calibrated intrinsic matrix.

void **SetProjection** (float near, float far, float width, float height)

Compute the OpenGL projection matrix based on the near, far plane and viewport size.

void **GetProjectionMat** (osg::Matrix &projMat)

Get the OpenGL projection matrix.

Protected Attributes

cv::Mat **m_camMat**

The calibrated physical camera intrinsic matrix.

cv::Mat **m_distCoeffs**

The calibrated physical camera distortion model.

osg::Matrix **m_projectionMat**

OpenGL projection matrix computed based on the physical camera matrix and the OpenGL camera near, far plane setting, and the viewport size.

Detailed Description

CARCalibration loads the camera calibration results (the K matrix), and computes the virtual camera's projection matrix.

Member Function Documentation

bool SmartMotion::CARCalibration::LoadCalibrationMatrix (const std::string & filename)

Load the camera calibrated intrinsic matrix.

Parameters

filename OpenCV calibration file that contains the intrinsic matrix.

Returns

Whether the intrinsic matrix has been successfully loaded from the file.

SmartMotion::CARTrackerCallback Class Reference

Update the modelview rotation matrix and objects' location. [More...](#)

Public Member Functions

CARTrackerCallback::CARTrackerCallback (**CARMotionTracker** *pTracker, std::map< std::string, osg::ref_ptr< **SmartCrew::CARObject** >> &objMap)

Constructor to create a **CARTrackerCallback** associated with a tracker.

Static Public Member Functions

static **CARTrackerCallback** * **CARTrackerCallback::addOrSet** (osg::Node *node, **CARMotionTracker** *pTracker, std::map< std::string, osg::ref_ptr< **SmartCrew::CARObject** >> &objMap)

Factory method to create a new **CARTrackerCallback**.

Protected Member Functions

void **updateObjGlobalLoen** ()

Update the the CARObject's global location.

virtual void **operator()** (osg::Node *node, osg::NodeVisitor *nv)

Protected Attributes

osg::ref_ptr< **CARMotionTracker** > **m_pTracker**

Pointer to the updated motion tracker.

int **m_nFrameNumber**

Only update the motion tracker if the frame number changes.

std::map< std::string, osg::ref_ptr
< **SmartCrew::CARObject** >> & **m_objMap**

Reference to the CARObject map to update its metric/global position relative to the user.

Detailed Description

Update the modelview rotation matrix and objects' location.

One instance of the TrackerCallback attached to the root node invokes the tracker to calculate the modelview rotation. Another instance attached to the Transformation node, modifies the transformation's rotation and updates the objects' global locations.

Member Function Documentation

virtual void

```
SmartMotion::CARTrackerCallback::operator() ( osg::Node * node,  
                                              osg::NodeVisitor * nv  
                                              )
```

`inline` `protected` `virtual`

Virtual method, which is called on every updated frame. Depending on the node type, it either updates the modelview quaternion, or updates the transform node.

SmartSensor Namespace Reference

Classes

class **CARLocation**

class **CAROrientation**

struct **CDeviceDelegate**

Delegate to notify the unexpected disconnection of physical location or orientation tracking device. More...

class **CARCompassSimulator**

class **CARGPSReader**

class **CARGPSSimulator**

class **CARTCMReader**

class **timer**

Tracking time ticking. More...

Typedefs

typedef unsigned char **UInt8**

typedef float **Float32**

typedef unsigned short int **UInt16**

typedef unsigned long int **UInt32**

Enumerations

```
enum {  
    kGetModInfo = 1, kModInfoResp, kSetDataComponents, kGetData,  
    kDataResp, kHeading = 5, kTemperature = 7, kPCalibrated = 21,  
    kRCalibrated, kIZCalibrated, kPAngle, kRAngle  
}
```

```
enum { kBufferSize = 512, kPacketMinSize = 5 }
```

Functions

void **OnEventCharReceivedFunc** (Object^sender,
PortController::EventCharReceivedEventArgs^e)
Event handler function to handle incoming data stream.

void **DotNetStringToStdString** (System::String^s, std::string &os)
Convert from .NET String to std::string.

void **AcqCurrentUTCTimeStamp** (struct tm ×tamp)
Retrieve the current timestamp in UTC time zone.

std::ostream & **operator<<** (std::ostream &os, **timer** &t)
Overloading operator Allow timers to be printed to ostreams using the syntax 'os << t'.

Variables

UInt8	mOutData	[kBufferSize]
UInt8	mInData	[kBufferSize]
UInt16	mExpectedLen	
const UInt8	kDataCount	= 4
timer	t	
double	mResponseTime	
double	mTime	
UInt32	mStep	
bool	tracker_loop	

Detailed Description

SmartSensor contains accessing methods for location or orientation data, the source can be GPS, compass, or simulator.

Function Documentation

```
std::ostream& SmartSensor::operator<< ( std::ostream & os,  
                                        timer & t  
                                        )
```

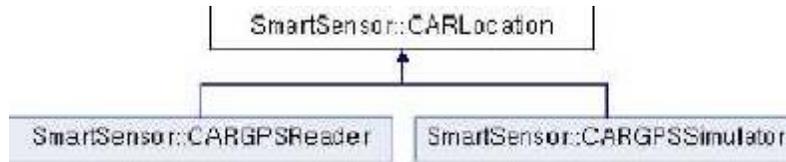


Overloading operator Allow timers to be printed to ostreams using the syntax 'os << t'.

For an ostream 'os' and a timer 't'. For example, "cout << t" will print out the total amount of time 't' has been "running".

SmartSensor::CARLocation Class Reference

Inheritance diagram for SmartSensor::CARLocation:



Public Member Functions

virtual void **GetGeoLocn** (osg::Vec3 &vGeoLocn, struct tm ×tamp)=0

Get the geographical location and its associated timestamp.

virtual int **GetGPSQuality** ()=0

Accessing geographical data quality.

Static Public Attributes

static osg::ref_ptr< struct

CDeviceDelegate > **m_statusDelegate** = NULL

Delegate for handling device connection status changes.

Protected Member Functions

CARLocation ()

The constructor is protected, so the class cannot be initialized.

Detailed Description

Abstract interface for accessing the geographical location and the data quality.

Member Function Documentation

```
virtual void SmartSensor::CARLocation::GetGeoLocn ( osg::Vec3 & vGeoLocn,  
                                                    struct tm & timestamp  
                                                    )
```

pure virtual

Get the geographical location and its associated timestamp.

Parameters

vGeoLocn Stores longitude, latitude and altitude in sequence.

timestamp Stores UTC year, month, day in a month, hour, minute, and second.

Implemented in **SmartSensor::CARGPSReader**, and **SmartSensor::CARGPSSimulator**.

```
virtual int SmartSensor::CARLocation::GetGPSQuality ( )
```

pure virtual

Accessing geographical data quality.

Fix quality: 0 = invalid 1 = GPS fix (SPS) 2 = DGPS fix 3 = PPS fix 4 = Real Time Kinematic 5 = Float RTK 6 = estimated (dead reckoning) (2.3 feature) 7 = Manual input mode 8 = Simulation mode

Implemented in **SmartSensor::CARGPSReader**, and **SmartSensor::CARGPSSimulator**.

Member Data Documentation

```
osg::ref_ptr< struct CDeviceStatusDelegate >  
SmartSensor::CARLocation::m_statusDelegate = NULL
```

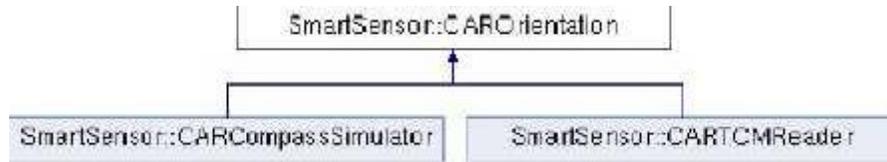
static

Delegate for handling device connection status changes.

Even though this won't be used by simulator, it sits in the base class for universal interface. If a third party inherits from this class, it makes their life easier.

SmartSensor::CAROrientation Class Reference

Inheritance diagram for SmartSensor::CAROrientation:



Public Member Functions

virtual void **GetOritn** (osg::Vec3 &vOritn, struct tm ×tamp)=0

Get orientation and its associated timestamp.

Public Attributes

osg::ref_ptr< struct

CDeviceDelegate > **m_statusDelegate**

Delegate for handling device connection status changes.

Protected Member Functions

CAROrientation ()

The constructor is protected, so the class cannot be initialized.

Detailed Description

Abstract interface for accessing the geographical location and the data quality.

Member Function Documentation

```
virtual void SmartSensor::CAROrientation::GetOritn ( osg::Vec3 & vOritn,  
                                                    struct tm & timestamp  
                                                    )
```

pure virtual

Get orientation and its associated timestamp.

Parameters

vOritn Stores yaw, pitch, and roll in sequence.

timestamp Stores UTC year, month, day in a month, hour, minute, and second.

Implemented in **SmartSensor::CARTCMReader**, and **SmartSensor::CARCompassSimulator**.

Member Data Documentation

```
osg::ref_ptr<struct CDeviceDelegate> SmartSensor::CAROrientation::m_statusDelegate
```

Delegate for handling device connection status changes.

Even though this won't be used by simulator, it sits in the base class for universal interface. If a third party inherits from this class, it makes their life easier.

SmartSensor::CDeviceDelegate Struct Reference

Delegate to notify the unexpected disconnection of physical location or orientation tracking device.
More...

Public Member Functions

virtual void **PhyLocnDeviceDisabled** (const std::string &msg)

Handle the emergency of location tracking device disconnection, it usually calls the corresponding simulator.

virtual void **PhyLocnDeviceEnabled** (const std::string &msg)

Handle the location device back online (the quality reaches the preferred quality).

virtual void **PhyOritnDeviceDisabled** (const std::string &msg)

Handle the emergency of orientation tracking device disconnection, it usually calls the corresponding simulator.

virtual void **PhyOritnDeviceEnabled** (const std::string &msg)

Handle the orientation device back online.

Detailed Description

Delegate to notify the unexpected disconnection of physical location or orientation tracking device.

Member Function Documentation

virtual void

SmartSensor::CDeviceDelegate::PhyLocnDeviceDisabled (const std::string & **msg**) [inline](#) [virtual](#)

Handle the emergency of location tracking device disconnection, it usually calls the corresponding simulator.

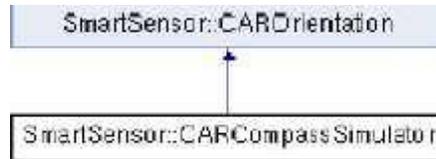
It is not necessary totally disconnected, but maybe the GPS quality drops below the preferred threshold.

See Also

[m_nPreferredGPSQuality](#)

SmartSensor::CARCompassSimulator Class Reference

Inheritance diagram for SmartSensor::CARCompassSimulator:



Public Member Functions

CARCompassSimulator ()

A default constructor.

CARCompassSimulator (const osg::Vec3 &vOritn)

A constructor sets up the simulated orientation.

~CARCompassSimulator (void)

A default destructor.

virtual void **GetOritn** (osg::Vec3 &vOritn, struct tm ×tamp)

Get the orientation.

void **SetOritn** (const osg::Vec3 &vOritn)

void **GetYaw** (float &yaw, struct tm ×tamp)

void **GetPitch** (float &pitch, struct tm ×tamp)

void **GetRoll** (float &roll, struct tm ×tamp)

void **SetYaw** (float yaw)

void **SetPitch** (float pitch)

void **SetRoll** (float roll)

Protected Attributes

osg::Vec3 **m_vOritn**

Orientation as yaw, pitch, and roll.

Additional Inherited Members

▀ **Public Attributes inherited from SmartSensor::CAROrientation**

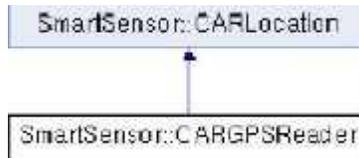
▀ **Protected Member Functions inherited from SmartSensor::CAROrientation**

Detailed Description

CARCompassSimulator inherits from **CAROrientation**, and accepts the user control and outputs the simulated orientation readings.

SmartSensor::CARGPSReader Class Reference

Inheritance diagram for SmartSensor::CARGPSReader:



Public Member Functions

CARGPSReader (std::string strPortName, int nBaudrate)

Constructor that initializes the communication with GPS.

~CARGPSReader (void)

Destructor that shuts down the communication with GPS.

virtual void **GetGeoLocn** (osg::Vec3 &vGeoLocn, struct tm ×tamp)

Get the geographical location.

int **GetGPSQuality** ()

Get the GPS data quality which is deciphered from GPS incoming data stream.

Static Public Member Functions

static void **SetPreferredGPSQuality** (int preferredGPSQuality)

static int **GetPreferredGPSQuality** ()

static bool **GetGPSHibernate** ()

Protected Member Functions

bool **Initialize** (std::string strPortName, int nBaudrate)

Set the serial port parameters and initialize the communication with the GPS device.

void **Shutdown** ()

Shutdown GPS serial port connection.

► **Protected Member Functions inherited from SmartSensor::CARLocation**

Protected Attributes

groot< SciCom::PortController^> **m_pGPSPort**

Object to communicate with GPS through the serial port.

Static Protected Attributes

static groot< System::String^> **m_strLon** = "00.00 E"

static groot< System::String^> **m_strLat** = "00.00 N"

static groot< System::String^> **m_strAlt** = "00.00 M"

static gcroot< System::String^> **m_strTime** = "00:00:00"

static gcroot< System::String^> **m_strFixQuality** = "0"

GPS data quality.

static int **m_nPreferredGPSQuality** = 2

static bool **m_bGPSHibernate** = false

Whether the GPS is active to supply data.

static gcroot< System::String^> **m_strNumSatellite** = "0"

Number of satellites being tracked.

Friends

void **OnEventCharRevFunc** (System::Object^sender, EventCharReceivedEventArgs^e)

Event handler function to handle incoming data stream.

Additional Inherited Members

▮ Static Public Attributes inherited from **SmartSensor::CARLocation**

Detailed Description

CARGPSReader inherits from **CARLocation**, and provides interface for accessing GPS readings, timestamp and data quality.

Member Function Documentation

int SmartSensor::CARGPSReader::GetGPSQuality ()

inline **virtual**

Get the GPS data quality which is deciphered from GPS incoming data stream.

Fix quality: 0 = invalid 1 = GPS fix (SPS) 2 = DGPS fix 3 = PPS fix 4 = Real Time Kinematic 5 = Float RTK 6 = estimated (dead reckoning) 7 = Manual input mode 8 = Simulation mode

Implements **SmartSensor::CARLocation**.

Friends And Related Function Documentation

```
void OnEventCharRevFunc ( System::Object^ sender,
                        EventCharReceivedEventArgs^ e
                        )
```

friend

Event handler function to handle incoming data stream.

The incoming data stream ending in \n (for use with NMEA GPS only).

Member Data Documentation

```
int SmartSensor::CARGPSReader::m_nPreferredGPSQuality = 2
```

static protected

The required GPS Quality specified by the user.

If the GPS quality drops below the preferred quality, notification will be thrown, and the GPS will be put into 'hibernation'. Once the GPS quality reaches preferred standard again, it will reactivate itself and throw messages.

See Also

[m_strFixQuality](#)

```
gcroot< String^> SmartSensor::CARGPSReader::m_strFixQuality = "0"
```

static protected

GPS data quality.

Fix quality: 0 = invalid 1 = GPS fix (SPS) 2 = DGPS fix 3 = PPS fix 4 = Real Time Kinematic 5 = Float RTK 6 = estimated (dead reckoning) 7 = Manual input mode 8 = Simulation mode

```
gcroot< String^> SmartSensor::CARGPSReader::m_strNumSatellite = "0"
```

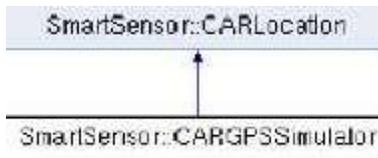
static protected

Number of satellites being tracked.

When the # of satellite is below 4, then the rover has lost the track of GPS.

SmartSensor::CARGPSSimulator Class Reference

Inheritance diagram for SmartSensor::CARGPSSimulator:



Public Member Functions

CARGPSSimulator ()

A default constructor.

CARGPSSimulator (const osg::Vec3 &vGeoLocn)

A constructor initializes with the specified location.

~CARGPSSimulator (void)

A default destructor.

virtual void **GetGeoLocn** (osg::Vec3 &vGeoLocn, struct tm ×tamp)

Get the geographical location.

void **SetGeoLocn** (const osg::Vec3 &vGeoLocn)

void **GetGeoLon** (float &lon, struct tm ×tamp)

void **GetGeoLat** (float &lat, struct tm ×tamp)

void **GetGeoAlt** (float &alt, struct tm ×tamp)

void **SetGeoLon** (float lon)

void **SetGeoLat** (float lat)

void **SetGeoAlt** (float alt)

virtual int **GetGPSQuality** ()

The quality is always 8 (simulation mode)

Protected Attributes

osg::Vec3 **m_vGeoLocn**

Geographical location as longitude, latitude, and altitude.

int **m_nFixQuality**

For **CARGPSSimulator**, the quality should be 8, which means in simulation mode.

Additional Inherited Members

▀ Static Public Attributes inherited from **SmartSensor::CARLocation**

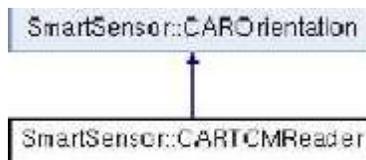
▀ Protected Member Functions inherited from **SmartSensor::CARLocation**

Detailed Description

CARGPSSimulator inherits from **CARLocation**, and accepts the user control and outputs the simulated GPS location.

SmartSensor::CARTCMReader Class Reference

Inheritance diagram for SmartSensor::CARTCMReader:



Public Member Functions

CARTCMReader (std::string strPortName, int nBaudrate)

A constructor initializes the communication with the TCM device.

virtual **~CARTCMReader** ()

A default destructor.

virtual void **GetOrirn** (osg::Vec3 &vOrirn, struct tm ×tamp)

Get the orientation and its associated timestamp.

bool **GetConnectSuccess** ()

Whether the connection with TCM device has been successfully established.

Protected Member Functions

bool **Initialize** (std::string strPortName, int nBaudrate)

Set the serial port parameters and initialize the communication with TCM device.

void **Shutdown** ()

Shutdown the TCM serial port connection.

▀ **Protected Member Functions inherited from SmartSensor::CAROrientation**

Protected Attributes

osg::Vec3 **m_orirn**

Orientation as yaw, pitch and roll.

struct tm **m_sTimestamp**

bool **m_bConnectSuccess**

Whether successfully connected with the TCM device.

groot< SciCom::PortController^> **m_pTCMPort**

Private Member Functions

UInt16 **CRC** (void *data, UInt32 len)

Calculate CRC-16.

void **CARTCMReader::TCMSend** (UInt8 frameType, void *dataPtr, UInt16 dataLen)

Monitor the timer and call SendData.

void **SendData** (Uln8 frameType, void *dataPtr, Uln32 len)

Takes a frame and data and puts that information into a TCM Protocol packet to send.

void **CARTCMReader::ParseTCMPacket** (Uln8 frameType, void *dataPtr, Uln16 dataLen)

Called by ReceivePacket when a complete frame is correctly received.

void **ReceivePacket** ()

Check the completeness of the received package, and stare parsing if complete.

void **TCMControl** ()

Extract the payload piece by piece.

virtual void **run** ()

Thread's run method. Must be implemented by derived classes.

Additional Inherited Members

▀ Public Attributes inherited from **SmartSensor::CAROrientation**

Detailed Description

CARTCMReader is an instance of **CAROrientation** that retrieves orientation data from the TCM device through the serial port.

Constructor & Destructor Documentation

```
SmartSensor::CARTCMReader::CARTCMReader ( std::string strPortName,  
                                           int nBaudrate  
                                           )
```

A constructor initializes the communication with the TCM device.

GetConeectSuccess can be called after this to test the establishment of the connection after this.

Member Function Documentation

```
void  
SmartSensor::CARTCMReader::CARTCMReader::ParseTCMPacket ( UInt8 frameType,  
                                                         void * dataPtr,  
                                                         UInt16 dataLen  
                                                         )
```

private

Called by ReceivePacket when a complete frame is correctly received.

Returns

Whether the data was successfully parsed.

```
void SmartSensor::CARTCMReader::ReceivePacket ( )
```

private

Check the completeness of the received package, and start parsing if complete.

Returns

Whether the data was successfully received and parsed.

```
void SmartSensor::CARTCMReader::run ( )
```

private virtual

Thread's run method. Must be implemented by derived classes.

This is where the action happens. The action includes sending data request to TCM, receiving data package, checking and parsing it.

SmartSensor::timer Class Reference

Tracking time ticking. [More...](#)

Public Member Functions

void **start** (const char *msg=0)

Start a timer. If it is already running, let it continue running.

void **restart** (const char *msg=0)

turn the timer off and start it again from 0

void **stop** (const char *msg=0)

Stop the timer.

void **check** (const char *msg=0)

Print out the current timer timing.

double **elapsed_time** ()

Return the total time that the timer has been in the "running" state since it was first "started" or last "restarted".

Private Attributes

bool **running**

clock_t **start_clock**

time_t **start_time**

double **acc_time**

Friends

std::ostream & **operator<<** (std::ostream &os, **timer** &t)

Overloading operator Allow timers to be printed to ostream using the syntax 'os << t'.

Detailed Description

Tracking time ticking.

Member Function Documentation

void SmartSensor::timer::check (const char * msg = 0) inline

Print out the current timer timing.

Parameters

msg Print an optional message.

double SmartSensor::timer::elapsed_time () inline

Return the total time that the timer has been in the "running" state since it was first "started" or last "restarted".

For "short" time periods (less than an hour), the actual CPU time used is reported instead of the elapsed time.

void SmartSensor::timer::restart (const char * msg = 0) inline

Turn the timer off and start it again from 0

Parameters

msg Print an optional message.

void SmartSensor::timer::start (const char * msg = 0) inline

Start a timer. If it is already running, let it continue running.

Parameters

msg Print an optional message.

void SmartSensor::timer::stop (const char * msg = 0) inline

Stop the timer.

Parameters

msg Print an optional message.

```
std::ostream& operator<< ( std::ostream & os,  
                           timer & t  
                           )
```

friend

Overloading operator Allow timers to be printed to ostream using the syntax 'os << t'.

For an ostream 'os' and a timer 't'. For example, "cout << t" will print out the total amount of time 't' has been "running".

SmartSite Namespace Reference

Classes

class **CARController**

struct **ARCameraConfig**

Virtual and real video camera settings used in the augmented reality. [More...](#)

class **CARGraph**

class **CARScene**

Detailed Description

SmartSite is where the virtual content (**SmartCrew**) and the real scene (**SmartVideo**) is shown, and it is like a `modelviewController`. **CARScene** is in charge of archiving and maintaining the virtual content. **CARGraph** builds the viewer to show the virtual content and the real scene. User input received by **CARController** is fed to **CARScene** or **CARGraph** to update the virtual content.

SmartSite::CARController Class Reference

Public Member Functions

CARController (void)

A default constructor.

~CARController (void)

A default destructor.

void **AddtoViewer** (osgViewer::Viewer *viewer)

Attach the GUIEventHandler to a viewer to update.

Protected Member Functions

virtual bool **handle** (const osgGA::GUIEventAdapter &ea, osgGA::GUIActionAdapter &aa)

Handle the keyboard/mouse event.

Protected Attributes

osg::ref_ptr< osgViewer::Viewer > **m_pViewer**

The GUIEventHandler is attached to the viewer for updating.

Detailed Description

CARController accepts the user keyboard/mouse control and invokes registered delegate to handle the event

SmartSite::ARCameraConfig Struct Reference

Virtual and real video camera settings used in the augmented reality. [More...](#)

Public Attributes

int	arEnabled
int	occlusionEnabled
double	fzNear
double	fzFar
int	windowX
int	windowY
int	windowWidth
int	windowHeight
int	camType
double	GLSLandMap
std::string	virtualCamConfig
std::string	rgbCamConfig
std::string	tofCamConfig
std::string	stereoCamConfig
int	windowCropX
double	windowCropY
double	windowCropWidth
double	windowCropHeight

Detailed Description

Virtual and real video camera settings used in the augmented reality.

SmartSite::CARGraph Class Reference

Public Member Functions

	CARGraph (CARScene *pScene, std::string szARConfigFile)
virtual	~CARGraph (void) A default destructor.
virtual void	OnInitialUpdate ()
virtual void	Run () Launch the viewer.
SmartSite::CARScene *	GetScene () const Get a pointer to the graph's CARScene .
void	SetScene (CARScene *pScene) Set a pointer to the graph's CARScene .
osg::Node *	GetGraphNode () Get a pointer to the graph's root node.
osg::Node *	GetGraphPositionAttitudeNode () Get a pointer to the graph's position/attitude transform node.
osgViewer::Viewer *	GetGraphViewer () Get a pointer to the viewer.
SmartVideo::CARVideoSource *	GetRGBImage () Get a pointer to the graph's RGB image.
SmartVideo::CARVideoSource *	GetDepthImage () Get a pointer to the graph's depth image.

Protected Member Functions

osg::Group *	CreateImageBackground (string stereoConfigPath, osg::Image *videoimage, osg::Image *depthImage) Use the video resource to create a video layer as the background.
osg::Node *	CARGraph::AddLight (osg::StateSet *rootStateSet) Add light to the scene.

Protected Attributes

osg::ref_ptr< SmartSite::CARScene >	m_pScene
osg::ref_ptr	
< osg::PositionAttitudeTransform >	m_pHeadRotate
osg::ref_ptr< osgViewer::Viewer >	m_pViewer
	A osgViewer that shows and updates both the real background and the virtual content.

osg::ref_ptr < SmartVideo::CARVideoSource >	m_pRGBImage	Pointer to the RGB image.
osg::ref_ptr < SmartVideo::CARVideoSource >	m_pDepthImage	Pointer to the TOF image.
osg::ref_ptr < SmartVideo::CARVideoGeode >	m_videoGeode	Quad that contains the RGB and TOF image, as well as the registration algorithm.
osg::ref_ptr< osg::Group >	m_pRoot	
std::string	m_strARConfigFile	Augmented reality video configuration file name.
ARCameraConfig	m_arCamConfig	Virtual and real camera settings.
osg::ref_ptr < SmartMotion::CARCalibration >	m_pCalib	Calibration that yields the projection matrix of the main camera.

Detailed Description

CARGraph creates a viewer to show the virtual and real background content. It creates a perspective camera to show the virtual content, and an ortho camera to show the real background. The projection matrix of the perspective camera is the same of the real physical camera to make sure the virtual content is aligned with the real content.

Constructor & Destructor Documentation

```
SmartSite::CARGraph::CARGraph ( CARScene * pScene,
                                std::string  szARConfigFile
                                )
```

Initialize the viewer for showing the virtual and background content. The virtual and background content are underneath perspective and ortho camera respectively.

Member Function Documentation

```
osg::Group *  
SmartSite::CARGraph::CreateImageBackground ( string stereoConfigPath,  
                                             osg::Image * videoimage,  
                                             osg::Image * depthImage  
                                             )
```

protected

Use the video resource to create a video layer as the background.

The video layer is essentially an ortho camera that shows the RGB video content in the color buffer. If occlusion enabled, then a depth map is also written to the depth buffer.

```
void SmartSite::CARGraph::OnInitialUpdate ( )
```

virtual

Set up the viewer and the perspective camera's projection matrix and viewport for showing the virtual content, as well as the ortho camera for showing the video background content and enabling the occlusion effect if any.

Member Data Documentation

```
osg::ref_ptr<osg::PositionAttitudeTransform> SmartSite::CARGraph::m_pHeadRotate
```

protected

It decides the modelview transformation so that virtual content can be aligned with the real background. This is updated by a CARMotionTracker callback. Only the attitude component is updated since the world origin is always at the same point as the viewer. Therefore the CARObject's positions are updated in the world coordinate system.

```
osg::ref_ptr<osg::Group> SmartSite::CARGraph::m_pRoot
```

protected

The root node in the scene graph, to which a CARMotionTracker callback is attached. The CARMotionTracker updates the modelview matrix's attitude with the latest location and orientation readings.

```
osg::ref_ptr<SmartSite::CARScene> SmartSite::CARGraph::m_pScene
```

protected

The pointer to the **CARScene** that archives the CARObject and the CARClass. The **CARScene** inherits from that **osg::Group** so that the **CARScene** can be attached to the viewer to show the virtual content.

SmartSite::CARScene Class Reference

Public Member Functions

	CARScene (void)	A default constructor.
virtual	~CARScene (void)	A default destructor.
virtual void	OnNewScene ()	Maps to the function when a new scene is created.
virtual void	OnOpenScene (const char *lpPathName)	Maps to the function when a scene is opened.
virtual void	OnSaveScene ()	Maps to the function when the scene is saved.
virtual void	OnCloseScene ()	Maps to the function when the scene is closed.
void	OnFileClose ()	Maps to the function when close the file.
void	OnFileSave ()	Maps to the function when save the file.
void	OnFileSaveAs ()	Maps to the function when save another copy of the file.
virtual bool	DoSave (const char *szPathName, bool bReplace=true)	Save the scene data to a file.
virtual bool	DoFileSave ()	Invoked by OnFileSave() .
std::map< std::string, osg::ref_ptr < SmartCrew::CARClass > > &	GetClassNameToPtrMap ()	
std::map< std::string, osg::ref_ptr < SmartCrew::CARObject > > &	GetObjectToPtrMap ()	
SmartCrew::CARObject *	GetSceneObject (std::string strObjName) const	Get the pointer to the CARObject by its name.
bool	SetSceneObject (std::string szObjName, SmartCrew::CARObject *pNewObj)	Add a new CARObjec to the map.
SmartCrew::CARClass *	GetObjClass (std::string strClsName)	Get the pointer to the CARClass by its name.

```
bool SetObjectClass (std::string strClsName, SmartCrew::CARClass
*pNewCls)
```

Add a new CARClass to the map.

Protected Attributes

```
std::map< std::string,
osg::ref_ptr
< SmartCrew::CARClass > > m_mpClassNameToClassPtr
```

```
std::map< std::string,
osg::ref_ptr
< SmartCrew::CARObject > > m_mpObjectNameToObjectPtr
```

Detailed Description

CARSence is the librarian of the virtual content, it archives the virtual content in two ways. Firstly, CARClass and CARObject are achieved in the map after being initialized. Secondly, once the CARObject is placed in the viewer, it is also attached to the OSG scenegraph.

Member Data Documentation

```
std::map<std::string, osg::ref_ptr<SmartCrew::CARClass> >  
SmartSite::CARScene::m_mpClassNameToClassPtr
```

protected

The map index is to find the CARClass with its name. Every CARClass needs to be archived into the map, when it is created.

```
std::map<std::string, osg::ref_ptr<SmartCrew::CARObject> >  
SmartSite::CARScene::m_mpObjectNameToObjectPtr
```

protected

The map index is to find the CARObject with its name. Every CARObject needs to be archived into the map, when it is created.

SmartVideo Namespace Reference

Classes

class [CARPGRSource](#)

class [CARRGBSource](#)

class [CARTOFSource](#)

class [CARVideoCamera](#)

struct [VideoConfiguration](#)

The web camera setting information, like size, pixel format, device factory configuration.
More...

class [CARVideoGeode](#)

class [CARVideoSource](#)

Enumerations

enum [PixelFormatType](#) {
 VIDEOFORMAT_RGB555 = 0, VIDEOFORMAT_RGB565, VIDEOFORMAT_RGB24,
 VIDEOFORMAT_BGR24,
 VIDEOFORMAT_RGBA32, VIDEOFORMAT_BGRA32, VIDEOFORMAT_ARGB32,
 VIDEOFORMAT_ABGR32,
 VIDEOFORMAT_Y8, VIDEOFORMAT_Y16, VIDEOFORMAT_YUV444,
 VIDEOFORMAT_YUV422,
 VIDEOFORMAT_YUV422P, VIDEOFORMAT_YUV411, VIDEOFORMAT_YUV411P,
 VIDEOFORMAT_YUV420P,
 VIDEOFORMAT_YUV410P, VIDEOFORMAT_GREY8, VIDEOFORMAT_MJPEG,
 VIDEOFORMAT_422YpCbCr8,
 VIDEOFORMAT_422YpCbCr8R, VIDEOFORMAT_ANY
}

Pixel Format type.

enum [FrameRateType](#) {
 VIDEOFramerate_1_875, VIDEOFramerate_3_75, VIDEOFramerate_5,
 VIDEOFramerate_7_5,
 VIDEOFramerate_10, VIDEOFramerate_15, VIDEOFramerate_30,
 VIDEOFramerate_PAL,
 VIDEOFramerate_NTSC, VIDEOFramerate_50, VIDEOFramerate_60,
 VIDEOFramerate_120,
 VIDEOFramerate_240, VIDEOFramerate_ANY, VIDEOFramerate_MAX
}

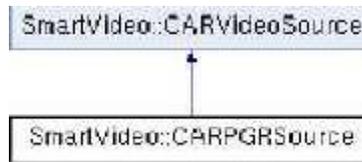
Framerate type - define frame rate of the video flux.

Detailed Description

SmartVideo creates an ortho camera that shows quad geometry with video textures pasted on it. The RGB texture serves as the AR background, and the depth texture, if any, resolves the occlusion issue. Several registration methods have been included: 1) The direct mapping between intensity and depth image from TOF camera; 2) The homography mapping between the RGB and depth image; 3) The 3D mapping between the RGB and depth image.

SmartVideo::CARPGRSource Class Reference

Inheritance diagram for SmartVideo::CARPGRSource:



Public Member Functions

CARPGRSource (**SourceMode** srcMode, const char *szCamereConfigFile)

A constructor that loads the camera calibration information.

~**CARPGRSource** (void)

A default destructor that shuts down the PGR camera.

CARPGRSource & **operator=** (const **CARPGRSource** &)

Copy operator.

virtual void **open** ()

virtual void **close** ()

virtual void **pause** ()

virtual void **play** ()

virtual void **update** (osg::NodeVisitor *nv)

Update the video texture by grabbing an image from the video instance.

virtual void **record** (unsigned char *image, size_t width, size_t height, size_t depth, size_t component)

Save one frame of the video stream.

► **Public Member Functions inherited from SmartVideo::CARVideoSource**

Private Member Functions

void **checkError** (const char *function, FlyCapture2::Error error, char *file, size_t line)

Error handling for PGR camera operation.

Private Attributes

FlyCapture2::BusManager **m_busMgr**

FlyCapture2::Image **m_flyImage**

FlyCapture2::Image **m_flyConvertedImage**

It contains the image whose format that can be read by the OpenGL.

FlyCapture2::Camera **m_camera**

Additional Inherited Members

- ▀ Public Types inherited from [SmartVideo::CARVideoSource](#)
- ▀ Protected Member Functions inherited from [SmartVideo::CARVideoSource](#)
- ▀ Protected Attributes inherited from [SmartVideo::CARVideoSource](#)
- ▀ Static Protected Attributes inherited from [SmartVideo::CARVideoSource](#)

Detailed Description

CARPGRSource is a subclass of [CARVideoSource](#) that handles the video input from Point Grey Research camera, and updates the video texture. Point Grey Research camera is an industrial level video camera delivering high quality images.

Constructor & Destructor Documentation

```
SmartVideo::CARPGRSource::CARPGRSource ( SourceMode srcMode,
                                         const char * szCamereConfigFile
                                         )
```

A constructor that loads the camera calibration information.

Parameters

- srcMode** Explains the service function of the camera.
- szCamereConfigFile** The camera calibration filename.

Member Function Documentation

```
void SmartVideo::CARPGRSource::close ( )
```

virtual

Close the video stream. Terminate the connection with the video stream and clean the handle. It is called by the destructor by default.

Implements [SmartVideo::CARVideoSource](#).

void SmartVideo::CARPGRSource::open ()

virtual

Open the PGR video stream. Access the video stream (hardware or file) config and get a handle on it.

Implements **SmartVideo::CARVideoSource**.

virtual void SmartVideo::CARPGRSource::pause ()

inline virtual

Pause the video stream grabbing, and this function is superfluous interface since TOF camera is updated by passively calling.

Implements **SmartVideo::CARVideoSource**.

virtual void SmartVideo::CARPGRSource::play ()

inline virtual

Start the video stream grabbing, and this function is superfluous interface since TOF camera is updated by passively calling.

Implements **SmartVideo::CARVideoSource**.

Member Data Documentation

FlyCapture2::BusManager SmartVideo::CARPGRSource::m_busMgr

private

The BusManager get a PGRGuid for a desired camera or device. Once the camera is found, it can connect to the camera through the camera class.

FlyCapture2::Camera SmartVideo::CARPGRSource::m_camera

private

It represents a physical camera, and must first be connected to using Connect() before any other operations can proceed.

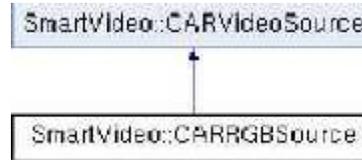
FlyCapture2::Image SmartVideo::CARPGRSource::m_flyImage

private

It retrieves raw images from a camera, convert between multiple pixel formats and save images to disk.

SmartVideo::CARRGBSource Class Reference

Inheritance diagram for SmartVideo::CARRGBSource:



Public Member Functions

CARRGBSource (**SourceMode** srcMode, const char *szCameraConfigFile)

A constructor that loads the camera calibration information.

~**CARRGBSource** (void)

A default destructor that shuts down the web camera.

CARRGBSource & **operator=** (const **CARVideoSource** &)

Copy operator.

virtual void **open** ()

virtual void **close** ()

virtual void **pause** ()

Pause the video stream grabbing.

virtual void **play** ()

Resume the video stream grabbing.

virtual void **update** (osg::NodeVisitor *nv)

Update the video texture by grabbing an image from the video instance.

▀ **Public Member Functions inherited from SmartVideo::CARVideoSource**

Protected Attributes

AR2VideoParamT * **m_pVideo**

Handle to the video camera.

▀ **Protected Attributes inherited from SmartVideo::CARVideoSource**

Additional Inherited Members

▀ **Public Types inherited from SmartVideo::CARVideoSource**

▀ **Protected Member Functions inherited from SmartVideo::CARVideoSource**

▀ **Static Protected Attributes inherited from SmartVideo::CARVideoSource**

Detailed Description

CARRGBSource is a subclass of **CARVideoSource** that handles the video input from normal web camera, and updates the video texture.

Constructor & Destructor Documentation

```
SmartVideo::CARRGBSource::CARRGBSource ( SourceMode srcMode,  
                                           const char * szCameraConfigFile  
                                           )
```

A constructor that loads the camera calibration information.

Parameters

srcMode Explains the service function of the camera.
szCamereConfigFile The camera calibration filename.

Member Function Documentation

```
void SmartVideo::CARRGBSource::close ( )
```

virtual

Close the video stream. Terminate the connection with the video stream and clean the handle.

Implements **SmartVideo::CARVideoSource**.

```
void SmartVideo::CARRGBSource::open ( )
```

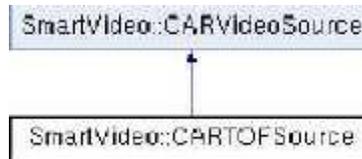
virtual

Open the RGB video stream. Access the video stream (hardware or file) config and get a handle on it.

Implements **SmartVideo::CARVideoSource**.

SmartVideo::CARTOFSource Class Reference

Inheritance diagram for SmartVideo::CARTOFSource:



Public Types

enum **TOFSourceNum** { **SINGLE**, **DOUBLE** }

enum **TOFIOMode** { **REC**, **LIVE** }

Whether the incoming TOF video source is from recorded file or the device lively.

Public Types inherited from SmartVideo::CARVideoSource

Public Member Functions

CARTOFSource (**SourceMode** srcMode, **TOFSourceNum** tofNum, float fzNear, float fzFar, const char *szCameraConfigFile)

A constructor that loads the camera calibration information and other settings.

CARTOFSource & **operator=** (const **CARTOFSource** &)

Copy constructor.

virtual void **open** ()

virtual void **close** ()

virtual void **pause** ()

virtual void **play** ()

virtual void **update** (osg::NodeVisitor *nv)

Update the video texture by grabbing an image from the video instance.

float **getDepthA** ()

Public Member Functions inherited from SmartVideo::CARVideoSource

Protected Attributes

TOFSourceNum **m_srcNum**

Distinguish whether both or only one, either depth or intensity, values are used.

TOFIOMode **m_ioMode**

ifstream **m_is**

Handle to the recording data.

int **m_nRow**

The number of rows contained in the TOF image.

int **m_nColumn**

The number of columns contained in the TOF image.

float **m_fzNear**

The near plane distance which must be set as the same value of the virtual camera.

float **m_fzFar**

The far plane distance which must be set as the same value of the virtual camera.

float **m_fDepthA**

An auxiliary attribute for computing depth buffer whose value is $m_fzFar/(m_fzFar - m_fzNear)$.

float * **m_pRGB2TOF**

float * **m_pImage2**

Store the pixels from the homography mapping.

float * **m_pAuxImage**

An auxiliary buffer used in **update()**.

size_t **m_nNumRecFrames**

The number of recorded frame.

▀ Protected Attributes inherited from **SmartVideo::CARVideoSource**

Static Protected Attributes

static char * **ms_pSourceData** = NULL

Store the source data from the recording TOF video stream.

▀ Static Protected Attributes inherited from **SmartVideo::CARVideoSource**

Additional Inherited Members

▀ Protected Member Functions inherited from **SmartVideo::CARVideoSource**

Detailed Description

CARTOFSource is a subclass of **CARVideoSource** that handles the video input from a PMD Time-of-flight (TOF) camera, and updates the video texture.

Member Enumeration Documentation

enum SmartVideo::CARTOFSource::TOFSourceNum

Specify explicitly that either single or double sources are needed. If single, then the instance itself is responsible for initializing the TOF camera. If double, let instance carrying depth value to initialize the TOF camera.

Constructor & Destructor Documentation

```
SmartVideo::CARTOFSource::CARTOFSource ( SourceMode   srcMode,  
                                          TOFSourceNum tofNum,  
                                          float        fzNear,  
                                          float        fzFar,  
                                          const char *  szCameraConfigFile  
                                          )
```

A constructor that loads the camera calibration information and other settings.

Parameters

srcMode Explains the service function of the camera.
tofNum Whether it is used for depth only, or both depth and intensity.
szCamereConfigFile The camera calibration filename.

Member Function Documentation

```
void SmartVideo::CARTOFSource::close ( )
```

virtual

Close the video stream. Terminate the connection with the video stream and clean the handle. It is called by the destructor by default.

Implements [SmartVideo::CARVideoSource](#).

void SmartVideo::CARTOFSource::open ()

virtual

Open the TOF video stream. Access the video stream (hardware or file) config and get a handle on it.

Implements **SmartVideo::CARVideoSource**.

virtual void SmartVideo::CARTOFSource::pause ()

inline virtual

Pause the video stream grabbing, and this function is superfluous interface since TOF camera is updated by passively calling.

Implements **SmartVideo::CARVideoSource**.

virtual void SmartVideo::CARTOFSource::play ()

inline virtual

Start the video stream grabbing, and this function is superfluous interface since TOF camera is updated by passively calling.

Implements **SmartVideo::CARVideoSource**.

Member Data Documentation

TOFIOMode SmartVideo::CARTOFSource::m_ioMode

protected

m_ioMode is LIVE when the data stream comes from the device. m_ioMode is REC when the data stream comes from the recording data.

float* SmartVideo::CARTOFSource::m_pRGB2TOF

protected

RGB 2 TOF pixel homography mapping, because the RGB image is shown as background, a corresponding depth map needs to be constructed instead of feeding TOF depth map into the depth buffer directly. Here a RGB2TOF map is precomputed to save the mapping computation.

SmartVideo::CARVideoCamera Class Reference

Public Member Functions

CARVideoCamera (float fzNear=-1.f, float fzFar=-1.f, bool occlusionEnabled=false)

A constructor that sets up the orthogonal camera's ortho projection matrix and the viewport. The ortho camera's viewport size can be reset by setSize.

CARVideoCamera (const **CARVideoCamera** &videoCamera, const osg::CopyOp ©op=osg::CopyOp::SHALLOW_COPY)

Copy constructor.

void **SetSize** (double width, double height)

Set up the orthogonal projection viewport size, the origin is at (0, 0).

void **SetSize** (double x, double y, double width, double height)

Set up the orthogonal projection viewport size and the origin point.

void **SetSize** (const osg::Image &image)

Set up the orthogonal projection viewport size based on a image size.

~CARVideoCamera (void)

A default destructor.

Protected Attributes

float **m_fzNear**

The near plane distance.

float **m_fzFar**

The far plane distance.

Detailed Description

CARVideoCamera sets up an ortho projection and show the quad geometry upon which RGB or depth texture(s) is(are) pasted. This RGB texture is shown as AR background. The depth texture is used for resolving occlusion effect.

Constructor & Destructor Documentation

```
SmartVideo::CARVideoCamera::CARVideoCamera ( float fzNear = - 1. f ,  
                                              float fzFar = - 1. f ,  
                                              bool  occlusionEnabled = false  
                                              )
```

A constructor that sets up the orthogonal camera's ortho projection matrix and the viewport. The ortho camera's viewport size can be reset by setSize.

See Also

[SetSize](#)

Parameters

fzNear Near plane distance of the ortho camera.

fzFar Far plane distance of the ortho camera. occlusionEnabled Set up the depth buffer comparison property if occlusion enabled.

SmartVideo::VideoConfiguration Struct Reference

The web camera setting information, like size, pixel format, device factory configuration. [More...](#)

Public Attributes

int **id**

int **width**

int **height**

FrameRateType **framerate**

PixelFormatType **type**

std::string **deviceconfig**

Detailed Description

The web camera setting information, like size, pixel format, device factory configuration.

SmartVideo::CARVideoGeode Class Reference

Public Types

-
- enum **TextureMode** { **USE_TEXTURE_2D**, **USE_TEXTURE_RECTANGLE** }
Whether the texture is pasted as normalized quad or a rectangle.
-
- enum **ImageMode** { **USE_VIDEO** = 0, **USE_DEPTH** = 1, **MISCEL** }
The image mode decides the OpenGL internal format, format and datatype.
-
- enum **TexSequence** { **NO_FLIP**, **FLIP_UP_BOTTOM**, **FLIP_LEFT_RIGHT**, **FLIP_DIAG** }
The flipping choice of the texture.

Public Member Functions

-
- CARVideoGeode** (std::string cameraConfigFile, **TextureMode** texturemode=USE_TEXTURE_2D, bool updateEnabled=true, osg::Image *videolmage=0L, osg::Image *dpethlimage=0L)
-
- CARVideoGeode** (const **CARVideoGeode** &, const osg::CopyOp ©op=osg::CopyOp::SHALLOW_COPY)
Copy constructor using CopyOp to manage deep vs shallow copy.
-
- osg::Camera * **GetPreRenderCamera** ()

Protected Member Functions

-
- void **ConfigTexture** (osg::Image *image, osg::Texture *&texture, osg::Geode *geode, unsigned int texId, const osg::Vec2f &tSize, bool texSubLoad=false, bool imageUpdate=false, **ImageMode** imagemode=MISCEL)
-
- osg::Texture * **GenTexture** (osg::Geometry *geom, osg::Geode *geode, osg::Image *image, unsigned int texId, unsigned int imgWidth, unsigned int imgHeight, **TexSequence** texSeq, bool texSubLoad=false, bool imageUpdate=false, **ImageMode** imageMode=MISCEL)
-
- void **RenderToDepthTex** (osg::Texture *&texture, osg::Image *depthImage)
-
- void **RenderToVideoDepthTex** (osg::Texture *&videoTex, osg::Texture *&depthTex, osg::Image *videolmage, osg::Image *depthlimage, unsigned int width, unsigned int height)
-
- void **BuildShader** ()
-
- void **BuildPrerenderHomoShader** (osg::Geode *geode, osg::Vec2 depthTexCoord)
-
- void **BuildPrerender3DShader** (osg::Geode *geode, osg::Vec2 depthTexCoord, osg::Vec2 imageSize, float a)
-
- void **LoadShaderSource** (osg::Shader *shader, const std::string &fileName)
Load the shader from physical file.
-
- osg::Texture * **GetIntensityTex** ()
Access the RGB texture used in the main camera.
-
- osg::Texture * **GetDepthTex** ()
Access the depth texture used in the main camera.
-
- void **LoadHomographyMapping** (unsigned int width, unsigned int height)

void	LoadStereoCameraSettings ()	Load the intrinsic and extrinsic (relative rotation and translation) calibration of RGB and TOF camera.
void	PrepareTexturesWithoutDepth (osg::Geometry *geom, osg::Geode *geode, osg::Image *videoImage)	Generate the RGB texture when the depth texture is disabled.
void	PrepareTexturesWithWellDepth (osg::Geometry *geom, osg::Geode *geode, osg::Image *videoImage, osg::Image *depthImage)	Generate the intensity and depth texture with both sources from TOF camera.
void	PrepareTexturesWithDepthInHomoShader (osg::Geometry *geom, osg::Geode *geode, osg::Image *videoImage, osg::Image *depthImage)	Generate the RGB texture and the depth texture processed in the GLSL with homography mapping.
void	PrepareTexturesWithDepthIn3DShader (osg::Geometry *geom, osg::Geode *geode, osg::Image *videoImage, osg::Image *depthImage)	Generate the RGB texture and the depth texture processed in the GLSL with stereo mapping.
osg::Camera *	SetupPrerenderCamera (unsigned int width, unsigned int height)	

Protected Attributes

	TextureMode	m_textureMode	It could be either TEXTURE_2D or TEXTURE_RECTANGLE.
osg::Camera *		m_prerenderCamera	
osg::Texture *		m_videoTex	RGB texture used in the main camera.
osg::Texture *		m_depthTex	The depth texture used in the main camera.
osg::Image *		m_mappingImageX	The homography mapping between the depth and video image in X dimension.
osg::Image *		m_mappingImageY	The homography mapping between the depth and video image in Y dimension.
int		m_nStereoImageWidth	In the stereo mapping, the width of the prerender image viewport.
int		m_nStereoImageHeight	In the stereo mapping, the height of the prerender image viewport.
int		m_nTOFImageWidth	The raw TOF image width.
int		m_nTOFImageHeight	

	int	m_nRGBImageWidth	The raw RGB image width.
	int	m_nRGBImageHeight	The raw RGB image height.
	int	m_nPrerenderVideoTexFlip	The texture flip choice of the RGB texture generated from the prerendering.
	int	m_nPrerenderDepthTexFlip	The texture flip choice of the depth texture generated from the prerendering.
	int	m_nRawVideoTexFlip	The texture flip choice of the RGB texture generated from the raw RGB image.
	int	m_nRawDepthTexFlip	The texture flip choice of the depth texture generated from the raw depth image.
osg::Camera::RenderTargetImplementation		m_renderImplementation	The frame_buffer_object option is used for render to texture.
	unsigned int	m_samples	A dummy value for render to texture.
	unsigned int	m_colorSamples	A dummy value for render to texture.
	osg::Program *	m_videoProgram	The shader program used in the main camera.
	osg::Shader *	m_videoVertObj	The vertex shader used in the main camera.
	osg::Shader *	m_videoFragObj	The fragment shader used in the main camera.
	osg::Program *	m_prerenderVideoProgram	The shader program used in the prerender camera.
	osg::Shader *	m_prerenderVideoVertObj	Vertex shader used in the prerender camera.
	osg::Shader *	m_prerenderVideoFragObj	The fragment shader used in the prerender camera.
	bool	m_bImageUpdateEnabled	
	cv::Mat	m_inTOFMat	The intrinsic matrix of the TOF camera used in the stereo mapping.
	cv::Mat	m_inRGBMat	The intrinsic matrix of the RGB camera used in the stereo mapping.
	cv::Mat	m_exTransMat	

	The translation component of the extrinsic matrix between the RGB and depth camera.
cv::Mat	m_exRotMat
	The rotation component of the extrinsic matrix between the RGB and depth camera.
std::string	m_szCameraConfigFile
	The filename of the intrinsic and stereo camera matrix.
bool	m_bPrerenderCameraCallback
	A debug parameter for the prerender camera, and prerender results will be shown if enabled.

Detailed Description

The basic function of **CARVideoGeode** is to generate the video texture from the raw RGB image and paste it on the geode as the AR background. To resolve the occlusion, it is also in charge of the depth texture. There are three types of registration methods. The first one is to use the raw intensity and depth image from the same TOF camera without any registration process. The second relates the RGB image to the depth image using homography mapping. The third relates the RGB image to the depth image using stereo mapping. The second and the third registration methods can benefit from the parallel computing by the GLSL .

Constructor & Destructor Documentation

```
SmartVideo::CARVideoGeode::CARVideoGeode ( std::string cameraConfigFile,
                                             TextureMode texturemode = USE_TEXTURE_2D,
                                             bool updateEnabled = true,
                                             osg::Image * videoImage = 0L,
                                             osg::Image * dpethImage = 0L
                                             )
```

Constructor that builds geode and geometry where texture(s) is(are) pasted, and prepares the texture(s) depending on the registration method like pure RGB, raw RGB and raw TOF, raw RGB and TOF with the homography mapping, RGB and TOF with the stereo mapping

Parameters

- cameraConfigFile** Stereo camera configuration for occlusion purpose.
- texturemode** Using texture 2D or texture rectangle.
- updateEnabled** Whether to retrieve a new image from the camera, this is disabled if it is passively updated as an eagle window.
- videoImage** Image handle to the RGB image stream.
- depthImage** Image handle to the TOF image stream. If it is NULL, then no occlusion effect.

Member Function Documentation

void

```
SmartVideo::CARVideoGeode::BuildPrerender3DShader ( osg::Geode * geode,  
                                                    osg::Vec2    depthTexCoord,  
                                                    osg::Vec2    imageSize,  
                                                    float        a  
                                                    )
```

protected

Build the shader program and add the raw (but normalized) depth texture and the raw RGB texture as the texture uniform into the fragment shader, which is used in the prerendered camera. The prerendered camera writes the processed depth and RGB image into both RGB and depth texture that is used in the main camera.

Parameters

- geode** The raw depth and RGB texture is attached to the geode.
- depthTexCoord** The texture coordinate of the raw depth texture.
- imageSize** Size of the video image.
- a** An auxiliary shader attribute whose value is $(m_fzFar/(m_fzFar-m_fzNear))$.

void

```
SmartVideo::CARVideoGeode::BuildPrerenderHomoShader ( osg::Geode * geode,  
                                                    osg::Vec2    depthTexCoord  
                                                    )
```

protected

Build the shader program and add the raw (but normalized) depth texture and the homography mapping texture as the texture uniform into the fragment shader, which is used in the prerendered camera. The prerendered camera writes the processed depth image into the depth texture that is used in the main camera.

Parameters

- geode** The raw depth texture and homography mapping texture is attached to the geode.
- depthTexCoord** The texture coordinate of the raw depth texture.

```
void SmartVideo::CARVideoGeode::BuildShader ( )
```

protected

Build the shader program and add the processed RGB texture and the depth texture as the texture uniform into the fragment shader. This shader is attached to the main camera.

```

void
SmartVideo::CARVideoGeode::ConfigTexture ( osg::Image *      image,
                                             osg::Texture *&    texture,
                                             osg::Geode *       geode,
                                             unsigned int      texId,
                                             const osg::Vec2f & tSize,
                                             bool             texSubLoad = f a l s e,
                                             bool             imageUpdate = f a l s e,
                                             ImageMode       imagemode = M SCEL
                                             )

```

protected

Associate a video or depth image to the texture which is attached to the geode, and add the callbacks to both geode and texture(s) in order to periodically update the texture

Parameters

- image** Pointer to the image which is the source of the texture.
- geode** The geode's updatecallback updates the texture, which essentially calls the image's update function.
- texId** Texture id for the texture on this geode.
- tSize** Texture size of the texture, usually it is the power of two.
- texSubLoad** No need for texture subload if the texture is static rather than the live video image. For example, it is disabled for the homo mapping texture.
- imageUpdate** Even if the texture is built from dynamic video stream, if image is for the eagle window, then its image is updated passively.
- imagemode** It decides OpenGL internalformat, format and datatype to build the texture.

```

osg::Texture*
SmartVideo::CARVideoGeode::GenTexture ( osg::Geometry * geom,
osg::Geode * geode,
osg::Image * image,
unsigned int texId,
unsigned int imgWidth,
unsigned int imgHeight,
TexSequence texSeq,
bool texSubLoad = false,
bool imageUpdate = false,
ImageMode imageMode = M_SCELE
)

```

protected

Compute the texture size, texture coordinate, generate a texture, attach image update callback to the texture and associate the texture with the geode.

Parameters

- geom** It stores the texture id and texture coordinates.
- image** Pointer to the image which is the image source of the texture.
- texId** Texture id for the texture on this geode.
- imgWidth** Image source width, the texture width is usually the power of two of the image source width.
- imgHeight** Image source height, the texture height is usually the power of two of the image source height.
- texSeq** Texture coordinate sequence which decides how the texture coordinate is assigned to the image source's corners. It can also be understand as whether the texture's source image needs to be rotated when pasted as a texture.
- texSubLoad** No need for texture subload if the texture is static rather than a live video image. For example, it is disabled for the homo mapping texture.
- imageUpdate** Even if the texture is built from dynamic video stream, if image is only for the eagle window, then its image is updated passively.
- imagemode** It decides OpenGL internalformat, format and datatype to build the texture.

```

osg::Camera* SmartVideo::CARVideoGeode::GetPreRenderCamera ( )

```

inline

Get the prerendered camera which writes the processed depth or RGB image to the texture(s) used in the main camera. It needs to be attached to the OSG scene tree to be able to function.

```
void SmartVideo::CARVideoGeode::LoadHomographyMapping ( unsigned int width,  
                                                         unsigned int height  
                                                         )
```

protected

Load the homography mapping look-up table between the depth and video image.

Parameters

width Width of the RGB image.

height Height of the RGB image.

```
void SmartVideo::CARVideoGeode::RenderToDepthTex ( osg::Texture *& texture,  
                                                    osg::Image *   depthImage  
                                                    )
```

protected

The prerender camera renders the homography mapping processed depth image to the depth texture used in the main camera.

Parameters

texture The homography mapping processed depth image is written to the texture.

depthImage The raw depth image to be processed in the prerender camera.

```
void  
SmartVideo::CARVideoGeode::RenderToVideoDepthTex ( osg::Texture *& videoTex,  
                                                    osg::Texture *& depthTex,  
                                                    osg::Image *   videoImage,  
                                                    osg::Image *   depthImage,  
                                                    unsigned int   width,  
                                                    unsigned int   height  
                                                    )
```

protected

The prerender camera renders the stereo mapping processed depth image and video image to the depth and video textures used in the main camera.

Parameters

videoTex The stereo mapping processed RGB image is written into the video texture.

depthTex The stereo mapping processed depth image is written into the depth texture.

videoImage The raw RGB image to be processed in the prerender camera.

depthImage The raw depth image to be processed in the prerender camera.

osg::Camera*

SmartVideo::CARVideoGeode::SetupPrerenderCamera

(**unsigned int width,**
unsigned int height
)

protected

Set up the prerender camera for writing the processed depth or RGB image to the texture(s) that is(are) used in the main camera.

Parameters

width Width of the viewport of the prerender camera.

height Height of the viewport of the prerender camera.

Member Data Documentation

bool SmartVideo::CARVideoGeode::m_bImageUpdateEnabled

protected

If this is the main camera, then update the video source. If this is only an eagle window-a passive camera updated by the main camera, then don't update it.

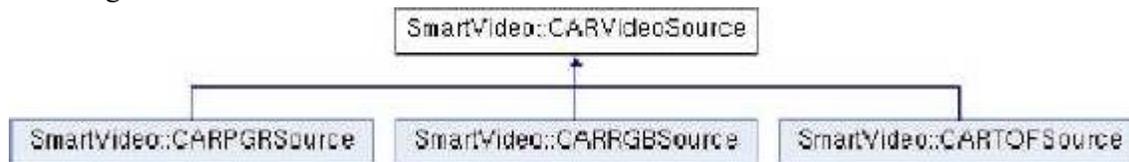
osg::Camera* SmartVideo::CARVideoGeode::m_prerenderCamera

protected

The prerender camera that writes the processed depth or RGB image to the texture(s) used in the main camera

SmartVideo::CARVideoSource Class Reference

Inheritance diagram for SmartVideo::CARVideoSource:



Public Types

```
enum SourceMode {  
    TOF_INTENSITY, TOF_DEPTH, TOF_GLSL_HOMO, TOF_GLSL_3D,  
    RGB_COLOR  
}
```

Public Member Functions

CARVideoSource (**SourceMode** srcMode, const char *szCameraConfigFile)

A constructor that reads in camera calibration information.

virtual **~CARVideoSource** (void)

A default destructor that saves the captured image, virtual makes sure it is always executed by decendants.

CARVideoSource & **operator=** (const **CARVideoSource** &)

Copy operator.

virtual void **open** ()=0

virtual void **close** ()=0

virtual void **pause** ()=0

Pause the video stream grabbing.

virtual void **play** ()=0

Resume the video stream grabbing.

virtual void **record** (unsigned char *image, size_t width, size_t height, size_t depth, size_t component)

Record a screen snapshot.

virtual void **save** ()

Save the screen snapshots.

virtual void **update** (osg::NodeVisitor *nv)=0

Update the video texture by grabbing an image from the video instance.

virtual **VideoConfiguration** * **GetVideoConfiguration** ()

void **Capture** ()

Set the capture mode bit, so that record can function.

SourceMode **GetSourceMode** ()

Protected Member Functions

bool **GetGLPixelFormatForARPixelFormat** (const int arPixelFormat, GLint *internalformat, GLenum *format_GL, GLenum *type_GL)

Translate the internal pixelformat to an OpenGL texture2D triplet.

Protected Attributes

SmartVideo::VideoConfiguration **m_config**

GLint **m_internalformat_GL**

Specifies the number of color components in the texture.

GLenum **m_format_GL**

Specifies the format of the pixel data.

GLenum **m_datatype_GL**

Specifies the data type of the pixel data.

CImgList< unsigned char > **m_listImg**

The image list to store the screenshot image.

std::vector< time_t > **m_vTimeStamp**

The timestamp of the screenshot image.

string **m_srnPath**

Where the screen shots are saved, temporarily it is saved where the dll is.

osg::ref_ptr< osg::Stats > **m_stats**

SourceMode **m_srcMode**

Identify the image function, registration method, and computing platform.

unsigned char * **m_pImage**

Contains the current captured image data.

bool **m_bCapture**

Whether to record current captured image.

const char * **m_szCameraConfigFile**

The video camera configuration file name.

cv::Mat **m_cameraMat**

The intrinsic camera matrix.

cv::Mat **m_distCoeffs**

The camera distortion model.

cv::Mat **m_mapX**

The first output map of the undistortion and rectification transformation map.

cv::Mat **m_mapY**

The second output map of the undistortion and rectification transformation map.

Static Protected Attributes

static OpenThreads::Mutex **_mutex**

Used for locking image in the update stage.

Member Enumeration Documentation

enum SmartVideo::CARVideoSource::SourceMode

Identify the function (depth/intensity) of the image, the registration method, and whether computed on CPU or GPU.

Constructor & Destructor Documentation

```
SmartVideo::CARVideoSource::CARVideoSource ( SourceMode srcMode,  
                                              const char * szCameraConfigFile  
                                              )
```

A constructor that reads in camera calibration information.

Parameters

srcMode Explains the function of the camera.
szCamereConfigFile The camera calibration filename.

Member Function Documentation

```
virtual void SmartVideo::CARVideoSource::close ( )
```

pure virtual

Close the video stream. Terminates the connection with the video stream and clean the handle.

Implemented in **SmartVideo::CARRGBSource**, **SmartVideo::CARPGRSource**, and **SmartVideo::CARTOFSource**.

OpenThreads::Mutex& SmartVideo::CARVideoSource::GetMutex () inline

Get the mutex for a video object to lock against read/write operations.

Returns

Reference to the internal mutex.

virtual void SmartVideo::CARVideoSource::open () pure virtual

Open the RGB/TOF video stream. Access the video stream (hardware or file) config and get an handle on it.

Implemented in **SmartVideo::CARRGBSource**, **SmartVideo::CARPGRSource**, and **SmartVideo::CARTOFSource**.

Member Data Documentation

SmartVideo::VideoConfiguration SmartVideo::CARVideoSource::m_config protected

It contains the string of the selected video configuration. See the <http://artoolkit.sourceforge.net/apidoc/video/> for more information on this parameter.

CResizeEventHandler Class Reference

Resize the camera's viewport when window size changes. [More...](#)

Public Member Functions

CResizeEventHandler (osg::Camera *camera)

osg::Vec2 **getWindowSize** ()

Access the width and height of the window.

virtual bool **handle** (const osgGA::GUIEventAdapter &ea, osgGA::GUIActionAdapter &aa)

Reset the camera's viewport size.

Protected Attributes

unsigned int **m_nWidth**

unsigned int **m_nHeight**

Private Attributes

osg::Camera * **m_videoCamera**

Detailed Description

Resize the camera's viewport when window size changes.

Member Data Documentation

unsigned int CResizeEventHandler::m_nWidth

protected

Store the current width and height of the window so as to resize the camera's viewport size. As a consequence, the RGB/Depth texture is resized as well.

PreCameraPostDrawCallback Class Reference

Private Member Functions

virtual void **operator()** (const osg::Camera &camera) const

virtual void **operator()** (const osg::Camera &camera) const

Detailed Description

PreCameraPostDrawCallback shows the prerender results for debugging purpose

See Also

m_bPrerenderCameraCallback

VideoCameraPostDrawCallback Class Reference

Private Member Functions

virtual void **operator()** (const osg::Camera &camera) const

Detailed Description

VideoCameraPostDrawCallback is called after the main camera is drawn. it reads from either depth or color buffer and show the image for debugging purpose

Appendix B

Occlusion Algorithm Pseudo Code

Appendix B.1 Optimized Equalization Algorithm

// Step 0: Initialize image array, auxiliary grey level array, cumulative histogram;

```
Raw_Image_Array[width*height];
```

```
Aux_Grey_Level_Array[width*height];
```

```
LEVEL=256; CumHistogram[LEVEL];
```

// Step 1: Looping the Raw_Image_Array[] to find the MaxVal, MinVal and Range = MaxVal-MinVal;

// Step 2: Build the cumulative histogram and Aux_Grey_Level_Array[];

```
For i = 0: (width*height-1)
```

```
    Aux_Grey_Level_Array[i] = Raw_Image_Array[i] / Range * (LEVEL-1);
```

```
    ++CumHistogram[Aux_Grey_Level_Array[i]];
```

```
End For;
```

```
For i = 0 : (Level-1)
```

```
    Accumulation += CumHistogram[i];
```

```
    CumHistogram[i] = Accumulation / (width*height) * (LEVEL-1);
```

```
End For;
```

// Step 3: Linearization of the cumulative histogram can be fulfilled by assigning cumulative histogram value at a certain grey level to all pixels at the same grey level;

```
For i = 0 : (width*height-1)
```

```
    Raw_Image_Array[i] = CumHistogram[Aux_Grey_Level_Array[i]];
```

```
End For;
```

Appendix B.2 Depth Buffer Population GLSL Fragment Shader

// Function Texture2D receives a sampler2D that is DepthTex or IntensityTex here, and fragment texture coordinates. It returns the texel value for the fragment which is the final depth of the fragment with the range of [0,1].

```
gl_FragDepth = texture2D(DepthTex, gl_TexCoord[1].xy).r;
```

// Since a fragment shader replaces ALL Per-Fragment operations of the fixed function OpenGL pipeline, the fragment color has to be calculated here as well.

```
gl_FragColor = texture2D(IntensityTex, gl_TexCoord[0].xy);
```

Appendix B.3 Homography Registration Using Render to Texture Fragment Shader

// Sample the TOF depth coordinate based on the homography lookup tables which are stored as MapX and MapY textures for the X and Y axis respectively.

// The sampled TOF depth coordinate needs to be left-right reversed, because the TOF camera stores image on the buffer from left to right, while the RGB camera does it from right to left.

// The sampled coordinate needs to be scaled by the ratio of image size over texture size (DepthTexRatio).

```
float depthX = (1.0 - texture2D(MapX, vec2(gl_TexCoord[1].xy)).s) *  
DepthTexRatio.x;
```

```
float depthY = (texture2D(MapY, vec2(gl_TexCoord[2].xy)).s) *  
DepthTexRatio.y;
```

// Sample the TOF depth value based on the interpolated coordinate. Here we show the nearest sampling approach instead of bilinear sampling approach to save text space here.

```
gl_FragDepth= texture2D(RawDepthTex, vec2(depthX, depthY)).s;
```

Appendix B.4 Stereo Registration Using Render to Texture Fragment Shader

// Sample the reciprocal of the physical depth map for perspective-correct Interpolation.

```
float recDist = texture2D(RawDepthTex, gl_TexCoord[1].xy).s;
```

// Compute depth value in depth buffer, DepthA is an input uniform.

```
gl_FragDepth = DepthA * (1.0 - recDist);
```

// Back project a TOF point from the 2D TOF screen space to the 3D TOF eye space using the TOF camera's intrinsic matrix. Magnify is an input uniform that translates the calibrated principle point according to the sampling ratio.

```
float phyDist = 1.0 / recDist; vec3 pTOF3d;
```

```
pTOF3d.x = (gl_FragCoord.x - InTOFMat[0][2]*Magnify) * phyDist /  
(InTOFMat[0][0]*Magnify);
```

```
pTOF3d.y = (gl_FragCoord.y - InTOFMat[1][2]*Magnify) * phyDist /  
(InTOFMat[1][1]*Magnify);
```

```
pTOF3d.z = phyDist;
```

// Transform the point from the 3D TOF eye space to the 3D RGB eye space using the extrinsic matrix.

```
vec3 pRGB3d;
```

```
pRGB3d = ExRotMat * pTOF3d - ExTransVec;
```

// Transform the point from the 3D RGB eye space to the 2D RGB screen space using the RGB camera's intrinsic matrix.

```
vec2 pRGB2d;
```

```
pRGB2d.x = (pRGB3d.x * InRGBMat[0][0] / pRGB3d.z) + InRGBMat[0][2];
```

```
pRGB2d.y = (pRGB3d.y * InRGBMat[1][1] / pRGB3d.z) + InRGBMat[1][2];
```

// Sample the RGB value based on the interpolated RGB coordinate.

```
vec2 pRGBTex2d;
```

```
pRGBTex2d.x = pRGB2d.x / RGBImageSize.x * DepthTexCoord.x;
```

```
pRGBTex2d.y = pRGB2d.y / RGBImageSize.y * DepthTexCoord.y;
```

```
gl_FragColor = texture2D(RawVideoTex, pRGBTex2d);
```

Appendix C

Aurora Code Documentation

Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

C CBuildingDamage	Store the building spatial information and edge, corner detection results
C CBuildingModel	Construct the building (damaged and undamaged) model
C CBuildingNodeCallback	Switch between the damaged/undamaged model
C CEagleViewHandler	The view handler of the eagle view
C CFirstPersonViewHandler	View handler of the first person view
C CMagnifyShot	Take snapshots of a specified area of Photo View
C Corner	Corner contains the position information about each building corner at the key location
C CPhotoViewHandler	The view handler of the photo view
C CSnapshot	Take snapshot of the first person view
C CThirdPersonViewHandler	The view handler of the third person view
C CViewOrganizer	The organizer of multiple views
C ErrorInfo	Store the drift config that satisfy the XY error constraint
C TexCoord	Texture coordinate associated with each vertex of the building
C Tripod	The region and coordinate where tripod is set up to observe the building

CBuildingDamage Class Reference

Store the building spatial information and edge, corner detection results. [More...](#)

Public Member Functions

	CBuildingDamage (bool useModel, float shakeDev=0.08)
	A constructor.
double *	getProjMat (size_t cornerPos, size_t story)
	Get the camera projection matrix when observing story at tripodPos (0 - 7).
osg::Vec2d &	getDetCorner2D (size_t tripodPos, size_t story)
	Get the 2D corner estimated result reference for story observed at tripodPos (0 - 7).
osg::Vec3d &	getDetCorner3D (size_t cornerPos, size_t story)
	Get the 3D corner estimated result reference for story at cornerPos (0 - 3).
osg::Vec4d &	getDetEdge (size_t tripodPos, size_t story)
	Get the detected edge end points reference centered at story observed at tripodPos (0 - 7).
short &	getAccuracyIndicator (size_t cornerPos, size_t story)
	Get the accuracy indicator, whether the estimation results satisfy the selection requirements, for story at cornerPos (0 - 3).
osg::Vec3d &	getDetShift3D (size_t cornerPos, size_t story)
	Get the 3D corner shift to its undamaged position for story at cornerPos (0 - 3).
osg::Vec3d	getTripodPos (int pos)
	Get the 3D position of tripod at pos (0 - 7).
osg::Group *	getOrCreateBuildingModel ()
	Create the building model and store the model.
const osg::Vec3d	getBuildingVerticesBeforeDamage (size_t cornerPos, size_t story)
	Get the building model vertex before damage at cornerPos (0 - 3) and story.
const osg::Vec3d	getBuildingVerticesAfterDamage (size_t cornerPos, size_t story)
	Get the building model vertex after damage at cornerPos (0 - 3) and story.
osg::PositionAttitudeTransform *	getTripodNode ()
	Get the transform node under which tripod geode is attached.
osg::PositionAttitudeTransform *	getCameraNode ()
	Get the transform node under which camera geode is attached.

const osg::Switch * **getBuildingModelOnly ()**

Get the switch node under which both damaged and undamaged building node is attached.

Protected Attributes

bool **m_bUseExistingModel**

Whether to use the existing model instead of creating a new one.

float **m_shakeDev**

The uniform support range of the model drifted vertices.

CBuildingModel * **m_buildingModel**

The building model of both before and after the damage.

vector< vector< double * > > **m_projMatVec**

Store the camera projection matrix.

vector< vector< osg::Vec2d > > **m_detectedCorner2D**

Store the estimated 2D corner after the damage.

vector< vector< osg::Vec3d > > **m_detectedCorner3D**

Store the estimated 3D corner after the damage.

vector< vector< osg::Vec4d > > **m_detectedEdge**

Store the detected 2D edge endpoint.

vector< vector< short > > **m_accuracyIndicator**

Indicate whether the result is from the 1st or the 2nd criteria.

vector< vector< osg::Vec3d > > **m_detectedShift3D**

Record the shift of each corner, this is designed for the unified X,Y shift in one floor.

vector< osg::Vec3 > **m_tripodPos**

The tripod position.

Detailed Description

Store the building spatial information and edge, corner detection results.

Provide interface for accessing the spatial building information and the edge, corner detection results.

Constructor & Destructor Documentation

```
CBuildingDamage::CBuildingDamage ( bool useModel,  
                                     float shakeDev = 0.08  
                                     )
```

A constructor.

Parameters

useModel Use the existing damaged building model or create a new damaged one with uniform distribution.

shakeDev Interval of the uniform distribution.

Member Function Documentation

```
short& CBuildingDamage::getAccuracyIndicator ( size_t cornerPos,  
                                              size_t story  
                                              )
```

inline

Get the accuracy indicator, whether the estimation results satisfy the selection requirements, for story at cornerPos (0 - 3).

See Also

selectEstimation in [driftEstimate.h](#)

Member Data Documentation

```
vector<vector<short >> CBuildingDamage::m_accuracyIndicator
```

protected

Indicate whether the result is from the 1st or the 2nd criteria.

True indicates results generated by the 1st criteria, false if from the 2nd criteria. Basically useless when counting for instrument error.

```
vector<vector<osg::Vec3d>> CBuildingDamage::m_detectedShift3D
```

protected

Record the shift of each corner, this is designed for the unified X,Y shift in one floor.

Store the average value if both sides measurement quality is true (m_accuracyIndicator is 1).

CBuildingModel Class Reference

Construct the building (damaged and undamaged) model. [More...](#)

Public Member Functions

	CBuildingModel (float shakeDev)
osg::Group *	getOrCreateBuildingModel (bool bUseModel=true) Get the building model node. if it does not exist, then initialize the model node.
const osg::Vec3d	getBuildingVerticesBeforeDamage (size_t cornerPos, size_t story) Get the building model vertex before damage at cornerPos (0 - 3) and story.
const osg::Vec3d	getBuildingVerticesAfterDamage (size_t cornerPos, size_t story) Get the building model vertex after damage at cornerPos (0 - 3) and story.
osg::PositionAttitudeTransform *	getTripodNode () Get the transform node under which the tripod geode is attached.
osg::PositionAttitudeTransform *	getCameraNode () Get the transform node under which the camera geode is attached.
const osg::Switch *	getBuildingModelOnly () Get the node under which both damaged and undamaged building node are attached.

Protected Member Functions

osg::Group *	createBuildingModel (bool bUseExistingModel=true) Create the building model node with the existing damaged model or create a new one with different uniform distribution value.
osg::Group *	createStory (int storyIndex, bool damage=false) Create one story at storyIndex.
osg::Geode *	createOrSetTerrain () Create the terrain geode if not exist, and return the geode.
osg::Geode *	createOrSetSoundRoof () Create the undamaged roof geode if not exist, and return the geode.
osg::Geode *	createOrSetDamagedRoof () Create the damaged roof geode if not exist, and return the geode.
osg::Geode *	createWall (wallFace face, int storyIndex, bool damage=false) Create a wall at the direction of the face at storyIndex.
void	

	generateLayerVertexProbablistically (osg::Vec3d &sw, osg::Vec3d &se, osg::Vec3d &ne, osg::Vec3d &nw, int storyIndex)
	Set the probabilistic (uniform) drift for each vertex on each layer.
void	readTexCoords ()
	Load the texture coordinates for walls.
void	configTexture ()
	Load the texture images for walls.
void	configBuildingVertex ()
	Create the building model before and after damage programmable.
osg::PositionAttitudeTransform *	createOrSetTripod (osg::Vec3d)
	Load the tripod model.
osg::PositionAttitudeTransform *	createOrSetCamera (osg::Vec3d)
	Load the camera model.

Protected Attributes

osg::ref_ptr< osg::Group >	m_buildingAndCameraModel	The model node and its satellites-like camera node and tripod node.
osg::ref_ptr< osg::Switch >	m_buildingModel	The building model node and the terrain node.
osg::ref_ptr< osg::Group >	m_buildingModelBeforeDamage	The building model node before damage.
osg::ref_ptr< osg::Group >	m_buildingModelAfterDamage	The building model node after damage.
vector< TexCoord >	m_NSTexCoords	The texture coords of the North/South faces.
vector< TexCoord >	m_WETexCoords	The texture coords of the East/West faces.
osg::ref_ptr< osg::Texture >	m_nsTexture	The texture of the South/North face.
osg::ref_ptr< osg::Texture >	m_ewTexture	The texture of the East/WEST face.
vector< vector< osg::Vec3d > >	m_buildingVerticesBeforeDamage	Store the vertex coordinate of the building.
vector< vector< osg::Vec3d > >	m_buildingVerticesAfterDamage	Store the vertex coordinate of the building.
float	m_shakeDev	The uniform distribution interval of the model drifted vertices.
boost::lagged_fibonacci607	rng	The random variables generator's seed.
	m_tripod	

osg::ref_ptr
< osg::PositionAttitudeTransform >

The transform node above the tripod node.

osg::ref_ptr
< osg::PositionAttitudeTransform > **m_camera**

The transform node above the camera node.

osg::ref_ptr< osg::Geode > **m_terrain**

The terrain geometry node.

osg::ref_ptr< osg::Geode > **m_soundRoof**

The undamaged roof node associated with the undamaged building model.

osg::ref_ptr< osg::Geode > **m_damagedRoof**

The damaged roof node associated with the damaged building model.

Detailed Description

Construct the building (damaged and undamaged) model.

Construct the (damaged and undamaged) building model, as well as the roof, terrain, camera, tripod etc.

Member Function Documentation

osg::Group * CBuildingModel::getOrCreateBuildingModel (bool bUseModel = true)

Get the building model node. if it does not exist, then initialize the model node.

Parameters

bUseModel Use the existing damaged model (distribution model) or create a new one (with new distribution model).

Member Data Documentation

vector<vector<osg::Vec3d> > CBuildingModel::m_buildingVerticesAfterDamage

protected

Store the vertex coordinate of the building.

11 layers, and 4 points each layer after damage.

vector<vector<osg::Vec3d> > CBuildingModel::m_buildingVerticesBeforeDamage

protected

Store the vertex coordinate of the building.

11 layers, and 4 points each layer before damage.

CBuildingNodeCallback Class Reference

Switch between the damaged/undamaged models. [More...](#)

Public Member Functions

CBuildingNodeCallback ()

A default constructor.

virtual void **operator()** (osg::Node *node, osg::NodeVisitor *nv)

An overloaded operator to turn on/off between the two models.

Detailed Description

Switch between the damaged/undamaged models.

CBuildingNodeCallback turns on/off the damaged and undamaged building model.

CEagleViewHandler Class Reference

The view handler of the eagle view. [More...](#)

Public Member Functions

bool **handle** (const osgGA::GUIEventAdapter &ea, osgGA::GUIActionAdapter &aa)

The overloaded handler to change the size and viewport of the eagle view.

Protected Attributes

int **m_preViewportWidth**

The previous viewport width of the eagle view.

int **m_preViewportHeight**

The previous viewport height of the eagle view.

Detailed Description

The view handler of the eagle view.

Redraw the specified area of the photo view in a magnified way, when the viewport of the eagle view is changed passively.

CFirstPersonViewHandler Class Reference

View handler of the first person view. [More...](#)

Public Member Functions

CFirstPersonViewHandler (**CBuildingDamage** &buildingDamage, **CSnapshot** &snapshot, bool &m_bProceed)

A constructor.

int **getCurrentPos** () const

int **getCurrentStory** () const

osg::Vec3d **getCurrentTripodPos** () const

bool **checkNewPhotoTaken** ()

Change the value to false if the value is true, otherwise no change.

bool **handle** (const osgGA::GUIEventAdapter &ea, osgGA::GUIActionAdapter &aa)

The overloaded handler to control the orientation, location of the camera.

Detailed Description

View handler of the first person view.

Control the position and orientation of the camera.

Constructor & Destructor Documentation

```
CFirstPersonViewHandler::CFirstPersonViewHandler ( CBuildingDamage & buildingDamage,
                                                    CSnapshot & snapshot,
                                                    bool & m_bProceed
                                                    )
```

A constructor.

Parameters

buildingDamage Reference to **CBuildingDamage** where the building spatial information is stored.

snapshot Reference to the snapshot of the first person view.

m_bProceed Reference to pause/resume switch of the auto detection.

CMagnifyShot Class Reference

Take snapshots of a specified area of Photo View. [More...](#)

Public Member Functions

CMagnifyShot (osg::Image &image)

A constructor.

void **setX** (int x)

Set the x coordinate of the original point of copying area.

void **setY** (int y)

Set the y coordinate of the original point of copying area.

void **setWidth** (int width)

Set the width of the copy area.

void **setHeight** (int height)

Set the height of the copy area.

void **needNewMagnifiedImage** ()

Protected Member Functions

virtual void **operator()** (osg::RenderInfo &renderInfo) const

Protected Attributes

osg::Image & **m_snapImage**

Reference to the snapshot image.

int **_x**

int **_y**

int **_width**

int **_height**

bool **m_bMagnifiedImageNeeded**

Whether the user wants to take a screenshot.

Detailed Description

Take snapshots of a specified area of Photo View.

CMagnifyShot reads the pixel buffer of the Photo View with specified size into an snapshot image and updates the texture with which it is associated.

Constructor & Destructor Documentation

CMagnifyShot::CMagnifyShot (osg::Image & image) inline

A constructor.

Parameters

image The snapshot to which the reading pixel buffer is copied.

Member Function Documentation

virtual void CMagnifyShot::operator() (osg::RenderInfo & renderInfo) const inline protected virtual

The overloaded operator to copy specified area of the pixel buffer into the snapshot.

Corner Struct Reference

Corner contains the position information about each building corner at the key location. [More...](#)

Public Member Functions

Corner (size_t _quadrant, size_t _story, osg::Vec3d _coord)

A constructor.

Corner ()

A default constructor.

Public Attributes

size_t **quadrant**

Can be 0, 1, 2, 3.

size_t **story**

Story level.

osg::Vec3d **coord**

3d coordinate.

Detailed Description

Corner contains the position information about each building corner at the key location.

CPhotoViewHandler Class Reference

The view handler of the photo view. [More...](#)

Public Member Functions

CPhotoViewHandler (**CBuildingDamage** &buildingDamage, **CFirstPersonViewHandler** &firstperson, osg::Image &image, **CMagnifyShot** &magnifyShot, float fShakeInterval)

A constructor.

bool **handle** (const osgGA::GUIEventAdapter &ea, osgGA::GUIActionAdapter &aa)

The overloaded handler for detecting vertical edge, searching horizontal baseline and estimating the corner 3D coordinate with human interference.

Protected Member Functions

void **clearDetectedLines** (osg::Group &root)

Clear all the detected lines by LSD, bounding box, and cross, when new snapshot of the first person view is taken.

osg::Vec4d **getVertexOfOriginalBaseLine** (osg::Group &root, const char *name, bool order=true)

Return the 2D vertices of the horizontal or vertical baselines.

void **drawCrossAtPoint** (osg::Group &root, osg::Vec2d center)

Draw the cross at the place where vertical edge and horizontal baseline intersects.

void **drawHorizontalBaseline** (osg::Group &root, int height, double shift)

Draw the horizontal baseline on the photo view based on the estimated shift.

void **drawVerticalBaseline** (osg::Group &root, int height)

Draw the original vertical baseline from the upper level to the lower level relative to the current story.

void **resizeMagnifyWindow** (int width, int height)

resize/reallocate the magnify window based on the location of the left button push or move

void **iterativeEstimate** (int corner, int story, int view_height, bool bAccurate=false)

Estimate the corner coordinate by iteratively testing all possible perpendicular shift of the horizontal baseline to its original position.

Protected Attributes

CFirstPersonViewHandler & **m_firstperson**

CBuildingDamage & **m_buildingDamage**

osg::Image & **m_snapImage**

Reference to the snapshot image of the first person view.

CMagnifyShot & **m_magnifyShot**

Reference to the magnifyShot responsible for taking the snapshot.

int **m_preViewportWidth**

int **m_preViewportHeight**

int **m_curTripodPos**

The current tripod position (can be from 0 ~ 7).

int **m_curStory**

The current story being observed at the center of the photo view.

double **m_horShift**

The estimation value of the gap between the horizontal baseline and the actual edge.

float **m_fLastX**

The x coordinate of the mouse click location.

float **m_fLastY**

The y coordinate of the mouse click location.

float **m_nExtraMagnifyScale**

The magnitude for enlarging the specified area in the eagle view.

bool **m_bNeedFiltering**

Whether necessary filtering work (bounding box filtering or right button selection) is done before drawing cross.

float **m_fShakeInterval**

The uniform distribution value.

Detailed Description

The view handler of the photo view.

The view handler of the photo view where edge detection and corner estimation take place.

Constructor & Destructor Documentation

```

CPhotoViewHandler::CPhotoViewHandler ( CBuildingDamage & buildingDamage,
                                          CFirstPersonViewHandler & firstperson,
                                          osg::Image & image,
                                          CMagnifyShot & magnifyShot,
                                          float fShakeInterval
                                          )

```

A constructor.

Parameters

- buildingDamage** Acquire and store the building information and detection results.
- firstperson** Notify the change of the observing target (story and corner).
- image** The snapshot of the first person view.
- magnifyShot** Take photos of the specified area of photo view.

Member Function Documentation

```

void CPhotoViewHandler::clearDetectedLines ( osg::Group & root )

```

protected

Clear all the detected lines by LSD, bounding box, and cross, when new snapshot of the first person view is taken.

Parameters

- root** The root node to which all the deleted elements were attached.

```

void CPhotoViewHandler::drawHorizontalBaseline ( osg::Group & root,
                                               int          height,
                                               double       shift
                                               )

```

protected

Draw the horizontal baseline on the photo view based on the estimated shift.

Because of the perpendicular shift to the horizontal baseline, there is a gap between the original baseline and the actual horizontal edge. That is why we need to add the estimated shift to compensate the gap.

Parameters

root The root node to which the horizontal baseline is attached.

height The height of the viewport of the photo view.

shift The estimated perpendicular shift to the original horizontal baseline.

```

void CPhotoViewHandler::drawVerticalBaseline ( osg::Group & root,
                                               int          height
                                               )

```

protected

Draw the original vertical baseline from the upper level to the lower level relative to the current story.

Parameters

root The root node to which the vertical baseline is attached.

height The height of the viewport of the photo view.

```

osg::Vec4d
CPhotoViewHandler::getVertexOfOriginalBaseLine ( osg::Group & root,
                                                  const char * name,
                                                  bool          order = true
                                                  )

```

protected

Return the 2D vertices of the horizontal or vertical baselines.

Parameters

root The root node to which baseline is attached.

name The node name of the baseline.

order If true, return the line in the order of minX, minY, maxX, and maxY.

```
void CPhotoViewHandler::iterativeEstimate ( int    corner,  
                                           int    story,  
                                           int    view_height,  
                                           bool   bAccurate = false  
                                           )
```

protected

Estimate the corner coordinate by iteratively testing all possible perpendicular shift of the horizontal baseline to its original position.

Parameters

- corner** The position of the building corner from 0 to 3.
- story** The story of the building.
- view_height** The height of the viewport of the photo view.
- bAccurate** Use the accurate mode means shifting both ends of the baseline with different distance, the inaccurate mode shifting both ends with the same distance.

Member Data Documentation

```
int CPhotoViewHandler::m_preViewportHeight
```

protected

The previous viewport height of the photo view. WARNING: Don't change viewport size, if the program is to reconstruct all the corners as a whole at the end (that is pressing 5) or do the error Analysis later on in errorAna.exe.

```
int CPhotoViewHandler::m_preViewportWidth
```

protected

The previous viewport width of the photo view. WARNING: Don't change viewport size, if the program is to reconstruct all the corners as a whole at the end (that is pressing 5) or do error Analysis later on in errorAna.exe.

CSnapshot Class Reference

Take snapshot of the first person view. [More...](#)

Public Member Functions

CSnapshot (osg::Image &image)

void **needNewSnapShotImage** ()

Public Attributes

bool **m_snapImageNeeded**

Whether to take a screenshot of the first person view.

Protected Member Functions

virtual void **operator()** (osg::RenderInfo &renderInfo) const

The overloaded operator to read first person view and sharpen it.

Protected Attributes

osg::Image & **m_snapImage**

Reference of snapshot image of the first person view.

Detailed Description

Take snapshot of the first person view.

CSnapshot reads the pixel buffer of the first person view and sharpen the image for the edge detection later on.

CThirdPersonViewHandler Class Reference

The view handler of the third person view. [More...](#)

Detailed Description

The view handler of the third person view.

A overloaded TrackballManipulator.

CViewOrganizer Class Reference

The organizer of multiple views. [More...](#)

Public Member Functions

CViewOrganizer (bool useModel, float shakeDev=0.04, bool thirdPersonView=false, int playSpeed=7)

A constructor.

~CViewOrganizer (void)

A destructor.

void **run** ()

Construct all the views and viewhandlers.

Protected Member Functions

osg::Group * **createSnapshotQuad** (osg::Image *image, const osg::Viewport *viewport)

Create a quad with texture showing the snapshot of the first person view.

osg::Group * **createQuadView** (osg::Image *image, const osg::Viewport *viewport, const char *geodeName)

Create a quad with the specified texture.

Protected Attributes

osg::ref_ptr

< osgViewer::CompositeViewer > **viewer**

The composite viewer that contains all the views.

CBuildingDamage * **m_pBuildingDamage**

The pointer to the building information and the damage detection results are stored.

osg::ref_ptr< osg::Image > **m_snapImage**

The snapshot of the first person view for updating the texture pasted on photo view.

osg::ref_ptr< osg::Image > **m_magnifyImage**

The snapshot of a certain zone of the photo view for updating the texture pasted on the magnify view.

int **m_nStep**

The step numbers for automatic detection.

int **m_nPlaySpeed**

The play speed of the automatic detection mode.

bool **m_bProceed**

Pause or resume the automatic detection.

bool **m_bShowThirdPersonView**

Whether to show the third person view or not.

float **m_fShakeInterval**

The uniform distribution value.

Detailed Description

The organizer of multiple views.

CViewOrganizer constructs all the views and attach functional viewhandlers to them.

Constructor & Destructor Documentation

```
CViewOrganizer::CViewOrganizer ( bool useModel,  
                                float shakeDev = 0.04,  
                                bool thirdPersonView = false,  
                                int playSpeed = 7  
                                )
```

A constructor.

Parameters

- | | |
|------------------------|--|
| useModel | Whether to use the existing damaged model or recreate a new one randomly. |
| shakeDev | The uniform distribution interval to create a new damaged model. |
| thirdPersonView | Whether to show the third person view where the user can maneuver the scene. |
| playSpeed | The play speed in the auto detection mode. |

Member Function Documentation

```
osg::Group * CViewOrganizer::createQuadView ( osg::Image *      image,  
                                              const osg::Viewport * viewport,  
                                              const char *      geodeName  
                                              )
```

protected

Create a quad with the specified texture.

Parameters

image The handler to the snapshot image.
viewport The viewport of the magnified view.
geodeName The geode's name for searching purpose.

```
osg::Group * CViewOrganizer::createSnapshotQuad ( osg::Image *      image,  
                                                  const osg::Viewport * viewport  
                                                  )
```

protected

Create a quad with texture showing the snapshot of the first person view.

Parameters

image The snapshot image of the first person view.
viewport The viewport of the photo view.

Member Data Documentation

```
bool CViewOrganizer::m_bProceed
```

protected

Pause or resume the automatic detection.

See Also

FirstPersonViewHandler::m_bProceed

```
int CViewOrganizer::m_nPlaySpeed
```

protected

The play speed of the automatic detection mode.

If it is 0, it means using the manual mode

ErrorInfo Struct Reference

Store the drift config that satisfy the XY error constraint. [More...](#)

Public Attributes

osg::Vec2d **shift**

osg::Vec2d **cornerShift**

osg::Vec3d **estimation**

osg::Vec3d **error**

Detailed Description

Store the drift config that satisfy the XY error constraint.

TexCoord Struct Reference

Texture coordinate associated with each vertex of the building. [More...](#)

Public Member Functions

TexCoord (float x, float y)

Public Attributes

float **_x**

float **_y**

Detailed Description

Texture coordinate associated with each vertex of the building.

Tripod Struct Reference

The region and coordinate where tripod is set up to observe the building. [More...](#)

Public Member Functions

Tripod (size_t _quadrant, osg::Vec3d _coord)

A constructor.

Tripod ()

A default constructor.

Public Attributes

size_t **quadrant**

Can be 0, 1,2,3,4,5,6,7.

osg::Vec3d **coord**

3d coordinate.

Detailed Description

The region and coordinate where tripod is set up to observe the building.

Appendix D

ARVita Quick Start Manual

D. 1 Preperation

D.1.1 Camera Selection

ARVita is capable of detecting all plug-in web cameras (not including 1394 camera) that are currently available on a certain computer. By default the first camera is selected on the right hand side with symbol . If ARVita senses multiple cameras present, the rest of them will be shown on the left hand side with symbol  as candidates. User can select them to the right hand side by clicking on . Cameras on the right hand side can also be deselected by clicking on . The selected camera will be used for capturing background video sequences and tracking. The number of initialized windows depends on how many cameras are selected. The program works best if no more than two cameras are selected. Additional cameras may sometimes fail in updating the image sequence.



Figure D.1 ARVita Logo.

D.1.2 Marker Selection

ARVita comes with 5 preinstalled markers, and UM marker is the default one. If users do not click on any one of them, UM marker will be used for tracking.

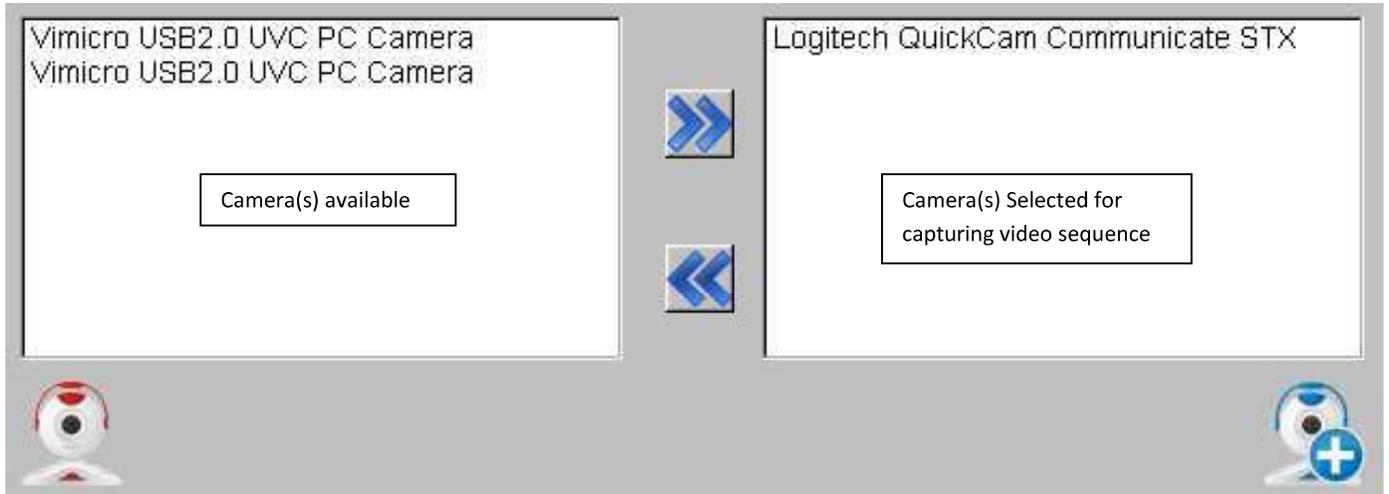


Figure D.2 Camera Selection.

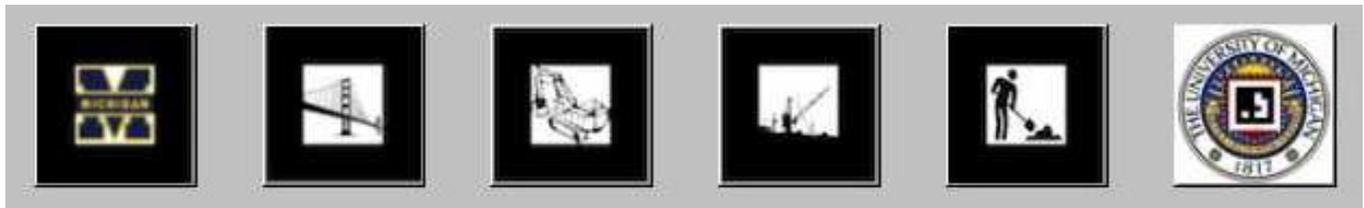


Figure D.3 Marker Selection.

To print a fiducial marker, users should first select a marker, then click on . An Adobe PDF file containing the marker will be produced.

To create and train customized fiducial markers, users can first create a marker at <http://www.roarmot.co.nz/ar/>, and then train it at <http://flash.tarotaro.org/blog/2009/07/12/mgo2/>. Save the marker pattern file to the ARVita application folder `\data\artoolkit2\diy.pat`. Select  in the marker list and use the customized fiducial marker for tracking.

If the user selects natural marker , the software will automatically shift to the KEG tracker library which can tolerate partial occlusion. User is encouraged to point the camera perpendicular to the marker at the initial stage or after the loss of tracking.

D.1.3 Help Document

If users need help for any part of the operation, they can refer to  for documentation, or  for an online video clip of a demonstration.

D. 2 Operation

The video should start automatically after clicking on . A default axis appears once the selected marker is detected by the camera(s).

D.2.2 Select Animation Script File

To observe a Vitascope animation, it is suggested to first download and install Vitascope.exe from the software page at <http://pathfinder.engin.umich.edu/>, then clicking on  and navigating to SAMPLE folder under the Vitascope application installation folder. It is important to note that ARVita only supports the basic set of the Vitascope authoring language, therefore those demos that require add-on DLLs are not compatible. A warning will be displayed if users select incompatible animation scripts. Some of the compatible Vitascope example animations include:

SAMPLE / Airport_Operations / ReaganNationalAirport.VTF

SAMPLE / Steel_Erection / SteelFrameErection.VTF

SAMPLE / One_Way_Curve / OneWayCurve.VTF

Users are encouraged to create their own animation script files conforming to Vitascope authoring language syntax but not requiring add-on DLLs. For example, Stroboscope is one of the tools for creating Vitascope animation file. It is available at: <http://www.ezstrobe.com/2009/10/stroboscope.html>. Once a successful selection is made, the script file can be viewed in wordpad by clicking on 

D.2.3 Playing Animation

Playing animation in ARVita is as easy as playing media player.  and  resumes and pauses animation.  and  slows down and speeds up the animation viewing ratio. The progress bar indicates the current animation time stamp, and users can also drag the cursor to jump to an arbitrary point of time (before the animation ending time).

D.2.4 Eagle Window

A construction jobsite model may easily block the visibility of a fiducial marker, which makes it difficult to cover the marker using camera. Eagle window is thus helpful in the process of identifying advantage point without compromising the coverage of the marker. Once the camera is set static, users can choose to toggle the eagle window off/on by clicking on /.

D.2.5 Navigating in the Model

To further ease the process of locating advantage point, ARVita provides the functionalities of zooming in/out and shifting of the model:

Scrolling the mouse wheel or pressing PageUp/PageDown button has the same effect in zooming in and out.

Dragging the model with left button pressed will shift the model. To make the model keep drifting, clicking middle button/wheel, and the model will move in the direction where the cursor is pointing to. The position of the model can also be controlled by WASD keys. W/S control forward and backward, while A/D control left and right. Additionally Up and Down button is used to lift the model vertically.



Figure D.4 The user interface of ARVita.

Appendix E

Biography

Suyang Dong was born in Nanjing, P.R.China on September 27, 1986. He earned a Bachelor of Science (B.E.) degree Geographical Information System at Wuhan University (WHU), Wuhan, P.R.China, in 2008 and a Master of Engineering (M.E.) degree in Civil and Environmental Engineering (majoring in Construction Engineering and Management) at the University of Michigan, Ann Arbor, MI, in 2010. He also had the second master degree as a Master of Science (M.S.) in Computer Science and Engineering at the University of Michigan, Ann Arbor, MI, in 2012. He pursued his education in Construction Engineering and Management at the University of Michigan, Ann Arbor, MI since 2008 and earned a Doctor of Philosophy (Ph.D.) degree in Civil and Environmental Engineering in 2012.